# Funciones

*Una* ***función*** *es un* ***bloque*** *de* ***código*** *que* ***realiza*** *una* ***tarea específica.***

**Ventajas** de usar funciones:

* Nos permiten definir tareas específicas.
* Nos permite evitar duplicidad de código.
* Solo tenemos que llamar la función para que se ejecute.

# Formas de llamar una función

Las **funciones** pueden **Invocarse** de **4 formas distintas:**

* Forma de función global.
* Forma de método de un objeto.
* Forma de constructor.
* Forma de apply.

Además **es muy importante** saber cómo **funcionan cada una de ellas** ya que su *this* **no siempre** apunta al **elemento** que **nosotros esperamos**.

## Forma de función global

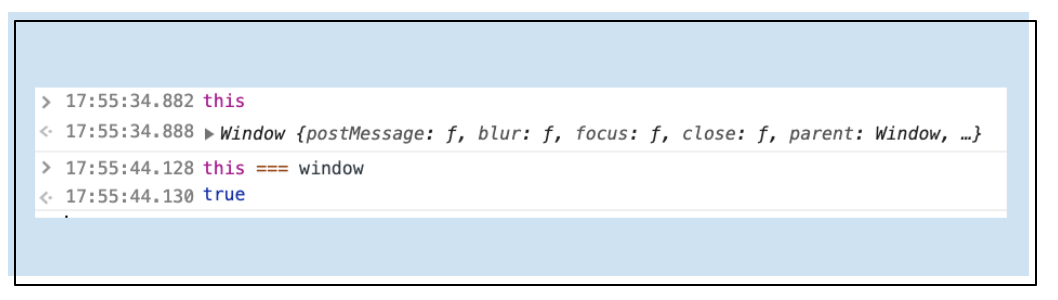
function foo(){

console.log('Hello');

}

foo();

Cuando la función es llamada como una **función global**, *this* se **establece** como **objeto global,** entonces sabemos que window es un objeto **global** en el navegador, si escribimos *this* en la **consola**, debería devolver el objeto de *window*.



Una función definida a nivel **global** su *this* sigue siendo *window*, ya que su **contexto es el mismo (global),**veamos el siguiente ejemplo:

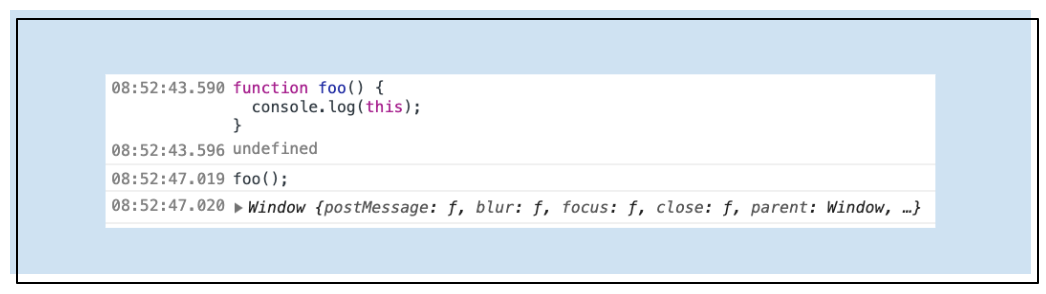
/\* Ejemplo con el contexto global \*/

function foo() {

console.log(this);

}

foo();



La **función** *foo* se **define** a nivel **global** y su objeto se llama dentro de *window,* es decir *window* llama a *foo*, por lo que *window.foo* y *foo* son lo mismo.

Para poder **manipular** el *this*, **esta forma no es muy útil** ya que no podemos trabajar con ella de manera **correcta y segura**, ya que estaríamos **manipulando** el *this* **global**.

Podemos evitar “equivocarnos”, si hacemos uso del *use strict*, veamos el siguiente ejemplo:

'use strict';

function foo(){

console.log(this);

}

foo(); //output undefined

Podemos observar que el *this* **ya no es global**, pero nos va a **retornar** *undefined*, digamos que tampoco nos va a ser útil, pero podemos evitar trabajar con el *this* **global.**

## Forma de constructor (new)

Si creamos una **instancia** con el **operador** *new* como **prefijo**, lo que estaremos haciendo es la **instancia** de la función *foo* (**local**), por lo que el **contexto** de *this* cambia a ser la **función**.

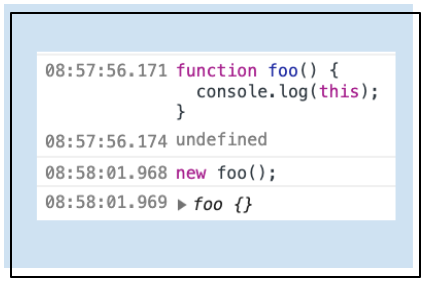
/\* Ejemplo con el contexto local \*/

function foo() {

console.log(this);

}

new foo();



## Forma de método de un objeto

Si creamos un **objeto**, y dentro **creamos** una **función**, el *this* va a hacer **referencia** a su objeto:

const foo = {

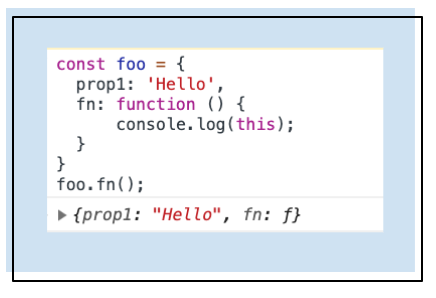
fn: function () {

console.log(this);

}

}

foo.fn();



Solo hay que tener **cuidado** y ver donde se está **ejecutando** el **contexto** de la **función**, ya que **puede cambiar** muy fácilmente. Con el ejemplo anterior, solo hagamos un pequeño cambio:

const foo = {

fn: function () {

console.log(this);

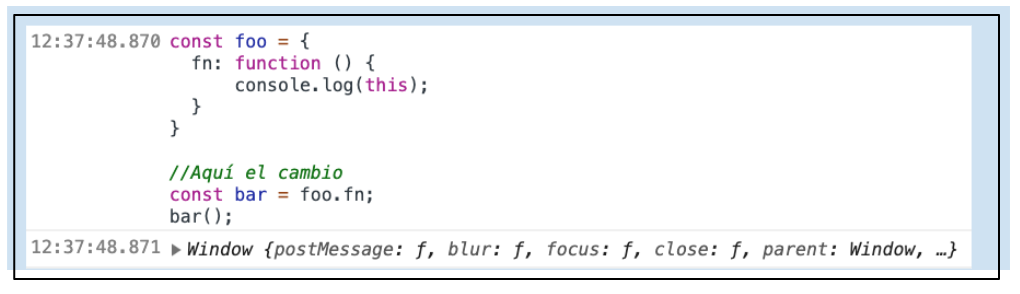
}

}

//Aquí el cambio

const bar = foo.fn;

bar();



Como podemos observar su **contexto** ha **cambiado**, ya que **no** es el del **objeto** *foo*, si no que su **contexto** es **global**.

¿Cómo podemos saber a dónde o a quién apunta el *this*?

El truco es *ver* ***quién es el de la izquierda.***

Veamos un ejemplo.

foo.fn(); //Su contexto es foo, por que esta a su izquierda

const bar = foo.fn;

window.bar(); //Su contexto es global (window), por que esta a la izquierda del punto

## Forma de apply

Esta forma de hacerlo nos permite **especificar de forma explícita** quien va a ser el *this* de la **función**, además podemos **mandarle un arreglo** (*apply*) o **secuencia de parámetros** (*call*); entonces:

*El método* ***apply()*** *invoca una determinada función asignando explícitamente el objeto* this *y un* ***array*** *o similar (array like object) como parámetros (****argumentos****) para dicha función.*

//Creamos un objeto con dos propiedades y una función

const person = {

name: 'Mauricio',

lastname: 'García',

fullname: function() {

return this.name + ' ' + this.lastname;

}

}

//Función

function print(greet, adj) {

console.log(greet, this.fullname(), 'you are', adj);

}

//Le decimos que el this de la funnción será person

print.apply(person, ['hello', 'special']);

//output hello Mauricio Garcia you are special

*El método* ***call()*** *llama a una función con un valor* this *asignado y* ***argumentos provistos de forma individual.***

//Creamos un objeto con dos propiedades y una función

const person = {

name: 'Mauricio',

lastname: 'García',

fullname: function() {

return this.name + ' ' + this.lastname;

}

}

//Función

function print(greet, adj) {

console.log(greet, this.fullname(), 'you are', adj);

}

//Le decimos que el this de la función será person

print.call(person, 'hello', 'special');

//output hello Mauricio Garcia you are special

En **JavaScript** las funciones se definen de dos formas:  
1. **Funciones** de **primera clase** (**first-class**).

2. **Entidades** de **orden superior**.

# Funciones de primera clase

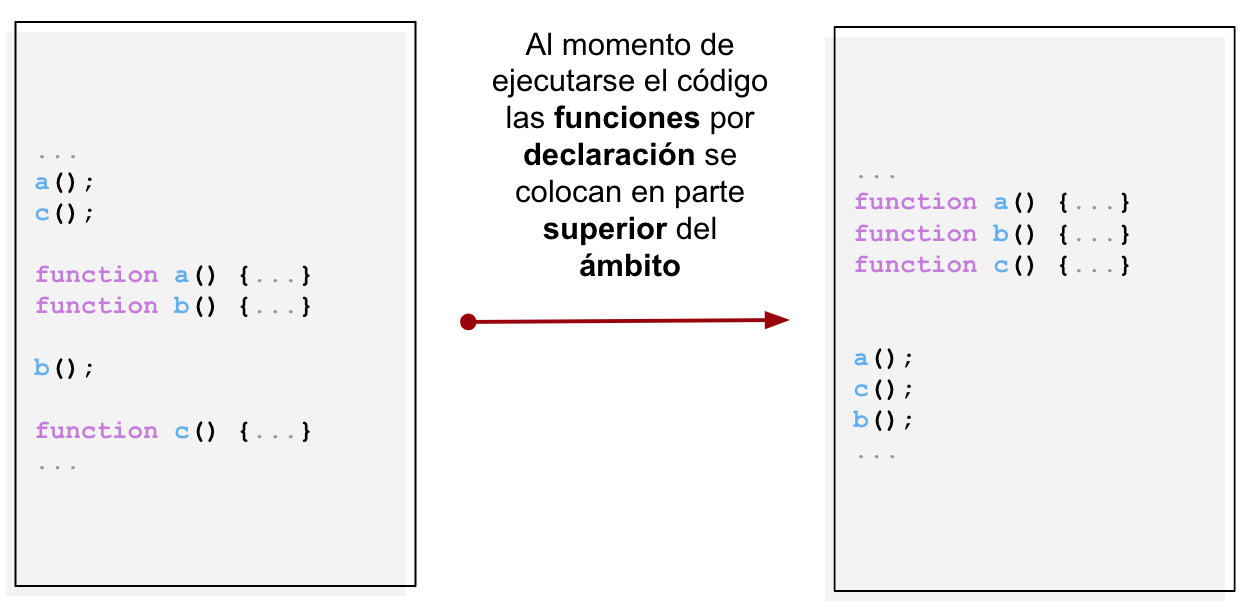
Las **funciones** en **JavaScript** son **objetos** de **primera clase**, ya que son **objetos** que se pueden **manipular** y **transmitir** al igual que **cualquier otro objeto**, es decir es un elemento que **soporta todas** las **operaciones** **generalmente disponibles** para otras **entidades** importantes como pueden ser los **objetos**.

En **JavaScript** tenemos **dos tipos** de funciones:

* Por **declaración**
* Por **expresión** (asignación a variable)

## Funciones por declaración

Las funciones **declaradas** se **colocan** en la parte **superior** del **ámbito** en el que se **definen**, lo que permite **utilizar** la **función antes** de su **declaración.**



doSomething();

function doSomething() {

// …

}

El ejemplo anterior es válido, ya que como se explicó: las **funciones declaradas** se **colocan** en la parte **superior** del **ámbito.**

Realmente **JS** hace algo así:

//"Sube todas las funciones"

function doSomething(){

//...

}

doSomething();

## Expresión

Si estás **asignando** una **función** a una **variable** o **propiedad** estás tratando con una **función** por **expresión** ya que éstas se pueden **tratar** como **cualquier otro valor**, esto quiere decir que se  **puede almacenar** en **variables** o **propiedades** y se **puede** pasar **por parámetros** y a su vez **devolver funciones.**

//Expresión

var myFunction = function () {

// …

};

Para las de **expresión**, **primero** debemos **definir** la **función** y **después usarla**, ya que puede **generar** un **error**.

doSomething(); // error!

var doSomething = function () {

// …

};

Una **función creada** en el **contexto** de una **expresión** es también una **función** por **expresión.**

// Todo lo que está dentro de los paréntesis es parte de una expresión

(function () { /\* código aquí \*/ });

# Funciones de orden superior o alto orden

Las **funciones** que **llaman a otras funciones** o que **devuelven funciones**, se conocen como *funciones de orden superior*, muchas veces también recibe el nombre de *función de devolución de llamada* (**callbacks**\*\*)

Entonces las funciones son capaces de:

1. Tomar una función como parámetro.
2. Retornar la función enviada como parámetro u otra función.

Ejemplo:

function validateAge(n, fn) {

let age = 18;

if(n >= age) {

//Al cumplir con la condición ejecuta la función

fn();

}

}

function isOlder(){

console.log('Is older!!');

}

validateAge(20, isOlder); //output Is older

En el ejemplo anterior la función *validateAge* **recibe 2** parámetros:

* Una variable numérica (n)
* Una *función* (*isOlder*)

Si cumple con la condición invoca la función que se le envió por parámetro.

Veamos un ejemplo más complejo:

function map(list, fn) {

let n = list.length;

for (let x = 0; x < n; x++) {

fn(list[x]); //Regresa el valor en la función que se envio por parámetro

}

}

//Tenemos un arreglo de numeros

let list = [1, 2, 3, 4, 5];

//Creamos un arreglo vacio donde vamos a guardar nuestros nuevos valores

let squared = [];

//Ejecutamos la función

map(list, function (value) {

squared.push(value \* value);

});

console.log(squared); //Output 1,4,9,16,25

El ejemplo anterior es una forma muy parecida (*quizás mal hecha*) de cómo funciona el **map** de **JS**, lo que hace **map** es aplicar la función recibida como parámetro sobre cada elemento de una colección.

const list = [1, 2, 3, 4, 5];

const squared = list.map(x => x \* 2);

console.log(squared); // output => [1, 4, 9, 16, 25]

# Funciones anónimas

**JavaScript** permite crear **funciones anónimas**, esto es una función sin nombre:

(function () {

console.log('Yo soy una función sin nombre y me ejecuto automáticamente');

}());

Las **funciones anónimas**, también tienen la capacidad de **recibir parámetros**:

(function (a, b, c) {

let sum = a + b + c;

console.log('La suma de a + b + c =', sum); // output La suma de a + b + c = 6

}(1, 2, 3));

Estas funciones reciben el **nombre** de **IIFE.**

# Immediately Invoked Function Expression (IIFE)

Una de las **principales** razones para usar las **IIFE** es por la **privacidad** de los **datos**, ya que las variables declaradas con *var* en **JavaScript** su **alcance** es a **nivel función**, cuando usamos las **IIFE**, cualquier **variable** declarada dentro, no puede ser usada fuera:

(function () {

var foo = 'abc';

console.log(foo); //output abc

})();

console.log(foo); // ReferenceError: foo is not defined

Otra forma de ejecutar las funciones:

// Classic

(function () { })();

// Crockford's

(function () { }());

// Facebook version

!function () { }();

# return Statement

En **JavaScript** las **funciones** tienen **declaraciones de retorno** (*return statement*); eso quiere decir que **detiene por completo la ejecución de una función** y ésta **devuelve el control a quien la invocó** y además **puede o no devolver algún valor**, veamos el siguiente ejemplo:

//Regresa un valor

function foo(){

return 'Hello';

}

console.log(foo()); //output Hello

//No regresa un valor (realmente regresa undefined)

function bar(){

return;

}

console.log(bar()); //output undefined

Es importante observar que **si no regresamos algún valor**, por **default** la **función** nos va a **regresar** el valor *undefined*, **exceptuando** cuando se usa el **operador** *new*, para este caso nos **regresa** el *this* del **objeto instanciado**, veamos un ejemplo:

//Cuando no regresamos valor

function bar(){

return;

}

//Sin operador new

console.log(bar()); //output undefined

//Con el operador new

console.log(new bar()); //ouput bar{}

# Funciones flecha

Las **funciones** de **flecha** se recomienda para **devoluciones de llamadas** (*callbacks*) o **argumentos de función** (*function arguments*).

***Usar****: Cuando se usa una función de flecha se ejecuta en el contexto de this (comúnmente es lo que buscamos), su sintaxis es más limpia y legible.*

***No usar****: Cuando la función es compleja, aquí se recomienda tener su propia función con nombre.*

// Mal

[1, 2, 3].map(function (x) {

const y = x + 1;

return x \* y;

});

// Bien

[1, 2, 3].map(x => {

const y = x + 1;

return x \* y;

});

La expresión de **función de flecha** (*arrow function*), sus principales características son:

* Su sintaxis es más corta que una expresión de función.
* ***No vincula su propio this, arguments, super.***
* **SIEMPRE** son anónimas.
* Son **funciones no relacionadas** con métodos por lo que **NO** pueden ser **usadas** como **constructores.**

Antes de explicar más a fondo las **arrow functions**, veamos un ejemplo en **ES5** y cómo quedaría en **ES6**:

**Con ES5**

// Ejemplo 1

var odds = evens.map(function (v) { return v + 1; });

// Ejemplo 2

var pairs = evens.map(function (v) { return { even: v, odd: v + 1 }; });

// Ejemplo 3

var nums = evens.map(function (v, i) { return v + i; });

**Con ES6**

// Ejemplo 1

let odds = evens.map(v => v + 1);

// Ejemplo 2

pairs = evens.map(v => ({ even: v, odd: v + 1 }));

// Ejemplo 3

nums = evens.map((v, i) => v + i);

## Sintaxis básica

De acuerdo a la documentación de *MDN Web Docs de moz://a*, su **sintaxis básica** es la siguiente:

(param1, param2, …, paramN) => { sentencias }

(param1, param2, …, paramN) => expresion

// Equivalente a: () => { return expresion; }

// Los paréntesis son opcionales cuando sólo dispone de un argumento: singleParam => { statements }

(singleParam) => { sentencias } singleParam => { sentencias }

// Una función sin argumentos requiere paréntesis:

() => { sentencias }

## Primeros pasos

La inclusión de las **arrow** **functións** es por la *reducción* y *simplicidad* de código.

var names = ['Mauricio', 'Benjamin', 'David', 'Fabiola'];

// Con ES5

names.map(function (name) {

return name.length;

}); // [8, 8, 5, 7]

// Con ES6 pero sin desestructuración y con el return

names.map(name => {

return name.length;

}); // [8, 8, 5, 7]

// Con Es6 con desestructuración y sin return

names.map(({ length }) => length); // [8, 8, 5, 7]

## El this

Uno de los **GRANDES** problemas que se tiene con las **funciones** es el *this*, ya que cada función define su propio valor del *this* (como lo hemos visto en ejemplos anteriores), lo cual genera muchos problemas y errores al momento de programar, veamos un ejemplo:

**Con ES5**

function Persona() {

this.edad = 0;

setInterval(function crecer() {

this.edad++; //El this es el del método crecer

}, 1000);

}

var p = new Persona();

Una de las soluciones en **ES5** es el famoso *self* o *that*

function Persona() {

var self = this;

self.edad = 0;

setInterval(function crecer() {

self.edad++;

}, 1000);

}

var p = new Persona();

Otra de las soluciones era con el *bind*

function Persona() {

this.edad = 0;

setInterval(function crecer() {

this.edad++; //Es el this de Persona, que se ha indicado por medio del bind

}.bind(this), 1000);

}

var p = new Persona();

Pero ahora con **ES6** y las **funciones de flecha** es más sencillo y limpio.

Con **ES6**

function Persona() {

this.edad = 0;

setInterval(() => {

this.edad++; // this apunta al objeto Persona

}, 1000);

}

let p = new Persona();

*Las* ***expresiones*** *de* ***función de flecha*** *son las más* ***adecuadas*** *para* ***funciones*** *que* ***no son métodos****, ya que por las características que posee te puedes meter en problemas.*

Veamos el siguiente ejemplo:

let obj = {

i: 10,

b: () => console.log(this, this.i),

c: function () {

console.log(this, this.i);

}

};

obj.b(); // undefined, Window {…} (or the global object)

obj.c(); // 10, Object {…}

Cuando invocamos *obj.b()* el *this* **no** es el de *obj* es más bien (para el caso del ejemplo) *window*, por lo que *this.i* no existe así que retorna *undefined*. (preguntar si tiene que ver con el scope o el ámbito de la función).

Cuando invocamos *obj.c()* el *this* **si** es el de *obj*, por lo que *this.i* vale 10.

## Casos especiales

// Cuando solo es un parámetro, NO es necesario los paréntesis

let single = x => x \* 2;

// Cuando tiene más de un parámetro, SI es necesario los paréntesis

let double = (x, y) => x \* y;

// Si no necesitamos parámetros, tenemos que incluir el paréntesis vacío:

let hi = () => 'Hello World';

// El objeto arguments NO funciona

let test = () => arguments;

// Con el operador spread SI regresa los argumentos

let test = (…args) => args;

// Si tenemos más de una línea de código SI es necesario agregar las llaves { }

let foo = (x, y) => {

let result = 10;

return result;

};

// Si queremos regresar un objeto sin el return hay que ponerlo entre paréntesis, ya que de lo contrario no retorna lo esperado.

let func = () => ({ foo: 1 });

// Recordemos que no tienen constructor, por lo que el siguiente código genera un error en consola

let Foo = () => { };

let foo = new Foo(); // TypeError: Foo is not a constructor

// Así como tampoco tiene la propiedad prototype.

let Foo = () => { };