Chuletilla Javascript

Javascript

Scope

```
function foo(){
//scope de foo.
  { //begin another scope
    ...
  } //end another scope
}; // end scope de foo
```

var vs. let

```
if (true) {
  var test = true;
}
alert(test); // the variable lives after the if block

if (true) {
  let test = true; // use "let"
}
alert(test); // Error: test is not defined
```

Functions

Function declaration:

```
function add(x, y) {
  return x + y;
}
```

Function expression asignada a una variable:

```
let add = function myFunc(x, y) { return x + y };
```

Función anónima:

```
let add = function (x, y) {
  return x + y };
```

Callback:

```
let add = function (x, f2) { // el parametro f2 es la
    ↪ declaracion de una funcion
  return (x+f2(1,2)); // invocamos la callback f2
};
// Ahora definimos la callback f2 como anonima y la pasamos
    ↪ como parametro
add(7, function(a,b){return a+b});
```

Self-invoquing function

Una función anónima que llamamos justo despues de declararla. Esto sirve para definir un scope privado ya que todas las variables que definamos dentro de esta funcion seran inaccesibles desde fuera.

```
(function () { // declara funcion anonima
  let a = 0; // inaccesible desde fuera
})(); // termina de declarar y llama la funcion
```

Clausuras

Las clausuras son funciones (inner function) que pueden acceder al entorno de una outer function. Actuan como una fabrica de funciones. Util para conservar variables entre distintas llamadas a la funcion.

```
function createInc(startValue) { // outer function
  return function (step) { // inner function
    startValue += step;
    return startValue;
  };
}
```

Arrow function

Las arrow function no son objectos al contrario que el resto de funciones por lo que el this no cambia. Si hacemos referencia a this en una arrow function, hacemos referencia al this de la outer function.

```
function outerFunction(a, b){
  let foo = (a,b) => a+b;
  let bar = () => console.log(this); //this hace referencia
      ↪ a outerFunction
  foo(a,b); bar();
}

() => x es lo mismo que () => {return x;}
```

Objetos literales, instancias de la clase Object

```
let cuck = {
  name: 'cuck',
  age: 10,
};
cuck.name == cuck[name] // true
cuck.kill = function() {} // si no existe se crea
Object.keys(cuck) // imprime propiedades
Object.values(cuck) // imprime valores
```

Herencia, prototype

Mediante la propiedad __proto__ un objeto puede heredar los atributos de otro

```
let cuck2 = {
  __proto__: cuck, // cuck2 hereda de cuck
  age: 20,
};
console.log(cuck.name); // 'cuck' heredado de cuck
```

Constructores

Un constructor es una funcion que produce objetos y se invoca con el operador `new`.

```
function Person(name, age) {
  this.name = name; // propiedad publica
  this.age = age;
  let address = 'c/␣foo'; // propiedad privada
}
let alberto = new Person('Albertillo', 20);
Object.getPrototypeOf(alberto); // Person {}
alberto.address; // undefined
```

Si definimos una propiedad fuera del constructor de la clase, estamos definiendo una propiedad estática

```
Person._DEFAULT_DNI = 12345678P; //(1)
```

Herencia en clases

A traves de __proto__ en el constructor:

```
function Person(name, age){
  this.__proto__ = new lifeForm();
  ...
}
```

A traves del atributo prototype fuera del constructor

```
Person.prototype = new lifeForm();
```

Module Pattern

El module pattern sirve para revelar algunas variables de un scope privado. Es muy similar a la self-invoquing function pero devolvemos un diccionario con las variables que queremos hacer publicas.

```
let myModule = //(1)
(function () {
  let c = 1;
    return { //(2)
      inc: function () {
        c++
      },
      value: function() {
        return c;
      },
  };
})(); // self-invoquing function
> myModule.value() //(3)
1
> myModule.inc() //(4)
undefined
> myModule.value()
2
```

Vemos como podemos acceder a inc y value pero no a c.

Promises

Creando promesas

El constructor de Promise recibe una funcion (normalmente anonima) que llama a una de las dos funciones callback (`resolve` o `reject`) dependiendo del resultado. Al llamar a la callback el estado de la promesa cambia. Las funciones `resolve` y `reject` son de la API de las promesas y no hay que crearlas, solo llamarlas.

```
let promise = new Promise(function(resolve, reject) {
  result = hazalgo(); // normalmente asincrono
  if (/* todo bien */)
    resolve(result);
  else
    reject(error);
});
```

Utilizando promesas

Las promesas tienen un metodo then que sirve para encadenar promesas con funciones de callback. Es decir, la funcion then es asi por dentro:

```
then = function(succeedCallback, errorCallback) {
  if(/* la promesa se ha resuelto bien */)
    succeedCallback(result);
  else
    errorCallback(error);
}
```

Es decir, cuando la promesa se resuelva bien, se llama al callback `succeedCallback` y en caso contrario a `errorCallback`. Esto es útil para encadenar promesas.
A `then` se le puede pasar un solo callback (el `succeedCallback`) o los dos. Si solo ponemos un callback para que la ejecucion del programa vaya a algun lado en caso de que falle la promesa, utilizamos `catch`.
Tanto `then` como `catch` devuelven una promesa por lo que podemos hacer cadenas de promesas que se ejecuten en orden pero de manera asincrona (ver ejemplo mas abajo).

```
promise1.then(succeedCallback, errorCallback);
// es lo mismo que lo siguiente
promise1.then(succeedCallback).catch(errorCallback);
// tambien es lo mismo que lo siguiente (raro)
promise1.then(succeedCallback).then(undefined,errorCallback)
```

Cadenas de promesas y errores

Podemos tener una estructura similar a un `try catch` en un codigo asincrono poniendo un `catch` al final de la cadena de promesas. Si cualquiera de las promesas falla se ejecutara el callback `failureCallback`.

```
doSomething()
.then(result => doSomethingElse(result))
.then(newResult => doThirdThing(newResult))
.then(finalResult => {
console.log("Got␣the␣final␣result:␣"+ finalResult);
})
.catch(failureCallback);
```

NodeJS

Lectura de ficheros

```
const fs = require('fs')

fs.readFile('a1.txt', 'utf-8', (err, data) => {
  if(err) {
    // gestionar error
  }else{
    // hacer algo con los datos
  }
})
```

Servidor web basico

```
const http = require("http");
const fs = require('fs');
http.createServer(function(request, response){
  response.writeHead(200, {'Content-Type': 'text/html'});
  fs.createReadStream('index.html').pipe(response)
  .on("finish", ()=>{
    response.end();
  });
}).listen(8888);
```

En el stream response escribimos lo que el servidor devuelve. Con pipe() enviamos todo lo que lee el stream de lectura de index.html a response. Con on() escuchamos el evento 'finish' que finalizara la conexion con end() al acabar la escritura de index.html en response.

Como debugar

Con `node --inspect app.js` ejecutamos node en modo debug. Vamos a Chrome y entramos en `about:inspect`. Seleccionamos el enlace `inspect` de nuestra aplicacion node. Se abrira el debuger y tenemos que buscar nuestro codigo `.js` en el arbol de ficheros para poder debugar. En la columna de la derecha en la seccion `local` encontraremos las variables del entorno de la aplicacion.

Express.js

Servir directorio public como estatico

```
var app = express();
app.use(express.static(path.join(__dirname,'public')));
```

Middleware

Usamos `next()` para pasar la solicitud al siguiente middleware y que la app no se cuelgue.

```
app.use('/time',function (req, res, next) {
  console.log('Time:', Date.now());
  next();
});
```

En este ejemplo respondemos con un fichero a una peticion get.

```
app.get("/", (req, res) => {
  res.sendFile(path.join(__dirname,'index.html'));
})
```

En este ejemplo respondemos `200 OK` a una peticion post donde enviamos dos user y pass y los obtenemos de la request.

```
app.post("/login", (req, res) => {
  let user = req.body.user;
  let pass = req.body.pass;
  ...
  res.status(200).end()
})
```

Tenemos el objeto req.body para acceder a la peticion y req.session para la sesion actual.

Response

```
res.redirect("/");
res.sendFile(path.join(__dirname,'index.html'));
res.status(200); // 200 OK, 404 not found, 500 server error
res.json(Object);
res.end();
```

Parametros

Con `:parametro` podemos especificar un parametro en la URL de la peticion. Con `req.params.parametro` obtenemos el valor.

```
app.delete("/mail/:mailId", (req, res) => {
  let id = req.params.mailId;
  ...
})
```