# Curso HTML, CSS y JavaScript

**Junio de 2017**

Autor: Fran Linde Blázquez

## TEMA 2:
## JAVASCRIPT AVANZADO

# ÍNDICE

# 1. INTRODUCCIÓN

En este tema vamos a tratar varios de los aspectos más importantes de JavaScript tales como: herencia, closures, scope, this…

En su mayoría trataremos la programación orientada a objetos con JavaScript y todas sus particularidades en JavaScript.

# 2. GUARDANDO FUNCIONES EN VARIABLES

JavaScript es un lenguaje que contiene lo que en programación es conocido como “First-class function”. Las funciones “First-class function” son tratadas como “ciudadanos de primer orden”.

Estos términos técnicos traducidos a un lenaguaje más natural indica que el lenguaje permite almacenar funciones en variables, pasar funciones como argumentos de otras funciones, devolver funciones como resultados de otras funciones e incluso almacenarlas como datos de objetos.

En temas posteriores trataremos varias de estas funcionalidades, por ahora veremos cómo almacenar funciones en variables:

```
// Recibe un string y una longitud máxima
// Devuelve el string acortado
var acortarString = function(string, longitud){
        return string.substring(0, longitud);
};

console.log(acortarString);

var resultado = acortarString("Hola este es un mensaje muy largo.", 10);
console.log(resultado);
```

En el ejemplo hemos definido una función que hemos almacenado en la variable “acortarString”. Podemos observar que su contenido mediante el console.log y posteriormente su ejecución:

# 3. ASIGNANDO Y PASANDO VARIABLES

En JavaScript la asignación de variables o valores se realiza como en la mayoría de lenguajes con un simple igual (**=**). Lo único que debemos tener en cuenta es si esta asignación está realizando una asignación de valor o de referencia.

En JavaScript la asignación y el paso de variables primitivas se realiza siempre por valor, sin embargo, el resto de variables siempre se realiza por referencia.

Resumiendo:

- Por valor los tipos: **string**, **number** y **boolean** (también **null** y **undefined**, pero es lógico, ya que no referencian nada)
- Por referencia: El resto de variables

Debemos tener muy en cuenta estas consideraciones ya que la modificación de una variable primitiva no produce ningún efecto colateral, al contrario que la modificación del valor de un objeto o de un array.

Vamos a realizar una pequeña demostración:

```
// Definimos dos variables
var variable1 = "Soy un tipo primitivo";
var variable2 = variable1;

// Imprimimos el valor antes de modificarlo
console.log(variable1);

// Modificamos el valor
variable2 = "Han modificado mi valor";

// Imprimimos el valor después de modificarlo
console.log(variable1);
```

Si ejecutamos este código en un navegador veremos el siguiente resultado:

```
// Definimos dos variables
var variable1 = "Soy un tipo primitivo";
var variable2 = variable1;

// Imprimimos el valor antes de modificarlo
console.log(variable1);

// Modificamos el valor
variable2 = "Han modificado mi valor";

// Imprimimos el valor después de modificarlo
console.log(variable1);
```

```
Soy un tipo primitivo
Soy un tipo primitivo
```

Se puede comprobar que variable2 ha sido modificada, pero no ha tenido efecto sobre variable1.

Veamos el mismo ejemplo con un array:

```
// Definimos dos variables
var variable1 = ["primer valor", "segundo valor", "tercer valor"];
var variable2 = variable1;

// Imprimimos el valor antes de modificarlo
console.log(variable1);

// Modificamos el valor
variable2.push("Un valor nuevo sobre variable 2");
variable2[0] = "primer valor modificado sobre variable2";

// Imprimimos el valor después de modificarlo
console.log(variable1);
```

```
▶ (3) ["primer valor", "segundo valor", "tercer valor"]
▶ (4) ["primer valor modificado sobre variable2", "segundo valor", "tercer valor", "Un valor nuevo sobre variable 2"]
undefined
>
```

Como se puede observar en el ejemplo las modificaciones realizadas sobre variable2 han tenido efecto sobre variable1. Esto se debe a que la asignación de objetos en JavaScript se realiza por referencia, de manera que variable2 contiene una referencia al mismo espacio de memoria que variable1.

De igual manera sucede cuando pasamos variables a funciones.

Ejercicio: Prueba a realizar una función que modifique un valor que reciba por parámetros, después imprime la variable. Realiza estas pruebas con varios tipos de variables: **number**, **boolean**, **object**... etc.

# 4. HOISTING

Una de las particularidades de JavaScript es lo que se conoce comúnmente como hoisting. Dicha característica consiste en que con independencia de donde esté la declaración de una variable, ésta es movida al inicio del ámbito al que pertenece. Es decir, aunque nuestro código sea como el siguiente:

```
function foo() {
    console.log(x);
    var x=10;
}
```

Realmente se tratará a todos los efectos como si hubiésemos escrito:

```
function foo() {
    var x;
    console.log(x);
    x=10;
}
```

Por supuesto, al ejecutar el código este imprime “undefined” en pantalla, pero no es porque la variable x no esté definida al momento de ejecutar el console.log, es porque no tiene valor (y las variables sin valor asignado se les asigna el valor de undefined). Esto hace que el hoisting pase indvertido muchas veces, pero debemos tener cuidado con él.

Por ejemplo, supongamos el siguiente código:

```
var x='global value';
function foo() {
    console.log(x);
    var x='local value';
    console.log(x);
}

foo();
```

Uno podría esperar que se imprimiese primero “global value” y luego “local value”, ya que parece que cuando se ejecuta el primer console.log(x) la variable x local todavía no existe, por lo que se imprimiría el valor de la variable x global. Pero no ocurre esto. En su lugar dicho código muestra “undefined” y luego “local value”. Eso es debido a que gracias al (o por culpa del) hoisting es como si realmente hubiésemos escrito:

```
var x='global value';
function foo() {
    var x;
    console.log(x);
    x='local value';
    console.log(x);
}

foo();
```

Ahora se puede observar claramente como la variable x local oculta a la variable x global incluso antes del primer console.log.

Es importante además recalcar que, a diferencia de otros lenguajes, el código dentro de las llaves de un if o de un for no abre un ámbito nuevo. Supongamos un código como el que viene a continuación:

```
function foo() {
    var item={v:'value'};
    for (var idx in [0,1]) {
        var item = {i: idx};
        console.log(item);
    }
    console.log(item);
}

foo();
```

Este código parece que tenga que imprimir {i:0}, {i:1} (al iterar dentro del for) y luego {v:'value'} (el valor de la variable item de fuera del for). Pero realmente la declaración de la variable item dentro del for se mueve fuera de este, ya que el bloque for no declara un nuevo ámbito de visibilidad. De este modo el código es equivalente a:

```
function foo() {
    var item;       // Primer item declarado
    var item;       // Segundo item declarado dentro del for
    var idx;        // Variable idx declarada en el for
    item={v:'value'};
    for (idx in [0,1]) {
        item = {i: idx};
        console.log(item);
    }
    console.log(item);
}

foo();
```

Declarar dos veces una misma variable en JavaScript no da error y ahora se puede ver como la salida real de nuestro programa será {i:0}, {i:1} y luego{i:1} otra vez. Fíjate que lo mismo ocurriría en el caso de la variable idx si hubiese otra variable idx declarada antes del for.

En resumen, el hoisting es una característica de JavaScript que aunque muchas veces pasa inadvertida debemos comprender, para así entender algunos comportamientos del lenguaje que de otro modo nos parecerían totalmente erráticos.

# 5. “USE STRICT”

La directiva use strict es una directiva que indica el modo en que el navegador debe ejecutar el código JavaScript. Podríamos hablar de dos modos de ejecución JavaScript: el “modo normal”, que es el que hemos visto hasta ahora, y el “modo estricto”, que vamos a ver a continuación.

Veamos las diferencias entre un modo y otro:

1. En modo estricto es obligatoria la declaración de variables, mientras que en el modo normal no es necesario declarar una variable para poder usarla.

2. En modo estrcito no se puede definir una variable más de una vez

3. En modo estricto no se pueden definir nombres de parámetros duplicados

4. En modo estricto no se permite borrar una variable mediante **delete**

Para hacer uso del modo estricto debemos incluir la sentencia “use strict” al inicio de un fichero JavaScript si queremos que afecte a todo ese documento, o podemos incluirla delante de una función y que solo afecte a dicha función:

```
"use strict";

// Esto va a producir un error, porque variable no ha sido definida
variable = 99;
```

Desafortunadamente, el modo estricto no está soportado por todos los navegadores, por eso debemos conscientes de los errores que nos puede ocasionar su uso, ya que puede hacer que el mismo código puede funcionar de forma diferente dependiendo del navegador en el que se ejecute.

# 6. DEFINIENDO CLASES EN JAVASCRIPT

En JavaScript no se encuentra el concepto de clase como tal (salvo en las últimas versiones), por este motivo vamos a ver de qué maneras podemos simular el comportamiento de una clase en JavaScript.

## 6.1 Definiendo clases mediante objetos

Como hemos visto en temas anteriores, un objeto puede almacenar distintas variables y también hemos aprendido a guardar funciones en variables.

Por tanto, ya tenemos los ingredientes necesarios para crear una primera versión de clase:

```
var animal = {
    nombre: "Perro",

    emitirSonido: function(){
        console.log("GRRRRR");
    }
}

animal.emitirSonido();
```

Los objetos pueden considerarse la primera aproximación a las clases, ya que nos permiten crear variables que contienen tanto propiedades como funciones. Pero aún, nos falta algo.

¿Cómo podemos acceder desde las funciones de nuestra “clase” a sus variables? En este punto entra **this** en juego

## 6.2 El uso de “this”

La variable **this** tiene una funcionalidad especial en JavaScript: **this** se resuelve (obtiene un valor) cuando llamamos a una función y nos permite acceder al objeto que la posee durante su ejecución.

Veamos un ejemplo:

```
var animal = {
    nombre: "Perro",
    sonido: "Guau!",

    emitirSonido: function(){
        console.log(this.sonido);
    }
}

animal.emitirSonido();
```

Mediante **this** podemos acceder en cada momento a los valores que tiene el objeto animal, de otra manera no nos sería posible.

## 6.3 Definiendo clases mediante funciones y “new”

Para mejorar nuestras clases vamos a usar funciones en vez de usar objetos. Vamos a ver cómo el uso de funciones nos ofrecerá un “constructor” para poder crear instancias de nuestra clase, lo cual no nos era posible creando clases directamente como objetos.

```
var Animal = function(){
    this.nombre = "Perro"
    this.sonido = "Guau!";

    this.emitirSonido = function(){
        console.log(this.sonido);
    }
}

var miPerro = new Animal();

miPerro.emitirSonido();
```

Mediante el uso de new estamos creando un nuevo objeto con la estructura definida en la función **“animal”,** y que se inicializa con una variable **“this”** nueva.

La primera ventaja que comentábamos, es que ahora podemos hacer uso de **new** para crear nuevas instancias. Vamos a ver cómo además, podemos hacer uso de variables en nuestro constructor:

```
var Animal = function(nombre, sonido){
    this._nombre = nombre;
    this._sonido = sonido;

    this.emitirSonido = function(){
        console.log("El " + this._nombre + " hace " + this._sonido);
    }
}

var miPerro = new Animal("Perro", "Guau!");
var miGato = new Animal("Gato", "Miau!");

miPerro.emitirSonido();
miGato.emitirSonido();
```

Si ejecutamos en un navegador este código podemos comprobar que el funcionamiento es el esperado:

```
> var Animal = function(nombre, sonido){
      this._nombre = nombre;
      this._sonido = sonido;

      this.emitirSonido = function(){
          console.log("El " + this._nombre + " hace " + this._sonido);
      }
  }

  var miPerro = new Animal("Perro", "Guau!");
  var miGato = new Animal("Gato", "Miau!");

  miPerro.emitirSonido();
  miGato.emitirSonido();


  El Perro hace Guau!
  El Gato hace Miau!
```

## 6.4 Prototype

Ya hemos visto cómo podemos definir una clase en JavaScript y también hemos creado varias instancias de nuestra clase haciendo uso de un constructor parametrizado. Pero hay una cosa que no estamos teniendo en cuenta: el rendimiento.

Después de crear nuestras instancias **“miPerro”** y **“miGato”** hemos creado dos variables que ocupan dos espacios en memoria idénticos: ambos han reservado espacio en memoria para el método **emitirSonido()**

Para comprobarlo, basta con imprimir por consola ambas variables:

```
var miPerro = new Animal("Perro", "Guau!");
var miGato = new Animal("Gato", "Miau!");

console.log(miPerro);
console.log(miGato);
```

```
▼Animal {_nombre: "Perro", _sonido: "Guau!", emitirSonido: function}
  ▶emitirSonido: function ()
    _nombre: "Perro"
    _sonido: "Guau!"
  ▶__proto__: Object
▼Animal {_nombre: "Gato", _sonido: "Miau!", emitirSonido: function}
  ▶emitirSonido: function ()
    _nombre: "Gato"
    _sonido: "Miau!"
  ▶__proto__: Object
```

Como puede observarse, ambas variables guardan la función completa en su interior, con su correspondiente carga en memoria. Si definimos 100 animales, tendremos 100 funciones “**emitirSonido**” idénticas. Para evitarlo, haremos uso de Prototype.

**Prototype** permite que varios objetos de la misma clase compartan métodos y estos no sean replicados en memoria de manera masiva e ineficiente.

Veamos cómo deberíamos haber creado nuestra clase animal:

```
var Animal = function(nombre, sonido){
    this._nombre = nombre;
    this._sonido = sonido;
}

Animal.prototype = {
    emitirSonido: function(){
        console.log("El " + this._nombre + " hace " + this._sonido);
    }
}

var miPerro = new Animal("Perro", "Guau!");
var miGato = new Animal("Gato", "Miau!");

miPerro.emitirSonido();
miGato.emitirSonido();

console.log(miPerro);
console.log(miGato);
```

Podemos comprobamos nuestro código en un navegador:

```
> var Animal = function(nombre, sonido){
      this._nombre = nombre;
      this._sonido = sonido;
  }

  Animal.prototype = {
      emitirSonido: function(){
          console.log("El " + this._nombre + " hace " + this._sonido);
      }
  }

  var miPerro = new Animal("Perro", "Guau!");
  var miGato = new Animal("Gato", "Miau!");

  miPerro.emitirSonido();
  miGato.emitirSonido();

  console.log(miPerro);
  console.log(miGato);
  El Perro hace Guau!
  El Gato hace Miau!
  ▼Animal {_nombre: "Perro", _sonido: "Guau!"} i
      _nombre: "Perro"
      _sonido: "Guau!"
    ▶__proto__: Object
  ▼Animal {_nombre: "Gato", _sonido: "Miau!"} i
      _nombre: "Gato"
      _sonido: "Miau!"
    ▶__proto__: Object
```

# 7. ÁMBITO DE UNA FUNCIÓN/VARIABLE (SCOPE)

En JavaScript cuando declaramos una variable mediante **var** estamos haciendo que este disponible de forma global y la hace accesible desde cualquier parte de nuestra aplicación.

Por este motivo hay que tener mucho cuidado a la hora de definir nombres de variables o funciones, ya que podemos estar sobbrescribiendo un valor o una función definidas en otra parte de nuestra página.

Sin embargo, no todas las las variables definidas con var están disponibles a nivel de toda la aplicación. Las variables que se definen dentro de una función, tienen ámbito local. Esto quiere decir que solo serán accesibles desde esa función “hacia abajo”, pero no será visible desde fuera de la propia función.

Vamos a verlo con un ejemplo:

```
// Definimos una variable global
var variable = "global";

// Definimos una función global que define
// otra variable con el mismo nombre en su interior
function comprobarAmbitos(){
    var variable = "local";
    return variable;
}

// Comprobamos el valor inicial de variable
console.log(variable);

// Llamamos a nuestra función y comprobamos
// el "nuevo" valor de variable
console.log(comprobarAmbitos());

// Comprobamos que no a afectado a la variable global
console.log(variable);
```

Ejecutamos el código en un navegador para comprobar el resultado:

```
// Definimos una variable global
var variable = "global";

// Definimos una función global que define
// otra variable con el mismo nombre en su interior
function comprobarAmbitos(){
    var variable = "local";
    return variable;
}

// Comprobamos el valor inicial de variable
console.log(variable);

// Llamamos a nuestra función y comprobamos
// el "nuevo" valor de variable
console.log(comprobarAmbitos());

// Comprobamos que no a afectado a la variable global
console.log(variable);
```

```
global
local
global
undefined
```

Se puede comprobar que el ámbito de la función “**comprobarAmbitos()**” es un ámbito nuevo, y por este motivo a la hora de definir una variable, esta no afecta al ámito global.

# 8. CLOSURES

Las closures son un patrón de diseño muy utilizado en JavaScript. Las closures nos aportan en JavaScript la funcionalidad de variables privadas y públicas que no encontramos en JavaScript por defecto.

Pero antes de ver cómo qué es una closure en detalle, debemos repasar varios aspectos de JavaScript:

## 8.1 Funciones como Objeto

En JavaScript las funciones son objetos. Ya hemos visto anteriormente que podemos guardar una función en una variable. Esto podemos hacerlo porque las funciones en JavaScript heredan de Object, y por tanto podemos tratarlas como tal.

## 8.2 Anidando funciones

Puesto que las funciones en JavaScript heredan de Object, podemos encontrar funciones dentro de funciones en JavaScript.

Cada función dentro de una función definirá un ámbito nuevo como ya hemos visto en el punto anterior y sus valores no serán visibles desde el exterior a menos que los devolvamos mediante un return, como veremos más adelante.

## 8.3 Definición de closure y patrón module

Tras haber visto estos dos puntos, ya podemos hacer una definición de closure:

Una **closure** es una funció que encapsula una serie de variables y definiciones locales, que únicamente son accesibles a través de su return.

En JavaScript (sin contar las últimas especificaciones) no existe el concepto de clase como tal, y por este motivo las closures son usadas para simular el concepto de clase en JavaScript.

Veamos un ejemplo de closure:

```
var mochilaSecreta = function() {
    var _objetos = [];

    function introducirObjeto(objeto) {
        _objetos.push(objeto);
    }

    function extraerObjeto() {
        var objetoSaliente = _objetos.pop(objeto);
        return objetoSaliente;
    }

    function numeroDeObjetos() {
        return _objetos.length;
    }

    return {
        introducirObjeto: introducirObjeto,
        extraerObjeto: extraerObjeto,
        numeroDeObjetos: numeroDeObjetos
    }
}

var varMochilaSecreta = mochilaSecreta();

varMochilaSecreta.introducirObjeto("bocadillo");
varMochilaSecreta.introducirObjeto("linterna");
varMochilaSecreta.introducirObjeto("botella");

// Vemos el número de objetos
console.log(varMochilaSecreta.numeroDeObjetos());
```

En este ejemplo hemos creado una “mochila” que nos permitirá almacenar objetos, pero nadie podrá ver lo que hay en su interior, ya que el array “**_objetos**” se encuentra definido bajo el ámbito de la función mochila secreta.

Cabe destacar que hemos nombrado la variable “**_objetos**” con el guión bajo por delante por convención, ya que es una recomendación muy extendida, el que las variables privadas se nombren comenzando con guión bajo.

Mediante closures podemos también simular el comportamiento de una clase, ya que la llamada a nuestra función realiza las funciones de constructor, pudiendo incluso recibir parámetros que inicialicen nuestro objeto:

```
var mochilaSecreta = function(nombreDeLaMochila) {
    var _nombre = nombreDeLaMochila;
    var _objetos = [];

    function introducirObjeto(objeto) {
        _objetos.push(objeto);
    }

    function extraerObjeto() {
        var objetoSaliente = _objetos.pop(objeto);
        return objetoSaliente;
    }

    function numeroDeObjetos() {
        return _objetos.length;
    }

    function getNombre(){
        return _nombre;
    }

    return {
        introducirObjeto: introducirObjeto,
        extraerObjeto: extraerObjeto,
        numeroDeObjetos: numeroDeObjetos,
        getNombre: getNombre
    }
}

var varMochilaSecreta = mochilaSecreta("Para la excursión");

varMochilaSecreta.introducirObjeto("bocadillo");
varMochilaSecreta.introducirObjeto("linterna");
varMochilaSecreta.introducirObjeto("botella");


// Vemos el número de objetos
console.log(varMochilaSecreta.numeroDeObjetos());

// Vemos el nombre
console.log(varMochilaSecreta.getNombre());
```

# 9. HERENCIA EN JAVASCRIPT

Como hemos comentado en puntos anteriores JavaScript no es un lenguaje orientado a objetos ya que no ofrece soporte a la creación de clases (salvo en las últimas versiones). De igual modo sucede con la herencia.

Aun así, haciendo uso de Prototype se puede conseguir simular la herencia en JavaScript.

## 9.1 Herencia con Prototype

Prototype es una propiedad que tienen todos los objetos JavaScript y representa un puntero a otro objeto. Cuando el navegador intenta acceder a una propiedad del objeto, si no la encuentra mirará después en su prototipo.

Por ejemplo:

```
var Animal = function(nombre, sonido) {
    this._nombre = nombre;
    this._sonido = sonido;
}

Animal.prototype = {
    emitirSonido: function() {
        console.log("El " + this._nombre + " hace " + this._sonido);
    }
}

var Perro = function(raza) {
    this._raza = raza;
}

Perro.prototype = new Animal("perro", "Guau!");

var miPerro = new Perro("Pastor Alemán");

miPerro.emitirSonido();
```

De esta manera estamos definiendo que la clase Perro heredará de la clase Animal.

¿Qué sucedería si quisiéramos añadir métodos a nuestra clase Perro? Si lo hiciéramos mediante **Perro.prototype** estaríamos sobrescribiendo el valor que acabamos de setearle, por lo que debemos añadir las nuevas funciones sobre la propiedad Prototype actual de la siguiente manera:

```
var Animal = function(nombre, sonido) {
    this._nombre = nombre;
    this._sonido = sonido;
}

Animal.prototype = {
    emitirSonido: function() {
        console.log("El " + this._nombre + " hace " + this._sonido);
    }
}

var Perro = function(raza) {
    this._raza = raza;
}

Perro.prototype = new Animal("perro", "Guau!");

// añadiendo funciones al prototype
Perro.prototype.dimeRaza = function() {
    console.log(this._raza);
}

var miPerro = new Perro("Pastor Alemán");

miPerro.emitirSonido();
miPerro.dimeRaza();
```

## 9.2 Sobrescribiendo métodos heredados

Como comentábamos al comienzo de este punto, JavaScript hace uso de la propiedad ProtoType de la siguiente manera: si al buscar una propiedad en un objeto no la encuentra, buscará en su propiedad ProtoType.

Por este motivo sobrescribit un método heredado es tan simple como definirlo en el objeto hijo, de manera que el navegador non tenga que ir a buscar en el padre.

Veamos un ejemplo:

```
var Animal = function(nombre, sonido) {
    this._nombre = nombre;
    this._sonido = sonido;
}

Animal.prototype = {
    emitirSonido: function() {
        console.log("El " + this._nombre + " hace " + this._sonido);
    }
}

var Perro = function(raza) {
    this._raza = raza;

    this.emitirSonido = function(){
        alert("GUAUUU !!");
    };
}

Perro.prototype = new Animal("perro", "Guau!");

var miPerro = new Perro("Pastor Alemán");

miPerro.emitirSonido();
```

Esta implementación no estaría haciendo uso de lo que explicamos en el punto 2.4. Para ello en vez de definir una variable “emitirSonido” dentro de Animal, lo correcto sería añadir la función a su ProtoType:

```
var Animal = function(nombre, sonido) {
    this._nombre = nombre;
    this._sonido = sonido;
}

Animal.prototype = {
    emitirSonido: function() {
        console.log("El " + this._nombre + " hace " + this._sonido);
    }
}

var Perro = function(raza) {
    this._raza = raza;
}

Perro.prototype = new Animal("perro", "Guau!");
Perro.prototype.emitirSonido = function(){
    alert("GUAUUU !!");
};

var miPerro = new Perro("Pastor Alemán");

miPerro.emitirSonido();
```

# 10. LA VARIABLE ARGUMENTS

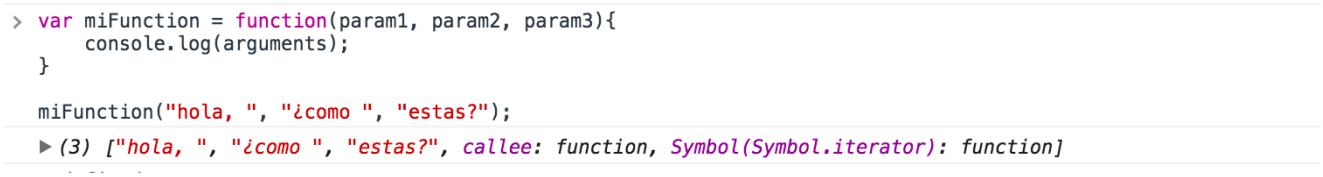
La variable argumente es un objeto similar a un array que se corresponde con los objetos que una función ha recibido como parámetros. Hay que tener en cuenta que “arguments” realmente no es un array, pero permite ser accedido mediante corchetes e índices (como un array) y posee la propiedad length (también del mismo modo que los arrays).

Veamos un ejemplo:

```
var miFunction = function(param1, param2, param3){
    console.log(arguments);
}

miFunction("hola, ", "¿como ", "estas?");
```

Esto producirá el siguiente resultado si lo ejecutamos en un navegador:




Si deseamos obtener un verdadero array de arguments debemos hacerlo de la siguiente manera:

```
var miFunction = function(param1, param2, param3){
    var args = Array.prototype.slice.call(arguments);
    console.log(args);
}

miFunction("hola, ", "¿como ", "estas?");
```

Una de las potencias de arguments, es realizar funciones que acepten u número de parámetros indefinidos.

Por ejemplo, si quisiéramos realizar una función que concatene strings lo podemos hacer de la siguiente forma:

```
function miConcat(separador) {
    var resultado = "";

    // Iterar a través de los otros argumentos enviados
    for (var i = 1; i < arguments.length; i++)
        resultado += arguments[i] + separator;

    return resultado;
}

// Devuelve "rojo, naranja, azul, "
miConcat(", ", "rojo", "naranja", "azul");

// Devuelve "salvia. albahaca. oregano. pimineta. perejil. "
miConcat(". ", "salvia", "albahaca", "oregano", "pimineta", "perejil");
```

Nuestra función “**concat**” aceptará ser sobrecargada con tantas variables como deseemos.

# 11. Asincronía en JavaScript

JavaScript no es un lenguaje multihilo, por lo que cuando debamos enfrentarnos a la ejecución de una parte del código que requiera esperar un determinado tiempo, no podremos lanzar el proceso en otro hilo.

Veamos cómo se trata la asincronía en JavaScript:

## 11.1 Callbacks

Un callback es una función que se pasa por parámetros a una segunda función y que será ejecutada cuando haya terminado esta segunda.

Veamos un ejemplo:

```
function loadCSS(url, callback) {
    var elem = window.document.createElement('link');
    elem.rel = "stylesheet";
    elem.href = url;
    window.document.head.appendChild(elem);
    callback();
}
loadCSS('styles.css', function() {
    console.log("Estilos cargados");
});
```

En este ejemplo tenemos una función llamada loadCSS a la que pasamos una url , presumiblemente que apunte a un fichero .css y una función de callback como parámetros, la función básicamente crea un elemento link y lo añade al final de la etiqueta <head>

Cuando ejecutamos esta función, le pasamos la url de styles.css y una función anónima que será el callback. Lo que hará será imprimir por la consola Estilos cargados cuando finalice la carga.

Este es un ejemplo básico de una función asíncrona con callbacks.

## 11.2 Promesas

Con la llegada de ES5 aparecieron las promesas.

Veamos cómo se utilizan:

```
// Asumamos que loadCSS devuelve una promesa
var promise = loadCSS('styles.css');
promise.then(function() {
    console.log("Estilos cargados");
});
promise.catch(function(err) {
    console.log("Ocurrió un error: " + err);
});
```

Si loadCSS fuese una función asíncrona que devuelve una promesa, la resolveríamos utilizando la función then . Esta función se ejecuta cuando la promesa ha sido resuelta. Si hubiese ocurrido algún error, se ejecutaría la función catch , donde recogemos el error y lo tratamos.

La función la guardamos en la variable promise . Como es posible aplicar "chaining", es decir, anidamiento de funciones, podemos escribir la ejecución de la siguiente manera, que es más legible y elegante:

```
loadCSS('styles.css')
    .then(function() {
        console.log("Estilos cargados");
    })
    .catch(function(err) {
        console.log("Ocurrió un error: " + err);
    });
```

# 12. EVENTOS EN JS

JavaScript nos permite, por su entorno de programación, una programación orientada a eventos. Podemos detectar eventos que ocurran en el navegador (o en el servidor) y actuar en base a ellos. También podemos crear nuestros propios eventos y suscribirnos, sería lo que se conoce como patrón PubSub (Publicador-Suscriptor)

## 12.1 Manejando eventos

Imaginemos que hacemos clic en un elemento HTML de una página web, que no necesariamente sea un enlace, en ese momento se dispara un evento que podemos capturar y realizar la función que estimemos conveniente, una llamada AJAX, un cambio de estilo, etc...

Veamos un ejemplo con código:

```
function onClickHandler(e) {
    e.preventDefault();
    console.log(e);
}

// Asociamos a un elemento de la web el evento
var target = document.querySelector('#respuesta');
target.addEventListener('click', onClickHandler, false);
// Función que manejará el evento
```

la función e.preventDefault() evita que se dispare una acción por defecto. Imaginemos que este evento lo estamos realizando sobre un enlace o sobre un botón de un formulario. Gracias a esta función, evitaremos que el enlace nos redireccione al hipervínculo o que el botón dispare la acción por defecto del formulario. De esta forma podemos controlar el flujo de la acción en cualquier evento.

En el ejemplo de código anterior, estamos asociando la función onClickHandler al evento click en un elemento HTML con el id respuesta. Esto quiere decir que cuando hagamos clic con el ratón en ese elemento HTML, se ejecutará la función, que en el ejemplo hemos puesto mostrará en la consola la información del evento:

## 12.2 Propagación de eventos

Los eventos pueden propagarse hacia arriba en el documento. En el siguiente ejemplo vamos a escuchar el evento click en el elemento header que contiene un h1 y un h2:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Es es mi blog</title>
        <link rel="stylesheet" type="text/css" href="blog.css">
        <script type="text/javascript" src="blog.js"></script>
        <meta name="viewport" content="width=device-width, initial-scale=1">
    </head>

    <body>
        <header>
            <h1>Hola Mundo</h1>
            <h2>SubHeader</h2>
        </header>
    </body>
</html>
```

Nuestro fichero JS luce así:

```
window.onload = function(){
    var header = document.querySelector('header');
    header.addEventListener('click', function(e) {
        console.log('Has clickado en ' + e.target.nodeName);
    });
}
```

Con el manejador creado, se imprimirá en consola el mensaje Has clicado en seguido del nombre del elemento gracias a e.target.nodeName . Si clicamos dentro del h1 nos mostrará Has clickado en H1 y si lo hacemos en el h2 mostrará Has clicado en H2 .

Aunque estemos escuchando el elemento header , tenemos acceso a todos los nodos que se encuentren dentro de el.

Ahora imaginemos que también añadimos un escuchador de eventos al documento raíz document como el siguiente:

```
document.addEventListener('click', function(e) {
    console.log('Has clickado en el documento');
});
```

Cuando hagamos clic en cualquier parte de la página, nos mostrará el mensaje Has clicado en el documento . Pero si clicamos en una parte del header tendremos los dos mensajes por consola.

Si queremos mantener el escuchador en el document pero cuando hagamos clic en header no salte el otro evento, podemos hacer uso de e.stopPropagation() , para evitar que se propague de abajo a arriba.

```
header.addEventListener('click', function(e) {
    e.stopPropagation();
    console.log('Has clickado en ' + e.target.nodeName);
});
```

De esta forma si clicamos en h1 o en h2 obtendremos Has clicado en HX , y sin que aparezca el evento asociado a document .

Tenemos a nuestra disposición numerosos eventos sobre los que podemos actuar. En este enlace tienes la lista completa: https://developer.mozilla.org/en-US/docs/Web/Events

## 12.3 Patrón Pub-Sub

Además de los eventos que nos proporciona el DOM, podemos crear los nuestros propios. Esto se le conoce como el patrón PubSub. Realizamos una acción y publicamos o emitimos un evento. En otra parte de nuestro código escuchamos ese evento y cuando se produzca realizamos otra acción. Veamos un ejemplo con código.

Vamos a crear un closure llamado pubsub donde tendremos 2 funciones, la función subscribe donde escucharemos los eventos, y la función publish que los publicará

```
window.onload = function() {
    var pubsub = (function() {
        // Este objeto actuará como cola de todos los eventos que se
        // produzcan. Los guardará con el nombre del evento como clave
        // y su valor será un array con todas las funciones callback encoladas.
        var suscriptores = {};

        function subscribe(event, callback) {
            // Si no existe el evento, creamos el objeto y el array de callbacks
            // y lo añadimos
            if (!suscriptores[event]) {
                var suscriptorArray = [callback];
                suscriptores[event] = suscriptorArray;
                // Si existe, añadimos al array de callbacks la función pasada por
                // parámetro
            } else {
                suscriptores[event].push(callback);
            }
        }

        function publish(event) {
            // Si el evento existe, recorremos su array de callbacks y los
            // ejecutamos en orden.
            if (suscriptores[event]) {
                suscriptores[event].forEach(function(callback) {
                    callback();
                });
            }
        }
        return {
            // Los métodos públicos que devolvemos serán `pub` y `sub`
            pub: publish,
            sub: subscribe
        };
    }());
}
```

Por tanto, para escuchar un evento y realizar una operación asociada, deberemos llamar a pubsub.sub , pasarle como primer parámetro el nombre del evento, en este caso le vamos a llamar miEvento , seguido de una función manejadora. En este caso simplemente vamos a imprimir por consola que el evento se ha disparado.

```
pubsub.sub('miEvento', function(e) {
    console.log('miEvento ha sido lanzado!');
});
```

Para poder lanzar el evento y que posteriormente sea recogido por la función pubsub.sub , lo emitimos con la función pubsub.pub pasándole como parámetro el nombre del evento. En este caso miEvento :

```
pubsub.pub('miEvento');
```

## 12.4 Patrón Pub-Sub con Datos

Además de emitir y escuchar el evento, podemos pasar un objeto con datos en la operación, y así utilizarlo a lo largo de nuestra aplicación.

Por ejemplo, queremos que al emitir un evento, poder pasar un objeto de la siguiente manera:

```
pubsub.pub('MiEvento', {
    misDatos: 'Estos son mis datos'
});
```

Y al escucharlo, poder mostrarlo:

```
pubsub.pub('MiEvento', {
    misDatos: 'Estos son mis datos'
});
```

Para lograrlo, debemos modificar un poco la función pubsub creando un objeto dónde almacenaremos los datos que publiquemos. Nuestro clousure pubsub quedaría así:

```
var pubsub = (function() {
    var suscriptores = {};

    function EventObject() {};

    function subscribe(event, callback) {
        if (!suscriptores[event]) {
            var suscriptorArray = [callback];
            suscriptores[event] = suscriptorArray;
        } else {
            suscriptores[event].push(callback);
        }
    }

    function publish(event, data) {
        var eventObject = new EventObject();
        eventObject.type = event;
        if (data) {
            eventObject.data = data;
        }
        if (suscriptores[event]) {
            suscriptores[event].forEach(function(callback) {
                callback(eventObject);
            });
        }
    }
    return {
        pub: publish,
        sub: subscribe
    };
}());
```

# 13. MODIFICANDO EL CONTEXTO

Imaginemos el siguiente ejemplo:

```
var alice = {
    nombre: "Alice",
    cansarse: function() {
        console.log(this.nombre);
    }
};

var myFunction = alice.cansarse;
```

Si llamamos a myFunction directamente lo estaríamos llamando sin contexto por lo que la variable this tendría el objeto global dentro de myFunction, como podemos hacer que ejecute myFunction pero pasándole alice como this? Para ésto tenemos las funciones .call() y .apply(), empecemos por la función .call().

## 13.1. Call

La función .call() recibe los mismos argumentos que la función más uno, el valor que tendrá this que se pasa antes que los demás argumentos. Es decir, nuestra función myFunction no recibe ningún argumento así que si llamamos a su método .call() y le pasamos lo que queremos que sea this es decir, alice conseguiremos que el método funcione igual que si lo hubiésemos llamado con alice.cansarse

```
myFunction.call(alice);
```

Ahora vamos a probar lo mismo con una función que reciba argumentos:

```
var alice = {
    nombre: "Alice",
    saludar: function(amigo1, amigo2) {
        alert("Hola " + amigo1 + " y " + amigo2 + ", yo soy " + this.nombre);
    }
};

var myFunction = alice.saludar;
myFunction.call(alice, "Bob", "Rob");
```

## 13.2. Apply

El método .apply() actúa de forma bastante similar a .call(), pero con una variación, solo recibe dos argumentos, el primero es el contexto de la función, el valor de this y el segundo será un array que contendrá los argumentos que se le pasarán a la función, veamos su uso en el ejemplo anterior:

```
myFunction.apply(alice, [ "Bob", "Rob" ]);
```

Esto aunque en un principio parezca bastante inútil nos servirá cuando, queriendo o no cambiar el contexto de una función, querramos llamarla y no sepamos ni nos interese saber cuántos argumentos tiene, supongamos que tenemos la función callWithAlice() que llama a la función .saludar() de alice y le pasa todos los argumentos que recibe.

Nota 1: Para ésto hace falta aclarar que el objeto arguments es una especie de array con los argumentos pasados a la función, más adelante profundizaremos en ello.

Nota 2: En éste caso no queremos cambiar el contexto, pero como estamos llamando a .apply() tenemos que darle uno, por lo que le damos alice que es el contexto que ya tenía.

```
function callWithAlice() {
    alice.saludar.apply(alice, arguments);
}
callWithAlice("Rob", "Bob");
```

## 13.3. Bind

Ahora que ya entendemos el contexto, .call() y .apply() sabremos que cuando pasemos una función como callback si no queremos perder el contexto de la función deberemos hacer:

```
function esperarUnSegundo(callback) {
    setTimeout(function() {
        callback();
    }, 1000);
});

esperarUnSegundo(function() {
    alice.myMethod();
```

O bien:

```
function esperarUnSegundo(callback, context) {
    setTimeout(function() {
        callback.call(context);
    }, 1000);
});

esperarUnSegundo(alice.myMethod, alice);
```

Pero ésto puede ser un poco tedioso cuando manejas muchos callbacks de éste tipo, para ello se ha creado el método .bind(). Es un método de Function que devuelve otra función. Confuso, verdad?

.bind() recibe un argumento, el contexto que se le podrá a la función sobre la que se aplica el .bind() y devolverá una función que cuando sea llamada ejecutará la función original con el contexto que se le pasó a .bind(). Lo veremos mejor con un ejemplo:

```
var alice = {
    nombre: "Alice",
    saludar: function() {
        console.log("Hola! Soy " + this.nombre);
    }
};

var myFunction = alice.saludar.bind(alice);
myFunction();
```

Lo que hemos hecho en la línea 8 es crear una función que cuando sea invocada llamará a saludar y le pasará alice como contexto.