

FUNCIONES



► Funciones

- Se declaran con **function** y se les pasan los parámetros entre paréntesis. La función puede devolver un valor usando **return** (si no tiene *return* es como si devolviera *undefined*).
- Puede usarse una función antes de haberla declarado por el comportamiento de Javascript llamado *hoisting*: el navegador primero carga todas las funciones y mueve las declaraciones de las variables al principio y luego ejecuta el código.

EJERCICIO: Haz una función que te pida que escribas algo y muestre un alert diciendo 'Has escrito...' y el valor introducido. Pruébala en la consola (pegas allí la función y luego la llamas desde la consola)

- Podemos dar un **valor por defecto** a los parámetros por si no los pasan asignándoles el valor al definirlos:

```
function potencia(base, exponente=2) {  
  console.log(base);           // muestra 4  
  console.log(exponente);      // muestra 2 la primera vez y 5 la segunda  
  let valor=1;  
  for (let i=1; i<=exponente; i++) {  
    valor=valor*base;  
  }  
  return valor;  
}  
  
console.log(potencia(4));      // mostrará 16 (4^2)  
console.log(potencia(4,5));    // mostrará 1024 (4^5)
```

NOTA: En ES5 para dar un valor por defecto a una variable se hacía

```
function potencia(base, exponente) {  
  exponente = exponente || 2;    // si exponente vale undefined se la asigna el valor 2  
  ...  
}
```

- En Javascript las funciones son un tipo de datos más por lo que podemos hacer cosas como pasarlas por argumento o asignarlas a una variable:

```
const cuadrado = function(value) {  
  return value * value  
}  
function aplica_fn(dato, funcion_a_aplicar) {  
  return funcion_a_aplicar(dato);  
}  
  
aplica_fn(3, cuadrado);    // devolverá 9 (3^2)
```

► Funciones anónimas

- Como acabamos de ver podemos definir una función sin darle un nombre. Dicha función puede asignarse a una variable, autoejecutarse o asignarse a un manejador de eventos. Ejemplo:

```
<script>
(function () {
    console.log("Welcome to GeeksforGeeks!");
})();
</script>
```

```
let holaMundo = function() {
    alert('Hola mundo!');
}

holaMundo();           // se ejecuta la función
```

Función tradicional anónima
Con argumento y una línea
con return

```
function (a){  
  return a + 100;  
}
```



Funciones flecha

```
// 1. Elimina la palabra "function" y coloca la flecha entre el argumento y el  
corchete de apertura.
```


```
(a) => {  
  return a + 100;  
}
```

```
// 2. Quita los corchetes del cuerpo y la palabra "return" – el return está  
implícito.
```

```
(a) => a + 100;
```

```
// 3. Suprime los paréntesis de los argumentos
```

```
a => a + 100;
```

- 
- ▶ **Arrow functions (funciones *lambda*)**
 - ▶ ES2015 permite declarar una función anónima de forma más corta.
Ejemplo sin *arrow function*:

EJERCICIO: Haz una *arrow function* que devuelva el cubo del número pasado como parámetro y pruébala desde la consola. Escríbela primero en la forma habitual y luego la “traduces” a *arrow function*.

Si tienes **varios argumentos** o **ningún argumento**, deberás volver a introducir paréntesis alrededor de los argumentos

```
function (a, b){  
  return a + b + 100;  
}
```



```
// Función flecha  
(a, b) => a + b + 100;
```

```
// Función tradicional (sin argumentos)  
let a = 4;  
let b = 2;  
function (){  
  return a + b + 100;  
}
```



```
// Función flecha (sin argumentos)  
let a = 4;  
let b = 2;  
() => a + b + 100;
```


Si el cuerpo requiere **líneas de procesamiento adicionales**, deberás volver a introducir los corchetes **Más el "return"** (las funciones flecha no adivinan mágicamente qué o cuándo quieres "volver"):

```
// Función tradicional
function (a, b){
  let chuck = 42;
  return a + b + chuck;
}
```



```
// Función flecha
(a, b) => {
  let chuck = 42;
  return a + b + chuck;
}
```

- **funciones con nombre** tratamos las expresiones de flecha como variable

```
// Función tradicional  
function bob (a){  
  return a + 100;  
}
```



```
// Función flecha  
let bob = a => a + 100;
```

Anidación de funciones

Una función anidada dentro de otra tiene acceso al contexto de ejecución de su función padre, pero su función padre no tiene acceso a su contexto de ejecución.

La función anidada sólo puede ser llamada desde dentro de la función padre.

Closure

Si una función devuelve una referencia a una función anidada dentro de ella se denomina closure

```
<!DOCTYPE html>
<html lang="en">
<head>
<script>
```

```
function iniciar() {
  var nombre = "Mozilla"; // La variable nombre es una variable local creada
  function mostrarNombre() { // La función mostrarNombre es una función interna, una clausura.
    alert(nombre); // Usa una variable declarada en la función externa.
  }
  mostrarNombre();
}
iniciar();
```

```
</script>
</head>
```

```
<body>
```

```
</body>
</html>
```

La función `iniciar()` crea una variable local llamada `nombre` y una función interna llamada `mostrarNombre()`. Por ser una función interna, esta última solo está disponible dentro del cuerpo de `iniciar()`. Notemos a su vez que `mostrarNombre()` no tiene ninguna variable propia; pero, dado que las funciones internas tienen acceso a las variables de las funciones externas, `mostrarNombre()` puede acceder a la variable `nombre` declarada en la función `iniciar()`.

¿Cómo se reciben los argumentos, por valor o por referencia? Como en cualquier otra asignación; todos por valor excepto los objetos (y sus descendientes como Arrays y funciones) que se reciben por referencia.

```
001  var cantidades = new Array(3,5,7,9);
002  function suma(sumandos) {
003      var i;
004      var resultado = 0;
005      while (sumandos.length>0) {
006          resultado += sumandos.shift();
007      }
008      return resultado;
009  }
010  alert(suma (cantidades));
011  alert(cantidades.length);
```

Un closure permite pasar argumentos a la función anidada.

```
function exterior(cadena1){  
  function interior(cadena2){  
    return cadena1 + cadena2;  
  }  
  return interior;  
}  
alert(exterior ('hola ')('juanfe'));
```