Guía de ejercicios 3 - Funciones en JavaScript

 ¡Hola! Te damos la bienvenida a esta nueva guía de ejercicios.

¿En qué consiste esta guía?

En la siguiente guía podrás trabajar los siguientes aprendizajes:

- Crear una función con un parámetro
- Crear una función con múltiples parámetros
- Crear una función con parámetros que tengan valor por defecto
- Crear una función que retorne un valor
- Crear una expresión de función
- Crear una expresión de función con arrow functions
- Diferenciar el uso de var, let y const
- Explicar que es hoisting

¡Vamos con todo!

Tabla de contenidos

¡Comencemos!

Introducción a funciones

Las funciones son conjuntos de instrucciones que podemos programar una vez y utilizarlas cada vez que necesitemos. A veces ocuparemos funciones creadas por otros, así como también tendremos que crear las nuestras.

Para crear una función tenemos que escribir:

```
function nombre_funcion(){
  /* Aquí agregaremos lo que hace nuestra función */
}
```

Luego podemos utilizar la función creada escribiendo:

```
nombre_funcion()
```

"Usar una función" es sinónimo de **llamar** o invocar una función. En general utilizaremos el término llamar a la función.

Por ejemplo, podemos crear una función que cambie el fondo del sitio a color negro:

```
/* Creamos la función */
function pintar_negro(){
  elemento = document.querySelector("body")
  elemento.style.backgroundColor = "black"
}

/* LLamamos a la función */
pintar_negro();
```

Podemos probar el código anterior ejecutándolo en la consola del inspector de elementos o en una página nueva como en el siguiente ejemplo:

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="es">
 <head>
   <meta charset="UTF-8" />
   <meta http-equiv="X-UA-Compatible" content="IE=edge" />
   <meta name="viewport" content="width=device-width, initial-scale=1.0" />
   <title>Pintar</title>
 </head>
 <body>
   <script>
     /* Creamos la función */
     function pintar_negro() {
       elemento = document.querySelector("body");
       elemento.style.backgroundColor = "black";
     }

     /* LLamamos a la función */
     pintar_negro();
   </script>
 </body>
</html>
```

Funciones y parámetros

Los **parámetros** nos permiten ingresar valores de entrada a una función. Por ejemplo, podríamos crear una nueva función pintar que reciba un color de entrada y luego utilizar este color para pintar la página.

```
pintar = function(color){
  elemento = document.querySelector("body")
  elemento.style.backgroundColor = color
 }
 pintar("black");
```

Los parámetros van dentro de los paréntesis al momento de definir la función. En este caso, el parámetro es color mientras que el valor pasado, al llamar la función, fue "black". Los valores pasados al llamar la función reciben formalmente el nombre de **argumentos**.
Si cargamos el script anterior en una nueva página web llamada pagina2.html y la abrimos con el navegador, veremos nuevamente el fondo negro.

```
<!-- pagina2.html -->
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8" />
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Pintar 2</title>
</head>
<body>
  <script>
   pintar = function(color){
     elemento = document.querySelector("body")
     elemento.style.backgroundColor = color
    }
    pintar("black");
  </script>
</body>
</html>
```

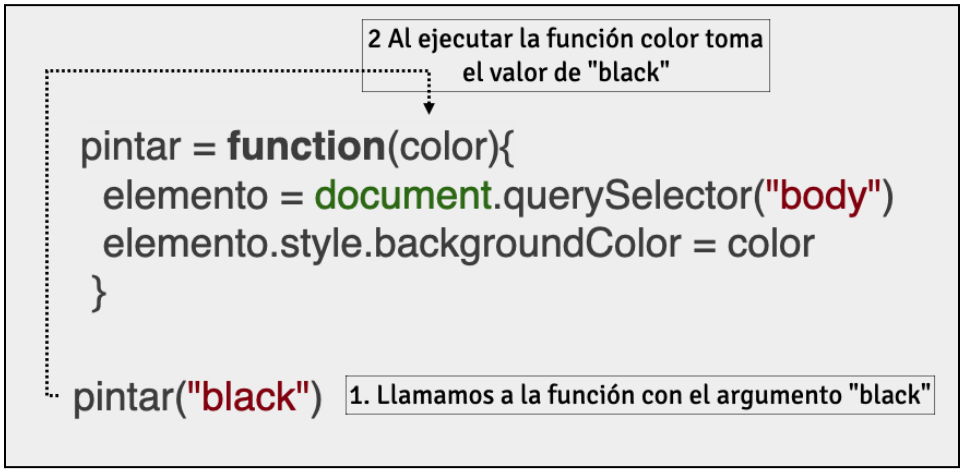

Imagen 1. Función con el argumento “black”
Fuente: Desafío Latam.

Estas dos expresiones significan lo mismo:

1. Llamamos a la función con el argumento "black".

2. Llamamos a la función y le pasamos el valor black.

El término “argumentos” y “parámetros” es típicamente confundido en los primeros pasos de un desarrollador. Su diferencia radica en el momento en el que se escriben:

A. cuando estamos definiendo una función, el término que ocupamos es ***parámetros.***
B. cuando estamos ejecutando una función, los valores que asignados entre los paréntesis se conocen como ***argumentos.***

Funciones con múltiples parámetros

Las funciones pueden tener más de un parámetro. Para lograrlo simplemente tenemos que separarlos con coma al momento de definir la función.

```
funcionMultiplesParametros(par1, par2, par3) {
}
```

Al llamar a la función, tenemos que pasar los argumentos en el mismo orden que están definidos los parámetros.

```
funcionMultiplesParametros("azul", "#id-2", 5)
```

De esta forma par1 tomaría el valor azul, par2 el valor "#id-2" y par3 el valor 5.

Funciones y onclick

Pintando la página

Ahora que sabemos crear funciones con parámetros podemos hacer cosas nuevas, como por ejemplo, llamar una función al hacer clic en un botón.

```
<!-- pagina3.html -->
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
/>
    <title>Document</title>
  </head>
  <body>
    <button onclick="pintar('black')"> Pintar negro</button>
    <button onclick="pintar('red')"> Pintar rojo</button>
    <button onclick="pintar('green')"> Pintar verde</button>
    <script>
      function pintar(color) {
        elemento = document.querySelector("body");
        elemento.style.backgroundColor = color;
      }
    </script>
  </body>
</html>
```

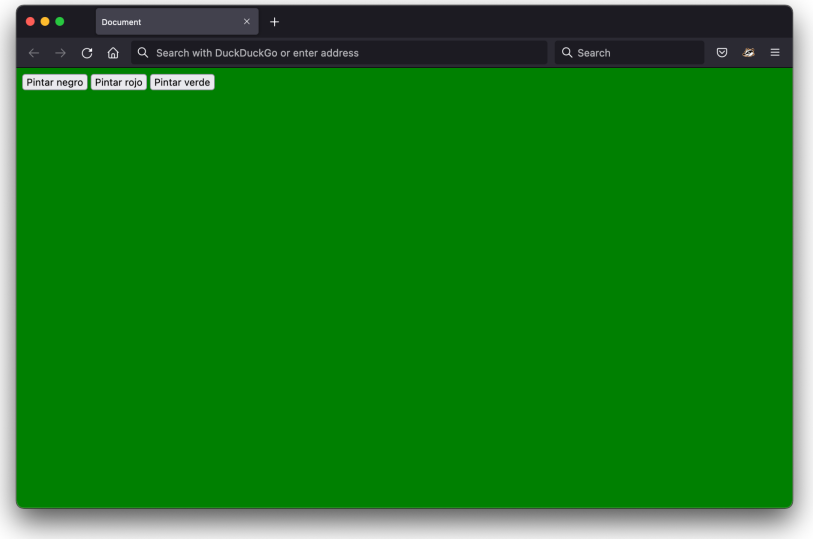

Imagen 2. Página pintada.
Fuente: Desafío Latam.

Ejercicio

Agrega dos botones a la página anterior, uno para pintar amarillo (yellow), el otro para pintar azul (blue).

Agregando bordes

Podemos realizar un ejercicio similar en el que tengamos varias imágenes las cuales queremos que sean seleccionables de modo que al hacer click en ellas se les agregue un borde.

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
/>
    <title>Document</title>
  </head>
  <body>
```

```
    <img id="img-1" src="http://placekitten.com/g/300/300" alt=""
onclick="agregarBordes('img-1')">
    <img id="img-2" src="http://placekitten.com/g/300/300" alt=""
onclick="agregarBordes('img-2')">
    <img id="img-3" src="http://placekitten.com/g/300/300" alt=""
onclick="agregarBordes('img-3')">

    <script>
      /* Creamos la función */
      function agregarBordes(elementId) {
        elemento = document.querySelector('#' + elementId);
        elemento.style.border="dashed 3px brown"
      }
    </script>
  </body>
</html>
```

Para lograr que nuestra función sea reutilizable, hacemos que el ID del elemento sea un parámetro. De esta forma, podemos llamarla con cualquier ID y agregar bordes a cualquier imagen, e incluso podríamos utilizarla con otros tipos de elementos.

Actividad 1: Bordes de distinto color

1. Modifica la función `agregarBordes` para que pueda recibir un color.
2. Modifica el llamado a la función para que en la imagen 1 se llame con el color azul, en la segunda verde y en la tercera con el color rojo.

Parámetros por defecto

En algunas situaciones va a ser conveniente que una función tenga un valor por defecto. Por ejemplo, nuestra función de agregar bordes podría agregar bordes rojos salvo que se especifique otro color.

```
      /* Creamos la función */
```

```
function agregarBordes(elementId, color = 'red') {
  elemento = document.querySelector('#' + elementId);
  elemento.style.border="dashed 3px " + color
}
```

Si ahora llamamos a la función `agregarBordes('img-1')` el color rojo `'red'` se asumirá por defecto. En cambio, si llamamos a la función de la siguiente forma `agregarBordes('img-1', 'blue')` el borde será de color azul `'blue'`

Actividad 2: Creando una función con parámetros para modificar el DOM

Crear una página web con un párrafo que contenga el párrafo “Incremento de Font Size” y agrega 3 botones, uno con el texto “aumentar en 1px”, otro “aumentar en 2px”, y otro con “aumentar 10px”.

1. Crea la función incrementar con el valor por defecto de 1.
2. Dentro de la función busca el elemento y modifica el contenido actualizando su cantidad.
3. Llama a la función con los argumentos correspondientes en cada botón.

Tips:

- Utiliza la propiedad .fontSize
  - Ejemplo: elemento.style.fontSize=12 + 'px'
  - Al aumentar el número debes hacerlo solo sobre el 12 (o valor actual) y no sobre 12px
  - Antes de sumar debes convertir el valor a número, de otra forma al sumar 0 con 1 obtendrás 01 y luego 011.
- Solución al final de la guía

Retorno

Así como las funciones reciben valores de entrada, también tienen un valor de salida. Podemos imaginarnos las funciones como una caja negra que realiza alguna acción en particular. Una vez definida sólo nos preocupamos de pasarle los valores de entrada y obtener el valor de salida.

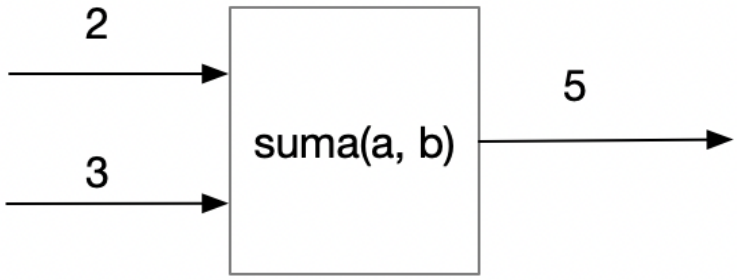

Imagen 3. Entradas y salidas de una función.
Fuente: Desafío Latam.

El valor de salida de una función se especifica ocupando la instrucción **return**. Probemos la siguiente función:

```
function suma(a, b){
  return a + b
}

alert(suma(2,3))
```

También podemos guardar la salida en una variable para ocupar el valor nuevamente si es necesario.

```
function suma(a, b){
  return a + b
}

var resultado = suma(2,3)
console.log(resultado)
```

No todas las funciones devuelven un valor

Hay funciones que retornan valores. Estos valores usualmente los guardaremos en una variable o los ocuparemos para realizar alguna operación. Por ejemplo, cuando hacemos `document.querySelector("#elemento")` esto nos devuelve un elemento, que ocupamos para cambiar el estilo o contenido. Cuando ocupamos prompt nos devuelve un string que guardamos en una variable, ese lo ocupamos para definir un nuevo contenido en la página web. Sin embargo, en otras ocasiones, como por ejemplo cuando hacemos `alert()`, las funciones no devuelven ningún valor.

Usualmente las funciones que construiremos para modificar el DOM no devolverán valor. Cuando una función no devuelve un valor devuelve algo llamado undefined.

Undefined: Una función que no devuelve valor realmente devuelve algo llamado "undefined" , o sea, no definido.

```
a = alert("hola")
console.log(a) /* Mostrará undefined, porque alert no devuelve valor y eso
lo guardamos en la variable a
```

Probemos creando una función que no devuelve valor:

```
function ejemplo(){
  console.log("hola mundo")
}

a = ejemplo()
console.log(a) /* también mostrará undefined */
```

Actividad paso a paso: Crear la función por_dos

En esta actividad vamos a crear paso a paso una función para multiplicar por dos. Se entiende que no es algo que se necesite hacer en el día a día, ya que es algo que podemos hacer con el operador *, pero es un buen ejercicio para reforzar los conceptos aprendidos.

1. Crea una función llamada por_dos.
2. Agrega el parámetro num a la función.
3. La función debe retornar num * 2.
4. Llamar a la función por_dos utilizando el valor 10.
5. Guardar el valor devuelto de la función en una variable llamada resultado.
6. Mostrar la variable resultado en pantalla con console.log o alert, el resultado mostrado debería ser 20.

Veamos el paso a paso de como resolver esta actividad:

1. Crea una función llamada por_dos:

```
por_dos = function(){
}
```

2. Agregar el parámetro num a la función:

```
por_dos = function(num){
}
```

3. La función debe retornar num * 2:

```
por_dos = function(num){
  return num *2
}
```

4. Llamar a la función por_dos utilizando el valor 10:

```
por_dos = function(num){
  return num *2
}

por_dos(10)
```

5. Guardar el valor devuelto de la función en una variable llamada resultado:

```
por_dos = function(num){
  return num *2
}

resultado = por_dos(10)
```

6. Mostrar la variable resultado en pantalla con console.log o alert. El resultado mostrado debería ser 20:

```
por_dos = function(num){
  return num *2
}

var resultado = por_dos(10)
console.log(resultado)
```

Actividad 3: Crea la función por_tres

Intenta hacer esta actividad desde cero, sin mirar los resultados de la anterior.

1. Crea una función llamada por_tres
2. Agrega el parámetro num a la función
3. La función debe retornar num * 3
4. Llamar a la función por_tres utilizando el valor 10
5. Guardar el valor devuelto de la función en una variable llamada resultado
6. Mostrar la variable resultado en pantalla con console.log o alert, el resultado mostrado debería ser 30.

Actividad 4: Creando la función multiplicar

Crea la función multiplicar. Esta debe tener dos parámetros: num1 y num2. La función debe retornar la multiplicación de ambos números. Luego llama a la función utilizando los valores 5 y 10 guardando el resultado en nueva variable. Finalmente muestra con alert o console.log el resultado.

Consejos para nombrar funciones

Así como a las variables, a las funciones se les puede asignar nombres, decisión que suele ser compleja en el desarrollo de aplicaciones.

No existe una regla específica de cómo nombrar a una función. Podríamos incluso asignar una combinación de letras y números y esto no afectaría su funcionamiento. No obstante, en el desarrollo existen "las buenas prácticas" que no son más que consejos que podemos considerar al momento de escribir nuestro código.

La recomendación más común consiste en agregar un verbo en infinitivo al comienzo del nombre. Además, se debe tratar que el nombre represente de manera específica el propósito del bloque de código que la función representa.

A continuación tenemos varios ejemplos de funciones cuyos nombres siguen la buena práctica mencionada anteriormente:

```
function calcularPromedio(){
 // código...
}

function getData(){
 // código...
}

function registrarUsuario(){
 // código...
}

function eliminarTarea(){
 // código...
}
```

Siguiendo esta práctica nuestro código tendrá una interpretación más intuitiva.

Eventos

Introducción a eventos

Hasta ahora hemos trabajado con el evento onclick, el cual hemos utilizado en los botones. Ahora aprenderemos cómo se agregan los eventos fuera del HTML lo que nos permitirá crear código javascript más sencillo de reutilizar.

Para agregar un evento con Javascript, debemos seleccionar un elemento, por ejemplo con document.querySelector y luego agregar un listener. Un listener es un oyente, un elemento que está escuchando si hay algún evento y nos permite ejecutar código Javascript en función de ese cambio.

Veamos un ejemplo en donde tenemos un botón HTML que al ser presionado muestra un mensaje.

```
<body>
 <button>Mostrar mensaje</button>
 <script>
   function alertar(){
     alert("hola")
   }

   btn = document.querySelector("button")
   btn.addEventListener("click", alertar)
 </script>

</body>
```

En este código no se utiliza onclick para agregar el script, si no que todo el js se encuentra dentro del archivo. Para escuchar el evento click se utiliza .addEventListener("click", alertar). En este caso “alertar” es la función, la cual definimos previamente.

Separando Javascript del HTML

Con lo aprendido podemos dividir la página web en dos archivos distintos, uno para el HTML y otro para el JS. Para este primer ejemplo ambos estarán en la raíz del proyecto. index.html y script.js

Html

```
<body>
  <button>Mostrar mensaje</button>
  <script src="script.js"></script>
</body>
```

JS

```
/* script.js */
function alertar(){
  alert("hola")
}

btn = document.querySelector("button")
```

```
btn.addEventListener("click", alertar)
```

Probemos que funciona abriendo el archivo index.html en el navegador.

Ahora que podemos separar la lógica de JS del HTML será más sencillo reutilizar nuestros archivos JS y entender cómo trabajar con archivos JS creados por otros.

Ordenando nuestros archivos

Ahora que aprendimos a manipular el DOM con eventos dentro de la etiqueta script, o sea sin necesidad del onclick o del onchange, podemos volver a la estructura de carpetas que estábamos trabajando en módulos anteriores. Es importante recordar que ésta es sólo una estructura más, no existe una estructura perfecta de archivos y cada equipo de trabajo acuerda una al momento de desarrollarla. Sin embargo, aprendiendo a trabajar bajo una estructura podremos movernos fácilmente a otras.

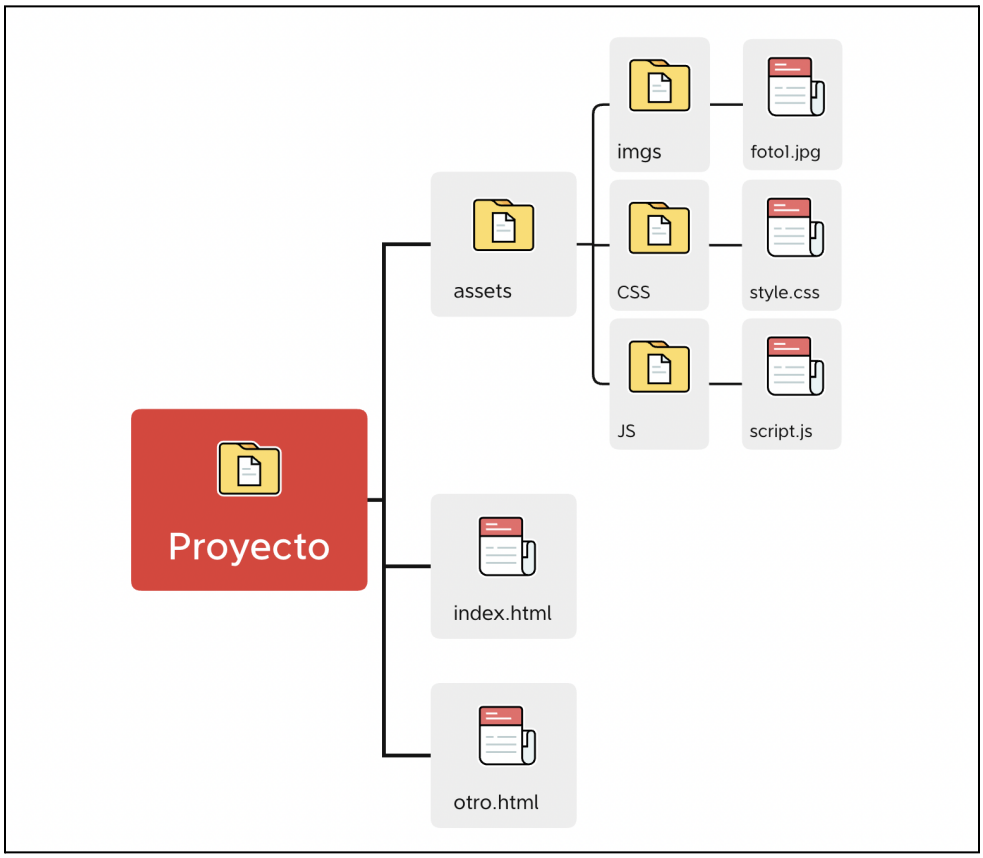

Imagen 4. Diagrama de archivos ordenados.
Fuente: Desafío Latam.

Si nuestro proyecto está organizado de esta manera, el script lo agregaremos de la siguiente forma:

```
<body>
  <button>Mostrar mensaje</button>
  <script src="assets/js/script.js"></script>
</body>
```

Ejercicio con eventos

Retomemos el proyecto del botón que pinta la página de color negro y esta vez realicémoslo separando el index.html del script.

- El html tiene simplemente un botón.

- El js debe tener una función para pintar en negro, y debe buscar el botón y agregar un oyente al evento click llamando a la función pintar en negro.

Respuesta al ejercicio

Crea una carpeta nueva, la llamaremos pintarX

- Dentro, crearemos el archivo index.html y el archivo script.js

```
/* index.html */
<body>
  <button>Pintar negro</button>
  <script src="script.js"> </script>
</body>
```

Es importante que el archivo script.js se cargue casi al final del HTML, puesto que el navegador tiene que haber leído los elementos que vamos a seleccionar desde JS.

```
/* script.js */
function pintar(color) {
  elemento = document.querySelector("body");
  elemento.style.backgroundColor = color;
      }

btn = document.querySelector("button")
btn.addEventListener("click", pintar("negro"))
```

Funciones anónimas

¿Qué son las funciones anónimas?

Una función anónima, como su nombre lo indica, es una función sin nombre. Estas funciones son útiles cuando se requiere pasarlas como argumento a otra función.

Un caso muy común en el que se utilizan las funciones anónimas es al momento de añadir un Event Listener

```
btn = document.querySelector("button")

/* Ejemplo de uso Con función normal */
function alertar(){
  alert("hola")
}

btn.addEventListener("click", alertar)


/* Ejemplo de uso Con función anónima */
btn.addEventListener("click", function() {
      alert("hola")
});
```

Como vemos, la sintaxis para añadir un Event Listener es la misma. La única diferencia es que en el caso de la función anónima, la definimos directamente en el lugar donde se llama a "`addEventListener`", sin necesidad de declararla previamente.

¿Cuando ocupar funciones anónimas?
Si vas a llamar a la función una única vez, probablemente sea mejor utilizar una función anónima.

Funciones como argumentos

En Javascript es muy frecuente que una función reciba otra función como argumento.
Por ejemplo:

```
function ejecutarFuncion(funcionArgumento){
  funcionArgumento()
}
```

Al llamar a la función, pasamos como argumento una función anónima.

```
ejecutarFuncion(function() {
       alert("hola");
});
```

La función anónima ahora se ejecutará dentro de la función `"ejecutarFuncion"` cuando ésta sea llamada.

Tenemos que tener cuidado de no confundir los siguientes casos:

`btn.addEventListener("click", alerta())`	**Forma incorrecta** Aquí simplemente se llama la función, es decir, se está pasando como argumento el resultado de la función, no la función misma. Eventualmente puede que el resultado nos sirva, pero si queremos pasar una función como argumento, esta no es la forma correcta. Si intentamos esto se mostraría la alerta al cargar la página, y no al hacer click sobre el elemento.
`btn.addEventListener("click", alerta)`	**Forma que funciona en algunas ocasiones** Al igual que en el caso anterior, se está pasando el resultado de una función, lo cual sería de utilidad sólo si queremos pasar la función sin argumentos.
`btn.addEventListener("click", function(){ alerta("p1") })`	**Forma que utilizaremos en el curso** Pasando una función anónima. Dentro podemos llamar a las funciones que necesitemos con los argumentos que necesitemos.

Expresión de función

Adicionalmente, es posible asignar una función anónima a una variable, lo que se conoce como Expresión de Función:

```
alertar = function() {
      alert("hola")
}
```

Luego podemos llamar a la función alertar de la siguiente forma:

```
alertar()
```

Vemos que es muy similar a la forma anterior que estudiamos, la cual se conoce como Declaración de Función:

```
function alertar(){
  alert("hola")
}
```

Podemos ocupar cualquiera de las dos formas, ya que son técnicamente iguales, salvo por una característica particular llamada Hoisting que estudiaremos más adelante.

Arrow functions

Las funciones arrow son una forma alternativa de escribir expresiones de funciones o funciones anónimas utilizando una flecha **=>** lo que permite escribirlas de forma mucho más concisa.

Ejemplo:

```
suma = (a, b) => a + b
```

Sería lo mismo que:

```
suma = function (a, b) {
      return a + b
}
```

Cuando queremos ocupar múltiples líneas en las arrow function tenemos que ocupar las llaves y especificar el return.

```
suma = (a, b) => {
  console.log(a)
  console.log(b)
  return a+ b
}
```

Cuando es una sola línea el return es implícito.

```
suma = (a, b) => a + b /* a + b aquí es lo mismo que return a + b
```

Si la función no recibe ningún parámetro puede ser escrita de la siguiente forma:

```
suma = () => alert('hola')
```

Arrow functions y addEventListener

Dijimos que las arrow functions nos permiten escribir funciones anónimas y, por lo mismo, podemos utilizarlas dentro del método addEventListener.

```
<body>
 <button>Mostrar mensaje</button>
 <script>
   btn = document.querySelector("button")
   btn.addEventListener("click", () => alert('hola'))
 </script>
</body>
```

Resumen de formas de declarar una función

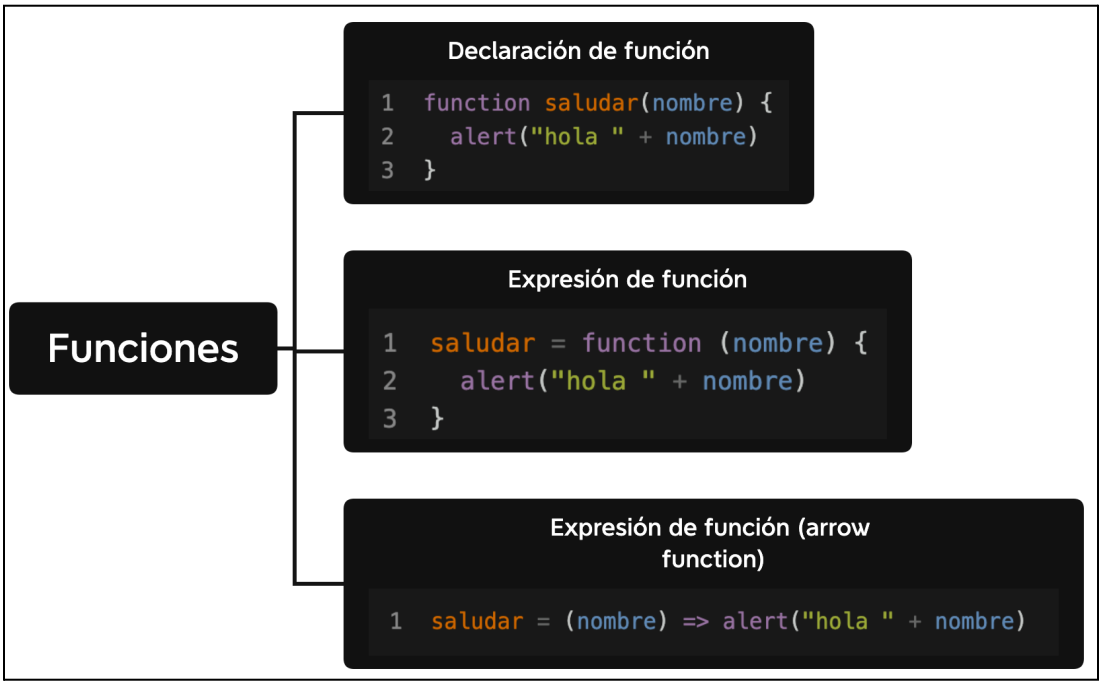

Imagen 5. Cómo declarar funciones.
Fuente: Desafío Latam.

Scope

El scope (ámbito o alcance en español) es un concepto que explica desde donde podemos acceder a las variables que definimos.

En javascript hay 3 tipos de scopes:

- Global
- Función
- Bloque

```
<script>
  /* Aquí estamos en el scope global */

  if (a == 5) {
    /* Aquí estamos en un scope de bloque */
  }
```

```
   function hola(){
     /* Aquí estamos en un scope de función */
   }
</script>
```

Cuando definimos una variable como lo hemos hecho hasta ahora, por ejemplo a = 5, esta variable, independiente desde donde la definamos, automáticamente queda en el scope global. Esto quiere decir que se puede ocupar desde cualquier lado.

Definir variables dentro del scope global puede ser perjudicial, ya que es fácil que pasemos a llevar una variable importante escribiendo otro valor sobre ella.

Definiendo variables con var

Al momento de definir una variable es posible hacerlo de la siguiente forma:

```
var nombreVariable = valor
```

Por ejemplo:

```
var a = 5
```

Las variables definidas con var dentro de una función, tendrán el scope de la función.

```
function hola () {
  var a = 5
}

hola()
alert(a) /* Esto mostrará el siguiente error: ReferenceError: Can't find
variable: a */
```

La razón por la que no puede ser encontrada es porque la variable fue definida dentro del scope de la función. En otras palabras, fuera de la función la variable no existe.

Cuando definimos var fuera de una función, la variable tendrá el scope global.

```
var a = 5
function hola (){
  console.log(a); /* mostrará 5 :) */
}
```

El scope de las variables definidas con var es fácil de recordar: las variables definidas con var dentro de una función tienen el scope de la función y aquellas definidas dentro del scope global entonces son globales.

Las variables definidas con var no entienden el concepto de bloque.

```
a = 5
if (a == 5) {
  var b = 8
}

console.log(b) /* 8 */
/* Lo anterior funciona porque las definiciones con var ignoran los
bloques, por lo tanto, se puede acceder a b desde afuera del bloque*/
```

Entendiendo que hay 3 scopes (global, función y bloque) esperaríamos que en el caso de bloques, var se comportara similar a como lo hace con las funciones, sin embargo, si una variable var es definida dentro de un bloque, esta puede ser accedida desde fuera.

Para utilizar el scope de bloques debemos utilizar **let.**

Definiendo variables con let

Let es similar a var pero adicionalmente se agrega el scope de los bloques.

```
a = 5
if (a == 5){
  let b = 8 /* La variable b está definida dentro del bloque y tiene un
```

```
alcance del bloque */
}

console.log(b) /* Error. No podemos ocupar la variable fuera del bloque */
```

La otra diferencia importante es que let solo lo podemos utilizar una vez:

```
let a = 5
let a = 8 /* error */
```

Utilizamos let al momento de declarar la variable, luego la modificamos sin hacer uso de let, ejemplo:

```
let a = 5
a = 8
```

Entendiendo que utilizar innecesariamente el scope global es peligroso, a partir de ahora definiremos en el curso las variables con let.

Definiendo con const

Const es abreviación de constante, lo que implica que su valor no cambia. Cuando necesitemos guardar un valor para su uso posterior y dicho valor no cambiará a lo largo del programa, entonces lo guardaremos con const. Si intentamos modificar el valor de una constante obtendremos un error.

```
const a = 5
a = 7 /* Error: Assignment to constant variable. */
```

Los variables que guarden valores que pueden ir cambiando las guardaremos con let y si no cambian las guardaremos con const. Las funciones de expresión también las guardaremos con const. Un ejemplo de esto:

```
const suma = (a, b) => a + b
```

Resumen

Scope (Ámbito)				
	Global	Funcion	Bloque	¿Puede cambiar?
sin modificador	si	no	no	si
var	si	si	no	si
let	si	si	si	si
const	si	si	si	no

Imagen 6. Resumen de Scope.
Fuente: Desafío Latam.

Hoisting

Hoisting (en español alzar) es un mecanismo por el cual todas las declaraciones se mueven al principio del scope (global o de función) en el que fueron definidas.

La implicancia de esto es que podemos utilizar funciones incluso antes de haberlas definido. Por ejemplo, el siguiente script funcionará correctamente:

```
x() /* Llamamos a la función x */
function x() { /* Declaramos la función x */
    console.log("hola");
}
```

Es importante recordar que hay dos formas de crear funciones en JavaScript: a través de declaraciones de función (functions declarations) o expresiones de función (functions expressions).

En las expresiones de función el hoisting no aplica.
Veamos un ejemplo del error obtenido:

```
y() /* Uncaught ReferenceError: y is not defined */
y = function () { /* Esta es una expresión de función */
console.log("hola");
}
```

En este caso el script fallaría porque las expresiones de función no son alzadas.

Resumen

1. Hoisting es un mecanismo que alza las definiciones al principio del scope donde fueron definidas.

2. Existen dos formas de crear funciones:
    a. Utilizando declaraciones de funciones.
    b. Utilizando expresiones de funciones.

3. Las declaraciones de funciones son alzadas.

Preguntas de entrevista laboral

Intenta contestar con tus palabras las siguientes preguntas, anota las respuestas y luego revisa en la guía las definiciones para evaluar si las aprendiste correctamente.

1. ¿Cuál es la diferencia entre var y let?

2. ¿Qué es hoisting, en qué casos aplica y en cuáles casos no?

3. ¿Qué uso tiene una función anónima?

4. ¿Qué son las arrow functions?

5. ¿Qué tipos de scopes existen en JavaScript? ¿Cuáles son las diferencias entre estos tipos de scope?

Resumen

- Las funciones pueden crearse siguiendo una estructura representada por una entrada, proceso y salida.
- Definir los parámetros y el retorno en una función es opcional.
- Con las funciones podemos ahorrarnos muchas líneas de código al reutilizar una estructura dinámica que por buenas prácticas debe incluir un verbo infinitivo al comienzo de su nombre.
- Al momento de crear una variable podemos decidir ocupar var, let o const, entendiendo que varían en sus alcances y restricciones.

Solucionario Actividades

Solución Actividad 1:

```
<!DOCTYPE html>
<html lang="es">
 <head>
   <meta charset="UTF-8" />
   <meta http-equiv="X-UA-Compatible" content="IE=edge" />
   <meta name="viewport" content="width=device-width, initial-scale=1.0" />
   <title>Document</title>
 </head>
 <body>
   <img id="img-1" src="http://placekitten.com/g/300/300" alt=""
onclick="agregarBordes('img-1', 'blue')">
   <img id="img-2" src="http://placekitten.com/g/300/300" alt=""
onclick="agregarBordes('img-2', 'green')">
   <img id="img-3" src="http://placekitten.com/g/300/300" alt=""
onclick="agregarBordes('img-3', 'red')">

   <script>
     /* Creamos la función */
     function agregarBordes(elementId, color = 'red') {
       elemento = document.querySelector('#' + elementId);
       elemento.style.border="dashed 3px " + color
     }
   </script>
 </body>
</html>
```

Solución Actividad 2:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Incremento de Font Size</title>
</head>
<body>
  <button onclick="incrementar()">Aumentar en 1px</button>
  <button onclick="incrementar(2)">Aumentar en 2px</button>
  <button onclick="incrementar(10)">Aumentar en 10px</button>
  <p id="numero" style="font-size: 16px;">Incremento de Font Size</p>

  <script>
     function incrementar(valor = 1) {
        var elemento = document.getElementById("numero");
        var fontSizeActual = parseInt(elemento.style.fontSize);
        var nuevoFontSize = fontSizeActual + valor + “px”;
        elemento.style.fontSize = nuevoFontSize;
     }
  </script>
</body>
</html>
```

Solución Actividad 3:

```
function por_tres(num) {
  return num * 3;
}

var resultado = por_tres(10);
console.log(resultado); // Muestra el resultado en la consola (30)
```

Solución Actividad 4:

```
function multiplicar(num1, num2) {
  return num1 * num2;
}

var resultado = multiplicar(5, 10);
console.log(resultado); // Muestra el resultado en la consola (50)
```