**JAVASCRIPT**



DE CERO A LOS DETALLES

|  |  |
| --- | --- |
| Resultado de imagen de udemy logo | CURSO UDEMY  Link: <https://www.udemy.com/javascript-de-cero-hasta-los-detalles>  **ÁLVARO MORÁN GARCÍA** |

# Introducción

## Introducción al lenguaje

Inicialmente se llama LiveScript este lenguaje. Posteriormente se cambió a JavaScript debido al apogeo de Java en la época, para que el nombre resultase similar, estrategia que en su día funcionó.

Las similitudes entre ambos lenguajes no son tales; la sintaxis es solo muy superficialmente parecida y el objetivo de ambas es muy diferente, siendo el objetivo de ambos idiomas muy diferentes.

Los exploradores web tenían un intérprete de código Javascript interno, siendo capaces de ejecutar código JavaScript de lado del cliente. NodeJS también nos permite ejecutar JavaScript del lado del servidor, aunque en este curso nos centraremos en el lado de cliente.

JavaScript tiene sus limitaciones:

* No podemos conocer la estructura (ni modificarla) de datos del servidor.
* No puede ser utilizado para aplicaciones de red por sí solo.
* No tiene capacidad multihilo o múltiples procesos simultáneos. Sí tiene algunas herramientas para encolar distintos procedimientos en un stack pero no tiene una capacidad multihilo real.

## Primer programa

Para realizar programas en JavaScript solo necesitaremos:

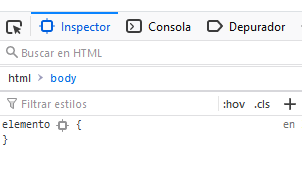
* Un explorador web (Chrome, preferiblemente) que nos permita ejecutar el código.
* Un editor de texto (será Microsoft Visual Studio).

Comenzamos con la creación de un fichero index.html (este archivo se llama siempre así debido a una convención muy arraigada que hace que los servidores comiencen por buscar un archivo que tenga este nombre).

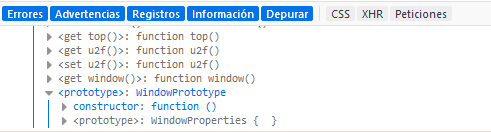
|  |
| --- |
| <html>  <head>  <meta charset="UTF-8">  <title>JavaScript</title>  </head>  <body>  <script type="text/JavaScript">  </script>  </body>  </html> |

Este código HTML básico lo que le dice es al intérprete (que será Chrome) que en el cuerpo de la página tendrá que ejecutar código JavaScript.

Si abrimos el archivo que hemos creado con el explorador web y ejecutamos las herramientas de administración, veremos que tenemos el siguiente contenido:



Esto quiere decir que el explorador, al ver que el cuerpo tiene código JavaScript que interpretar, ha creado un objeto padre vacío para alojar todo lo necesario. Si en la consola del explorador escribimos “window”, nos dará detalles sobre este objeto, como los siguientes, que serán detalles propios de un objeto en JavaScript:



Podemos modificar el cuerpo de nuestro script de la siguiente forma para añadir un par de variables:

<body>

<script type="text/javascript">

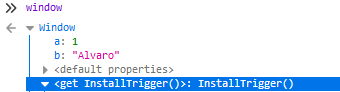
var a = 1;

var b = "Alvaro";

</script>

</body>

Si ahora analizamos el objeto window en el explorador web vemos lo siguiente, debajo de este objeto window principal (es solo window cuando ejecutamos el código en cliente; si estamos usando NodeJS en un servidor el objeto será diferente):



Con lo que vemos que el explorador web ha sido capaz de interpretar nuestras variables correctamente.

Estos scripts deberán estar siempre situados en el body. Podemos poner el script en cualquier lugar no obstante (en el header, entre el header y el body…) pero lo más lógico y la convención índica que deberemos usarlo en el body.

## Consideraciones sobre el objeto window

Como hemos visto al analizar el objeto window en el apartado anterior, existen muchas más variables y valores ya inicializados dentro del objeto window de antemano. Si no tenemos cuidado y llamamos a alguna de nuestras variables con el mismo nombre de alguna ya existente, la sobreescribiremos. Por lo tanto, no es recomendable escribir código directamente sobre el objeto principal window. Lo ideal será encapsularlo en otras regiones.

JavaScript nos permite además redefinir un mismo valor que ya ha sido definido con anterioridad, cambiar su valor etc. sin ningún tipo de control, con lo que deberemos tener cuidado con esa forma de proceder.

## Comprendiendo la consola

Vamos a cambiar nuestro archivo index.html por el siguiente cuerpo:

<body>

<script src="app.js"></script>

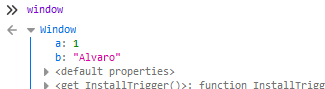
</body>

Hemos especificado una nueva fuente para nuestro script, que será un archivo llamado app.js que crearemos junto a nuestro index.html. Su contenido será el siguiente:

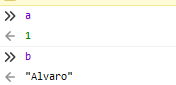
var a = 1;

var b = "Alvaro";

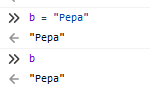
Si escribimos ese código, seguiremos modificando el objeto window principal. Es más, si tenemos muchos archivos, se pisarán entre ellos todas las variables.



Podemos usar la consola para buscar variables y su valor. Si en vez de window escribimos a o b, automáticamente la consola nos imprimirá el contenido de dichas variables:



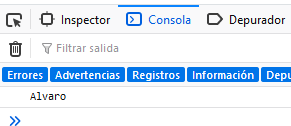
Desde la consola podemos crear nuevas variables o modificar las ya existentes. Por ejemplo, podemos hacer lo siguiente:



Si queremos que se nos escriba en tiempo real el valor de alguna variable por consola, podemos introducir en nuestro código lo siguiente:

console.log(b);

Entonces, al abrir la página web en el explorador directamente se nos escribirá lo siguiente:



Existen muchos tipos de logs diferentes que podemos escribir por consola para ayudarnos a la depuración de código:

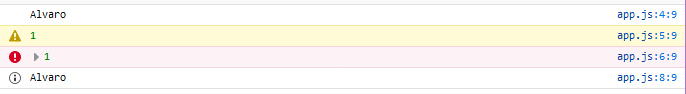
console.log(b);

console.warn(a);

console.error(a);

console.info(b);

Que tendrán la siguiente representación por consola:



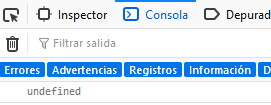
## Undefined

Supongamos que dentro de nuestro archivo app.js definimos nuestro código JavaScript tal que así:

console.log(b);

var b = "Alvaro";

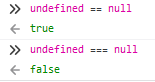
Lo que sucederá entonces es que, si observamos la consola veremos como el valor impreso de b es undefined, dado que en el momento de la impresión su valor todavía no ha sido definido:



JavaScript es un lenguaje interpretado, pero en contexto. Esto es, cuando el explorador empieza a analizar el archivo JavaScript hace un barrido de arriba abajo secuencialmente. Al leer la declaración de una variable le reserva posición en memoria desde el inicio, por ejemplo, para b. Pero no le almacena un valor hasta que en tiempo de ejecución no llega a la línea donde se realiza la asignación de su valor.

Esto hace que, **desde el arranque** de una ejecución de JavaScript todas las variables sean utilizables desde el inicio (aunque valgan undefined, existir ya existen) pero la definición de su valor se realiza en el momento en el que la ejecución del código llega al punto en que se asigna su valor.

Es importante tener en cuenta que **undefined** no es lo mismo que **null**. Más adelante veremos las diferencias, pero basta saber que se trata de valores de objetos o entidades diferentes (su valor es el mismo, pero si comparamos el objeto en sí mediante el triple igual, el resultado será que son diferentes):



## JavaScript no es asíncrono

JavaScript no es asíncrono, aunque gracias a su pila de llamadas a funciones intenta simular un comportamiento asíncrono. En realidad, JavaScript encola las llamadas a las funciones en una pila que va ejecutando de forma síncrona.

Podemos ejemplificarlo creando un bucle que intentemos romper de forma asíncrona, para ver qué sucede. Para ello, dentro de nuestro app.js escribiremos una función que realice muchas repeticiones y la ejecutamos:

function imprimir() {

for (var i = 0; i < 8000; i++) {

console.log("Imprimo");

}

}

imprimir();

Para poder cortar su ejecución, definimos en el index.html un botón que llamará a otra función diferente, tal que así:

<body>

<script src="app.js"></script>

<button onclick="presionoClick()">Boton</button>

</body>

Es indiferente poner el botón encima o debajo de la llamada al script, ya que antes de la ejecución todo el código se lee entero por el intérprete, como hemos visto en la sección anterior.

Ahora, en el contenido de app.js escribimos esta otra función:

function presionoClick() {

console.log("Click en boton");

}

Si ejecutamos en el explorador web y mientras el contador del bucle está subiendo pulsamos el botón, veremos que el log de click en el botón no se escribe hasta que el bucle no termina:



Esto sucede así por que el comportamiento de JavaScript no es asíncrono; tiene encoladas las 8000 peticiones del bucle y, aunque mientras las está ejecutando pulsemos el botón, esta pulsación cae en la pila o stack de funciones por realizar (al final de la misma) y se ejecuta en el momento en que han terminado las otras 8000 iteraciones del bucle.

## Orden de las importaciones

Vamos a ilustrar el problema de definir las mismas variables en distintos archivos. Supongamos que tenemos dos archivos, que son referenciados desde nuestro index.html de la siguiente manera:

<body>

<script src="app.js"></script>

<script src="app2.js"></script>

</body>

Cuyo contenido es el que sigue:

* app.js:

var nombre = "Pepe";

* app2.js:

console.log(nombre);

Al ejecutar este código veremos lo siguiente por consola:



Sin embargo, si cambiamos el orden de la importación de scripts en el index.html de la siguiente forma:

<body>

<script src="app2.js"></script>

<script src="app.js"></script>

</body>

El resultado es el que sigue:



Esto es porque cuando lee el archivo HTML no crea primero un contexto con todo el contenido, como sí que vimos que sucedía cuando lee código JavaScript (y guardaba espacio para las variables previamente a su uso). En lo que es sección de código HTML, lee lo que sucede secuencialmente y, en este caso, leerá el contenido primero de app2.js e intentará acceder a la referencia a “nombre”, que no existe todavía en el objeto global por que no ha sido todavía leído por el HTML.

Más adelante se enseñarán buenas prácticas de código para evitar estos problemas.

# Variables, objetos, Typeof y condicionales