

Agilizar los procesos de las aplicaciones web

"Todos deberían aprender a programar, les enseña a estructurar el pensamiento."

Steve Jobs— Cofundador de Apple y cofundador de Pixar



Photo by [William Iven](#) on [Unsplash](#)

¿Con cuántos retos te encuentras día a día? A medida que avanzas en este camino como developer algo es seguro: ¡tendrás el desafío de desarrollar programas de mayor complejidad! [MDN](#), la plataforma de aprendizaje para las tecnologías web más consultada por developers en la actualidad, lo explica así: *"...cada vez más una página web hace más cosas que sólo mostrar información estática..."*. Hoy, por ejemplo, nos permite interactuar con mapas o animaciones gráficas 2D/3D.

¿Cómo podemos hacer para abordar estos desafíos continuos propios de nuestro rol? Según Yeison Daza en su [artículo](#) *"...debemos dividir estos [desafíos complejos] en problemas pequeños que podamos resolver e implementar, luego componemos estas soluciones. Esto es lo que hacemos cada día como programadores..."*.



Esto significa que a medida que se complejizan las tareas, las vamos a simplificar en pequeñas porciones de código que podamos reutilizar, ¡Este es nuestro superpoder! Convertirlos en varios pasos más sencillos.

En esta bitácora ampliarás tus conocimientos sobre JavaScript e incorporarás un concepto nuevo: las **funciones**, que son la materia prima para crear programas complejos y garantizar su mantenimiento. En la bitácora 21 hicimos una primera aproximación, y hoy profundizaremos en su uso y las diferentes maneras para escribir funciones.

Que comience la función

Una **función** es el procedimiento de un conjunto de sentencias que realizan una tarea, es decir, una serie de instrucciones ejecutadas en un mismo proceso que se van a encontrar disponibles cuando las llames. Según el artículo [“Funciones de Javascript”](#), una característica fundamental es que *“...las funciones son un objeto. Por ello se les puede asignar variables, arrays u otros objetos.”*

Nos permiten ordenar, categorizar y evitar repetir una y otra vez código en nuestros scripts. Sin ella, un software más complejo no sería posible de desarrollar ya que contaría con infinitas líneas de código desordenadas. ¡Por esto las queremos! (y queremos que aprendas a usarlas a tu favor).

Su estructura cuenta con tres partes:

- **El input (o información de entrada).** Es la información que recibe la función para tener disponible y realizar las acciones para la cual fue concebida. Si tomamos como ejemplo una función que suma 2 números, el input son **los números a sumar**.
- **El body (o cuerpo).** Esta es la porción de código que estás encapsulando con la función. Aquí es donde desarrollamos la lógica de la función. Siguiendo con el ejemplo de la suma, la **operación sumar** sería el body.
- **El output (o información de salida).** Se trata del resultado que obtendrás luego de ejecutar la función. En nuestro ejemplo retornará el **resultado de la operación** de la suma de los 2 números.



Tres pasos con funciones

Para comenzar a trabajar con funciones primero es necesario aprender a definirlas y luego dimensionar qué cosas podemos hacer con ellas. Víctor de Andrés nos explica en su artículo "[Funciones en Javascript](#)" con qué nos vamos a encontrar:

"Las funciones en Javascript, son funciones, y poseen el constructor Function. Por ello podremos crear una variable que invoque a un objeto Function. Cuando invoquemos a este objeto podremos enviar todos los argumentos que deseemos. Los primeros n argumentos serán los parámetros de nuestra función y el último argumento será el código de nuestra función."

Resulta muy útil, entonces, tener un procedimiento estándar para utilizar funciones cada vez que las necesitemos (¡y las necesitarás muy seguido!). A continuación, describimos los 3 pasos que debes realizar:

1. Definir una función.

Para usar una función primero necesitas definirla en algún lugar desde el cual luego la vas a llamar. Consiste en darle la siguiente forma:

```
function + nombre + argumentos
```

Debes utilizar la palabra clave reservada `function`, seguida por el nombre (opcional). Luego, una lista de argumentos para la función, que estarán encerrados entre paréntesis y separados por comas. La cantidad de instrucciones que va a ejecutar las debes englobar con llaves `{ }` así:

```
function nombre() {  
    instruccion 1  
    instruccion 2  
    instruccion 3  
    instruccion 4  
}
```

2. Enviar datos a las funciones.

A las funciones también les puedes enviar valores para que estén disponibles y puedan ser utilizados. A estos valores los llamamos [parámetros](#). Son los que indicamos al definir una función. Sirven para poner nombres a los valores recibidos y usarlos con esa palabra. No es



necesario que sean iguales a los indicados al invocar la función. Se pueden enviar arrays, objetos, números, cadena de texto, booleanos y hasta otras funciones:

```
function suma(valor1, valor2){
    var resultado = valor1 + valor2;
    return resultado;
}
```

3. 📱 Llamar a las funciones.

Para llamar a una función hay que invocar el nombre que definiste y enviar los parámetros que indicaste en su definición. La palabra reservada `return` devolverá el resultado al punto donde la función ha sido llamada. No solo permite definir qué queremos que la función devuelva, sino que también detiene la ejecución de una función. Si colocamos código después de un `return`, este nunca se ejecutará.

```
console.log( suma(3, 4) ) //imprime 7
console.log( suma(1, 2) ) //imprime 3
```

En determinadas ocasiones podemos escribir funciones que retornan ningún valor. En el caso que te mostramos abajo, la función ejecutará el saludo pero no devolverá nada al punto donde fue invocada, y automáticamente generará un "undefined".

```
function saludo(nombre){
    alert("Bienvenido " + nombre);
}

var lo_que_vuelve = saludo("Juan");
console.log(lo_que_vuelve); //undefined
```

Accesibilidad de las variables

El `scope` puede definirse como el alcance que tendrá nuestra variable dentro de la aplicación, y decide a qué variables tiene acceso en cada parte del código.



Existen dos tipos de scopes: local o global.

- Las variables con **scope local** son las creadas dentro de una función que sólo pueden ser accedidas desde su propia función o anidadas.
- Las variables con **scope global** son a las que puedes acceder desde cualquier parte del código.

Como aprendiste en la bitácora 20, existen tres maneras de declarar una variable: `var`, `let` y `const`. Ahora que conoces el concepto de scope, puedes volver a ellas y entender más en profundidad sus diferencias.

¿Recuerdas que `const` era utilizada en casos donde no se quiere modificar el contenido de la variable, mientras que `var` y `let` si podían ser modificadas posteriormente? Existen otras diferencias entre ellas que tienen que ver con su scope:

- **Todas pueden tener scope global**, pero su comportamiento varía al tratarse de scope local.
- **Tanto `const` como `let` tienen scope de bloque**. Es decir, sólo se podrá acceder a ellas dentro del bloque donde sean declaradas.
- `var` **tiene scope de función**. Sólo está disponible y se puede acceder a ella dentro de la función donde es declarada.
- scope distinto.
- `var` **puede ser tanto actualizada como re-declarada** sin importar el scope.
- `const` **no puede ser modificada ni re-declarada**. Dentro de su scope, `let` puede ser actualizada pero no re-declarada. Si es posible, declarar variables en ambos casos. De esta manera, con el mismo nombre en un

Arrow functions

Ahora que hemos visto el concepto de **scope**, podemos ver las funciones Arrow. Es una manera de definir funciones desde otra sintaxis y con la particularidad que nos permite definir un scope.

En las funciones arrow no es necesario utilizar la palabra reservada **function**, solamente vamos a definir su nombre

Veamos las maneras que tenemos de escribir las funciones con esta nueva sintaxis:



FORMA #1

Esta es la forma más simplificada de escribir una función. Solamente permite ejecutar una instrucción y retorna automáticamente el resultado de esta instrucción.

```
scope nombre_de_funcion = parámetro => única sentencia
```

En javascript podemos verla así:

```
let saludar = nombre => "hola " + nombre;
saludar("Josefina"); //hola Josefina
```

Si necesitamos que una función reciba más de un parámetro, los contendremos con paréntesis y separaremos cada uno con coma.

```
let saludar = (nombre, apellido) => "hola " + nombre + " "
+ apellido;
saludar("Josefina", "Suarez"); //hola Josefina Suarez
```

FORMA #2

```
scope nombre_de_funcion = parámetro => { sentencia 1,
sentencia2, sentencia 3}
```


En javascript:

```
//recibe un parámetro y retorna si es mayor a cero, menor
o igual a cero
let mayor_menor = numero => {
  if(numero > 0){
    r = 'Mayor';
  }else{
    if(numero < 0){
      r = 'Menor';
    }else{
```



```
    r = 'Cero';  
  }  
}  
return r;  
}
```

Aquí es necesario el uso de la palabra reservada **return**.

 *Mini challenge: Escribe esta función con la Forma #1 con un operador ternario.*

Creación de un Hoisting

Al declarar las variables en JavaScript existe lo que llamamos [Hoisting](#), que es el **proceso de mover declaraciones al principio del código**. Indistintamente de dónde hayan ocurrido esas declaraciones (variables o funciones) al momento de que el código es ejecutado, JavaScript moverá todas estas declaraciones a lo más “alto” —o principal— del código.

Si tuvieras una invocación de función en la línea 2, pero esta fuera declarada en la línea 40, esa invocación se haría correctamente gracias al hoisting.

Aunque escribirás funciones para agrupar porciones de tu código, éstas no siempre funcionarán como esperas. Cuando una función no puede continuar con su ejecución a causa de un error, es necesario detenerla y proceder a una parte del programa que sepa como manejar este error. Esto se llama [manejo de excepciones](#), es una técnica de programación que nos permite controlar los errores ocasionados durante la ejecución de una función. Por ejemplo, en una función desarrollada para dividir dos números, una excepción será si el divisor es cero.

Una [excepción](#) puede tomar cualquier valor. Decimos que son “arrojadas” por el programa una vez que se encuentra con ellas. Después es necesario “atraparlas” para manejar el error y permitirle al programa continuar normalmente.

En caso que haya que arrojar un error, lo que hacemos es:

- Utilizamos un constructor estándar de JavaScript que crea un objeto con una propiedad “mensaje” con la propiedad [error](#).



- Utilizamos la propiedad `keyword throw` cuando se produce una excepción y los bloques `try/catch` para atraparlo en caso de que suceda. Si el código ejecutado en el bloque `try` produce una excepción, será atrapado por el `catch` y el programa ejecutará el código contenido dentro de este segundo bloque. De lo contrario, terminará con el código del `try` y continuará ejecutándose normalmente.

Resumiendo

¡Llevas hecho un gran recorrido! Te has metido de lleno en el concepto de función, y has aprendido sobre el funcionamiento de las variables `var`, `let` y `const`, y el término *hoisting* ya no es un enigma para tí!

Recuerda que estos conceptos te acompañarán a lo largo de este Bloque y son claves para el aprendizaje de Javascript, por lo que tener claro estos conceptos tanto desde lo conceptual como desde lo práctico es absolutamente fundamental.

Live Coding

¿Te quedaste con ganas de más? ¿Prefieres escuchar más que leer para aprender? ¿Quieres ver al director de la carrera - Daniel Segovia - codear lo que te contamos en la bitácora? Aquí te dejamos un video:

[JavaScript - Funciones](#)

¡Prepárate para el próximo encuentro!

Profundiza



[Funciones Arrow](#)




[JavaScript Visualized: Hoisting](#)




[Cómo funciona el scope de JavaScript](#)



Challenge

 Escribe una función arrow que reciba como parámetro un array y calcule el promedio de los elementos tipo numérico.

Comunidad

 *Googlea como un/a developer.* Busca material en cualquier formato que explique la diferencia entre funciones y funciones arrow. Compártelo con tus compañeros/as en el próximo encuentro.

