



Introducción

El producto principal de la industria del *software* es un conjunto de instrucciones de computador que, correctamente implantadas, pueden llamarse programa, módulo de programa, aplicación, servicio o simplemente producto *software*. Codificar este conjunto de instrucciones es la tarea que trata de convertir algoritmos en un programa o módulo que compone un programa y que una computadora o teléfono inteligente (*smartphone*) es capaz de ejecutar. Estas instrucciones están escritas en un lenguaje de programación que puede ser interpretado o compilado por una máquina.

El lenguaje de programación es como un sistema de comunicación entre la persona (programador) y la máquina que consiste en un conjunto de reglas gramaticales bien definidas, que le ofrecen al programador la capacidad de escribir (programar) los algoritmos de los que está compuesto un sistema informático. En particular se emplea el lenguaje JavaScript, ya que casi todos los navegadores lo interpretan, y se puede usar para publicar servicios y aplicaciones en internet.

1. Conceptos básicos de programación

Programar es el proceso de crear *software* escribiendo, probando, depurando y dando mantenimiento a las instrucciones del computador en un lenguaje de programación. A este conjunto de instrucciones se les denomina código fuente del *software* creado.

Existen centenares de lenguajes de programación, muchos de ellos fueron creados para dar respuesta a problemas particulares o máquinas específicas. No es de interés ahora discutir todos los tipos de lenguajes existentes, sino, más bien, conocer aquellos que pueden ser interpretados por computadores, *smartphones* o que pueden ser empleados para proveer servicios informáticos a través de internet y su clasificación más general.

Un lenguaje de programación es diferente al lenguaje de códigos que puede entender la máquina (lenguaje de máquina). Los lenguajes de programación, pueden dividirse en dos categorías: lenguajes interpretados o lenguajes compilados, a saber:

Tabla 1

Lenguajes de programación

Compilado	Interpretado
Definición	
El código creado debe traducirse de manera que el procesador del computador o <i>smartphone</i> pueda comprenderlo, a este proceso se le llama compilación.	Un programa escrito en un lenguaje interpretado, necesita de otro programa auxiliar (el intérprete), que traduce los códigos fuentes de un programa y los interpreta para la máquina.
El lenguaje compilado necesita que antes de ser ejecutada una compilación se convierta el código fuente escrito a un lenguaje de máquina.	Un lenguaje interpretado es convertido a lenguaje de máquina cada vez que es ejecutado (para ello requiere del intérprete), que traduce las instrucciones.
Características	



Para un lenguaje compilado luego de compilar un programa se debe crear ejecutables para cada uno de los sistemas operativos en los que lo vaya a utilizar. Es decir, un programa compilado en Linux no servirá en Windows o en Mac OS.	Por lo general, en el ciclo de desarrollo de <i>software</i> (tiempo entre el momento en que escribe el código y lo prueba) es más corto en un lenguaje interpretado dado que no se necesita realizar el proceso de compilación, cada vez que cambia el código fuente.
Cuando se está ejecutando un programa escrito en un lenguaje compilado es mucho más rápido que uno interpretado, dado a que se encuentra en código de máquina.	Un lenguaje interpretado para ser ejecutado, debe tener instalado el interpretador.
Los lenguajes compilados están optimizados para el momento de la ejecución, pero esto significa más trabajo para el programador (compilarlo).	El código fuente en un lenguaje interpretado está optimizado para hacerle la vida más fácil al programador, aunque esto signifique una carga adicional de procesamiento para la máquina.

JavaScript, al igual que cualquier otro lenguaje de programación, tiene algunas características especiales: sintaxis, modelo de datos, etc. JavaScript **es un lenguaje esencial para cualquier principiante de programación y también para quienes ya saben programar**, debido a que las diferencias con otros lenguajes de programación son numerosas siendo un lenguaje necesario para el desarrollo de aplicaciones web y móviles hoy en día.

En adelante, JavaScript es el lenguaje de programación abordado y es un lenguaje interpretado que inicialmente fue creado para que los navegadores lo interpreten y hoy en día muchos dispositivos tienen el intérprete que permite correr programas escritos en él **por lo que será más sencillo el trabajo tanto para la máquina como para el programador**.

2. Entornos de codificación e instalación JavaScript

En su mayoría los navegadores (Chrome, Mozilla, Edge) tienen intérprete de JavaScript y adicional se puede instalar el intérprete para el sistema operativo de preferencia (Framebits, 2020).

Dado lo anterior, existen herramientas, manuales o videos que se emplean para el desarrollo de programas en JavaScript y a continuación se comparten recursos de acceso y material audiovisual de apoyo de cada herramienta, tal como se indica en la siguiente tabla.

Tabla 2
Entornos de codificación

Herramienta	Recurso Web	Uso y/o instalación
Editor y compilador en línea PlayCode	https://playcode.io/	Woodrow, J (2018) How to use playcode.io. https://www.youtube.com/watch?v=9rtsybc4KZI
Intérprete de JavaScript: Node js	https://nodejs.org/es/	Framebits. (19 de enero. 2020). Descargar e instalar Node Js en Windows 10. Youtube. https://youtu.be/v0x1Ku5Tgac
Editor de código Visual Studio Code	https://code.visualstudio.com/	Code Compadre. (30 de junio. 2020). How to Download and Install Visual Studio Code for Windows 10. Youtube. https://youtu.be/KpzKPIh_HsU
El navegador: Google Chrome	https://www.google.com/chrome/	Choque, R. C. (2020, May 10). Cómo usar la consola de google chrome para Javascript. Youtube. Retrieved from https://www.youtube.com/watch?v=Hf3n-p3VYx4

3. Sintaxis del lenguaje JavaScript



En adelante, se presentan las características básicas de la sintaxis de JavaScript, codificando algoritmos básicos, para ejemplificar la estructura propia del lenguaje de programación, sin embargo, se recomienda que siempre que se esté codificando algoritmos, se pueda tener a disposición la documentación técnica oficial del lenguaje y material complementario propuesto.

3.1. Funciones y operaciones matemáticas

En JavaScript se pueden utilizar muchas funciones matemáticas; por ejemplo, un algoritmo que toma dos números (8 y 6) y le aplica las 4 operaciones básicas y en el editor de código online PlayCode (disponible en <https://playcode.io/>) se revisan las características de la sintaxis del JavaScript y a la vez se conocen las operaciones matemáticas del lenguaje.

Ingresa en la ruta del navegador a <https://playcode.io/new/>, escribir el código de la línea 1 a la 12 en la pestaña script.js como se muestra en la siguiente figura:

Figura 3

Operaciones aritméticas con JavaScript

```
index.html x styles.css x script.js x

1  let numero1 = 8;
2  let numero2 = 6;
3
4  var suma = numero1 + numero2;
5  var resta = numero1 - numero2;
6  var producto = numero1 * numero2;
7  var division = numero1 / numero2;
8
9  console.log("la suma es: " + suma );
10 console.log("la resta es: " + resta );
11 console.log("el producto es: " + producto );
12 console.log("la división es: " + division );

CONSOLE x
la suma es: 14
la resta es: 2
el producto es: 48
la división es: 1.3333333333333333
```

Explicación: en las líneas 1 y 2 aparece la palabra reservada **let** que sirve para indicar que se define una variable accesible en el contexto donde se crea (cuando se vea programación modular quedará más claro esto del contexto).

```
let numero1 = 8;
let numero2 = 6;
```



Las palabras reservadas **numero1** y **numero2**, se les asigna el valor correspondiente. Es de observar que se definen y se inicializa la variable en una misma línea, esto es una característica del lenguaje.

```
var suma = numero1 + numero2;  
var resta = numero1 - numero2;
```

Luego en las líneas 4 al 7 se crean o se definen unas variables con la palabra reservada **var**, a diferencia de la palabra **let** estas variables se pueden ver desde cualquier parte o módulo del script.

```
var producto = numero1 * numero2;  
var division = numero1 / numero2;
```

Una vez se crean las variables con identificadores **suma**, **resta**, **producto**, **division**, también son inicializados sus valores empleando las operaciones matemáticas respectivas.

Una de las formas que JavaScript permite mostrar los resultados es con la función **log** del objeto **console** y se accede **console.log()** (con el operador punto **.**) y entre paréntesis lo que se desea mostrar en la pantalla de la consola.

```
console.log("la suma es: " + suma);  
console.log("la resta es: " + resta);
```

Validación: dentro de los paréntesis `"la suma es: " + suma`, aparece la operación más + usado para sumar dos números antes, solo que acá no hay dos números sino una cadena de texto y un número, pero en este caso lo que hace el operador + es concatenar o juntar el texto "la suma es:" y el valor de la variable suma. A esta característica de los operadores se les llama sobrecarga de operadores, porque según el tipo de datos suma y según otra concatena.

Cuando las operaciones matemáticas son más complejas como la raíz cuadrada, la función exponencial, las funciones trigonométricas, existe un objeto que agrupa estas funciones se llama **Math**, puede ver la documentación técnica de la clase completa en Math (MDN, 2021b), a continuación, un ejemplo de su uso.

Se debe crear un algoritmo que muestre el seno del valor PI (3,1416) multiplicado por 3 la función, y también que nos eleve 6 a la tercera potencia mostrando su resultado:

Figura 4

Uso del objeto Math

```
Index.html x styles.css x script.js x  
1 var angulo = Math.PI * 3;  
2 console.log( 'seno(3PI) =' + Math.sin(angulo))  
3  
4 var potencia = Math.pow(6,3);  
5 console.log( '6^3 =' + potencia)  
  
CONSOLE x  
seno(3PI) =3.6739403974420594e-16  
6^3 =216
```



Se debe escribir el código fuente de las figuras 3 y 4, y obtener los mismos resultados en la consola, como se muestra en cada imagen, de esta forma se familiariza con la herramienta de trabajo.

A continuación, puede revisar el listado de funciones de la clase Math.



Tabla 1

Métodos del objeto Math

Nota: Tomada de MDN (2021b)

Todas las funciones matemáticas se pueden combinar con otras operaciones aritméticas o emplear operadores de evaluación de condición y previo a ello es preciso conocer los diferentes tipos de datos que existen.

3.2. Tipos de datos, operadores y orden de evaluación

Existen muchos tipos de operadores para JavaScript, cuya función es realizar una operación entre dos o más valores contenidos en variables, constantes o acumuladores:

A. Tipos de datos

JavaScript tiene los siguientes tipos de operadores (se describirán los principales para aprender a programar).

Según sea el tipo que las variables o constantes almacenan, se pueden clasificar en seis (6) tipos de datos primitivos la referencia del lenguaje dice que son:

- **Undefined:** indeterminado o indefinido
- **Boolean:** tipo booleano los valores posibles son *true* o *false*.
- **Number:** números enteros, o decimales.
- **String:** cadenas de texto.
- **BigInt:** números enteros grandes.
- **Symbol:** referencia a otros datos

Otros tipos de datos más complejos o abstractos de datos:

- **Null:** tipo primitivo especial que tiene un uso adicional para su valor: si el objeto no se hereda, se muestra *null*.
- **Object:** tipo estructural especial que no es de datos, pero para cualquier instancia de objeto construido que también se utiliza como estructuras de datos (*new Object*, *new Array*, *new Map*, *new Set*, *new WeakMap*, *new WeakSet*, *new Date* y casi todo lo hecho con la palabra clave *new*).
- **Function:** una estructura sin datos, aunque también responde al operador *t*.

Cuando se declaran variables se debe considerar los siguientes tipos, ya se han usado dos (2) de ellas:

- **var:** declara una variable, opcionalmente la inicia a un valor.
- **let:** declara una variable local con ámbito de bloque, opcionalmente la inicia a un valor.
- **const:** declara un nombre de constante de solo lectura y ámbito de bloque.

Para ampliar la información de acuerdo con los tipos de datos y estructuras, se debe revisar el material complementario dispuesto:



Tipos de datos y estructuras en JavaScript. (MDN, 2021e).
https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Grammar_and_Types#tipos_de_datos



Una variable en JavaScript puede tener cualquier tipo de dato, entero, real, cadena de texto etc. JavaScript no es un lenguaje fuertemente tipado de manera que queda en el programador la responsabilidad de saber qué tipo de dato está almacenando en cada variable.

B. Operadores

Se considera una expresión, aunque de un tipo distinto, el asignar un valor a una variable. Para todas estas operaciones se emplean los denominados '**operadores**'. Se listan a continuación algunos de los operadores que se utilizan con mucha frecuencia.

❖ Operadores de asignación

Un operador de asignación asigna un valor a la variable a la izquierda basándose en el valor de su operando derecho. El operador de asignación más simple es igual (=), que asigna el valor de su operando derecho a su operando izquierdo. Es decir, $x = y$ asigna el valor de y a x . A continuación, se presenta una lista de operadores más usados.

También hay operadores de asignación compuestos que son una abreviatura de las operaciones enumeradas en la siguiente tabla:

Tabla 2

Operadores de asignación básicos

Nombre	Operador abreviado	Significado
Asignación	$x = y$	$x = y$
Asignación de adición	$x += y$	$x = x + y$
Asignación de resta	$x -= y$	$x = x - y$
Asignación de multiplicación	$x *= y$	$x = x * y$
Asignación de división	$x /= y$	$x = x / y$
Asignación de residuo	$x \% = y$	$x = x \% y$
Asignación de exponenciación	$x ** = y$	$x = x ** y$

Nota: Tomada de MDN (2021a)

❖ Operadores de comparación

Son operadores cuya función es comparar dos expresiones y como resultado de la comparación devuelven un valor falso o verdadero (booleano), que representa la relación de sus valores comparados. Existen operadores para comparar valores numéricos, pero también operadores para comparar cadenas y operadores para comparar otros tipos de datos (tabla 3).

Tabla 3

Operadores lógicos

Operador	Descripción	Ejemplos que devuelven true
----------	-------------	-----------------------------



Igual (==)	Devuelve true si los operandos son iguales.	3 == var1 "3" == var1 3 == '3'
No es igual (!=)	Devuelve true si los operandos no son iguales.	var1 != 4 var2 != "3"
Estrictamente igual (===)	Devuelve true si los operandos son iguales y del mismo tipo. Consulta también Object.is y similitud en JS.	3 === var1
Desigualdad estricta (!==)	Devuelve true si los operandos son del mismo tipo, pero no iguales, o son de diferente tipo.	var1 !== "3" 3 !== '3'
Mayor que (>)	Devuelve true si el operando izquierdo es mayor que el operando derecho.	var2 > var1 "12" > 2
Mayor o igual que (>=)	Devuelve true si el operando izquierdo es mayor o igual que el operando derecho.	var2 >= var1 var1 >= 3
Menor que (<)	Devuelve true si el operando izquierdo es menor que el operando derecho.	var1 < var2 "2" < 12
Menor o igual (<=)	Devuelve true si el operando izquierdo es menor o igual que el operando derecho.	var1 <= var2 var2 <= 5

Nota: Tomada de MDN (2021a)

❖ Operadores de aritméticos

Los operadores aritméticos toman valores numéricos (ya sean literales o variables) como sus operandos y devuelve un solo valor numérico. Los operadores aritméticos básicos son suma (+), resta (-), multiplicación (*) y división (/). Estos operadores funcionan como en la mayoría de los otros lenguajes de programación cuando se usan con números de punto flotante, teniendo en cuenta que la división por cero produce un error (tabla 4).

Tabla 4

Operadores aritméticos

Operador	Descripción	Ejemplo
Residuo (%)	Operador binario. Devuelve el resto entero de dividir los dos operandos.	12 % 5 devuelve 2.
Incremento (++)	Operador unario. Agrega uno a su operando. Si se usa como operador prefijo (++x), devuelve el valor de su operando después de agregar uno; si se usa como operador sufijo (x++), devuelve el valor de su operando antes de agregar uno.	Si x es 3, ++x establece x en 4 y devuelve 4, mientras que x++ devuelve 3 y, solo entonces, establece x en 4.
Decremento (--)	Operador unario. Resta uno de su operando. El valor de retorno es análogo al del operador de incremento.	Si x es 3, entonces --x establece x en 2 y devuelve 2, mientras que x-- devuelve 3 y, solo entonces, establece x en 2.
Negación unaria (-)	Operador unario. Devuelve la negación de su operando.	Si x es 3, entonces -x devuelve -3.
Positivo unario (+)	Operador unario. Intenta convertir el operando en un número, si aún no lo es.	+"3" devuelve 3. +true devuelve 1.



Operador de exponenciación (* *)	Calcula la base a la potencia de exponente, es decir, base exponente	2 ** 3 retorna 8. 10 ** -1 retorna 0.1.
-----------------------------------------	----------------------------------------------------------------------	--------------------------------------------

Nota: Tomada de MDN (2021a)

❖ Operadores lógicos

Con los operadores lógicos, se pueden crear condiciones compuestas, por ejemplo, cuando se deben cumplir dos o más condiciones para elegir las operaciones ejecutar; además, se pueden describir estas combinaciones de condiciones (tabla 5 y figura 5).

Tabla 5

Operadores lógicos

Operador	Uso	Descripción
AND Lógico (&&)	expr1 && expr2	Devuelve expr1 si se puede convertir a false; de lo contrario, devuelve expr2. Por lo tanto, cuando se usa con valores booleanos, && devuelve true si ambos operandos son true; de lo contrario, devuelve false.
OR lógico ()	expr1 expr2	Devuelve expr1 si se puede convertir a true; de lo contrario, devuelve expr2. Por lo tanto, cuando se usa con valores booleanos, devuelve true si alguno de los operandos es true; si ambos son falsos, devuelve false.
NOT lógico (!)	!expr	Devuelve false si su único operando se puede convertir a true; de lo contrario, devuelve true.

Nota: Tomada de MDN (2021a)

Figura 5

Ejemplos de operadores lógicos

```

index.html × styles.css × script.js ×
1 console.log( true && true );
2 console.log( true && false);
3 console.log( false && true );
4 console.log( false && (3 == 4) );
5 console.log( 'Cat' && 'Dog');
6 console.log( false && 'Cat');
7 console.log('Cat' && false );

CONSOLE ×
true
false
false
false
Dog
false
false

```




Se han presentado los operadores más comunes, aunque existen otros tipos de operadores que tal vez no son los más empleados y una referencia completa la puede obtener desde el recurso web denominado *Expresiones y operadores* (MDN, 2021a).



La lista completa de los principales operadores de JavaScript se puede revisar en *Expresiones y operadores* (MND, 2021a).
https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Expressions_and_Operators

C. Orden de evaluación de operadores

Es importante recordar que los computadores ejecutan los operadores en un orden establecido. El siguiente es el orden (jerarquía), de acuerdo con los expuestos anteriormente, y deben conservar el orden de precedencia de operadores:

1. Paréntesis (se ejecutan primero los más internos).
2. Signo (-2), si un valor es positivo o negativo.
3. Potencias (^) y Raíces y otras funciones (Math.sqrt()); Productos y Divisiones (* y /), en este mismo orden.
4. Sumas y Restas (+ y -).
5. Concatenación (+).
6. Relacionales (=, <, >).
7. Negación (no).
8. Conjunción (y).
9. Disyunción (o).

3.3. Expresiones y comentarios

JavaScript permite poner comentarios en el código fuente, como la mayoría de los lenguajes de programación y existen dos tipos de comentarios en línea que comienzan con una doble barra: //, y los comentarios multilínea, que comienzan con /* y terminan con */ tal como se observa en la siguiente figura.

Figura 6

Tipos de comentarios



```
index.html x styles.css x script.js x
1  /*
2   Comentario que puede tener mas de una linea
3   de texto
4   */
5  var var1 = 3;
6  var var2 = 4;
7  console.log( var2 > var1 );    // Este es un comentario de linea
8  console.log( var2 < var1 );    // este es otro comentario de linea
9  console.log( var1 !== var2 );

CONSOLE x
true
false
true
```

Los comentarios son importantes porque le dan mantenibilidad al programa, es decir que otro programador, o el mismo, tiempo después puede revisar el código y apoyarse en los comentarios para saber qué hace el algoritmo y como lo hace.

En adelante se procura que exista al menos un comentario en cada código fuente empleado para ejemplificar los elementos constitutivos del lenguaje de programación JavaScript y puede hacer lo mismo durante la creación de los diferentes códigos.

3.4. Estructuras de selección

La estructura de selección se necesita cuando el código de tu programa ejecuta uno de varios resultados posibles, basado en el valor de una condición.

Ejemplo: un aprendiz aprueba un examen cuando la calificación de este es mayor o igual a 3. Elaborar un programa en JavaScript donde que dada una calificación, aplique el criterio de aprobación e imprima “Aprobado” o “Reprobado”, según sea el caso.

Como resultado del **análisis** a este problema tenemos el diseño del siguiente algoritmo:

Figura 7

Seudocódigo calificaciones.

```
ALGORITMO Calificación del usuario;
VAR
    REAL calificacion
INICIO
    calificacion = 4;
    SI ( calificación >= 3)
```



```

        ESCRIBIR ( "Aprobo" );
SINO
        ESCRIBIR ( "Reprobo" );
FINSI
FIN

```

El mismo algoritmo se puede visualizar, pero ya en lenguaje de programación:

Figura 8

Algoritmo calificaciones en JavaScript

```

1  /**
2   * Algoritmo calificaciones
3   */
4  const calificacion = 4;
5
6  if (calificacion >= 3 ){
7      console.log('Aprobó');
8  }else{
9      console.log('Reprobó');
10 }

```

CONSOLE x

Aprobó

Como la línea 1 empieza con `/*` quiere decir que toda continuación será un comentario hasta que aparezca los caracteres `*/` (línea 3), en él se debe escribir que hace el algoritmo y otra información que se verá más adelante.

En la línea 4 se define un valor constante (es decir no se modificará más en todo el algoritmo donde definimos) se ha puesto por identificador la palabra **calificacion**.

Para recordar: una palabra reservada es una palabra que no se puede utilizar como identificador de algún dato.

En la línea 6 ocurren varias cosas que se deben tener en cuenta:

1. Se usa la palabra reservada **if** y entre paréntesis la condición que la se quiere comparar.
2. Se emplea el símbolo corchete abierto `{` se llama abrir corchete.
3. En la línea 7 se envía la instrucción de que imprima el resultado de que aprobó.
4. En la línea 8 se cierra un bloque de código con el símbolo corchete cerrado `}`, esto quiere decir que hasta este punto termina el bloque. A todo lo que esté entre estos corchetes se le llamará CONTEXTO, en el primer contexto están agrupadas las operaciones que se ejecutarán si la condición **calificacion >= 3** es verdadera.
5. También en la línea 8 se observa la palabra reservada **else**, que sirve para indicarle al intérprete de JavaScript que se creará un bloque o contexto para cuando la condición evaluada por el **if** no se cumpla, y enseguida se abre el contexto con el carácter corchete abierto `{`.
6. Luego, en la línea 9 se ha puesto la operación de escribir en la consola que ha reprobado.
7. Para finalizar el contexto, se cierra con el carácter corchete cerrado `}`.

Ahora se debe escribir el programa y cambiar los valores de la constante para validar que el algoritmo está bien diseñado.

De la misma forma, se puede realizar el algoritmo que, dada una edad en años, evalúe e imprima si es mayor de edad:

**Figura 9**

Escribir en pantalla si es o no mayor de edad

```

index.html x styles.css x script.js x
1  /**
2   * Algoritmo saber si es o no
3   * mayor de edad
4   */
5  const edad = 41;
6
7  if (edad >= 18 ){
8      console.log('Es mayor de edad');
9  }

```

CONSOLE x

Es mayor de edad

De acuerdo con la figura 9, no existe un contexto si no se cumple la condición de mayoría de edad a esta estructura se denomina **CONDICIONAL SIMPLE**.

El flujo de ejecución del algoritmo anterior es básico y puede haber otros mucho más complejos como las estructuras de repetición o iterativas que también tienen una sintaxis particular.

3.5. Estructuras de repetición

Las estructuras de repetición permiten repetir un bloque de instrucciones, determinado o no, un número de veces. A continuación, se ejemplifican los más comunes (*FOR*, *WHILE*) que se pueden codificar también en la sintaxis de JavaScript.

A. Estructura de repetición FOR

Ejemplo 1: escribir un procedimiento que muestre siete (7) veces en pantalla la frase “Esto es un algoritmo”.
Mostrar en pantalla la salida del ejercicio:

Figura 9

Imprimir 7 veces un mensaje

```

index.html x styles.css x script.js x
1  /**
2   * Algoritmo imprimir 7 veces un mens
3   */
4  for (let i = 0; i < 7; i++) {
5      console.log('esto es un algoritmo')
6  }

```

CONSOLE x

esto es un algoritmo
esto es un algoritmo
esto es un algoritmo
esto es un algoritmo
esto es un algoritmo
esto es un algoritmo
esto es un algoritmo

Como se puede observar en la línea 4 de la figura 9, se ha usado la palabra reservada **for** y dentro del paréntesis aparecen tres (3) instrucciones (separadas por punto y coma).

La primera instrucción **let i = 0;** es la creación e inicialización de una variable con identificador **i**, como se usó la palabra reservada **let** se está indicado que la variable **i** solo podrá ser accedida dentro del contexto del **for**.

Luego se ve una condición **i < 7;**, esta condición significa que mientras que **i** tenga valores de 0 a 6 debe ejecutar el bloque o contexto definido en el **for** (desde el corchete abierto hasta el corchete cerrado).

Una vez ha terminado de ejecutar el contexto (línea 6) se ejecutará la tercera sentencia de la línea 7 o sea **i++**, esta línea es equivalente a **i = i + 1**; incrementar en 1 la variable **i**, luego de incrementarla procede a evaluar nuevamente la condición y si se cumple volverá a ejecutar las instrucciones que están dentro el contexto (lo que está dentro de paréntesis).



	Es de notar que los valores toman la variable <i>i</i> son 0, 1, 2, 3, 4, 5, 6. En total 7 valores el número de veces que se necesita que se repita el bucle.
Para recordar: el carácter punto y coma se usa para decirle al computador, o intérprete, que ha finalizado una instrucción. La estructura de repetición PARA (FOR) se usa cuando se sabe cuántas veces se debe repetir un grupo de instrucciones.	

Ejemplo 2: se necesita elaborar un algoritmo en JavaScript que, dado un número entero, sume todos los números naturales que hay hasta ese número. Por ejemplo, si el usuario digita 3, el programa debe sumar: 1 + 2 + 3, si el usuario digita 5 el programa debe sumar 1+2+3+4+5. Al finalizar, debe imprimir el resultado.

Figura 10
Sumar los números desde 1 hasta un número

```

1  /**
2   * sumar los numero que dede 1 hasta el
3   * valor dado
4   */
5
6  const cantidad = 5;
7  var suma = 0;
8
9  for (let i = 1; i <= cantidad; i++) {
10     suma += i;    // esto es igual suma = suma + i;
11  }
12  console.log(suma);

```

CONSOLE x

15

Como se puede ver, se decide una constante con valor 5, y una variable que será para el algoritmo un acumulador donde se almacenarán las sumas. También, note que los valores de *i* van desde 1 hasta exactamente el valor que tiene la constante (es decir 5).

B. Estructura de repetición WHILE

A través del ejemplo del cálculo de factorial se explicará el uso y comportamiento de la sentencia **while**. Recordando el algoritmo diseñado:

Figura 10

Seudocódigo algoritmo cálculo de factorial

```

ALGORITMO  cálculo de factorial;
VAR
    ENTERO  contador;
    ENTERO  factorial;
INICIO
    contador = 1;
    factorial = 1;
    ESCRIBIR( "Escriba el número" );
    LEER(  numero );
    MIENTRAS  contador <= numero  HACER

```



```

    factorial = factorial * contador;
    contador = contador + 1;
FINMIENTRAS
ESCRIBIR( factorial );
FIN

```

De acuerdo con el anterior algoritmo, el resultante para el cálculo de factorial es:

Figura 11

Seudocódigo algoritmo cálculo de factorial

```

index.html x styles.css x script.js x
1 // Cálculo del factorial
2 const numero = 5;
3 var contador = 1;
4 var factorial = 1;
5
6 while (contador <= numero){
7     factorial = factorial * contador; // se puede abreviar factorial *= contador;
8     contador ++; // es equivalente a contador = contador + 1;
9 }
10 console.log(factorial);

```

CONSOLE x

120

Es ideal poder transcribir el código usando <https://playcode.io/new/>, para lo cual se debe escribir la condición del **while**, es decir, **contador <= numero** de último, para que el sistema PlayCode no entre a un ciclo que nunca termina. Una vez funcione correctamente, se propone el código que está entre comentarios (//) para practicar las dos formas de sintaxis.

Es de recordar que las estructuras de repetición básicas y su sintaxis pueden pasar su aplicación más común para recorrer la estructura de datos.

3.6. Estructuras de datos

Las estructuras de datos también conocidas como arreglos, son importantes en el desarrollo de algunos algoritmos son obligatorias a la hora de hacer aplicaciones web o móviles; por ello, se explica su sintaxis en JavaScript para los vectores, matrices y registros.

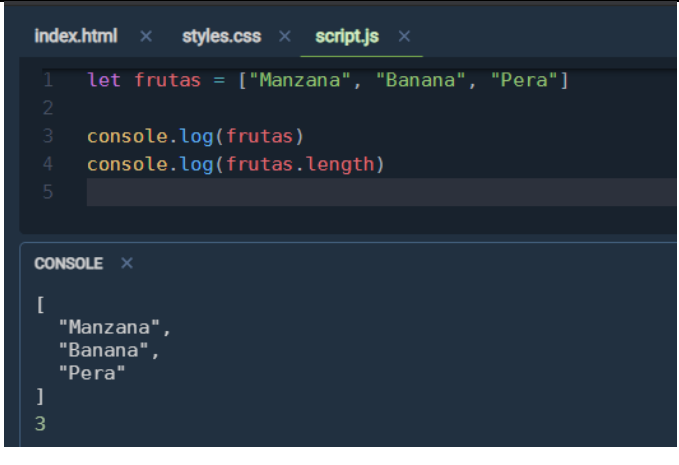
A. Vectores

Los vectores se definen en JavaScript como un tipo de dato con métodos, y atributos que lo definen. Es ideal analizar lo siguiente:

Figura 12
Definición de un vector

En la línea 1 se ve cómo se define un vector con identificador **frutas**, de tres (3) elementos, cada uno de ellos es una cadena de texto con una fruta.



 <pre>1 let frutas = ["Manzana", "Banana", "Pera"] 2 3 console.log(frutas) 4 console.log(frutas.length) 5</pre> <p>CONSOLE</p> <pre>["Manzana", "Banana", "Pera"] 3</pre>	<p>En la línea 3 se manda imprimir en la consola todo el vector con su contenido, como se ve en la consola.</p> <p>En la línea 4 se le llama al atributo frutas.length que contiene el tamaño del vector (es decir 3).</p>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

❖ Acceder a los elementos de un vector

Los índices de los vectores se comienzan en cero, en otras palabras, el índice del primer elemento de un vector es 0, y el del último elemento es igual al valor de la propiedad *length* del *array* restándole 1. Si se utiliza un número de índice no válido, se obtendrá **undefined**.

Figura 13

Indexar un vector



```
1 let arreglo = [5, 7, 23, 12, 43];
2
3 console.log( arreglo[0]);
4 console.log( arreglo[2]);
5 console.log( arreglo[ arreglo.length -1]);
6 console.log( arreglo[arreglo.length]);
7
```

CONSOLE

```
5
23
43
undefined
```

Ahora, se debe crear un programa que guarde los resultados de la tabla de 5 en un vector, con estos resultados recorrer el vector e imprimir la tabla del 5:

Figura 15

Crear la tabla del 5



```

Index.html x styles.css x script.js x
1 let resultados = [];
2
3 for(let i =1; i<=10; i++){
4     resultados.push( i * 5);
5 }
6
7 for(let i =1; i<=10; i++){
8     console.log(' 5 x '+ i +' = ' + resultados[i-1]);
9 }

```

CONSOLE x

```

5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50

```

Es ideal revisar la lista de los métodos más comunes en los arreglos:

Tabla 6

Funciones comunes de un vector

Función	Descripción
indexOf()	Devuelve el índice del primer elemento del <i>array</i> que sea igual a elemento Buscado, o -1 si no existe.
join()	Concatena en un <i>string</i> todos los elementos de un <i>array</i> .
push()	Añade uno o más elementos al final de un <i>array</i> y devuelve el nuevo valor de su propiedad <i>length</i> .
pop()	Elimina el último elemento de un <i>array</i> , y devuelve dicho elemento.
sort()	Ordena los elementos de un <i>array</i> , modificando este, y devuelve el <i>array</i> ordenado.
shift()	Elimina el primer elemento de un <i>array</i> , y devuelve dicho elemento.



Finalmente, desde el material complementario se encuentra una lista completa o accediendo desde Array - JavaScript (MDN, 2021c).
<https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Functions>

B. Matrices

Las matrices se pueden ver lógicamente como un vector, y cada uno de sus elementos es otro vector, así la representación de la matriz queda codificada como muestra la figura 16:

**Figura 16**

Crear la tabla del 5

"Mazana"	"Pera"	"Papaya"	"Piña"
"Papa"	"Tomate"	"Yuca"	"Ajo"
"Arroz"	"Frijol"	"Cebada"	"Garbanzo"

Matriz de 3x3

```

index.html x styles.css x script.js x
1 let datos = [
2   ["Mazana", "Pera", "Papaya", "Piña"],
3   ["Papa", "Tomate", "Yuca", "Ajo"],
4   ["Arroz", "Frijol", "Cebada", "Garbanzo"]
5 ];
6 console.log(datos);

```

C. Registros

Hasta ahora se ha visto cómo un arreglo es la colección de datos del mismo tipo, pero un registro es una colección de datos de diferente tipo que se relacionan entre sí:

Tabla 7

Registros

Nombre	Correo	Edad	Saldo
Juan	juan@sindato.com	19	36.234
Luis	luis@sindato.com	18	23.234
Andrea	andrea@sindato.com	22	0
Pedro	pedro@sindato.com	12	65.234
Maria	maria@sindato.com	16	123

Exceptuando el encabezado de la tabla 7, cada fila representa un registro y cada dato de un registro puede tener un tipo de dato diferente. Si se está interesado en codificar la tabla 7, en una estructura con JavaScript el código sería como se tiene en la figura 17:

Figura 17

Registros

```

index.html x styles.css x script.js x
1 let registros =
2 [
3   ["Juan", "juan@sindato.com", 19, 36.234],
4   ["Luis", "luis@sindato.com", 18, 23.234],
5   ["Andrea", "andrea@sindato.com", 22, 0],
6   ["Pedro", "pedro@sindato.com", 12, 65.234],
7   ["Maria", "maria@sindato.com", 16, 123]
8 ];

```

Para recorrer todos los registros se precisan dos (2) ciclos **for**, uno dentro de otro (anidado) como se muestra a continuación:

Figura 18

Recorrer registros



```
Index.html x styles.css x script.js x
1 let registros = [ ["Juan", "juan@sindato.com", 19, 36.234],
2   ["Luis", "luis@sindato.com", 18, 23.234],
3   ["Andrea", "andrea@sindato.com", 22, 0],
4   ["Pedro", "pedro@sindato.com", 12, 65.234],
5   ["Maria", "maria@sindato.com", 16, 123] ];
6
7 var total = 0;
8
9 for(let i=0; i < registros.length; i++){
10  console.log("Registro # " + (i+1) );
11  for(let j=0; j< registros[i].length; j++){
12    console.log( "      " +registros[i][j] );
13  }
14 }
```

```
CONSOLE x
Registro # 1
  Juan
  juan@sindato.com
  19
  36.234
Registro # 2
  Luis
  luis@sindato.com
  18
  23.234
Registro # 3
  Andrea
  andrea@sindato.com
  22
  0
```

Como se puede observar en la figura 18, se requieren dos ciclos **for** anidados, y cada ciclo tiene una variable de índice de nombre diferente **i** y **j** y la manera de indexar la matriz vista anteriormente en el ejemplo de **registros [i][j]**.

3.7. Estructuras de salto

Estas estructuras son instrucciones que permiten romper la ejecución natural o secuencial de los programas, permitiendo que se salte a otro punto de la ejecución del programa, estas instrucciones tienen las palabras reservadas (*continue*, *break* y *return*):

- **Sentencia *continue***

La sentencia ***continue*** se salta a la siguiente iteración del ciclo *for* o *while* como se presenta en el siguiente ejemplo y teniendo en cuenta los resultados.

**Figura 19***Sentencia continue*

```
index.html x styles.css x script.js x
1  var i = 0;
2
3  while (i < 5) {
4      i++;
5
6      console.log(i)
7  }
8
CONSOLE x
1
2
3|
4
5
```

```
index.html x styles.css x script.js x
1  var i = 0;
2
3  while (i < 5) {
4      i++;
5      if(i == 3){
6          continue;
7      }
8      console.log(i)
9  }
```

```
CONSOLE x
1
2
4
5
```

Como se observa, cuando la variable *i* tiene el valor 3, se ejecuta la sentencia **continue** se salta la sentencia donde debe imprimir el valor de 3, por ese motivo no aparece en la consola.

- **Sentencia break**

Por otro lado, la sentencia *break*, detiene la ejecución del ciclo independientemente de cuántas veces este configurando el ciclo. Mire el siguiente ejemplo en la Figura 20, prestando especial atención en la salida de consola.

Figura 20*Sentencia break*

```
index.html x styles.css x script.js x
1
2  for (let i = 1; i <= 5; i++) {
3      console.log(i);
4  }
5
CONSOLE x
1
2
3
4
5
```

```
index.html x styles.css x script.js x
1
2  for (let i = 1; i <= 5; i++) {
3      if(i == 3){
4          break;
5      }
6      console.log(i);
7  }
```

```
CONSOLE x
1
2|
```



En la anterior figura, cuando *i* tiene el valor de 3 ya no se sigue ejecutando más el ciclo, aunque esté programado para ejecutarse 5 veces, solo ejecuta 3 iteraciones y, ya que en el tercero luego de evaluar la condición *i* == 3 ejecuta la sentencia **break** haciendo que termine todo.

- **Sentencia *return***

Sirve para terminar la ejecución de un bloque de instrucciones, se usa mucho en programación modular cuando se quiere retornar un resultado, se usará más adelante.

Por ejemplo, se tiene una función que divide un número entre 4 y se ha creado para obtener resultados. En el primer caso se le pasa 27 para que lo divida entre 4 pero como no hay *return* dentro, devuelve “*undefined*”. No accede al número porque el número solo vive dentro de la función y no hay nada que dé el valor. Si se le agrega *return*, efectivamente se obtiene el resultado de la división.

Figura 21

Sentencia return.

<pre> > var divideByFour = function (num) { num / 4; } var divided = divideByFour(27); < undefined > divided < undefined > </pre> <p>Sin return</p>	<pre> > var divideByFour = function (num) { return num / 4; } var divided = divideByFour(27); < undefined > divided < 6.75 </pre> <p>Con return</p>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------

Nota. Tomada de Vedia (2018)

3.8. Métodos de ordenamiento y búsqueda

Existen varias técnicas de ordenamiento de vectores las cuales son importantes para realizar búsquedas y, al igual que en la vida real, es más fácil buscar en un lugar que esté ordenado que en uno que no lo esté, como JavaScript está diseñado para pequeños algoritmos que resuelven pequeños problemas y convertirlos en servicios, el uso de esas técnicas no tiene mucha aplicabilidad, pues la diferencia en tiempos computacionales de ejecución es considerable solo cuando se manejan volúmenes de datos grandes, mientras que para volúmenes de datos cortos o pequeños los tiempos computacionales no tienen diferencias comparables. Es por eso que, si en JavaScript el volumen de datos en los arreglos no será grande, no es necesario la aplicación de esas técnicas.

JavaScript provee un *stack* de funciones básicas para realizar el ordenamiento y búsqueda que es muy útil para la implementación de servicios basados en algoritmos.

A. Ordenamiento de cadenas de texto

Se ordenará un vector de mayor a menor valor y de acuerdo con los datos que tiene, se debe tener en cuenta que el arreglo se modifica al ordenarlo, devolviendo la misma matriz ordenada, pero no una nueva. Esto es importante, si se quiere mantener inmutable el vector y obtener otro ordenado, lo que se tendría que hacer es una copia del arreglo antes de ordenarlo.



Por defecto el método `Array.sort()` ordena los elementos del `array` como si fueran cadenas inclusive si los datos son de tipo entero, como en este ejemplo:

Figura 22

Ordenamiento de números como cadenas de texto

```
index.html × styles.css × script.js ×  
1 var arg = [10, 53, 12, 56, 120];  
2 console.log(arg.join(', '));  
3 arg.sort();  
4 console.log(arg.join(', '));  
  
CONSOLE ×  
10, 53, 12, 56, 120  
10, 12, 120, 53, 56
```

La figura 22 muestra el arreglo a ordenar en la línea 1, en la línea 2, se muestra en consola el vector, se ha usado la función `join(', ')` que sirve para concatenar los datos del vector o arreglo poniendo entre ellos una coma (,) y espacio (línea 2). Luego se aplica la función `sort()` en la tercera línea de código y, posteriormente, se muestra el vector ya ordenado. Como se observa, el resultado en consola es un ordenamiento como si los elementos fueran cadenas de texto.

Figura 23

Ordenamiento de cadenas de texto

```
index.html × styles.css × script.js ×  
1 var data = [ "Montería", "Sincelejo", "Cartagena", "Bogotá" ];  
2 data.sort();  
3 console.log (data);  
  
CONSOLE ×  
[  
  "Bogotá",  
  "Cartagena",  
  "Montería",  
  "Sincelejo"  
]
```



El ejemplo de la figura anterior muestra un ordenamiento de cadenas de texto. También podemos ordenar el arreglo de forma descendente con el método ***Array.reverse()*** como se muestra en la figura 24.

Figura 24

Invertir el orden de los elementos

```
index.html × styles.css × script.js ×  
1 var data = [ "Montería", "Sincelejo", "Cartagena", "Bogotá" ];  
2 data.sort().reverse();  
3 console.log (data);  
  
CONSOLE ×  
[  
  "Sincelejo",  
  "Montería",  
  "Cartagena",  
  "Bogotá"  
]
```

La función ***Array.reverse()*** no ordena los elementos, simplemente toma los elementos y les invierte el orden, es decir, el primero pasa a ser el último, el segundo, el penúltimo y así hasta que el último pasa al primer puesto, es por esto que primero se ordena en orden ascendente y luego se invierte para lograr el orden descendente.

B. Ordenamiento de datos numéricos

Como se ve el ordenamiento con la función ***Array.sort()*** se hace considerando los elementos del arreglo como cadenas de texto. Antes de ver cómo se ordenan con datos numéricos se explicará que es una función en *JavaScript*.

Las funciones son uno de los bloques de construcción fundamentales en *JavaScript*. Una función en *JavaScript* es similar a un procedimiento o “un conjunto de instrucciones que realiza una tarea o calcula un valor, pero para que un procedimiento califique como función, debe tomar alguna entrada y devolver una salida donde hay alguna relación obvia entre la entrada y la salida”. Para usar una función, se debe definir en algún lugar del ámbito en el que se desea llamarla (MDN, 2021d).

Ejemplo: se quiere construir una función que reciba dos (2) datos numéricos, hay que multiplicar estos dos números, al resultado se le suma el valor de 100 y, por último, retorna la mitad de este resultado.

Figura 25

Definición de una función



```
index.html × styles.css × script.js ×  
1 function operacion(a, b){  
2   |   | return ( a * b + 100) /2;  
3 }  
4 var dato1 = operacion(3, 8);  
5 var dato2 = operacion(2, 4);  
6 console.log(dato1);  
7 console.log(dato2);  
  
CONSOLE ×  
  
62  
54
```

Como se puede observar en la línea 1 de la figura 25, se usa la palabra reservada **function** y luego se escoge identificador para la función que el caso es **operacion**, luego entre paréntesis se seleccionan nombres para las variables de entrada (en el ejemplo **a**, y **b**), a continuación, se define un contexto (espacio de código entre corchetes) para indicar cuáles son las instrucciones que ejecutará la función. En la línea 2 se muestra que la función retorna la multiplicación de $a * b$ sumando 100 y a este resultado dividido entre 2. En la línea 4 se ve como invocamos la función **operacion**, y le pasamos los parámetros 3 y 8 para las variables **a** y **b** respectivamente, el resultado es almacenado en la variable **dato1**. Y como se observa la función puede invocarse más de una vez (las que se necesiten).

Para ordenar un vector con datos numéricos se debe pasar como argumento una función al método **Array.sort()**, esta función indica qué operación debe hacerse sobre los datos a comparar entre dos datos del arreglo. Para comparar si un número es mayor que otros es muy común usar la resta porque si se resta un número solo existen tres tipos de respuestas. Cuando a un número **a** le restamos un número **b** ($a - b$), si el resultado es negativo es porque **b** es mayor que **a**, si el resultado es positivo porque **a** es mayor que **b** si el resultado es cero solo si **a** y **b** son iguales.

Figura 26

Ordenamiento de números



```
index.html x styles.css x script.js x
1  var arg  = [ 10, 53, 12, 56, 120 ];
2  arg.sort( function(a, b){
3      return a-b;
4  });
5  console.log (arg.join(', '));

CONSOLE x
10, 12, 53, 56, 120
```

Como se puede ver en la figura 26 se pasa una función (sin identificador) que recibe dos datos del arreglo (a y b) y los compara con la diferencia (resta) de ellos. A las funciones que no se les pone identificador se les llaman funciones anónimas, también existe otra forma de sintaxis para funciones anónimas, se denomina funciones flecha. Se verá, con la sintaxis de función flecha, cómo ordenar un arreglo número de manera inversa en la figura 27.

Figura 27

Ordenamiento inverso de números

```
index.html x styles.css x script.js x
1  var arg  = [ 10, 53, 12, 56, 120 ];
2  arg.sort( (a, b) => b-a );
3  console.log (arg.join(', '));

CONSOLE x
120, 56, 53, 12, 10
```

La figura 27 muestra cómo no se usa la palabra reservada **function**, tampoco se usan la identificación de contexto (corchetes) porque la función tiene una sola línea y se reemplaza por el operador **=>**. Y por defecto se retorna el resultado de la operación **b – a**, que es la diferencia de los dos números, pero en el otro orden (este orden de la operación es el que determina si el ordenamiento es de mayor a menor o viceversa).

C. Ordenamiento de registros

Suponiendo que se tienen los registros de la tabla, se desea ordenarlos de mayor edad a menor edad.

Tabla 8

Registros a ordenar



Usuario	Edad	Rol
Mariela	31	SAC
Eduardo	30	CEO
Andrés	34	Project Manager

Figura 28*Ordenamiento de registros*

```
index.html x styles.css x script.js x

1  var data = [
2      {usuario: 'Mariela', edad: 31, rol: 'SAC'},
3      {usuario: 'Eduardo', edad: 30, rol: 'CEO'},
4      {usuario: 'Andrés', edad: 34, rol: 'Project Manager'}
5  ];
6  data.sort((a, b) => {
7      return b.edad - a.edad
8  });
9  console.log(data);
10

CONSOLE x ...

[
  {
    "usuario": "Andrés",
    "edad": 34,
    "rol": "Project Manager"
  },
  {
    "usuario": "Mariela",
    "edad": 31,
    "rol": "SAC"
  },
  {
    "usuario": "Eduardo",
    "edad": 30,
    "rol": "CEO"
  }
]
```

En esta oportunidad en la figura 28 se muestra en las líneas 6 y 8, que se puede usar contexto (corchetes) a pesar de que dentro solo hay una sentencia (línea 7), el resultado a retornar es la diferencia entre **b.edad** y **a.edad** para que el ordenamiento sea de mayor a menor.

D. Buscar un elemento



El método `find()` devuelve el valor del primer elemento del *array* que cumple la función de prueba proporcionada (`Array.prototype.find()` – JavaScript, MDN, 2021).

Figura 29

Buscar un elemento

```
index.html × styles.css × script.js ×  
1 const array1 = [5, 12, 8, 130, 44];  
2 const found = array1.find(element => element > 10);  
3 console.log(found);  
  
CONSOLE ×  
12
```

Como se puede ver en la figura 29, se usa el `element` para referenciar a la variable que se va a aplicar el criterio que en el ejemplo es encontrar el primer elemento que sea mayor que 10.

Muchas veces es importante que, en lugar de retornar un solo valor, el resultado sea más de uno, por ejemplo, si en el arreglo de la figura 30, se retornan los valores que son mayores que 10, por lo tanto, debe retornar otro vector con los valores resultados. Para esto existe la función `Array.filter()`, que crea un nuevo arreglo con los datos que cumplan con la condición.

Figura 30

Buscar varios elementos

```
index.html × styles.css × script.js ×  
1 const array1 = [5, 12, 8, 130, 44];  
2 const found = array1.filter((element) => element > 10);  
3 console.log(found);  
  
CONSOLE ×  
[  
  12,  
  130,  
  44  
]
```

Es de interés notar que la consola solo retornó los datos que cumplen la condición que el elemento es mayor que 10.



En la medida que se va avanzando en el conocimiento de la sintaxis del lenguaje, el código fuente se va enriqueciendo de elementos y por tanto de complejidad, esto hace que sea necesario conocer herramientas para la depuración de fallas y corregir los fallos en la redacción del código.

4. Depuración y fallas de sintaxis

La principal funcionalidad que se requiere en la depuración del código, es la mostrar salidas parciales de datos de depuración del estado de las variables, o tal vez se tenga la necesidad de mostrar que se está ejecutando determinada línea de código. Por lo general, la salida de JavaScript está dirigida hacia un navegador o hacia un flujo de datos de un servidor web, dejando la consola para realizar estas depuraciones.

Hasta ahora se ha usado la consola para mostrar datos de depuración cuando se emplea **console.log()**, que es un método usado para mostrar mensajes de confirmación de operaciones o mitificaciones de éxito en las operaciones del algoritmo. Pero también existen más:

- **console.log()** para enviar y registrar mensajes generales.
- **console.dir()** para registrar un objeto y visualizar sus propiedades.
- **console.warn()** para registrar mensajes de alerta.
- **console.error()** para registrar mensajes de error.

Figura 31

Mensaje de depuración

```
index.html x styles.css x script.js x
1 const array1 = [5, 12, 8, 130, 44];
2 console.dir(array1);
3 console.warn("Mensaje de alerta " + array1[3]);
4 console.error("Mensaje de alerta " + array1[4]);

CONSOLE x
[
  5,
  12,
  8,
  130,
  44
]
Mensaje de alerta 130
error: Mensaje de alerta 44
```



Se puede revisar el procedimiento de despliegue y uso de consola para el navegador *Google Chrome* en el siguiente enlace <https://youtu.be/Hf3n-p3VYx4> (Choque, R. C, 2020).

5. Fallas de lógica



Para realizar las depuraciones y correcciones en la lógica, es preciso realizar la ejecución paso a paso de algunas líneas de código, para poder apreciar los cambios que las variables tienen en el transcurso de la ejecución. A continuación, se indican algunas instrucciones para realizar una revisión paso a paso, cuando la depuración se hace desde un navegador web siguiente los siguientes pasos.

A. Chrome DevTools

Se puede ir a las herramientas de desarrollo (DevTools) pulsando "Command+Option+I" en Mac o "Control+Shift+I" en Windows y Linux para ver el panel de inspección en la parte izquierda de la ventana del navegador. Al inspeccionar el elemento ofrece diferentes opciones para trabajar.

Aparece varias pestañas las opciones presentes para depuración en Chrome son:

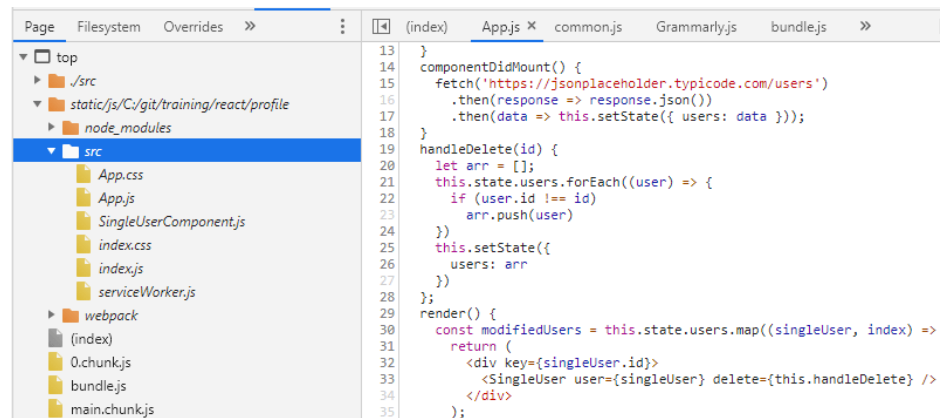
- **Console:** la consola se utiliza para imprimir registros, muestra errores y advertencias.
- **Elements:** se pueden seleccionar los elementos para inspeccionar los elementos HTML y hacer estilos en ellos.
- **Sources:** permite navegar los archivos que están presentes en la página cargada y se puede elegir el archivo que se desea consultar.
- **Network:** empleado para inspeccionar todas las llamadas de red que realizan los clientes al servidor, como las cabeceras HTTP, el contenido y el tamaño.

B. Seleccionar fuente (sources)

Una vez se tiene que ir a la sección del archivo con el código fuente a depurar, se puede navegar a través de él e identificar la porción de código en la que se sospecha ocurre un error en la lógica.

Figura 32

Seleccionar fuente

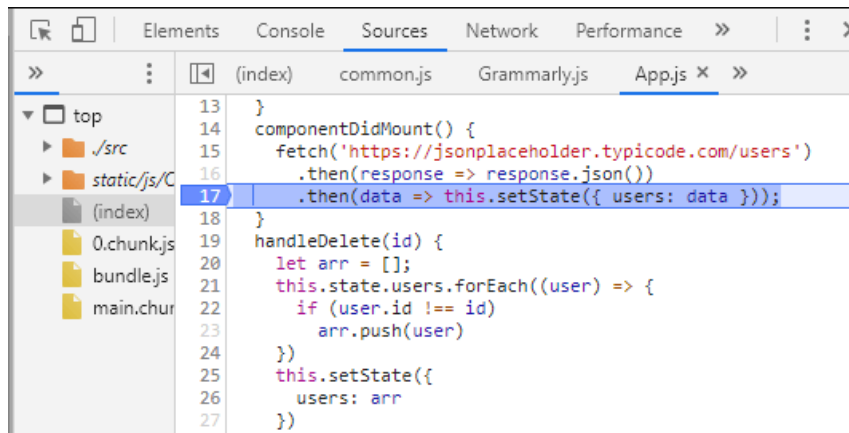


C. Marcar punto de interrupción

Un punto de interrupción es una línea de código donde el programa se detendrá. Ya seleccionado el archivo a ser depurado, se puede ver que hay números de línea escritos por cada línea de código. Se debe hacer clic en este número de línea y esta línea actuará como un punto de interrupción y la ejecución se detendrá una vez que el control llegue a esta línea.

Figura 33

Marcar un break point

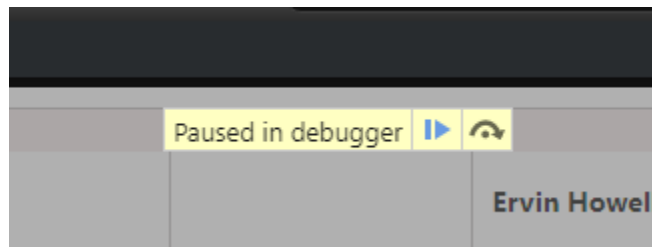


D. Avanzar línea a línea

Cuando la ejecución se detiene en el punto de interrupción, se puede reanudar la ejecución haciendo clic en el botón azul que se muestra en la figura 34, así la ejecución avanza línea por línea. Aquí se puede inspeccionar cada línea de código junto con los valores contenidos en las variables.

Figura 34

Ejecución línea a línea

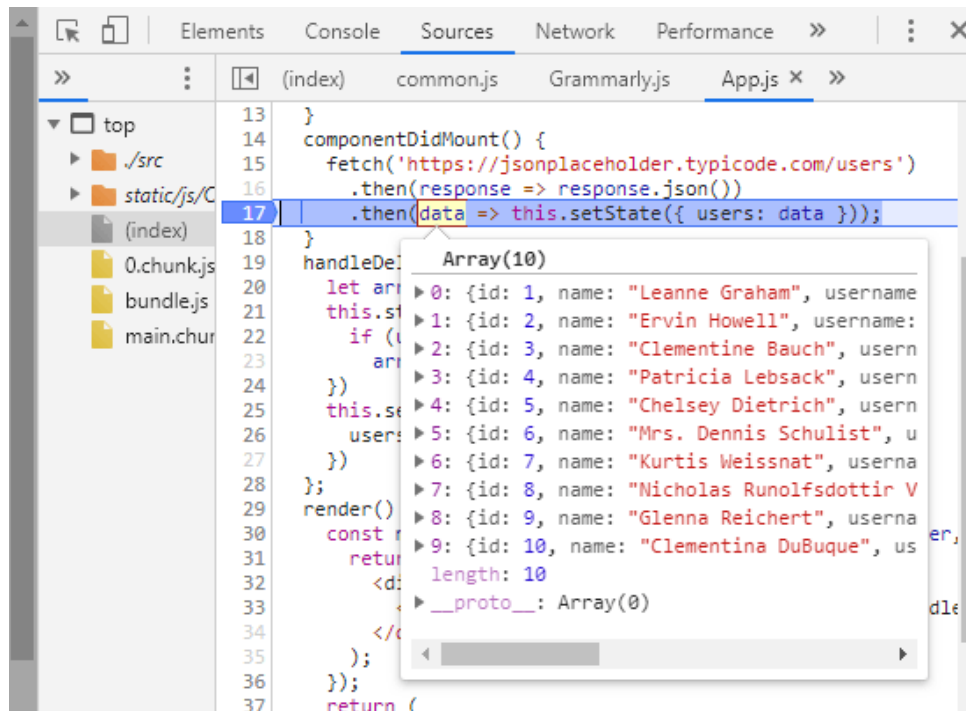


E. Comprobar el estado de las variables

El punto más importante es conocer los valores de las variables, esto se hace pasando el ratón por encima de la variable. La principal ventaja de usar puntos de ruptura es que se pueden conocer los valores de más de una variable al mismo tiempo, lo que no es posible en el caso de las consolas de registro, donde se debe escribir `console.log()` para cada variable.

Figura 35

Valores de las variables



	Se puede ver un ejemplo en el siguiente enlace: https://youtu.be/I388w3wDkjc (Autodidacta F, 2017).
--	------------------------------------------------------------------------------------------------------------------------------------------------

6. Manejo de errores y excepciones

Las excepciones son errores imprevistos que ocurren en la ejecución de un programa; son anomalías que impiden o perturban el comportamiento o el flujo normal de las instrucciones de un programa.

El objetivo principal en el manejo de errores es separar el código de la lógica del programa, del código de manejo de excepciones: de esta forma, al producirse la anomalía, si ocurre un error y existe un manejador de excepciones, él toma el control de la ejecución del programa.

En JavaScript, el control de errores resulta muy necesario tal vez más que en otros lenguajes debido a la dificultad natural de este para testear aplicaciones o los cambios de versión de intérprete en múltiples navegadores.

La forma más sencilla de disparar un error es con la palabra reservada **throw**, este comando permite enviar al intérprete de JavaScript el evento de que ha ocurrido un error, generalmente permite enviar cualquier tipo de dato, pero lo más común que envíe un "Error" como se muestra en la siguiente figura.

Figura 36

Lanzar una excepción

```
throw new Error( "No se puede dividir por cero" );
```

Existe un tipo de estructura de control en JavaScript que está pendiente de revisar en el flujo de ejecución de un programa para detectar comportamientos inesperados, la estructura es **try ... catch** que consta de un bloque que intenta la ejecución de una sesión de código (contexto), a la espera de que pueda ocurrir una posible excepción. Finalizado el contexto existe otro denominado **catch** que tiene un argumento o variable de contexto, donde se captura el elemento disparado.

**Figura 37***Bloque de captura de excepción*

```
index.html × styles.css × script.js ×  
  
1 let numerador = 4;  
2 let denominador = 0;  
3 var resultado = null;  
4 try{  
5     if(denominador == 0){  
6         throw new Error( "No se puede dividir por cero" );  
7     }else{  
8         resultado = numerador / denominador;  
9     }  
10 }catch(e){  
11     console.error(e);  
12     resultado = null;  
13 }
```



```
CONSOLE ×  
  
error: Error: No se puede dividir por cero
```

En la figura 37 se muestra un algoritmo que divide dos números y, como se sabe, en este proceso se debe validar que no exista una división por 0 al no estar definida aritméticamente.

Para eso existe el contexto **try** que va desde la línea 4 a la línea 10, y dentro de él se verifica el caso que el divisor si es igual a 0 debe disparar una excepción. Ella es capturada en la línea 10 por el contexto **catch** en la variable **e**, de esta forma, el contexto de catch de la línea 10 a la 13 maneja la excepción, mostrando un mensaje de error y volviendo la variable resultado a nulo. Así, la excepción es lanzada y capturada y el programa no es interrumpido abruptamente, sino que es controlado a pesar de los problemas que se puedan presentar.