Índice.

1. Funciones.

Las funciones tradicionales

Un ejemplo sencillo, función que devuelve la suma de los dos parámetros que se le pasan:

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

Para utilizar esta función:

```
console.log(sumar(3, 2)); // Mostrará 5
```

Las funciones anónimas

La función anterior en función anónima:

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

Las "arrow functions"

Las "funciones flecha" o arrow functions emplean expresiones lambda:

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

En el caso de que la función simplemente devuelva un valor, se puede prescindir de las llaves y de la palabra return, quedando así:

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

En el caso de que la función tenga un único parámetro, se pueden prescindir de los paréntesis. Por ejemplo, esta función devuelve el doble del número que recibe como parámetro:

```
let doble = num => 2 * num;
console.log(doble(3)); // Mostrará 6
```

Uso directo de arrow functions

Las arrow functions y las funciones anónimas, tienen la ventaja de poder utilizarse directamente en el lugar donde se precisan. Por ejemplo, dado el siguiente array de objetos con datos personales:

```
let datos = [
{nombre: "Nacho", telefono: "966112233", edad: 40},
{nombre: "Ana", telefono: "911223344", edad: 35},
{nombre: "Mario", telefono: "611998877", edad: 15},
{nombre: "Laura", telefono: "633663366", edad: 17}
];
```

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

```
let mayoresDeEdad = datos.filter(function(persona) {
      return persona.edad >= 18;
})
console.log(mayoresDeEdad);
```

Y también podemos emplear una arrow function en su lugar:

```
let mayoresDeEdad = datos.filter(persona => persona.edad >= 18);
console.log(mayoresDeEdad);
```

En ambos casos no asignamos la función a una variable para usarla más tarde, sino que se emplean en el mismo punto donde se definen.

Arrow functions y funciones tradicionales

La diferencia entre las arrow functions y la nomenclatura tradicional o las funciones anónimas es que con las arrow functions no podemos acceder al elemento this, o al elemento arguments, que sí están disponibles con las funciones anónimas o tradicionales. Así que, en caso de necesitar hacerlo, deberemos optar por una función normal o anónima, en este caso.

2. Callbacks.

Este concepto es fundamental para dotar a Node.js (y a Javascript en general) de un comportamiento asíncrono: se llama a una función, y se le deja indicado lo que tiene que hacer cuando termine, y mientras tanto el programa continúa ejecutándose.

Un ejemplo lo tenemos con la función setTimeout de Javascript. A esta función le podemos indicar una función a la que llamar, y un tiempo (en milisegundos) que esperar antes de llamarla. Ejecutada la línea de la llamada a setTimeout, el programa sigue su curso y cuando el tiempo expira, se llama a la función callback indicada. Probemos a escribir este ejemplo en un archivo llamado “callback.js”:

```
setTimeout(function() {
            console.log("Finalizado callback");
            }, 2000);
console.log("Hola");
```

Si ejecutamos el ejemplo, veremos que el primer mensaje que aparece es el de “Hola”, y pasados dos segundos, aparece el mensaje de “Finalizado callback”. Es decir, hemos llamado a setTimeout y el programa ha seguido su curso después, ha escrito “Hola” por pantalla y, una vez ha pasado el tiempo estipulado, se ha llamado al callback para hacer su trabajo.

Utilizaremos callbacks ampliamente durante este curso. De forma especial para procesar el resultado de algunas promesas que emplearemos (ahora veremos qué son las promesas), o el tratamiento de algunas peticiones de servicios.

3. Promesas.

Las promesas son otro mecanismo importante para dotar de asincronía a Javascript. Se emplean para definir la finalización (exitosa o no) de una operación asíncrona. En nuestro código, podemos definir promesas para realizar operaciones asíncronas, o bien (más habitual) utilizar las promesas definidas por otros en el uso de sus librerías.

A lo largo de este curso utilizaremos promesas para, por ejemplo, enviar operaciones a una base de datos y recoger el resultado de las mismas cuando finalicen, sin bloquear el programa principal. Pero para entender mejor qué es lo que haremos, llegado el momento, conviene tener clara la estructura de una promesa y las posibles respuestas que ofrece.

Crear una promesa.

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

- La función callback a la que llamar si todo ha ido correctamente
- La función callback a la que llamar si ha habido algún error

Estos dos parámetros se suelen llamar, respectivamente, resolve y reject. Por lo tanto, un esqueleto básico de

promesa, empleando arrow functions para definir la función a ejecutar, sería así:

```
let nombreVariable = 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 uno u otro caso. En el caso de resolve, se le 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 busca los mayores de edad de la lista de personas vista en un ejemplo anterior. Si se encuentran resultados, se devuelven con la función resolve. De lo contrario, se genera un error que se envía con reject. Copia el ejemplo en un archivo llamado "prueba_promesa.js":

```
let datos = [
{nombre: "Nacho", telefono: "966112233", edad: 40},
{nombre: "Ana", telefono: "911223344", edad: 35},
{nombre: "Mario", telefono: "611998877", edad: 15},
{nombre: "Laura", telefono: "633663366", edad: 17}
];
let promesaMayoresDeEdad = new Promise((resolve,reject) => {
      let resultado = datos.filter(persona =>persona.edad >= 18);
      if (resultado.length > 0)
```

```
                    resolve(resultado);
      else
                    reject("No hay resultados");
});
```

La llamada a la función seria

```
promesaMayoresDeEdad.then(result => {
                // If we are here the promise has been correctly processed

                console.log("Matched coincidences:");
                console.log(result);
                }).catch(error => {
                // if we are here there was an error

                console.log("Error:", error);
                });
```

La función que define la promesa también se podría definir de esta otra forma:

```
let promesaMayoresDeEdad = listado => {
      return new Promise((resolve, reject) => {
            let resultado = listado.filter(persona => persona.edad >= 18);
            if (resultado.length > 0)
                  resolve(resultado);
            else
                  reject("No hay resultados");
});
};
```

La llamada seria

```
promesaMayoresDeEdad(data).then(result => {

// If we are here the promise has been correctly processed
console.log("Matched coincidences:");
console.log(result);
}).catch(error => {
// if we are here there was an error

console.log("Error:", error);
});
```

Así no hacemos uso de variables globales, y el array queda pasado como parámetro a la propia función, que devuelve el objeto Promise una vez concluya. Deja definida la promesa de esta segunda forma en el archivo fuente de prueba.

Consumo de promesas

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

- Para recoger un resultado satisfactorio (resolve) empleamos la cláusula then.
- Para recoger un resultado erróneo (reject) empleamos la cláusula catch.

Así, la promesa anterior se puede emplear de esta forma (nuevamente, empleamos arrow functions para procesar la

cláusula then con su resultado, o el catch con su error):

```
promesaMayoresDeEdad(datos).then(resultado => {
	// Si entramos aquí, la promesa se ha procesado bien
	// En "resultado" podemos acceder al resultado obtenido
	console.log("Coincidencias encontradas:");
	console.log(resultado);
}).catch(error => {
	// Si entramos aquí, ha habido un error al procesar la promesa
	// En "error" lo podemos consultar
	console.log("Error:", error);
});
```

Copia este código bajo el código anterior en el archivo "prueba_promesa.js" creado anteriormente, para comprobar el funcionamiento y lo que muestra la promesa. Notar que, al definir la promesa, se define también la estructura que tendrá el resultado o el error. En este caso, el resultado es un vector de personas coincidentes con los criterios de búsqueda, y el error es una cadena de texto. Pero pueden ser el tipo de dato que queramos.

4. Creación de clases y objetos en ES6.

Sintáxis básica

Para crear una clase en ES6 vamos a utilizar la palabra reservada class, en el modo «use strict», de esta forma:

```
"use stric";
class Person {
    constructor(firstname, lastname, birthday) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.birthday = new Date(birthday);
    }
    getAge() {
        var ageDifMs = Date.now() - this.birthday.getTime();
        var ageDate = new Date(ageDifMs);
        return Math.abs(ageDate.getUTCFullYear() - 1970);
    }
}
```

Definición y uso

A diferencia de las funciones, las clases no pueden ser utilizadas antes de ser definidas. Las funciones en Javascript sufren un proceso denominado hoisting por el que las variables declaradas con var y las funciones son desplazadas al principio del alcance donde se encuentra y como consecuencia pueden ser llamadas antes de su declaración. En el caso de las clases esto no funciona así y si intentamos utilizar una clase antes de definirla obtendremos un error.

Expresiones de clases

De forma similar a las funciones, es posible asignar la definición de una clase a una variable. A diferencia de las funciones, no se aceptan clases anónimas, pero al igual que en caso de las funciones, el nombre de la clase queda oculto y sólo es posible utilizar el nombre de la variable que se ha utilizado en la asignación:

```
"use strict";
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();
console.log(p.getClassName());
```

```
var p2 = new Person();
```

Constructor

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 vamos a encontrar es la definición de métodos. Uno de estos métodos se comporta de una manera especial: constructor. Esta función será llamada cuando se instancie un objeto de esta clase por medio de new.

Métodos

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

```
"use strict";
class Person {
    constructor(firstname, lastname, birthday) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.birthday = new Date(birthday);
    }
    getAge() {
        var ageDifMs = Date.now() - this.birthday.getTime();
        var ageDate = new Date(ageDifMs);
        return Math.abs(ageDate.getUTCFullYear() - 1970);
    }
    toString() {
        return this.firstname + ' ' + this.lastname + ' (' + this.getAge() + ')';
    }
}
var p = new Person('Pablo', 'Almunia', '07-08-1966');
// Produce una llamada implícita a toString()
console.log('Person: ' + p);
```

Métodos estáticos

Si lo necesitamos podemos escribir métodos estáticos, es decir, que pueden ser invocados desde la clase sin necesidad de que se cree una instancia de la misma.

Para ello tenemos que poner la palabra static antes del nombre del método.

Propiedades

No es posible definir propiedades en el cuerpo de una clase. Para poderlas definir podemos utilizar el constructor (o desde cualquier otro método), tal y como hemos visto en el primer ejemplo, donde se definen las propiedades firstname, lastname y birthday por medio de this. Una alternativa a esta forma de crear las propiedades es utilizar métodos get y/o set para definir dos funciones que nos sirvan para gestionar su acceso.

Herencia

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

```
"use strict";
class Person {
    constructor(firstname, lastname, birthday) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.birthday = new Date(birthday);
    }
    get age() {
        var ageDifMs = Date.now() - this.birthday.getTime();
        var ageDate = new Date(ageDifMs);
        return Math.abs(ageDate.getUTCFullYear() - 1970);
    }
}
class Employee extends Person {
    constructor(firstname, lastname, birthday, position) {
        super(firstname, lastname, birthday);
        this.position = position;
    }
}
// Creamos un objeto
var p = new Employee('Pablo', 'Almunia', '07-08-1966', 'Architect');
// Consultamos sus propiedades
console.log(p.firstname, p.lastname, p.position);
console.log(p.age);
```

super

Para que un constructor o un método puedan llamar a miembros de la clase padre debe utilizar super que es una referencia a la clase superior. En el caso del constructor se le llamará simplemente con los parámetros, en el caso de querer llamara métodos o propiedades concretas usaremos super.nombredelmiembro.