



# JavaScript

clase n° 5

\*REC



recordá  
poner a grabar la clase

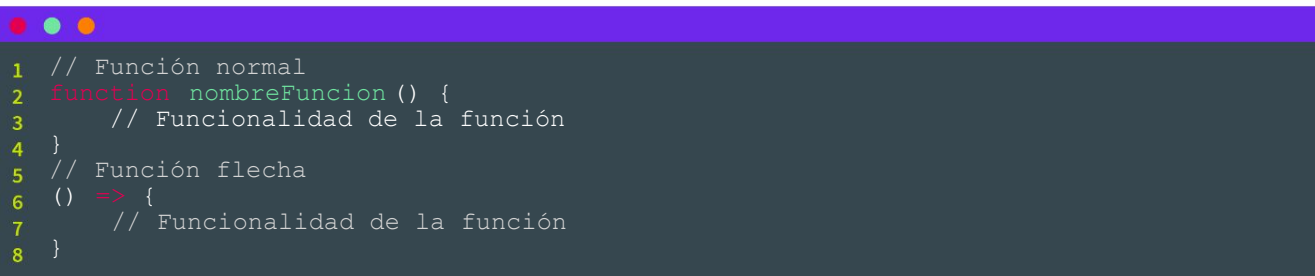
# arrows functions

Como vimos previamente en las funciones, la lógica seguirá siendo la misma. Estas siguen siendo un fragmento de código que se reserva para una utilidad específica la cual vamos a llamar o ejecutar según sea necesario.

Entonces ¿En qué se diferencia una función normal contra una función flecha? Bien, para empezar debemos entender que una función normal se declara y almacena en memoria mientras que una función flecha no contiene una declaración como tal, de ahí proviene su otra manera de nombrarlas como “Funciones anónimas”. En breves entenderán porque y como es que sucede lo explicado anteriormente.

Vayamos por el principio, su estructura base





```
1 // Función normal
2 function nombreFuncion() {
3     // Funcionalidad de la función
4 }
5 // Función flecha
6 () => {
7     // Funcionalidad de la función
8 }
```

Como se ve en la imagen, en la primera función le estamos asignando un nombre para poder llamarla y/o ejecutarla, mientras que en la segunda no contamos con un nombre, solamente está definida pero podría decirse que sin una identificación.

Contexto: Esta asignación de un nombre corresponde a el scope o contexto de un objeto, está el objeto base el cual se conoce como **this** y este al ser una función normal se le da contexto dentro del **this**, mientras que una función anónima funciona con un contexto heredado, quiere decir que asigna la llegada o referencia del objeto más cercano.

# partes de la función

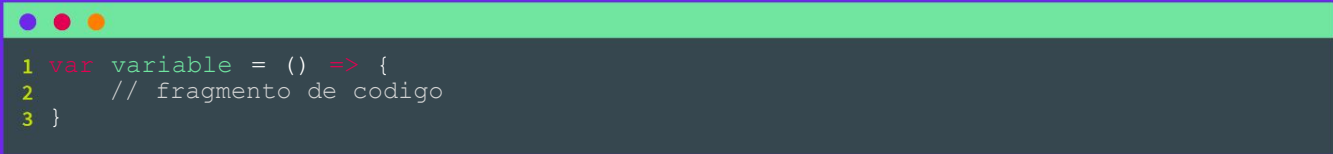
**() paréntesis** vamos a estar declarando donde van los parámetros, que vendrían siendo los valores que le vamos a la función.

**=> flecha** realizará la tarea de “asignar” el siguiente fragmento de código que va dentro de las llaves {}.

**{ } llaves** contendrán todo el fragmento de código que se ejecutara dentro de nuestra función, tal cual hacíamos con las funciones declaradas.

# actividad

*Al igual que como se realizó en la clase anterior, vamos a estar creando una función que tome un parámetro edad y retorne una cadena que diga "Tienes X años" pero a diferencia de que esta función se la vamos a estar asignando a una variable como si fuera un valor. Ejemplo:*



```
1 var variable = () => {  
2   // fragmento de codigo  
3 }
```

*Una vez que nosotros hagamos esto, nuestra variable de ahora en más se comportara como una función declarada, por lo que para ejecutarla necesitamos hacer solamente variable().*

# actividad

*Otra actividad que se puede realizar para ver mejor la funcionalidad de una arrow function es utilizar la función `setTimeout` que nos provee JS para poder realizar una acción cualquiera, en este caso vamos a estar mostrando un mensaje a X segundos de cargar nuestro código, este mensaje puede ser un `console.log()` o `alert()` según les guste más.*


*Primero una breve explicación como usar `setTimeout()`.*

```
1 // setTimeout(function, delay)
2 var delay = 1000 // 1000ms o 1s
3 setTimeout( () => {}, delay ) // Utilizando variable para el delay
4 setTimeout( () => {}, 1000 ) // Declarando el delay en el setTimeout
5
```

# return

Se entiende como el final de una ejecución, el cual su función en específico es realizar el **retorno o devolución de un valor** a donde o quien realice la ejecución.

Su uso se realiza de la siguiente manera:



```
1 function Prueba() {  
2     /* Fragmento de codigo */  
3     return  
4 }
```

Ahí mismo de esta realizando un retorno de básicamente nada, por lo que se entiende como un undefined.

Si nuestra meta es devolver un valor resultante o una variable, este mismo debe ser enviado después del return sobre la misma línea, de la siguiente manera.





```
1 function Prueba() {  
2   /* Fragmento de codigo */  
3   var variable = "Hola soy el valor devuelto"  
4   return variable  
5 }
```

Este valor también puede provenir de una operación realizada sobre el return, como por ejemplo de la siguiente manera.

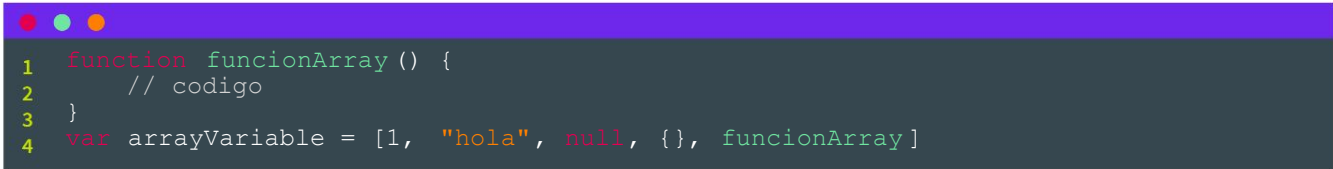
```
1 function Suma(a,b) {  
2   return a + b  
3 }
```

En las funciones flecha se utiliza exactamente de la misma manera.

```
1 () => {  
2   // fragmento de codigo  
3   return;  
4 }  
5 () => {  
6   var variable = "Hola soy el valor devuelto"  
7   return variable  
8 }  
9 (a,,b) => {  
10   return a + b  
11 }
```

# arrays con bucles base

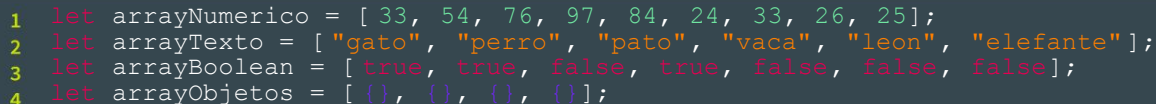
Primero que nada hagamos un repaso sobre qué son los **arrays o vectores**, son variables las cuales **acumulan información** dentro de ellas de manera separada por un espacio de memoria, para decirlo de una manera más “criolla” podríamos explicarlo como que el es una **variable que guarda variables**, y cada una de estas está identificada por una posición.

A code editor window with a dark background and a purple title bar. It contains four lines of JavaScript code, numbered 1 to 4 on the left. Line 1: `function funcionArray () {`. Line 2: `// codigo`. Line 3: `}`. Line 4: `var arrayVariable = [1, "hola", null, {}, funcionArray]`.

```
1 function funcionArray () {  
2     // codigo  
3 }  
4 var arrayVariable = [1, "hola", null, {}, funcionArray]
```

Este ejemplo es simple para poder entender la manera de declarar y que puede almacenar dentro, como vemos podemos guardar cualquier tipo de información.

Ahora yendo a lo más importante, por normas generales dentro de un array se suele guardar el mismo tipo de información, ¿a qué nos referimos? Que en caso de que tengamos texto dentro de un array, todo va a ser texto, lo mismo si son números, objetos, booleanos, funciones, etc.



```
1 let arrayNumerico = [ 33, 54, 76, 97, 84, 24, 33, 26, 25];  
2 let arrayTexto = [ "gato", "perro", "pato", "vaca", "leon", "elefante"];  
3 let arrayBoolean = [ true, true, false, true, false, false, false];  
4 let arrayObjetos = [ {}, {}, {}, {}];
```

La idea de que sean del mismo tipo es tanto de manera literal, como conceptual preferentemente. Si tenemos nombres dentro de nuestro array, todos deben ser nombres, si tenemos edades, todos deben ser edades, si tenemos animales... etc. Ya que ahora veremos maneras comunes de trabajar con arrays recorriéndolos y haciendo algún tipo de manejo.

# recorrer arrays

## Método bucle FOR

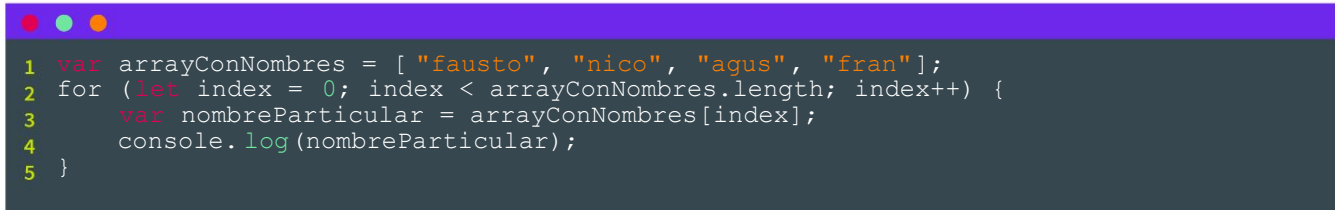
Para el primer método vamos a estar trabajando con la estructura repetitiva FOR, recuerden que su manera de trabajar es la siguiente:

```
1 for (/* Donde comienza */; /* Hasta donde llega */; /* Manera de progresar */ ) {  
2     // estructura a repetir  
3 }
```

Ahora vamos a ver como se debe actuar al momento de querer recorrer un array, esto tiene una lógica muy simple.

Como sabemos, cada elemento del array está identificado con una posición, mayormente conocida como index. Esta comienza desde la posición 0 y va incrementando por cada elemento que se encuentra dentro del array hasta el último. Bueno, nosotros tendremos para recorrer un bucle algo similar, también identificado como index el cual es la vuelta que se está recorriendo y va a ir incrementando por cada vuelta que se realiza del mismo. Podríamos asociar ambos index en cuando a la funcionalidad, ¿Qué queremos decir? Vayamos a verlo a nivel código.





```
1 var arrayConNombres = ["fausto", "nico", "agus", "fran"];
2 for (let index = 0; index < arrayConNombres.length; index++) {
3   var nombreParticular = arrayConNombres[index];
4   console.log(nombreParticular);
5 }
```

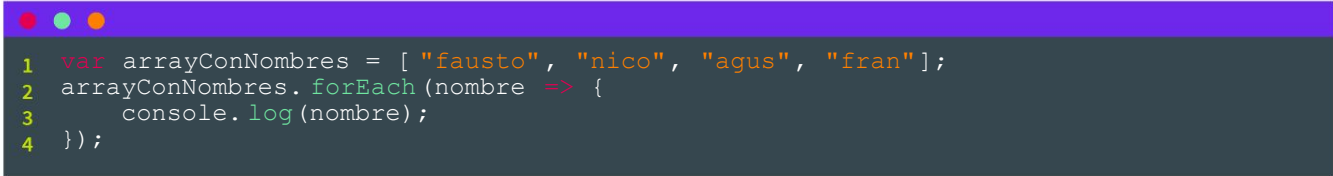
Como se ve en la imagen, tenemos **arrayConNombres** el cual almacena un conjunto de nombres. Este para recorrerlo decimos que vamos a iniciar el index desde 0, como la primer posición de nuestro array, después especificamos que vamos a recorrer hasta el último elemento del array pero con un pequeño detalle, usamos `.length` para sacar la totalidad de elementos, esto es distinto a la última posición, es un número más (si mi última posición es 5, el `.length` devuelve 6 ya que en las posiciones arrancamos contando desde el 0).

Ya con esta estructura base podemos ver que si realizamos un llamado a la posición que tenga el valor de index vamos a estar llamando a cada uno de los elementos de nuestro array por dentro del bucle. Esto nos sirve por si queremos realizar una acción por cada uno o alguna validación, etc. Dentro del bucle podemos implementar lo que creamos necesario.

# recorrer arrays

## Método bucle FOREACH

El bucle FOREACH recuerden que quiere decir “**Por cada uno**”, esta es una función la cual sirve propiamente para los arrays. Es la más específica para recorrer los mismos, y surge como solución a la gran estructura del bucle FOR. La misma se escribe de la siguiente manera:



```
1 var arrayConNombres = [ "fausto", "nico", "agus", "fran" ];  
2 arrayConNombres.forEach (nombre => {  
3   console.log (nombre);  
4 });
```

Como pueden ver, tenemos el array el cual contiene nombres y debajo de la misma ponemos **.forEach** el cual es una función sacada del array y como parámetro se le pasa una función anónima o Arrow Function, con la cual lleva de parámetro lleva la representación de cada iteración de nuestro bucle, en este caso sería un nombre, para luego dentro de la función poder hacer un `console.log()` de la misma.



revolución\*  
digital\_