

DESARROLLO DE APLICACIONES WEB EN ENTORNO CLIENTE

GRADO SUPERIOR FP
DESARROLLO DE APLICACIONES WEB

2/10/2023

JOSE MANUEL MARTINEZ ROCA

Tabla de contenido

INTRODUCCIÓN.....	5
CONFIGURANDO EL ENTORNO	5
CONCEPTOS BÁSICOS.....	6
HERRAMIENTAS	6
¿DÓNDE INSERTAMOS EL CÓDIGO JAVASCRIPT?	6
VARIABLES EN JAVASCRIPT	9
ESTRUCTURA BÁSICA APLICACIÓN JAVASCRIPT	11
DOS EJEMPLOS SENCILLOS.....	13
LA WEB MÁS SENCILLA	13
WEB NORMALIZADA	15
FUNDAMENTOS JAVASCRIPT	19
HISTORIA DE JAVASCRIPT.....	19
CARACTERÍSTICAS DE JAVASCRIPT	20
POO	21
DOM	22
BASADO EN EVENTOS	22
ASINCRONISMO	23
VARIABLES.....	24
VAR, LET Y CONST	24
FUNCIONES	25
ESTRUCTURAS Y BLOQUES.....	28
CONDICIONAL IF.....	28
CONDICIONAL SWITCH.....	30
BUCLE WHILE	30
BUCLE FOR	30
BUCLE	31
TIPOS DE DATOS BÁSICOS.....	31
CASTING DE VARIABLES	31
NUMBER.....	31
STRING	31
BOOLEAN.....	31
MANEJO DE ERRORES	31
BUENAS PRACTICAS	32
use strict.....	32
Variables.....	32

Errores.....	32
OBJETOS	32
INTRODUCCION.....	32
PROPIEDADES.....	33
METODOS.....	34
WEB API.....	46
REACT	47

INTRODUCCIÓN

Vamos a estudiar los fundamentos del lenguaje interpretado JavaScript, pero, antes de ponernos de lleno con esta tarea, es necesario:

- Definir un entorno de partida en el ordenador que vayamos a usar para ello.
- Sentar algunos conceptos básicos sobre los que se asienta este lenguaje de desarrollo web.
- Aprender la estructura básica de cualquier aplicación escrita en JavaScript.
- Establecer un conjunto de convenciones y buenas prácticas a la hora de manejarnos con el lenguaje JavaScript y con el ecosistema que hemos creado para tal fin.

CONFIGURANDO EL ENTORNO

Para el seguimiento de este aprendizaje vamos a usar como IDE de programación [Visual Studio Code](#) y, además, usaremos unas cuantas extensiones que nos van a facilitar la labor de programar y de testear la web que vamos desarrollando.

Las extensiones que vamos a usar son:

- Live Server
- Material Icon Theme
- Prettier – Code Formatter
- SonarLint

La extensión [Live Server](#) nos provee de un servidor virtual que es capaz de mostrar páginas dinámicas que usen tecnología JavaScript.

La extensión [Material Icon Theme](#) es una mejora visual para VS Code que nos permite ver los iconos de cada uno de los archivos que componen nuestros proyectos ya que es capaz de reconocer que contiene cada uno de esos ficheros y así nos muestra el icono correspondiente para cada tipo de fichero incluido en nuestros proyectos.

La extensión [Prettier](#) es una extensión, igual que la anterior, para mejora visual del código que nos colorea las distintas estructuras, tal como, condicionales IF, bucles de tipo WHILE, FOR, etc. Solamente para que nos sea más fácil leer e interpretar ese código que estamos construyendo.

La extensión SonarLint se encarga de mostrar los fallos de programación que vamos cometiendo al escribir nuestros proyectos. Y no solamente de mostrar el fallo, si no, también de darnos el lugar de nuestro código donde se ha cometido el error y posibles explicaciones sobre el mismo.

Para acabar de configurar el ecosistema de programación que estamos montando debemos acudir a la opción “Auto guardado” o “Auto Save” que se encuentra dentro del menú “Archivo” o “File”, dependiendo del idioma que tengamos configurado en VS Code. Y marcar dicha opción para que el IDE VS Code se encargue de guardar automáticamente todo el proyecto cada vez que realicemos un cambio en dicho proyecto.

CONCEPTOS BÁSICOS

HERRAMIENTAS

Para aprender JavaScript lo más práctico es empezar a utilizar dicho lenguaje desde el primer momento, y para ello, podemos hacer uso de:

- Un IDE especializado, en nuestro caso, VS Code.
- Ejecutar el código desde la consola del navegador que usemos.
- Ejecutar el código desde un editor de código online, como puede ser: jsfiddle.net

Ya veremos por qué se usa cada una de estas opciones para ejecutar código, pero, se puede adelantar que según lo que queramos conseguir al ejecutar código será más conveniente usar una herramienta u otra para dicho fin. Por ejemplo, si queremos ver el valor de alguna variable es común poner una orden “console.log” en el código de nuestro proyecto para espiar y conocer dicho valor. Pero, igual, nos es más cómodo usar la consola del navegador, que ir a modificar un fichero y después volver a la ejecución de la aplicación web a ver el resultado.

Por otro lado, pensad en una aplicación enorme, unas 10.000 líneas de código aproximadamente o similar, sería una barbaridad intentar escribir esa aplicación web en una consola e incluso en un editor de código online, ya que este último no da facilidades para manejar más allá de unos cuatro o cinco ficheros y no está preparado para integrar tremendo tamaño de código, para este cometido lo más lógico sería usar el IDE VS Code que si nos da esas facilidades.

Como vemos, en cada uno de los casos presentados, debemos valorar y saber hacer uso de la herramienta más adecuada para cada uno de los fines que pretendemos alcanzar, la herramienta más eficiente, rápida y cómoda para nuestro mejor aprovechamiento tanto del tiempo como de la eficacia a la hora de escribir código.

¿DÓNDE INSERTAMOS EL CÓDIGO JAVASCRIPT?

Este código JavaScript se puede enlazar con la web HTML de diversas formas:

- Directamente entremezclando etiquetas HTML5 con código JavaScript mediante la etiqueta que rodeara siempre al código JavaScript “SCRIPT”.
- En uno o varios ficheros aparte que contendrán exclusivamente el código JavaScript y enlazándolo desde el archivo HTML con la instrucción:

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

Al igual que enlazamos el archivo con los códigos JavaScript desde el fichero HTML con la instrucción que acabamos de ver, para que todo resulte armonioso y quede completamente ajustado y bonito para el usuario. Debemos enlazar el fichero CSS, también desde el HTML, mediante la siguiente instrucción.

```
<link rel="stylesheet" href="style.css">
```

Atención que si se nos olvida enlazar estos ficheros y ejecutamos la aplicación web seguramente no funcione y podemos consumir un gran espacio de tiempo en descubrir que es esta falta de enlaces con sus respectivos ficheros la que nos hace que la web falle. No suelta ningún error el navegador ni la web si se ha dejado de enlazar los ficheros, solo que ves que no funciona, pero silenciosamente.

CONVENCIONES JAVASCRIPT

JavaScript es un lenguaje interpretado que es case sensitive, o, sensible a mayúsculas, esto es fácil de comprender, si entendemos que cualquier variación en el nombre de una variable crea una nueva variable, es decir:

var miVariable, este nombre de variable es distinto de var mivariable. Aunque este último nombre de variable ya veremos un poco más adelante que no es aceptable en JavaScript, aunque si este permitido. También son variables distintas miVariable, miVARIABLE, mIVARIABLE, etc.

El nombre de las variables siempre se escribe con la tipografía camelCase, es decir, cuando el nombre de una variable tiene más de una palabra, la primera letra de la variable siempre es minúscula y el resto de primeras letras de cada palabra que forma el nombre se colocan en mayúsculas. Sin incluir espacios, ni caracteres especiales o ilegales. Son ejemplos correctos de camelCase:

miNombre, miApellidoPrimero, miApellidoSegundo, miEdad.

No serian correctos los siguientes nombres de variables ni ajustados a la nomenclatura camelCase:

MiNombre, mi Nombre, mi_Nombre, MINOMBRE, minombre.

Lo podemos ver en el funcionamiento del siguiente código:

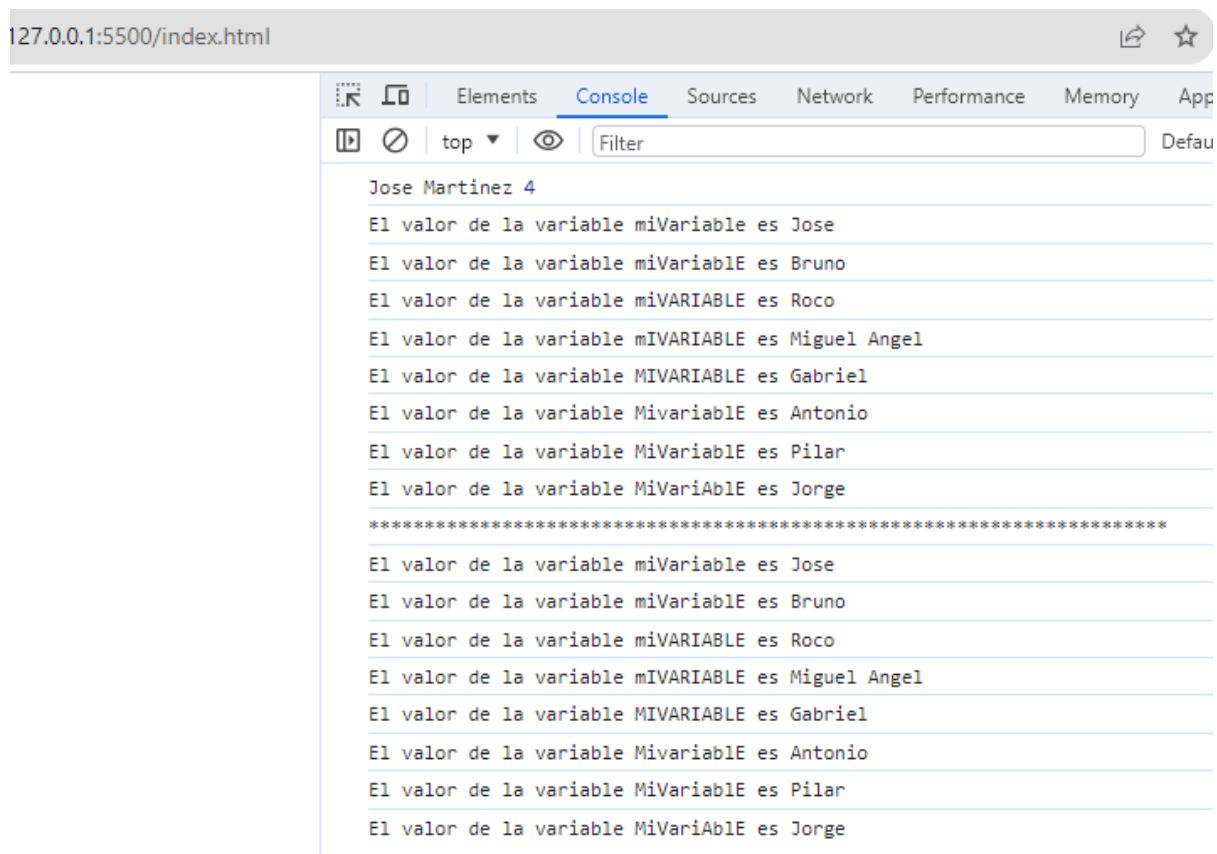
```
1. var miVariable = "Jose";
2. var miVariable = "Bruno";
3. var miVARIABLE = "Roco";
```

```

4. var mIVARIABLE = "Miguel Angel";
5. var MIVARIABLE = "Gabriel";
6. var MivariablE = "Antonio";
7. var MiVariablE = "Pilar";
8. var MiVariAblE = "Jorge";
9.
10.
11.     console.log('El valor de la variable miVariable es
' + miVariable);
12.     console.log('El valor de la variable miVariableE es
' + miVariableE);
13.     console.log('El valor de la variable mIVARIABLE es
' + mIVARIABLE);
14.     console.log('El valor de la variable MIVARIABLE es
' + MIVARIABLE);
15.     console.log('El valor de la variable MivariablE es
' + MivariablE);
16.     console.log('El valor de la variable MiVariablE es
' + MiVariablE);
17.     console.log('El valor de la variable MiVariAblE es
' + MiVariAblE);
18.
19.
20.     // Y si las vuelvo a imprimir contiene los mismos datos
21.     console.log("*****");
22.     console.log('El valor de la variable miVariable es
' + miVariable);
23.     console.log('El valor de la variable miVariableE es
' + miVariableE);
24.     console.log('El valor de la variable mIVARIABLE es
' + mIVARIABLE);
25.     console.log('El valor de la variable MIVARIABLE es
' + MIVARIABLE);
26.     console.log('El valor de la variable MivariablE es
' + MivariablE);
27.     console.log('El valor de la variable MiVariablE es
' + MiVariablE);
28.     console.log('El valor de la variable MiVariAblE es
' + MiVariAblE);
29.

```

Ya que la salida que provoca este código es la siguiente:



Como vemos, el más mínimo cambio en el nombre de la variable, aunque solamente sea en una mayúscula, genera una nueva variable totalmente independiente.

VARIABLES EN JAVASCRIPT

JavaScript tiene un tipado débil, no os asustéis por los tecnicismos, son fácilmente comprensibles y estamos aquí para aprenderlos y descubrir que significa cada cosa. Este tipado débil que en un principio puede parecer una ventaja, para mí, es un inconveniente. Ya que esa “libertad” que te ofrece JavaScript de declarar variables sin especificar el tipo de variable que es (literal, numérica, booleana, etc.) y después, cambiar al vuelo la variable, o sea, que declaro una variable, por ejemplo, tipo booleana que se llame apellidos y le asigno el valor de false porque en ese momento estoy pensando que no tiene apellidos conocidos, y después unas cuantas líneas más allá digo que esa variable ahora contiene el valor literal de los apellidos del usuario. Esto es, bajo mi humilde punto de vista, una bomba de relojería. Ya que además de la dificultad de la abstracción que debemos realizar para poder crear código, se añade la dificultad de que JavaScript nos vuelva locos cambiando las variables de tipo en cualquier momento.

De ahí, que se recomiende siempre, no usar variables globales con la orden “var” y se recomiende el uso de variables locales con la orden “let”, se pueden usar cualquiera de las dos, pero como buenas prácticas de programación se recomienda el uso de la orden “let” sobre la orden “var”, seguramente el programa funcionará bien en ambos casos, sobre todo si es una aplicación de pocas líneas de código. Pero es extremadamente aconsejable y hasta exigible que se use “let” sobre “var”

```

1. // Variables y constantes
2.
3. const nombre = "Jose";
4. const apellido = "Martinez";
5.
6. let valorDado = 5;
7. valorDado = 4
8.
9. // let valorDado = 4;
10.
11.     /* Esta última variable está mal declarada y salta un
12.        error porque no
13.        se pueden declarar dos variables con el mismo nombre dos
14.        veces. Si
15.        descomentamos la segunda línea donde se declara la
16.        variable por segunda vez
17.        veremos que el navegador envía un error por consola
18.        avisando de ello */
19.
20.     if (true) {
21.         let nombre2 = nombre + " Manuel";
22.         //Si quitas el siguiente comentario antes comenta la
23.         línea anterior
24.         //var nombre2 = nombre + " Manuel"; //Prueba a
25.         descomentar esta línea a ver que sucede
26.         // ¿Qué ha ocurrido?
27.     }
28.     //console.log(nombre2); //Quita este comentario para
29.     probar el caso anterior
30.
31.     //¿Qué ocurrirá si quito el comentario de la línea
32.     siguiente?
33.     //console.log(construirNombreCompleto("Luis","Alfonso"));
34.
35.     function construirNombreCompleto(nombre1, nombre2){
36.         return nombre1 + " " + nombre2;
37.     };
38.
39.     console.log(nombre, apellido, valorDado);
40.     // ¿Qué ocurrirá si quito los comentarios de las líneas
41.     siguientes?
42.     //console.log(nombre2);
43.     //console.log(nombre1);

```

Con el código anterior podemos ver varios ejemplos de ámbito de variables y la comentada recomendación de usar var lo mínimo posible y el uso de let en su lugar tomaran consistencia práctica.

Quitando comentarios y volviendo a colocarlos podemos hacer que la aplicación funcione o no, según las instrucciones que vemos en el propio código.

En este caso concreto, entre las líneas 1 a 14, están dedicadas a la repetición de variables con el mismo nombre, si quitamos el comentario de la línea nueve y dejamos que se vuelva a declarar una variable con el mismo nombre salta un error en la consola avisándonos de ello.

En el caso de las líneas 16 y 22, están dedicadas al ámbito de una variable según se declare la misma.

En las líneas 27 a 30 además de ver por primera vez como se declara una función en JavaScript y como se usan parámetros de entrada para la misma función, tocamos el asunto espinoso de los ámbitos de las variables dando una vuelta de tuerca más a la dificultad que ello conlleva. Ya que el resultado puede parecer sorprendente a quien no este familiarizado con ámbitos de variables en JavaScript.

Por último, las líneas 33 y 34 van relacionadas también con los ámbitos de las variables implicadas.

ESTRUCTURA BÁSICA APLICACIÓN JAVASCRIPT

Siempre que durante este curso vayamos a crear una aplicación básica de JavaScript se debe recurrir a la estandarización en dicho funcionamiento, para ello, crearemos los siguientes archivos:

- index.html
- style.css
- main.js

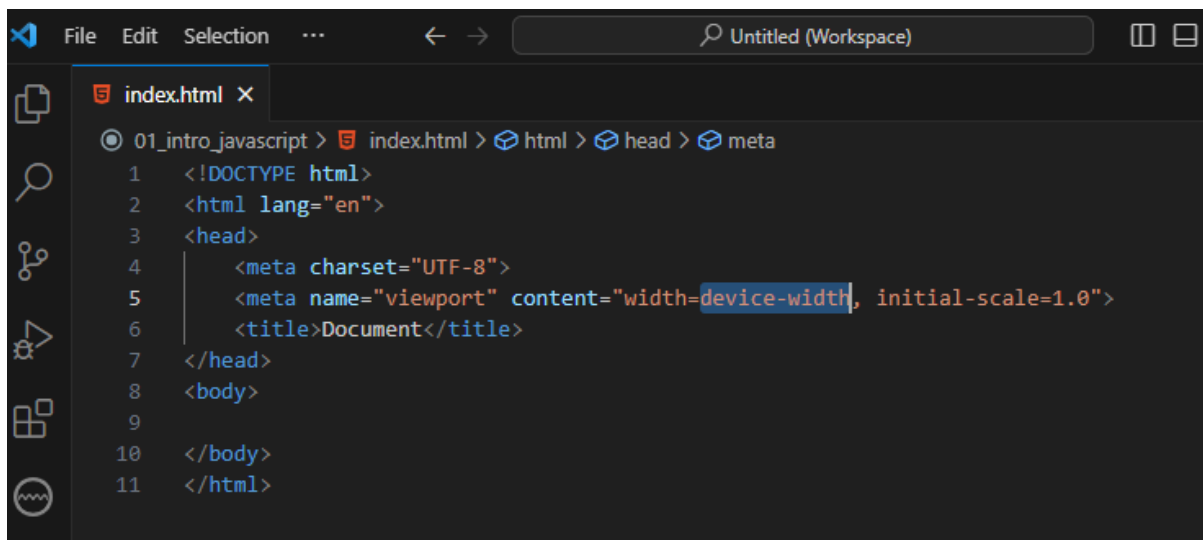
Ojo con las extensiones de los archivos implicados que deben ser exactamente esas y no otras, para que el IDE y el resto de aplicaciones que deban usar dichos ficheros, lo veremos cuando descubramos el framework REACT, puedan ser capaces de funcionar sin fallos, ni problemas adicionales.

Toda esta estructura se complicará posteriormente con el uso del Framework, pero hasta que llegue ese momento haremos la estructura más sencilla posible, sin subcarpetas donde guardar cada uno de los distintos tipos de ficheros que componen cada uno de los proyectos que realicemos. Esto quiere decir que por cada proyecto crearemos una carpeta con su nombre y dentro de esa carpeta se encontraran los tres archivos anteriores que hemos visto.

Fichero index.html

Este fichero contiene toda la estructura e información de la web que vamos a construir, en formato HTML5 y no debe, normalmente, incluir código JavaScript, ni estilos CSS que irán alojados cada uno de ellos en el resto de los ficheros destinados a tal fin.

Si al crear el documento en VS Code, usamos el símbolo “!”, admiración hacia abajo, como primer carácter a incluir en el documento y pulsamos la tecla intro veremos que VS Code nos rellena el documento con el texto básico y mínimo para que ese fichero HTML se distinga como un fichero HTML5, también podemos conseguir esto escribiendo como primeros caracteres en el fichero recién creado de HTML el código “HTML5”, tiene el mismo resultado en ambos casos.



```
File Edit Selection ...  ← →  Untitled (Workspace)

index.html X
01_intro_javascript > index.html > html > head > meta
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Document</title>
7 </head>
8 <body>
9
10 </body>
11 </html>
```

Ilustración 1: Código básico de un archivo HTML5

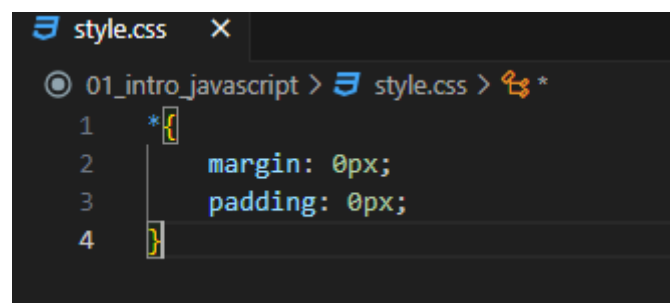
El código resultante de aplicar la técnica explicada se puede ver en la imagen anterior.

Por otro lado, podemos crear un fichero HTML con mucha menos información, o etiquetas HTML, ya que simplemente escribiendo la etiqueta “HTML” en dicho archivo funcionaria igualmente, pero estamos omitiendo mucha información que posteriormente puede ser contraproducente. Así que el archivo HTML de nuestros proyectos, aunque estos proyectos sean ahora muy sencillos, siempre con la estructura básica de html5.

Fichero style.CSS

Dicho fichero va a albergar todos los estilos del proyecto, es decir, que solamente incluirá las instrucciones CSS para tal fin.

Como curiosidad, para estandarizar las webs, los profesionales del desarrollo web usan la estructura básica siempre al inicio de cualquier CSS ya que los navegadores no alinean todos los elementos igualmente, y para evitar problemas, tienen la costumbre de incluir siempre al inicio de sus CSS:



```
style.css X
01_intro_javascript > style.css > *
1 *{
2   margin: 0px;
3   padding: 0px;
4 }
```

Ilustración 2: código básico CSS.

```
1. *{
```

```
2. margin: 0px;  
3. padding: 0px;  
4. }
```

Fichero main.js

Este fichero contendrá todos los códigos JavaScript necesarios para que la aplicación web a desarrollar funcione correctamente.

DOS EJEMPLOS SENCILLOS

Para comprobar y experimentar con todo lo dicho hasta el momento vamos a realizar un par de ejemplos sencillos.

LA WEB MÁS SENCILLA

Aunque escribamos nuestras aplicaciones desde cero no tiene por qué ser una realización lenta y pesada. Es más, podemos escribir la web más sencilla que existe con poquísimas líneas de código, y, para ello vamos a verlo con un ejemplo:

El nombre de los ficheros, las carpetas o subcarpetas que incluyan son una fuente de problemas si no se siguen una serie de normas. Como las más importantes e inevitables:

- No incluir espacios en los nombres de ficheros o carpetas.
- No incluir caracteres especiales.
- Usar el guion bajo para representar espacios.
- Usar un numero al principio por mantener nuestros proyectos ordenados y saber cual es el orden que seguimos al crearlos.
- No usar mayúsculas y minúsculas en los nombres.

Muchos de estos consejos vienen por que los servidores son incapaces de trabajar con estos caracteres o estos nombres “mal generados”. No es que estén mal escritos simplemente es que si no seguimos estas normas cuando estemos trabajando con servidores en la nube vamos a tener muchísimos problemas a la hora de ejecutar nuestras aplicaciones.

Vamos primeramente a crear una carpeta en la que incluir los ficheros habituales de nuestros proyectos en JavaScript, yo la he llamado 01_web_mas_sencilla, y dentro de ella crearemos los siguientes documentos con el código que se indica a continuación.

Fichero index.html:

```
<html>
  web más sencilla
</html>
```

Fichero style.css:

```
*{
  margin: 0px;
  padding: 0px;
}
```

Fichero main.js.

En este caso este fichero estará vacío.

Como vemos, el fichero HTML no contiene más que una etiqueta, que es la única realmente imprescindible para crear una web. El fichero style.css contiene lo convenido para ajustar la web. Y el fichero main.js no incluye nada de código en su interior. Y realmente funciona:

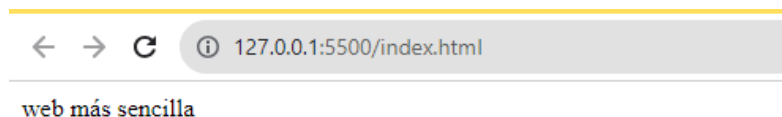


Ilustración 3: resultado de salida de la web más sencilla.

Aunque esa web funcione, deja mucho que desear, sobre todo en cuanto a estandarización y buenas prácticas se refiere.



Ilustración 4: Inspeccionando la web más sencilla.

Al inspeccionar dicha web mediante el inspector de código del navegador que estemos usando podemos visualizar que el propio interprete del fichero HTML ha añadido etiquetas HTML por su cuenta sin la participación nuestra. Se puede observar que ha añadido las etiquetas HEAD y BODY. Dichas etiquetas aparecen en el inspector de elementos que hemos conseguido mostrar al llamar a la consola del navegador (tecla F12 en Google Chrome) y seleccionando la pestaña elementos, ya en la consola.

Estas etiquetas no se han añadido a nuestro fichero original en HTML, porque, si lo revisamos ahora mismo, se puede comprobar que no se encuentran dentro del fichero. Pero el intérprete de Google Chrome sí que las incluye virtualmente con la copia que tiene de la web en memoria para estandarizar la web.

Como se puede comprobar tenemos una web totalmente funcional, aunque no podemos esperar mucho de ella, pero es que el código que hemos introducido tampoco ha sido demasiado, para poder esperar algo más de esas pocas líneas de código.

WEB NORMALIZADA

Ahora vamos a volver a crear una web similar a la anterior pero esta vez con algún que otro código más para ver la diferencia entre una web reducida al máximo con las carencias que tenía y una web bastante más estándar y ajustada a los cánones actuales de programación web.

Fichero HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="style.css">
  <title>Web normalizada</title>
</head>
<body>
  <div>
    Ejemplo de web sencilla pero completa y normalizada para HTML5
  </div>
  <script src="main.js"></script>
</body>
</html>
```

A resaltar en este fichero:

- Hemos incluido una estructura simple de HTML5.
- Hemos incluido un DIV, que no es más que un subespacio creado dentro de la etiqueta BODY.
- También hemos incluido un texto dentro de este espacio DIV.

Fichero CSS;

```
*{
  margin: 0px;
  padding: 0px;
}
body{
  background-color: blue;
  color: white;
  font-size: 30px;
  font-family: Arial, Helvetica, sans-serif;
}
div{
  background-color: black;
  border: 5px;
  margin: 5px;
  border-radius: 15px;
  padding: 10px;
}
```

Para el fichero CSS se puede ver que además de los valores del “margin” y “padding” que solemos incluir en todos los proyectos hemos hecho:

- Para el BODY hemos asignado:
 - Un color de fondo, en este caso, azul.
 - Un color de letra blanco.
 - Un tamaño de letra de 30 pixeles.
 - La familia de fuentes Arial.
- Para el subespacio asignado a DIV hemos fijado:
 - Un color de fondo negro.
 - Un borde de 5 pixeles por todos los lados del espacio que ocupa la etiqueta DIV.
 - Un margen de 5 pixeles por todos los lados del espacio que ocupa la etiqueta DIV.
 - Un radio de borde de 15 pixeles para hacer esa caja del espacio DIV con los bordes redondeados.
 - Un padding o espacio interior de 10 pixeles para que el texto se ajuste perfectamente a esta subárea que hemos creado.

Fichero main.js

Como nuestro empeño en este módulo es aprender JavaScript y aunque la web debe ser lo más sencilla posible, vamos a poner un comentario en el archivo main.js para que aparezca algo en consola y se pueda saber que se está ejecutando correctamente el JavaScript que incluimos.

```
console.log("Ejecutando la web más sencilla....Pero normalizada.");
```

Solamente incluimos la línea anterior en nuestro fichero “main.js” y el resultado será algo similar a lo que se puede ver en la siguiente imagen:



Ilustración 5: resultado web normalizada.

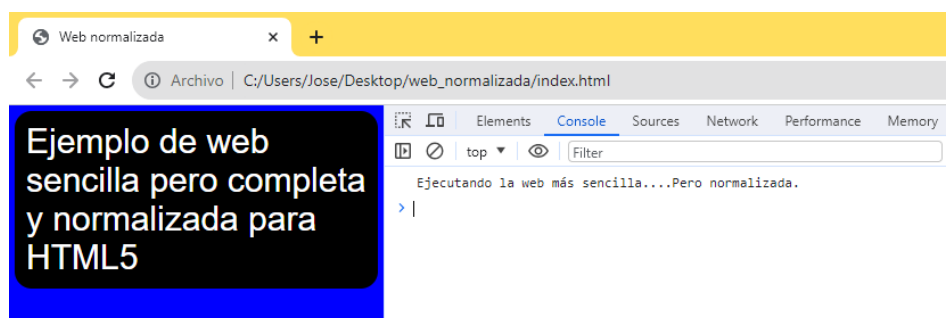


Ilustración 6: resultado y consola de web normalizada.

Se puede observar que se muestra por la consola lo que hemos escrito en el fichero main.js, por tanto, nuestro proyecto está funcionando correctamente y sin ningún fallo, si tuviéramos algún fallo nos aparecería justamente ahí donde nos ha mostrado el texto que le pedíamos a la aplicación que mostrase.

Todos estos mini proyectos que hacemos tienen una explicación de por qué se hacen ahora al principio.

- Sirven para situar al usuario y hacerle ver cómo se va a actuar en cada proyecto estandarizando y aplicando siempre la misma metodología.
- Nos muestra sitios claves a los que controlar a la hora de escribir nuestro código.
- Nos van enseñando gradualmente la potencia, complejidad y elegancia al escribir código y como se debe hacer correctamente.

Con cada ejemplo nos van introduciendo en la magia del desarrollo de las webs dinámicas que se consigue mezclando tecnologías como HTML5, CSS, ambos estáticos, y JavaScript que es quien pone toda la magia para conseguir esa dinamicidad que hacemos mención.



Ilustración 7: Iconos de HTML5, JavaScript y CSS3.

Resumiendo, para acabar esta larga introducción, hemos aprendido a:

- Configurar el entorno para ser más rápidos y eficientes escribiendo código.
- Manejar las herramientas necesarias en cada momento según las necesidades que tengamos: Consola, IDE o editor en línea.
- Evitar errores con las variables en JavaScript.
- Construir la estructura básica de cualquier proyecto JavaScript.

Además, hemos visto dos pequeños proyectos en los que hemos visto aplicado casi todo lo aprendido:

- La web más sencilla que existe. Nos ha enseñado que con unas pocas líneas ya tenemos una web totalmente funcional.
- La web normalizada. Nos ha mostrado que para escribir una web normalizada no hace falta tanto.

Y además de todo esto hemos dado nuestros primeros pasitos en JavaScript sin darnos apenas cuenta.

FUNDAMENTOS JAVASCRIPT

Tal como hemos visto, JavaScript es un lenguaje interpretado, o sea, no es compilado, de desarrollo para webs dinámicas. Con este lenguaje se pueden crear estructuras de datos y manejar elementos del DOM y el BOM para conseguir que una web sea totalmente dinámica y no tan estática como queda con HTML y CSS.

JavaScript interactúa con el DOM para manipular y modificar elementos HTML y CSS en la página. Puede acceder a los elementos del DOM utilizando métodos y propiedades proporcionados por el navegador. Además, JavaScript puede registrar callbacks para eventos específicos, como clics de botón o cambios en el valor de un campo de entrada, y responder a ellos ejecutando el código asociado.

HISTORIA DE JAVASCRIPT

JavaScript nació en el año 1995 centrado en el navegador (ya desaparecido) Netscape. Todos los navegadores lo adoptaron desde entonces para dotar a las webs de ese dinamismo que no se consigue con HTML y CSS únicamente.

Aunque JavaScript surgió como un lenguaje de script para mejorar las capacidades de la web de la época allá por 1995 por la extinta Netscape, JavaScript no ha dejado de evolucionar desde entonces. Originalmente el lenguaje se basaba en CEnví desarrollado por Nombas.

Brendan Eich, un programador que trabajaba en Netscape pensó que podría solucionar **las limitaciones de la web de entonces**, adaptando otras tecnologías existentes (como ScriptEase) al navegador Netscape Navigator 2.0, que iba a lanzarse en aquel año. Inicialmente, Eich denominó a su lenguaje LiveScript y fue un éxito.

Fue entonces cuando, justo antes del lanzamiento, Netscape decidió **cambiar el nombre por el de JavaScript y firmó una alianza con Sun Microsystems** para continuar el desarrollo del nuevo lenguaje de programación.

Microsoft, al ver el movimiento de uno de sus principales competidores, también decidió incorporar **su propia implementación de este lenguaje**, llamada JScript, en la versión 3 de su navegador Internet Explorer.

Esto contribuyó todavía más al empuje y popularización del lenguaje, pero comenzaron a presentarse pequeños problemas por las **diferencias entre implementaciones**. Por lo que se decidió **estandarizar ambas mediante la versión JavaScript 1.1** como propuesta a ECMA, que culminó con el estándar ECMA-262. Este estándar dicta la base del lenguaje ECMAScript a través de su sintaxis, tipos, sentencias, palabras clave y reservadas, operadores y objetos, y sobre la cual se

pueden construir distintas implementaciones. La versión **JavaScript 1.3** fue la primera **implementación completa del estándar ECMAScript**.

VERSION	AÑO	
ECMAScript 3	1999	<ul style="list-style-type: none">• Soporte expresiones regulares.• Manejo de excepciones (try-catch)• Nuevas sentencias de control.• Definición de errores más precisa.• Formateo de salidas numéricas.• Manejo de cadenas literales más avanzado.
ECMAScript 4	2004	<ul style="list-style-type: none">• Introduce tipado de variables.• Introduce el concepto tradicional de POO con clases e interfaces.
ECMAScript 5	2009	<ul style="list-style-type: none">• Getters y setters.• Manipulación de propiedades de un objeto.• Crear objetos de forma dinámica.• Impedir que un objeto sea modificado.• Cambios en el objeto Date.• Soporte nativo JSON.
ECMAScript 6	2015	<ul style="list-style-type: none">• Uso de módulos.• Ámbito a nivel de bloque (let)• Desestructuración.• Y muchas novedades de JavaScript avanzado...
ECMAScript 7	2016	

Como curiosidad aquí tenéis un [timeline](#) realizado usando JavaScript que habla sobre la historia de JavaScript. Un claro ejemplo de las cosas espectaculares que se pueden hacer con este lenguaje de script llamado JavaScript.

CARACTERISTICAS DE JAVASCRIPT

JavaScript es un lenguaje interpretado debido a la forma en que se ejecuta y procesa su código.

- **Entorno de ejecución en el navegador:** JavaScript se ejecuta principalmente en los navegadores web. Cuando se carga una página web que contiene código JavaScript, el navegador interpreta y ejecuta ese código directamente en su entorno de ejecución interno. Esto significa que no se requiere un paso de compilación previo antes de ejecutar el código JavaScript.
- **Análisis y ejecución línea por línea:** a diferencia de los lenguajes compilados, en los que todo el código se traduce en instrucciones de máquina antes de su ejecución, JavaScript se interpreta línea por línea a medida que se encuentra. El motor de JavaScript del navegador analiza y ejecuta cada línea de código a medida que se encuentra durante el proceso de carga de la página.

- **Flexibilidad y facilidad de desarrollo:** el enfoque interpretado de JavaScript brinda flexibilidad y facilidad de desarrollo. No es necesario compilar el código antes de probarlo, lo que permite un desarrollo rápido y una iteración ágil. Los cambios realizados en el código JavaScript se pueden probar y ver inmediatamente sin la necesidad de un proceso de compilación.
- **Plataforma independiente:** el hecho de que JavaScript sea un lenguaje interpretado contribuye a su capacidad de ser ejecutado en diferentes plataformas y sistemas operativos. Los navegadores web son la plataforma principal para la ejecución de JavaScript, y los navegadores modernos han implementado sus propios motores para interpretar y ejecutar el código en diferentes sistemas.
- **Fácil actualización y distribución:** con JavaScript interpretado, los cambios o actualizaciones en el código se pueden implementar fácilmente, ya que los usuarios solo necesitan cargar la nueva versión del código en sus navegadores. No es necesario que los usuarios realicen una instalación manual o actualización de un programa o aplicación.

POO

JavaScript es un lenguaje de programación orientado a objetos (OOP), que proporciona funcionalidades para trabajar con objetos y clases. Aunque no sigue un enfoque de OOP estricto como lenguajes como Java o C++, ofrece características que permiten la implementación de conceptos orientados a objetos.

- **Objetos:** en JavaScript, los objetos son colecciones de propiedades y métodos que representan entidades del mundo real. Las propiedades son variables que almacenan valores, mientras que los métodos son funciones que realizan acciones relacionadas con el objeto. Los objetos se crean utilizando la sintaxis de llaves {} y pueden ser personalizados para contener propiedades y métodos específicos.
- **Prototipos:** utiliza un modelo de herencia basado en prototipos en lugar de clases tradicionales. Cada objeto tiene un prototipo, que es otro objeto del cual hereda propiedades y métodos. Cuando se accede a una propiedad o método en un objeto, si no se encuentra directamente en ese objeto, JavaScript busca en su prototipo y continúa escalando por la cadena de prototipos hasta que se encuentra o se alcanza el objeto raíz.
- **Constructores:** permite la creación de objetos utilizando funciones constructoras. Una función constructora es una función regular que se invoca utilizando la palabra clave «new». Dentro de la función constructora se definen las propiedades y métodos del objeto que se está creando. Cada vez que se crea un objeto utilizando una función constructora, se crea una instancia independiente con sus propias propiedades y métodos.
- **Herencia:** aunque JavaScript no ofrece una sintaxis de clase tradicional para la herencia, se puede conseguir la herencia utilizando prototipos. Un objeto puede heredar propiedades y métodos de otro objeto estableciendo su prototipo en ese objeto padre. Esto permite la reutilización de código y la creación de jerarquías de objetos.
- **Encapsulamiento:** JavaScript no tiene un mecanismo nativo de encapsulamiento como las clases en otros lenguajes, pero se puede lograr usando funciones para crear ámbitos locales. Al definir variables y funciones dentro de una función, se pueden ocultar y proteger del acceso externo, creando así un nivel básico de encapsulamiento.
- **Polimorfismo:** permite la implementación de polimorfismo, que es la capacidad de objetos de diferentes tipos para responder al mismo mensaje o llamada de método. Dado que JavaScript es un lenguaje dinámico, las variables pueden contener diferentes tipos de

objetos en distintos momentos, y se puede acceder a sus métodos y propiedades de manera flexible.

DOM

La integración de JavaScript con el DOM (Document Object Model) es una de las características fundamentales de JavaScript y es lo que permite interactuar y manipular elementos HTML y CSS en una página web.

- **Acceso al DOM:** JavaScript utiliza el objeto `document` para acceder y manipular el DOM. El objeto `document` representa el documento HTML cargado en el navegador y proporciona métodos y propiedades para interactuar con él. Puedes acceder a elementos específicos del DOM utilizando métodos como `getElementById()`, `querySelector()`, `getElementsByName()`, entre otros.
- **Manipulación de elementos:** una vez que tienes una referencia a un elemento en el DOM, puedes utilizar las propiedades y métodos disponibles para manipularlo. Por ejemplo, puedes cambiar el contenido de un elemento utilizando la propiedad `innerHTML`, modificar los estilos utilizando la propiedad `style`, agregar o eliminar atributos utilizando los métodos `setAttribute()` y `removeAttribute()`, y mucho más.
- **Eventos del DOM:** JavaScript permite responder a eventos que ocurren en los elementos del DOM, como hacer clic en un botón, pasar el cursor sobre un elemento o enviar un formulario. Puedes registrar eventos utilizando métodos como `addEventListener()`, especificando el tipo de evento y una función de callback que se ejecutará cuando ocurra el evento. Dentro de la función de callback, puedes escribir el código que deseas ejecutar en respuesta al evento.
- **Manipulación de la estructura del DOM:** además de manipular los elementos individuales, JavaScript también permite manipular la estructura del DOM, es decir, agregar, eliminar y modificar elementos en la página. Puedes crear nuevos elementos utilizando el método `createElement()`, agregarlos al DOM usando métodos como `appendChild()` o `insertBefore()`, y eliminar elementos empleando `removeChild()`.
- **Actualización dinámica de la página:** una de las ventajas clave de la integración de JavaScript con el DOM es la capacidad para realizar actualizaciones dinámicas en la página sin tener que recargarla completa. Puedes cambiar el contenido de un elemento en respuesta a eventos, actualizar estilos, mostrar u ocultar elementos, realizar animaciones y mucho más. Esto permite crear experiencias interactivas y dinámicas para los usuarios.

BASADO EN EVENTOS

JavaScript es un lenguaje «event-driven» o basado en eventos, lo que significa que se ejecuta en respuesta a eventos que ocurren en el entorno en el que se está ejecutando, como acciones del usuario o eventos del sistema.

- **Registro de eventos:** para comenzar a responder a eventos, se deben registrar los eventos que se desean detectar. En JavaScript, esto se realiza utilizando el método `addEventListener()` en un elemento específico del DOM (Document Object Model) o en un

objeto relevante. Se especifica el tipo de evento (por ejemplo, «click» para un clic de ratón) y una función de «callback» que se ejecutará cuando ocurra el evento.

- **Funciones de callback:** las funciones de callback se ejecutan en respuesta a un evento específico. Estas funciones se definen previamente y se pasan como argumento al método `addEventListener()`. Cuando se produce el evento registrado, JavaScript ejecuta la función de callback correspondiente.
- **Ejecución asíncrona:** el enfoque «event-driven» en JavaScript permite la ejecución asíncrona, lo que significa que el código no se bloquea esperando eventos. El flujo de ejecución continúa mientras el programa espera eventos, y cuando ocurre un evento registrado, se dispara la ejecución de la función de callback asociada.
- **Event bubbling y event capturing:** JavaScript también proporciona dos modelos de propagación de eventos: «event bubbling» (burbujeo de eventos) y «event capturing» (captura de eventos). El «event bubbling» hace que los eventos se propaguen desde el elemento objetivo hacia los elementos padres, mientras que el «event capturing» hace que los eventos se propaguen desde los elementos padres hacia el elemento objetivo. Esto permite controlar cómo se manejan los eventos en diferentes elementos.
- **Manipulación del DOM:** una de las principales aplicaciones del enfoque «event-driven» en JavaScript es la manipulación del DOM. Al registrar eventos en elementos del DOM, se puede responder a las interacciones del usuario, como clics, cambios de valor en campos de entrada o movimiento del ratón. Esto posibilita crear interacciones dinámicas y responsivas en aplicaciones web.

ASINCRONISMO

El asincronismo en JavaScript es fundamental para realizar tareas sin bloquear la ejecución del código, lo que permite que el programa sea más eficiente y receptivo.

- **Callbacks:** antes de que las promesas y el `async/await` estuvieran disponibles, eran una forma común de implementar la asincronía en JavaScript. Un callback es una función que se pasa como argumento a otra función y se ejecuta una vez que se completa una operación asíncrona. Por ejemplo, en una llamada AJAX, se proporciona un callback que se ejecuta cuando se recibe la respuesta del servidor.
- **Event Loop:** JavaScript utiliza un mecanismo llamado Event Loop (bucle de eventos) para manejar la asincronía. El Event Loop se encarga de monitorear constantemente la pila de llamadas y la cola de tareas. Cuando todas las tareas en la pila de llamadas se han ejecutado, el Event Loop verifica la cola de tareas en busca de tareas pendientes y las ejecuta en orden.
- **Promesas:** las promesas son una forma más moderna y legible de manejar la asincronía en JavaScript. Una promesa es un objeto que representa la eventual finalización o falla de una operación asíncrona y permite encadenar acciones después de que se haya resuelto o rechazado. Puedes definir una promesa que realiza una tarea y luego utilizar los métodos `then()` y `catch()` para manejar el resultado.
- **Async/await:** el `async/await` es la sintaxis más reciente que se ha introducido en JavaScript y simplifica aún más el manejo de la asincronía. Permite escribir código asíncrono de manera más similar al código síncrono, lo que lo hace más legible y fácil de entender. Puedes marcar una función como `async` y utilizar la palabra clave `await` para esperar la resolución de una promesa antes de continuar la ejecución.

- **Llamadas a API:** una de las aplicaciones comunes del asincronismo en JavaScript es realizar llamadas a API, como solicitudes AJAX o fetch para obtener datos de servidores externos. Estas operaciones suelen ser asíncronas, y el código espera la respuesta de la API antes de continuar con otras tareas.

VARIABLES

El lenguaje JavaScript es un lenguaje débilmente tipado lo que significa, entre otras implicaciones, que no es necesario declarar que tipo de variable es al declarar dicha variable, así, como tampoco es necesario que se mantengan durante toda la ejecución de la aplicación siendo de ese mismo tipo con el que se inicio al principio de dicha aplicación. Ejemplos:

```
let miVariable;           // declaro miVariable y como no se asignó un valor valdrá undefined
miVariable='Hola';        // ahora su valor es 'Hola', por tanto, contiene una cadena de texto
miVariable=34;            // pero ahora contiene un número
miVariable=[3, 45, 2];    // y ahora un array
miVariable=undefined;     // para volver a valer el valor especial undefined
```

EJERCICIO: Ejecuta en la consola del navegador las instrucciones anteriores y comprueba el valor de miVariable tras cada instrucción (para ver el valor de una variable simplemente ponemos en la consola su nombre: miVariable).

JavaScript es tan laxo con el uso de las variables que ni siquiera nos obliga a declarar dichas variables antes de usarlas, pero, es un problema que posteriormente pagaremos en la depuración, perdiendo más tiempo que el que puede suponer declarar correctamente una variable.

VAR, LET Y CONST

Se usa la instrucción var para declarar variables de tipo global, es decir, el ámbito de dicha variable será la aplicación completa.

Se usa la instrucción let (recomendado desde ES2015) para declarar variables de bloque o locales ya que su ámbito es reducido al bloque o a un ámbito más pequeño que en el caso de la instrucción var.

Veamos un ejemplo de su uso:

```
let edad = 17;

if (edad >= 18) {
  let textoLet = 'Eres mayor de edad';
  var textoVar = 'Eres mayor de edad';
} else {
  let textoLet = 'Eres menor de edad';
}
```



```

var textoVar = 'Eres menor de edad';
}
//console.log(textoLet); // mostrará undefined porque fuera del if no existe la variable
console.log(textoVar); // mostrará la cadena

```

Es altamente recomendable, por no decir, obligatorio el uso de la tipología camelCase en JavaScript (ej.: miPrimerApellido)

Desde ES2015 podemos hacer uso de la palabra reservada “const” para declarar constantes se les asigna un valor al declarar dichas constantes y si posteriormente intentamos cambiar su valor nos producirá un error.

```

const miNombre = "José Manuel";
const miTelefono = 123456789;
const miSituacionLaboral = true; // true --> trabajando false ---> desempleado
const misAnimales = [true,true, true]; // 0: perro 1:gato 2:pajaro

```

Cualquier variable que no sea declarada dentro de un bloque o una función, o si usamos una variable sin declarar previamente es considerada una variable global.

FUNCIONES

Las funciones se declaran con la palabra reservada “function” y en el caso de que necesite parámetros se le pasan entre paréntesis. Las funciones siempre devuelven un valor, en el caso de contener un return dentro de la función se devuelve el valor asignado a esa orden, en caso contrario, se devuelve undefined por parte de la función implicada.

Las funciones en JavaScript no es necesario declararlas al principio (hoisting de JavaScript), se pueden declarar incluso todas al final del código que estamos creando ya que el interprete del navegador al principio lee todas las variables y funciones y las mueve al principio del código para posteriormente empezar la ejecución línea por línea.

Ejercicio: Crea una función con un parámetro de entrada llamado “texto”, la ejecución de esa función debe mostrar una alerta con el texto que has introducido como parámetro al hacer la llamada de dicha función y al que le agregaras anteriormente el texto: “Has escrito...” seguido de tu texto escrito en la llamada a la función.

Parámetros

Si llamo a una función con menos parámetros de los que se han declarado al crear la función, el valor de esos parámetros no declarados será undefined.

```

function potencia(base, exponente) {
    console.log(base);           // muestra 4
    console.log(exponente);      // muestra undefined
    let valor=1;
    for (let i=1; i<=exponente; i++) {

```

```

        valor=valor*base;
    }
    return valor;
}

potencia(4);    // devolverá 1 ya que no se ejecuta el for

```

Pregunta: ¿Cómo sería la llamada correcta a la función que acabamos de declarar? Escribe al menos una llamada correcta a esa función.

Atención: ¿Por qué hace llamadas a la consola dentro de la función? ¿Hacer esas llamadas a consola desde dentro de la función es correcto?

Para evitar esta problemática con los parámetros de entrada de las funciones podemos usar la preasignación de valores a los parámetros de la siguiente forma:

```

function potencia(base, exponente=2) {
    console.log(base);           // muestra 4
    console.log(exponente);      // muestra 2 la primera vez y 5 la segunda
    let valor=1;
    for (let i=1; i<=exponente; i++) {
        valor=valor*base;
    }
    return valor;
}

console.log(potencia(4));        // mostrará 16 (4^2)
console.log(potencia(4,5));     // mostrará 1024 (4^5)

```

Podemos acceder a los parámetros de una función desde el array `arguments[]` en el caso de no conocer el número de parámetros que finalmente nos pasó el usuario:

```

function suma () {
    var result = 0;
    for (var i=0; i<arguments.length; i++)
        result += arguments[i];
    return result;
}

console.log(suma(4, 2));          // mostrará 6
console.log(suma(4, 2, 5, 3, 2, 1, 3)); // mostrará 20

```

JavaScript trata las funciones como un tipo de dato más y podemos pasarlas como argumento o asignarlas a una variable:

```

const cuadrado = function(value) {
    return value * value
}

function aplica_fn(dato, funcion_a_aplicar) {
    return funcion_a_aplicar(dato);
}

```

```
}  
  
aplica_fn(3, cuadrado);    // devolverá 9 (3^2)
```

A las funciones así usadas se les llama funciones de primera clase y son muy típicas de lenguajes funcionales.

Funciones anónimas

Acabamos de usar una función, no sé si os habréis fijado en el detalle, que no tiene nombre. Se denominan funciones anónimas. Esta función, o cualquier función, puede ser asignada a una variable, autoejecutarse o ser asignada a un manejador de eventos. Ejemplo:

```
let holaMundo = function() {  
    alert('Hola mundo!');  
}  
  
holaMundo();    // se ejecuta la función
```

En este caso concreto, debemos asignar esa función anónima a una variable para que llamando a esa variable se pueda ejecutar la función. Se debe observar que la variable que ahora contiene la función anónima se debe hacer uso de ella utilizando el paréntesis, ya que de lo contrario no ejecutaría la función que nos ocupa.

Funciones flecha

Las funciones flecha o funciones lambda es una forma de declarar una función más corta que la forma tradicional de declaración de funciones. Se permiten desde ES2015. Ejemplo de declaración de una función de manera tradicional:

```
let potencia = function(base, exponente) {  
    let valor=1;  
    for (let i=1; i<=exponente; i++) {  
        valor=valor*base;  
    }  
    return valor;  
}
```

Ahora, si pretendemos reescribir esta misma función en modo función flecha debemos:

- Eliminar la palabra reservada function.
- Si solamente posee un parámetro podemos eliminar los paréntesis de los parámetros.
- Colocamos el símbolo =>

- Si la función solamente tiene una línea podemos eliminar las llaves {} y la palabra reservada return.

El mismo ejemplo anterior en formato flecha sería:

```
let potencia = (base, exponente) => {  
  let valor=1;  
  for (let i=1; i<=exponente; i++) {  
    valor=valor*base;  
  }  
  return valor;  
}
```

Otro ejemplo declarando la función de manera tradicional:

```
let cuadrado = function(base) {  
  return base * base;  
}
```

El mismo ejemplo en formato flecha:

```
let cuadrado = base => base * base;
```

EJERCICIO: Haz una función flecha que devuelva el cubo del número pasado como parámetro y pruébala desde la consola. Escríbela primero en la forma habitual y luego la “traduces” a formato flecha.

ESTRUCTURAS Y BLOQUES

CONDICIONAL IF

Ejemplo de un condicional de tipo IF sencillo

```
let edad = 17;  
  
if (edad >= 18) {  
  let textoLet = 'Eres mayor de edad';  
  var textoVar = 'Eres mayor de edad';  
}  
  
//console.log(textoLet); // mostrará undefined porque fuera del if no existe la variable  
console.log(textoVar); // mostrará la cadena
```

En el caso anterior se pueden distinguir dos partes principales en el condicional IF, la primera parte principal se encuentra entre paréntesis y nos indica la condición que debe cumplirse para que se

ejecute lo que existe dentro de la segunda parte principal a destacar que está dentro de las llaves del final del condicional. En este ejemplo no existe la cláusula ELSE que se ejecutaría en caso de no cumplirse la condición anteriormente comentada. Vemos a continuación el mismo condicional, esta vez sí, con cláusula ELSE:

```
let edad = 17;

if (edad >= 18) {
  let textoLet = 'Eres mayor de edad';
  var textoVar = 'Eres mayor de edad';
} else {
  let textoLet = 'Eres menor de edad';
  var textoVar = 'Eres menor de edad';
}

//console.log(textoLet); // mostrará undefined porque fuera del if no existe la variable
console.log(textoVar); // mostrará la cadena
```

Como hemos dicho anteriormente, en esta última propuesta del condicional se evalúa siempre la condición entre paréntesis, si es verdadera esa condición se ejecuta el código encerrado entre llaves a continuación de la condición. Si fuese falsa esa condición se ejecutaría la segunda sección de código que va entre llaves después de la condición.

El condicional IF admite encadenar sucesivos condicionales mediante el uso de ELSE IF, esta construcción lógica se usa para contemplar todos los casos posibles a tratar frente a unas especificaciones, veamos un ejemplo:

```
let edad = 29;

if(edad >= 18 && edad < 120){
  console.log("Eres mayor de edad");
}else if(edad < 12){
  console.log("Eres un niño.");
}else if(edad >= 12 && edad < 18){
  console.log("Eres un adolescente");
}else{
  console.log("Tu edad no es correcta.");
}
```

En el caso anterior al valer la variable edad 29 el condicional primero se cumple y se ejecuta el código mostrado entre llaves a continuación, mostrando por consola el mensaje que nos indica que el usuario es un adulto.

Haciendo suposiciones y que funcionase correctamente el código, si imaginamos que ahora la variable edad puede cambiar y valer 11 se evaluaría el primer condicional y como no es verdadero ya que 11 es menor de 120 pero no es mayor que 18 y puesto que hemos usado un operador lógico AND para unir dichas condiciones, el resultado de una operación falso AND verdadero es falso, por tanto, se evaluaría el segundo condicional resultando que es verdadero, ya que 11 es menor que 12.

Por lo que se ejecutaría el código entre llaves justo tras esa condición mostrando el mensaje en consola "Eres un niño".

Volvemos a hacer otra suposición, ahora igualamos la variable edad a 15, entramos en el primer condicional y al evaluar esa condición vemos que su resultado es falso, ya que aunque es verdadero que 15 es menor que 120, también, es falso que 15 es mayor o igual a 18., por tanto, no se ejecuta el código entre llaves del primer condicional y pasamos a evaluar la segunda condición, esta vez, 15 no es cierto que sea menor que 12, o sea, condición falsa, no se ejecuta el código entre llaves tras esa segunda condición y pasamos a evaluar la tercera condición. La tercera condición nos asegura que si edad es menor que 18, si es menor que 18 el numero 15, y la edad es mayor o igual que 12, que también es cierta entonces se ejecuta el código entre llaves a continuación, indicándonos por consola que "Eres un adolescente."

Y así se irían evaluando cada una de las condiciones descritas asegurándonos cubrir todas las posibilidades al crear dichos condicionales. Y si lo hemos diseñado bien y lo hemos implementado correctamente, tal y como lo hemos diseñado. Nunca deben fallar nuestros condicionales. En el caso que nos ocupa, si llegase al ultimo condicional y ninguno hubiera sido cierto nos daría un mensaje de error que en este caso hemos definido como: "Tu edad no es correcta."

CONDICIONAL SWITCH

BUCLE WHILE

BUCLE FOR

Ya hemos visto en funcionamiento el bucle FOR al definir las distintas formas de declarar funciones en JavaScript. Pero volvamos a él, desmenuzándolo adecuadamente para disipar cualquier duda existente sobre esta construcción.

```
let potencia = (base, exponente) => {  
  let valor=1;  
  for (let i=1; i<=exponente; i++) {  
    valor=valor*base;  
  }  
  return valor;  
}
```

En todo bucle FOR tenemos dos partes principales:

- La parte encerrada entre paréntesis. Este apartado nos muestra la configuración del bucle FOR, ya que aquí decidimos que variable usamos para iterar en cada ocasión, es el caso de nuestro ejemplo donde se define la variable con el nombre "i". También decidimos la condición que se debe cumplir y que debe ser cierta para que se continúe iterando, en nuestro ejemplo viene definido por esa condición "i <= exponente". Y para finalizar, también definimos los saltos de valor que va a dar la variable en cada una de las iteraciones, en nuestro ejemplo viene definido como "i++". Es decir, si saltamos de uno en uno en los valores asignados a la variable "i", como indica esta última instrucción "i++" analizada, en cada iteración. Las iteraciones se irán produciendo hasta que el valor en la variable "i" sea menor o igual que el valor del parámetro "exponente". Si esa condición llegase a ser falsa en esa iteración ya ejecutaría el código entre llaves y se daría por finalizado el bucle FOR. CUIDADO con los bucles que podemos definir condiciones imposibles y que siempre se cumplan y entrar en un bucle infinito que colgaría nuestra aplicación.
- La parte encerrada entre llaves. Como ya hemos comentado el código incluido entre llaves se ejecuta, si y solo si, el condicional de la declaración del bucle FOR es verdadero. Esta parte puede incluir cualquier código incluidos más bucles FOR, condicionales, o cualquier código que se te ocurra, como funciones de cualquier tipo, objetos, etc.

BUCLE

TIPOS DE DATOS BÁSICOS

CASTING DE VARIABLES

NUMBER

STRING

BOOLEAN

MANEJO DE ERRORES

BUENAS PRACTICAS

use strict

Variables

Errores

OBJETOS

INTRODUCCION

En JavaScript podemos declarar un objeto de tres formas distintas:

- Objetos declarativos o literales.
- Objetos contruidos.
- Objetos usando new Object.

Objetos literal o declarativo

```
let persona = {  
  nombre : 'Jack',  
  edad : 30,  
  saludar: function () {  
    console.log('Hola');  
  }  
};
```

Se declara exactamente igual que cualquier variable con la salvedad de usar las llaves para definir las propiedades o métodos que incluya ese objeto recién creado.

Objeto new Object

El ejemplo anterior construido de esta manera sería:

```
let persona = new Object({
```



```
nombre : 'Jack',  
edad : 30,  
saludar : function () {  
    console.log('Hola');  
}  
});
```

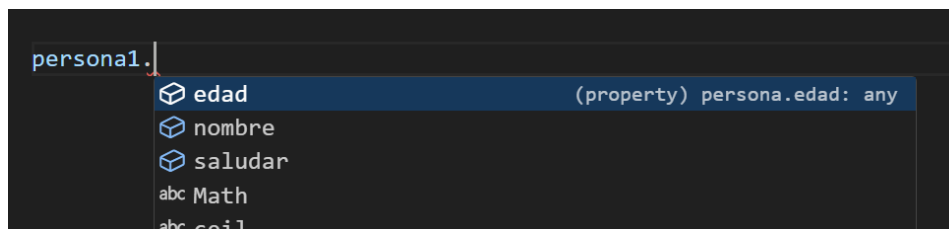
Objeto construido

El ejemplo que vamos siguiendo construido de esta forma sería:

```
function persona(nombre, edad) {  
    this.nombre = nombre;  
    this.edad = edad;  
  
    this.saludar = function(){  
        console.log('Hola');  
    }  
}  
  
let persona1 = new persona('Jack', 30);
```

PROPIEDADES

La manera de acceder a las propiedades de cada objeto que creemos, se declare como se declare el objeto, es siempre con el “.”, es decir, para acceder a la propiedad “nombre” del objeto anteriormente declarado “persona”, se haría así, al pulsar el punto después de añadir el nombre del objeto referenciado, vemos que nos aparecen los métodos y propiedades en la ayuda de VS Code:



Se puede observar que aparecen las propiedades definidas: edad y nombre, así como también, el método definido: saludar.

```
console.log(persona1.edad); // nos mostraria en consola un 30.
```

Como vemos al seleccionar esa propiedad “edad”, nos indicaría por consola la instrucción anterior un 30, ya que fue el numero que asignamos a ese parámetro al definir el objeto.

METODOS

Con los métodos ocurre igual que con las propiedades de los objetos, se referencian con el punto. Y se usa el paréntesis puesto que un método no es más que una función de un objeto.

```
console.log( persona1.saludar() );
```

Esta instrucción anterior ejecuta el código definido en el método declarado en la declaración de dicho objeto y, en nuestro ejemplo, mostraría por consola la palabra “hola”. Su funcionamiento, aunque sea un método de un objeto, es igual a cualquier otra función, la única salvedad es que solamente conoce lo que hace esa función el objeto al que pertenece. Si intentásemos ejecutar dicha función sin hacer referencia a dicho objeto y no existiera una función con ese nombre a nivel global de la aplicación obtendríamos un error.

ARRAYS

Son un tipo de objeto y no tienen tamaño fijo, sino que, podemos añadirle elementos en cualquier momento.

Se recomienda crearlos usando notación JSON:

```
let a = []  
let b = [2,4,6]
```

aunque también podemos crearlos como instancias del objeto Array (NO recomendado):

```
let a = new Array()      // a = []  
let b = new Array(2,4,6) // b = [2, 4, 6]
```

Sus elementos pueden ser de cualquier tipo, incluso podemos tener elementos de tipos distintos en un mismo array. Si no está definido un elemento su valor será *undefined*. Ej.:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
console.log(a[0]) // imprime 'Lunes'
console.log(a[4]) // imprime 6
a[7] = 'Juan'     // ahora a = ['Lunes', 'Martes', 2, 4, 6, , , 'Juan']
console.log(a[7]) // imprime 'Juan'
console.log(a[6]) // imprime undefined
console.log(a[10]) // imprime undefined
```

Acceder a un elemento de un array que no existe no provoca un error (devuelve *undefined*) pero sí lo provoca acceder a un elemento de algo que no es un array. Con ES2020 (ES11) se ha incluido el operador **?.** para evitar tener que comprobar nosotros que sea un array:

```
console.log(alumnos?.[0])
// si alumnos es un array muestra el valor de su primer
// elemento y si no muestra undefined pero no lanza un error
```

Arrays de objetos

Es habitual almacenar datos en arrays en forma de objetos, por ejemplo:

```
let alumnos = [
  {
    id: 1,
    name: 'Marc Peris',
    course: '2nDAW',
    age: 21
  },
  {
    id: 2,
    name: 'Júlia Tortosa',
    course: '2nDAW',
    age: 23
  },
]
```

Operaciones con Arrays

Vamos a ver los principales métodos y propiedades de los arrays.

length

Esta propiedad devuelve la longitud de un array:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
console.log(a.length) // imprime 5
```

Podemos reducir el tamaño de un array cambiando esta propiedad, aunque es una forma poco clara de hacerlo:

```
a.length = 3 // ahora a = ['Lunes', 'Martes', 2]
```

Añadir elementos

Podemos añadir elementos al final de un array con push o al principio con shift:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
a.push('Juan') // ahora a = ['Lunes', 'Martes', 2, 4, 6, 'Juan']
a.unshift(7) // ahora a = [7, 'Lunes', 'Martes', 2, 4, 6, 'Juan']
```

Eliminar elementos

Podemos borrar el elemento del final de un array con pop o el del principio con shift. Ambos métodos devuelven el elemento que hemos borrado:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let ultimo = a.pop() // ahora a = ['Lunes', 'Martes', 2, 4] y ultimo = 6
let primero = a.shift() // ahora a = ['Martes', 2, 4] y primero = 'Lunes'
```

splice

Permite eliminar elementos de cualquier posición del array y/o insertar otros en su lugar. Devuelve un array con los elementos eliminados. Sintaxis:

```
Array.splice(posicion, num. de elementos a eliminar, 1º elemento a insertar, 2º elemento a insertar, 3º...)
```

Ejemplo:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let borrado = a.splice(1, 3) // ahora a = ['Lunes', 6] y borrado = ['Martes', 2, 4]
a = ['Lunes', 'Martes', 2, 4, 6]
borrado = a.splice(1, 0, 45, 56) // ahora a = ['Lunes', 45, 56, 'Martes', 2, 4, 6] y borrado = []
a = ['Lunes', 'Martes', 2, 4, 6]
borrado = a.splice(1, 3, 45, 56) // ahora a = ['Lunes', 45, 56, 6] y borrado = ['Martes', 2, 4]
```

EJERCICIO: Guarda en un array la lista de la compra con Peras, Manzanas, Kiwis, Plátanos y Mandarinas. Haz los siguiente con splice:

- Elimina las manzanas (debe quedar Peras, Kiwis, Plátanos y Mandarinas)
- Añade detrás de los Plátanos Naranjas y Sandía (debe quedar Peras, Kiwis, Plátanos, Naranjas, Sandía y Mandarinas)
- Quita los Kiwis y pon en su lugar Cerezas y Nísperos (debe quedar Peras, Cerezas, Nísperos, Plátanos, Naranjas, Sandía y Mandarinas)

slice

Devuelve un subarray con los elementos indicados, pero sin modificar el array original (sería como hacer un substr pero de un array en vez de una cadena). Sintaxis:

```
Array.slice(posicion, num. de elementos a devolver)
```

Ejemplo:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let subArray = a.slice(1, 3)
// ahora a = ['Lunes', 'Martes', 2, 4, 6] y subArray = ['Martes', 2, 4]
```

Es muy útil para hacer una copia de un array:

```
let a = [2, 4, 6]
let copiaDeA = a.slice()
// ahora ambos arrays contienen lo mismo pero son diferentes arrays
```

Arrays y Strings

Cada objeto (y los arrays son un tipo de objeto) tienen definido el método `.toString()` que lo convierte en una cadena. Este método es llamado automáticamente cuando, por ejemplo, queremos mostrar un array por la consola. En realidad `console.log(a)` ejecuta `console.log(a.toString())`. En el caso de los arrays esta función devuelve una cadena con los elementos del array dentro de corchetes y separados por coma.

Además podemos convertir los elementos de un array a una cadena con `.join()` especificando el carácter separador de los elementos. Ej.:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let cadena = a.join('-') // cadena = 'Lunes-Martes-2-4-6'
```

Este método es el contrario del método `.split()` que convierte una cadena en un array. Ej.:

```
let notas = '5-3.9-6-9.75-7.5-3'
let arrayNotas = notas.split('-') // arrayNotas = [5, 3.9, 6, 9.75, 7.5, 3]
let cadena = 'Que tal estás'
let arrayPalabras = cadena.split(' ') // arrayPalabras = ['Que', 'tal', 'estás']
```

```
let arrayLetras = cadena.split('') // arrayLetras = ['Q','u','e`,`',  
, 't', 'a', 'l', ' ', 'e', 's', 't', 'á', 's']
```

sort

Ordena **alfabéticamente** los elementos del array

```
let a = ['hola', 'adios', 'Bien', 'Mal', 2, 5, 13, 45]  
let b = a.sort() // b = [13, 2, 45, 5, "Bien", "Mal", "adios", "hola"]
```

También podemos pasarle una función que le indique cómo ordenar que devolverá un valor negativo si el primer elemento es mayor, positivo si es mayor el segundo o 0 si son iguales. Ejemplo: ordenar un array de cadenas sin tener en cuenta si son mayúsculas o minúsculas:

```
let a = ['hola', 'adios', 'Bien', 'Mal']  
let b = a.sort(function(elem1, elem2) {  
  if (elem1.toLocaleLowerCase > elem2.toLocaleLowerCase)  
    return -1  
  if (elem1.toLocaleLowerCase < elem2.toLocaleLowerCase)  
    return 1  
  return 0  
}) // b = ["adios", "Bien", "hola", "Mal"]
```

Como más se utiliza esta función es para ordenar arrays de objetos. Por ejemplo si tenemos un objeto *alumno* con los campos *name* y *age*, para ordenar un array de objetos *alumno* por su edad haremos:

```
let alumnosOrdenado = alumnos.sort(function(alumno1, alumno2) {  
  return alumno1.age - alumno2.age  
})
```

Usando *arrow functions* quedaría más sencillo:

```
let alumnosOrdenado = alumnos.sort((alumno1, alumno2) => alumno1.age - alumno2.age)
```

Si lo que queremos es ordenar por un campo de texto debemos usar la función *toLocaleCompare*:

```
let alumnosOrdenado = alumnos.sort((alumno1, alumno2) => alumno1.name.toLocaleCompare(alumno2.name))
```

EJERCICIO: Haz una función que ordene las notas de un array pasado como parámetro. Si le pasamos [4,8,3,10,5] debe devolver [3,4,5,8,10]. Pruébalo en la consola

Otros métodos comunes

Otros métodos que se usan a menudo con arrays son:

`concat()`: concatena arrays

```
let a = [2, 4, 6]
let b = ['a', 'b', 'c']
let c = a.concat(b) // c = [2, 4, 6, 'a', 'b', 'c']
```

`.reverse()`: invierte el orden de los elementos del array

```
let a = [2, 4, 6]
let b = a.reverse() // b = [6, 4, 2]
```

`.indexOf()`: devuelve la primera posición del elemento pasado como parámetro o -1 si no se encuentra en el array.

`.lastIndexOf()`: devuelve la última posición del elemento pasado como parámetro o -1 si no se encuentra en el array.

Functional Programming

Se trata de un paradigma de programación (una forma de programar) donde se intenta que el código se centre más en qué debe hacer una función que en cómo debe hacerlo. El ejemplo más claro es que intenta evitar los bucles *for* y *while* sobre arrays o listas de elementos.

Normalmente cuando hacemos un bucle es para recorrer la lista y realizar alguna acción con cada uno de sus elementos. Lo que hace *functional programming* es que a la función que debe hacer eso se le pasa como parámetro la función que debe aplicarse a cada elemento de la lista.

Desde la versión 5.1 javascript incorpora métodos de *functional programming* en el lenguaje, especialmente para trabajar con arrays:

filter

Devuelve un nuevo array con los elementos que cumplen determinada condición del array al que se aplica. Su parámetro es una función, habitualmente anónima, que va interactuando con los elementos del array. Esta función recibe como primer parámetro el elemento actual del array (sobre el que debe actuar). Opcionalmente puede tener como segundo parámetro su índice y como tercer parámetro el array completo. La función debe devolver **true** para los elementos que se incluirán en el array a devolver como resultado y **false** para el resto.

Ejemplo: dado un array con notas devolver un array con las notas de los aprobados. Esto usando programación *imperativa* (la que se centra en *cómo se deben hacer las cosas*) sería algo como:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = []
for (let i = 0; i++ < arrayNotas.length) {
  let nota = arrayNotas[i]
  if (nota >= 5) {
    aprobados.push(nota)
  }
}
// aprobados = [5.2, 6, 9.75, 7.5]
```

Usando *functional programming* (la que se centra en *qué resultado queremos obtener*) sería:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = arrayNotas.filter(function(nota) {
  if (nota >= 5) {
    return true
  } else {
    return false
  }
}) // aprobados = [5.2, 6, 9.75, 7.5]
```

Podemos refactorizar esta función para que sea más compacta:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = arrayNotas.filter(function(nota) {
  return nota >= 5 // nota >= 5 se evalúa a 'true' si es cierto o
  'false' si no lo es
})
```

Y usando funciones lambda la sintaxis queda mucho más simple:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = arrayNotas.filter(nota => nota >= 5)
```

Vamos a construir un array con los días de la semana que usaremos en los próximos ejercicios:

```
const semana = ['Lunes', 'Martes', 'Miercoles', 'Jueves', 'Viernes', 'Sabado',
'Domingo'];
```

Las 7 líneas del código usando programación *imperativa* quedan reducidas a sólo una.

EJERCICIO: Dado un array con los días de la semana obtén todos los días que empiezan por 'M'

find

Como *filter* pero NO devuelve un **array** sino el primer **elemento** que cumpla la condición (o *undefined* si no la cumple nadie). Ejemplo:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let primerAprobado = arrayNotas.find(nota => nota >= 5) //
primerAprobado = 5.2
```

Este método tiene más sentido con objetos. Por ejemplo, si queremos encontrar la persona con DNI '21345678Z' dentro de un array llamado *personas* cuyos elementos son objetos con un campo 'dni' haremos:

```
let personaBuscada = personas.find(persona => persona.dni ===
'21345678Z') // devolverá el objeto completo
```

EJERCICIO: Dado un array con los días de la semana obtén el primer día que empieza por 'M'

findIndex

Como *find* pero en vez de devolver el elemento devuelve su posición (o -1 si nadie cumple la condición). En el ejemplo anterior el valor devuelto sería 0 (ya que el primer elemento cumple la condición). Al igual que el anterior tiene más sentido con arrays de objetos.

EJERCICIO: Dado un array con los días de la semana obtén la posición en el array del primer día que empieza por 'M'

every / some

La primera devuelve **true** si **TODOS** los elementos del array cumplen la condición y **false** en caso contrario. La segunda devuelve **true** si **ALGÚN** elemento del array cumple la condición. Ejemplo:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let todosAprobados = arrayNotas.every(nota => nota >= 5) // false
let algunAprobado = arrayNotas.some(nota => nota >= 5) // true
```

map

Permite modificar cada elemento de un array y devuelve un nuevo array con los elementos del original modificados. Ejemplo: queremos subir un 10% cada nota:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let arrayNotasSubidas = arrayNotas.map(nota => nota + nota * 10%)
```

EJERCICIO: Dado un array con los días de la semana devuelve otro array con los días en mayúsculas

reduce

Devuelve un valor calculado a partir de los elementos del array. En este caso la función recibe como primer parámetro el valor calculado hasta ahora y el método tiene como 1º parámetro la función y como 2º parámetro al valor calculado inicial (si no se indica será el primer elemento del array).

Ejemplo: queremos obtener la suma de las notas:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let sumaNotas = arrayNotas.reduce((total,nota) => total + = nota, 0) //
total = 35.35
// podríamos haber omitido el valor inicial 0 para total
let sumaNotas = arrayNotas.reduce((total,nota) => total + = nota) //
total = 35.35
```

Ejemplo: queremos obtener la nota más alta:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let maxNota = arrayNotas.reduce((max,nota) => nota > max ? nota : max) //
max = 9.75
```

En el siguiente ejemplo gráfico tenemos un "array" de verduras al que le aplicamos una función *map* para que las corte y al resultado le aplicamos un *reduce* para que obtenga un valor (el sandwich) con todas ellas:

WEB API

Acronimo ingles que hace referencia a Application Program Interface, o, Interfaz de Aplicaciones del Programa. No es más que una interfaz, o herramienta, que hace de intermediario entre un sistema y otro.

Que dos sistemas intervienen en este caso y une este Web API, por un lado, el motor de interpretación de código del navegador y por el otro lado el propio lenguaje JavaScript.

Para entenderlo podemos pensar que un API, en este caso, Web API, es como un traductor en una conferencia internacional. Es decir, JavaScript se intenta comunicar con el motor del navegador, pero el motor del navegador no lo entiende por qué no sabe que quiere decir cuando dice JavaScript que quiere recargar una web.

Entonces, Web API interviene para poder mediante su propia mediación recibir la orden desde JavaScript y decir al motor del navegador que es lo quiere conseguir.

Para ello, Web API tiene un montón de Interfaces que ofrecer a JavaScript para que se comunique fácil y eficientemente con el navegador que toque, pero dada la enorme cantidad y la dificultad, por la falta de tiempo, nos centramos en 4 elementos de la API de todas las que nos ofrece Web API:

- Window: representa una ventana del navegador que contiene un DOM.
- Document: Representa el propio DOM que se encuentra en una ventana del navegador.
- Event: representa los eventos que ocurren en ese DOM.
- Element: representa un nodo (elemento) del DOM.

Todas las APIs se pueden consultar en la dirección web de la WEB API:

<https://developer.mozilla.org/es/docs/Web/API>

Por ejemplo, si queremos ver distintas maneras de usar la WEB API de Window podemos probar los siguientes comandos por consola:

Instrucción	Resultado
API window	
<code>window.console.log("Hola");</code>	Muestra en la consola el mensaje "hola"
<code>window.alert("Hola...");</code>	Muestra una ventana modal con el mensaje "hola"
<code>window.print();</code>	Muestra el menu "Imprimir" del navegador.
<code>window.scrollTo(0, 100);</code>	Baja 100 pixeles verticalmente.
API document	
<code>document.querySelector(".o3j99.n1xJcf");</code>	Selecciona un objeto del DOM
<code>document.querySelector(".o3j99.n1xJcf").remove();</code>	Borra un objeto del DOM
<code>document.querySelectorAll("input");</code>	Selecciona todos los elementos del DOM de tipo input.
<code>document.getSelection();</code>	Selecciona el elemento del DOM que el usuario seleccione.

<code>document.location</code>	Propiedad que nos devuelve un objeto con muchos datos que nos muestran, entre otros, los datos del host, protocolos, href, puerto, origin, servidor, hash, etc...
<code>document.title</code>	Propiedad que nos devuelve el valor de la etiqueta title que se define dentro de la etiqueta Head.
<code>document.cookie</code>	Propiedad que devuelve las cookies del documento.
RECORDATORIO: UN ID se selecciona con el símbolo hashtag.	#
RECORDATORIO: UNA CLASE se selecciona con el punto.	.
<code>document.querySelector("#SivCob");</code>	Nos devuelve un objeto que es un elemento del DOM.
<code>document.querySelector("#SivCob").innerHTML;</code>	Nos devuelve el código HTML incluido dentro de un elemento del DOM.
<code>document.querySelector("#SivCob").innerText;</code>	Nos devuelve el texto incluido dentro de un elemento del DOM.

REACT