APUNTES HTML5:
JavaScript

TOMO II:
Lenguaje JavaScript

José Juan Urrutia Milán

Reseñas

- Curso HTML5: https://www.youtube.com/playlist?list=PLU8oAlHdN5BnX63lyAeV0LzLnpGudgRrK
- Curso JavaScript: https://www.youtube.com/playlist?list=PLU8oAlHdN5BmpobVmj1IlneKlVLJ84TID
- w3c (página que decide los estándares CSS/HTML...): www.w3c.org

Siglas/Vocabulario

- **IDE:** Entorno de Desarrollo Integrado.
- **OS:** Operator System (Sistema Operativo).

Leyenda

Cualquier abreviatura o referencia será subrayada.
Cualquier ejemplo será escrito en **negrita.**
Cualquier palabra de la que se pueda prescindir irá escrita en cursiva.
Cualquier abreviatura viene explicada a continuación:

Abreviaturas/Referencias:

- 123: Hace referencia a cualquier número.
- nombre: Hace referencia a cualquier palabra/cadena de caracteres.
- a: Hace referencia a cualquier caracter.
- cosa: Hace referencia a cualquier número/palabra/cadena.
- Tipo_var o **…** : Hace referencia a cualquier tipo de variable primitiva o de tipo String.
- nombre_var: Hace referencia a cualquier nombre que se le puede dar a una variable.
- código: Hace referencia a cualquier instrucción. (Se usará para indicar dónde se podrá inscribir código.)
- variable: Hace referencia a cualquier variable.
- condición: Hace referencia a cualquier condición. Entiéndase por condición, una afirmación que devuelve un true o un false. Ej: (variable == 123). *Una variable del tipo boolean puede ser usada como una condición.

Índice

Título IV: Condicionales

if

switch

Título V: Bucles

for

while

do while

clearInterval(intervalo)

Título VI: Funciones

Llamar a función

Funciones que reciben parámetros

Funciones que devuelven parámetros

Funciones anónimas

Título VII: Creación de JSON

Acceder a JSON

Capítulo II: Prototipos y POO

Título I: Creación de un prototipo

Arrays en prototipos

Funciones en prototipos

Título II: Lectura de un prototipo

Lectura de arrays dentro de prototipos

Lectura de prototipos dentro de prototipos

Título III: Creación de una clase

Instanciación de una clase

Título IV: Creación de campos de clase

this

Introducción

El objetivo de este curso es aprender el lenguaje JavaScript, su sintaxis y demás.
La principal función de JavaScript es aportar funcionalidad a páginas web creadas con HTML y CSS3.
JavaScript es un lenguaje de programación.
En JavaScript, todas las sentencias terminan en un “;”.
En JavaScript, para especificar textos podemos usar tanto comillas simples ‘ como dobles “ aunque se recomiendan las segundas.
El código JavaScript de una página HTML se puede indicar en 3 sitios diferentes:

1. En un archivo independiente con extensión .js
2. Dentro de la etiqueta head de un documento html.
3. Dentro de una etiqueta inline en casos concretos (desaconsejada).

JavaScript es case sensitive , por lo que tener cuidado con las mayúsculas y las minúsculas.
JavaScript es un lenguaje orientado a prototipos, aunque dispone de una opción para crear clases (aunque verdaderamente son prototipos con una máscara).

Capítulo I: Conceptos básicos

En este capítulo, abordamos JavaScript en sí (sintaxis y demás), dejando a un lado html y css.

Comentarios

Para realizar comentarios en JavaScript (partes que no se tendrán en cuenta), debemos especificar /* antes y */ después:

/*alert(“Hola”);
alert(“Adios”);*/

*Si sólo queremos comentar una línea, podemos usar // delante de la línea a comentar:

//alert(“Hola”);

alert(mensaje)

Imprime el mensaje **mensaje** en una ventana emergente que se ejecuta en primer plano, autofocal y que detiene la ejecución del hilo.

prompt(mensaje)

Imprime el mensaje **mensaje** en una ventana emergente que se ejecuta en primer plano, autofocal y que detiene la ejecución del hilo, la cual tiene un campo de texto en el que el usuario puede escribir.
Devuelve lo que ha escrito el usuario en formato **String** .

document.write(codigo)

Nos permite añadir código HTML al final del último código HTML.
*Como HTML no tiene en cuenta los saltos de línea, podemos crear una etiqueta con diferentes **document.write()** , tantos como necesitemos.

Signo decimal

El signo decimal en JavaScript es un punto (.):

var n = 0.5;

Operadores

Aritméticos

-Suma/Concatenación: +
-Aumento: ++
-Incremento en: +=5 **/Concatenar a: nombre** += “.”
-Resta: -
-Decremento: --
-Decremento en: -=5
-Multiplicación: *
-Multiplica por: *=5
-División: /
-Divide entre: /=5
-Resto: %
Idéntico a Java.
*La concatenación funciona igual a la de Java.

Comparación

-Igual a: ==
-(*)Estrictamente igual a: ===
-Menor: <
-Menor o igual: <=
-Mayor: >
-Mayor o igual: >=
-Diferente: !=
-(*)Estrictamente diferente de: !==

*Comparan además el tipo de variable:
“5”==5 : true.
“5”===5 : false.

Lógicos

-AND: &&
-OR: ||

Título I: Variables

Para declarar variables, especificamos la palabra reservada **var**, seguido del nombre de la variable. Esta variable la podemos inicializar si escribimos un igual = y el valor que le queremos dar. JavaScript al igual que python, es el compilador el que indica el tipo de variable de cada variable.
var nombre;
var nombre = valor;

Podemos declarar varias variables en la misma fila:
var nombre, nombre2, nombre3;
O declararlas e inicializarlas en la misma fila:
var nombre = "Juan", edad = 15, verdad = false;

Tipos de datos

Existen tres tipos de datos:

- Numéricos: Tanto enteros como decimales.
- Strings: Cadenas de texto, los cuales deben de ir dentro de comillas dobles o simples.
- Booleans: 2 valores: **true** o **false**.

Variables globales

Si dentro de una función declaramos una variable sin indicar **var** , esta variable será de ámbito global, por lo que podremos usarla en otras funciones fuera de la función en la que la hemos declarado.

pi

Número pi: **Math.PI** .
(PI es una constante estática de la clase Math igual a 3.1415...).

typeof(v)

Devuelve qué tipo de variable es la variable **v** .

isNan(n)

Devuelve **true** si la variable **n** no es un número.
Si es un número, **false**.
***NaN** = Not-a-Number.

Acortar parte decimal / toFixed(n)

Podemos limitar el número de caracteres decimales de un número decimal en JavaScript con la ayuda del método **toFixed(n)** , donde **n** es el número de decimales que se mostrarán:

```
var num = 3.14159;
alert(num.toFixed(2));          //Salida: 3.14
```

toLowerCase() / toUpperCase()

Convierte la cadena de texto sobre la que se ejecuta a minúsculas (**toLowerCase()**) o a mayúsculas (**toUpperCase()**).

Título II: Casings y Math

Los castings sirven para convertir un tipo de variable en otro tipo de variable, sin variar el contenido de esta.

Casting a numérico

Number(valor)
Convierte el valor **valor** a un número (Si no se puede, devuelve **NaN**).

Casing a entero

parseInt(valor)

Convierte el valor **valor** a un número entero (siempre que sea posible).

Math.round(n)

Redondea el número decimal **n**, convirtiéndolo en un entero.

Math.random()

Devuelve un número aleatorio entre 0 y 1.

Título III: Arrays

Los arrays se declaran de forma similar a las variables. Hay dos formas:

1. Declaramos cuantos huecos tendrá el array para luego inicializarlos:

var array = new Array(n);
array[0] = valor;
...

(Donde n es el número de elementos que este puede contener.)

2. Declaramos e inicializamos el array en la misma línea:

var array = [valor, valor, ...];
var array = new Array(valor, valor, ...);

*Los arrays en JavaScript pueden tener diferentes tipos de datos en un mismo array:

var array = [“Juan”, 18, false, “Marta”, 21, true];

Los arrays en JavaScript son dinámicos, por lo que podemos seguir añadiendo elementos después de especificar su tamaño, eliminar elementos con **pop() o **shift()** reduciendo así su tamaño o incluso añadir un elemento en una posición mayor:

var array = new Array(4);
array[100] = 86;

Así, se han creado huecos desde la posición 3 de creación del array hasta la 100, donde hemos introducido el valor 86.
Los huecos intermedios tendrán el valor **undefined**.

***Se puede crear un array vacío: **var array = [];**

Longitud

Podemos saber la longitud de un array gracias a la propiedad **.length** de los arrays.

push(elemento)

Agrega el elemento **elemento** al final del array sobre el que se ejecuta.
*También permite agregar multitud de elementos a la vez:
array.push(valor, valor, ...);

unshift(elemento)

Agrega el elemento **elemento** al principio del array sobre el que se ejecuta.
*También permite agregar multitud de elementos a la vez:
array.unshift(valor, valor, ...);

pop()

Elimina el último elemento del array sobre el que se ejecuta.

shift()

Elimina el primer elemento del array sobre el que se ejecuta, desplazando todos los elementos un hueco hacia delante.

Título IV: Condicionales

if

if(condición){

```
        código;
}else if(condición){
        código;
}else{
        código;
}
```

(Igual que Java).
*No hace falta indicar siempre un **else if** y un **else**.
Se pueden añadir tantos **else if como se desee.
***Se puede especificar una variable booleana sola como condición. Si esta es **true** , la condición será **true** y viceversa.

switch

```
switch(variable){
        case valor:
                código;
                break;
        case valor:
                código;
                break;
        …
____default:
                código;
}
```

(Igual que Java).
*Se pueden especificar tantos case como se desee.
No es necesario indicar siempre un **default.
***Los default no llevan break.

Título V: Bucles

for

for(var i=valor; condiciónConI; cambiarI){
código;
}

for(var i=0; i<10; i++){}

while

while(condición){
código
}
(Igual que en Java).

do while

do{
código;
}while(condición);
(Igual que en Java).

Bucle infinito en diferente hilo / setInterval(funcion, delay)

Podemos crear un bucle infinito (para al cerrar el programa) que ejecute una función repetidas veces especificando un delay entre llamada y llamada (esto sin que detenga el hilo en el que se encuentra).
Esto lo hacemos ejecutando la función:
setInterval(funcion, delay) , a la que le especificamos la función que se ejecutará varias veces y los milisegundos de delay:

var bucle = setInterval(escribir, 1000);
La función escribir será llamada infinitamente cada 1000 milisegundos = 1 segundo.

clearInterval(intervalo)

Elimina el **setInterval()** que almacenamos en la variable **intervalo** :

```
var bucle;
function escribir(){alert("Hola"); clearInterval(bucle);}
bucle = setInterval(escribir, 1000);
```

Dentro de un segundo, se escribirá Hola una única vez ya que hemos detenido el **setInterval()** .

Título VI: Funciones

```
function nombre(){
        código;
}
```

Llamar a función

```
nombre();
```

*Si la función no recibe parámetros, no es necesario indicar paréntesis:

```
nombre;
```

Funciones que reciben parámetros

```
function nombre(param1, param2, ...){
        código;
}
```

*No se debe especificar el tipo de variable, ya que las variable funcionan de forma similar a python.
**Se pueden especificar tantos parámetros como se desee.

Funciones que devuelven parámetros

Una función puede (o no) devolver parámetros. Esto lo hace con una instrucción **return** . Ej:

```
function suma(a, b){
```

```
    return a+b;
}
```

Funciones anónimas

Podemos crear funciones anónimas para, por ejemplo, pasarlas como parámetros de un método.
Para crear una función anónima, creamos una función normal en una línea (separando las diferentes líneas de código por puntos y comas) y no le damos ningún nombre. Ej:

```
function () {código;}
boton.addEventListener("click", function(){parrafo.width=50;}, false);
```

Título VII: Creación de JSON

JSON es un formato de texto sencillo para el intercambio de datos. Se trata de un subconjunto de la notación literal de objetos de JavaScript, aunque, debido a su amplia adopción como alternativa a XML, se considera un formato independiente del lenguaje.

Para crear un objeto JSON, tenemos que crear un objeto JavaScript, indicando entre corchetes un formato de **clave:valor** y separar estos formatos por comas. Dentro de un objeto JSON podemos indicar arrays (estructura similar al JSON pero con corchetes) u otros objetos JSON:

```
json = {
    "nombre" : "Juan",
    "edad" : 18,
    "apellido" : "García",
    "hermanos" : true,
    "coches" : ["Renault", "Nissan"],
    "casa" : {"duplex" : false, "piso" : true}
}
```

Acceder a JSON

Para leer un objeto JSON , primero tenemos que convertirlo a un objeto JavaScript con la función **JSON.parse(json)** y luego simplemente indicar el objeto JSON y seguir la nomenclatura del punto con sus valores:

(la función recibe un archivo json):

```
function leer(json){
        var obj = JSON.parse(json);

        obj.nombre;          //Juan
        obj.hermanos;        //true
        obj.coches[1];       //Nissan
        obj.casa.duplex;     //false
}
```

Capítulo II: Prototipos y POO

JavaScript, a diferencia de otros lenguajes, es un lenguaje orientado a prototipos u objetos de notación literal. Estos se diferencian de otros objetos tradicionales en lenguajes como Java o C + + por no tener métodos, sólo campos de clase.
La sintaxis de los objetos de notación literal es idéntica a la de los objetos JSON. Con la única diferencia de que estos últimos son utilizados para transmitir datos entre aplicaciones.
A partir de un prototipo podemos crear un objeto JSON (ver Capítulo I, Título VIII).

Título I: Creación de un prototipo

Para crear un prototipo, creamos una variable y la igualamos a unos corchetes que contendrán una serie de campos de clase con un formato de **clave: valor**, separando cada campo de clase por comas.

var pedro = {edad: 16, apellidos: “Martínez”, mascotas: true};

Arrays en prototipos

Dentro de un prototipo podemos incluir arreglos:
var juan = {edad: 18, coches: [“Toyota”, “Renault”]};

Prototipos en prototipos

Dentro de un prototipo podemos incluir a su vez otro prototipo:
var familiaGomez = {padre: {nombre: ”Juan”}, hijo: {nombre: “Paco”}};

Funciones en prototipos

Podemos añadir funciones dentro de prototipos para usarlos como si fueran métodos. Para esto, igualar una clave de un campo de un

prototipo a una función anónima. Esta función podrá acceder a campos de clase de su propio prototipo:

```
var prototipo = {
	edad: 15,
	incrementar: function(paso){
		this.edad+=paso;
		return this.edad
	}
};
```

Título II: Lectura de un prototipo

Para leer los elementos de un prototipo, simplemente usar la nomenclatura del punto:

```
alert(pedro.edad);		//16
alert(pedro.apellidos);	//Martínez
alert(pedro.mascotas);	//trur
```

Lectura de arrays dentro de prototipos

Para leer los elementos que se encuentran dentro de un array, usamos la nomenclatura del punto junto con la de los arrays:

```
alert(juan.coches[1]);		//Renault
```

Lectura de prototipos dentro de prototipos

Para leer los elementos de los prototipos que se encuentran dentro de otros prototipos, usamos la nomenclatura del punto:

```
alert(familiaGomez.padre.nombre);	//Juan
```

Llamar a funciones dentro de prototipos

Para llamar a funciones dentro de prototipos, simplemente tenemos que especificar su clave y especificar parámetros siempre y cuando esta reciba alguno:

```
alert(prototipo.incrementar(1));
```

Título III: Creación de una clase

JavaScript no dispone verdaderamente de clases, sino de prototipos con una máscara externa haciendo que parezcan clases.
Esta versión se implementó en 2015.
Para crear una clase, especificamos la palabra reservada **class** seguida del nombre de la clase y de unos corchetes.:

```
class Coche{

}
```

Instanciación de una clase

Para instanciar una clase, simplemente tendremos que crear una variable e igualarla a la palabra reservada **new** seguida del nombre de la clase en cuestión y unos paréntesis del método constructor.
*Especificar parámetros en los paréntesis si el método constructor recibe alguno.

```
c = new Coche();
```

Título IV: Creación de campos de clase

Para crear un campo de clase, simplemente indicar su nombre.
Aunque esto no es del todo necesario, ya que podemos instanciar campos de clase en el constructor y en los métodos.

```
class Coche{
    width;
}
```

this

Para acceder a un campo de clase de la misma clase en la que estamos desde el método constructor o cualquier otro método, tenemos que usar el objeto **this** seguido de un punto y el campo de clase en

cuestión. Si no especificamos **this** estaremos accediendo a una variable privada del método.
this.width;

Campos privados

Para crear campos de clase privados, tenemos que especificar un hashtag delante del campo en cuestión. Sólo podremos acceder a esta variable desde dentro de la clase y especificando el hashtag. Para acceder a ella desde fuera de la clase tendremos que usar SETTERS y GETTERS.

```
class Coche{
        #width;             //(*)

        alertar(){
                this.#width = 500;
        }
}

c = new Coche();
c.alertar();
c.#width;           //undefined
```

*Para crear un campo de clase privado, a diferencia de los normales nos es obligatorio especificarlo en el cuerpo de la clase.

Título V: Creación de método constructor

Para crear un método constructor, creamos una función dentro de la clase sin especificar la palabra reservada **function** y cuyo nombre sea "constructor".
El método constructor se utiliza para inicializar campos de clase.

```
class Coche{
        constructor(marca){
```

```
            this.marca = marca;
        }
}
```

*JavaScript no permite la sobrecarga de métodos constructores.
JavaScript cuenta con un constructor por defecto que no recibe parámetros e inicializa todos los campos de clase en **undefined.

Título VI: Creación de métodos

Para crear métodos en clases, simplemente tenemos que crear funciones sin especificar la palabra **function**.

```
class Coche{
        width = 100;
        aumentar(paso){
            this.width += paso;
        }
}
c = new Coche();
c.aumentar(10);
```

Título VII: SETTERS y GETTERS

Los SETTERS y los GETTERS funcionan de forma diferente en JavaScript en comparación con otros lenguajes. (Podemos decir que estos funcionan de forma parecida a las properties de C#).
Los SETTERS y GETTERS sólo pueden acceder a campos de clase privados (ver Título IV, Campos privados).

```
class Coche{
        constructor(){
            this.#ruedas = 4;
        }
}
```

SETTERS

Para crear un SETTER, debemos crear una función que tenga el mismo nombre que el campo privado al que accedemos (sin el hashtag) y especificar delante suya la palabra **set** . Además, este debe recibir un parámetro. Dentro del setter especificaremos cómo se igualará al campo de clase privado (a elección de cada uno):

```
class Coche{
        constructor(){this.#ruedas = 4;}
        set ruedas(ruedas){
                this.#ruedas = ruedas;
        }
}
```

GETTERS

Para crear un GETTER, debemos crear una función que tenga el mismo nombre que el campo privado al que accedemos (sin el hashtag) y especificar delante suya la palabra **get** . Dentro del getter especificaremos cómo se iguala al campo de clase privado.

```
class Coche{
        constructor(){this.#ruedas = 4;}
        set ruedas(ruedas){this.#ruedas = ruedas;}
        get ruedas(){
                return this.#ruedas;
        }
}
```

Uso de SETTERS y GETTERS

Para usar los setters y los getters, simplemente tenemos que acceder al campo de clase como si este fuera público aunque lo que de verdad estamos haciendo es acceder a los setters y getters del objeto:

```
c = new Coche();
c.ruedas = 6;           //Accediendo al setter
alert(c.ruedas);        //Accediendo al getter
```