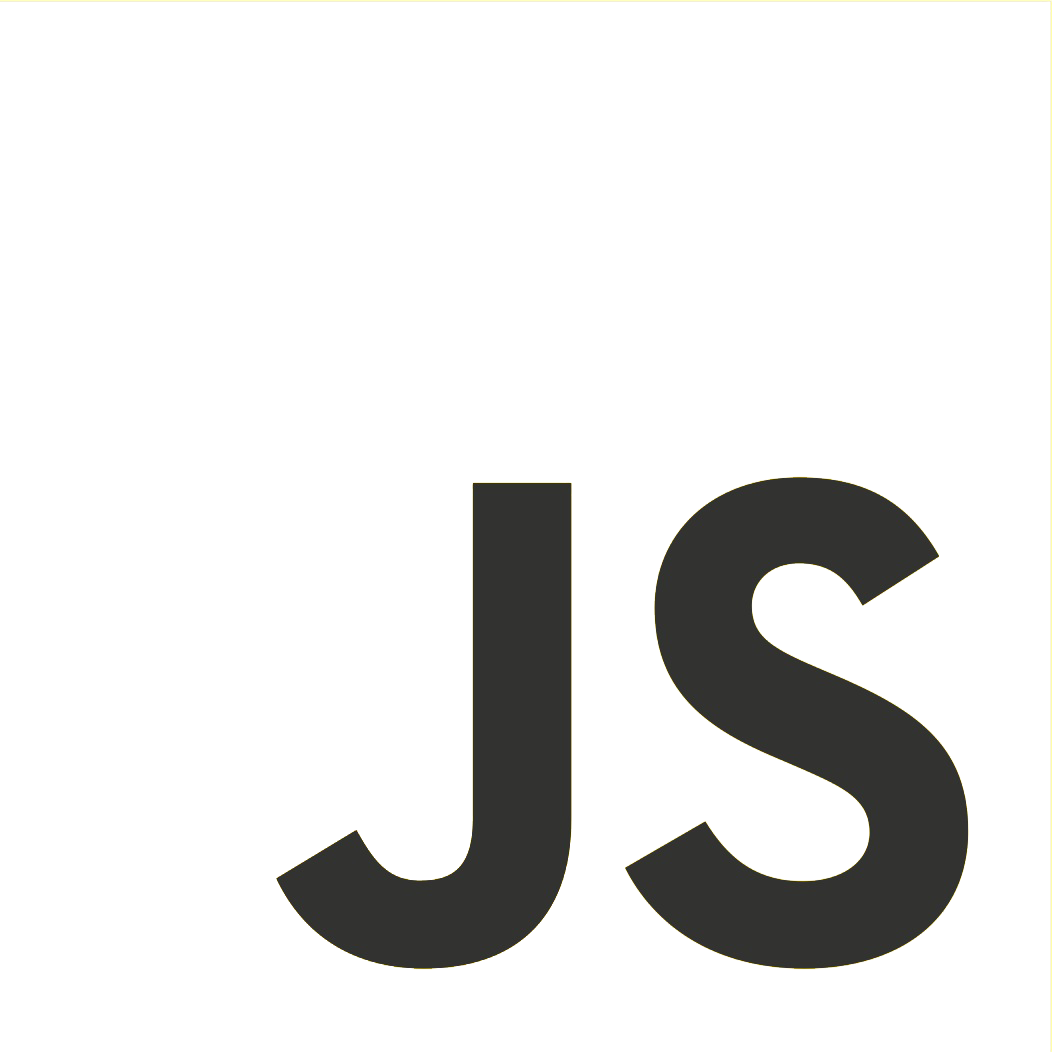
*Lanford Gabriel Murillo*

Resumen final JavaScript intermedio



**¿Qué es JavaScript?**

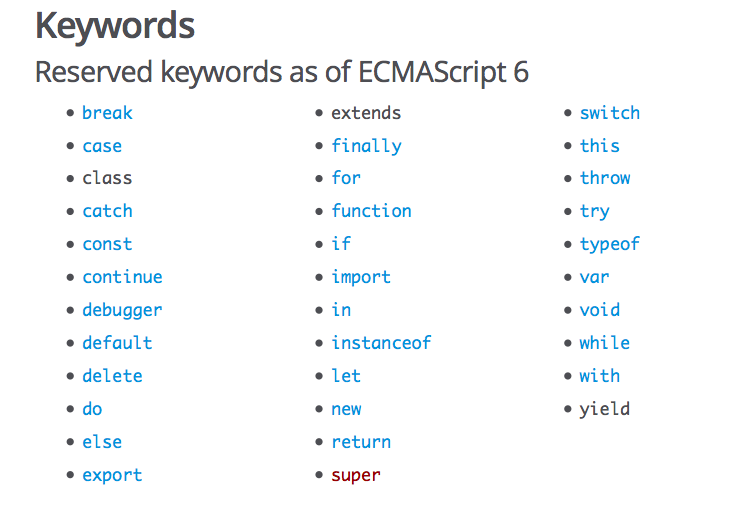
JavaScript es un lenguaje de scripting multiplataforma, orientado a objetos. JavaScript es un pequeño y liviano lenguaje; no es útil como lenguaje independiente, pero está diseñado para ser fácilmente embebido en otros productos y aplicaciones, como ser web browsers. Dentro de un entorno de desarrollo, JavaScript puede ser conectado a los objetos de este entorno y proveer un control programático sobre ellos.

**JavaScript y Java**

Javascript y Java son similares en algunos puntos, pero fundamentalmente diferentes en otros. El lenguaje Javascript se parece al de Java pero no tiene el tipado estático y comprobación de tipos fuerte. Javascript tiene sintaxis, convenciones de nombres y controles básicos de flujo parecidos a los de Java,  por eso es que se le cambio el nombre de Livescript a JavaScript.

En contraste con el sistema de clases construidas por declaraciones en tiempo de compilación de Java, Javascript soporta un sistema en tiempo de ejecución basado en un pequeño número de tipos de datos que representan valores numéricos, booleanos, y cadenas. Javascript tiene un modelo de objetos basado en prototipos en lugar del más común modelo de objetos basado en clases. El modelo basado en prototipos provee herencia dinámica; esto es, lo que se hereda puede variar para objetos individuales. Javascript también soporta funciones sin ningún requerimiento declarativo especial. Las funciones pueden ser propiedades de objetos, ejecutándose como métodos débilmente tipados.



 **Buenas prácticas de JS**

* Evita usar variables globales.
* Siempre declara variables locales.
* Nunca declares: números, strings o booleans como objetos.
* No uses: new Object()
* Usa {} en lugar de new Object().
* Usa “” en lugar de new String.
* Usa 0 en lugar de new Number().
* Usa false en lugar de new Boolean().
* Usa [] en lugar de new Array().
* Usa /()/ en lugar de new RegExp().
* Usa function (){} en lugar de new function().
* Usa el comparador de tipo estricto “===”.
* Evita utilizar eval(). Esta función puede ejecutar texto como código y representa problemas de seguridad.
* Undefined no es lo mismo que null:
* En JS, **null** es para objetos, mientras que **undefined** es para variables.
* Trata de no usar **with**, ya que tiene un efecto negativo en la velocidad de procesamiento y crea un desorden en el scope de las funciones.

**Funciones en JavaScript**

Las funciones en JS son objetos, y los objetos a su vez, son una colección de pares name/value, los cuales tienen un enlace oculto hacia un objeto prototype*.*

A diferencia de los objetos, que son enlazados a **object.prototype**, las funciones son enlazadas a **function.prototype** (el cual a su vez está enlazado a **object.prototype**). Adicionalmente, cada función es creada con 2 propiedades ocultas más: El contexto de la función, y el código que implementa el comportamiento de la función. Dado que las funciones son objetos, se pueden utilizar como cualquier otro valor.

Las funciones pueden ser almacenadas en variables, objetos y arrays, pueden pasar como argumentos a otras funciones, pueden ser devueltas por funciones. Además, puesto que las funciones son objetos, pueden tener métodos. Y una de las características más sobresalientes de las funciones, es que pueden ser invocadas de diferentes maneras.

Las funciones constan de 4 partes:

* La palabra reservada **function.**
* El nombre de la función. (El nombre de la función puede servir para: llamarse a sí misma recursivamente; los depuradores y herramientas de desarrollo lo utilizan para identificar a la función. Y si la función no tiene nombre, se dice que es una función ***anónima***).
* Los parámetros de la función. (Envueltos por paréntesis. Dentro de los paréntesis, pueden haber 0 o más parámetros separados por comas, estos nombres serán definidos como variables en la función).
* conjunto de sentencias envueltos entre llaves. Estas declaraciones son el cuerpo de la función y se ejecutan cuando se invoca la misma.

**Patrones de invocación de funciones**

Invocar una función suspende la ejecución de la función actual, pasando el control y parámetros a la nueva función. Además de los parámetros declarados, cada función recibe dos parámetros adicionales: **this** y **arguments**. El parámetro **this** es muy importante en la programación orientada a objetos, y su valor se determina por el patrón de invocación. Hay cuatro patrones de invocación en JavaScript:

* El patrón de invocación por **método**.
* El patrón de invocación de **función**.
* El patrón de invocación de **constructor**.
* El patrón de invocación **apply**.

Los patrones difieren la forma en la que el parámetro **this** es inicializado.

El operador de invocación consiste en un par de paréntesis que siguen a cualquier expresión que produce un valor de función. Los paréntesis pueden contener cero o más expresiones, separadas por comas. Cada expresión produce un valor de argumento. Cada uno de las valores de los argumentos serán asignados a los nombres de parámetros de la función. Si hay demasiados argumentos, los adicionales serán ignorados. Si hay muy pocos, el valor **undefined** será sustituido para los valores perdidos. No hay ninguna comprobación de tipos en los argumentos, cualquier tipo de valor puede pasarse a cualquier parámetro.

**Invocación por método**

Cuando una función es almacenada como propiedad de un objeto, lo llamamos: **método**. Cuando un método es invocado, dicho método está ligado al objeto.

Un método puede usar **this** para acceder al objeto recuperar los valores del objeto o modificarlo. La unión de este con el objeto pasa en el **tiempo de invocación**. Esta unión tardía hace a las funciones que utilizan **this** altamente reusables. Métodos que consiguen su contexto del objeto de **this**, son llamados métodos públicos.

**Invocación de función**

Cuando una función no es propiedad de un objeto, es invocada solamente como función.

Cuando la función es invocada de esta forma, **this** es ligado al objeto global. Esto es el resultado de un error en el diseño del lenguaje.

**(OOP) Object Oriented Programming**

**Clase:** Definiciones de las propiedades y comportamiento de un tipo de objeto concreto. La instanciación es la lectura de estas definiciones y la creación de un objeto a partir de ella.

**Herencia:** Es la facilidad mediante la cual una clase “X” hereda en ella cada uno de los atributos y operaciones de otra clase “Y”.

**Objetos:** Instancia de una clase. Entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos), los mismos que consecuentemente reaccionan a eventos. Se corresponden con los objetos reales del mundo que nos rodea, o con objetos internos del sistema (del programa). Es una instancia a una clase.

* **Estado:** Compuesto por datos o informaciones; serán uno o varios atributos a los que se habrán asignado unos valores concretos (datos).
* **Comportamiento:** Está definido por los **métodos o mensajes** a los que sabe responder dicho objeto, es decir, qué operaciones se pueden realizar con él.
* **Identidad:** Es una propiedad de un objeto que lo diferencia del resto; dicho con otras palabras, es su identificador (concepto análogo al de identificador de una variable o una constante).

Un objeto contiene toda la información que permite definirlo e identificarlo frente a otros objetos pertenecientes a otras clases e incluso frente a objetos de una misma clase, al poder tener valores bien diferenciados en sus atributos.

**Método:** Algoritmo asociadoaun objeto (o a una clase de objetos), cuya ejecución se desencadena tras la recepción de un “mensaje”. Desde el punto de vista del comportamiento, es lo que el objeto puede hacer. Un método puede producir un cambio en las propiedades del objeto, o la generación de un “evento” con un nuevo mensaje para otro objeto del sistema.

**Evento:** Es un suceso en el sistema (tal como una interacción del usuario con la máquina, o un mensaje enviado por un objeto).

El sistema maneja el evento enviando el mensaje adecuado al objeto pertinente. También se puede definir como evento la reacción que puede desencadenar un objeto; es decir, la acción que genera.

**Atributos:** Son las características que tiene la clase.

**Mensaje:** Una comunicación dirigida a un objeto que le ordena que ejecute uno de sus métodos con ciertos parámetros asociados al evento que lo generó.

**Propiedad o atributo:** Contenedor de un tipo de datos asociados a un objeto (o una clase de objetos), que hace los datos visibles desde fuera del objeto y esto se define como sus características predeterminadas, y cuyo valor puede ser alterado por la ejecución de algún método.

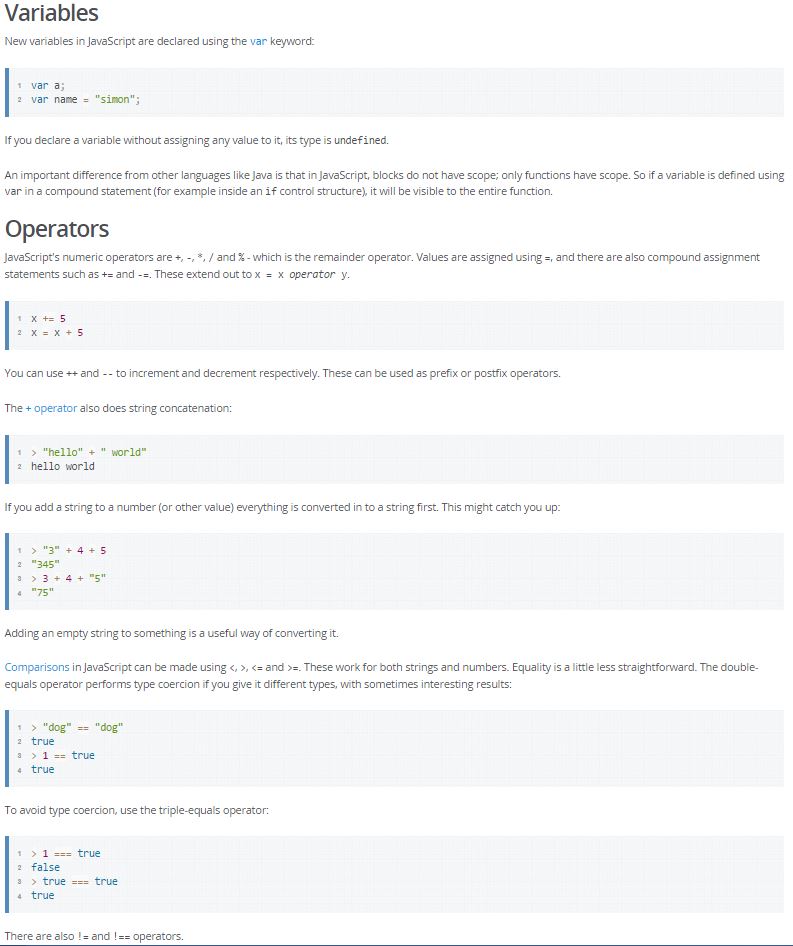
**Estado interno:** Variable que se declara privada, que puede ser únicamente accedida y alterada por un método del objeto, y que se utiliza para indicar distintas situaciones posibles para el objeto (o clase de objetos). No es visible al programador que maneja una instancia de la clase.

**Componentes de un objeto:** Atributos, identidad, relaciones y métodos.

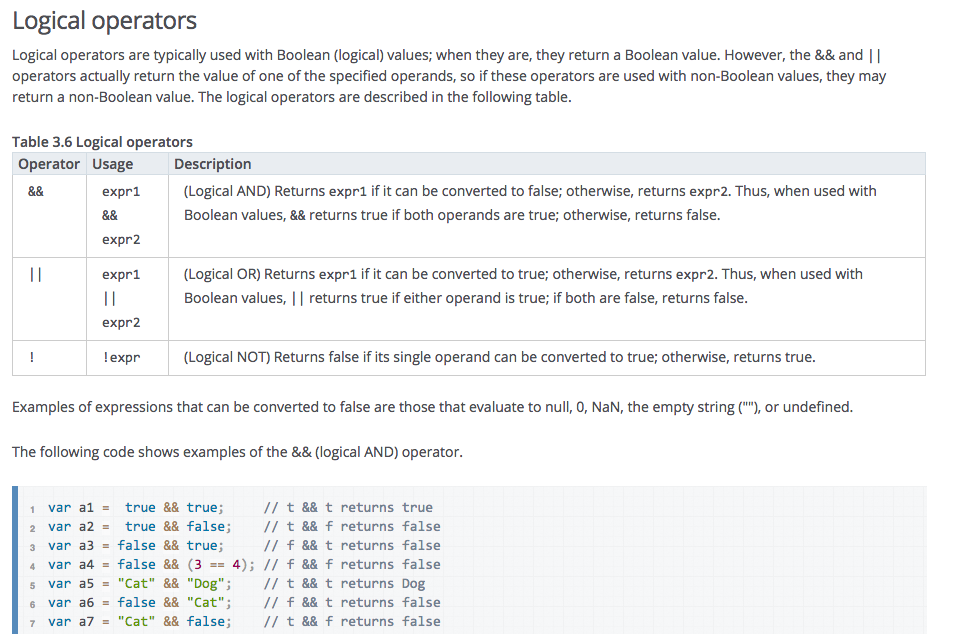
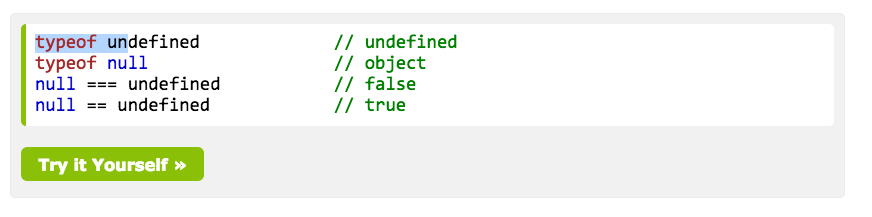
**Identificación de un objeto:** Un objeto se representa por medio de una tabla o entidad que esté compuesta por sus atributos y funciones correspondientes.

**Que es V8?** 

Es un motor OpenSource para JS creado por Google, siendo su programador jefe Lars Bak. Está escrito en C++ e Implementa ECMAScript como norma específica ECMA-262 5.







**Tipos de expresiones de JS**

**Arithmetic:** Para números, ejemplo: 3.14159.

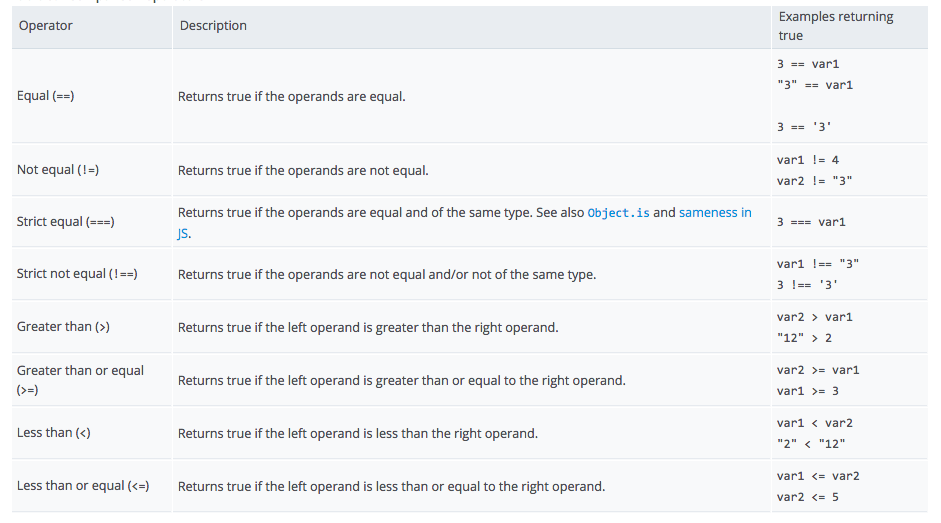
**String:** Para texto, ejemplo: “fred”, “12546”. (Siempre entre comillas).

**Logical:** Para booleanos. True o false.

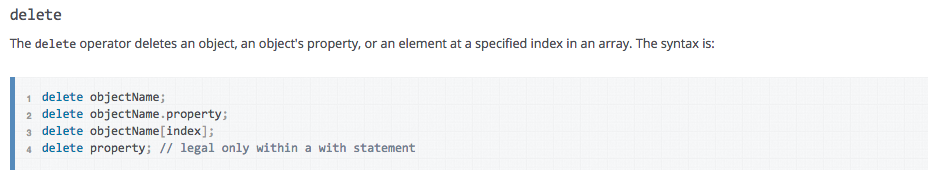
**Object:** Para objetos.

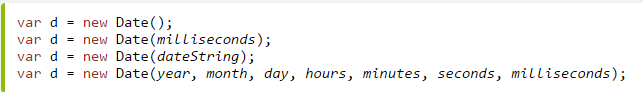


**Comparadores de JS:**



**Operador Delete:** 



**Objeto Date:**

**Server-side & Client-side programming:**

El desarrollo web es todo acerca de la comunicación. En este caso, la comunicación entre dos partes, a través del protocolo HTTP:

El servidor - Esta parte es responsable de servir páginas.

El cliente - Esta parte solicita páginas desde el servidor, y los muestra al usuario. En la mayoría de los casos, el cliente es un navegador web.

El Usuario - El usuario utiliza el cliente con el fin de navegar por la web, rellenar formularios, ver videos en línea, etc.

La programación de cada lado, se refiere al código que se ejecuta en la máquina específica, del servidor o el cliente de.

**Algunos lenguajes de programación comunes en Server-side:**

* PHP
* ASP.Net in C#, C++, or Visual Basic.
* Nearly any language (C++, C#, Java). No están diseñados específicamente para esta tarea, pero son ocasionalmente usados para aplicaciones y servicios a nivel web.

**Algunos lenguajes de programación comunes en Client-side:**

* JavaScript (principalmente)
* HTML\*
* CSS\*
* Cualquier lenguaje que corra en un dispositivo cliente y que interactúe con un servicio remoto es un lenguaje Client-side.
* HTML y CSS no son realmente lenguajes de programación. Estos son formas de sintaxis de marcado por las cuales el cliente renderiza la página para el usuario.

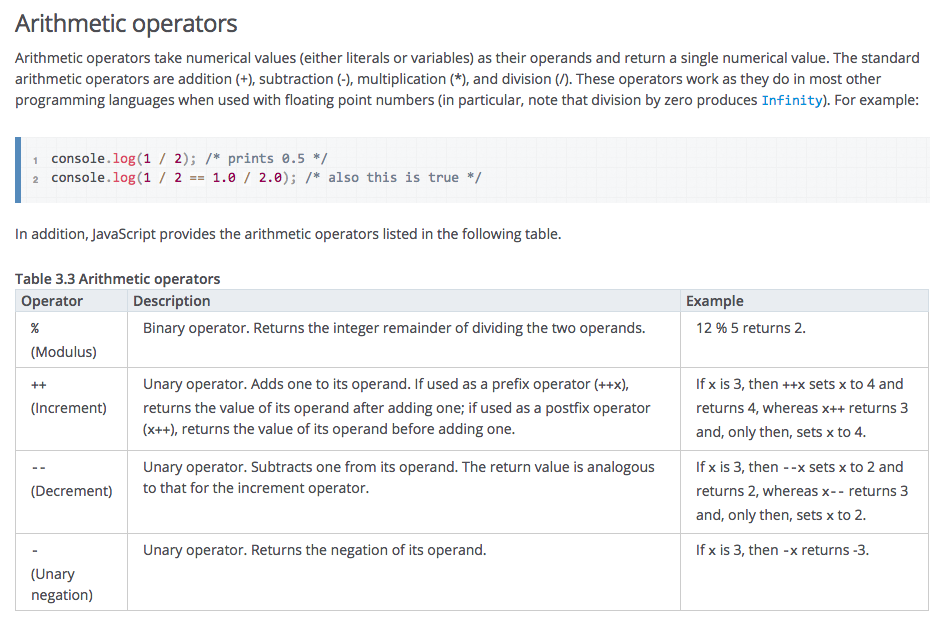
**¿Qué es un Scripting language?**

Un lenguaje scripting es un tipo de lenguaje de programación que es generalmente interpretado en lugar de ser compilado.

Los programas compilados, son convertidos de forma permanente a un código especial antes de que puedan ejecutarse (proceso de compilación). En cambio los scripts permanecen en su forma original (su código fuente en forma de texto) y son interpretados comando por comando cada vez que se ejecutan. De todas maneras, los scripts pueden ser compilados también, aunque no es usual.

Características de los lenguajes scripting

* Los scripts suelen escribirse más fácilmente, pero con un costo sobre su ejecución.
* Suelen implementarse con intérpretes en lugar de compiladores.
* Tienen fuerte comunicación con componentes escritos en otros lenguajes.
* Los scripts suelen ser almacenados como texto sin formato.
* Los códigos suelen ser más pequeños que el equivalente en un lenguaje de programación compilado.



**Precedencia de operadores:**

La precedencia de operadores determina el orden en que se evalúan los operadores. Los operadores con mayor precedencia se evalúan primero.

Un ejemplo común:

3 + 4 \* 5 // returns 23

El operador de multiplicación ("\*") tiene mayor precedencia que el operador de suma ("+") y por lo tanto será evaluado primero.

**Asociatividad**

La asociatividad determina el orden en que se procesan los operadores con la misma precedencia. Por ejemplo, considere la siguiente expresión:

a OP b OP c

Asociatividad izquierda (de izquierda a derecha) significa que se procesa como

(a OP b) c OP, mientras asociatividad derecha (de derecha a izquierda) significa que se interpreta como una OP (b OP c). Los Operadores de asignación son asociativos por la derecha, por lo que puede escribir:

a = b = 5;

Con el resultado esperado que **a** y **b** obtienen el valor 5. Esto se debe a que el operador de asignación devuelve el valor que se le asigna. En primer lugar, **b** se establece en 5. A continuación, la **a** se establece en el valor de **b**.



**Tabla de Precedencia**

Esta tabla está ordenada desde el nivel más alto de precedencia hasta el más bajo.

|  |  |  |  |
| --- | --- | --- | --- |
| Precedence | Operator type | Associativity | Individual operators |
| 19 | [Grouping](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Grouping) | n/a | ( … ) |
| 18 | [Member Access](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Property_Accessors#Dot_notation) | left-to-right | … . … |
| [Computed Member Access](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Property_Accessors#Dot_notation) | left-to-right | … [ … ] |
| [new](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Operators/Special/new) (with argument list) | n/a | new … ( … ) |
| 17 | [Function Call](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions) | left-to-right | … ( *…*) |
| [new](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new) (without argument list) | right-to-left | new … |
| 16 | [Postfix Increment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Increment) | n/a | … ++ |
| [Postfix Decrement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Decrement) | n/a | … -- |
| 15 | [Logical NOT](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_Operators#Logical_NOT) | right-to-left | ! … |
| [Bitwise NOT](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators#Bitwise_NOT) | right-to-left | ~ … |
| [Unary Plus](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Unary_plus) | right-to-left | + … |
| [Unary Negation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Unary_negation) | right-to-left | - … |
| [Prefix Increment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Increment) | right-to-left | ++ … |
| [Prefix Decrement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Decrement) | right-to-left | -- … |
| [typeof](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof) | right-to-left | typeof … |
| [void](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/void) | right-to-left | void … |
| [delete](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete) | right-to-left | delete … |
| 14 | [Multiplication](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Multiplication) | left-to-right | … \* … |
| [Division](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Division) | left-to-right | … / … |
| [Remainder](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Remainder) | left-to-right | … % … |
| 13 | [Addition](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Addition) | left-to-right | … + … |
| [Subtraction](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Subtraction) | left-to-right | … - … |
| 12 | [Bitwise Left Shift](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators) | left-to-right | … << … |
| [Bitwise Right Shift](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators) | left-to-right | … >> … |
| [Bitwise Unsigned Right Shift](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators) | left-to-right | … >>> … |
| 11 | [Less Than](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Less_than_operator) | left-to-right | … < … |
| [Less Than Or Equal](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Less_than__or_equal_operator) | left-to-right | … <= … |
| [Greater Than](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Greater_than_operator) | left-to-right | … > … |
| [Greater Than Or Equal](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Greater_than_or_equal_operator) | left-to-right | … >= … |
| [in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/in) | left-to-right | … in … |
| [instanceof](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/instanceof) | left-to-right | … instanceof … |
| 10 | [Equality](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Equality) | left-to-right | … == … |
| [Inequality](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Inequality) | left-to-right | … != … |
| [Strict Equality](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Identity) | left-to-right | … === … |
| [Strict Inequality](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Nonidentity) | left-to-right | … !== … |
| 9 | [Bitwise AND](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators#Bitwise_AND) | left-to-right | … & … |
| 8 | [Bitwise XOR](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators#Bitwise_XOR) | left-to-right | … ^ … |
| 7 | [Bitwise OR](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators#Bitwise_OR) | left-to-right | … | … |
| 6 | [Logical AND](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_Operators#Logical_AND) | left-to-right | … && … |
| 5 | [Logical OR](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_Operators#Logical_OR) | left-to-right | … || … |
| 4 | [Conditional](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator) | right-to-left | … ? … : … |
| 3 | [Assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment_Operators) | right-to-left | … = … |
| … += … |
| … -= … |
| … \*= … |
| … /= … |
| … %= … |
| … <<= … |
| … >>= … |
| … >>>= … |
| … &= … |
| … ^= … |
| … |= … |
| 2 | [yield](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/yield) | right-to-left | yield … |
| 1 | [Spread](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator) | n/a | ... … |
| 0 | [Comma / Sequence](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comma_Operator) | left-to-right | … , … |

**Hoisting:**  
Hoisting es el comportamiento de JS de mover las declaraciones a la parte de arriba.

En JS, una variable puede ser declarada después de que se ha utilizado.

En otras palabras; una variable puede ser utilizada antes de ser declarada.

Ejemplo:



NOTA: JS solo hace hoist a las declaraciones, no a las inicializaciones.

Consejo: Siempre declara las variables en arriba.

**Método IndexOF()**

El  método **indexOf()** devuelve el índice, dentro del objeto que realiza la llamada, de la primera ocurrencia del valor especificado, comenzando la búsqueda desde la primera posición conocida.

**Use Strict:**

“Use Strict” Define que el código de JS debe ser ejecutado en modo estricto.

* La directiva "use strict" es nueva en JavaScript 1.8.5 (ECMAScript versión 5).
* No es una declaración, sino una expresión literal, ignorada por las versiones anteriores de JS.
* El propósito de "use strict" es para indicar que el código debe ser ejecutado en el "modo estricto".
* Con el modo estricto, no se puede, por ejemplo, utilizar variables no declaradas.

**Errores comunes que se cometen en JS**

**Usar accidentalmente el operador de asignación:**

Los programas de JS tienden a generar resultados inesperados si el programador accidentalmente usa un operador de asignación (=) en lugar de uno de comparación (==).

**Tipo de comparación regular vs estricta:**

En comparación regular, el tipo de data no importa. Este ejemplo retorna true:

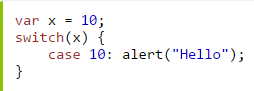
var x = 10;  
var y = "10";  
if (x == y)

En comparación estricta, el tipo de data sí importa. Este ejemplo retorna false:

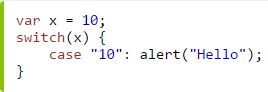
var x = 10;  
var y = "10";  
if (x === y)

**Switch siempre utiliza comparación estricta:**

Este caso nos desplegara una alerta



Este caso no desplegara la alerta



**Confundir adición y concatenación:**

La adición es para sobre agregar números.

La concatenación es para de agregar strings.

En JS, ambas operaciones utilizan el mismo operador **+**.



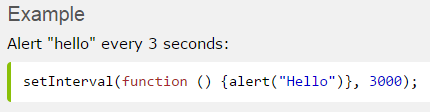
****

**Timing**

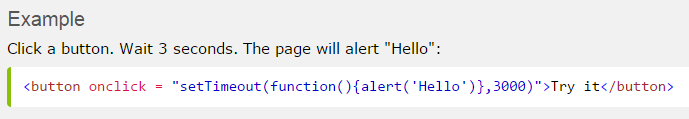
Con JS, es posible ejecutar código en intervalos de tiempo específicos. Esto se conoce como “timing events”.

Es muy simple temporizar eventos en JS, los 2 métodos clave que se usan son los siguientes:

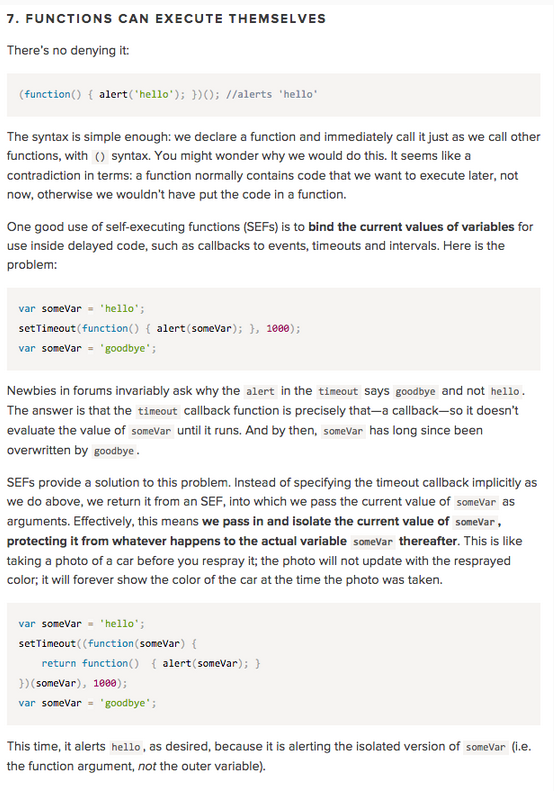
**setinterval()** – Ejecuta una función una y otra vez, en intervalos específicos de tiempo.



**setTimeout()** – Ejecuta la función una vez, después de haber esperado una cantidad específica de milisegundos.



NOTA: El setInterval() y setTimeout() son dos métodos del DOM HTML.



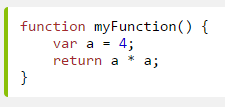
**Closure**

Las variables de JS pueden pertenecer tanto al scope local como al global.

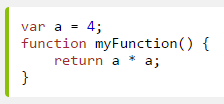
Las variables privadas se pueden crear gracias por medio de closures.

Variables globales:

Una función que puede accesar a todas las variables definidas dentro de la función:



Pero también una función puede acceder a variables definidas fuera de la función, como se muestra a continuación:



En el último ejemplo, **a** es una variable global. En una página web, las variables globales pertenecen al objeto **window**. Las variables globales se pueden usar (y cambiarse) por todos los scripts en la página (y en la ventana). En el primer ejemplo, **a** es una variable local. Una variable local sólo se puede utilizar dentro de la función donde se define. Se oculta de funciones y otro código de scripting. Las variables globales y locales con el mismo nombre son diferentes variables. La modificación de una, no modifica a la otra.

**Tiempo de vida de las variables:**

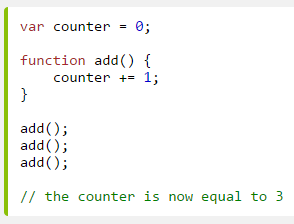
Las variables globales viven tanto como la aplicación (la ventana del app - la página web) vive. Las variables locales tienen vidas cortas. Se crean cuando se invoca la función, y se borran cuando la función ha terminado.



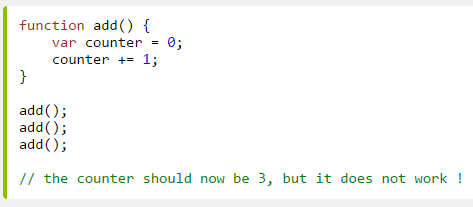


**El dilema del contador:**

Ok. Supongamos que queremos usar una variable para contar algo, y además, queremos que ese contador esté disponible para todas las funciones. Pues podemos usar una variable global, y una función para incrementar el contador pensarás.



El contador debería ser cambiado solamente por la función add(). El problema es que, cualquier script en la página puede cambiar el contador, sin llamar a add().



Pero no funcionó! Cada vez que llamo a la función add(), el contador está seteado a 1.

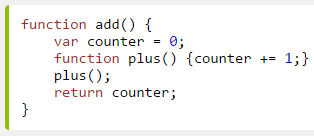
Una función interna de JS puede resolver esto.

**Funciones anidadas de JS**

Todas las funciones tienen acceso al scope global. De hecho, en JS todas las funciones tienen acceso al scope encima de ellas.

JS soporta funciones anidadas. Las funciones anidadas tienen acceso al scope que etá sobre ellas. De manera simple. Tienen acceso a la función que las contiene.

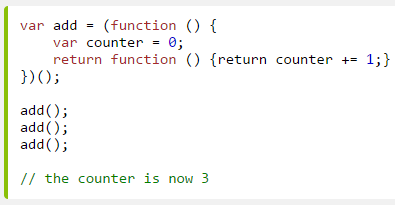
En este ejemplo, la función interna **plus()**, tiene acceso a la variable **counter** en la función padre



Esto nos puede resolver el dilema del contador, porque si podemos alcanzar la función **plus()** desde afuera. Pero también necesitamos una forma de ejecutar **counter = 0** solo una vez. Así que necesitamos un **closure**.

**Closures de JS**

En este ejemplo a la variable **add()** se le asigna el valor de una función de auto-invocación.



La función de auto-invocación se ejecuta una vez. Setea el contador a “0”, y retorna una expresión de función. De esta forma add se convierte en una función. La parte “maravillosa” es que puede acceder al contador en el scope padre. Esto se llama  **closure**. Hacer posible que una función tenga variables “privadas”.

El contador es protegido por una función anónima, y solo puede ser cambiado usando la función add.

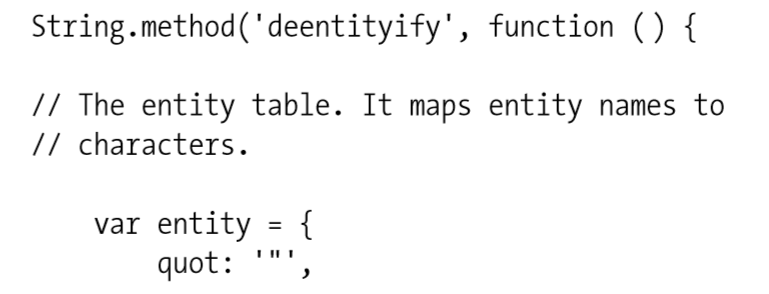


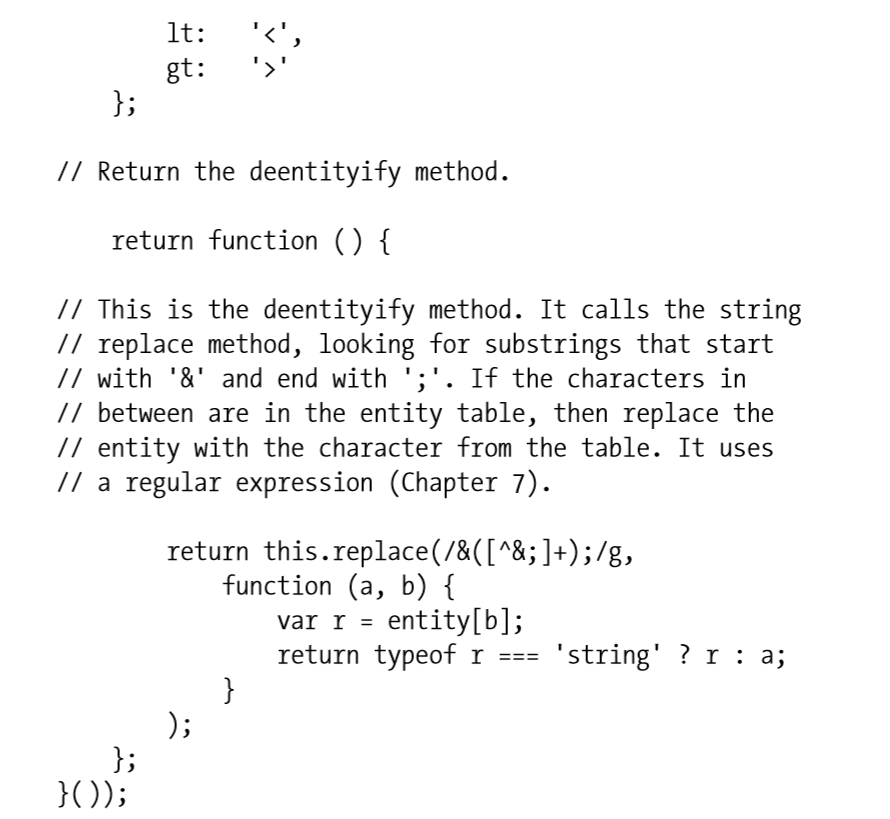
**Module**

Podemos utilizar funciones y closures para crear módulos.

Un módulo es una función u objeto que representa una interface pero a la vez esconde su estado e implementación. Usando una función para producir módulos, podemos eliminar casi por completo el uso de variables globales, y así mitigar una de las peores características de JS.

Por ejemplo, supongamos que queremos aumentar una string con un método **deentityify**. Su trabajo es buscar entidades HTML en una string y reemplazarlas con sus equivalentes. Tiene sentido mantener los nombres de las entidades y sus equivalentes en un objeto. Pero donde deberíamos mantener el objeto? Deberíamos colocarlo en una variable global, pero como las variables globales no son buenas, entonces debemos ponerla en un closure:





Si ponemos atención a la última línea del ejemplo, podemos ver que se invoca la función inmediatamente que acabamos de hacer con el operador **()**. Esta invocación crea y retorna la función que se cconvierte en el método **deentityify**.



El patrón de módulo toma ventaja del scope de la función y el closure para crear relaciones de unión que son privadas. En este ejemplo, solo el método **deentityify** tiene acceso a estructura de datos de la entidad.

**Exceptions**

JS, como otros muchos lenguajes de programación, incluye soporte nativo para el manejo de excepciones.

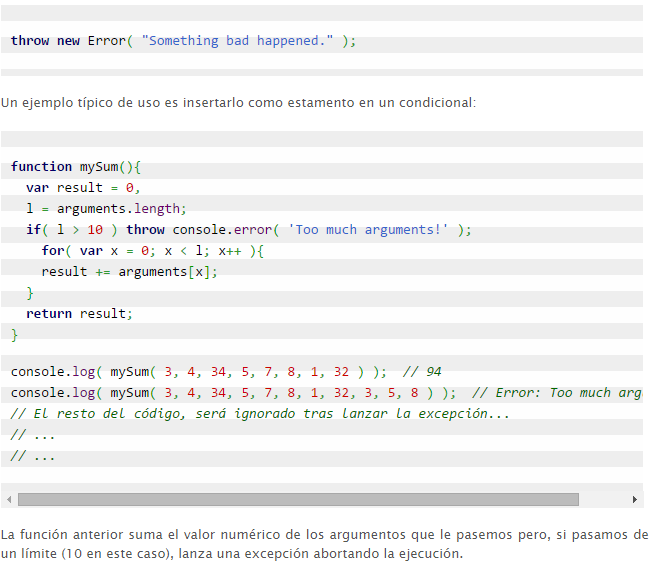
Las excepciones, son imprevistos que ocurren durante la ejecución de un programa; anormalidades que impiden o alteran el comportamiento o flujo normal de un software.

Su función, es separar el código para el manejo de errores de la lógica de aplicación del programa. En aquellos lenguajes que incluyen soporte para el manejo de excepciones, al producirse la anomalía, se desciende en la pila de ejecución hasta encontrar un manejador para la excepción, el cual toma el control en ese momento de la aplicación.

En JS, el manejo de errores puede resultar más necesario que en otros lenguajes debido a la dificultad natural de este para testear aplicaciones. Por lo general, los errores en JS son crípticos y poco informativos, especialmente en Internet Explorer. Esto hace interesante el contar con operadores que nos permitan lanzar nuestros propios mensajes advirtiendo al usuario de que algo ha ido mal.

**Excepciones con Throw:**

La forma más sencilla para lanzar errores es utilizando **throw**. Este comando permite enviar al navegador un evento similar al que se produce cuando ocurre algún imprevisto o nos encontramos ante un tipo inesperado de datos. El lenguaje permite enviar todo tipo de elementos, incluyendo texto, números, valores booleanos o incluso objetos. Sin embargo, la opción más usual es enviar el objeto nativo Error:

Ejemplo de throw

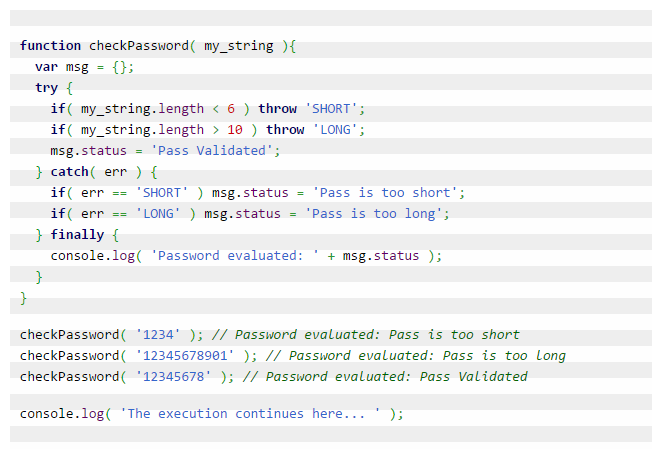


**Excepciones con Try / Catch**

Try… Catch corresponde a un tipo de estructura de control JS con la que comprobar el flujo de un programa frente a comportamientos inesperados. La diferencia entre esta estructura y la anterior es que mientras throw detiene completamente la ejecución, catch realiza una acción determinada frente a los errores para proseguir después con el flujo definido.

Este tipo de excepciones se estructuran mediante un bloque de código que evalúa una condición previa y propone en consecuencia una ejecución predefinida y otra alternativa frente a anormalidades. La sintaxis es sencilla y actúa como un condicional más:

Ejemplo de Try/Catch





Fuentes externas de información:

<http://www.etnassoft.com/2011/01/30/excepciones-en-javascript/>