**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:

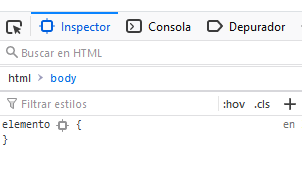
* Un explorador web (Chrome, preferiblemente) que nos permita ejecutar el código.
* Un editor de texto (será Microsoft Visual Studio).

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).

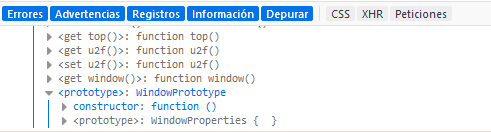
|  |
| --- |
| <html>  <head>  <meta charset="UTF-8">  <title>JavaScript</title>  </head>  <body>  <script type="text/JavaScript">  </script>  </body>  </html> |

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">

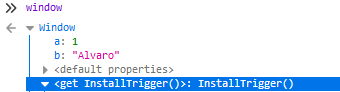
var a = 1;

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>

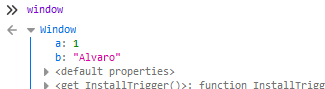
</body>

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:

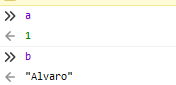
var a = 1;

var b = "Alvaro";

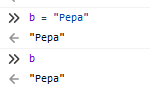
Si escribimos ese código, seguiremos modificando el objeto window principal. Es más, si tenemos muchos archivos, se pisarán entre ellos todas las variables.



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:



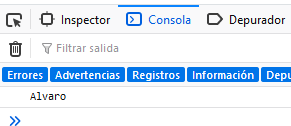
Desde la consola podemos crear nuevas variables o modificar las ya existentes. Por ejemplo, podemos hacer lo siguiente:



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:

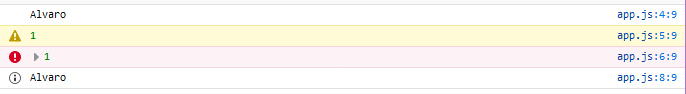
console.log(b);

console.warn(a);

console.error(a);

console.info(b);

Que tendrán la siguiente representación por consola:



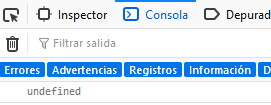
## Undefined

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

console.log(b);

var b = "Alvaro";

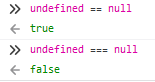
Lo que sucederá entonces es que, si observamos la consola veremos como el valor impreso de b es undefined, dado que en el momento de la impresión su valor todavía no ha sido definido:



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

## Variables explícitas y anónimas

Las variables pueden ser creadas de manera explícita o bien ser usadas de forma anónima. Si las usamos de manera anónima, no les asignamos un nombre de variable en concreto, como en el siguiente caso:

console.log("Juan");

En el caso anterior estamos escribiendo una variable que tiene como contenido ‘Juan’, pero no hemos declarado explícitamente dicha variable. Para usar las variables de forma explícita debemos hacerlo como hemos visto con anterioridad:

var nombre = "Juan";

console.log(nombre);

## Tipos primitivos

En JavaScript existen los objetos y los tipos primitivos, como en otros lenguajes que pueden ser más orientados a objetos (ya veremos que en JavaScript el tipado de los objetos es muy débil).

Tenemos los siguientes tipos primitivos: numero, texto, boolean, indefinido y null:

// Tipos

var num = 10;

var str = "texto";

var bol = true;

var und = undefined;

var nul = null;

Cuando decimos que JavaScript no tiene un tipado fuerte nos podemos fijar en el código anterior; vemos que las variables están definidas como var (variables genéricas), a las que les damos un nombre y les asignamos un valor primitivo cualquiera. En ningún momento estamos especificando el tipo del valor que van a guardar.

Es más, incluso podemos cambiar el tipo de una de estas variables de numero a texto, por ejemplo, sin que la ejecución se resienta de ninguna manera. Por ejemplo:

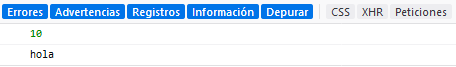
var cambio = 10;

console.log(cambio);

var cambio = "hola";

console.log(cambio);

Nos da el siguiente resultado:



Se trata de una escritura dinámica, donde las variables no están asociadas a un tipado concreto.

## Objetos

Los objetos son una colección de tipos de datos primitivos o bien, de otros objetos. Siempre tiene que respetar la nomenclatura de pares, esto es, todas las variables que introduzcamos tienen que tener su par.

Veamos el siguiente ejemplo de cómo se define un objeto con sus distintas propiedades (variables u objetos internos):

var obj = {

numero: 10,

texto: "Texto"

};

console.log(obj);

Que tendrá su siguiente representación en consola:



Es muy importante hacer notar que las llaves ‘{…}’ sirven para establecer objetos en JavaScript y su contenido.

Si nos fijamos en el objeto window que hemos visto en capítulos anteriores veremos que es una colección de tipos primitivos y objetos que tiene en su interior.

Podemos introducir un objeto dentro de un objeto, como hemos dicho anteriormente, como vemos en el siguiente ejemplo:

var obj = {

numero: 10,

texto: "Texto",

objHijo: {

numero2: 20,

texto2: "Texto2"

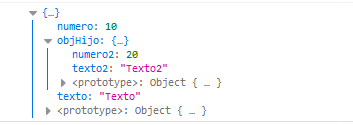
}

};

console.log(obj);

Para los nombres de objetos utilizaremos nomenclatura camel-case.

Si vemos ahora la consola observamos lo siguiente (atención al orden en que muestra las propiedades, es alfabético):



## Valor y referencia

En esta sección veremos cómo y cuando hacer que variables pasen valores de una a otra mediante su valor o la referencia (de forma similar a como sucede con otros lenguajes de programación).

Supongamos el siguiente código:

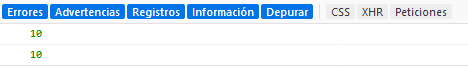
var a = 10;

var b = a;

console.log(a);

console.log(b);

El resultado por consola es el esperado, que ambas variables tendrán el valor 10:



Podemos intentar cambiar ahora el valor de a y ver lo que sucede:

var a = 10;

var b = a;

console.log(a);

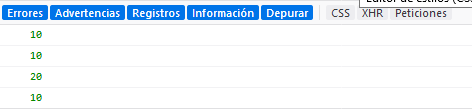
console.log(b);

a = 20;

console.log(a);

console.log(b);

En este caso el resultado por consola es:



Lo que ha sucedido en este caso es que cuando imprimimos a por segunda vez, le hemos cambiado el valor. Sin embargo, b mantiene el valor del a inicial por que en el momento en que le dimos este valor a la variable b lo hicimos por valor, le pasamos el valor de a en ese momento del tiempo. Al fin y al cabo, es normal; al declarar b hemos asignado un espacio de memoria distinto al de a, y le hemos dado un valor en un momento dado. Por lo tanto, su espacio en memoria es distinto y sus valores son distintos, aunque hayan coincidido en ese momento del tiempo.

Veremos lo que pasa con objetos. Supongamos que tenemos el siguiente código:

var c = {

nombre: "Juana"

};

var d = c;

console.log(c);

console.log(d);

En este caso, el resultado es el siguiente:



No obstante, si cambiamos ahora el nombre de la variable c y volvemos a imprimir por consola el resultado, obtenemos lo siguiente:

var c = {

nombre: "Juana"

};

var d = c;

console.log(c);

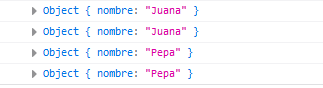
console.log(d);

c.nombre = "Pepa";

console.log(c);

console.log(d);

Con resultado:



Vemos que en este caso el nombre ha cambiado en ambos objetos. Esto es así porque, a diferencia de cuando igualamos variables primitivas cada una con su propia sección de memoria, cuando igualamos objetos se igualan sus referencias, esto es, su posición en memoria.

Como apuntan a la misma posición de memoria, si cambiamos el valor, este cambio se propaga a todos los demás objetos que hemos declarado apuntando a esa misma porción en memoria, esto es, al mismo objeto. Esto es un paso de valor por referencia. Este concepto es similar a lo que ocurre en otros lenguajes de POO como Java. Hay maneras de hacer clones para que los objetos sean distintos y apunten a distintas posiciones de memoria, pero esto se verá más adelante.

## Notación de . y { }

Hemos visto en la anterior sección como accedíamos a las variables del interior de un objeto mediante la notación de ‘.’. En esta sección analizaremos más en profundidad este comportamiento.

La notación de punto nos permite obtener los valores internos a un objeto. Más en detalle:

var objeto = {

nombre: "Pepe",

edad: 30

};

console.log(objeto.nombre);

console.log(objeto.edad);

El punto nos permite acceder al interior de cada nivel jerárquico. Si tenemos un objeto dentro de un objeto accederíamos de la siguiente forma:

var objeto = {

nombre: "Pepe",

edad: 30,

direccion: {

calle: "Calle",

numero: 46

}

};

console.log(objeto.nombre);

console.log(objeto.edad);

console.log(objeto.direccion.calle);

console.log(objeto.direccion.numero);

También, por supuesto, podemos acceder a todo el objeto dirección si así lo deseamos por las razones que sean. En ese caso nos retornará todo el objeto dirección, y dentro de él sus correspondientes propiedades.

Imaginemos que a mitad de código queremos añadir una nueva variable al objeto, porque se nos haya olvidado o nos interese hacerlo en ese momento por lo que queramos:

var objeto = {

nombre: "Pepe",

edad: 30,

direccion: {

calle: "Calle",

numero: 46

}

};

objeto.direccion.zipcode = 28100;

console.log(objeto.direccion.zipcode);

No hay ningún problema en realizar esto, el código se ejecutará sin ningún problema. Recordemos que esto es posible dado el débil tipado de objetos de JavaScript; podemos seguir modificando el interior de los objetos en tiempo de ejecución sin ningún problema.

Imaginemos que tenemos varios niveles de complejidad, como en el siguiente caso:

var objeto = {

nombre: "Pepe",

edad: 30,

direccion: {

calle: "Calle",

numero: 46,

edificio: {

nombre: "Casoplon",

telefono: "23415141"

}

}

};

Por supuesto, podemos acceder al contenido utilizando la notación de punto que hemos visto, aunque empieza a ser un poco más largo y farragoso:

console.log(objeto.direccion.edificio.nombre);

Hay distintas formas de afrontar este problema. Si, por ejemplo, vamos a trabajar bastante con el objeto edificio, podemos simplificar el tratar con ese objeto creando un nuevo objeto que apunte por referencia al objeto edificio, de la siguiente forma:

var edificio = objeto.direccion.edificio;

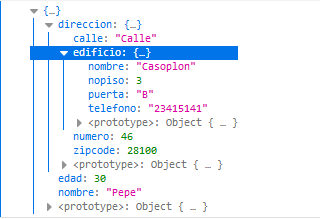
// Operamos de manera directa

edificio.nopiso = 3;

edificio.puerta = "B";

console.log(objeto);

Si ejecutamos el código y vemos en consola lo que ha sucedido, veremos que efectivamente hemos podido realizar modificaciones sobre el interior del objeto principal gracias a esta asignación por referencia.



Respecto a la notación de corchete, se utiliza cuando queremos obtener datos de manera dinámica. Partiendo del objeto que tenemos creado en el ejemplo anterior. Si queremos obtener el nombre de la persona objeto del ejemplo anterior y su calle (que está dentro del objeto interior dirección) se puede hacer lo siguiente:

// Corchetes

console.log(objeto["nombre"] + "," + objeto["direccion"]["calle"]);

Esto permite buscar por nombre de propiedades interiores a un objeto de forma más sucinta. Es ideal para representar qué tipo de información estamos intentando obtener, en lugar de la notación seguida de puntos para obtener dicha información.

Supongamos que intentamos acceder a una propiedad que no existe, de la siguiente forma:

var field = "edad2";

console.log(objeto[field]);

El resultado de esto es undefined. Esta forma anterior de acceder al contenido de los objetos es muy útil para tener centralizado (y dinamizado) el nombre de las variables a las que queremos acceder, de forma que sirva para organizar mucho mejor el código.

## Funciones

Las funciones en JavaScript son muy similares en sintaxis a las funciones definidas en otros lenguajes de programación. Sin embargo, tienen una serie de utilidades que iremos viendo que las hacen especiales respecto a otros lenguajes en general.

Las funciones se declaran utilizando camel-case, igual que cuando declaramos objetos. Veamos un ejemplo de cómo se declara una función:

function primeraFuncion() {

var a = 20;

console.log(a);

}

Hay varias cosas interesantes aquí. Primeramente, utilizamos llaves ‘{..}’ para definir el contenido del objeto. Antes habíamos dicho que esto está reservado a declaración de objetos. Tomemos nota de ello.

Si el script tiene solamente este contenido, no se ejecutará nada; aquí lo que hemos hecho es definir el contenido de la función. Cuando el intérprete lee este archivo JavaScript, prevé que tiene que reservar espacio en memoria para una variable a y que hay una definición de una función, pero no la ejecuta. Para llamar a la función tenemos que llamarla desde algún lugar del código JavaScript:

primeraFuncion();

Entonces sí se ejecutará la función.

¿Cómo afecta la ejecución de una función al resto del código? Supongamos que tenemos código con una variable ‘a’ definida múltiples veces tal que así:

var a = 30;

console.log(a);

function primeraFuncion() {

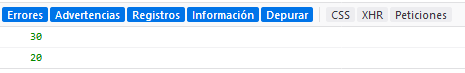
var a = 20;

console.log(a);

}

primeraFuncion();

El resultado será el siguiente:



Lo que sucede es que la ejecución, como sabemos, se realiza de forma secuencia. De forma que, cuando imprimimos primeramente el valor de a tiene el valor inicial 30. Posteriormente, al final del código, llamamos a la función y entra a la misma y se ejecuta ese bloque de código interior, cambiando el valor y volviendo a imprimirlo, ahora con su nuevo valor 20.

Supongamos que tenemos el siguiente código:

var a = 30;

console.log(a);

function primeraFuncion() {

//var a = 20;

console.log(a);

}

primeraFuncion();

En este código, la variable ‘a’ no está definida dentro de la función. Sin embargo, termina por encontrar su valor de la siguiente forma:

* Primeramente, busca dentro del contexto de la función si existe la variable.
* De no haberlo encontrado, busca dentro en el contexto global la variable (window), que es dónde la ha encontrado.

Adicionalmente, las funciones en JavaScript siempre devuelven algo. En este caso no hemos especificado valor de vuelta, en este caso el valor que devuelve es undefined. Pero las funciones en JavaScript siempre retornan un valor, aunque no sea definido. Se puede comprobar fácilmente mediante el siguiente código:

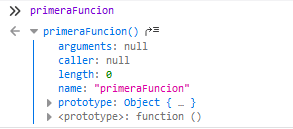
var retorno = primeraFuncion();

console.log(retorno);

Con el siguiente resultado por consola:



Las funciones, además, las podemos buscar por consola como si fueran objetos, y obtendremos una serie de detalles de las mismas:

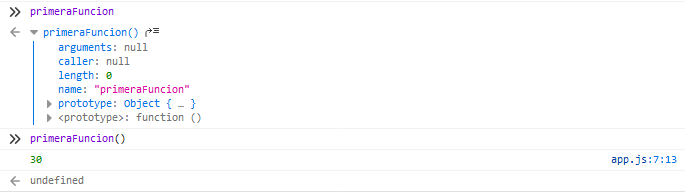


Además, podemos llamar a las funciones antes de que estén definidas, esto es, en una parte anterior del código; esto es posible por lo que ya hemos comentado de que el intérprete primero lee todo el código antes de ejecutarlo (para almacenar valores en memoria etc), por lo que es consciente de que la función existe y tiene cuerpo, aunque aparezca en una parte inferior del código.

Podemos llamar a las funciones de dos maneras distintas, una de las cuales ya la hemos visto en la imagen anterior:

* Sin los paréntesis: Esto es, hacemos una asignación de la función. Ya dejamos entrever con anterioridad que las funciones se comportan como objetos y es así; cuando hacemos una asignación de función sin poner paréntesis, JavaScript asume que lo que estamos haciendo es asignar esta función a una variable, variable que ahora se comportará como si fuera dicha función:
* Con los paréntesis: En este caso lo que hacemos es invocar el comportamiento de la función, especificando sus argumentos de entrada.

Esto se ilustra perfectamente si lo ejecutamos por consola, donde vemos que si no ponemos los paréntesis se nos está devolviendo el objeto función, mientras que si lo ejecutamos con los paréntesis sin argumentos (en este caso), vemos como se ejecuta su contenido (escribe el valor de a) y retorna el valor indefinido:



## Funciones con parámetros

Para utilizar una función con argumentos de entrada:

function imprimir (nombre) {

console.log(nombre);

}

imprimir("Juan");

Con varios argumentos, realizamos la llamada separando los argumentos por comas:

function imprimir (nombre, apellido) {

console.log(nombre + " " + apellido);

}

imprimir("Juan", "Perez");

Si al realizar la llamada no especificamos el segundo argumento, la ejecución del código no lanzará un error; el intérprete ya ha reservado un hueco para esa variable y tiene un valor undefined. Si no la especificamos o utilizamos, su resultado será undefined, pero el código no morirá:

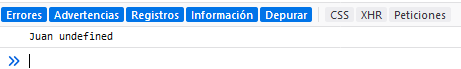
function imprimir(nombre, apellido) {

console.log(nombre + " " + apellido);

}

imprimir("Juan");

Su resultado sería:



Una buena práctica es colocar todas las funciones al comienzo de los archivos, para mantener el código ordenado.

Una buena práctica para comprobar que los argumentos de entrada están definidos correctamente, comprobando que no sean iguales al objeto indefinido:

function imprimir(nombre, apellido) {

if (apellido === undefined) {

apellido = "XXXX";

}

console.log(nombre + " " + apellido);

}

imprimir("Juan");

Una forma de simplificar este condicional es mediante la siguiente cláusula, que evaluará si una variable está como indefinida y, de no ser así, la utilizará:

apellido = apellido || "XXXX";

Las funciones pueden recibir objetos completos, no solo variables primitivas. Veamos el siguiente ejemplo donde enviamos un objeto anónimo como argumento de entrada:

function imprimirOjbeto(persona) {

console.log(persona.nombre + " " + persona.apellido);

}

imprimirObjeto({

nombre: "Juan",

apellido: "Perez"

});

Si utilizamos objetos explícitos el caso sería similar:

function imprimirOjbeto(persona) {

console.log(persona.nombre + " " + persona.apellido);

persona.nombre = "Maria";

}

var obj = {

nombre: "Juan",

apellido: "Perez"

};

imprimirObjeto(obj);

console.log(obj.nombre);

En este caso, además, dentro de la función le hemos cambiado el nombre al objeto de entrada. Recordemos que los objetos funcionan por acceso a su referencia en memoria, por tanto, si dentro del método cambiamos el parámetro nombre asociado al objeto, se cambiará para todo el objeto y todos los otros objetos que estén referenciados a él.

También podemos hacer algo más exótico; ya hemos visto que las funciones se pueden comportar como objetos en JavaScript. Podemos enviar una función por parámetro (para ello se suele utilizar por convenio la notación fn, se utiliza mucho en jQuery, por ejemplo) a una función (en este caso una función anónima, ya que una función es un objeto también):

function reciboFuncion(fn) {

fn();

}

reciboFuncion(function() {

console.log("Funcion anonima");

});

Como hemos visto con anterioridad, las funciones se ejecutan cuando escribimos su nombre y luego ponemos los paréntesis con los argumentos. En el caso anterior, declaramos una función de manera anónima como argumento de entrada a nuestra función principal, dentro de la cual le asignamos el nombre fn. Posteriormente la ejecutamos mediante fn().

Podemos hacerlo de igual forma sin utilizar funciones anónimas:

function reciboFuncionExplicita(fn) {

fn();

}

var miFuncion = function() {

console.log("miFuncion");

}

reciboFuncionExplicita(miFuncion);

Con todo esto debemos ser muy ordenados, porque si no realizamos una buena técnica de codificación podemos montar un lio de funciones tremendo. No obstante, si lo hacemos bien, da la flexibilidad necesaria para que el código sea muy legible.

## Retorno de funciones

En este apartado veremos como hacer que una función devuelva un valor de retorno. Los valores de retorno admitidos son:

* Primitivos
* Objetos
* Funciones
* Nulos
* Indefinidos

La palabra reservada para devolver un valor de una función es ‘return’.

Veamos el siguiente código de ejemplo de una función donde el objeto Math (declarado por defecto dentro de window) nos devuelva un número aleatorio:

function obtenerAleatorio() {

return Math.random();

}

function obtenerNombre() {

return "Juan";

}

console.log(obtenerAleatorio() + 10);

var nombre = obtenerNombre();

console.log(nombre + "Padillo");

Como vemos en el caso anterior, simplemente dentro de la función haremos un return con el valor que deseamos que sea devuelto. Este valor lo podemos asignar a una variable y utilizarla en consecuencia.

Las funciones pueden llamar a otras funciones. Veamos el siguiente ejemplo con una función que retorna true o false en función de su el valor aleatorio devuelto por otra función es mayor o menor que cierto valor:

function esMayorMitad() {

if (obtenerAleatorio() > 0.5) {

return true;

} else {

return false;

}

}

console.log(esMayorMitad());

En el siguiente ejemplo veremos como devolver un objeto anónimo:

En el caso anterior, hemos llamado a las variables dentro del objeto de igual forma que el argumento de entrada de la función. JavaScript en este caso no se va a confundir porque sabe que los objetos son colecciones de pares-valor, por lo que sabe que el primer ‘nombre’ está solo dentro del objeto y no es el argumento de entrada de la función.

function crearPersona(nombre, apellido) {

return {

nombre: nombre,

apellido: apellido

};

}

var persona = crearPersona("Maria", "Pax");

Podemos ver, adicionalmente, funciones que devuelvan funciones, como en el siguiente ejemplo:

function crearFuncion() {

return function() {

console.log("Funcion creada");

};

}

var funcion = crearFuncion();

funcion();

Esto puede ser todo lo complejo que queramos; esta función que retornamos puede devolver a su vez otra función. Imaginemos el siguiente caso:

function funcionEnFuncion() {

return function(nombreF) {

console.log("Me creo: " + nombreF);

return function() {

console.log("Segunda funcion");

};

};

}

var primeraFuncion = funcionEnFuncion();

var segundaFuncion = primeraFuncion("Pepe");

segundaFuncion();

## Funciones de primera clase

Hemos visto en anteriores capítulos que JavaScript, al tratar los objetos como elementos muy débilmente tipados, podemos modificar los mismos declarativamente en cualquier momento del código y, así, añadirles variables y lo que necesitemos.

Hemos visto también que JavaScript considera las funciones como objetos. ¿Qué sucede, entonces, si intentamos modificar una función e introducirle una variable nombre como si fuera un objeto, tal que así?:

function a () {

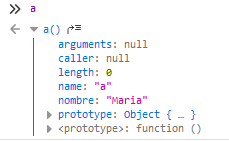
console.log("Funcion a");

}

a();

a.nombre = "Maria";

Si lo ejecutamos e intentamos imprimir en consola el valor de ‘a’ nos devuelve lo siguiente:



Efectivamente, podemos ver como la función adopta una nueva variable interna, tal como hemos descrito. ¿Qué sucede si llamamos a esta variable name? Porque, como vemos en la captura anterior, el objeto función ‘a’ ya tiene una variable name. Veámoslo:

function a() {

console.log("Funcion a");

}

a();

a.name = "Maria";

Si por consola deseamos obtener la variable name, vemos lo siguiente:



La variable name es una de las variables protegidas de la clase función, que no se pueden cambiar una vez ésta ha sido creada. Prácticamente cualquier otra propiedad de la función puede ser cambiada (o añadida), menos su nombre. Esto implica que podemos transformar propiedades en lo que deseamos: funciones, objetos complejos o sencillos, etc. con total flexibilidad.