Funciones, objetos y promesas

Contenido

1. Funciones.

1.1. Funciones tradicionales

Empecemos con un ejemplo simple. Supongamos que esta función tradicional devuelve la suma de dos parámetros que le pasemos:

```
function add (num1, num2) {
	return num1 + num2;
}
```

Cuando usemos esta función, simplemente la llamamos en el sitio deseado pasandole los parámetros apropiados. Por ejemplo:

```
console.log (add (3, 2)); // Se mostrará 5
```

1.2. Funciones anónimas

También podemos expresar la misma función como anónima. Esas funciones se declaran "sobre la marcha", y se suelen asignar a una variable para que después puedan ser llamadas:

```
let addAnonymous = function (num1, num2) {
	return num1 + num2;
};
console.log (addAnonymous (3, 2));
```

1.3. Las funciones “flecha”

Las funciones "flecha” o “arrow functions” definen una manera de definir funciones que usan la función lambda para especificar los parámetros (entre paréntesis) y el código de la función entre llaves separado por una flecha. La palabra function se omite al definirlas. La misma función anterior, expresada como una función flecha, sería así:

```
let add = (num1, num2) => {
	return num1 + num2;
};
```

Al igual que con las funciones anónimas, puede asignar su valor a una variable para su

uso posterior o definirlas sobre la marcha en un fragmento de código particular. De hecho, el código anterior se puede simplificar aún más: en el caso de que la función simplemente devuelva un valor, se puede prescindir de las llaves y de la palabra de retorno, así:

```
let add = (num1, num2) => num1 + num2;
```

Si la función tiene un solo parámetro, se puede prescindir de los paréntesis. Por ejemplo, esta función devuelve el doble del número que recibe como parámetro:

```
let double = num => 2 * num;
console.log (double (3)); // Will display 6
```

Uso directo de las funciones flecha

Como mencionamos anteriormente, las funciones de flecha, al igual que las funciones anónimas, tienen la ventaja de poder usarse directamente en el lugar donde se necesitan. Por ejemplo, dada la siguiente matriz de objetos con datos personales:

```
let data = [
{name: "Nacho", telephone: "966112233", age: 40},
{name: "Ana", telephone: "911223344", age: 35},
{name: "Mario", phone: "611998877", age: 15},
{name: "Laura", telephone: "633663366", age: 17}
];
```

Si queremos filtrar a las personas mayores de edad, podemos hacerlo con una función anónima combinada con la función de filtro:

```
let olderOfAge = data.filter (function (person) {
       return person.age >= 18;
})
console.log (olderOfAge);
```

Y también podemos usar la función flecha en su lugar:

```
let olderOfAge = data.filter (person => person.age >= 18);

console.log (olderOfAge);
```

Tenga en cuenta que, en estos casos, no asignamos la función a una variable para usarla más tarde, sino que se usan en el mismo punto donde se definen. Tenga en cuenta también que el código es más compacto usando una función de flecha.

Funciones flecha y funciones tradicionales

La diferencia entre las funciones de flecha y la nomenclatura tradicional o funciones anónimas es que con las funciones de flecha no podemos acceder a este elemento, ni al elemento arguments, que están disponibles con funciones anónimas o tradicionales. Así que, en caso de que necesitemos hacerlo, tendremos que optar por una función normal o anónima, en este caso.

2. Callbacks.

Uno de los dos pilares en los que se basa la programación asíncrona en Javascript lo constituyen los callbacks. Un callback es una función A que se pasa como parámetro a otra B, y que será llamada en algún momento durante la ejecución de B (normalmente cuando B termina su tarea). Este concepto es fundamental para darle a Node.js (y a Javascript en general) un comportamiento asíncrono: se llama a una función, y se le indica lo que tiene que hacer cuando finaliza, y mientras tanto el programa puede dedicarse a otras cosas.

Un ejemplo lo tenemos con la función de Javascript setTimeout. Podemos indicarle a esta función una función a llamar, y un tiempo (en milisegundos) a esperar antes de llamarla. Una vez ejecutada la línea de la llamada a setTimeout, el programa sigue su curso y cuando expira el tiempo, se llama a la función de devolución de llamada indicada. Intentemos escribir este ejemplo en un archivo llamado "callback.js":

```
setTimeout (function () {
            console.log ("Callback done");
            }, 2000);
console.log ("Hello");
```

Si ejecutamos el ejemplo, veremos que el primer mensaje que aparece es “Hello”, y después de dos segundos, aparece el mensaje “Callback done”. Es decir, hemos llamado a setTimeout y el programa ha seguido su curso después, ha escrito "Hello" en pantalla y, una vez transcurrido el tiempo estipulado, se ha llamado al callback para que haga su trabajo.

Utilizaremos los callbacks de forma extensiva durante este curso. Sobre todo, para procesar el resultado de algunas promesas que usaremos (ahora veremos qué son las promesas), o el tratamiento de algunas solicitudes de servicio.

3. Promesas.

Las promesas son otro mecanismo importante para hacer que Javascript sea asíncrono. Se utilizan para definir la finalización (con éxito o no) de una operación asíncrona. En nuestro código, podemos definir promesas para realizar operaciones asincrónicas o (más comúnmente) usar promesas definidas por otros en el uso de sus bibliotecas.

A lo largo de este curso utilizaremos promesas para, por ejemplo, enviar operaciones a una base de datos y recoger sus resultados cuando finalicen, sin bloquear el programa principal. Pero para entender mejor lo que vamos a hacer, llegado el momento, conviene tener claro la estructura de una promesa y las posibles respuestas que ofrece.

3.1. Crear una promesa.

En el caso de que queramos o necesitemos crear una promesa, se creará un objeto de tipo Promise. Una función con dos parámetros se pasa a este objeto como parámetro:

- La función callback para llamar si todo salió correctamente.
- La función callback para llamar si ha habido un error.

Estos dos parámetros suelen denominarse, respectivamente, resolve y reject. Por lo tanto, un esqueleto de promesa básico, usando funciones de flecha para definir la función a ejecutar, se vería así:

```
let variableName = new Promise ((resolve, reject) => {
// Código a ejecutar
// Si todo va bien, llamamos a "resolve"
// Si algo falla, llamamos a "reject"
});
```

Internamente, la función hará su trabajo y llamará a sus dos parámetros, en cualquier caso. En el caso de resolve, se suele pasar como parámetro el resultado de la operación, y en el caso de reject, se le suele pasar el error producido.

Veámoslo con un ejemplo. La siguiente promesa encuentra a los adultos de la lista de personas pasadas como parámetro a través de la variable *listing*. Si se encuentran resultados, se devuelven con la función de resolve. De lo contrario, se genera un error y se envía con reject. Copie el ejemplo en un archivo llamado "promise_test.js":

```
let OlderOfAgePromise = listing => {
	return new Promise ((resolve, reject) => {
		let result = listing.filter (person => person.age> = 18);
		if (result.length> 0)
			resolve (result);
		else
			reject ("No results");
});
};
```

3.2. Consumo de promesa

En el caso de querer utilizar una promesa previamente definida (o creada por otros en una librería), simplemente llamaremos a la función u objeto que dispara la promesa, y recopilamos el resultado. En este caso:

- Para recolectar un resultado satisfactorio (**resolve**) usamos la cláusula **then**.
- Para recoger un resultado erróneo (**reject**) utilizamos la cláusula **catch**.

Por lo tanto, la promesa anterior se puede usar de esta manera (nuevamente, usamos funciones de flecha para procesar la cláusula then con su resultado, o el catch con su error):

```
	let data = [
	{name: "Nacho", telephone: "966112233", age: 40},
	{name: "Ana", telephone: "911223344", age: 35},
	{name: "Mario", phone: "611998877", age: 15},
	{name: "Laura", telephone: "633663366", age: 17}
```

```
];

OlderOfAgePromise(data)
    .then(result => {
        // si estamos aquí la promesa ha sido correctamente procesada
        console.log("Matched coincidences:");
        console.log(result);
    })
    .catch(error => {
        // si estamos aquí ha habido un error
        console.log("Error:", error);
    });
```

Copie este código debajo del código anterior en el archivo "test_promise.js" creado anteriormente, para verificar la operación y lo que muestra la promesa. Tenga en cuenta que, al definir la promesa, también se define la estructura que tendrá el resultado o error. En este caso, el resultado es un vector de personas que coinciden con los criterios de búsqueda y el error es una cadena de texto. Pero pueden ser el tipo de datos que queremos.

4. Creación de clases y objetos.

4.1. Sintaxis básica

Vamos a crear una clase:

```
class Person{
constructor(firstname,lastname,birthday) {
this.firstname=firstname;
this.lastname=lastname;
this.birthday=newDate(birthday);
}
getAge() {
const today = new Date();
let age = today.getFullYear() - this.birthday.getFullYear();
return age;
}
}
const p = new Person("Paul", "Almunia", "1966-08-07");
console.log(p.getAge()); // calcula la edad real
```

A diferencia de las funciones, las clases no se pueden usar antes de definirlas. Las funciones en Javascript pasan por un proceso llamado hoisting por el cual las variables declaradas con var y las funciones son trasladadas al inicio del ámbito donde se encuentran y como consecuencia pueden ser llamadas antes de su declaración. En el caso de las clases esto no funciona así y si tratamos de usar una clase antes de definirla obtendremos un error.

4.2. Expresiones de clase

Similar a las funciones, puede asignar la definición de una clase a una variable. Como en el caso de las funciones, se oculta el nombre de la clase y solo se puede utilizar el nombre de la variable que se ha utilizado en la asignación:

```
const People=class Person{
constructor(firstname,lastname,birthday) {
this.firstname=firstname;
this.lastname=lastname;
this.birthday=new Date(birthday);
}
getClassName() {
return Person.name;
}
};
var p=new People("Ana", "López", "1990-05-12");
console.log(p.getClassName());
var p2=new Person(); // esto dará error ya que no está definida en el ámbito externo
```

Dentro de la definición de la clase no es posible escribir código directamente, ya que como tal no se comporta como una función, lo que nos vamos a encontrar es la definición de métodos. Uno de estos métodos se comporta de manera especial: **constructor**. Esta función será llamada cuando un objeto de esta clase sea instanciado por medio de new.

4.3. Métodos

El resto de métodos se definen en el cuerpo de la clase sin usar función, basta con poner el nombre del método y entre paréntesis los parámetros que recibirá (o ninguno) y escribir el cuerpo de la función.

```
class Person{
constructor(firstname,lastname,birthday) {
this.firstname=firstname;
this.lastname=lastname;
this.birthday=new Date(birthday);
}
getAge() {
const today = new Date();
let age = today.getFullYear() - this.birthday.getFullYear();
return age;
}
toString() {
return this.firstname+''+this.lastname+'('+this.getAge() +')';
}
}
var p=new Person('Paul','Almunia','07 -08-1966 ');
// Produce una llamada implícita to toString () ya que intenta concatenar una cadena
con un //objeto “p”. Si hay uno lo usa, sino usa el heredado de Object.
console.log('Person:'+p);
```

4.4. Propiedades

En las primeras versiones de ES6 no era posible definir propiedades directamente en el cuerpo de una clase, por lo que se utilizaba el constructor (o cualquier otro método) para inicializarlas.

Sin embargo, desde ES2022 sí es posible declarar **propiedades** dentro de la clase. El constructor sigue siendo útil cuando queremos recibir valores como parámetros o ejecutar lógica de inicialización que modifique esas propiedades. También podemos emplear los métodos get y set para controlar el acceso y la modificación de las propiedades.

4.5. Herencia

Aunque en ES5 podemos usar la cadena de prototipos para realizar una herencia, con la sintaxis de clases de ES6 esto es mucho más simple y claro. Para esto debemos usar extends en la declaración de la clase:

```
class Person{
constructor(firstname,lastname,birthday) {
this.firstname=firstname;
this.lastname=lastname;
this.birthday=new Date(birthday);
}
get age() {
const today = new Date();
let age = today.getFullYear() - this.birthday.getFullYear();
return age;
}
}
class Employee extends Person{
constructor(firstname,lastname,birthday,position) {
super(firstname,lastname,birthday);
this.position=position;
}
}
// Creamos un objeto
var p=new Employee('Paul','Almunia','07 -08-1966 ','Architect');
// Comprobamos las propiedades
console.log(p.firstname,p.lastname,p.position);
console.log(p.age);
```

Para que un constructor o método pueda llamar a miembros de la clase principal, debe usar **super**, que es una referencia a la clase superior. En el caso del constructor se llamará simplemente con los parámetros, en el caso de querer llamar a métodos o propiedades concretas usaremos super.membername.