

Módulo 1. El lenguaje de programación Javascript

Introducción

En este módulo enseñaremos las bases del lenguaje de programación JavaScript, centrándonos en su uso en aplicaciones *web frontend*. Abordaremos temas como variables, constantes, condicionales, ciclos, funciones y la creación de objetos. Además, exploraremos la interacción de JavaScript con HTML a través del DOM y el manejo de estructuras de datos basadas en *arrays* de objetos.

Utilizaremos funciones de orden superior para manipular información clave en aplicaciones web y se adquirirá un manejo óptimo de las principales características de JavaScript. Así nos prepararemos para utilizar *frameworks* y librerías en el desarrollo de aplicaciones.

Video de inmersión

Unidad 1: Las bases del lenguaje JavaScript

El lenguaje JavaScript se desarrolló en 1996 como un lenguaje de *scripting* para generar partes dinámicas en páginas web estáticas. Netscape, la empresa detrás del navegador web homónimo, estableció JavaScript como un estándar a través de Ecma International. Con el tiempo, JavaScript evolucionó y se convirtió en el lenguaje principal para programar aplicaciones web dinámicas.

A diferencia de los sitios web estáticos, las aplicaciones web son interactivas, pueden requerir registro y tienen funcionalidades específicas. JavaScript (o JS) también facilitó la interacción con HTML, CSS y aplicaciones de servidor para obtener y almacenar datos.

Alcances del lenguaje JS

Si bien JavaScript se hizo conocido por permitirnos crear aplicaciones web dinámicas, actualmente es un lenguaje que extendió su funcionalidad a un montón de otros campos en el mundo de la programación. Por ello, hoy encontramos a JavaScript como un lenguaje de programación que nos permite crear aplicaciones de diferente índole.

Tabla 1: Segmentos

Segmento	Descripción
Frontend web	Podemos crear aplicaciones web <i>frontend</i> (ejecutables en un navegador web), mediante el uso de JavaScript en combinación con HTML5 y CSS. También podemos complementar esto al integrar <i>frameworks JS</i> como Angular o Vue JS, o una librería como lo es React JS.
Backend	El entorno de ejecución Node JS, y <i>frameworks</i> adicionales como Express y Passport JS, permiten crear aplicaciones <i>backend</i> . Además, librerías como Mongoose o Sequelize nos permiten interactuar desde una aplicación <i>backend</i> con diferentes bases de datos.
Mobile	En el terreno <i>mobile</i> , JavaScript y React JS se complementan muy bien con React Native. Esta plataforma nos permite crear aplicaciones móviles para dispositivos Android, iOS y iPad OS. Siempre, mediante el uso del código base construido con HTML5, CSS y JavaScript.
Nativo	React Native también permite crear aplicaciones nativas con el mismo código base utilizado para aplicaciones móviles. Por lo tanto, podemos crear <i>apps</i> instalables en Windows, Linux y Mac OS.
IoT	El <i>framework</i> Johnny-Five es utilizado por JavaScript para escribir aplicaciones que comanden sensores electrónicos que controlen luces, regadores automáticos, temperatura y humedad, medidores de distancia ultrasónicos, detector de lúmenes, entre otros tantos sensores y actuadores existentes en el mercado denominado como 'internet de las cosas' (IoT).
PWA	PWA (<i>progressive web apps</i>) son aplicaciones web <i>frontend</i> creadas con JavaScript, que poseen una característica que les permite ser instaladas en un dispositivo móvil o computadora, sin ser una aplicación nativa, y funcionar tanto con acceso a internet como también de manera <i>offline</i> .

Fuente: elaboración propia.

Como podemos apreciar, JavaScript no se centra solamente en sitios o aplicaciones web. También está disponible en otros tantos segmentos, lo que hace extensivas las capacidades de este lenguaje de programación a muchos otros nichos. Por lo tanto, aprender sus bases nos abre las puertas de poder llevarlo a otros terrenos casi sin esfuerzo adicional.

Funcionamiento de JavaScript

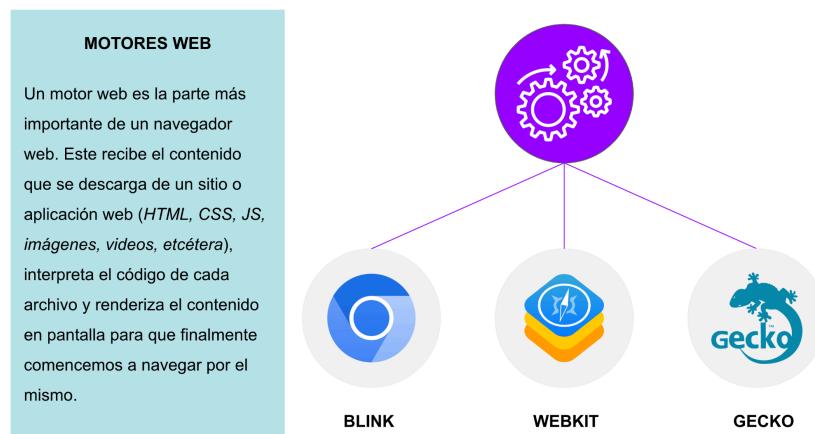
Al ser un lenguaje de *scripting*, JS depende de un intérprete para que su lógica pueda ser ejecutada. En el caso de JS orientado a aplicaciones web *frontend*, cada navegador web cuenta con un motor. Este motor es quien se encarga de recibir el contenido de un sitio web o de una aplicación web e interpretarlo, para luego renderizar su interfaz en una pestaña del navegador web.

Cada *web browser* cuenta con un motor web, y a su vez, este motor se subdivide internamente en dos intérpretes: HTML y CSS, por un lado, y JavaScript por otro.

Motores web

Si bien existen muchos navegadores web en el mercado comercial de internet, solamente existen tres motores que son los principales motores de renderizado, y que les dan vida a los principales navegadores web como también a los navegadores web secundarios.

Figura 1: Motores web



Fuente: [Imagen sin título sobre motores]. (s.f.).

Cada motor web mencionado en la figura 1, cuenta con dos intérpretes de código: uno dedicado a HTML y CSS, y otro intérprete dedicado a JavaScript. A su vez, cada intérprete de JS posee un nombre descriptivo interno.

Para Blink, el intérprete de JS se llama **V8**. Para WebKit, el intérprete de JS se llama **JavaScript Core**, y el intérprete JS en Gecko se llama **Spider Monkey**.

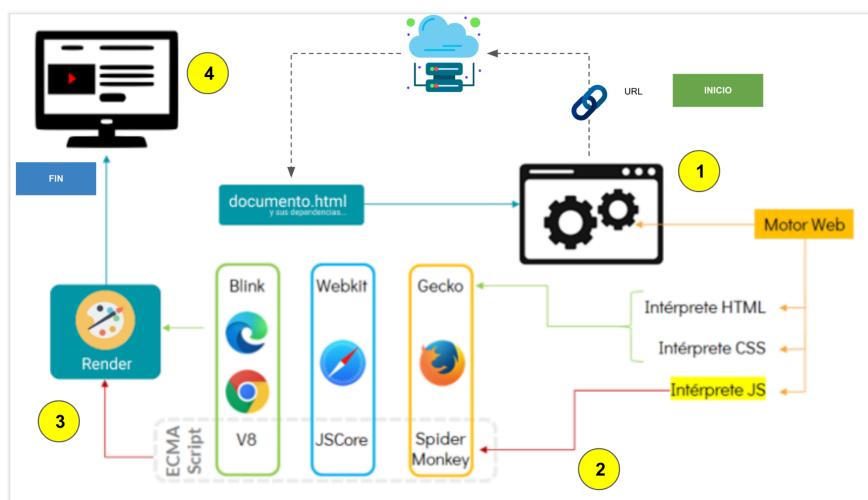
Es importante destacar que cada motor web evoluciona por separado, e integra las nuevas características de HTML, CSS y JavaScript a destiempo. Esto debemos tenerlo presente porque no siempre todo código nuevo o moderno de cualquiera de estos tres lenguajes pilares de la web, funcionará correctamente en cada uno de los navegadores web.

A su vez, el resto de los navegadores web del mercado (Opera, Samsung Internet, Konqueror, Epiphany browser, Maxthon, Brave, Vivaldi, etcétera) utilizan alguna versión de estos motores web, aunque siempre más antigua en el tiempo, por lo tanto, cuentan con soporte de menos características modernas que podemos encontrar integradas en los principales navegadores web.

Qué sucede cuando ingresamos a un sitio web

Veamos en la figura 2 qué es lo que sucede en un navegador web, de la mano de los motores de renderizado, cada vez que accedemos a un sitio o aplicación web.

Figura 2: Motor web



Fuente: elaboración propia con base en Luna, 2021.

1. Abrimos el *web browser* e ingresamos una URL (o dirección web) para navegar. Este se ocupa de rastrear el servidor web correspondiente y descargar el archivo HTML principal del sitio web en cuestión. Junto con este archivo HTML, vienen

referenciados el resto de los recursos que componen la aplicación o sitio web (imágenes, videos, textos, archivos CSS, archivos JS, etcétera).

2. Toda esta información es recibida por el motor web o motor de renderizado. Se separa cada uno de los recursos asociados a la web que accedemos para que el motor interprete cada línea de código. HTML, CSS y las imágenes y videos, son interpretadas por el motor de renderizado de HTML y CSS. Los archivos JavaScript son derivados al intérprete dedicado a entender este lenguaje.
3. Procesados todos los datos e intercambiadas las partes de interacción entre JS y HTML o CSS, todo este contenido llega a la etapa de renderizado. Aquí es cuando el mismo se traduce en el estilo gráfico que se mostrará en el navegador web.
4. Finalmente, todo el contenido del sitio o aplicación web es visualizado en el navegador web, y ya podemos comenzar a interactuar con este.

Este es el proceso que ocurre dentro de un navegador web cada vez que accedemos a algún sitio o aplicación web. Internamente, el intérprete de JS ‘habla’ con el intérprete de HTML y CSS cuando JavaScript debe modificar, agregar o eliminar algo directamente relacionado con la interfaz de usuario cargada en el navegador web.

Tema 1: Fundamentos del lenguaje JS

Los fundamentos del lenguaje JavaScript son esenciales para todo desarrollador de aplicaciones, tanto web como cualquier otro segmento de mercado que JS abarca. Para poder considerar su lógica, exploraremos los conceptos de sintaxis básica de variables y constantes, y los ciclos de análisis condicionales y de iteración que el lenguaje posee.

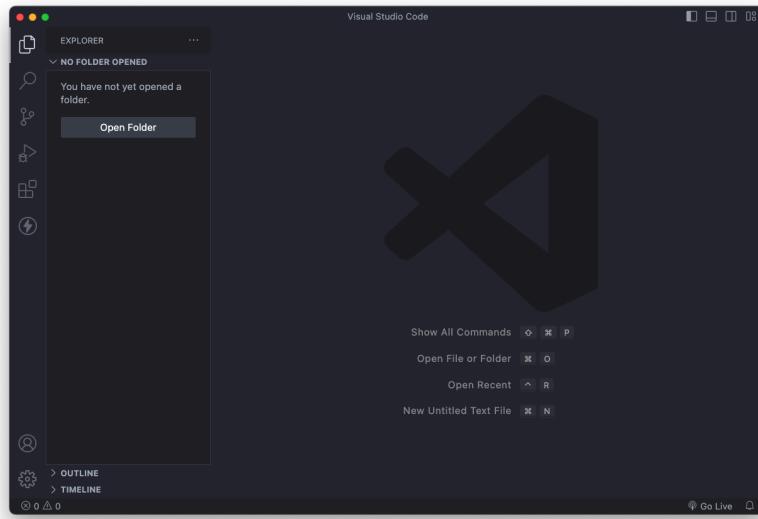
También veremos las funciones JS para definir pequeñas tareas simples, y el uso de objetos para mejorar las estructuras de código, sin tener que repetir innecesariamente bloques de este.

Finalizada esto nos zambulliremos en la parte más interesante del lenguaje JS: la interacción con HTML5 y CSS. Veamos entonces qué nos ofrece este fabuloso lenguaje de programación.

Editor de código

El desarrollo de todos los ejemplos de código de estas unidades, los realizaremos con Visual Studio Code (VS Code). Este editor, propiedad de Microsoft, es el editor más popular desde el año 2018. VS Code posee soporte nativo para HTML5, CSS y JavaScript además de cualquier otro lenguaje de programación.

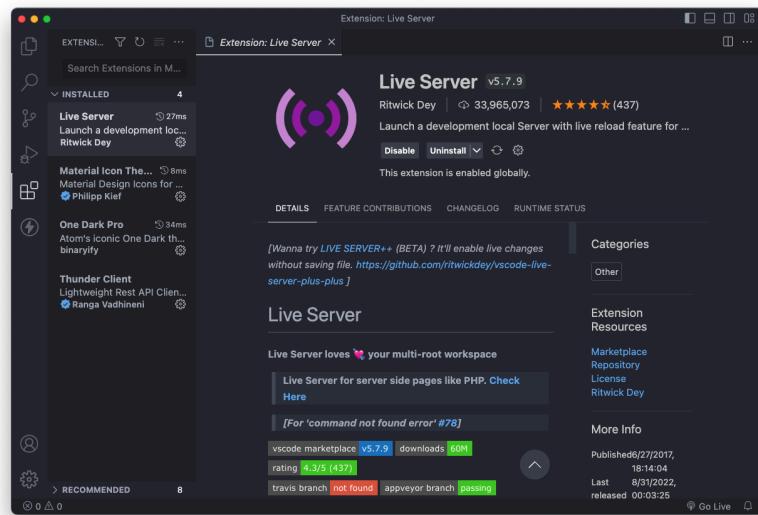
Figura 3: Editor de código Visual Studio Code



Fuente: captura de pantalla de VS Code, s.f.

Mediante el *marketplace* de extensiones, podemos incluir más funcionalidades en este editor, a medida que necesitemos hacerlo.

Figura 4: Extensión Live Server



Fuente: captura de pantalla de VS Code, s.f.

Actividad 1

Ingresar a <https://code.visualstudio.com/> y descargar el programa de instalación de VS Code.

Ejecutar el mismo y seguir los pasos del asistente de instalación.

Finalizado el asistente de instalación, ejecutar Visual Studio Code.

Acceder al apartado 'Extensiones' (*Extensions*), y buscar e instalar Live Server.

Para conocer la forma correcta de instalación, descargá el PDF al final de la lectura con la resolución.

Iniciar un proyecto web con Visual Studio Code

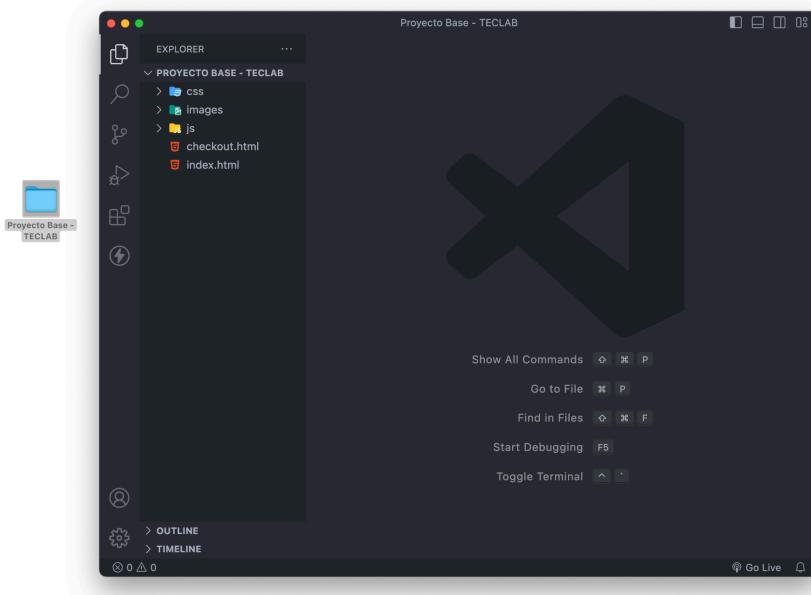
Descargaremos, a continuación, el código fuente de un proyecto base web, correspondiente a la estructura de un futuro *ecommerce*.



Utilizaremos esto como proyecto base a lo largo de estas unidades, para integrar las diferentes características de JavaScript en ejemplos prácticos.

Descomprimiremos el archivo .ZIP y dejaremos la carpeta de proyecto a mano. Acto seguido, abrimos VS Code y arrastramos esta carpeta a nuestra aplicación. Una vez abierto el proyecto, veremos en el apartado **explorer** (explorador), la estructura de carpetas y archivos que lo componen.

Figura 5: Visual Studio Code y la estructura de nuestro proyecto base



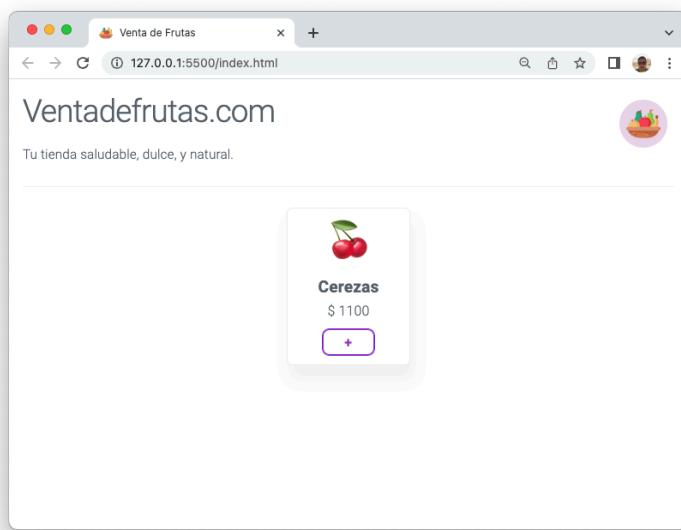
Fuente: captura de pantalla de VS Code, s.f.

Estructura del proyecto

Encontramos, dentro del proyecto, una estructura simple: dos documentos HTML, una subcarpeta llamada **CSS** que aloja un archivo CSS para estilizar nuestra aplicación web, y una subcarpeta llamada **JS**, que ya contiene un archivo JavaScript que utilizaremos más adelante. Además, una subcarpeta **images** donde se almacenan imágenes e íconos que se utilizan dentro del proyecto web.

Seleccionemos el archivo **index.html** para que se abra en el editor de código. Luego podemos pulsar el botón **Go Live**, ubicado en el extremo inferior derecho de VS Code, para iniciar **Live Server** y así cargar este proyecto en el navegador web predeterminado.

Figura 6: Proyecto web base en ejecución dentro del navegador web



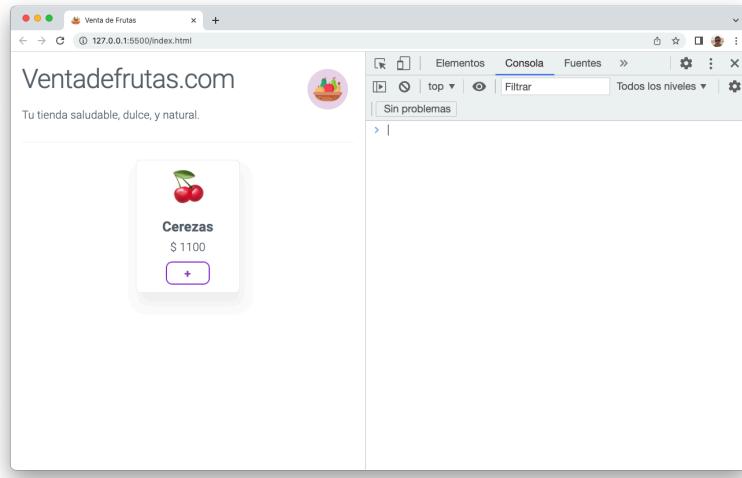
Fuente: elaboración propia.

Por el momento no realizaremos ninguna interacción. Solo probamos Visual Studio Code y Live Server para validar que funcionen correctamente en nuestra computadora.

Si tenemos algún AdBlocker instalado, recomendamos deshabilitarlo para que no interfiera en la ejecución de Live Server.

Como trabajaremos de forma exhaustiva con JavaScript, activaremos las '**herramientas para desarrollador**' integradas en todo navegador web. Los ejemplos que abordaremos se realizarán sobre Google Chrome, por lo tanto, recomendamos utilizar el mismo navegador web para las pruebas, así no hay conflictos entre lo que haremos y la presente lectura.

Figura 7: Google Chrome y DevTools en ejecución



Fuente: elaboración propia.

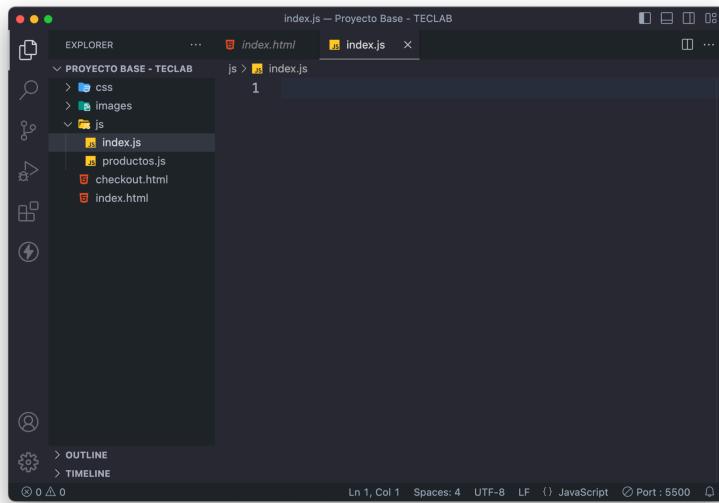
Parados en el navegador web, con la tecla **F12** activamos las ‘herramientas para desarrollador’, también nombradas como **DevTools** o *developer tools*. En la pestaña ‘Console’ veremos la ejecución del código JavaScript que desarrollemos. También, todos aquellos errores que generemos de forma accidental en nuestra lógica de código, serán notificados en este apartado dentro de **DevTools**. Todas las pruebas que realicemos mientras programamos, deben ejecutarse sobre **DevTools**.

Archivos JavaScript

Dentro de nuestro proyecto web base, crearemos nuestro primer archivo JavaScript para comenzar a interactuar con el código de este lenguaje de programación. Para ello, en Visual Studio Code, nos paramos en la subcarpeta **JS**, y hacemos clic con el botón secundario del *mouse* sobre esta subcarpeta.

Desde el menú contextual que se despliega, seleccionamos el punto de menú denominado ‘*new file*’. Escribimos **index.js** y pulsamos la tecla **Enter** para que se cree dicho archivo. El mismo será visible en el apartado **Explorer**, y se abrirá en el panel central de **Visual Studio Code**, por lo que quedará listo para comenzar a escribir código.

Figura 8: Visualización del archivo index.js que acabamos de crear



Fuente: captura de pantalla de VS Code, s.f.

Antes de comenzar a programar, referenciamos el archivo creado en el documento HTML. Para ello, editamos **index.html** y dentro de su apartado **<head>** declaramos la sintaxis de referencia de archivos JavaScript justo antes del cierre de este apartado **</head>**.

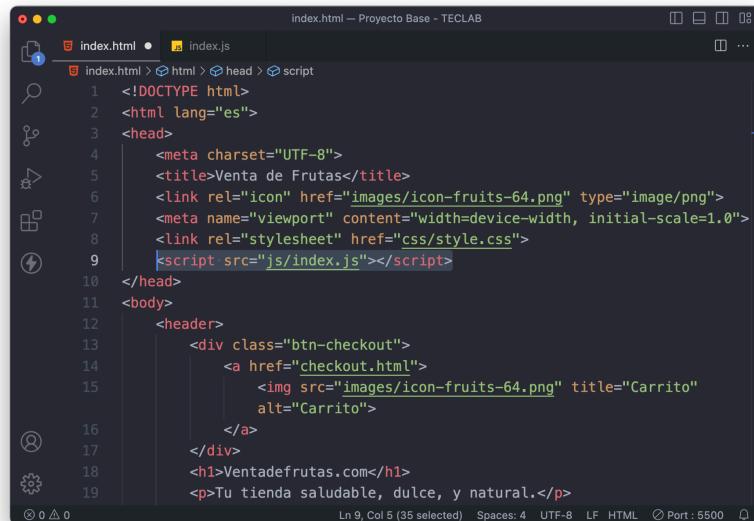
Figura 9. Código

A screenshot of a code editor showing a tooltip. The tooltip title is "Referenciar archivo JS en HTML". The content of the tooltip is a single line of HTML code: <script src="js/index.js"></script>. The background of the code editor is light gray, and the tooltip has a dark gray background with rounded corners.

Fuente: elaboración propia.

El tag HTML **<script>** nos permite referenciar recursos adicionales a un documento HTML; en este caso, un archivo JavaScript. En el atributo **src** indicamos la ruta relativa al archivo. En nuestro ejemplo, el mismo se almacena dentro de la subcarpeta **JS**. El resultado de la referencia de este archivo JS dentro del documento HTML debe ser similar a lo que se muestra a continuación.

Figura 10. Código



The screenshot shows a code editor window titled "index.html - Proyecto Base - TECLAB". The file content is an HTML document with the following structure:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Venta de Frutas</title>
    <link rel="icon" href="images/icon-fruits-64.png" type="image/png">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet" href="css/style.css">
    <script src="js/index.js"></script>
</head>
<body>
    <header>
        <div class="btn-checkout">
            <a href="checkout.html">
                
            </a>
        </div>
        <h1>Ventadefrutas.com</h1>
        <p>Tu tienda saludable, dulce, y natural.</p>
    </header>
</body>
```

At the bottom of the editor, status information includes "Ln 9, Col 5 (35 selected)", "Spaces: 4", "UTF-8", "LF", "HTML", and "Port: 5500".

Fuente: elaboración propia.

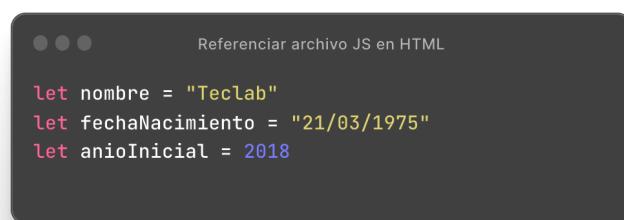
Con la inclusión del archivo JS, podemos interactuar con las bases de este lenguaje de programación para la web.

Tipado de datos

JavaScript es un lenguaje de programación dinámicamente tipado, lo que significa que no es necesario especificar el tipo de datos de una variable al declararla. Por lo tanto, una variable puede contener cualquier tipo de dato: números, cadenas de texto, booleanos, *arrays*, objetos, funciones, entre otros. Además, el tipo de datos puede cambiar en cualquier momento durante la ejecución del programa.

La forma de declarar variables en JavaScript, es utilizar la palabra reservada '**let**'. Seguida a esta, definimos el nombre de la variable, en lo posible con una sola palabra. Si debemos utilizar dos palabras, lo ideal es seguir la convención de datos del tipo '**camel case**'.

Figura 11. Código



Fuente: elaboración propia.

JavaScript es un lenguaje dinámicamente tipado, lo que significa que el tipo de datos de una variable puede cambiar durante la ejecución del programa. Esto se debe a que JavaScript determina automáticamente el tipo de datos en tiempo de ejecución. Aunque el dinamismo ofrece flexibilidad, es importante comprender los conceptos relacionados con el tipado de datos, como la conversión de tipos y las comprobaciones de tipos, para evitar operaciones inválidas y escribir un código más eficiente.

Aunque JavaScript permite cambios en los valores y tipos de las variables, recomendamos seguir las mejores prácticas y utilizar la palabra reservada ‘const’ para declarar aquellos datos que no cambiarán a lo largo de la aplicación, como funciones, objetos, *arrays* y enlaces con HTML mediante el uso de DOM.

Declaración de constantes

Cuando declaremos constantes, sabemos que sus datos no cambiarán. Para ello, utilizamos la palabra reservada ‘const’. Veamos a continuación un ejemplo.

Figura 12. Código



```
Referenciar archivo JS en HTML

const USUARIO = "Joe McMillian"
const SecurityID = '044b429d-911a-a5b1d9b7efbd'
```

Fuente: elaboración propia.

En la mayoría de los lenguajes de programación, la declaración de constantes se realiza al definir su nombre en mayúsculas. En JavaScript no es habitual hacer esto, al menos en la mayoría de los ejemplos abordados en su documentación oficial. Las constantes se declaran casi siempre en minúsculas, o mediante el uso del sistema ‘CamelCase’.

En un ambiente corporativo, los equipos de desarrollo son quienes deciden cómo declarar las mismas. Nosotros seguiremos la tendencia de uso de acuerdo a lo que nos informen.

Declaración de arrays

Los *arrays* son estructuras de datos que nos permiten guardar varios valores, usualmente de un mismo tipo, bajo una única estructura. Estos también suelen declararse como constantes.

Figura 13. Código



```
Referenciar archivo JS en HTML

const paisesDelSur = ['Argentina', 'Uruguay', 'Brasil', 'Venezuela', 'Chile']

const carrito = [{codigo: 123, nombre: 'Teclado Bluetooth', importe: 15500},
  {codigo: 234, nombre: 'Mouse Bluetooth', importe: 7800},
  {codigo: 345, nombre: 'SSD Portátil', importe: 47350}]
```

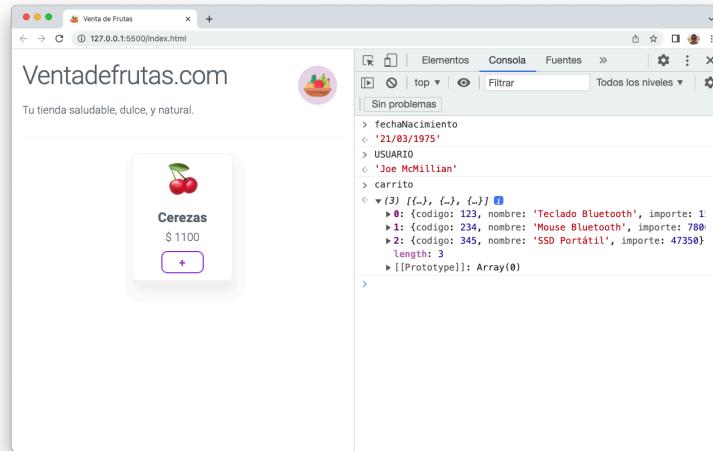
Fuente: elaboración propia.

Si bien un array puede crecer o decrecer, la cantidad de elementos que este almacena, este proceso lo realiza a través de métodos internos, por lo tanto, no será problema incrementar o eliminar parte de su contenido.

Depuración de código

Declaremos todos los ejemplos de código anteriores dentro de nuestro archivo JavaScript denominado **index.js**. Asegúremos de que el proyecto esté en ejecución y que DevTools esté abierto. En la consola JS de **DevTools**, escribimos cualquiera de los nombres de estas variables o constantes definidas y pulsamos ‘enter’ para ver el contenido de las mismas.

Figura 14. Código



Fuente: elaboración propia.

Otra forma de depurar el código que escribimos en JS, es utilizar el objeto ‘`console`’, nativo de este lenguaje de programación, y sus diferentes métodos.

Tabla 2: Método

Método	Descripción
console.log ('texto estático, o variable JS').	Muestra un texto estático o el valor de una variable, constante u objeto en la consola JS de DevTools.
console.warn ('mensaje estático con tono de advertencia').	Muestra un texto estático con una impronta de advertencia. Utiliza color amarillo y un ícono de advertencia apropiado.
console.error ('Houston, tenemos un problema').	Muestra un texto estático con una impronta de error. Utiliza un color rojo y un ícono de error apropiado.

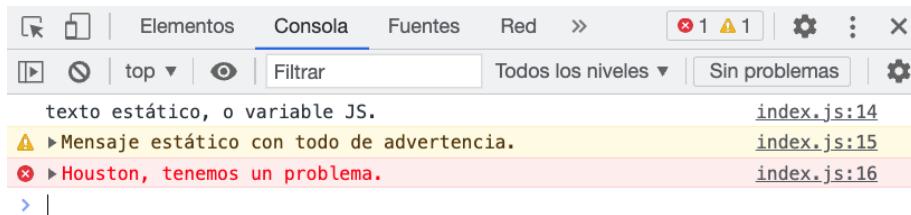
Fuente: elaboración propia.

Estos métodos incluidos en el objeto ‘`console`’ nos permiten escribir código JS y depurar el mismo en **DevTools > Console**, para validar el camino que nuestro código toma.

Los mensajes de advertencia o de error son claves para remarcar, entre muchos posibles mensajes de `log()` enviados a la consola JS, cuando nuestro código toma un camino, el cual, de acuerdo a la lógica de nuestra aplicación web, no debería tomar.

Los veremos en acción con el uso de condicionales y de ciclos de iteración.

Figura 15. Código



Fuente: elaboración propia.

Actividad 2

Ya declaramos o referenciamos un archivo JS dentro del proyecto que te compartimos anteriormente.

Ahora, definir dos variables: una de tipo numérico y otra de tipo *string*, y asignarle los valores que

consideramos óptimos.

Luego, definir una constante llamada '**arrayFrutas**'. La misma tendrá la estructura de elementos. Declarar en ella, en formato array de elementos, al menos 5 elementos que refieran a frutas.

Finalizados estos pasos, utilizar el objeto '*console*' para mostrar los valores de las variables y constante que acabamos de crear.

Definir un '*console.log*' para ver el valor almacenado en cada variable.

Definir '*console.table*' para visualizar los datos almacenados en el '*arrayFrutas*'.

Ejecutar el proyecto con Live Server y visualizar en DevTools > Console, el resultado de ejecución de los métodos del objeto '*console*' utilizados.

Para conocer la respuesta correcta, descargá el PDF al final de la lectura.

Tema 2: Condicionales y ciclos

Los condicionales son una estructura de control de flujo en JavaScript que permite ejecutar diferentes bloques de código según se cumplan ciertas condiciones. Los ciclos de iteración son estructuras de control de flujo que permiten repetir bloques de código varias veces. Veamos a continuación algunos ejemplos.

Condicionales en JS

El condicional '*if*' permite evaluar una expresión booleana y ejecuta un bloque de código si la expresión es verdadera. Aquí, un ejemplo de uso.

Figura 16. Código



```
let anioInicial = 2017

if (anioInicial === 2017) {
    console.log('La variable anioInicial tiene como valor el año 2017.')
}
```

Fuente: elaboración propia.

Dentro de los paréntesis del bloque **if()** evaluamos si la variable '**anioIncial**' posee el valor **2017**. En el caso de cumplirse la evaluación de esta expresión, la misma retornará un valor del tipo

'true', por lo tanto, se ejecutará la línea de código definida dentro del bloque de ejecución conformado por las llaves {} y }. También existen los condicionales 'else' y 'else if', que se utilizan para ejecutar diferentes bloques de código según el resultado de varias expresiones booleanas.

Figura 17. Código



```
Referenciar archivo JS en HTML
let anioInicial = 2017

if (anioInicial === 2023) {
    console.log('La variable anioInicial tiene como valor el año 2017.')
} else {
    console.warn('El valor de anioInicial no es el esperado.')
}
```

Fuente: elaboración propia.

En este último ejemplo de código vemos que la expresión 'if' espera como valor, el número 2023. Al no cumplirse esta condición, se ejecuta el bloque de código 'else', el cual tiene otro mensaje a ejecutar en la **consola JS**. El método **warn()** permite darle una impronta diferente cuando nuestro código va por un camino usualmente no esperado.

Cuando comparamos una variable o constante con un valor específico dentro de un bloque condicional, debemos utilizar siempre el triple signo igual '===' . El uso de este nos permite comparar el valor de la variable o constante y su tipo de datos.

Además de comparar con la utilización del operador igual, también podemos realizar comparaciones con los otros operadores de comparación existentes.

Tabla 3: Operadores

Operadores de comparación	
Símbolo	Descripción
>	Mayor a
<	Menor a
>=	Mayor o igual a
<=	Menor o igual a
!=	Distinto de
!==	Estrictamente distinto de

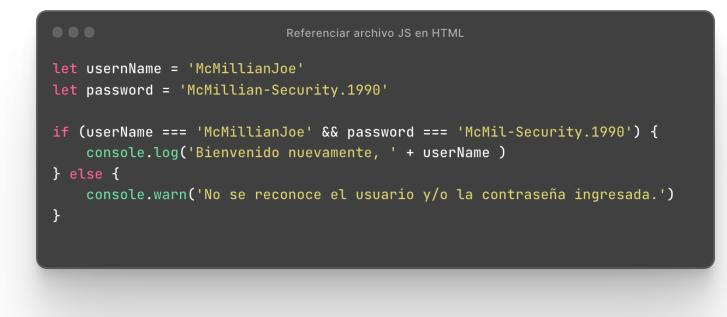
Fuente: elaboración propia.

Tabla 4: Operadores

Operadores lógicos	
Operador	Descripción
&&	Operador lógico AND
	Operador lógico OR
!	Operador lógico NOT

Fuente: elaboración propia.

Figura 18. Código



```

let userName = 'McMillianJoe'
let password = 'McMillian-Security.1990'

if (userName === 'McMillianJoe' && password === 'McMil-Security.1990') {
    console.log('Bienvenido nuevamente, ' + userName )
} else {
    console.warn('No se reconoce el usuario y/o la contraseña ingresada.')
}

```

Fuente: elaboración propia.

La cláusula 'switch'

La cláusula ***switch*** es una estructura de control condicional en JavaScript que se utiliza para evaluar una expresión y ejecutar diferentes bloques de código, según el valor de esa expresión. Su estructura se muestra a continuación.

Figura 19. Código



```
let valorOfertado = 2500

switch (valorOfertado) {
    case 1000:
        console.warn("Su oferta es muy baja. Realiza una mejor oferta.")
        break;
    case 2000:
        console.log("Gracias por tu oferta. Puedes retirar el producto hoy mismo.")
        break;
    default:
        console.error("No pudimos interpretar tu oferta. Intenta nuevamente.")
}
```

Fuente: elaboración propia.

Primero evaluamos la expresión especificada en la cláusula ***switch***. Luego, comparamos el valor de la expresión con cada caso (***case***) especificado en el bloque de código. Si encontramos un caso que coincide con el valor de la expresión, ejecutamos el bloque de código correspondiente a dicho caso. Si no coincide en ningún caso, ejecutamos el bloque de código especificado en la cláusula ***default***.

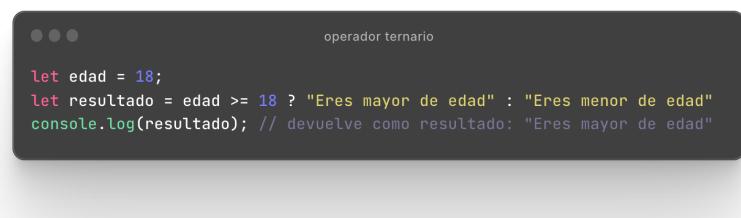
Debemos considerar siempre que cada bloque de código ***case*** debe finalizar con una declaración ***break***. Así evitamos que se ejecuten los bloques de código siguientes al caso, una vez que encontramos coincidencia. Si no incluimos la declaración ***break***, se ejecutarán todos los bloques de código siguientes al caso que coincide.

El uso de ***switch*** es poco visto en aplicaciones web reales, dado que la comparación de cada caso nos obliga a que sea una comparación estricta. No podemos aplicar casos mediante la comparación 'si es mayor que' o 'si es menor o igual que'. Esta falencia torna a la sentencia ***switch***, poco útil en aplicaciones JavaScript dinámicas.

Operador ternario

El operador ternario es una estructura de control de flujo utilizada para tomar decisiones simples en una sola línea de código. Actualmente se lo utiliza para simplificar el manejo de condicionales ***if*** - ***else***, cuando estos pueden resolverse de manera simple. La sintaxis del operador ternario se muestra a continuación.

Figura 20. Código



Fuente: elaboración propia.

En este ejemplo, la condición es **edad >= 18**, que se evalúa como verdadera, dado que la variable tiene como valor el número **18**. Por lo tanto, se asigna el valor ‘eres mayor de edad’ a la variable **resultado**. Sumado al retorno implícito del operador ternario, podremos atrapar cualquier dato que este retorne, en una variable o constante.

Ciclos de iteración

Los ciclos de iteración son estructuras de control de flujo que permiten repetir bloques de código varias veces. Actualmente JavaScript cuenta con tres ciclos de iteración convencionales, los cuales se separan en dos segmentos:

- **ciclo de iteración por conteo, y**
- **ciclos de iteración por repetición.**

• El ciclo ‘for’

En JavaScript, el ciclo más común es el **for**, que se utiliza para repetir un bloque de código un número específico de veces. Este ciclo entra en el segmento de ciclo de iteración por conteo.

Su estructura principal está compuesta por paréntesis, los cuales reciben tres parámetros, **valor inicial**, **valor límite**, **valor incremental**. Luego, definimos el código a iterar repetidas veces, dentro de las llaves de bloque.

1. El valor inicial es representado comúnmente por una variable inicializada con un valor numérico. El mismo suele ser el 0 (cero).
2. El valor límite hace referencia a cuál será el valor numérico máximo que condicione las repeticiones del bloque de código interno del ciclo **for**.

3. El valor incremental aumenta en un dígito el valor inicial, cada vez que el ciclo **for** es iterado (cuando se ejecuta el bloque de código interno).

A continuación, un ejemplo aplicado mediante el uso de un *array* denominado **paisesDelSur**.

Figura 21. Código

```
... Ciclo for
const paisesDelSur = ['Argentina', 'Uruguay', 'Brasil', 'Venezuela', 'Chile']
// índice del array: 0 1 2 3 4
```

Fuente: elaboración propia.

Mediante la propiedad '**.length**' (longitud), obtenemos cuántos ciclos de iteración contendrá este ciclo **for**. Cada elemento es accedido a través del índice que asume cada uno de ellos. Entonces, para iterar el *array*, definimos una variable **i** inicializada en **0**, como primer parámetro. El segundo parámetro será comparar 'mientras **i** sea menor al total de elementos del array' y, finalmente, el tercer parámetro será **i++** el cual incrementa el valor de **i** en un dígito.

Figura 22. Código

```
... Ciclo for
const paisesDelSur = ['Argentina', 'Uruguay', 'Brasil', 'Venezuela', 'Chile']
for (let i = 0; i < paisesDelSur.length; i++) {
    console.log(paisesDelSur[i])
}
```

Fuente: elaboración propia.

De esta forma, veremos los nombres de todos los países del *array* impresos en la consola JS. Para imprimir cada valor del *array*, escribimos el mismo con la variable **i** encerrada entre corchetes, para que esta nos permita acceder al valor de dicha posición.

- **El ciclo while**

While repite un bloque de código si este cumple una expresión determinada. Dentro del mismo definimos el bloque de código a ejecutar y también el control de cambio de la variable booleana.

Figura 23. Código



Ciclo while

```
let contador = 0

while (contador < 5) {
    console.log("El contador es " + contador)
    contador++
}
```

Fuente: elaboración propia.

En este ejemplo de código, realizamos un conteo de 0 a 4 mediante el uso de la variable **contador** inicializada en **0** (cero). Al ejecutar el ciclo **while**, evaluamos como expresión si dicha variable tiene un valor menor a 5. Como es afirmativo, ejecutamos el bloque de código asociado al ciclo **while**. En el bloque de código, incrementamos el valor de la variable **contador** en un dígito, y utilizamos **contador++**.

En **DevTools > Console** veremos impreso el mensaje de **console.log** con cada uno de los valores que asume la variable ‘**contador**’.

- **El ciclo do-while**

Este otro ciclo realiza la comparación booleana al final de su estructura.

Figura 24. Código



Ciclo do-while

```
let contador = 0;
do {
    console.log("El contador es " + contador);
    contador++;
} while (contador < 5);
```

Fuente: elaboración propia.

La diferencia de **do-while** con el ciclo **while** es que, en este último, el bloque de código se ejecuta al menos una vez antes de verificar la condición. Es decir, incluso si la condición es falsa desde el principio, el bloque de código se ejecuta al menos una vez. En el caso de **while**, si la variable ‘contador’ se inicializa con el valor 5 o superior, el bloque de código **while** nunca se ejecutará.

Salteo de iteración

En cualquiera de los tres ciclos de iteración, si tuviésemos la necesidad de saltar una iteración por cualquiera sea el motivo, podemos realizar esto al integrar la palabra reservada **continue**, previo a la ejecución del bloque de código en ejecución.

Interrumpir la iteración

De igual forma, en el caso de que necesitemos interrumpir la iteración de un ciclo en ejecución, podemos definir la palabra reservada **break**, en el apartado donde necesitemos interrumpir el bloque de código.

Tema 3: Funciones JS

Las funciones en JavaScript se estructuran casi de forma similar a como se utilizan en el lenguaje PHP. Pueden ser funciones simples, con parámetros, y hasta retornar un valor de salida.

Funciones simples o convencionales

Estas son funciones que no tienen mayor complejidad. Se definen, se les agrega la lógica de la tarea que deben cumplimentar, y listo. Luego están disponibles para ser invocadas cuantas veces necesitemos.

Funciones con parámetros

Las funciones con parámetros permiten recibir entre los paréntesis de su estructura, uno o más valores como parámetros. Estos valores son usualmente utilizados en el cuerpo de esta función, para realizar operaciones, cálculos, o tareas específicas.

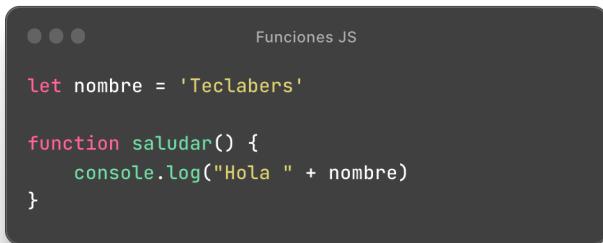
Funciones con retorno

Este tipo de función realiza una tarea específica al retornar algún valor resultante como dato. Este valor que retorna, podemos capturarlo en una variable o constante, y utilizarlo en otras partes de nuestra aplicación. Las funciones con retorno pueden basarse en una estructura de función simple, como también pueden recibir uno o más parámetros para utilizar sus valores internamente.

Definir una función

Para definirlas, su nombre puede ser simple o compuesto, y se sigue la convención mencionada anteriormente para las variables y constantes, aunque, a diferencia de estas, una función debe representar siempre, a través de su nombre, una acción imperativa.

Figura 25. Código



```
● ● ● Funciones JS

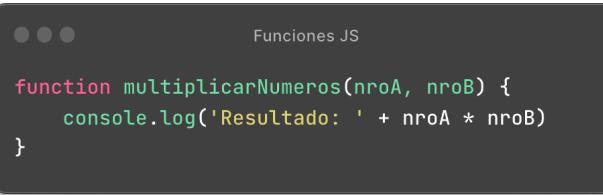
let nombre = 'Teclabers'

function saludar() {
    console.log("Hola " + nombre)
}
```

Fuente: elaboración propia.

La función '**'saludar()'**' es una función simple, que contiene un bloque de código como acción, el cual imprime un mensaje en la consola JS, combina un texto estático y el valor de la variable 'nombre'.

Figura 26. Código



```
● ● ● Funciones JS

function multiplicarNumeros(nroA, nroB) {
    console.log('Resultado: ' + nroA * nroB)
}
```

Fuente: elaboración propia.

En este otro ejemplo, la función '**'multiplicarNumeros()'** recibe dos parámetros. Toma los valores de estos parámetros (los cuales son equivalentes a dos variables), y los utiliza para llevar a cabo una multiplicación. Finalmente, imprime el resultado en la consola JS.

Figura 27. Código

```
function dividirNumeros(nroA, nroB) {
    return nroA / nroB
}
```

Fuente: elaboración propia.

En este último ejemplo, combinamos una función con parámetros y retorno. La misma toma en su lógica los valores que recibirá como parámetros, los divide entre sí, y retorna el resultado a través de la misma función, mediante la palabra reservada **return**. Este retorno podrá ser capturado en una constante o variable, o en otro elemento de nuestra lógica.

Llamar a una función

Para llamar a cualquiera de estas funciones representadas anteriormente, escribimos el nombre de la misma en cualquier parte de nuestra aplicación web. Recordemos agregar los parámetros y definir parámetros si es que así lo requiere la función.

Figura 28. Código

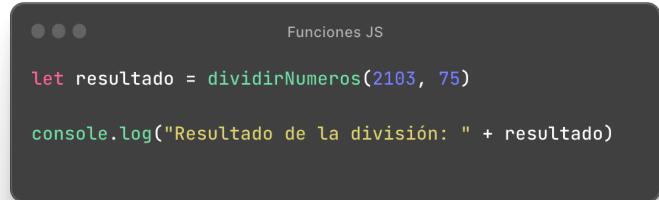
```
function dividirNumeros(nroA, nroB) {
    return nroA / nroB
}

//llamar a la función
dividirNumeros(2103, 75)
```

Fuente: elaboración propia.

Al ser una función con retorno, podemos capturar su resultado en una variable como se muestra a continuación.

Figura 29. Código



```
● ● ● Funciones JS
let resultado = dividirNumeros(2103, 75)
console.log("Resultado de la división: " + resultado)
```

Fuente: elaboración propia.

Como podemos apreciar hasta aquí, las funciones son una herramienta importante para organizar nuestro código y reutilizarlo en diferentes partes de una aplicación.

Actividad 3

En la aplicación de ejemplo que trabajamos, crear una función llamada **iterarArrayFrutas()** en la cual utilizaremos el ciclo **for** para recorrer **arrayFrutas**.

Investigar en la ayuda oficial de MDN o W3Schools correspondiente al lenguaje JS, cómo utilizar la cláusula **continue**.

Iterar el *array* de frutas con el ciclo *for*, mediante el envío de un mensaje a la consola JS con el valor de cada elemento iterado. Agregar una condición dentro del ciclo *for* que valide si el índice del *array* que encontramos en iteración es 1 o 3, ejecute la cláusula *continue*. De esta forma evitaremos escribir el nombre de las frutas que se encuentren en estas posiciones del *array*.

Para conocer la respuesta correcta, descargá el PDF al final de la lectura.

Arrow functions

Las funciones flecha, conocidas también como **arrow functions**, son una forma diferente de escribir funciones en JS, pero con el mismo propósito de las funciones convencionales. Este tipo de funciones fueron incluidas en el lenguaje JS desde 2015 con el lanzamiento de la versión 6 del lenguaje JavaScript (ES6).

Las funciones flechas cuentan con algunas características técnicas que las diferencian de las funciones convencionales. Veamos un ejemplo de ellas, a continuación.

Figura 30. Código

```
const saludar = ()=> {
    console.log("Hola Teclabers")
}
```

Fuente: elaboración propia.

En este ejemplo de *arrow function*, vemos que la función ‘saludar’ estructura su nombre a partir de una constante. Luego definimos sus paréntesis seguidos de un **signo igual y el símbolo mayor que**. De este apartado es de donde proviene su nombre ‘función flecha’.

Figura 31. Código

```
let nombre = 'Teclabers'

const saludar = (param)=> {
    console.log("Hola " + param)
}
```

Fuente: elaboración propia.

Su estructura es similar, al definir el parámetro dentro de su apartado paréntesis. Veamos una estructura de función flecha con parámetros y retorno.

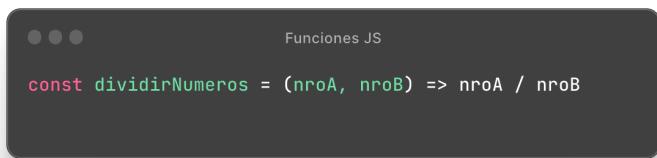
Figura 32. Código

```
const dividirNumeros = (nroA, nroB) => {
    return nroA / nroB
}
```

Fuente: elaboración propia.

Mantenemos siempre la misma estructura de *arrow function*. Aunque en este último ejemplo, podemos simplificar la función al llevar su lógica a una sola línea de código.

Figura 33. Código



```
••• Funciones JS  
const dividirNumeros = (nroA, nroB) => nroA / nroB
```

Fuente: elaboración propia.

Una ventaja de utilizar *arrow functions* es que, cuando su estructura retorna un valor y resuelve la tarea a realizar en una sola línea de código, podemos prescindir de las llaves de bloque y de la palabra reservada *return*. De igual forma pasa cuando definimos una función con su solo parámetro: en este caso, al usar un solo parámetro, podemos prescindir de los paréntesis de la función flecha, tal como muestra el siguiente ejemplo de código.

Figura 34. Código



```
••• Funciones JS  
const saludar = param => console.log("Hola " + param)
```

Fuente: elaboración propia.

También podemos obviar las llaves de bloque si es que la función resuelve en una sola línea de código la tarea que debe realizar.

Si bien las *arrow functions* simplifican mucho nuestro código al obviar paréntesis y llaves de bloque en determinados casos, no es obligatorio que siempre las utilicemos así. Conviene acostumbrarse a utilizar *arrow functions*, porque los *frameworks* y librerías JS modernos promueven el uso de estas últimas por sobre las funciones convencionales.

Actividad 4

Para adaptarnos a la modernidad, convertir la función creada en la microactividad anterior, a un formato de función flecha.

Definir, en esta función flecha, un parámetro llamado **array** y cambiar el código del ciclo de iteración **for**, para que itere sobre el parámetro **array**.

Cuando se invoque la función, pasarle como parámetro el **arrayFrutas**. De esta forma llegaremos al mismo resultado de iteración, pero al hacer que la función itere un **array** que recibe como parámetro, y que no quede atada a un **array** definido de forma global.

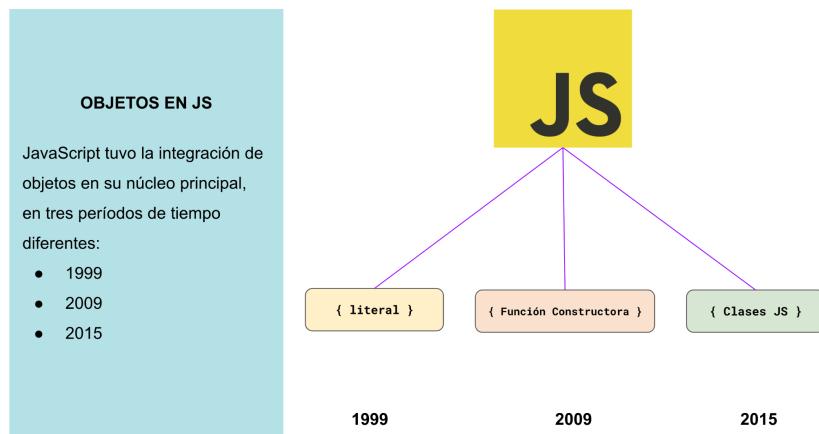
Para conocer la respuesta correcta, descargá el PDF al final de la lectura.

Tema 4: Objetos

Dentro de la evolución del lenguaje JavaScript, encontramos tres períodos diferentes donde se agregó soporte para el manejo de objetos. Al ser pensado inicialmente como un lenguaje de *scripting* con poca participación en la lógica media o avanzada de un sitio o aplicación web, JavaScript no tuvo la intención inicial de manejar internamente objetos.

El tiempo y las mejores condiciones de navegación por internet, además de la potencia de procesamiento, hicieron que JS pueda integrar mejores opciones para el manejo de objetos.

Figura 35: Evolución de los objetos en el lenguaje JavaScript



Fuente: elaboración propia.

Todas las opciones de objetos pueden utilizarse hoy en día. Veamos a continuación cada una de ellas y qué diferencias tienen entre sí.

Objeto literal

Un objeto literal JS es una forma de crear un objeto mediante la utilización de la sintaxis de llaves y valores. Los objetos literales se definen como una lista de pares **clave-valor** separados por comas y encerrados entre llaves. Cada clave es una propiedad del objeto y su valor puede ser cualquier tipo de dato válido en JavaScript.

Los objetos literales se utilizan con frecuencia para representar datos estructurados en forma accesible y, actualmente, son el elemento esencial para representar un *array* de objetos, el cual permite manejar estructuras de datos consistentes en JS.

Figura 36. Código



```
const persona = {  
    nombre: "Juan",  
    apellido: "Pérez",  
    edad: 35,  
    ocupacion: "Programador"  
}
```

Fuente: elaboración propia.

Se vio que un objeto es definido de forma singular. Su nombre debe describir lo que encontraremos en su estructura. La propiedad de cada objeto es de tipado débil y su valor puede cambiar desde un tipo de datos a otro, sin problema alguno.

Figura 37. Código



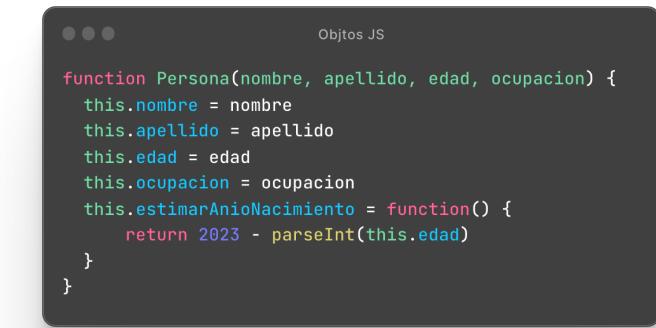
```
const persona = {  
    nombre: "Juan",  
    apellido: "Pérez",  
    edad: 35,  
    ocupacion: "Programador",  
    estimarAnioNacimiento: function() {  
        return 2023 - this.edad  
    }  
}  
persona.estimarAnioNacimiento() // retornará 1988
```

Fuente: elaboración propia.

Funciones constructoras

Una función constructora es una función avanzada que se utiliza para crear objetos con un conjunto de propiedades y métodos predefinidos. Veamos el ejemplo del objeto persona, convertido en una función constructora.

Figura 38. Código

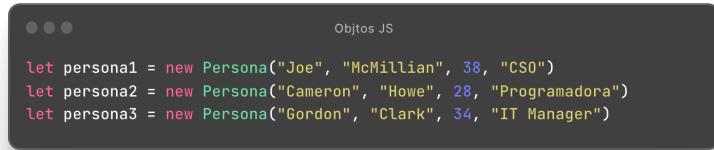


```
function Persona(nombre, apellido, edad, ocupacion) {
    this.nombre = nombre
    this.apellido = apellido
    this.edad = edad
    this.ocupacion = ocupacion
    this.estimarAnioNacimiento = function() {
        return 2023 - parseInt(this.edad)
    }
}
```

Fuente: elaboración propia.

A diferencia de un objeto literal, una función constructora es desarrollada para que podamos crear múltiples objetos, a partir de una instancia generada de la misma. Cuando creamos una instancia de una función constructora debemos utilizar la palabra clave **new**, se crea un nuevo objeto que hereda las propiedades y métodos de la función constructora.

Figura 39. Código



```
let persona1 = new Persona("Joe", "McMillian", 38, "CSO")
let persona2 = new Persona("Cameron", "Howe", 28, "Programadora")
let persona3 = new Persona("Gordon", "Clark", 34, "IT Manager")
```

Fuente: elaboración propia.

Estructura de una función constructora

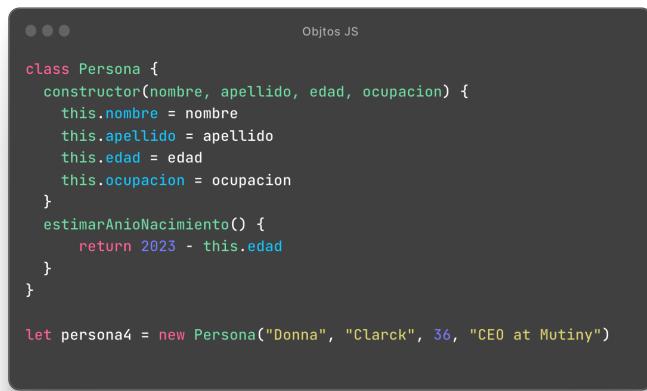
Para crear una función constructora, definimos una función JS pero su nombre debe comenzar en mayúscula, y ser una palabra (sustantivo) singular. Dentro de la función constructora, definimos las propiedades del objeto mediante el uso de la sintaxis de **this.propiedad = valor**. La palabra reservada **this** se refiere al objeto que se crea a través de la función constructora. Es decir, cuando creamos una nueva instancia de la función constructora mediante la utilización de la cláusula **new**, **this** se refiere al objeto recién creado como si se tratara de un sinónimo.

Esta es la forma en la cual podremos instanciar múltiples objetos a partir de esta función constructora. La misma oficiará como ‘plantilla’.

Clases JS

Las clases JS son la forma más moderna de crear objetos y estructuras de datos que encapsulan propiedades y métodos relacionados. Una clase define la forma de un objeto y luego, mediante la creación de instancias de la clase, se pueden crear objetos específicos que heredan las propiedades y métodos de la clase.

Figura 40. Código



```
Objeto JS

class Persona {
    constructor(nombre, apellido, edad, ocupacion) {
        this.nombre = nombre
        this.apellido = apellido
        this.edad = edad
        this.ocupacion = ocupacion
    }
    estimarAnioNacimiento() {
        return 2023 - this.edad
    }
}

let persona4 = new Persona("Donna", "Clarck", 36, "CEO at Mutiny")
```

Fuente: elaboración propia.

Nacieron en el año 2015, y son denominadas ***syntactic sugar***, ya que no son más que una máscara para escribir funciones constructoras, con la diferencia de que las clases JS se estructuran en otros lenguajes de programación 100% orientados a objetos, de la misma forma en la cual JS permite escribirlas. Esto hace que cualquier desarrollador de **software** que provenga de otro lenguaje más estricto, se sienta cómodo en JS al escribir clases JS.

Si comparamos la clase JS y la función constructora, veremos que la clase JS es más limpia de estructurar que la función constructora. Además, las clases JS poseen soporte para propiedades y métodos estáticos, lo cual las hace todavía más interesantes para programadores foráneos.

Una vez creada, la instanciamos tal cual lo haríamos con una función constructora JS.

Cuando utilizar cada uno

Entre las funciones constructoras y las clases JS, recomendamos aprender e implementar en el día a día, las clases JS. De acuerdo a las buenas prácticas y en las empresas que utilizan a JS como lenguaje de programación para todo tipo de desarrollo de **software**, promueven las buenas y modernas prácticas.

Debemos tener presente cómo se crean las funciones constructoras por si en algún momento participamos en la migración de *software legacy* a una aplicación web más moderna. Es probable que, en estos casos, que abundan en las empresas de tecnología, nos encontremos con funciones constructoras implementadas en código JavaScript.

Los objetos literales, por su parte, son implementados más en estructuras de datos del tipo *array* de objetos, que creados y utilizados de forma aislada. Próximamente comenzaremos a trabajar más de cerca con *arrays*, y veremos que los objetos literales son una parte esencial en estos.

Actividad 5

Analizar el archivo **productos.js** del proyecto modelo que compartimos. Con lo que es la estructura fija de un producto, imaginar cómo debería ser una clase JS que pueda manejar el valor del stock de este y calcular el importe del producto con diferentes porcentajes de descuento.

Ahora escribir una clase llamada 'producto', la cual recibirá, en su método constructor, los siguientes datos: *id*, *precio*, *stock*.

Agregar un método llamado **descontarDeStock()**. En el mismo, debemos recibir como parámetro las **unidades** a descontar de *stock*. En su lógica, aplica un control del valor que recibe como parámetro para saber que éste es numérico. Si no es numérico, envía un error a la consola JS y corta la ejecución del método.

En un segundo control, dentro de este mismo método, validar que el stock a descontar, menos el stock que tiene definido el producto en sí, no dé como resultado un valor menor a 0 (cero). Si da un valor negativo, envía un error a la consola y corta la ejecución del método.

Si pasa todas estas validaciones, entonces aplicar el descuento sobre la propiedad *stock* de la clase 'producto', y retornar el valor numérico del stock actualizado a través del mismo método.

Para conocer la respuesta correcta, descargá el PDF al final de la lectura.

Unidad 2. Gestión de datos y acceso al DOM

En este nuevo módulo, abordaremos la manipulación de estructuras de datos en JavaScript. Se explorarán las estructuras de *arrays*, tanto de elementos como de objetos, y aprenderemos los métodos comunes a ambos tipos de *arrays*, así como los métodos específicos para estructuras de datos más complejas con el uso de funciones de orden superior. Esto nos permitirá trabajar con

información proveniente de una aplicación *backend*.

Además, integraremos la lógica JavaScript en documentos HTML mediante el uso del DOM, para lograr una interacción dinámica entre el usuario y los datos. Esta nos ahorra código y facilita el mantenimiento de las aplicaciones web en el ecosistema *online*.

Tema 1: *Arrays*

JS soporta el uso de *arrays* simples, o *array* de elementos, además de poder integrar *array* de objetos. Veamos un ejemplo de cada uno.

Figura 41. Código



```
const paisesDelSur = ['Argentina', 'Uruguay', 'Brasil', 'Venezuela', 'Chile']  
//indice del array: 0 1 2 3 4
```

Fuente: elaboración propia.

Figura 42. Código



```
const productos = [  
  {id: 1, nombre: "Laptop", stock: 10, precio: 1500, categoria: "Computadoras"},  
  {id: 2, nombre: "Monitor", stock: 5, precio: 300, categoria: "Accesorios"},  
  {id: 3, nombre: "Teclado", stock: 20, precio: 50, categoria: "Accesorios"},  
  {id: 4, nombre: "Mouse", stock: 15, precio: 20, categoria: "Accesorios"},  
  {id: 5, nombre: "Impresora", stock: 8, precio: 200, categoria: "Periféricos"}];
```

Fuente: elaboración propia.

Métodos de arrays

Veamos los métodos de *arrays* más importantes de JS.

Tabla 5: Métodos aplicables en arrays

Método	Descripción
push() - unshift()	Agrega elementos a un array, al final o inicio (respectivamente).
pop() - shift()	Elimina elementos de un array, en el final o inicio del mismo (respectivamente).

concat()	Combina dos o más <i>arrays</i> y crea uno nuevo.
slice()	Devuelve una copia superficial de una porción del <i>array</i> .
splice()	Elimina parte del contenido de un <i>array</i> .
indexOf()	Devuelve el primer índice en el cual se encuentra un elemento específico.
includes()	Verifica si un <i>array</i> contiene un elemento específico, y retorna true o false .

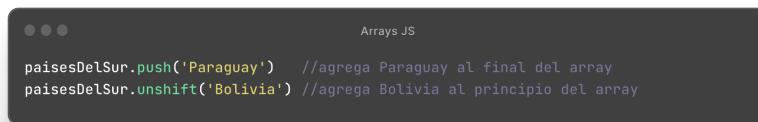
Fuente: elaboración propia.

Si bien son muchos más los métodos de *arrays* existentes, aquí encontramos algunos de los más importantes. Existen otros tantos métodos, definidos como funciones de orden superior, y que son más efectivos cuando trabajamos con *arrays* con estructuras de objetos literales. Estos nos ayudan a obtener un manejo de datos mucho más efectivo sobre un conjunto de información usualmente recibida desde una aplicación *backend*.

Interactuar con los métodos de *arrays*

Aquí tenemos un ejemplo de cómo agregar un país al final del *array*, y cómo agregar otro país al inicio del mismo.

Figura 43. Código



```
••• Arrays JS
paisesDelSur.push('Paraguay') //agrega Paraguay al final del array
paisesDelSur.unshift('Bolivia') //agrega Bolivia al principio del array
```

Fuente: elaboración propia.

Si quisieramos obtener la posición de un país en el *array* de elementos, podemos utilizar para ello el método **indexOf()**. Combinamos con el operador ternario, la respuesta de sí se encuentra o no el país buscado.

Figura 44. Código

```
••• Arrays JS  
const paisesDelSur = ['Bolivia', 'Argentina', 'Uruguay', 'Brasil', 'Venezuela',  
'Chile', 'Paraguay']  
  
const indice = paisesDelSur.indexOf('Paraguay')  
indice > -1 ? console.log("Se encuentra en la posición:", indice)  
: console.warn("No se encontró el país especificado.")
```

Fuente: elaboración propia.

Y, en el caso de querer quitar dicho elemento del *array*, utilizamos el método ‘.splice()’ junto con el índice obtenido mediante el método **indexOf()** ejecutado anteriormente.

Figura 45. Código

```
••• Arrays JS  
paisesDelSur.splice(indice, 1) //Elimina el elemento ubicado en el índice definido  
console.warn("Se ha eliminado el elemento ubicado en el índice:", indice)
```

Fuente: elaboración propia.

Verificar si existe un elemento en el *array*

El método *includes* es el más utilizado para definir si un elemento ya existe en el *array*, o no. Usualmente se lo utiliza cuando debemos dar de alta nuevos elementos, para verificar si existe o no, y así se evita tener elementos duplicados.

Figura 46. Código

```
••• Arrays JS  
const resultado = paisesDelSur.includes('Chile') //verifica si existe el elemento  
  
resultado ? console.warn("El elemento existe en el array")  
: paisesDelSur.push('Chile')
```

Fuente: elaboración propia.

Iterar elementos de un *array*

El método **forEach()** es un método creado íntegramente para iterar un *array* de objetos o elementos. Su uso escapa a la forma de invocar el resto de los métodos de *array*, porque el

mismo recibe una función como parámetro, en la cual se define qué tarea ejecutar ante la iteración de cada elemento del *array*. Veamos un ejemplo.

Figura 47. Código

```
arrays.js
paisesDelSur.forEach(pais => {
    console.log(pais)
})
```

Fuente: elaboración propia.

ForEach itera cada elemento existente en este *array*, lo pasa como parámetro a la función flecha, para imprimir a este en la consola JS. Si quisiéramos mostrar estos elementos en el documento HTML, deberíamos utilizar este mismo método para iterar el *array*.

Si bien esta acción es posible llevarla a cabo con el uso del ciclo *for* convencional, el método *forEach* es mucho más efectivo para esta tarea porque está optimizado en velocidad para responder de mejor forma ante *arrays* con cientos o miles de elementos. Además, es más simple su forma de escribir.

Actividad 6

En línea con nuestro *array* de elementos llamado **paisesDelSur**, crear una función llamada **agregarElemento()**, la cual recibe un parámetro llamado **país**. La lógica será agregar el elemento recibido como parámetro en el *array*, pero mediante la realización de validaciones previas.

1. Validar si el *array* no lo incluye previamente. Agregarlo si no existe.
2. En la validación considerar que el parámetro puede venir en mayúsculas o minúsculas. Por ello, deben ‘normalizarse’ los datos recibidos, previo a compararlos.
3. Si existe el valor del parámetro en el *array*, arroja una advertencia a la consola JS, y corta la ejecución de la función. Si no existe, agrega el mismo en el *array*.

Para conocer la respuesta correcta, descargá el PDF al final de la lectura.

Las funciones de orden superior en JavaScript son aquellas funciones que pueden recibir otras funciones como argumentos o devolver funciones como resultado. Son versátiles y poderosas, ya que permiten operar y manipular funciones de manera flexible.

Con estas funciones, podemos abstraer la lógica común, reutilizar código y crear estructuras más dinámicas, y lograr así un código más conciso, legible y modular, lo que facilita el desarrollo de aplicaciones y la implementación de conceptos como **callback**, **mapeo** y **filtrado de arrays**, entre otros.

Veamos a continuación todas las funciones de orden superior que existen como métodos de **arrays**, y su aplicación específica.

Tabla 6: Métodos aplicables en arrays

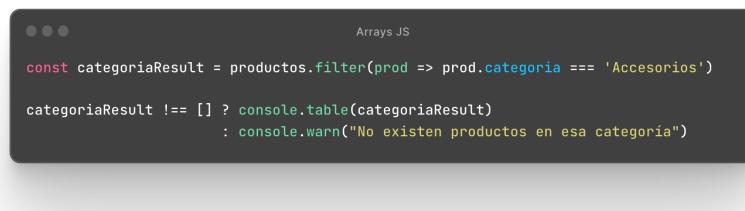
Método	Descripción
forEach()	Itera los elementos u objetos de un array , de principio a fin.
filter()	Permite aplicar un filtro sobre elementos de un array , y retorna un nuevo array con el o los resultados obtenidos.
find()	Busca entre los elementos u objetos de un array , un elemento específico de acuerdo al parámetro indicado.
reduce()	Permite simplificar los elementos de un array a un único valor resultante.
map()	Permite generar un mapeo de datos sobre los elementos de un array .
sort()	Permite ordenar los elementos de un array de forma ascendente.
reverse()	Aplicable solo sobre un array de elementos, invierte la posición de ordenamiento de dichos elementos.
findIndex()	Permite buscar un objeto dentro de un array , y obtener su índice. Es el método equivalente a <code>indexOf()</code> aplicable a un array de objetos.

El método `forEach` ejemplificado anteriormente, representa la estructura básica que cada una de las funciones de orden superior aquí mencionadas utiliza para trabajar con un conjunto de datos almacenados dentro de un *array* de objetos. Algunas también son aplicables a los *arrays* de elementos, aunque son más efectivas sobre *array* de objetos.

Filtrar productos de un *array*

Aquí tenemos un ejemplo aplicado con el uso de la función de orden superior `filter()` sobre el *array* `productos`.

Figura 48. Código



```
const categoriaResult = productos.filter(prod => prod.categoría === 'Accesorios')

categoriaResult !== [] ? console.table(categoriaResult)
                      : console.warn("No existen productos en esa categoría")
```

Fuente: elaboración propia.

Este método nos retorna un *array* de objetos nuevo, con los elementos coincidentes que posean en su propiedad **categoría**, el valor ‘accesorios’. Si no existe coincidencia, retorna un *array* vacío. Debemos tener esto presente para validar el resultado previo a utilizarlo.

Buscar un producto en un *array*

El método `find()` nos retorna el primer elemento coincidente que encuentre sobre el *array* en cuestión. En un *array* de objetos, debemos definir una propiedad de este por la cual buscar.

Figura 49. Código



```
const categoriaResult = productos.find(prod => prod.id === 4)

productoResult !== undefined ? console.table(productoResult)
                           : console.warn("No se encontró producto con ese ID.")
```

Fuente: elaboración propia.

Itera el *array* de principio a fin. Ante la primera coincidencia, retorna el objeto como resultado y deja de iterar el resto del *array*. Si no existe coincidencia, retorna ***undefined***. Esto último también

debemos tenerlo presente para validar el resultado previo a utilizarlo. No debemos confundir su acción con el resultado que devuelve el método `.filter()`.

Reducir valor a un total

El método `reduce` se utiliza para reducir la estructura de un `array` a un único valor. Su uso más común se basa en iterar un `array` y sumar valores numéricos determinados, para así obtener un resultado. Veamos un ejemplo a continuación.

Figura 50. Código

```
... Arrays JS

const carrito = [{id: 3, nombre: 'Teclado', precio: 50},
                 {id: 4, nombre: 'Mouse', precio: 50}]

const totalCarrito = carrito.reduce((acc, prod)=> acc + prod.precio, 0)
console.log("El total del carrito es: $", totalCarrito)
```

Fuente: elaboración propia.

Ante un hipotético carrito conformado por un `array` de objetos, iteramos el mismo con el método `'reduce()'`. Este método recibe como parámetros un primer parámetro denominado `acc`, de acumulador. El segundo parámetro es cada objeto del `array`.

En la variable `acc` nos ocupamos de acumular el valor numérico de la propiedad `precio`, de cada producto del `array carrito`. El parámetro cero, definido al final del método, refiere al valor inicial que asume el parámetro `acc`.

De esta forma, calculamos fácilmente el total de un carrito de productos.

Mapear objetos de un array

El método `map()` nos permite convertir, transformar, o mapear la estructura de objetos de un `array` específico. Este nos ayuda a, por ejemplo, simplificar la estructura de un `array` de objetos, si es que no necesitamos trabajar dicho `array` con todas las propiedades que este tenga definidas.

Figura 51. Código

```
const productosMapeado = productos.map(prod => {
    return {
        id: prod.id,
        nombre: prod.nombre,
        precio: prod.precio
    }
})

console.table(productosMapeado)
```

Fuente: elaboración propia.

En este ejemplo simple, mapeamos el *array* *productos*, eliminamos las propiedades *stock* y *categoría*, si es que no necesitamos tenerlas presentes dentro del *array*.

El uso de **map()** no destruye el *array* original. Solo crea un nuevo *array* a partir de los datos mapeados, que sean de nuestro interés.

Figura 52. Código

```
const productosMapeado = productos.map(prod => {
    return {
        id: prod.id,
        nombre: prod.nombre,
        precio: prod.precio,
        precio100ff: prod.precio * 0.90
    }
})

console.table(productosMapeado)
```

Fuente: elaboración propia.

También es completamente útil para, por ejemplo, generar campos calculados. Si necesitáramos contar con un nuevo campo que muestre el precio del producto con un 10 % de descuento, podremos generar dicha propiedad como campo calculado, en el mapeo de datos del *array* en cuestión.

Ubicar el índice del objeto de un *array*

findIndex() es un método que nos permite obtener el índice del objeto de un *array*. Si el objeto dentro del *array* existe, **findIndex()** retornará su índice, el cual podemos atrapar en una variable o constante. Si el objeto no existe, retorna **-1** como resultado.

Figura 53. Código

```
● ● ● Arrays JS
const nuevoProducto = {id: 4, nombre: 'Mouse', precio: 50, cantidad: 1}

const carrito = [{id: 3, nombre: 'Teclado', precio: 50, cantidad: 1},
                 {id: 4, nombre: 'Mouse', precio: 50, cantidad: 1}]

const indice = carrito.findIndex(prod => prod.id === nuevoProducto.id)
indice > -1 ? carrito[indice].cantidad++
              : carrito.push(nuevoProducto)
```

Fuente: elaboración propia.

Uno de los usos más efectivos para **findIndex()** es identificar si existe un objeto dentro de un **array**. Por ejemplo, si tenemos un **array carrito** con productos que poseen una propiedad **cantidad**, podemos ubicar un objeto dentro del **array** con **findIndex** y, si este existe, incrementar en un dígito su propiedad **cantidad**. Si no existe, lo agregamos como un nuevo producto.

Ordenamiento de un array

El ordenamiento de un **array** de elementos se realiza con el método **sort()**, sin ningún parámetro adicional. Y se puede concatenar **reverse()** al final para invertirlo.

Figura 54. Código

```
● ● ● Arrays JS
const paisesDelSur = ['Chile', 'Argentina', 'Uruguay', 'Brasil', 'Venezuela']
paisesDelSur.sort()
```

Fuente: elaboración propia.

Pero el ordenamiento de un **array** de objetos, cambia la estructura en sí, por la complejidad de la estructura de datos. Podemos ordenarlo por cualquiera de los campos que posea, y utilizar también el método **sort()**.

Figura 55. Código

```
Arrays JS

const productos = [
  {id: 1, nombre: "Laptop", stock: 10, precio: 1500, categoria: "Desktop"}, 
  {id: 2, nombre: "Monitor", stock: 5, precio: 300, categoria: "Accesorios"}, 
  {id: 3, nombre: "Teclado", stock: 20, precio: 50, categoria: "Accesorios"}, 
  {id: 4, nombre: "Mouse", stock: 15, precio: 20, categoria: "Accesorios"}, 
  {id: 5, nombre: "Impresora", stock: 8, precio: 200, categoria: "Periféricos"}]

productos.sort((a, b)=> {
  if (a.nombre > b.nombre) {
    return 1
  }
  if (a.nombre < b.nombre) {
    return -1
  }
  return 0
})
```

Fuente: elaboración propia.

Y para invertir el mecanismo de ordenamiento, solo debemos cambiar de orden el signo mayor que y menor que, utilizados en cada expresión **if()**. Como alternativa, también podemos cambiar el signo de ordenamiento de los dos primeros **return**.

Figura 56. Código

```
Arrays JS

productos.sort((a, b)=> {
  if (a.nombre < b.nombre) {
    return 1
  }
  if (a.nombre > b.nombre) {
    return -1
  }
  return 0
})
```

Fuente: elaboración propia.

Con este cambio logramos fácilmente un **ordenamiento descendente**.

Combinar funciones de orden superior con métodos de arrays

Dentro de cada búsqueda o tratamiento de **arrays** con las funciones de orden superior, podemos combinar el uso de estas con métodos de **arrays** que nos ayuden a transformar elementos, entre otras posibles acciones.

Por ejemplo, cuando buscamos o filtramos productos por un valor del tipo *string* y existen productos que puedan tener un nombre combinado, encadenamos el método ‘*.includes()*’ para realizar una búsqueda aproximada y no estricta, como pasaría con el **operador de comparación igual**.

Figura 57. Código

```
Arrays JS

const categoriaResult = productos.find(prod => prod.id.includes('Mouse'))

productoResult !== undefined ? console.table(productoResult)
                            : console.warn("No se encontró el producto.")
```

Fuente: elaboración propia.

De igual forma cuando, por ejemplo, queremos normalizar el nombre de productos y definir todos ellos en mayúsculas. Podemos combinar el método ‘*map()*’ junto con ‘*toUpperCase()*’ para pasar a mayúsculas los nombres de productos.

Figura 58. Código

```
Arrays JS

const productosSimples = productos.map(prod => {
    return {
        id: prod.id,
        nombre: prod.nombre.toUpperCase(),
        precio: prod.precio.toLocaleString()
    }
})
console.table(productosSimples) //array original simplificado
```

Fuente: elaboración propia.

A su vez, el método ‘*toLocaleString()*’ nos ayuda a visualizar valores numéricos simples en el formato monetario configurado en la computadora del usuario que utiliza nuestra web app.

Tema 3: DOM (parte I)

Estudiamos el uso de DOM HTML mediante el uso de JavaScript, e integramos para ello el objeto global **document**. A través de diferentes métodos de acceso integrados en el objeto **document**, nos conectamos desde JS a elementos HTML específicos, para controlar los mismos de forma efectiva y dinámica, a través del lenguaje JavaScript.

Es importante considerar que siempre conviene definir un archivo JS con todas las conexiones a DOM (entre JS y HTML) que sean globales. Al momento de referenciar el archivo JavaScript en el documento HTML podemos agregarlo dentro del apartado **<head>** y no al final del documento HTML, como siempre se sugiere de forma errónea.

Figura 59. Código



```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Venta de Frutas</title>
    ... Aquí va el resto de archivos CSS y meta tags...
    <script defer src="js/index.js"></script>
</head>
```

Fuente: elaboración propia.

Desde el año 2014 está disponible el atributo **defer** (del inglés, ‘diferir’; como se ve en la muestra anterior) en todos los navegadores web. El mismo se utiliza junto al tag **<script>**. Este atributo hace que, cuando el motor web se encuentra con el atributo **defer**, retrase la carga e interpretación del código JS, y espere a que se termine de interpretar la última línea del documento HTML.

Defer aporta orden en un documento HTML, al referenciar archivos CSS y JS en el apartado **<head>**, además de que las buenas prácticas en **SEO** piden que todo sitio web público refiera CSS y JS en el apartado **<head>**, para que los buscadores web mejoren el posicionamiento en resultados del sitio o aplicación web.

Métodos de acceso convencionales

Los métodos de acceso convencionales que utiliza JS y que siguen vigentes hoy, son: **getElementById**, **getElementsByName**, **getElementsByClassName** y **getElementsByTagName**. Si bien no es incorrecto que los utilicemos, estos limitan bastante la conexión con precisión a elementos HTML dado que, el avance de características que tiene HTML5 y la evolución constante de CSS los dejaron como métodos limitados.

Métodos de acceso modernos

Desde la llegada de ES6, en el año 2015, JavaScript sumó los métodos de acceso a tags HTML llamados **querySelector** y **querySelectorAll**. Estos métodos poseen una mayor flexibilidad tomando como referencia la estructura de estilos conocida como **selectores**, que utiliza de forma

predeterminada CSS.

El método `querySelector`

Como se muestra a continuación, `querySelector` es más flexible que `getElementById` para conectarse a un elemento HTML, ya que utiliza la lógica de selectores CSS.

Figura 60. Código



```
● ● ● Arrays JS
const container = document.querySelector('div#container')
//div con atributo id 'container'

const buttonView = document.querySelector('button.button-outline.button-add')
//button con dos clases CSS específicas

const footer = document.querySelector('footer')
//tag HTML footer

const imageCarrito = document.querySelector('#imgCarrito')
//tag HTML cuyo id es 'imgCarrito'
```

Fuente: elaboración propia.

Veamos la tabla 7 para ver la referencia a utilizar con `querySelector`, con los elementos HTML.

Tabla 7: QuerySelector

querySelector	Descripción
('tag')	Definimos un <i>tag</i> HTML específico.
('#id')	Definimos un atributo <i>id</i> específico.
('.clase-css')	Definimos un elemento HTML por su clase CSS.

Podemos combinar todas las opciones anteriores en un único `querySelector`, y lograr, así, una precisión quirúrgica para seleccionar un *tag* HTML.

Fuente: elaboración propia.

Propiedades de lectura-escritura en elementos HTML

Para trabajar con los elementos HTML de forma individual, modificar su texto o contenido en general, JavaScript nos provee las siguientes propiedades.

innerText || textContent

`innerText` nos permite leer el texto que contenga el elemento HTML al cual nos conectamos

desde JavaScript, y también modificarlo mediante el uso del operador de asignación igual.

Figura 61. Código

```
••• Arrays JS
const buttonView = document.querySelector('button.button-outline.button-add')

console.log(buttonView.innerText) //retorna el texto del botón HTML

buttonView.innerText = 'VER CARRITO' //definimos el texto del botón HTML
```

Fuente: elaboración propia.

textContent es una propiedad similar a **innerText**, que nació hace unos pocos años. Cumple el mismo propósito que **innerText**. Cualquiera de las dos opciones, es válida en la actualidad.

Figura 62. Código

```
••• DOM JS
const buttonView = document.querySelector('button.button-outline.button-add')

console.log(buttonView.textContent) //retorna el texto del botón HTML

buttonView.textContent = 'VER CARRITO' //definimos el texto del botón HTML
```

Fuente: elaboración propia.

innerHTML

Esta propiedad permite leer o escribir contenido HTML dentro de los *tags* HTML que ofician como contenedores.

Figura 63. Código

```
••• DOM JS
const ul = document.querySelector('ul')

ul.innerHTML = '<li>ítem nro. 1</li>' //agregamos un listItem al elemento ul
```

Fuente: elaboración propia.

También permite borrar un elemento o bloque HTML, al utilizar el operador de asignación igual seguido de comillas vacías. Y si, por ejemplo, deseamos aplicar una imagen predeterminada

sobre un *tag* HTML del tipo ****, podemos recurrir a su atributo **src**, el cual veremos desde JS como una propiedad.

Figura 64. Código

```
... DOM JS
const imageCarrito = document.querySelector('#imgCarrito')
//tag HTML cuyo id es 'imgCarrito'

imageCarrito.src = 'images/icono-carrito.jpg'
```

Fuente: elaboración propia.

El método **querySelectorAll**

Este método nos conecta a una colección de elementos HTML al seguir la misma flexibilidad de **querySelector**, pero retorna una colección de elementos HTML coincidentes con el selector especificado como parámetro.

Imaginemos tener que aplicar una misma clase CSS a varios elementos de un documento HTML. Para ello, **querySelectorAll** nos permite crear una colección de dichos elementos.

Figura 65. Código

```
... Arrays JS
const copetes = document.querySelectorAll('p#copete-noticia')
```

Fuente: elaboración propia.

Hay que tener presente que esta referencia nos retornará una colección, y no un *array*. Por lo tanto, para iterar la misma con aplicación de una clase CSS adicional, debemos utilizar la sentencia **for...of**. El método **forEach** no funciona. El uso de **for** convencional, sí, aunque nos conviene **for...of**, por ser más simple.

Figura 66. Código

```

    ...
        Arrays JS

    const copetes = document.querySelectorAll('p.copete-noticia')

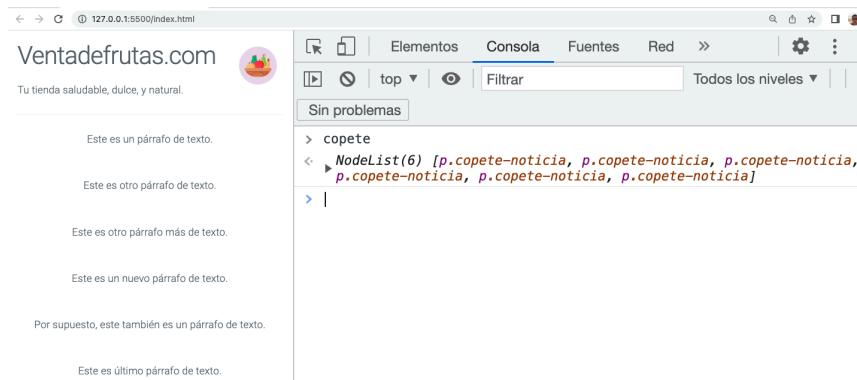
    for (copete of copetes) {
        copete.className += ' otra-clase-css'
    }

```

Fuente: elaboración propia.

Como vemos en el ejemplo anterior, **for...of** simplifica la colección de elementos HTML, al definir la iteración de cada elemento HTML en el primer parámetro del **for**. Dentro del ciclo de iteración nos ocupamos de utilizar la propiedad correspondiente para modificar cada elemento HTML desde JS.

Figura 67: Uso del método querySelectorAll verificado en la consola JS



Fuente: elaboración propia.

En próximos módulos, reflotaremos este ejemplo para definir un evento click masivo a varios botones HTML, y combinar esta lógica con el uso de `querySelectorAll`.

Actividad 7

Abrir con VS Code el proyecto modelo que compartido. Editar el documento `index.html` y ubicar el apartado que contiene la *card* HTML con una fruta.

1. Copiar la estructura HTML de la *card* completa, y pegarla al menos 7 u 8 veces más en el documento HTML en sí, siempre dentro del *tag* contenedor.
2. A continuación, asegurarnos de tener el archivo JS declarado en este documento HTML, con el atributo `defer` referencia. Si no tiene el atributo `defer`, agregarlo.
3. En el archivo JS, declarar una constante llamada *botones* y referenciar en esta mediante la utilización de `querySelectorAll`, que se conecte a todos los botones HTML del documento en cuestión.
4. Realizar la validación mediante el uso de **DevTools > Consola JS**, que la constante declarada genere una colección de botones, con un número igual a la cantidad de

cards que se replicaron en el documento HTML.

Para conocer la respuesta correcta, descargá el PDF al final de la lectura.

Tema 4: DOM (parte II)

Crear elementos HTML

Al momento, solo vimos cómo acceder a determinados elementos HTML creados de manera estática en el archivo homónimo, para leer o modificar su contenido. Pero, en determinadas situaciones, cuando la lógica de JS necesita recrear visualmente en HTML contenido dinámico proveniente, tal vez, de una aplicación backend, debemos crear dichos elementos HTML. Para ello, según la necesidad, contamos con diferentes formas de hacerlo.

Tabla 8: Elementos HTML

Crear elementos HTML	
Tipo de elemento	Utilizamos
Simple	El método createElement y el método append .
	La propiedad innerHTML más el uso de bloques HTML como <i>strings</i> .
Múltiples elementos	La propiedad innerHTML combinada con template string más template literals .

Fuente: elaboración propia.

Veamos a continuación, en qué situación aplicar a cada uno de ellos.

Crear un elemento HTML simple con innerHTML

La forma más rápida de crear un elemento simple, es utilizar la propiedad **innerHTML**, como vimos anteriormente. Supongamos que tenemos el *tag* **<main>** dentro del documento HTML y deseamos agregarle un título de primer nivel. Para ello, enlazamos dicho *tag* desde JS mediante el uso del método **querySelector**, y luego de ello referenciamos a la propiedad **innerHTML** para escribir el *tag* en cuestión.

Figura 68. Código



```
DOM  
const main = document.querySelector('main');  
main.innerHTML = '<h1>Título de primer nivel</h1>'
```

Fuente: elaboración propia.

Si el texto del título proviene de una constante o de una variable, podemos concatenar su valor en el uso de la propiedad `innerHTML`.

Figura 69. Código



```
DOM  
let titulo = 'Título de primer nivel';  
main.innerHTML = '<h1>' + titulo + '</h1>'
```

Fuente: elaboración propia.

Esta forma es totalmente válida de utilizar para generar estructuras de HTML dinámicas.

Ahora, cuando la estructura de HTML dinámico requiere no solo mostrar un texto, sino también configurar atributos dinámicos, como `ID` o `class`, entre otros tantos específicos que puedan llegar a contener determinados *tags* HTML, entonces allí debemos recurrir a otras herramientas algo más efectivas para este tipo de trabajo.

Crear elementos HTML con `createElement()`

El método '`createElement()`' forma parte del objeto JS `document`. Este nos permite crear un elemento HTML específico, informado como parámetro entre los paréntesis de este método, configurar sus diferentes atributos, y luego agregarlo manualmente al documento HTML.

Figura 70. Código



```
DOM

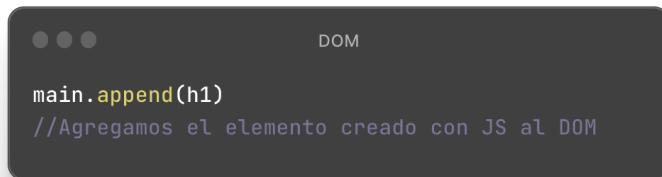
const main = document.querySelector('main')

const h1 = document.createElement('h1')
h1.textContent = 'Título de primer nivel'
h1.id = 'mainTitle'
h1.className = 'main-title blue-text'
```

Fuente: elaboración propia.

Hasta aquí, el elemento HTML del tipo **H1**, está creado en la memoria del navegador web. Solo falta agregarlo al documento HTML para que aparezca en el DOM. Esto lo realizamos con el método **append()**, disponible dentro del elemento HTML que oficiará de padre o de contenedor. En nuestro ejemplo, será **main**.

Figura 71. Código



```
DOM

main.append(h1)
//Agregamos el elemento creado con JS al DOM
```

Fuente: elaboración propia.

De esta forma, tendremos un título de primer nivel visible en el documento HTML, con el texto indicado, y sus atributos **ID** y **class** configurados directamente desde JS.

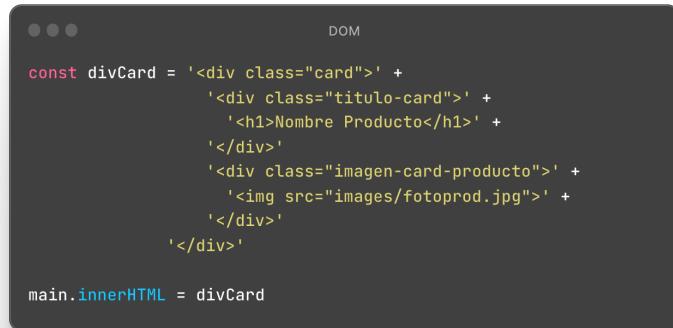
Tengamos presente siempre que a todo atributo de elementos HTML (cuando los enlazamos desde JS) podremos acceder para leer o configurar su valor, tratándolos como una propiedad del elemento al cual estamos enlazados.

Template string + literals

A partir de **EcmaScript 6** (desde el año 2015), llegó a JS una forma diferente de trabajar con la generación de tags HTML desde JS, de la mano de **template string**. Esta característica incorpora

la tilde ` denominado *backtick* como carácter que permite encerrar bloques HTML con atributos y valores definidos. La integración del carácter *backtick* nos libera del deber generar múltiples concatenaciones más un uso excesivo de comillas dobles y comillas simples. Veamos un ejemplo a continuación.

Figura 72. Código



```
DOM

const divCard = `<div class="card"> +
    <div class="titulo-card"> +
        <h1>Nombre Producto</h1> +
    </div>
    <div class="imagen-card-producto"> +
         +
    </div>
</div>

main.innerHTML = divCard
```

Fuente: elaboración propia.

Y peor aún si debemos intercalar datos de un producto a partir de un objeto literal JS.

Figura 73. Código



```
DOM

const divCard = `<div class="card"> +
    <div class="titulo-producto-card"> +
        <h1> + producto.nombre + '</h1>' +
    </div>
    <div class="imagen-card-producto"> +
        <img src="" + producto.rutaImagen + '"> +
    </div>
</div>

main.innerHTML = divCard
```

Fuente: elaboración propia.

Para subsanar esto, el uso de *backtick* alivia esta tarea. Así, los bloques de código HTML desde JS, se pueden manejar más fácilmente mediante el uso de **template string**.

Figura 74. Código

```
● ● ● DOM
const divCard = `<div class="card">
    <div class="titulo-producto-card">
        <h1>Nombre producto</h1>
    </div>
    <div class="imagen-card-producto">
        
    </div>
</div>

main.innerHTML = divCard
```

Fuente: elaboración propia.

Template literals

Esta característica permite definir una estructura de plantilla para representar el valor de la propiedad de un objeto literal o del valor de una variable o constante, dentro de un bloque HTML definido en una estructura de **template string**. Veamos a continuación un ejemplo dinámico de la *card*, adaptada para mostrar valores de las propiedades de un objeto literal.

Figura 75. Código

```
● ● ● DOM
const divCard = `<div class="card">
    <div class="titulo-producto-card">
        <h1>${producto.nombre}</h1>
    </div>
    <div class="imagen-card-producto">
        
    </div>
</div>

main.innerHTML = divCard
```

Fuente: elaboración propia.

El uso de **template literals** es totalmente válido para representar valores estáticos, tanto en los textos o contenidos de un elemento HTML, la ruta a una imagen, video o audio, como también para generar atributos dinámicos.

Figura 76. Código



```
DOM
const divCard = `<div class="card" id="${producto.id}">
  <div class="titulo-producto-card">
    <h1>${producto.nombre}</h1>
  </div>
  <div class="imagen-card-producto">
    
  </div>
</div>`
```

Fuente: elaboración propia.

Cuando aplicarlos

Este último ejemplo es, claramente, el ejemplo más apropiado para cuando debemos generar bloques de HTML de forma dinámica e iterativa. Pensemos en un **ecommerce**, donde tendremos *cards* HTML por doquier, las cuales representan diferentes productos.

Todas estas *cards* HTML se generarán a partir de contenido proveniente de una aplicación *backend*, y almacenadas en JavaScript dentro de un *array* de objetos.

Actividad 8

Abrir con VS Code el proyecto modelo que compartido. Editar el documento **index.html** y ubicar el apartado que contiene la card HTML con una fruta.

1. Referenciar una constante llamada **container**, que conecte con el **div.container** del documento HTML.
2. Copiar nuevamente la estructura HTML de la **card** completa, y crear en el archivo JS, una función llamada **retornarCardHTML()**. Pegar la estructura HTML dentro de esta función y adaptar la misma al formato *template string*. Agregar **return** en esta función para que nos retorne la *card* HTML completa.
3. Incorporar un parámetro llamado **producto** a la función anterior. Este parámetro recibirá un objeto literal con la estructura de cada fruta (de acuerdo al array de objetos almacenados en el archivo **productos.js**)
4. Reemplazar en la **card** HTML: el emoji, el nombre, y el precio de la fruta, por un *template literal* del objeto que recibe como parámetro. Esto permitirá iterar el *array* **productos**, pasar como parámetro a esta función cada uno de los productos, y que nos retorne el HTML armado para definir la estructura de frutas en el documento HTML.
5. Finalmente, crear una función llamada **cargarProductos()**. La misma recibe como parámetro un *array*. Internamente define que se itere el *array* recibido como parámetro mediante el método **forEach**. Cada producto se le pasará al método **forEach** como parámetro.

6. Dentro de la iteración `forEach`, escribir de forma acumulativa en el método `container.innerHTML` cada una de las `cards` retornadas en la iteración. De esta manera, armaremos todo el contenido del documento HTML con los productos a vender.

Para conocer la respuesta correcta, descargá el PDF al final de la lectura.

Manejar CSS desde JavaScript

CSSOM es la sigla que nos brinda un set de herramientas poderosas para trabajar en profundidad todo lo relacionado a CSS, desde el lenguaje JavaScript. Son demasiadas las herramientas y diferentes sus usos, por lo cual rescataremos, de todas ellas, los puntos más importantes que pueden facilitarnos una interacción con CSS desde JS, en nuestro día a día como desarrolladores de *software*.

Tabla 9: CSSOM

CSSOM (*CSS object model*)

Propiedad	Descripción
Style (propiedad)	Integrada a cada JS HTMLElement , nos brinda acceso de lectoescritura a cada propiedad o estilo CSS que existe. De esta forma, podremos leer, aplicar o modificar cualquier estilo CSS sobre un elemento HTML desde JavaScript.
classList (propiedad)	También está integrada a cada JS HTMLElement . Esta propiedad contiene una serie de métodos que nos permite manipular fácilmente desde JS diversas características de CSS en un elemento HTML.

Fuente: elaboración propia.

La propiedad *style*

Esta propiedad se utiliza directamente cuando nos enlazamos a un elemento HTML desde JS. Estará disponible dentro de cada **HTMLElement** para que trabajemos de forma individual los

diferentes posibles estilos CSS que podemos aplicarle.

Figura 77. Código

```
... CSSOM ...
const h1 = document.querySelector('h1')

h1.style.color = 'blue'
h1.style.backgroundColor = 'blueviolet'
```

Fuente: elaboración propia.

Como vemos en el ejemplo anterior, la propiedad `style` es aplicada sobre el elemento HTML al cual estamos conectados desde JS. Dentro de esta propiedad, estarán accesibles el resto de los estilos que podemos utilizar con CSS.

Tengamos presente, como se muestra en el ejemplo de código, que, cada estilo CSS se tratará como una propiedad desde JS, y que los estilos CSS con palabras combinadas (dos o más palabras), utilizan habitualmente un guion de separación entre una palabra y la siguiente, pero que desde JS debemos cambiar su nombre al formato **camelCase**.

Figura 78. Código

```
... CSSOM ...
//Selector en CSS con la propiedad background color
h1 {
    background-color: darkorange;
}

//HTMLElement en JavaScript trabajando la misma propiedad
h1.style.backgroundColor = 'darkorange'
```

Fuente: elaboración propia.

La propiedad `classList`

Esta otra propiedad, también se utiliza de forma directa desde un elemento JS enlazado a un elemento HTML, aunque se diferencia de `style`, permitiéndonos trabajar de forma dinámica con clases CSS que necesitan ser integradas o removidas en un elemento HTML.

Los métodos integrados a classList, son los siguientes, a saber.

- **classList.add()**

Permite definir una clase CSS en los paréntesis del método, para que esta sea agregada al elemento HTML desde JS.

Figura 79. Código



A screenshot of a browser's developer tools DOM panel. It shows a dark-themed interface with three dots at the top left and the word "DOM" at the top right. Below that is a code editor window containing the following JavaScript:

```
const h1 = document.querySelector('h1')
h1.classList.add('title-background')
```

Fuente: elaboración propia.

Si el elemento HTML tiene más de una clase CSS incorporada, el método **add()** se ocupa de sumar esta última clase definida, en la última posición de la lista de clases CSS existentes en este. Incluso maneja de manera inteligente los espacios necesarios entre clases CSS.

Figura 80. Código



A screenshot of a browser's developer tools DOM panel. It shows a dark-themed interface with three dots at the top left and the word "DOM" at the top right. Below that is a code editor window containing the following HTML:

```
<h1 class="title-main title-background">Título de primer nivel</h1>
```

Fuente: elaboración propia.

- **classList.remove()**

El método **remove()** realiza la operación inversa al método **.add()**. Remueve desde JS una clase CSS existente en el elemento HTML.

Figura 81. Código

```
... CSSOM  
const h1 = document.querySelector('h1')  
h1.classList.remove('title-main')
```

Fuente: elaboración propia.

No importa la posición dentro de CSS donde la clase a remover se encuentre. Este método maneja de manera inteligente esta tarea, al igual que `.add()`.

- `classList.toggle()`

Finalmente, el método `toggle()` se ocupa de alternar el manejo de clases CSS sobre el elemento HTML. Si la clase definida entre los paréntesis de `toggle()` existe en el elemento HTML, la quita, y si la clase no se encuentra dentro del elemento HTML, la agrega.

Resolución actividades

A continuación, te presentamos el PDF con la resolución de las actividades:



Fuente: elaboración propia.

Video de habilidades

Referencias

Luna, F. (30 de abril de 2021). *PWA: desarrolla webs multidispositivos fácil*. Red Users.
<https://www.redusers.com/noticias/publicaciones/progressive-web-apps/>.

Descarga

A continuación, podemos descargar la lectura completa en su formato PDF.