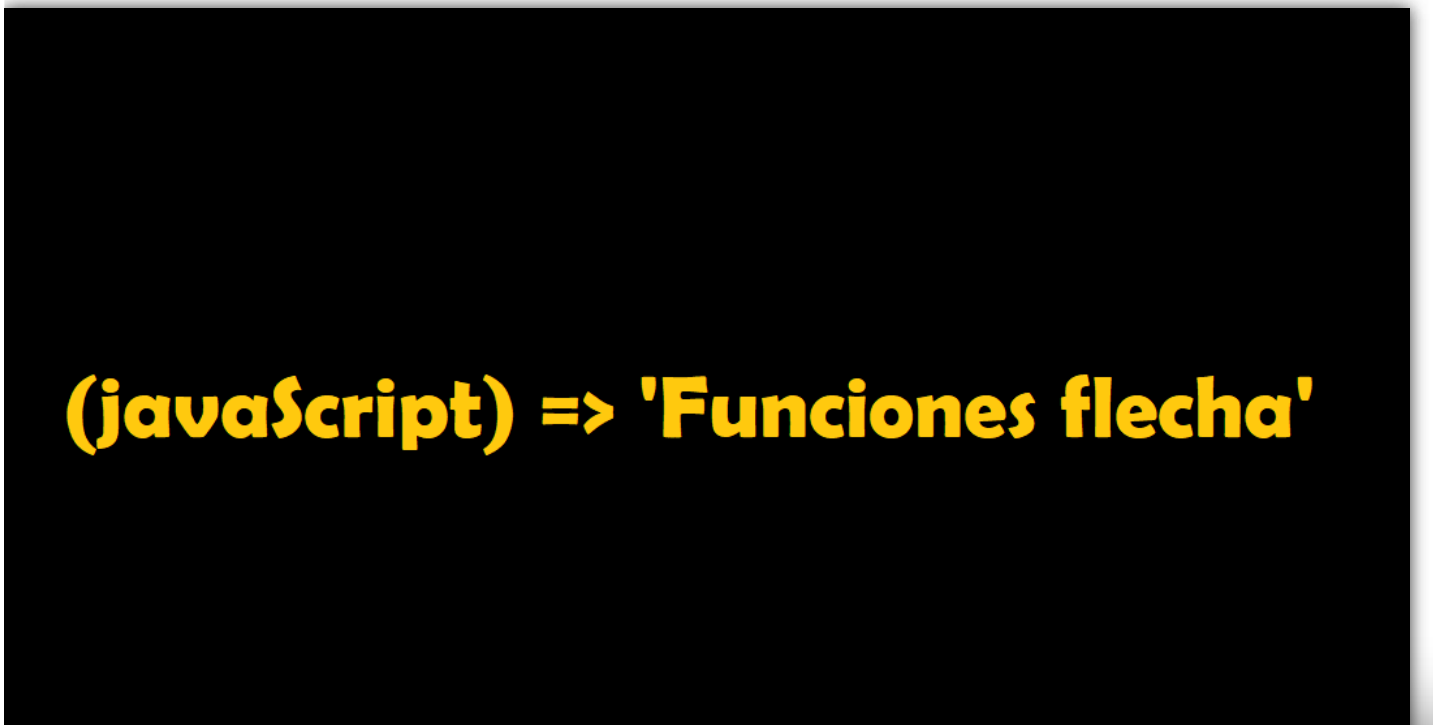
FUNCIONES FLECHA O ARROW FUNCTIONS EN JAVASCRIPT

JavaScript 12 mayo, 2019 0 Iván Salas



Recent Popular

COMO DESPLEGAR UNA APLICACIÓN SPRING BOOT + ANGULAR EN UN SERVIDOR CLOUD VPS
julio 4, 2019

SPRING BOOT PROFILES
julio 3, 2019

SPRING BOOT: INTERNACIONALIZACIÓN (I18N)
junio 11, 2019

FUNCIONES FLECHA O ARROW FUNCTIONS EN JAVASCRIPT
mayo 12, 2019

SPRING MVC VALIDATION GROUPS
abril 17, 2019

sintaxis también tienen algunas peculiaridades como que no vinculan su propio **this** o que no se pueden usar como constructores.

SINTAXIS DE LAS FUNCIONES FECHA

La sintaxis de las funciones flechas es un poco peculiar y para liarlo un poco más hay partes que son opcionales o necesarias dependiendo de los parámetros, el cuerpo y lo que retorne la función como resultado.

Vamos a empezar por definir una función sencilla de la forma clásica para sumar 2 números y vamos a ver cómo convertirla en una función flecha.

```
function suma(numero1, numero2) {
  return numero1 + numero2;
}
```

Sabemos que es una función precisamente porque lleva la palabra `function`, pues con las arrow funtions lo primero que tenemos que tener claro es que no llevan ni la palabra `function` ni ninguna otra, la forma por la que vamos a ver que es una función flecha es porque va a tener el operador flecha `=>` entre los parámetros y el cuerpo de la función, por lo tanto la primera sintaxis posible para nuestra función flecha es sustituir `function` por `=>`.

Las funciones flecha son funciones anónimas y por lo tanto si queremos poder llamar a nuestra función debemos de asignársela

SIGUENOS EN

```
var suma = (numero1, numero2) => {
  return numero1 + numero2;
}
```

Esta primera sintaxis no parece ser de mucha utilidad e incluso puede parecer hasta peor pero si como en este caso el cuerpo de la función solo tiene una sentencia podemos eliminar el `return` y las `{}` y ya tenemos una sintaxis mucho más compacta, ideal para usar en los callbacks por ejemplo.

```
var suma = (numero1, numero2) => numero1 + numero2
```

Por lo tanto podemos ver las funciones flecha como un flujo en el que dados unos parámetros (parte izquierda) nos devuelve un resultado (parte derecha).

```
// Asignamos la funcion a una variable para reutilizarla
var suma = (numero1, numero2) => numero1 + numero2
var a = suma(2, 3);
// 5
var b = suma(4, 5);
// 9

// O la usamos directamente
function calcular(operacion, numero1, numero2) {
  return operacion(numero1, numero2);
}

var c = calcular((numero1, numero2) => numero1 + numero2, 6, 7);
console.log(c);
// 13
```

SINTAXIS DE LAS FUNCIONES FECHA CON SOLO PARÁMETRO

Si la función tiene un solo parámetro entonces también se pueden eliminar los paréntesis de los parámetros.

```
var saludar = nombre => console.log(`Hola, ${nombre}`)
```

Con un parámetro podemos quitar los paréntesis pero si la función no recibe parámetros vuelven a ser igual de obligatorios que cuando tenemos varios parámetros.

```
var mostrarFecha = () => new Date().toLocaleDateString()
console.log(mostrarFecha());
// 1/5/2019
```

SINTAXIS DE LAS FUNCIONES FECHA QUE DEVUELVE UN OBJETO LITERAL

Si en la función flecha queremos devolver un objeto directamente vamos a tener que envolver el cuerpo de la función entre paréntesis aunque resulte raro, el motivo es que si no lo hiciésemos se interpretaría el objeto como el cuerpo de la función y no sería un código **javaScript** valido.

```
var usuario = (nombre, apellido) => ({nombre: nombre, apellido: apellido})
console.log(usuario("Iván", "Salas"));
// Object { nombre: "Iván", apellido: "Salas" }
```

PARÁMETROS REST EN LAS FUNCIONES FLECHA

Los parámetros rest se pueden usar en las funciones flecha sin ningún problema de la misma forma que en las funciones clásicas.

```
// Forma clasica
function miFuncion(parametro1, parametro2, ...otrosParametros) {
  return parametro1 + parametro2 + otrosParametros.reduce(
    function sumar(numero1, numero2) {
      return numero1 + numero2;
    });
}

console.log(miFuncion(1, 2, 3, 4));
// 10
```

```
console.log(miFuncionFlecha(1, 2, 3, 4));
// 10
```

Cuando la función se hace muy larga como en este último ejemplo normalmente incluimos saltos de línea para hacerlas más legibles, con las funciones flecha también podemos hacerlo siempre que no lo hagamos entre los parámetros y la flecha.

THIS Y LAS FUNCIONES FLECHA

Uno de los problemas clasicos a los que nos enfrentamos en **javaScript** es el uso de `this` dentro de funciones porque la forma en la que funciona parece ir en contra de lo que esperaríamos, vamos a verlo con un ejemplo.

Definimos una función que tiene una variable y mediante `setInterval` la vamos actualizando cada 1s y esperamos que si consultamos esa variable después de 10s por ejemplo el valor sea 10 pero nada más lejos de la realidad, siempre se va a quedar en 0.

```
function Cronometro() {
  this.tiempo = 0;

  setInterval(function() {
    this.tiempo++;
  }, 1000);
```

```
// Consultamos el valor cada 1s
setInterval(() => console.log(c.tiempo), 1000);
// 0
// 0
// ...
```

Esto es porque el objeto `this` no hace referencia al objeto `this` del objeto cronometro si no al `this` del objeto desde el que se invoca (Window) que no tiene definida la variable tiempo, si le metemos un log comprobamos que es eso lo que está pasando.

```
function Cronometro() {
  this.tiempo = 0;

  setInterval(function() {
    console.log(this.tiempo, this);
    this.tiempo++;
  }, 1000);
}

var c = new Cronometro();

// Consultamos el valor cada 1s
setInterval(() => console.log(c.tiempo), 1000);
// undefined, Window
// NaN, Window
// NaN, Window
// ...
```

Y si añadimos una variable global tiempo «si funciona» pero con el tiempo que no corresponde porque el `this` dentro del setInterval es distinto de el del objeto Cronometro, para solucionar este problema podemos usar el método `.bind()` o guardar su valor en una variable y utilizarla en su lugar (el típico `var self = this;` que se solía utilizar antes).

```
function Cronometro() {
  this.tiempo = 0;
```

```
var tiempo = 100;
var c = new Cronometro();

// Consultamos el valor cada 1s
setInterval(() => console.log('tiempo externo: ' + c.tiempo), 1000);
// tiempo interno: 100
// tiempo externo: 0
// tiempo interno: 101
// tiempo externo: 0
// tiempo interno: 102
// tiempo externo: 0
// ...
```

Con las funciones flecha no tenemos este problema porque no vinculan su `this` con el del contexto en el que se invocan ya que utilizan el `this` del contexto en el que están definidas que en este caso sería el del objeto y por lo tanto ahora sí que nos va a funcionar tal y como podríamos esperar.

```
function Cronometro() {
  this.tiempo = 0;

  setInterval(() => this.tiempo++, 1000);
}


var c = new Cronometro();

// Consultamos el valor cada 1s
setInterval(() => console.log(c.tiempo), 1000);
// 1
// 2
// 3
// ...
```

Es decir, el objeto al que hace referencia el `this` en una función flecha siempre va a ser el mismo independientemente del lugar

```
function Cronometro() {
  this.tiempo = 0;

  setInterval(() => {
    console.log('tiempo interno: ' + this.tiempo);
    this.tiempo++;
  }, 1000);
}

var tiempo = 100;
var c = new Cronometro();

// Consultamos el valor cada 1s
setInterval(() => console.log('tiempo externo: ' + c.tiempo), 1000);
// tiempo interno: 0
// tiempo externo: 1
// tiempo interno: 1
// tiempo externo: 2
// tiempo interno: 2
// tiempo externo: 3
// tiempo interno: 3
// ...
```

THIS Y LAS FUNCIONES FLECHA EN MÉTODOS

Todo parece muy bonito con `this` y las funciones flecha pero hay un caso en el que no las podemos utilizar, y es que si usamos una arrow function como método de un objeto tenemos el efecto contrario al que hemos visto antes.

```
var mueble = {
  nombre: 'silla',
  getNombreFlecha: () => console.log(this.nombre),
  getNombreNormal: function() {
    console.log(this.nombre);
  }
}
```

```
// silla

var nombreMueble = mueble.getNombreNormal();
console.log("nombreMueble: " + nombreMueble);
// nombreMueble: undefined
```

Aquí todo va bien con las funciones normales porque invocamos a la función desde el objeto y por eso el `this` es el del objeto, siempre que no se nos ocurra asignar la función a otra variable porque entonces estamos cambiando el contexto desde el que se ejecutará y por lo tanto el valor de `this` y ya tenemos un `undefined`.

Y con la función flecha también tenemos un `undefined` porque el contexto en el que está declarada la función es el global porque un objeto no define su propio ámbito.

POST RELACIONADOS

CREAR TOOLTIP USANDO JAVASCRIPT

15 marzo, 2013 3

COMO CREAR PESTAÑAS CON HTML

27 noviembre, 2012 64

CÓMO COMPROBAR SI UN ELEMENTO ESTÁ VISIBLE CON JQUERY

4 diciembre, 2017 0

COMO SABER SI UN NÚMERO ES DECIMAL O ENTERO EN JAVASCRIPT

2 abril, 2013 3