Introduccion

Arrow function: conveniente para funciones sencillas y “rápidas”. Ya que estas funciones tienen un alcance léxico, de modo que el valor de la variable this dentro de una arrow function viene determinado por el alcance que la rodea.

This es una variable especial que es creada para cada execution context (cada function). Toma el valor (apunta a) el “propietario” de cada function en el cual se use la palabra clave this. This No es estatica. Depende de como se llama la function, y el valor solo es asignado cuando se llama a la función.

Arrow function

Las funciones flecha o arrow funtions son funciones anónimas con una sintaxis más compacta y que aparte de la diferencia en la sintaxis también tienen algunas peculiaridades como que no vinculan su propio this o que no se pueden usar como constructores.Fueron introducidas en la especificación ES2015

La declaración de una arrow function es muy muy sencilla:

```
const giveMeAMessage = () => 'hi arrow functions';
```

Cuando se componen de una única sentencia no necesitamos emplear la palabra clave return , y el valor que devuelven será aquel empleado como cuerpo de la función.

Por tanto, cuando invoquemos nuestra función giveMeAMessage , lo que obtendremos será el texto hi arrow functions :

```
const giveMeAMessage = () => 'hi arrow functions'; const aMessage = giveMeAMessage();
console.log(aMessage); // hi arrow functions
```

Si queremos devolver un objeto, deberemos emplear los paréntesis para rodear el cuerpo de la función:

```
INCORRECTO const giveMeADeveloper = () => { name: 'Gerardo', age: '33' };

CORRECTO  const giveMeADeveloper = () => ({ name: 'Gerardo', age: '33' });
```

Las arrow functions pueden componerse de más de una línea, lo cual ya nos obligará a emplear los paréntesis y a emplear return para, ahora sí, especificar el valor de retorno:

```
const giveMeAMessage = () => { const foo = 'hi arrow functions'; return foo; }
```

Argumentos en las arrow functions

Por otra parte, cuando queramos pasar parámetros a este tipo de funciones podremos hacerlo de dos formas. En el caso de que nuestra función tenga un único argumento no será necesario especificar los paréntesis. Por ejemplo:

```
const triple = x => x*3; const foo = triple(3); // 9
```

En el resto de casos sí que necesitaremos especificar los paréntesis:

```
const sum = (a, b) => a + b; const foo = sum(1, 2); // 3
```

Por lo demás, esta clase de funciones también nos permite especificar valores por defecto:

```
const pow = (a, b = 2) => Math.pow(a, b); const foo = pow(2); // 4 const bar = pow(3, 3); // 27
```

This y las funciones flecha

Supongamos que tenemos un objeto con el siguiente aspecto:

```
const foo = { log: function() { return function() { console.log(this); } } }
```

Cuando ejecutemos lo siguiente lo que obtendremos en la consola será el objeto Window (si estamos dentro de un navegador, claro):

```
foo.log()(); // Window
```

Ya que la invocación de la función devuelta por el método log del objeto foo es realizada por window , por lo que no se respeta el valor de this tal y como esperaríamos.

```
const foo = {
  log: function() {
    console.log(this); // 1. foo
    return function() {
      console.log(this); // 2. Window
    }
  }
}
const log = foo.log(); // 1. foo
log(); // 2. Window
```

Sin embargo, cuando trabajamos con arrow functions el comportamiento que obtenemos es esperado, ya que estas funciones tienen un alcance léxico, de modo que el valor de la variable this dentro de una arrow function viene determinado por el alcance que la rodea. Es decir:

```
const foo = { log: function() { console.log(this); // 1. foo return () => {
```

```
// this is lexically scoped to the surrounding scope,

 // ie, the foo object

console.log(this);

// 2. foo } } } const log = foo.log();

 // 1. foo log(); // 2. foo
```

Arrow functions y destructuring

Una característica con la que las arrow functions se llevan genial es con el destructuring assignment.

De este modo, cuando pasemos un objeto a una función de este tipo, podremos acceder directamente a sus propiedades de una forma muy visual:

```
const printAge = ({age}) => console.log(age); printAge({name: 'Gerardo', age: 33});
```

Así como declarar parámetros por defecto empleando esta característica:

```
const printAge = ({age} = {age: 20}) => console.log(age); printAge({name: 'Gerardo', age: 33}); //
33 printAge(); // 20
```

Limitaciones de las arrow functions

Sin embargo, las arrow functions tienen algunas limitaciones.

No podemos emplearlas para construir objetos.

Por tanto, si intentamos algo de este estilo:

```
const MyClass = () => {}; const object = new MyClass();
```

Obtendremos un TypeError .

No poseen prototype

Es decir, no podemos extender el prototipo de esta clase de objetos.

```
const MyClass = () => {}; console.log(MyClass.prototype); // undefined
```

No pueden ser usadas como funciones generadoras

Las arrow functions no admiten la palabra yield dentro de su cuerpo, por lo que si queremos crear una función generadora deberemos seguir recurriendo a la forma habitual: function* .

This

El entorno global de ejecución

Si estamos ejecutando Javascript en un navegador, este entorno estará representado por el propio objeto window , de modo que por defecto this tendrá dicho valor:

```
window === this; // true
```

Por otra parte, si estamos trabajando con NodeJS el entorno estará representado por la variable global por lo que:

```
global === this; // true
```

Es decir, this está mucho más relacionado con el entorno o contexto de ejecución que con el concepto de esta misma variable en la programación orientada a objetos, donde this representa al objeto en sí.

Valor de this

Method this = Objeto que llama al metodo

Simple function call : this = undefined

Arrow functions this = <this de función circundante (this léxico)>

Event listener this = <DOM element al que está adjunto el handler>

Funciones

A diferencia de la programación orientada a objetos donde sólo los objetos tienen disponible la variable this , en Javascript podemos acceder a esta variable dentro de las funciones.

Como norma general el valor en estos casos estará asociado al objeto que invocó la función.

Supongamos que definimos la siguiente función en nuestro navegador:

```
function foo() {
  console.log(this);
}
```

Y la invocamos desde la propia consola:

```
foo(); // Window
```

Lo que obtendremos al imprimir la variable this será el objeto window .

¿Por qué? Porque el objeto que invoca la función foo() es, por defecto, el objeto global window .

Lo mismo sucederá si por ejemplo escribimos la siguiente función:

```
function Person(name, email) {
  this.name = name;
  this.email = email;
}const person = Person('Gerardo', 'gerardo@latteandcode.com');
console.log(person);
```

En este caso, lo que obtendremos por pantalla será undefined .

¿Por qué? Por la misma razón de antes. Person es una función que está siendo invocada por el objeto global window , de modo que al hacer la asignación this.name lo que realmente estamos haciendo es window.name = name .

Para lograr el efecto deseado deberemos recurrir al operador new , de modo que, ahora sí, tengamos un objeto person con sus propiedades name y email setteadas:

```
function Person(name, email) {
  this.name = name;
  this.email = email;
}const person = new Person('Gerardo', 'gerardo@latteandcode.com');
console.log(person);
```

New

En Javascript podemos añadir el operador new delante de la invocación de una función para convertir dicha invocación en un constructor. Fue lo que hicimos en el ejemplo anterior cuando invocamos la función Person :

```
const person = new Person('Gerardo', 'gerardo@latteandcode.com');
console.log(person);
```

Es decir, el operador new permite la creación de un nuevo objeto que pasa a ser el this de esa invocación y que es devuelvo implícitamente por la función invocada (salvo que explícitamente devolvamos otra cosa dentro de la función empleando return).

Métodos

¿Qué sucede cuando invocamos una función que pertenece a un objeto? Es decir, ¿qué valor adopta this cuando invocamos el método de un objeto?

En general, es lo que esperamos:

```
const person = {
  name: 'Gerardo',
  surname: 'Fernández',
```

```
  fullname: function() {
    console.log(`${this.name} ${this.surname}`);
  }
};person.fullname(); // Gerardo Fernández
```

Es decir, cuando invocamos la función fullname() la variable this es el objeto person y por eso podemos acceder a sus propiedades name y surname . Esto se debe a que es el objeto person quien está invocando el método fullname .

Pero si por ejemplo lo cambiamos a lo siguiente:

```
const person = {
  name: 'Gerardo',
  surname: 'Fernández',
  fullname: function() {
    return function() {
      console.log(`${this.name} ${this.surname}`);
    };
  }
};person.fullname()(); // undefined
```

Lo que obtenemos es undefined en la consola. ¿Por qué? Veámoslo de otro modo:

```
name = 'Chrome';
surname = 'Window';const person = {
  name: 'Gerardo',
  surname: 'Fernández',
  fullname: function() {
    return function() {
      console.log(`${this.name} ${this.surname}`);
    };
  }
};const printFullname = person.fullname();
printFullname(); // Chrome Window
```

Es decir, el objeto que está invocando la función devuelta por el método fullname del objeto person es el objeto global como sucedía al principio del artículo y, por tanto, this tiene el valor del objeto global (window si estamos trabajando con la consola del navegador).

Arrow functions

Con la llegada de ES6 se introdujo una nueva forma de declarar funciones que afecta directamente al valor tomado por this : las “arrow functions”.

Este tipo de funciones toman el valor de this del contexto de ejecución que las rodea sin importar la forma en que sea invocada la función y previniendo que apply , call o bind lo modifiquen.

En Javascript cada vez que se ejecuta una función se crea un contexto de ejecución. Sin embargo, las “arrow functions” no definen por sí mismas un contexto de ejecución por lo que lo toman del inmediatamente superior.

Veámoslo con un ejemplo:

```
const book = {
  currentPage: 1,
  readPage: function() {
    setInterval(function() {
      this.currentPage += 1;
      console.log(this.currentPage);
    }, 1000);
  }
}book.readPage();
```

Se imprime una sucesión de Nan ya que el “callback” de la función setInterval es ejecutado en el contexto global, es decir, allí donde el valor de this es window, es como si internamente Javascript estuviese haciendo lo siguiente:

```
var callback = function() {
  this.currentPage += 1;
  console.log(this.currentPage);
}callback(); // cada 1 segundo
```

Sin embargo, las “arrow functions” nos permiten esquivar este problema:

```
const book = {
  currentPage: 1,
  readPage: function() {
    setInterval(() =>  {
      this.currentPage += 1;
      console.log(this.currentPage);
    }, 1000);
  }
}book.readPage();
```

Puesto que las “arrow functions” toman su contexto del inmediatamente superior, el valor de this será el que tenga la función readPage . Puesto que estamos invocando el método readPage del siguiente modo: book.readPage , el valor de this dentro del método será el del objeto book y también así el de this dentro de la “arrow function”.

Si hiciéramos lo siguiente:

```
const book = {
  currentPage: 1,
  readPage: () => {
    setInterval(() =>  {
      this.currentPage += 1;
      console.log(this.currentPage);
    }, 1000);
  }
}const readPage = book.readPage;
readPage();
```

Volveríamos a recibir solo Nan por pantalla, ya que el contexto de readPage sería el global.