

FUNCIONES

FUNCIONES

[enlace a la documentación](#)

- ▶ **Son la base de JavaScript** y, por tanto de TypeScript.
- ▶ Describen **cómo** hacer las cosas.
- ▶ Se pueden crear con nombre o sin nombre:

```
1 // Named function
2 function add(x, y) {
3     return x + y;
4 }
5
6 // Anonymous function
7 let myAdd = function (x, y) { return x + y; };
8 console.log(myAdd(3, 5));
```

- ▶ En JavaScript, las funciones **pueden hacer referencia a variables que estén fuera de la función**:

```
let z = 100;

function addToZ(x, y) {
    return x + y + z;
}
```

FUNCIONES

[enlace a la documentación](#)

- ▶ Se puede **indicar el tipo de los parámetros y el tipo que devuelven**.
- ▶ Si no lo ponemos, lo infiere (si puede):

```
function add(x: number, y: number): number {  
    return x + y;  
}
```

```
let myAdd = function(x: number, y: number): number { return x + y; };
```

FUNCIONES

[enlace a la documentación](#)

- ▶ **IMPORTANTE**: Existe el **tipo función** que indica a una variable, o propiedad,... que va a ser de tipo función y en concreto le especifica de qué tipo de función va a ser.

Ej:

```
1 let myAdd: (x: number, y: number) => number =  
2     function (x: number, y: number): number { return x + y; };  
3  
4 console.log(myAdd(3, 2));
```

Estamos diciendo que “miAdd” es una variable de tipo **función**,

– en concreto será una función con esta pinta:

(x: number, y: number) => number

– Es decir, recibe 2 números y devuelve otro número.

– Además, estamos **inicializando** la variable con la función:

“function (x:number,y:number):number {return x+y;};

- ▶ **El nombre de los parámetros en el tipo y la inicialización no tiene por qué coincidir:**

```
let myAdd: (baseValue: number, increment: number) => number =  
    function(x: number, y: number): number { return x + y; };
```

FUNCIONES

[enlace a la documentación](#)

Parámetros opcionales y por defecto

► Si hacemos lo siguiente, obtendremos los errores que véis:

```
function buildName(firstName: string, lastName: string) {  
    return firstName + " " + lastName;  
}  
  
let result1 = buildName("Bob"); // error, too few parameters  
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters  
let result3 = buildName("Bob", "Adams"); // ah, just right
```

► Para evitar esos errores podemos usar **parámetros opcionales**:

```
function buildName(firstName: string, lastName?: string) {  
    if (lastName)  
        return firstName + " " + lastName;  
    else  
        return firstName;  
}  
  
let result1 = buildName("Bob"); // works correctly now  
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters  
let result3 = buildName("Bob", "Adams"); // ah, just right
```

FUNCIONES

[enlace a la documentación](#)

Parámetros opcionales y por defecto

► Además podemos dar **valores por defecto**:

```
function buildName(firstName: string, lastName = "Smith") {  
    return firstName + " " + lastName;  
}  
  
let result1 = buildName("Bob"); // works correctly now, returns "Bob Smith"  
let result2 = buildName("Bob", undefined); // still works, also returns "Bob Smith"  
let result3 = buildName("Bob", "Adams", "Sr."); // error, too many parameters  
let result4 = buildName("Bob", "Adams"); // ah, just right
```

FUNCIONES

[enlace a la documentación](#)

Resto de parámetros

► Cuando una función tiene unos parámetros fijos y otros opcionales, pero que pueden ser muchos, podemos usar “**rest parameters (se representa con ...)**”:

```
function buildName(firstName: string, ...restOfName: string[]) {  
    return firstName + " " + restOfName.join(" ");  
}  
  
let employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");
```

- El compilador mete todos los parámetros en un **array** con ese nombre (*restOfName en nuestro ejemplo*).
- Cuando se **define una variable de tipo función** que lleve restOfParams, también hay que poner los “...”:

```
1 function buildName(firstName: string, ...restOfName: string[]) {  
2     return firstName + " " + restOfName.join(" ");  
3 }  
4  
5 let buildNameFun: (fname: string, ...rest: string[]) => string = buildName;  
6 console.log(buildNameFun("Pedro", "Lucas", "Juan"));
```

FUNCIONES

[enlace a la documentación](#)

Funciones flecha o arrow functions

► Son un atajo para escribir funciones:

```
1 function suma(a: number, b: number) {  
2   return a + b;  
3 }  
4  
5 console.log(suma(5, 4));
```

```
7 let sumaFlecha = (a: number, b: number) => a + b;  
8 console.log(sumaFlecha(5, 4));
```

- Fijaros en los 2 cuadros rojos: **SON EQUIVALENTES con diferencias:**
 - Las funciones => son **SIEMPRE ANÓNIMAS** (ver cuadro derecha)
 - Tampoco generan su propio **this**, comparten el de la función donde estén encerradas.

```
1 var nate = {  
2   name: "Nate",  
3   guitars: ["Gibson", "Martin", "Taylor"],  
4   printGuitars: function() {  
5     var self = this;  
6     this.guitars.forEach(function(g) {  
7       // this.name is undefined so we have to use self.name  
8       console.log(self.name + " plays a " + g);  
9     });  
10  }  
11 };  
12  
13 nate.printGuitars();
```

```
1 var nate = {  
2   name: "Nate",  
3   guitars: ["Gibson", "Martin", "Taylor"],  
4   printGuitars: function() {  
5     var self = this;  
6     this.guitars.forEach(  
7       g => console.log(this.name + " plays a " + g));  
8   }  
9 };  
10  
11 nate.printGuitars();
```


FUNCIONES

[enlace a la documentación](#)

Funciones flecha o arrow functions

3. Tampoco se puede usar el objeto **arguments**


- Es una variable que poseen todas las funciones javascript
- Esa variable almacena en forma de array los argumentos pasados a una función: arguments[0], arguments[1],...

► Se usan **MUCHO**

► Otro ejemplo:

```
1 // ES5-like example
2 var data1 = ['Alice Green', 'Paul Pfifer', 'Louis Blakenship'];
3 data1.forEach(function (line) { console.log(line); });
```

```
5 // Typescript example
6 var data: string[] = ['Alice Green', 'Paul Pfifer', 'Louis Blakenship'];
7 data.forEach((line) => console.log(line));
```



FUNCIONES

enlace a la documentación

Funciones 'flecha' (arrow) TS

- Más conciso, 'return' opcional, mantienen 'this' del padre

```
var inc = function(x) { return x+1; };    // JS
var inc = (x) => x+1;                      // TS
```

```
var multiply = function(x, y) {            // JS
  return x * y;
};
let multiply = (x: number, y: number) => { return x * y }; // TS
```

```
var foo = () => {                          var foo = () => ({
  bar: 123                                bar: 123
});                                       });
```



En este ejemplo de foo, si queremos **devolver el objeto {bar:123}** hay que encerrarlo entre (), como se ve en el **lado derecho**. En otro caso devuelve undefined y da un warning.

FUNCIONES

[enlace a la documentación](#)

Funciones flecha o arrow functions

► Los paréntesis son opcionales.

Más ejemplos:

```
1 var evens = [2,4,6,8];  
2 var odds = evens.map(v => v + 1);
```

```
data.forEach( line => {  
    console.log(line.toUpperCase())  
});
```

Ver este [ENLACE](#) más completo

FUNCIONES

[enlace a la documentación](#)

NECESIDAD DE LAS FUNCIONES FLECHA:

- *PROBLEMA variable THIS:*

- ❑ Cuando usamos la variable “this” dentro de una función que está dentro de otra función, this pierde su referencia real y apuntaría al objeto WINDOW, con lo cual, el resultado no sería el esperado.

- ❑ Ejemplo:

```
let deck = {
  suits: ["hearts", "spades", "clubs", "diamonds"],
  cards: Array(52),
  createCardPicker: function() {
    return function() {
      let pickedCard = Math.floor(Math.random() * 52);
      let pickedSuit = Math.floor(pickedCard / 13);

      return {suit: this.suits[pickedSuit], card: pickedCard % 13};
    }
  }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

FUNCIONES

[enlace a la documentación](#)

NECESIDAD DE LAS FUNCIONES FLECHA:

- *SOLUCIÓN variable THIS:*

- ❑ Debemos cambiar la función que hay en “return” por una función en **NOTACIÓN FLECHA**. Ya que **las funciones flecha capturan el “this” dónde la función es creada, NO DONDE ES LLAMADA.**

- ❑ Quedaría:

```
let deck = {
  suits: ["hearts", "spades", "clubs", "diamonds"],
  cards: Array(52),
  createCardPicker: function() {
    return () => {
      let pickedCard = Math.floor(Math.random() * 52);
      let pickedSuit = Math.floor(pickedCard / 13);

      return {suit: this.suits[pickedSuit], card: pickedCard %
13};
    }
  }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

FUNCIONES

[enlace a la documentación](#)

Ejercicio: Filtrar una colección de elementos

- Filtrar los elementos de un `number[]` que son > 0

```
let col: number[] = [-3, -2, -1, 0, 1, 2, 3];  
let ret: number[] = [];  
for (let val of col) { if (val > 0) ret.push(val); }
```

- Filtrar un `number[]` por el valor de retorno de una función

```
function fn(x) { return x > 0 }  
for (let val of col) { if (fn(val)) ret.push(val); }
```

FUNCIONES

[enlace a la documentación](#)

- Definir función filter() que filtre un number[] por el valor de retorno de una función

```
function filter(col: number[], fn: (number)=>boolean) {  
    let ret: number[] = [];  
    for (let val in col) { if (fn(val)) ret.push(val); }  
    return ret;  
}
```

```
function fn1(x: number) { return x < 0; }
```

```
function fn2(x: number) { return x > 0; }
```

```
console.log(filter(col, fn1));
```

```
console.log(filter(col, fn2));
```

FUNCIONES

[enlace a la documentación](#)

- Definir función filter() que filtre cualquier vector por el valor de retorno de una función

```
function filter(col: any[], fn: (any)=>boolean) {  
    let ret: any[] = [];  
    for (let val of col) { if (fn(val)) ret.push(val); }  
    return ret;  
}
```

```
function fn1(x: string) { return x.length > 3; }  
function fn2(x: number) { return x > 0; }
```

```
let arrayString: string[] = ["María", "Ana", "Juan"];  
console.log(filter(arrayString, fn1));  
console.log(filter(col, fn2));
```