# Aprendiendo ECMAScript 6: Funciones Flecha (Arrow) - FrontEnd Labs


Este es el 3º de 20 capítulos de la serie Aprendiendo ECMAScript 6 de Frontend-labs, en donde cada semana veremos una nueva característica de ES6, el nuevo y actual estándar de Javascript. Si quieren estar al tanto de los próximos capítulos los invito a seguirme en Twitter como @lmsardon.

En el 1er capítulo, Aprendiendo ECMAScript 6: Constantes, aprendimos a declarar variables usando la sentencia *const*; y en el capítulo anterior, Aprendiendo ECMAScript 6: Scoping, hablamos del alcance que tienen las variables al ser declaradas usando las nuevas sentencias *let* y *const*.

En este capítulo hablaremos de una de las características más interesantes que trae ES6: Las *Funciones Flecha (Arrow)*. Gran parte de lo que veremos ha sido tomado del libro Understanding ECMAScript 6 de Nicholas C. Zakas, pero adaptado a mi forma de explicar las cosas y por supuesto al español...

## Funciones Flecha (Arrow)

Como su nombre lo indica, son funciones definidas usando una **flecha** =>, pero estas se comportan de una manera diferente a las *funciones tradicionales* en varios aspectos.

### Sintaxis

La sintaxis de las funciones flecha tiene diferentes formas, dependiendo de la tarea que se quiera realizar. Todas las variaciones comienzan con los

**argumentos de la función**, seguidos por la **flecha**, seguidos por el **contenido de la función**. Tanto los argumentos como el contenido pueden tener diferentes formas dependiendo de su uso. En el siguiente ejemplo, la *funcion flecha* toma un único argumento y simplemente lo devuelve.

```
var reflect = value => value;

// Que sería lo mismo que hacer esto:
var reflect = function (value) {
  return value;
};
```

> Cuando las *funciones flecha* reciben un solo argumento, ese único argumento puede ser utilizado directamente sin necesidad de agregar los paréntesis. La flecha viene después y la expresión final se evalúa y es retornada. No es necesario declarar el *return* de manera explícita, *la función flecha* retorna el primer argumento que encuentre.

En caso la función reciba más de un argumento, entonces sí debemos incluir los paréntesis, como en el siguiente ejemplo:

```
var sum = (n1, n2) => n1 + n2;

// Que sería lo mismo que hacer esto:
var sum = function (n1, n2) {
  return n1 + n2;
};
```

Un caso similar sería si la función no recibe ningún argumento, en ese caso van los paréntesis solos, como en el siguiente ejemplo:

```
var getName = () => "Luis Miguel";

// Que sería lo mismo que hacer esto:
var getName = function () {
  return "Luis Miguel";
};
```

Pero esto no quiere decir que no podemos usar una sintaxis más tradicional para el contenido de nuestras funciones flecha, es más, si vamos a realizar otras tareas, necesariamente vamos a tener que encapsular nuestro contenido entre llaves y declarar explícitamente el *return*, como en el siguiente ejemplo:

```
var sum = (n1, n2) => {
  return n1 + n2;
};

// Que sería lo mismo que hacer esto:
var sum = function (n1, n2) {
  return n1 + n2;
};
```

> Podemos tratar el contenido de las llaves casi como lo hacíamos de la manera tradicional, con la excepción de que el valor *arguments* no estará disponible.

Si queremos crear una función que no haga absolutamente nada, entonces ahí también debemos incluir las llaves, como en el siguiente ejemplo:

```
var doNothing = () => {};

// Que sería lo mismo que hacer esto:
var doNothing = function () {};
```

La cosa cambia un poco cuando lo que queremos retornar es un *objeto*, en ese caso tenemos que encapsularlo entre paréntesis. Por ejemplo:

```
var getTempItem = id => ({id: id, name: "Temp"});

// Que sería lo mismo que hacer esto:
var getTempItem = function (id) {
  return {
    id: id,
    name: "Temp"
  };
};
```

> Esto se hace por que al encapsular el *objeto* entre paréntesis declaramos que las llaves son el *objeto* y no que pertenecen al cuerpo de la función.

## Expresiones de Función Invocadas Inmediatamente (IIFE)

Las *IIFEs* por sus siglas en inglés, nos permiten crear funciones anónimas y llamarlas inmediatamente sin guardar una referencia de estas. Este patrón es muy útil cuando necesitamos crear un ámbito aislado del resto del programa. Por ejemplo:

```
let person = function (name) {
  return {
    getName: function() {
      return name;
    }
  }
}("Luis Miguel");

console.log(person.getName()); // <- Luis Miguel
```

> En este código, la *IIFE* es usada para crear un objeto con un método getName(). Este método usa el argumento name como valor de retorno, haciendo de name un valor *privado* dentro del objeto retornado.

Para lograr el mismo resultado usando *funciones flecha*, debemos encapsularlas entre paréntesis:

```
let person = ((name) => {
  return {
    getName: function() {
      return name;
    }
  }
})("Luis Miguel");

console.log(person.getName()); // <- Luis Miguel
```

> Notemos que los paréntesis sólo están alrededor de la *función flecha* y no llegan hasta ("Luis Miguel").

## No hay *this* binding

Algunos de los errores más comúnes en JavaScript ocurren precisamente por la asignación que tiene la palabra clave *this*. Dado que el valor de *this* puede cambiar dentro de una función dependiendo del ámbito donde ésta es ejecutada, es muy posible afectar por error un objeto cuando la intención es afectar a otro, veamos el siguiente ejemplo:

```
let PageHandler = {
  id: "123456",

  init: function () {
    document.addEventListener("click", function (event) {
      this.doSomething(event.type); // <- TypeError: this.doSomething is no
    }, false);
  },

  doSomething: function (type) {
    console.log("Handling " + type + "for " + this.id);
  }
};
```

En el código anterior, el objeto PageHandler está diseñado para manejar interacciones en la página. El método init() es llamado para establecer las interacciones, y este método a su vez asigna un controlador de eventos que ejecuta this.doSomething(). Sin embargo, este código no va a funcionar como esperamos.

La llamada a this.doSomething() falla debido a que this es una referencia del objetivo del evento (en este caso document), y no de PageHandler como esperamos. Si intentamos correr este código, tendremos un error cuando el controlador del evento se ejecute, porque this.doSomething() no existe en el objeto document.

Podríamos solucionar esto asignando el valor de this del PageHandler de manera explícita, usando el método bind() en la función, como en este ejemplo:

```
let PageHandler = {
```

```
  id: "123456",

  init: function () {
    document.addEventListener("click", function (event) {
      this.doSomething(event.type); // No hay error
    }.bind(this), false);
  },

  doSomething: function (type) {
    console.log("Handling " + type + "for " + this.id);
  }
};
```

Ahora nuestro código funciona, pero se ve un poco extraño. Al usar bind(this), lo que hacemos es crear una nueva función cuyo this está ligado al this actual, que en este caso es PageHandler. Para evitar crear una función extra, una mejor forma de solucionar este problema es usando *funciones flecha*.

Cuando hablamos de que no hay *this Binding*, significa que el valor de this dentro de la *función flecha* solo puede ser determinado buscando en la cadena de ámbitos. Por lo que, si la *función flecha* está dentro de una función contenedora tradicional, el valor de this va a ser igual a la función contenedora; de lo contrario, el valor de this será undefined. Veamos el ejemplo:

```
let PageHandler = {
  id: "123456",

  init: function () {
    document.addEventListener("click", event => this.doSomething(event.type
  },

  doSomething: function (type) {
    console.log("Handling " + type + "for " + this.id);
  }
};
```

## Conclusión

Las *funciones flecha (arrow)* son muy útiles a la hora de retornar un valor

de manera inmediata, escribiendo muy poco código y hasta en una sola línea, además nos permiten tener un mejor control sobre this con lo cual podemos hacer nuestro código mucho más mantenible.

Los espero en el próximo capítulo **Aprendiendo ECMAScript 6: Manejando Parámetros Extendidos**.

## Fuentes

- Understanding ECMAScript 6
- ECMAScript 6 — New Features: Overview & Comparison