

# JavaScript Desmitificado

Introducción

東京' TOKIOTA

 Microsoft  
Gold Partner

# Presentación

JS

# Presentación

¿Quién soy?

JS

```
var me = {  
    name: 'Roberto Huertas',  
    twitter: '@Newton_W',  
    worksAt: '@tokiota_IT',  
    job: 'developer'  
};
```

Sobre lo que vamos a hablar (si nos da tiempo):

1. Introducción
2. JavaScript en el HTML
3. Base del lenguaje
4. Tipos Primitivos y de Referencia
5. Funciones
6. Objetos

Herramientas que vamos a necesitar:

- Chrome DevTools

<https://developer.chrome.com/devtools>

- Firebug 3.0

<http://getfirebug.com/releases/firebug/3.0/>

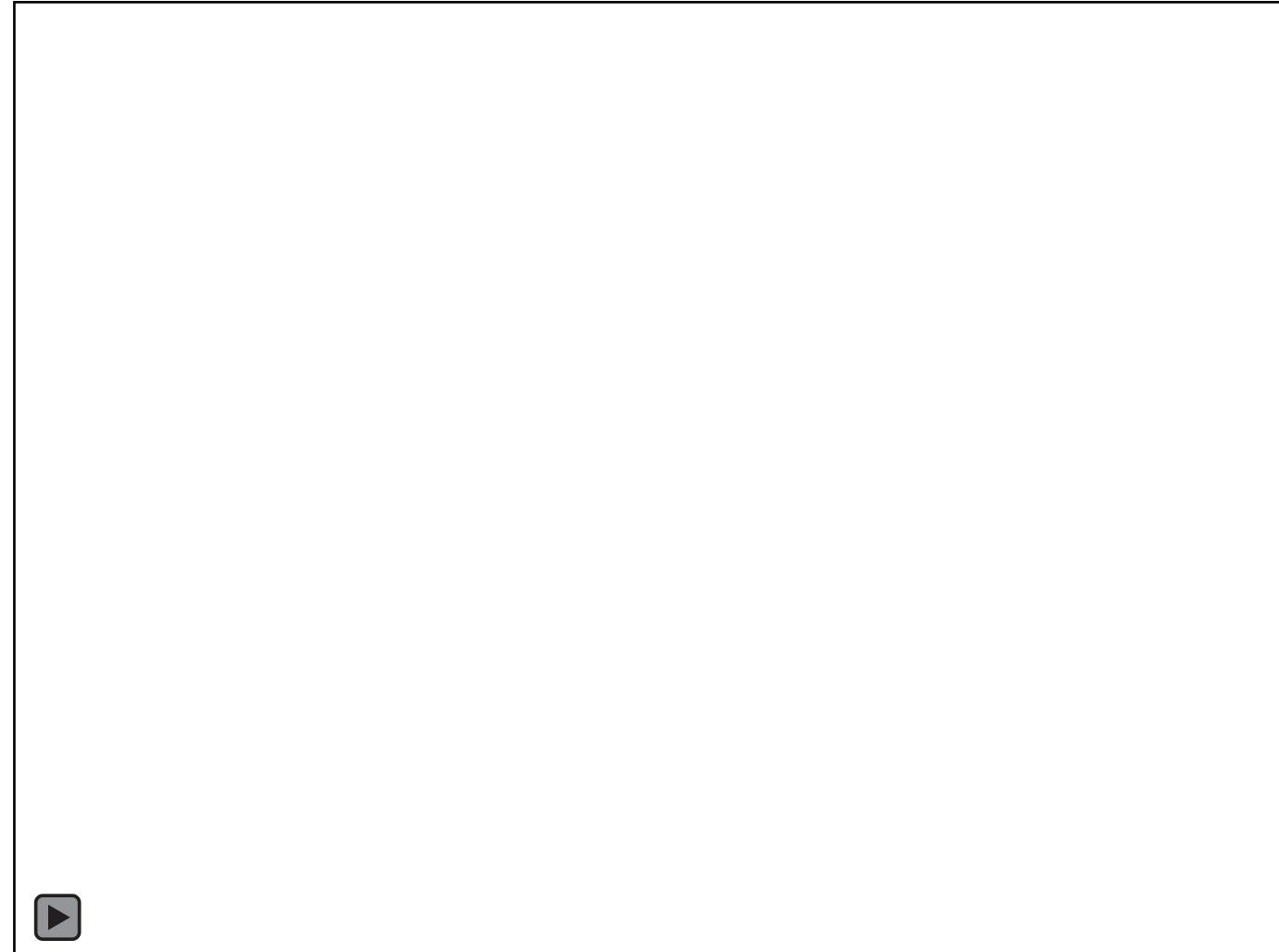
- Visual Studio Code

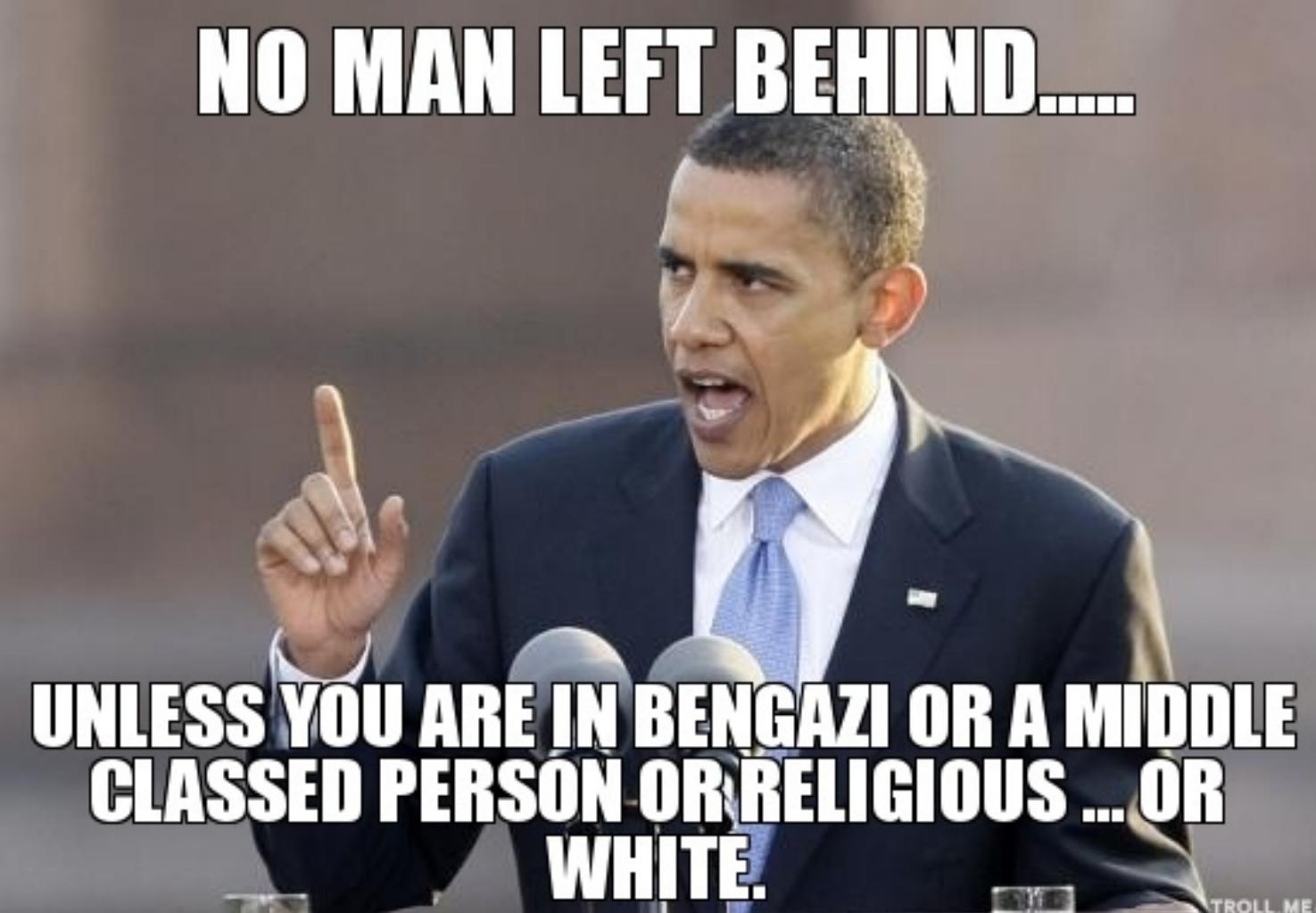
<https://code.visualstudio.com/>

- Node.js

<https://nodejs.org/>

- Antes de empezar es imprescindible ver este vídeo de [@garybernhardt](#).
- <https://www.destroyallsoftware.com/talks/wat>

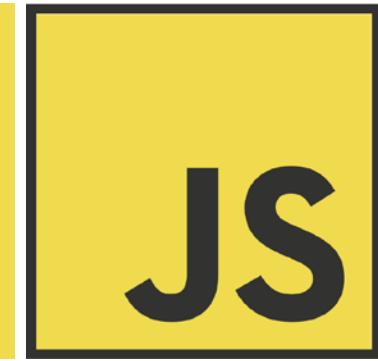




- Si surge alguna duda la comentamos.
- Es preferible que preguntéis a quedarnos con alguna duda que luego dificulte la comprensión de lo que se explique posteriormente.
- No hay problema en repasar algún concepto ya explicado si ayuda a entender mejor lo que se está explicando.

## Instalación de herramientas

# Introducción



- JavaScript **apareció en 1995**, creado por Brandan Eich.
- Su principal propósito era **validar entradas** en el cliente.
- En **1997 se presentó** al ECMA (European Computer Manufacturers Association) como propuesta.
- De ahí surgió ECMA-262, un estándar que definía un nuevo lenguaje llamado **ECMAScript**.
- Desde entonces, ECMAScript ha sido la **base** de todas las implementaciones de Javascript.
- En la actualidad es reconocido como un **lenguaje de programación completo**.
- JavaScript es a la vez muy simple y muy complicado: **lleva minutos aprenderlo pero muchos años dominarlo**.

Aunque JavaScript y ECMAScript se suelen usar como sinónimos, JavaScript es mucho más de lo que está definido en ECMA-262.

La implementación completa de JavaScript está compuesta por 3 partes:

1. El **Core** (ECMAScript)
2. El **DOM** (Document Object Model)
3. El **BOM** (Browser Object Model)

# ECMAScript



- ECMAScript no está ligado a los navegadores.
- ECMA-262 define ECMAScript como la base sobre la que otros lenguajes de script más robustos se han de construir.
- Los navegadores son uno de los entornos en los que puede existir una implementación de ECMAScript.
- Otros entornos en los que existe implementación de ECMAScript incluyen NodeJS y Adobe Flash.
- JavaScript implementa ECMAScript pero también lo hace Adobe ActionScript.

Básicamente, describe las siguientes partes del lenguaje:

- Sintaxis
- Tipos
- Sentencias
- Palabras clave
- Palabras reservadas
- Operadores
- Objetos

- Las diferentes versiones de ECMAScript se conocen como ediciones.
- La 2<sup>a</sup> edición fue básicamente editorial y no aportó nada.
- La 3<sup>a</sup> edición introdujo novedades: Try-catch, regular expressions, nuevas sentencias de control...
- La 4<sup>a</sup> fue una revisión completa de ECMA-262 que incluía tipado fuerte, clases, herencia clásica y nuevas estructuras.
- Paralelamente, se desarrolló una especificación llamada ECMAScript 3.1, fruto de una propuesta de evolución más pequeña.
- La versión 4 fue desechada antes de publicarse.
- ECMAScript 3.1 se convirtió en la 5<sup>a</sup> edición, lanzada en 2009: Objeto JSON, 'use strict', definición avanzada de propiedades, métodos de herencia nuevos...

# ECMAScript

Soporte en navegadores

JS

Si queréis saber qué navegadores tienen soporte para la edición actual y las que han de venir os recomiendo esta web: <http://kangax.github.io/compat-table/es5/>

ECMAScript 5 6 7 intl non-standard compatibility table

Please note that *some of these tests represent existence*, not functionality or full conformance. Sort by number of features? Show obsolete browsers?

The compatibility table displays support for various ECMAScript 5 features across different browsers. The columns represent different engines: es5-shim (yellow), IE 9 (light blue), IE 10+ (blue), FF 21+ (orange), SF 6+ (light green), WebKit (green), CH 23+, OP 15+ (red), OP 12.10 (dark red), Konq 4.13 (light blue), BESEN (orange), Rhino 1.7 (light orange), PhantomJS 2.0 (grey), EJS (yellow), and iOS7/8 (grey). The rows list features such as Object.create, Object.defineProperty, Object.defineProperties, Object.getPrototypeOf, Object.keys, Object.seal, Object.freeze, Object.preventExtensions, Object.isSealed, Object.isFrozen, Object.isExtensible, Object.getOwnPropertyDescriptor, Object.getOwnPropertyNames, Date.prototype.toISOString, Date.now, Array.isArray, JSON, Function.prototype.bind, String.prototype.trim, Array.prototype.indexOf, Array.prototype.lastIndexOf, Array.prototype.every, Array.prototype.some, Array.prototype.forEach, and Array.prototype.map. Most features are supported by all modern browsers, with some exceptions like Object.create and Object.defineProperty which are partially supported.

Feature name	Current browser	es5-shim	IE 9	IE 10+	FF 21+	SF 6+	WebKit	CH 23+, OP 15+	OP 12.10	Konq 4.13	BESEN	Rhino 1.7	PhantomJS 2.0	EJS	iOS7/8
Object.create	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Object.defineProperty	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Object.defineProperties	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Object.getPrototypeOf	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Object.keys	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Object.seal	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Object.freeze	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Object.preventExtensions	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Object.isSealed	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Object.isFrozen	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Object.isExtensible	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Object.getOwnPropertyDescriptor	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Object.getOwnPropertyNames	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Date.prototype.toISOString	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Date.now	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.isArray	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
JSON	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Function.prototype.bind	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
String.prototype.trim	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.indexOf	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.lastIndexOf	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.every	Yes <sup>[4]</sup>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.some	Yes	Yes <sup>[4]</sup>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.forEach	Yes	Yes <sup>[4]</sup>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.map	Yes	Yes <sup>[4]</sup>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.prototype.filter	Yes	Yes <sup>[4]</sup>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

DOM

JS

- El Document Object Model (DOM) es una API para XML cuyo uso se extendió al HTML.
- El DOM mapea toda la jerarquía de nodos de una página web. Estos nodos pueden ser creados, modificados, añadidos, borrados...
- El World Wide Web Consortium (W3C) se encarga de controlar la evolución del DOM.
- Existen, hasta el momento, 3 versiones o niveles.

Comprueba el soporte del DOM de los diferentes navegadores:

<http://www.webrowsercompatibility.com/dom/desktop/>

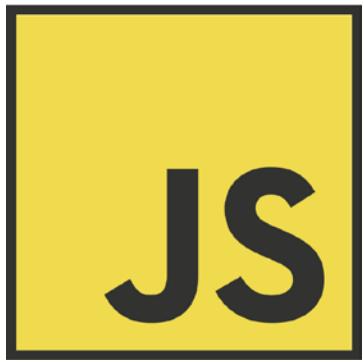
<http://www.w3.org/2003/02/06-dom-support.html>

BOM



- Mediante el Browser Object Model (BOM) se nos permite el acceso y manipulación de la ventana del navegador (window).
- El BOM era la única parte no estándar de JavaScript.
- Esto ha cambiado con la introducción de HTML5, que ha intentado formalizar buena parte del BOM en una especificación.
- A parte de trabajar con el window y los frames del navegador, el BOM se encarga de sus extensiones específicas, por ejemplo:
  - La capacidad de lanzar un pop up en una nueva ventana.
  - La capacidad de mover, modificar y cerrar una ventana.
  - Los objetos navigator, location y screen.
  - El soporte para cookies.
  - Objetos personalizados como el XMLHttpRequest.

# Multiparadigma



JavaScript permite y facilita el uso de múltiples paradigmas de programación:

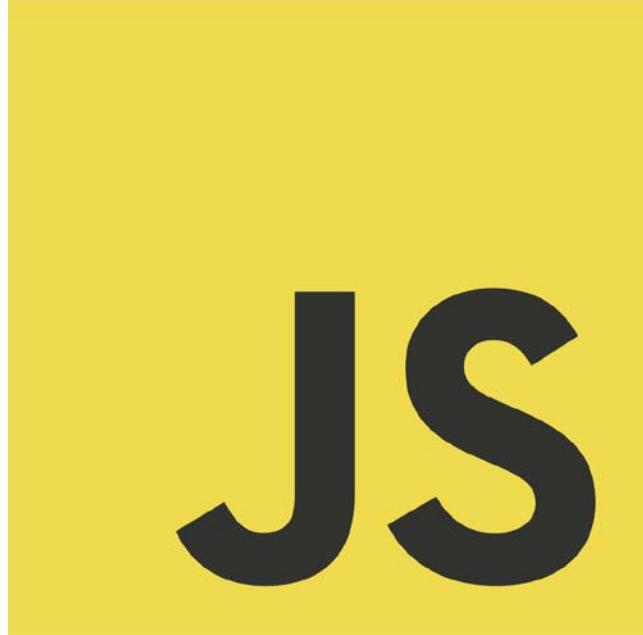
- Programación Imperativa
  - Basada en describir acciones en detalle.
- Programación Orientada a Objetos basada en Prototipo
  - Basada alrededor de los objetos prototípicos.
- Metaprogramación
  - Manipulación de la base del modelo de ejecución de JavaScript.
- Programación funcional
  - Basada en la utilización de funciones aritméticas sin manejo de datos mutables o de estado.

# Sumario



JS

- JavaScript es un lenguaje de script diseñado para interactuar con páginas webs y formado por tres partes diferenciadas:
  - ECMAScript, definido en ECMA-262 que provee la funcionalidad central.
  - El DOM, que provee métodos para trabajar con el contenido de las páginas web.
  - El BOM, que provee métodos para interactuar con el navegador.
- La mayoría de navegadores soportan ECMAScript 3.
- El soporte para ECMAScript 5 está creciendo.
- El soporte del DOM y el BOM es bastante variable.
- JavaScript soporta múltiples paradigmas de programación.



# JavaScript Desmitificado

JavaScript en el HTML

東京' TOKIOTA

 Microsoft  
Gold Partner

<script>

JS

- El elemento `<script>` es el método principal para insertar JavaScript en un HTML.
- Consta de 6 atributos:
  - `async`: no bloquea la carga de otros recursos. Sólo válido para scripts externos.
  - `charset`: la mayoría de navegadores no lo tiene en cuenta.
  - `defer`: espera a que se cargue el documento. Sólo valido para scripts externos.
  - `language`: obsoleto. No usar.
  - `src`: script externo a ejecutar.
  - `type`: indica el MIME type. Si no aparece se asume “`text/javascript`”.
- Se puede usar de dos modos diferentes:
  - Embebiendo JavaScript directamente en la página.
  - Incluyendo JavaScript de un fichero externo.

- Ejemplo embebido:

```
<script type="text/javascript">
    function test(){
        console.log('Hello');
    }
</script>
```

- Ejemplo carga externa:

```
<script src="test.js" type="text/javascript"></script>
```

- El elemento <script> permite incluir ficheros JavaScript fuera del dominio.
- Los elementos <script> de una página son interpretados en orden de aparición siempre que no contengan los atributos *defer* o *async*.
- El elemento <script> bloquea las descargas progresivas de la página.
- No omitir el tag de cierre </script> y usar <script .../> porque es HTML inválido.

- Tradicionalmente, los elementos `<script>` se han venido posicionando en el elemento `<head>` de las páginas.
- La idea era mantener las referencias a los archivos CSS y JavaScript en la misma área.
- Esto implicaba que todo el código JavaScript debía ser descargado, parseado e interpretado antes de que la página empezase a renderizar.
- El renderizado de una página empieza cuando el navegador recibe la etiqueta de apertura `<body>`.
- Las aplicaciones web modernas incluyen la referencias JavaScript en el elemento `<body>`, después del contenido de la página para agilizar el renderizado de la página.

# Estrategias de carga

JS

- Si añadimos el atributo *defer* al elemento `<script>` la descarga comenzará de inmediato pero la ejecución se efectuará después del renderizado.
- En teoría, estos scripts serán ejecutados en orden. A la práctica, no siempre.
- En teoría, estos scripts se ejecutarán antes del evento *DOMContentLoaded*. A la práctica no siempre.
- Se recomienda usar un único elemento `<script>` para evitar problemas con los dos anteriores puntos.
- El atributo *defer* sólo se soporta para carga de scripts externos.

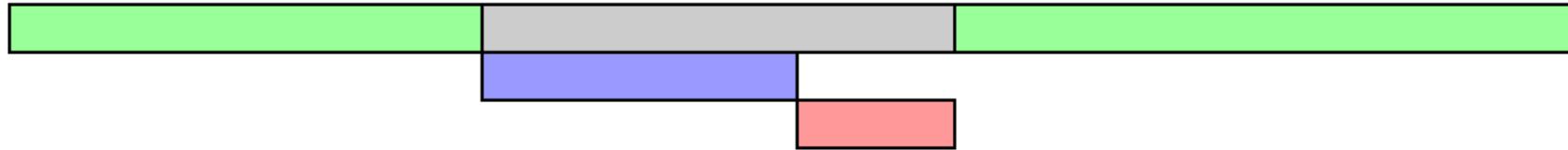
- HTML5 introduce el atributo *async* para el elemento `<script>`.
- El atributo *async* provocará la descarga y ejecución inmediatas pero la descarga no producirá ningún bloqueo.
- Sólo aplica a scripts externos.
- No se garantiza que los scripts se carguen en orden.

# Estrategias de carga

normal vs async vs defer

JS

<script>



Legend

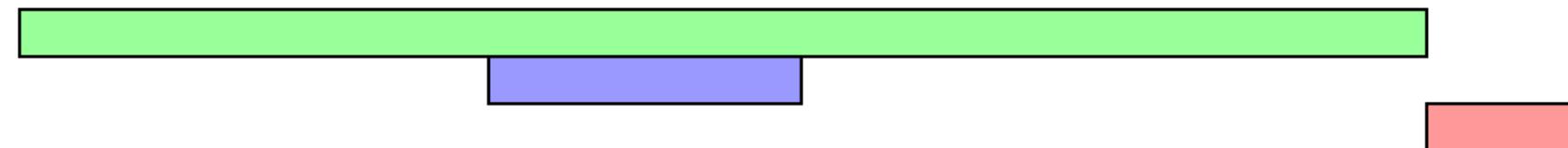
- HTML parsing
- HTML parsing paused
- Script download
- Script execution

<script async>

<http://www.growingwiththeweb.com/2014/02/async-vs-defer-attributes.html>



<script defer>



- El DOM nos permite crear nodos dinámicamente por lo que podemos usar sus métodos para crear un elemento `<script>` de manera dinámica.

```
var script = document.createElement('script');
script.type = 'text/javascript';
script.src = 'app.js';
document.getElementsByTagName('head')[0].appendChild(script);
```

- El fichero comenzaría a descargarse al momento en que añadamos el nodo a la página.
- El fichero se descargaría y se ejecutaría sin bloquear ningún proceso de la página, independientemente desde donde se inicie la descarga.
- Esta técnica nos permite cargar scripts de fuera del dominio.

- Esta técnica se refiere a la carga de un script dinámico después de la carga de la página.
- Se recomienda separar la carga de los scripts en dos paquetes:
  - Una parte, requerida para inicializar la página.
  - Otra que sólo se necesite bajo ciertas condiciones.
- La idea es cargar la página lo antes posible con lo mínimo necesario y cargar el resto en el *background*.
- El primer script se carga normalmente y el segundo de manera dinámica.

# Estrategias de carga

Lazy Loading

JS

```
<script src="core.js"></script>  
<script>  
    window.onload = function () {  
        var script = document.createElement('script');  
        script.src = 'lazyloaded.js';  
        document.documentElement.firstChild.appendChild(script);  
    };  
</script>  
</body>  
</html>
```

- Como el primer script bloqueará la carga del segundo, éste no se ejecutará hasta después del primero.

- Imaginemos un escenario en el cual un script sólo es necesario al realizar determinada acción. Por ejemplo, pulsar un determinado botón cargará una animación.
- Si el usuario no pulsa ese botón el script es totalmente innecesario. Por tanto, ante un escenario de este tipo, la carga de scripts bajo demanda cobra sentido.
- Para cargar un script bajo demanda podemos usar la técnica de inserción de scripts dinámicos.
- Una vez cargado el script ejecutamos una función (callback) encargada de empezar la animación.
- ¿Cómo podemos controlar la carga de este script?

# Estrategias de carga

Función getScript

JS

```
function getScript(url, callback) {  
    var script = document.createElement('script');  
    script.type = 'text/javascript';  
    script.src = url;  
    script.onload = function () {  
        callback && callback();  
    };  
    document.getElementsByTagName('head')[0].appendChild(script);  
}
```

- Armados con la anterior función podemos, al hacer click en nuestro botón, cargar nuestro script y luego efectuar la lógica que necesitemos.
- Este patrón también nos es útil si el script que cargamos necesita inicializarse externamente:

```
getScript('comm.js', function(){  
    comm.init(); //inicializamos nuestra recién cargada librería  
});
```

- Podemos mejorar nuestra función getScript detectando si el script ha sido ya cargado en nuestra página o no.

# Estrategias de carga

Función getScript mejorada

JS

```
function getScript(url, callback) {  
    if(document.querySelector('script[src=' + url + ']')) {  
        callback && callback();  
        return;  
    }  
  
    var script = document.createElement('script');  
    script.type = 'text/javascript';  
    script.src = url;  
    script.onload = function () {  
        callback && callback();  
    };  
  
    document.getElementsByTagName('head')[0].appendChild(script);  
}
```

- Varias librerías implementan una función getScript o similar.

<https://api.jquery.com/jquery.getscript/>

- Se pueden cargar scripts usando hacks de este tipo:

```
location.href="javascript:var s =  
document.createElement('script');s.type='text/javascript';document.body.appendChild(s);s.src='http://erkie.git  
hub.com/asteroids.min.js';void(0);";
```

- Esta técnica es útil para cargar scripts desde la barra de favoritos, pero no está recomendada.

- Otra aproximación para cargar scripts de manera no bloqueante es usar el objeto XMLHttpRequest (XHR) para obtener el script e injectarlo en la página.
- La ventaja de esta técnica es que podemos descargar el código JS sin ejecutarlo de inmediato.
- La desventaja es que el archivo JS debe pertenecer al mismo dominio que la página.
- Esto dificulta la descarga de ficheros desde CDNs.
- Esta técnica no se suele usar en aplicaciones web grandes.

# Estrategias de carga

Inyección vía XMLHttpRequest

JS

```
var xhr = new XMLHttpRequest();
xhr.open('get', 'myremotefile.js', true);
xhr.onreadystatechange = function(){
    if (xhr.readyState == 4){
        if (xhr.status >= 200 && xhr.status < 300 || xhr.status == 304){
            var script = document.createElement('script');
            script.type = 'text/javascript';
            script.text = xhr.responseText;
            document.body.appendChild(script);
        }
    }
};
xhr.send(null);
```

- Esta técnica consiste en cargar ficheros JS que no son necesarios en la página actual pero si en las subsiguientes.
- Imaginad un usuario haciendo login. Casi seguro que seguirá navegando por nuestra web.
- Si cargamos el fichero JS mientras entra los datos, lo tendremos cacheado y la carga en las páginas siguientes será más rápida.
- El problema, si usamos alguna de las técnicas vistas hasta ahora, es que no queremos que se parsee ni se ejecute.
- Para ello usaremos el patrón Image Beacon.

- Se trata de crear una imagen usando JavaScript y usar su atributo *src* para cargar el script:

```
function preload(src){  
    new Image().src='ourScript.js';  
}
```

- Si el navegador no soporta la opción de usar Image Beacon podemos usar la siguiente técnica:

```
function preload(src){  
    var obj = document.createElement('object');  
    obj.data = 'https://code.jquery.com/jquery-2.1.4.min.js';  
    obj.height = obj.width = '0';  
    document.body.appendChild(obj);  
}
```

- La aproximación recomendada para cargar gran cantidad de JavaScript en una página sin problemas de bloqueo es un proceso de dos pasos:
  1. Cargar el código necesario para cargar dinámicamente el JavaScript (nuestra función `getScript`).
  2. Cargar el resto del código JavaScript necesario para la inicialización de la página.

```
<script type="text/javascript" src="loader.js"></script>  
  
<script type="text/javascript">  
  getScript('the-rest.js', function(){  
    Application.init();  
  });  
</script>
```

- Por cada elemento `<script>` encontrado en la página habrá una pausa mientras el código se ejecuta; tanto para scripts *inline* como externos.
- Las peticiones HTTP son muy costosas y cada una aporta una sobrecarga de rendimiento a la página.
- Descargar un fichero de 100KB será más rápido que 4 de 25KB.
- Es importante:
  - minimizar la aparición de elementos `<script>` en la página.
  - cargar la menor cantidad de scripts externos posibles.
- Para ello, concatenaremos todos nuestros ficheros en un solo archivo.
- Podemos usar un Task Runner (Gulp, Grunt...) para automatizar esta tarea.

- Minificación es el proceso por el cual se elimina de los ficheros JavaScript todo aquello que no contribuye a su ejecución.
- Esto incluye comentarios y espacios en blanco innecesarios.
- Este proceso habitualmente reduce el tamaño del archivo a la mitad, resultando en descargas más rápidas.
- Hay diferentes herramientas que permiten también optimizar esta minificación acortando nombres de variables y demás técnicas.
- Este proceso se puede automatizar con el uso de Task Runners como Gulp o Grunt.
- Es importante ser estricto en la sintaxis (puntos y coma) para evitar errores durante el proceso de minificación que luego pueden ser difíciles de detectar.

- La compresión gzip en el servidor es obligatoria.
- La compresión reducirá de media tus archivos en un 70%.

# Addendum

A yellow square containing the letters "JS" in a bold, black, sans-serif font.

JS

# Inline vs Ficheros externos

JS

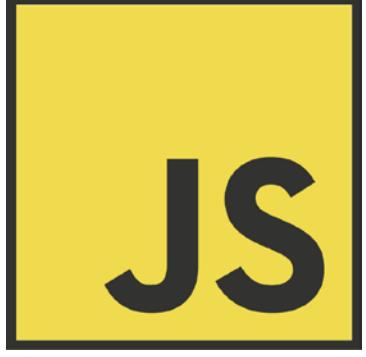
- Aunque hemos visto que se puede incluir JavaScript en el HTML es considerada una buena práctica incluir todo lo que sea posible en ficheros externos.
- Los argumentos a favor de esta práctica son:
  1. **Mantenibilidad**: nos evitamos tener esparcido el código entre diversos ficheros HTML.
  2. **Cache**: aprovechamos la caché de los navegadores. Si dos páginas usan el mismo fichero sólo se descargará una vez.

# <noscript>

- El elemento `<noscript>` se creó para proveer de contenido alternativo a todos aquellos navegadores que no soportan JavaScript.
- Puede contener cualquier elemento HTML.
- Su contenido se mostrará sólo si:
  1. El navegador no soporta JavaScript.
  2. El navegador tiene JavaScript deshabilitado.

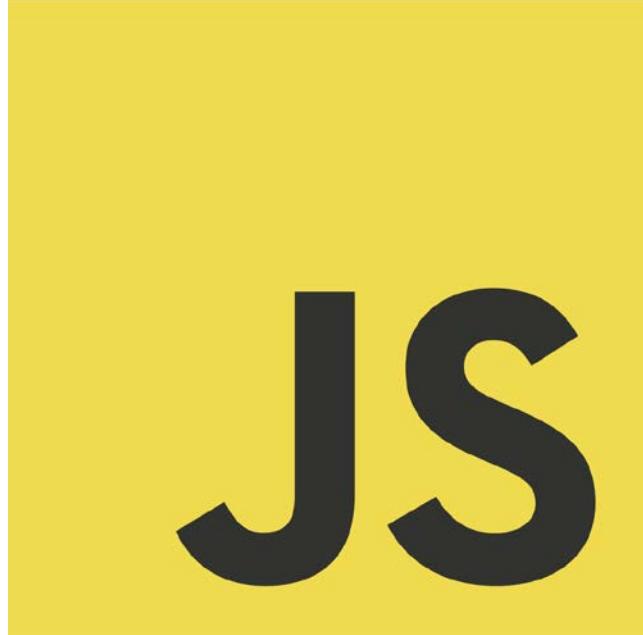
```
<noscript>
  <p>Esta página requiere un navegador con JavaScript activado.</p>
</noscript>
```

# Sumario



JS

- El elemento <script> sirve para insertar JavaScript en el HTML.
- Se recomienda minimizar su uso y usarlos al final del elemento <body> para evitar bloquear el renderizado de la página.
- Hemos visto diversas técnicas para evitar o minimizar este bloqueo: defer, async, carga dinámica, XHR...
- Agrupar, minificar y servir comprimidos los ficheros JavaScript puede reducir al 85% el tamaño de los archivos servidos.
- Usar archivos externos es considerada una buena práctica.



# JavaScript Desmitificado

Base del lenguaje

東京' TOKIOTA



# Sintaxis

A yellow square containing the letters "JS" in a bold, black, sans-serif font.

JS

- La sintaxis de ECMAScript toma prestadas muchas cosas de C y otros lenguajes parecidos como Java y Perl.
- Case sensitive.
- Los identificadores (nombres de variables, funciones...) deben de comenzar por una letra, un guion bajo (\_) o un signo de dólar (\$).
- Los comentarios son tipo C:
  - //una línea
  - /\*  
 \* multilínea  
 \*/

- ECMAScript 5 introduce el concepto de *strict mode*.
- Es un modelo diferente de parseo y ejecución del JavaScript que trata de enmendar ciertos comportamientos erráticos de ECMAScript 3.
- Se habilita añadiendo '*use strict*'; al inicio de un script.
- El uso de este pragma se puede hacer también a nivel de función:

```
function helloWorld(){  
  'use strict';  
  //function body...  
}
```

- Las sentencias acaban en punto y coma.
- Si se omiten, el parseador determina el final de la sentencia.

`var sum = a + b //válido aunque no recomendado`

`var diff = a -b; //válido y preferido`

- Se recomienda siempre incluir punto y coma al final de una sentencia para evitar problemas en la minificación.
- Incluir puntos y coma mejora el rendimiento en ciertas circunstancias ya que el parseador no ha de tratar de insertarlos.
- Se pueden anidar varias sentencias en un bloque de código usando las llaves {}.

- Las sentencias de control, como if, requieren bloques de código sólo cuando se ejecutan múltiples sentencias.
- Es una buena práctica incluir siempre las llaves aunque sólo se ejecute una sola sentencia.

```
if(valid)
  console.log('Hello'); //válido pero es proclive a generar futuros errores.
```

```
if(valid){
  console.log('Hello'); //recomendado
}
```

# Palabras clave

JS

- Las palabras clave están reservadas y no se pueden usar para los identificadores.
- Palabras clave actuales:

break	var	switch	if
do	catch	while	throw
instanceof	finally	debugger	delete
typeof	return	function	in
case	void	this	try
else	continue	with	
new	for	default	

- Palabras clave reservadas para uso futuro en ECMA-262, 3<sup>a</sup> edición:

abstract

enum

int

short

boolean

export

interface

static

byte

extends

long

super

char

final

native

synchronized

class

float

package

throws

const

goto

private

transient

debugger

implements

protected

volatile

double

import

public

- Palabras clave reservadas para uso futuro en 5<sup>a</sup> edición sin strict mode:

class

extends

const

import

enum

super

export

- 5<sup>a</sup> edición con strict mode añade éstas a las anteriores:

implements

interface

let

package

private

protected

public

static

yield

# Variables

JS

- Las variables pueden contener cualquier tipo de dato.
- Las variables son contenedores con nombre para un valor.
- Para definirlas se usa el operador *var* seguido del nombre de la variable.
- Si no se inicializan contendrán el valor *undefined*.

```
var game; //undefined
```

- Si se inicializan podemos cambiar el valor contenido aunque sea de otro tipo.
- ```
var game= 'pang';
game = 1000;
```
- En general, no se recomienda cambiar el tipo del valor contenido aunque es completamente legal en ECMAScript.

- Usando el operador `var` la variable se define como local en el *ámbito (scope)* en que ha sido definida.

```
function test() {  
    var message = 'hola'; //variable local  
}  
test();  
alert(message); //error!
```

- La variable *message* se define dentro de la función y es destruida una vez la función se ha ejecutado.

- Se puede definir una variable globalmente omitiendo el operador *var*.

```
function test() {  
    message = 'hola'; //variable local  
}  
test();  
alert(message); //hola
```

- En los navegadores, una función definida globalmente se asigna al elemento *window*.
- Esta práctica no está recomendada ya que causa confusión.
- En strict mode obtendremos un error.

- Se pueden definir varias variables en una sola sentencia separando las definiciones mediante una coma.

```
var position = 1,  
    color = 'red',  
    game = {  
        cover: 123,  
        name: 'double dragon'  
    },  
    message;
```

# Tipos de datos

JS

- Hay 5 tipos de datos simples o tipos primitivos:
  1. Undefined
  2. Null
  3. Boolean
  4. Number
  5. String
- Hay un tipo de datos complejo:
  1. Object
- Todos los valores pueden ser representados mediante estos 6 tipos de datos.

- El operador *typeof* nos ayuda a determinar el tipo de dato del valor de una variable.
- Usado sobre un valor retorna una de estas cadenas:
  - “undefined”
  - “boolean”
  - “string”
  - “number”
  - “object” (excepto en las funciones)
  - “function”
- Lo podemos usar así:

```
typeof 95; //number
```

- Cuando una variable se declara pero no se define recibe el valor *undefined*.

```
var game;  
console.log(game === undefined); //true
```

- Es importante resaltar la diferencia entre una variable declarada y no definida y una directamente no declarada.

```
var game;           //undefined  
//var device;      //not declared  
console.log(game); //undefined  
console.log(device); //error  
console.log(typeof game); // "undefined"  
console.log(typeof device); // "undefined"
```

- El valor *null* es un puntero a un objeto vacío, por eso *typeof* retorna “object”.

```
typeof null; // "object"
```

- El valor *undefined* deriva de *null*. Por eso:

```
null == undefined; // true
```

```
null === undefined; // false
```

- Es aconsejable inicializar los objetos con *null* si luego se espera que tengan algún valor.
- Esto nos permite diferenciar claramente entre *null* y *undefined*.

- Tiene dos posibles valores literales: *true* y *false*.
- Todos los tipos de valores tienen equivalencia booleana en ECMAScript.
- Para hacer el casting podemos usar la función *Boolean()*.
- También podemos usar el operador ! dos veces:

```
console.log(!!'blue'); //true
```

| DATA TYPE | VALUES CONVERTED TO TRUE                | VALUES CONVERTED TO FALSE                             |
|-----------|-----------------------------------------|-------------------------------------------------------|
| Boolean   | true                                    | false                                                 |
| String    | Any nonempty string                     | "" (empty string)                                     |
| Number    | Any nonzero number (including infinity) | 0, NaN (See the "NaN" section later in this chapter.) |
| Object    | Any object                              | null                                                  |
| Undefined | n/a                                     | undefined                                             |

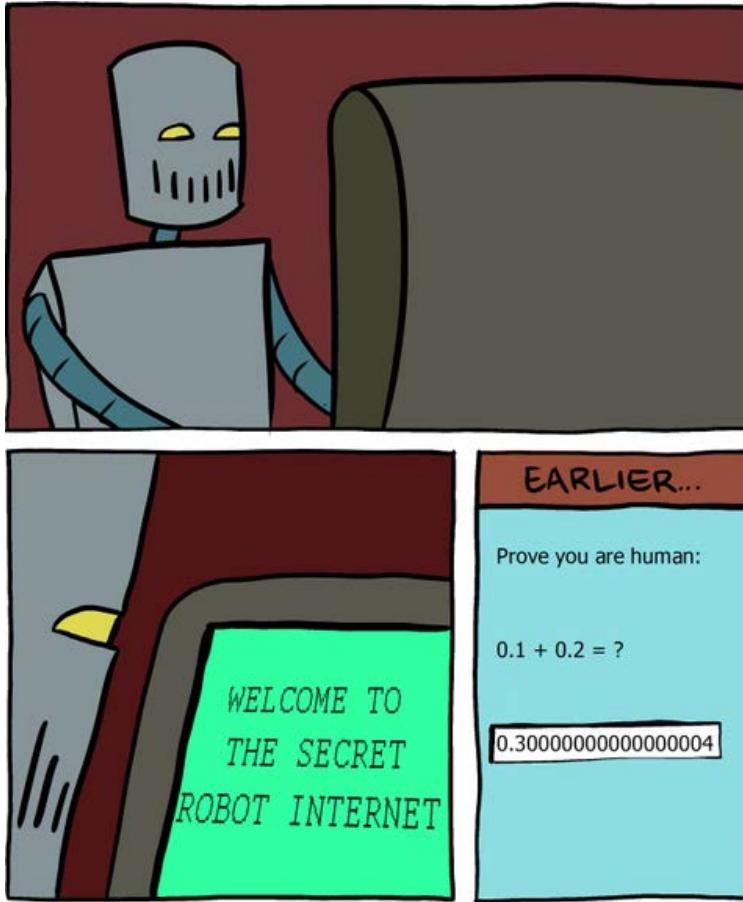
Es importante entender estas conversiones ya que las sentencias de control de flujo automáticamente hacen una conversión a Boolean.

```
var game = 'golden axe';
if(game){ // true
    console.log('do something');
}
```

# Tipos de datos

Number

JS



<http://goo.gl/qjAgX7>

- Usa el formato [IEEE-754](#) para representar enteros y valores de coma flotante.
- Es importante entender los efectos del redondeo que presenta el formato IEEE-754.
- $0.1 + 0.2 \neq 0.3 // 0.30000000000000004$
- No usar comparaciones específicas con números de coma flotante.
- Este hecho afecta también a C# y Java.
- Number.MIN\_VALUE = [5e-324](#)
- Number.MAX\_VALUE = [1.7976931348623157e+308](#)

- Los enteros se pueden representar en formato decimal, octal y hexadecimal.

```
var intNum = 55;  
var octalNum1 = 070; //56  
var octalNum2 = 079; //octal inválido – 0 seguido de números de 0 a 7. Se interpreta como decimal.  
var octalNum3 = 08; //octal inválido. Decimal.  
var hexNum1 = 0xA; //10  
var hexNum2 = 0x1f; //31
```

- Los octales son inválidos en *strict mode*.
- Los números creados con formato octal o hexadecimal son tratados como decimales en las operaciones aritméticas.

- Los números de coma flotante necesitan incluir un punto decimal y al menos un número detrás.
- Para ahorrar memoria ECMAScript siempre intenta convertir todos los números a enteros.

```
var floatNum1 = 1.; //interpretado como entero 1  
var floatNum2 = 10.0; //interpretado como entero 10
```

- Para números muy grandes o muy pequeños podemos usar la *notación e*.

```
var floatNum3 = 3.12e7; // 31200000  
var floatNum4 = 3e-7; // 0.0000003
```

- Si nos pasamos los rangos numéricos soportados el número obtiene uno de los siguientes valores:
  - Infinity
  - -Infinity
- La función *isFinite()* nos puede ayudar a determinar si estamos dentro del rango soportado.

```
var result = Number.MAX_VALUE + Number.MAX_VALUE;  
console.log(isFinite(result)); // false
```

- En ECMAScript, un número dividido entre 0 retorna *Infinity* o *-Infinity*.

- *NaN* es un valor numérico especial (Not a Number) que indica un error en una operación que debería haber retornado un número.
- Cualquier operación con *NaN* retorna *NaN*.
- *NaN* no se iguala a ningún valor. Ni siquiera a *NaN*. Por eso tenemos disponible la función *isNaN()*.

```
NaN*10;      //NaN  
NaN == NaN; //false  
isNaN(NaN); //true  
isNaN('blue'); //true
```

- Hay 3 funciones para convertir valores a números:
  - *Number()*, puede ser usada sobre cualquier tipo de dato.
  - *parseInt()* y *parseFloat()*, sólo sirven sobre cadenas.
- El operador unario + también puede usarse para hacer conversión a número.  
`+'blue'; //NaN`
- Se aconseja siempre pasar el *radix* en la función *parseInt()*.  
`var num1 = parseInt('10', 2); //2 - binario`

- Representa una secuencia de 0 o más caracteres Unicode de 16-bit.
- Se puede usar comillas simples (' ) o dobles(" ). Ambas son legales.
- Las cadenas (*strings*) son inmutables.
- Para convertir a cadena podemos usar:
  - el método *toString()* que tienen casi todos los valores, excepto null y undefined.
  - la función *String()*, se puede usar sobre cualquier valor.

- El tipo de datos String incluye varios caracteres literales para representar ciertos caracteres no imprimibles o útiles:

| LITERAL | MEANING                                                                                                                                                                              |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \n      | New line                                                                                                                                                                             |
| \t      | Tab                                                                                                                                                                                  |
| \b      | Backspace                                                                                                                                                                            |
| \r      | Carriage return                                                                                                                                                                      |
| \f      | Form feed                                                                                                                                                                            |
| \\\     | Backslash ()                                                                                                                                                                         |
| \'      | Single quote ('') — used when the string is delineated by single quotes. Example:<br>'He said, \'hey.\''.                                                                            |
| \\"     | Double quote (") — used when the string is delineated by double quotes. Example:<br>"He said, \"hey.\"".                                                                             |
| \xnn    | A character represented by hexadecimal code <code>nn</code> (where <code>n</code> is a hexadecimal digit 0-F). Example: \x41 is equivalent to "A".                                   |
| \unnnn  | A Unicode character represented by the hexadecimal code <code>nnnn</code> (where <code>n</code> is a hexadecimal digit 0-F). Example: \u03a3 is equivalent to the Greek character Σ. |

- Object es la base de todos los objetos en ECMAScript así que todas las instancias de Object comparten las siguientes propiedades y métodos:
  - constructor
  - hasOwnProperty(propertyName)
  - isPrototypeOf(object)
  - propertyIsEnumerable(propertyName)
  - toLocaleString()
  - toString()
  - valueOf()

# Operadores

JS

Los operadores unarios son aquellos que sólo operan sobre un solo valor.

```
var n = 1;  
var r = ++n; // n = 2, r = 2
```

```
var n = 1;  
var r = --n; // n = 0, r = 0
```

```
var n = 1;  
var r = +n; // n = 1, r = 1
```

```
+ '01'; // 1  
+ '1.1'; // 1.1  
+ 'blue'; // NaN  
+ true; // 1
```

```
var n = 1;  
var r = n++; // n = 2, r = 1
```

```
var n = 1;  
var r = n--; // n = 0, r = 1
```

```
var n = 1;  
var r = -n; // n = 1, r = -1
```

```
- '01'; // -1  
- '1.1'; // -1.1  
- 'blue'; // NaN  
- true; // -1
```

Estos operadores trabajan sobre números a nivel de bits.

Bitwise NOT: ~

```
var n = 25;  
var r = ~n; // -26  
var r = -n-1; // -26
```

Bitwise AND: &

| BIT FROM FIRST NUMBER | BIT FROM SECOND NUMBER | RESULT |
|-----------------------|------------------------|--------|
| 1                     | 1                      | 1      |
| 1                     | 0                      | 0      |
| 0                     | 1                      | 0      |
| 0                     | 0                      | 0      |

Bitwise OR: |

| BIT FROM FIRST NUMBER | BIT FROM SECOND NUMBER | RESULT |
|-----------------------|------------------------|--------|
| 1                     | 1                      | 1      |
| 1                     | 0                      | 1      |
| 0                     | 1                      | 1      |
| 0                     | 0                      | 0      |

Bitwise XOR: ^

| BIT FROM FIRST NUMBER | BIT FROM SECOND NUMBER | RESULT |
|-----------------------|------------------------|--------|
| 1                     | 1                      | 0      |
| 1                     | 0                      | 1      |
| 0                     | 1                      | 1      |
| 0                     | 0                      | 0      |

Estos operadores trabajan sobre números a nivel de bits.

Left Shift: <<

"Secret" sign bit                          The number 2  
↓  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0

The number 2 shifted to the left five bits (the number 64)  
↓  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0  
↓  
Padded with zeros

Unsigned Right Shift: >>>

Para valores positivos es igual que >>

Para valores negativos, este operador considera la representación binaria del número como si fuera positiva.

Signed Right Shift: >>

"Secret" sign bit                          The number 64  
↓  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0

The number 64 shifted to the right five bits (the number 2)  
↓  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0  
↓  
Padded with zeros

### Negación: !

- Se puede aplicar a cualquier valor.
- Se puede usar doble negación para convertir a booleano (!!).

### Conjunción: &&

| OPERAND 1 | OPERAND 2 | RESULT |
|-----------|-----------|--------|
| true      | true      | true   |
| true      | false     | false  |
| false     | true      | false  |
| false     | false     | false  |

### Disyunción: ||

| OPERAND 1 | OPERAND 2 | RESULT |
|-----------|-----------|--------|
| true      | true      | true   |
| true      | false     | true   |
| false     | true      | true   |
| false     | false     | false  |

### Falsy

- false
- 0
- "" (cadena vacía)
- null
- undefined
- NaN

### Truthy

- Todo lo demás es truthy, incluyendo '0', 'false'.

Es importante tener claros estos conceptos para poder usarlos efectivamente en nuestras sentencias de control de flujo.

- Multiplicación: \*    `var r = 4 * 2; //8`
- División: /            `var r = 4 / 2; //2`
- Módulo: %            `var r = 4 % 2; //0`
- Suma: +              `var r = 4 + 2; //6`      `var r = 5 + '5'; //'55'`
- Resta: -            `var r = 4 - 2; //2`

- Mayor que: >
- Menor que: <
- Mayor o igual que: >=
- Menor o igual que: <=

## Casos interesantes:

```
'a' < 3;           //false, 'a' se transforma en NaN
NaN < 3;          //false
NaN >= 3;         //false
'Casa' < 'autobus'; //true
'casa' < 'autobus'; //false, C char code es menor que el c char code.
'23' < '3';        //true, comparación de chars.
'23' < 3;          //false, se convierte el '23' a número.
```

Determinar si dos variables son equivalentes es una de las operaciones más importantes en programación.

Con coerción de tipos:

- Igual: ==
- No igual: !=

| EXPRESSION        | VALUE |
|-------------------|-------|
| null == undefined | true  |
| "NaN" == NaN      | false |
| 5 == NaN          | false |
| NaN == NaN        | false |
| NaN != NaN        | true  |
| false == 0        | true  |
| true == 1         | true  |
| true == 2         | false |
| undefined == 0    | false |
| null == 0         | false |
| "5" == 5          | true  |

Sin coerción de tipos:

- Idénticamente Igual: ===
- Idénticamente No Igual: !==

```
'55' == 55; //true  
'55' === 55; //false
```

- Ternario: booleano ? valor\_cierto : valor\_falso

```
var r = true ? 1 : 2; //1  
var r = false ? 1 : 2; //2
```

- Asignación: =
- Asignación compuesta: \*=, /=, %=, +=, -=, <<=, >>=, >>>=
- Coma: ,

- permite la ejecución de más de una operación en una sola sentencia

```
var n = 1, n2 = 2, n3 = 3;
```

- se puede usar para asignar valores

```
var n = (1, 2, 3, 4, 0); // n = 0
```

# Control de flujo

JS

*if*(condición) sentencia1 *else* sentencia2

*if*(condición) sentencia1 *else if*(condición2) sentencia2 *else* sentencia3

- La condición puede ser cualquier valor. ECMAScript usará internamente la función *Boolean()*.
- Se considera una buena práctica usar siempre llaves, aunque se ejecute una sola sentencia, para evitar confusiones.
- Se pueden encadenar sentencias *if*

*do* {

sentencia

} *while* (expresión);

- Se trata de un bucle de *post-test*. La condición de escape se evalúa después de que el código dentro del bucle se ejecute.
- El cuerpo del bucle se ejecuta al menos una vez.

## *while*(expresión) sentencia

- Es un bucle de *pre-test*. La condición de escape se evalúa antes de que el código dentro del bucle se ejecute.

**for** (inicialización; expresión de control; expresión post-bucle) sentencia

- Bucle de pre-test.
- La inicialización, la expresión de control y la expresión post-bucle son opcionales.
- Este tipo de bucle es muy versátil y es uno de los más usados.

## **for**(propiedad *in* expresión) sentencia

- Es estrictamente iterativo, usado para enumerar las propiedades de un objeto.
- En cada iteración, la variable *propiedad* contendrá el nombre de una de las propiedades del objeto.
- El orden en que aparecen las propiedades no se puede predecir.
- Se puede usar con *arrays* pero no se recomienda:
  - Orden impredecible
  - Arrays con propiedades extendidas

*etiqueta*: sentencia

- Se pueden etiquetar sentencias para ser referenciadas a posteriori mediante el uso de las sentencias *break* o *continue*.

- Estas sentencias proveen de un control estricto sobre la ejecución del código en un bucle.
- *break*: sale del bucle de manera inmediata y ejecuta la siguiente sentencia encontrada después del bucle.
- *continue*: sale del bucle inmediatamente pero la ejecución continúa al inicio del bucle.
- Estas dos sentencias se pueden usar junto a las etiquetas para obtener más control sobre bucles anidados.

## [with](#) (expresión) sentencia

- Establece el ámbito del código en un objeto particular.
- El uso de esta sentencia está altamente desaconsejado.

<https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Statements/with>

# Control de flujo

switch

JS

```
switch (expression) {  
    case value: statement  
        break;  
    case value: statement  
        break;  
    default: statement  
}
```

- Es buena práctica usar siempre *break* y evitar casos de *fall-through*.

# Funciones

JS

```
function nombreFuncion (arg0, arg1, ..., argN){  
    sentencias  
}
```

- Las funciones son el núcleo del lenguaje.
- Permiten la encapsulación de sentencias para ser ejecutadas a voluntad.
- Las funciones se declaran usando la palabra clave *function*, seguida por los argumentos y el cuerpo de la función.
- El nombre de la función es opcional.

- Las funciones se llaman usando el nombre de la función o de la variable que contiene una referencia a las mismas.
- No necesitan especificar si retornan un valor.
- Pueden retornar un valor usando la sentencia *return*.
- Una función para su ejecución y sale de su cuerpo al encontrarse con *return*.
- Puede existir más de una sentencia *return* en una misma función.
- La sentencia *return* puede ser usada sin especificar un valor de retorno.
- <https://www.youtube.com/watch?v=hQVTIJBZook> //32:39

- Una función declarada con n argumentos puede aceptar m argumentos, siendo  $m == n$  o  $m != n$ .
- Dentro de la función tenemos acceso a un objeto arguments.

```
function f(val) {  
    console.log(arguments); // {0: 1, 1: 2}  
}  
f(1, 2);
```

- En el ejemplo anterior, podemos acceder al primer argumento usando *val* o *arguments[0]*.

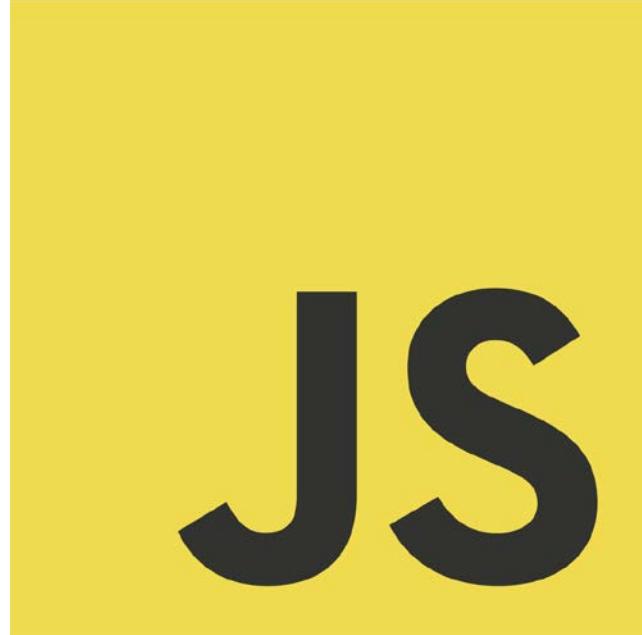
- Los argumentos con nombre son una conveniencia. No son necesarios.
- Nombrar los argumentos no crea ninguna firma o signatura.
- Se puede usar la propiedad *length* del objeto arguments para saber cuántos argumentos ha recibido la función.
- Los espacios de memoria de los argumentos con nombre y los valores del objeto *arguments* son diferentes pero se mantienen sincronizados internamente.
- En *strict mode* no se pueden sobrescribir los valores del objeto arguments directamente.

- En ECMAScript no existe el concepto de sobrecarga.
- Si se definen dos funciones con el mismo nombre, la segunda sobrescribe a la primera.

# Sumario

JS

- Los tipos de dato básicos son *Undefined*, *Null*, *Boolean*, *Number* y *String*.
- El tipo *Number* representa a enteros y coma flotantes por igual.
- El tipo complejo *Object* es la base de todos los objetos del lenguaje.
- *strict mode* incluye cierta restricciones que evitan partes del lenguaje proclives a provocar errores.
- ECMAScript dispone de numerosos operadores y estructuras de control de flujo.
- Las funciones no necesitan especificar el valor de retorno.
- Las funciones que no especifican un valor de retorno, retornan *undefined*.
- Se pueden pasar cualquier número de argumentos a una función. Éstos serán accesibles a través del objeto *arguments*.



東京' TOKIOTA  
Microsoft  
Gold Partner

# JavaScript Desmitificado

Tipos primitivos y de referencia

# Tipos



- JavaScript usa dos clases de tipos:
  - primitivos
  - de referencia
- En ES3 se usa el *variable object* para rastrear las variables de un determinado scope o ámbito.
- En ES5 se usan *lexical environments* para tal fin.
- Los valores primitivos son almacenados directamente en el *variable object*.
- Los valores de referencia son almacenados como un puntero en el *variable object*, apuntando a una ubicación en memoria donde el objeto está almacenado.

- Los tipos primitivos son *Boolean*, *Number*, *String*, *Null* y *Undefined*.
- Cuando asignamos un valor primitivo a una variable, el valor es copiado en la variable.
- Si igualamos una variable con un valor primitivo a otra, cada variable tiene su propia copia del dato.
- Se suele decir que este tipo de variables son accedidas por valor, porque estamos manipulando el valor real almacenado en la variable.

- Los tipos de referencia son estructuras usadas para agrupar datos y funcionalidad.
- También se llaman *Definiciones de Objeto (Object Definitions)*, porque describen las propiedades y métodos que los objetos tendrán.
- Los objetos se consideran *instancias* de un determinado tipo de referencia.
- ECMAScript dispone de varios tipos de referencia nativos: Object, Array, Date, RegExp, Function, Primitive Wrappers, Math, Error...
- Algunos tipos de referencia nativos tienen formas literales (aconsejadas).
- JavaScript no permite trabajar directamente con las ubicaciones en memoria.

- Cuando manipulamos un objeto, realmente trabajamos con una referencia al objeto (almacenado en el *montón – heap*).
- Se suele decir que este tipo de valores son accedidos por referencia.
- Si igualamos una variable con un valor de referencia a otra, cada variable tiene su propia copia de la referencia.

# Tipos

Formas literales

JS

```
var obj = new Object();
obj.prop = 1;
obj.fun = function () { return 2; };
```

```
var obj = {
  prop: 1,
  fun: function () { return 2; }
};
```

```
var arr = new Array();
var arr = [];
```

```
var reg = new RegExp('\\d+');
var reg = /d+/;
```

```
var fun = new Function('return 1;');
var fun = function() { return 1; };
```

- Al trabajar con un valor de referencia, podemos añadir, cambiar o borrar propiedades siempre que queramos.

```
var person = new Object();
person.name = 'Rob';
console.log(name); //Rob
```

- En valores primitivos no funcionará aunque no dará ningún error.

```
var name = 'Rob';
name.age = 38;
console.log(age); //undefined
```

- Hay 3 tipos de tipos de referencia especiales que hacen más fácil la interacción con valores primitivos.
- Estos tipos son creados automáticamente cuando accedemos a una cadena, un número o un booleano.

```
var s1 = 'Rob';
var s2 = s1.substring(2); //b
```

Cuando s1 es accedida ocurre lo siguiente:

1. Se crea una instancia de tipo *String*
2. Se llama al método *substring* de la instancia.
3. Se destruye la instancia.

```
var s1 = new String('Rob');
var s2 = s1.substring(2); //b
s1 = null;
```

Podemos pensar los 3 pasos anteriores así también.

Global

JS

- El objeto Global es inaccesible en ECMAScript.
- Toda variable o función definida globalmente es una propiedad del objeto Global.
- Hay varios métodos existentes en el objeto Global, como isNaN(), isFinite(), parseInt(), eval(), encodeURI()...

| PROPERTY                    | DESCRIPTION                                 | PROPERTY              | DESCRIPTION                           |
|-----------------------------|---------------------------------------------|-----------------------|---------------------------------------|
| <code>undefined</code>      | The special value <code>undefined</code>    | <code>Object</code>   | Constructor for <code>Object</code>   |
| <code>NaN</code>            | The special value <code>NaN</code>          | <code>Array</code>    | Constructor for <code>Array</code>    |
| <code>Infinity</code>       | The special value <code>Infinity</code>     | <code>Function</code> | Constructor for <code>Function</code> |
| <code>EvalError</code>      | Constructor for <code>EvalError</code>      | <code>Boolean</code>  | Constructor for <code>Boolean</code>  |
| <code>RangeError</code>     | Constructor for <code>RangeError</code>     | <code>String</code>   | Constructor for <code>String</code>   |
| <code>ReferenceError</code> | Constructor for <code>ReferenceError</code> | <code>Number</code>   | Constructor for <code>Number</code>   |
| <code>SyntaxError</code>    | Constructor for <code>SyntaxError</code>    | <code>Date</code>     | Constructor for <code>Date</code>     |
| <code>TypeError</code>      | Constructor for <code>TypeError</code>      | <code>RegExp</code>   | Constructor for <code>RegExp</code>   |
| <code>URIError</code>       | Constructor for <code>URIError</code>       | <code>Error</code>    | Constructor for <code>Error</code>    |

- eval() es el método más potente en ECMAScript.
- eval() actúa como un interprete de ECMAScript sobre un argumento de tipo cadena.

```
eval('console.log("hola")'); //hola
```

- Aunque ECMA-262 no indica como acceder al objeto Global, los navegadores lo implementan haciendo que *window* sea el delegado del objeto Global.
- Todas las variables y funciones declaradas en el scope global se convierten en propiedades de *window*.

```
var game= 'pang';
function getGame() {
    console.log(window.game);
}
window.getGame(); //pang
```

# Pasando argumentos

A yellow square containing the letters "JS" in a bold, black, sans-serif font.

JS

# Pasando argumentos

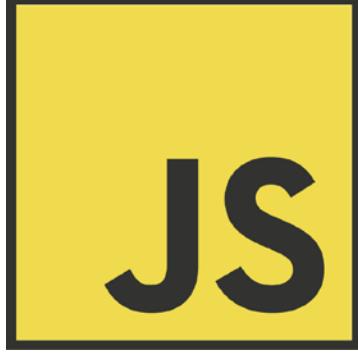
JS

- Todos los argumentos en ECMAScript son pasados por valor.
- El valor de fuera de la función es copiado en el argumento dentro de la función.
- Las variables se pueden acceder por valor o por referencia pero los argumentos siempre se pasan por valor.

```
function setName(person){  
    person.name = 'Rob';  
}  
  
var person = new Object();  
setName(person);  
console.log(person.name); //Rob
```

```
function setName(person){  
    person = new Object();  
    person.name = 'Rob';  
}  
  
var person = new Object();  
setName(person);  
console.log(person.name); //undefined
```

# Contexto y Scope



JS

- El concepto de *contexto de ejecución* o *contexto* es de máxima importancia en JavaScript.
- El *contexto* de una variable o función define a qué otros datos tiene acceso.
- Cada *contexto* tiene asociado un *variable object* donde existen todas las variables y funciones del mismo.
- Este objeto no es accesible y es usado para manejar los datos.
- El *contexto global* es el *contexto* más externo.
- Cuando un *contexto* ejecuta todo su código, se destruye, junto con todas las variables y funciones definidas en él.
- El *contexto global* sólo se destruye al salir de la aplicación.

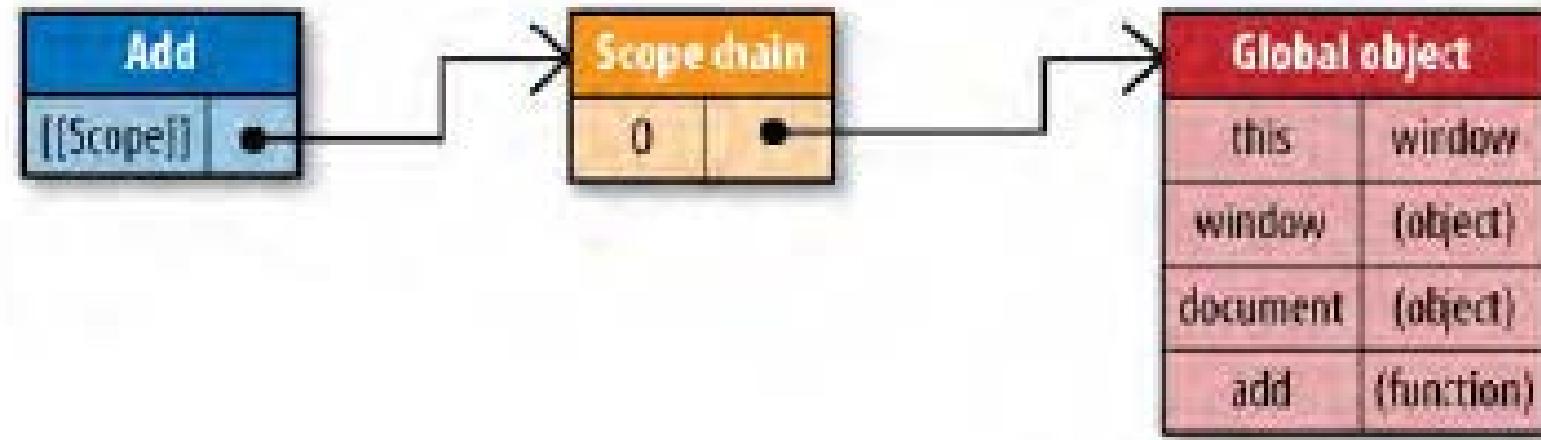
- Cada llamada a una función tiene su propio contexto de ejecución que se inserta en una pila de contextos.
- Una vez se ejecuta el código de la función se retorna al anterior contexto y el contexto de la función es destruido.

- Cuando se ejecuta el código de un contexto, se crea un *scope chain* o una *cadena de scope*.
- El *scope chain* es una colección con todos los *variable objects* a los que tiene acceso un determinado *contexto de ejecución*.
- El primer *variable object* del *scope chain* es siempre el del contexto cuyo código se está ejecutando.
- Si el contexto es una función, el *activation object* es usado como *variable object*.
- El *activation object* se inicia con una sola variable llamada *arguments*.
- El siguiente *variable object* en el *scope chain* es el *contexto contenedor*.
- Este patrón se repite hasta llegar al contexto global.

- Los identificadores (variables y funciones) se resuelven navegando el scope chain en busca de sus nombres.
- Esta navegación empieza de dentro hacia fuera.
- Si existen dos variables con el mismo nombre se usará la primera que se encuentre en este proceso de búsqueda o navegación.
- Veamos un ejemplo.

```
function add(num1, num2) {  
    var sum = num1 + num2;  
    return sum;  
}
```

- Scope chain de la función `add()`

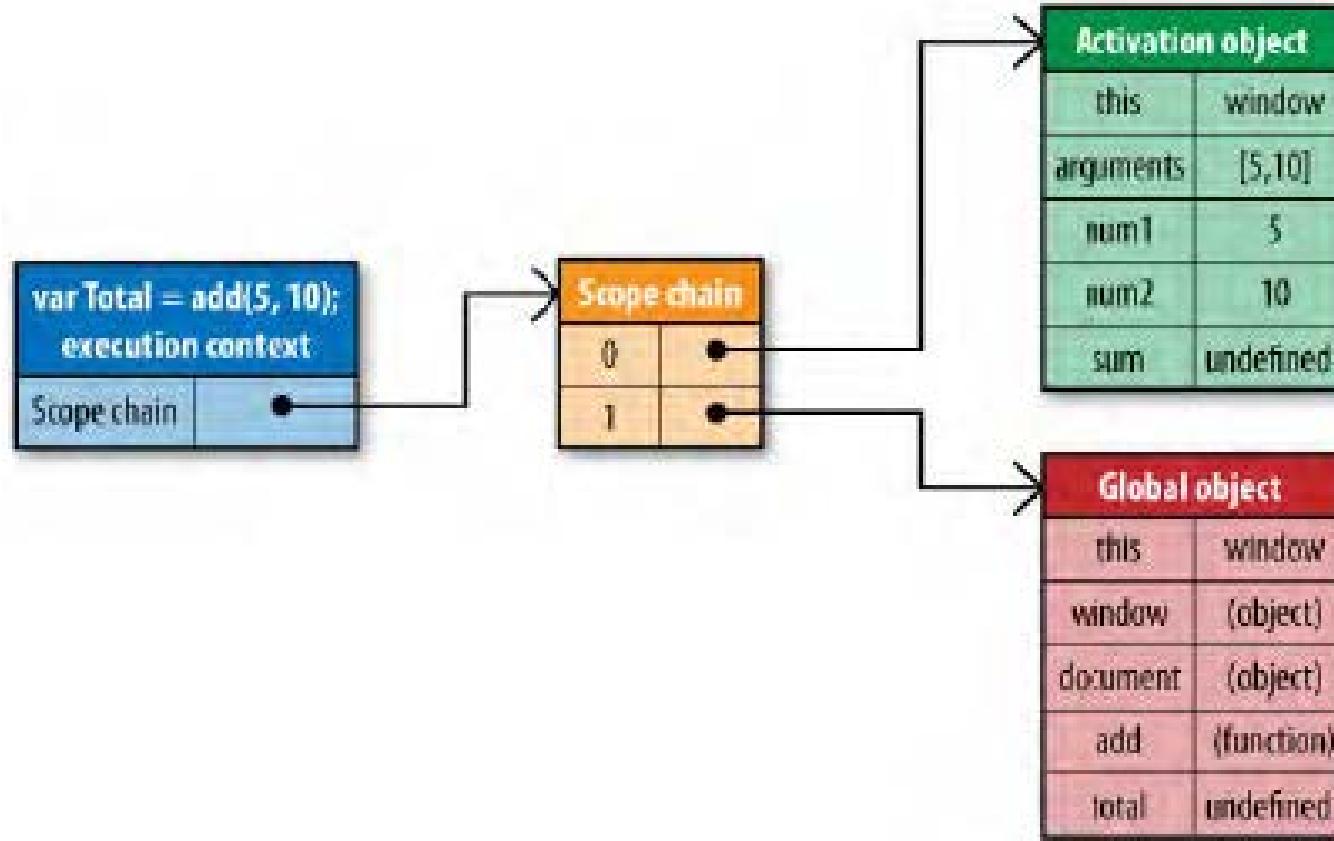


# Contexto y Scope

Scope

JS

- Scope chain de la función `add()` al ser ejecutada

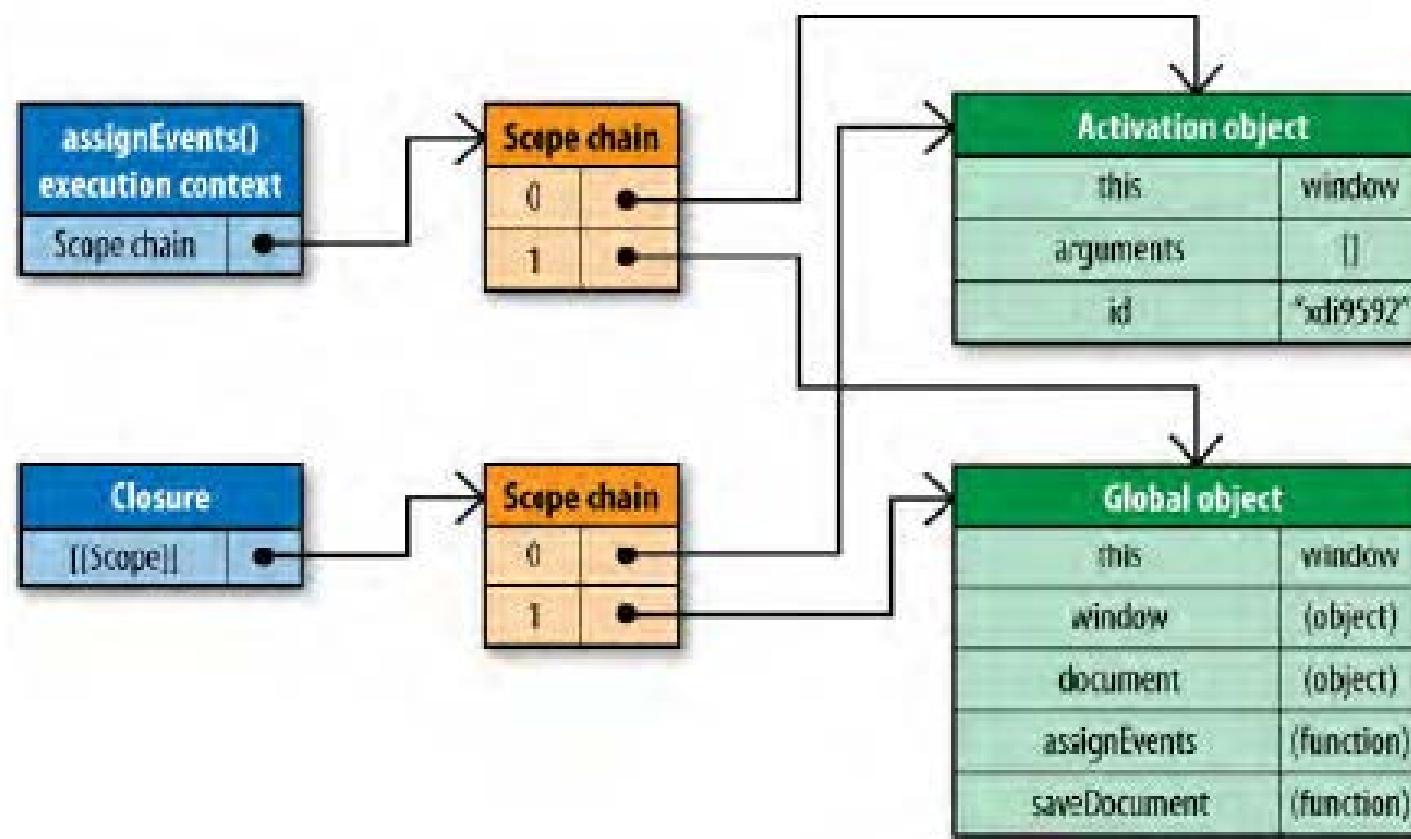


- A nivel de performance acceder a variables fuera de contexto tiene un coste.
- Cuanto más fuera se encuentre un identificador más lenta es su lectura.
- El acceso a las variables locales es siempre más rápido.
- El uso de *with* y *try catch* aumenta el *scope chain*.
- Las anteriores sentencias junto a *eval()* se consideran *scopes dinámicos*.
- Un scope dinámico es aquel que sólo puede ser determinado en tiempo de ejecución.

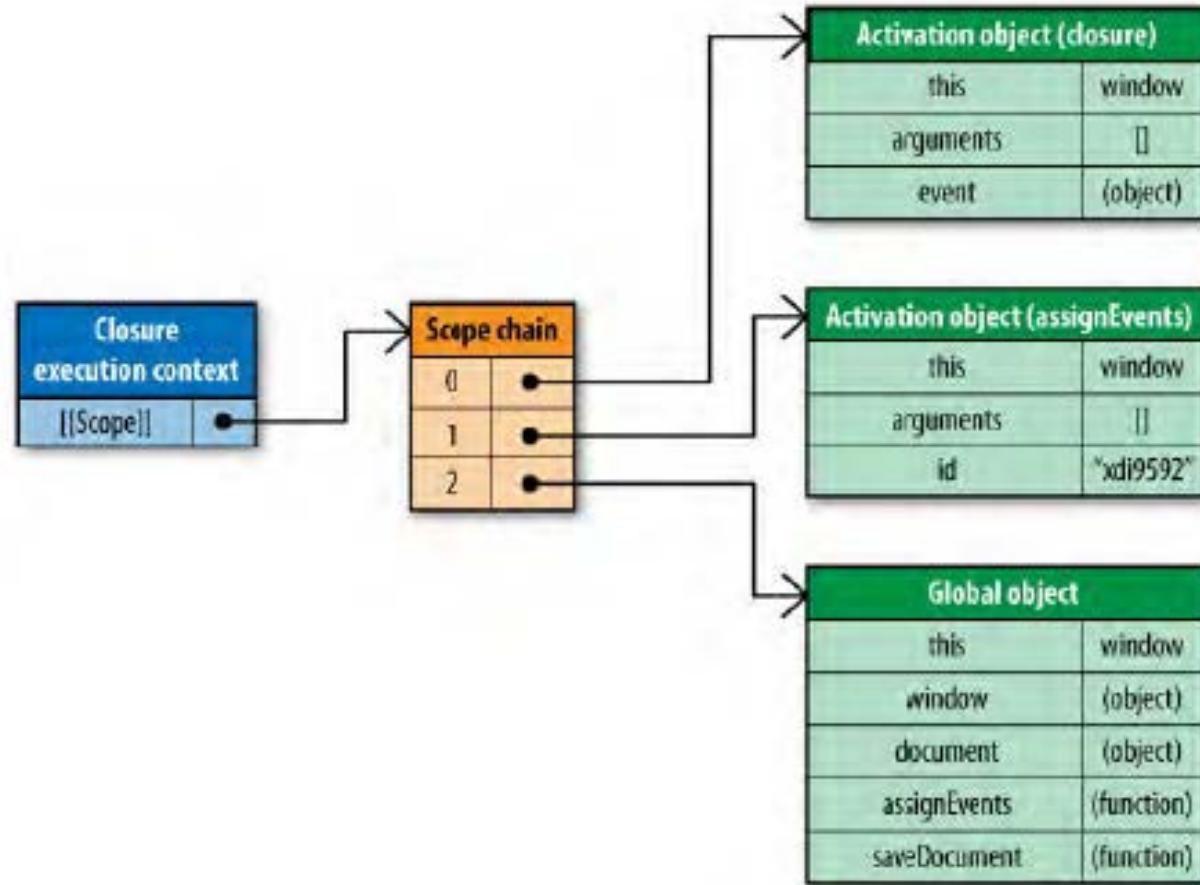
- Las *closures* son uno de los aspectos más potentes de JavaScript.
- Permiten a una función acceder a datos ajenos a su *scope local*.
- Tenemos que ser conscientes del impacto a nivel de performance que pueden tener, a nivel de memoria y velocidad de ejecución.
- Ejemplo a continuación:

```
function assignEvents() {  
    var id = 'xdi9592';  
    document.getElementById('save-btn').onclick = function (event) {  
        saveDocument(id);  
    };  
}
```

- Scope chain de la función *assignEvents()* al ser ejecutada y de la closure.



- Scope chain ejecutando la *closure*.



# Contexto y Scope

¿Block Scope?

JS

- En JavaScript los bloques no tienen su propio contexto de ejecución.

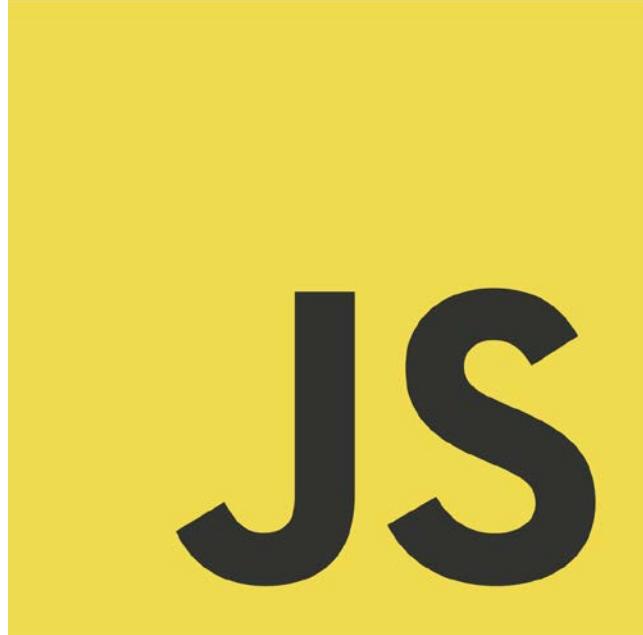
```
if (true) {  
    var game = 'bomb jack';  
}  
console.log(game); //bomb jack
```

- Ojo con los bucles for

```
for (var i = 0; i < 10; i++) {  
    console.log(i);  
}  
console.log(i); //10
```

- Las declaraciones de variables y funciones son procesadas antes de que el código se ejecute.
- Esto implica que son aupadas al inicio del contexto.

```
var varOut = 1;
function hello(){
    console.log(varOut); //undefined
    var varOut;
    varOut = 2;
    console.log(varOut); //2
}
hello();
```



# JavaScript Desmitificado

Funciones

東京' TOKIOTA



# El tipo Function

A yellow square containing the letters "JS" in a bold, black, sans-serif font.

JS

# El tipo Function

JS

- Las funciones en ECMAScript son objetos.
- Son instancias del tipo *Function*, que tienen propiedades y métodos como cualquier otro de los tipos de referencia.
- Los nombres de las funciones son punteros a los objetos función.

Podemos definir funciones de 3 maneras diferentes:

- Function declaration:

```
function helloWorld(name){  
    console.log('hello world ' + name);  
}
```

- Function expression:

```
var helloWorld = function(name){  
    console.log('hello world' + name);  
};
```

- Function constructor:

```
var helloWorld = new Function('name', 'console.log("hello world " + name)');
```

- Es importante resaltar el punto y coma final en la *function expression*, ya que se trata de una inicialización de variable.
- El uso del constructor Function está desaconsejado ya que se produce una doble interpretación.

# Declarations vs Expressions

JS

# Declarations vs Expressions

Hoisting

JS

- Function Declarations y Function Expressions se diferencian en un concepto capital: el *hoisting*.
- Las *Function Declarations* son aupadas al inicio del *contexto de ejecución*.
- Las *Function Declarations* son leídas y están disponibles en un *contexto* antes de que este se ejecute.
- Las *Function Expressions* no están disponibles hasta que la ejecución alcanza la línea donde están declaradas.

```
helloWorld('John'); //hello world John
function helloWorld(name){
  console.log('hello world ' + name);
}
```

```
helloWorld('John'); //error
var helloWorld = function(){
  console.log('hello world ' + name);
}
```

- Es posible usar un nombre en las Function Expressions.
- Este nombre sólo está disponible en el scope de la función.
- Facilitan la experiencia de depuración cuando tenemos escenarios complejos.
- Permiten llamadas recursivas.
- Presentan problemas en IE8 y versiones antiguas de Safari.

<https://kangax.github.io/nfe/>

```
var f = function foo() {  
    return typeof foo; // foo está disponible aquí  
};
```

```
typeof foo; // "undefined"  
f(); // "function"
```

# Funciones como valor

A yellow square containing the letters "JS" in a bold, black, sans-serif font.

JS

# Funciones como valor

JS

- Las funciones se pueden usar como cualquier otro valor.
- Se pueden asignar a variables, añadir a objetos, pasar como argumento a otra función y retornar desde otras funciones.
- Para ejecutar una función añadimos () a su nombre o el de la variable correspondiente.
- Si no añadimos () estamos asignando directamente la referencia.

```
var f1 = function () { return 'hi'; };
var f2 = f1; //puntero a la función
var f3 = f1(); //hi
```

this

JS

- *this* es una referencia al contexto del objeto sobre el que opera una función.

```
var game = 'double dragon';
```

```
var o = { game: 'shinobi' };
```

```
function getGame(){  
    console.log(this.game);  
}
```

```
getGame(); //double dragon
```

```
o.getGame = getGame;  
o.getGame(); //shinobi
```

- El valor de *this* no se determina hasta que la función es llamada.
- Cuando una función es llamada como método de un objeto, *this* equivale al objeto.
- La habilidad de manipular el valor de *this* en las funciones es clave para programar orientado a objetos en JavaScript.
- Existen 3 métodos que nos permitirán cambiar el valor de *this*:
  - *call()*
  - *apply()*
  - *bind()*

- Este método ejecuta una función con un valor particular de *this* y unos parámetros específicos.
- El primer parámetro es el valor de *this* que queremos que tenga la función cuando se ejecute.
- Los parámetros subsiguientes son los parámetros que se pasarán a la función.

this

call()

JS

```
var title = 'Super Sprint';
function getTitle(caller) {
    console.log(caller + ':' + this.title);
}
var game1 = {
    title: 'Pole Position'
};
var game2 = {
    title: 'Arkanoid'
};
```

```
getTitle.call(game1, 'game1');
// game1 : Pole Position
```

```
getTitle.call(game2, 'game2');
// game2 : Arkanoid
```

```
getTitle.call(this, 'Global');
//Global : Super Sprint
```

//En strict mode => Ojo con el this, si llamamos getTitle() directamente, en vez de estar asignado al objeto global, será undefined.

- Este método funciona igual que *call()* pero sólo acepta dos parámetros.
- El primer parámetro es el valor de *this* que queremos que tenga la función cuando se ejecute.
- El segundo parámetro es un *array* con los parámetros que le pasaremos a la función.
- Esta característica es clave para poder usar el objeto *arguments*.

this

apply()

JS

```
var title = 'Super Sprint';
function getTitle(caller) {
    console.log(caller + ':' + this.title);
}
var game1 = {
    title: 'Pole Position'
};
var game2 = {
    title: 'Arkanoid'
};
```

```
getTitle.apply(game1, ['game1']);
// game1 : Pole Position
```

```
getTitle. apply(game2, ['game2']);
// game2 : Arkanoid
```

```
getTitle. apply(this, ['Global']);
//Global : Super Sprint
```

//En strict mode => Ojo con el this, si llamamos getTitle() directamente, en vez de estar asignado al objeto global, será undefined.

this

apply()

JS

```
function getTitle(caller) {  
    console.log(caller + ':' + this.title);  
}  
  
function superGamer(caller) {  
    this.title = 'Phoenix';  
    getTitle.appy(this, arguments);  
}  
  
superGamer('SuperGamer'); //SuperGamer : Phoenix
```

- Este método sólo está disponible en ECMAScript 5.
- El método *bind()* crea una nueva función cuyo valor de *this* está *enlazado* al que nosotros le pasemos por parámetro.
- El primer parámetro es el valor de *this* que queremos *enlazar*.
- El resto de parámetros se pasarán a la función creada y también quedarán *enlazados*.
- La función resultante no permitirá el cambio del valor de *this*\*

\* [Atención al uso de new con una función enlazada.](#)

# this

apply()

JS

```
function getTitle(caller) {  
    console.log(caller + ':' + this.title);  
}  
  
var game1 = {  
    title: 'Pole Position'  
};  
  
var game2 = {  
    title: 'Arkanoid'  
};
```

```
var getGame1Title =  
getTitle.bind(game1);  
  
var getGame2Title =  
getTitle.bind(game2, 'game2');  
  
getGame1Title('game1');  
// game1 : Pole Position  
getGame1Title('game2');  
// game2 : Arkanoid
```

# Prototype

A yellow square containing the letters "JS" in a bold, black, sans-serif font.

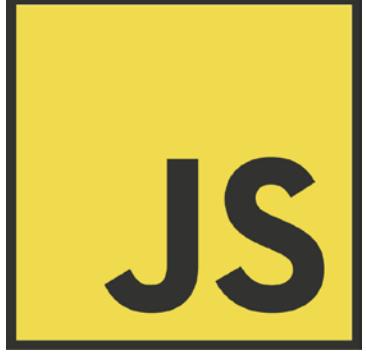
JS

# Prototype

JS

- Toda función tiene una propiedad llamada *prototype*.
- Aquí residen todos los métodos de instancia de un objeto (*toString()*, *valueOf()*).
- Esta propiedad es fundamental a la hora de definir nuestros propios tipos de referencia y para la herencia.
- En ES5 (ECMAScript 5) esta propiedad no es enumerable, por lo que no aparecerá en un for-in.

# Function Expressions

A yellow square containing the letters "JS" in a bold, black, sans-serif font.

JS

# Function Expressions

Funciones anónimas

JS

```
var helloWorld = function(name){  
    console.log('hello world' + name);  
};
```

- Creamos una función y la asignamos a la variable *helloWorld*.
- La función creada es considerada como una *función anónima*, ya que no tiene *identificador* trás la palabra clave *function*.
- Las *funciones anónimas* a veces se llaman *funciones lambdas*.
- Recordemos que el *hoisting* sólo afecta a las *Function Declarations*.

# Function Expressions

Funciones anónimas

JS

```
var helloWorld = function(name){  
    console.log('hello world' + name);  
};
```

- Creamos una función y la asignamos a la variable *helloWorld*.
- La función creada es considerada como una *función anónima*, ya que no tiene *identificador* trás la palabra clave *function*.
- Las *funciones anónimas* a veces se llaman *funciones lambdas*.
- Recordemos que el *hoisting* sólo afecta a las *Function Declarations*.

# Function Expressions

Funciones anónimas

JS

```
if (true) {  
    function getGame() {  
        return ('Livingstone supongo');  
    }  
} else {  
    function getGame() {  
        return ('Joust');  
    }  
}  
getGame();  
//el resultado variará en función del navegador  
//NO HACER ESTO
```

```
var getGame;  
if (true) {  
    getGame = function () {  
        return ('Livingstone supongo');  
    };  
} else {  
    getGame = function () {  
        return ('Joust');  
    };  
}  
getGame(); //Livingstone supongo
```

- El término *closure* y *función anónima* a veces se usan incorrectamente como sinónimos.
- Recordemos que una *closure* es una función que tiene acceso a variables del *scope* de otra función.
- Esto se logra habitualmente anidando funciones.

# Function Expressions

Closures

JS

```
function createComparisonFunction(propertyName) {
    return function (object1, object2) {
        //uso de la variable propertyName perteneciente a la función contenedora
        var value1 = object1[propertyName];
        var value2 = object2[propertyName];
        if (value1 < value2) {
            return -1;
        } else if (value1 > value2) {
            return 1;
        } else {
            return 0;
        }
    };
}

var compare = createComparisonFunction('title');
var result = compare({ title: 'Rampage' }, { title: 'Out Run' });
compare = null; //el activation object de createComparisonFunction podrá ser destruido por el GC.
```

- IIFE = Immediately-Invoked Function Expressions.
- El término fue acuñado por [Ben Alman](#).

```
(function(){ /*code here*/ })();
```

```
(function($){ /*code here*/ })(jQuery);
```

# Function Expressions

Closures y variables

JS

```
function createFunctions() {  
    var result = [];  
    for (var i = 0; i < 10; i++) {  
        result[i] = function () {  
            return i; //siempre 10  
        };  
    }  
    return result;  
}
```

```
function createFunctions() {  
    var result = [];  
    for (var i = 0; i < 10; i++) {  
        result[i] = (function (num) {  
            return function(){  
                return num;  
            };  
        })(i);  
    }  
    return result;  
}
```

# Function Expressions

Closures y this

JS

```
var title = 'Bubble Bobble';
var game = {
  title: '1942',
  getTitle: function(){
    return function(){
      return this.title;
    };
  };
}
game.getTitle(); // Bubble Booble
```

```
var title = 'Bubble Bobble';
var game = {
  title: '1942',
  getTitle: function(){
    var self = this;
    return function(){
      return self.title;
    };
  };
}
game.getTitle(); // 1942
```

# Function Expressions

Fugas de memoria

JS

```
function assignHandler() {  
    var element = document.getElementById('element');  
    element.onclick = function () {  
        console.log(element.id); //el elemento no podrá ser destruído  
    };  
}
```

Bug en JScript afectando a IE <= 8

```
function assignHandler() {  
    var element = document.getElementById('element'),  
        id = element.id;  
    element.onclick = function () {  
        console.log(id);  
    };  
    element = null;  
}
```

# Function Expressions

Block Scope

JS

- En ECMAScript no existe scope a nivel de bloque.
- Podemos usar las IIFEs para emularlo.

```
function outputNumbers(count) {  
  (function () {  
    for (var i = 0; i < count; i++) {  
      console.log(i);  
    }  
  })();  
  console.log(i); //error, i sólo está disponible dentro de la IIFE.  
}
```

# Function Expressions

Variables privadas

JS

- En JavaScript no existen los miembros privados, todas las propiedades de un objeto son públicas.
- Las variables definidas dentro de una función son consideradas como privadas ya que no pueden accederse desde fuera de la función.

```
function Game(){  
  var title = 'Donkey Kong';  
  this.getGame = function (){  
    return title;  
  };  
}  
  
var g = new Game();  
g.getGame(); //Donkey Kong
```

```
function Game(title){  
  this.getGame = function (){  
    return title;  
  };  
}  
  
var g = new Game('Donkey Kong');  
g.getGame(); //Donkey Kong
```

# Function Expressions

Variables privadas

JS

```
(function (w){  
    var title = "";  
    w.Game = function (value){  
        title = value;  
    };  
    w.Game.prototype.getTitle = function (){  
        return title;  
    };  
    w.Game.prototype.setTitle = function (value){  
        title = value;  
    };  
})(window);
```

```
var game = new Game('Who dares wins');  
game.getTitle(); //Who dares wins  
game.setTitle('Cabal');  
game.getTitle(); //Cabal
```

```
var game2 = new Game('Ninja Turtles');  
game.getTitle(); //Ninja Turtles  
game2.getTitle(); //Ninja Turtles
```

En este ejemplo la variable *title* se convierte en una variable estática privada.

# Module Pattern



- Son objetos únicos a nivel de instancia.
- En JavaScript se suele usar *notación literal* para crearlos.

```
var singleton = {  
    property: 1,  
    method: function(){  
        /*code here*/  
    }  
};
```

# Module Pattern

Module Pattern

JS

- El *patrón Módulo* aumenta el *Singleton* básico permitiendo el uso de variables privadas y métodos privilegiados.
- Nos ayudaremos de una IIFE.
- Esta función anónima acaba devolviendo un *objeto literal* con métodos privilegiados.
- El *objeto literal* define la interfaz pública del *singletón*.

```
var singleton = (function (){  
    var privateVar = 10;  
    function privateFunc(){  
        return privateVar;  
    }  
    return {  
        publicProp: true,  
        publicMethod: function (){  
            privateVar++;  
            return privateFunc();  
        }  
    };  
}());
```

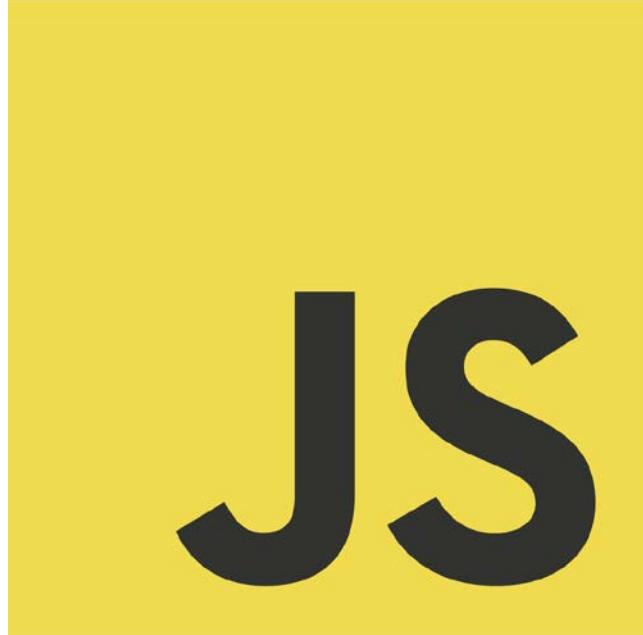
# Module Pattern

## Revealing Module Pattern

JS

- Es una pequeña variación del patrón módulo que define todos los miembros como privados.
- El objeto literal retornado expone referencias a ciertos miembros privados.

```
var singleton = (function (){  
    var privateVar = 10;  
    function privateFunc(){  
        privateVar++;  
        return privateVar;  
    }  
    return {  
        publicProp: privateVar,  
        publicMethod: privateFunc  
    };  
})();
```



# JavaScript Desmitificado

Objetos

東京' TOKIOTA



# Objetos

JS

- Según el ECMA-262 se define objeto como: “*an unordered collection of properties each of which contains a primitive value, object, or function.*”
- Un objeto es una especie de *array* de valores desordenados.
- Cada propiedad o método es identificado por un nombre mapeado a un valor.
- Ayuda pensar en los objetos como si fueran *Hash Tables*, un grupo de pares clave-valor en donde el valor puede ser un dato o una función.
- Como hemos visto anteriormente se pueden crear usando formato literal o mediante el constructor Object.

- En ES5 se ofrece la posibilidad de crear propiedades teniendo un mayor grado de control.
- Existen dos tipos de propiedades:
  - *Data properties*
  - *Accesor properties*

- Contienen una ubicación para almacenar el valor del dato.
- El valor se lee y se escribe directamente en esta ubicación.
- Estas propiedades tienen 4 atributos que describen su comportamiento:
  - **Configurable**\*: Indica si la propiedad se puede borrar (usando *delete*), cambiar sus atributos o convertirla en una *accesor property*.
  - **Enumerable**\*: Indica si se retorna en un bucle *for-in*.
  - **Writable**\*: Indica si el valor se puede alterar.
  - **Value**: Contiene el valor de la propiedad. Por defecto es *undefined*.

\**false* por defecto, pero *true* para las propiedades definidas directamente (*o.prop = 123*).

- Para cambiar o añadir atributos a una propiedad podemos usar el método *Object.defineProperty()*.
- Acepta 3 argumentos:
  - El objeto cuya propiedad vamos a añadir o modificar.
  - El nombre de la propiedad a añadir o modificar.
  - Un objeto descriptor con las siguientes propiedades:
    - configurable
    - enumerable
    - writable
    - value

```
var book = {};
```

```
Object.defineProperty(book, 'name', {  
    configurable: true, //default false  
    enumerable: false, //default true  
    writable: true, //default false  
    value: 'Ulises'  
});
```

- No contienen ningún valor de dato.
- Pueden contener una función *getter* y/o una función *setter*.
- Estas propiedades tienen 4 atributos:
  - **Configurable**\*: Indica si la propiedad se puede borrar (usando *delete*), cambiar sus atributos o convertirla en una *accesor property*.
  - **Enumerable**\*: Indica si se retorna en un bucle *for-in*.
  - **Get**: Función que se llama en la lectura de la propiedad. Por defecto *undefined*.
  - **Set**: Función que se llama en la escritura de la propiedad. Por defecto *undefined*.

\**false* por defecto, pero *true* para las propiedades definidas directamente (*o.prop = 123*).

# Objetos

Accesor properties

JS

```
var book = {  
  __year: 2015  
};
```

```
Object.defineProperty(book, 'year', {  
  get: function () {  
    return 'year' + this.__year;  
  },  
  set: function (newValue) {  
    this.__year = newValue;  
  }  
});
```

- Mediante el método *Object.defineProperties()* podemos definir múltiples propiedades de una sola vez.
- Este método tiene dos argumentos:
  - El objeto cuyas propiedades vamos a añadir o modificar.
  - Un objeto cuyas propiedades se corresponden a los que queremos añadir o modificar en el objeto del primer argumento.

# Objetos

Múltiples propiedades

JS

```
var book = {};  
Object.defineProperty(book, {  
  __year: {  
    value: 2004  
  },  
  year: {  
    get: function () {  
      return 'year' + this.__year;  
    },  
    set: function (newValue) {  
      this.__year = newValue;  
    }  
  }  
});
```

- Mediante el método *Object.getOwnPropertyDescriptor()* podemos obtener información sobre los atributos aplicados a una propiedad.
- Este método tiene dos argumentos:
  - El objeto cuyas propiedades queremos observar.
  - El nombre de la propiedad a observar.
- Retorna un objeto con las propiedades *configurable*, *enumerable*, *get*, *set*, *writable* y *value*.

```
var descriptor = Object.getOwnPropertyDescriptor(book, '_year');
console.log(descriptor.value); //2004
console.log(descriptor.configurable); //false
console.log(typeof descriptor.get); //'undefined'
```

```
var descriptor = Object.getOwnPropertyDescriptor(book, 'year');
console.log(descriptor.value); //undefined
console.log(descriptor.enumerable); //false
console.log(typeof descriptor.get); //'function'
```

- Incorrecto, si `book.year` vale *0, false, null, undefined* o `""` no accederá al bloque.

```
if (book.year){  
    /*code here */  
}
```

- Correcto, busca propiedades de instancia y en la cadena de prototipos.

```
if ('year' in book){  
    /*code here */  
}
```

- Correcto, busca propiedades de instancia.

```
if (book.hasOwnProperty('year')){  
    /*code here */  
}
```

```
delete book.year
```

- Los objetos, igual que sus propiedades, tienen atributos que definen su comportamiento.
- Hay varias maneras de prevenir la modificación de un objeto:
  - `Object.preventExtensions(obj)`, no nos permite añadir propiedades.
  - `Object.seal(obj)`, no permite añadir ni modificar las propiedades (no son configurables).
  - `Object.freeze(obj)`, no permite añadir ni modificar las propiedades ni su valor (no configurables ni writable).
- También disponemos de métodos para comprobar el estado del objeto:
  - `Object.isExtensible(obj)`
  - `Object.isSealed(obj)`
  - `Object.isFrozen(obj)`

# Creación de Objetos

JS

```
function createGame(title, year){  
    var o = new Object();  
    o.title= title;  
    o.year = year;  
    o.getTitle = function(){  
        return this.title;  
    }  
    return o;  
}  
  
var game = createGame('Karate Chan', 1988);
```

- Este patrón no nos permite saber qué tipo de objeto es el objeto game.

# Creación de objetos

Constructor Pattern

JS

```
function Game(title, year){  
    this.title = title;  
    this.year = year;  
    this.getTitle = function(){  
        return this.title;  
    };  
}  
  
var game = createGame('Karate Chan', 1988);  
console.log(game instanceof Game); //true  
console.log(game instanceof Object); //true
```

- Los constructores, por convención, empiezan con mayúsculas.
- Observemos que:
  - No se crea ningún objeto explícitamente.
  - Las propiedades y métodos se asignan directamente al objeto `this`.
  - No hay sentencia de retorno.
- Para crear una nueva instancia usamos el operador `new`, lo que provoca que:
  - Se cree un nuevo objeto.
  - Se asigna el valor de `this` al nuevo objeto.
  - Se ejecuta el código dentro del constructor.
  - Se retorna el nuevo objeto.

- Cualquier función puede ser llamada con el operador *new*, actuando así como un constructor.
- Cualquier constructor puede ser llamado sin el operador *new*. Esto provocará que sus miembros sean añadidos al objeto global (si no estamos en *strict mode*).
- También podemos usar *call()* o *apply()* para que los miembros se añadan a un objeto concreto.
- El problema con los constructores es que los métodos se crean cada vez por cada instancia. No hacen referencia a la misma función (objeto).

```
var game1 = new Game('Galaxian', 1988);
var game2 = new Game('Froggy', 1989);
console.log(game1.getTitle === game2.getTitle); //false
```

# Creación de objetos

Prototype Pattern

JS

```
function Game() { }

Game.prototype.title = 'Galaxian';
Game.prototype.year = 1988;
Game.prototype.getTitle = function () {
    return this.title;
};

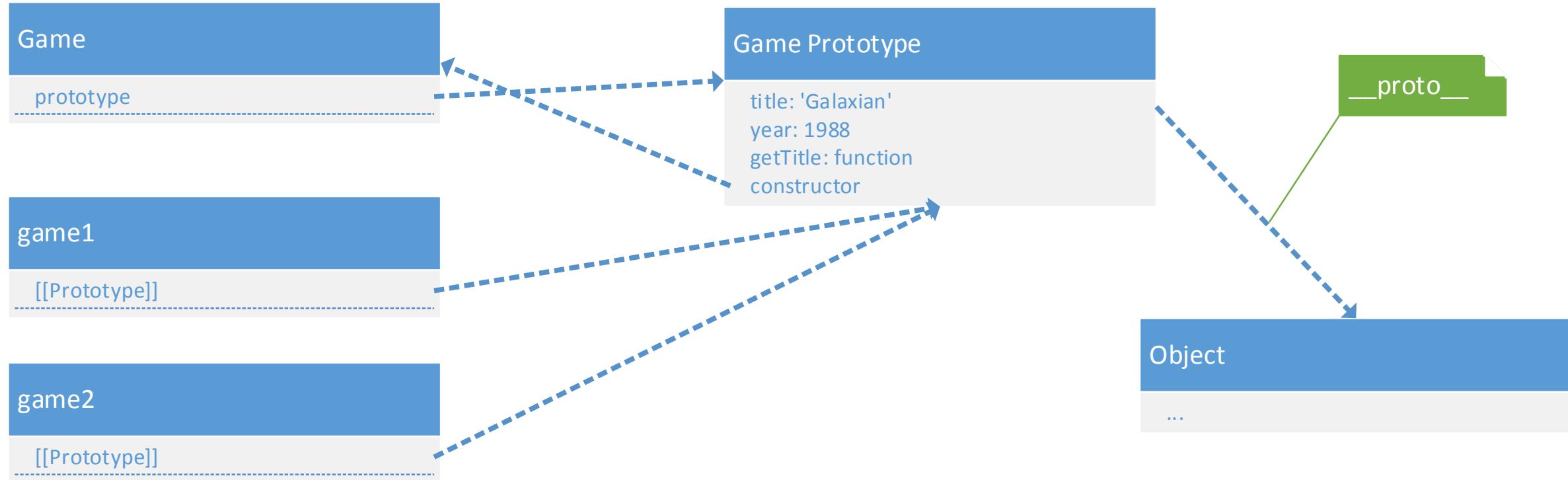
var game1 = new Game();
var game2 = new Game();
console.log(game1.getTitle === game2.getTitle); //true
```

- Toda función tiene una propiedad *prototype* que contiene un objeto con sus propiedades y métodos y que está disponible para todas las instancias de un tipo de referencia particular.
- Todos los miembros del *prototype* son compartidos por las instancias.
- Al crear una función, su *prototype*, automáticamente, obtiene una propiedad *constructor* que apunta a si misma (a la función).
- Algunos navegadores dan acceso al objeto *prototype* mediante la propiedad no estándar *\_\_proto\_\_*.

# Creación de objetos

Prototype Pattern

JS



- Cada vez que se accede a alguna propiedad de un objeto, empieza una búsqueda.
- Primero se busca en las propiedades de la instancia, luego en su *prototype* y así hasta llegar al objeto *Object*.
- Esta cadena se conoce como *Prototype Chain*.
- Podemos leer valores del *prototype* desde las instancias pero no modificarlos.
- Si añadimos una propiedad con el mismo nombre que una del *prototype*, creamos una nueva propiedad de instancia, enmascarando la del *prototype*.
- El método *hasOwnProperty()* determina si la propiedad es de la instancia o de su *prototype*.

- El método *Object.keys(obj)* obtiene todos los nombres de propiedades enumerables y de instancia de un objeto concreto.
- Sintaxis alternativa para el Prototype Pattern:

```
function Game() {}  
Game.prototype = {  
    constructor: Game, //makes this enumerable!! use Object.defineProperty if ES5  
    title: 'Galaxian',  
    year: 1988,  
    getTitle: function () {  
        return this.title;  
    }  
};
```

- Uno de los problemas del Prototype Pattern es que no podemos pasar argumentos de inicialización al constructor.
- La naturaleza compartida de sus propiedades es un gran problema cuando nos encontramos con valores de referencia.

```
function Obj() {}  
Person.prototype.items = [1,2,3];  
var o1 = new Obj();  
var o2 = new Obj();  
o1.items.push(4);  
console.log(o2.items); //1,2,3,4
```

```
function Game(title, year) {  
    this.title = title;  
    this.year = year;  
}
```

```
Game.prototype.getTitle = function () {  
    return this.title;  
};
```

- Este es el patron más usado para definir tipos de referencia.
- Se comparten los métodos, conservando memoria.
- Permite usar argumentos en el constructor.

```
function Game(title, year) {  
    this.title = title;  
    this.year = year;  
    if (typeof this.getTitle != 'function') {  
        Game.prototype.getTitle = function() {  
            return this.title;  
        };  
    }  
}
```

- Los métodos del *prototype* se definen sólo al crear la primera instancia.
- Aquí no podemos sobrescribir el *prototype* con un objeto literal.

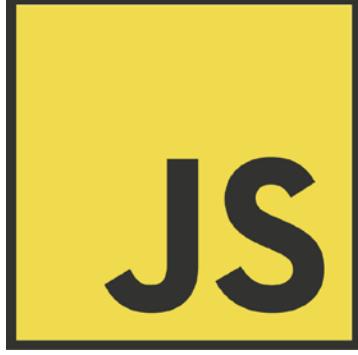
```
function SpecialArray() {  
    var values = new Array(); //create the array  
    values.push.apply(values, arguments); //add the values  
    values.toPipedString = function () {  
        return this.join('|');  
    };  
    return values; //return it  
}  
  
var colors = new SpecialArray('red', 'blue', 'green');  
console.log(colors.toPipedString()); //red|blue|green'
```

- Igual que el Factory Pattern pero usando el operador new.
- Nos permite crear constructores que no podríamos crear de otra manera.
- Este patrón debería ser evitado a ser posible.

- Un *objeto durable* es aquel que carece de propiedades públicas y cuyos métodos no hacen referencia a *this*.
- Se usan en entornos seguros que no permiten el uso de *this* o *new*.
- Este patrón es similar al Factory Pattern pero genera *objetos durables*.

```
function Game(title, year) {  
    var o = new Object();  
    o.getTitle = function () {  
        return title;  
    };  
    return o;  
}  
var game = Game('Karate Chan', 1988);
```

# Herencia

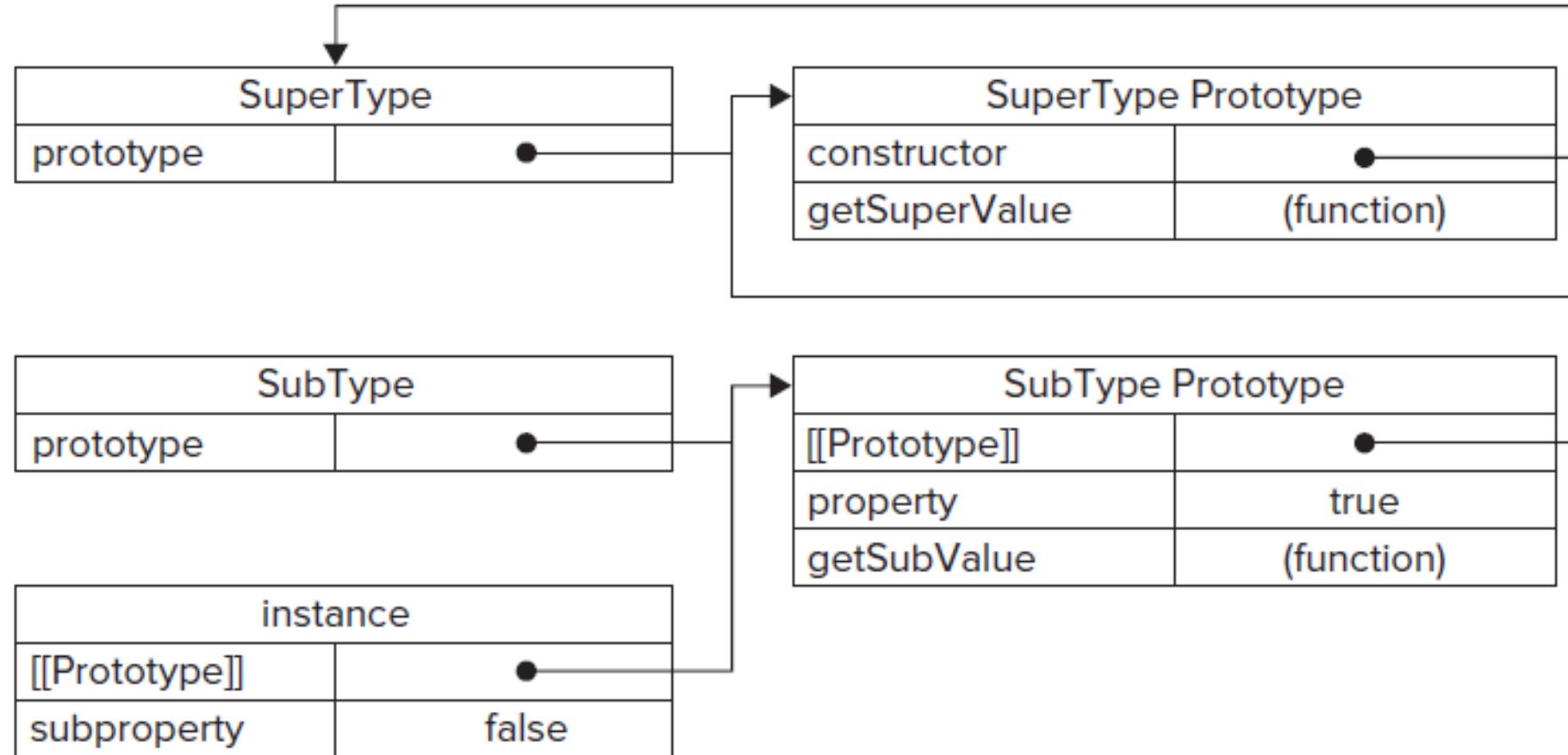


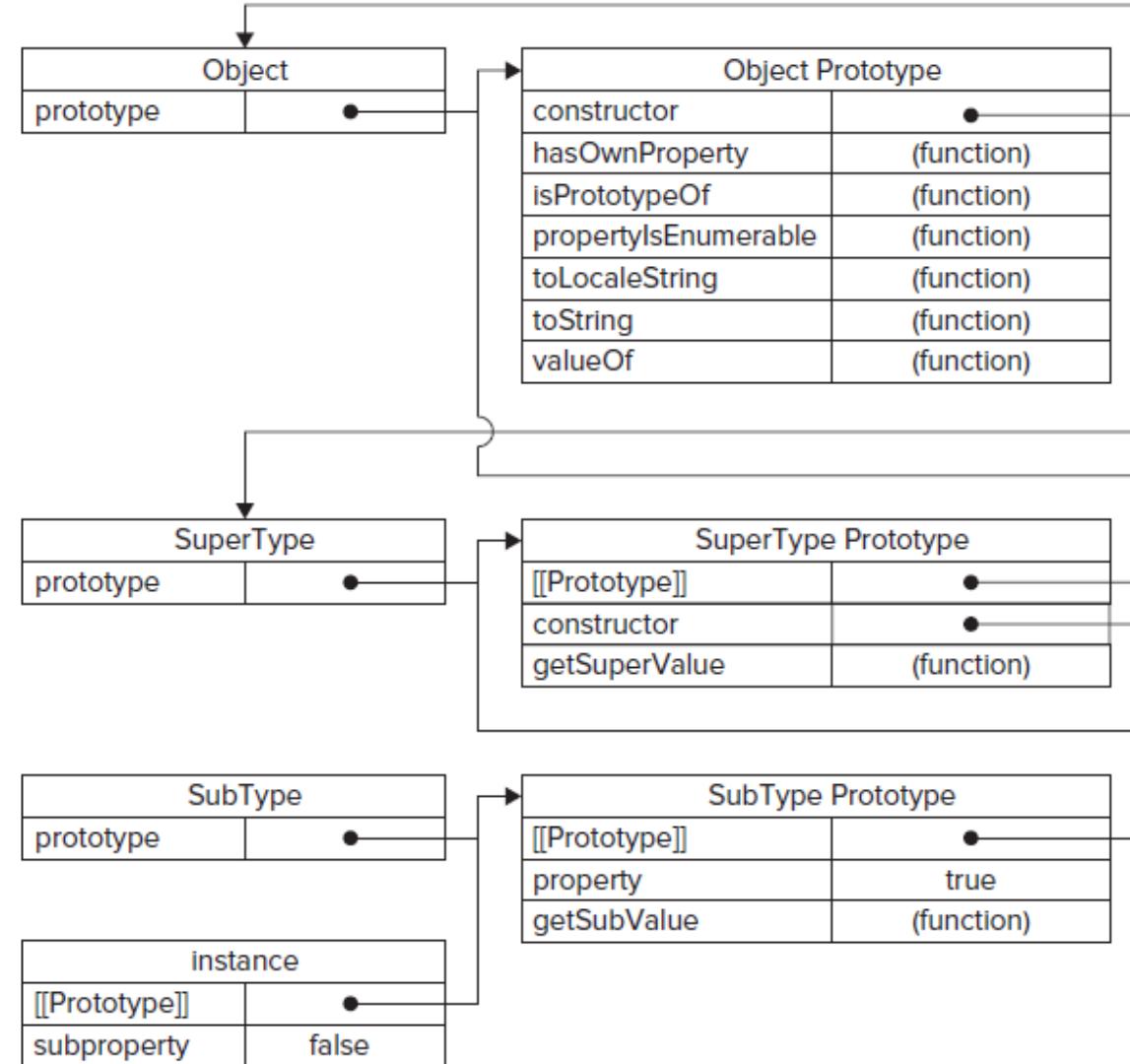
JS

ECMA-262 lo describe como el método principal de herencia en ECMAScript.

```
function SuperType() {  
    this.property = true;  
}  
SuperType.prototype.getSuperValue = function () {  
    return this.property;  
};
```

```
function SubType() {  
    this.subproperty = false;  
}  
//inherit from SuperType  
SubType.prototype = new SuperType();  
SubType.prototype.getSubValue = function () {  
    return this.subproperty;  
};  
  
var instance = new SubType();  
console.log(instance.getSuperValue()); //true
```





- Uno de los problemas de este tipo de herencia es el de los valores de referencia.
- Las propiedades de instancia del objeto padre, ahora pasan a ser propiedades del *prototype*, ya que hemos asignado una instancia del padre al *prototype* del hijo.
- Otro problema es que no podemos pasar argumentos al constructor del padre sin afectar a todas las instancias.

```
function SuperType() {  
    this.colors = ['red', 'blue', 'green'];  
}  
  
function SubType() {}  
SubType.prototype = new SuperType();
```

```
var instance1 = new SubType();  
instance1.colors.push('black');  
console.log(instance1.colors); // 'red,blue,green,black'  
  
var instance2 = new SubType();  
console.log(instance2.colors); // 'red,blue,green,black'
```

- En el robo de constructor o herencia clásica la idea es llamar al constructor del tipo padre desde el constructor del tipo hijo, cambiando el contexto.
- Esto solventa los problemas con los valores de referencia y nos permite pasar argumentos al constructor del tipo padre.

```
function SuperType(name) {  
    this.name = name;  
    this.colors = ['red', 'blue', 'green'];  
}  
function SubType(name) {  
    SuperType.apply(this, arguments);  
}
```

```
var instance1 = new SubType('John');  
instance1.colors.push('black');  
console.log(instance1.colors); // 'red,blue,green,black'  
var instance2 = new SubType('Stuart');  
console.log(instance2.colors); // 'red,blue,green'
```

- El problema de este patrón es que introduce los mismos problemas que vimos en el *Constructor Pattern* para la creación de objetos.
- No se reutilizan las funciones.
- Además, los métodos definidos en el *prototype* del objeto padre no son accesibles desde el objeto hijo.
- Por eso este patrón se suele utilizar junto a otro.

- La idea es usar *Prototype Chaining* para heredar las propiedades y métodos en el *prototype* y *Robar el Constructor* para heredar las propiedades de instancia.

```
function SuperType(name) {  
    this.name = name;  
    this.colors = ['red', 'blue', 'green'];  
}  
SuperType.prototype.sayName = function () {  
    console.log(this.name);  
}  
  
function SubType(name, age) {  
    SuperType.apply(this, arguments);  
    this.age = age;  
}  
SubType.prototype = new SuperType();  
SubType.prototype.sayAge = function () {  
    console.log(this.age);  
}
```

```
var instance1 = new SubType('John', 29);  
instance1.colors.push('black');  
console.log(instance1.colors); // 'red,blue,green,black'  
instance1.sayName(); // John;  
instance1.sayAge(); // 29  
  
var instance2 = new SubType('Stuart', 15);  
console.log(instance2.colors); // 'red,blue,green'  
instance2.sayName(); // Stuart;  
instance2.sayAge(); // 15
```

- Este tipo de herencia es una de las más usadas.
- Uno de los problemas de ineficiencia que presenta es que se llama 2 veces al constructor del tipo padre:
  - Una vez al crear el prototype del tipo hijo.
  - Otra vez al llamar al constructor del tipo padre desde el del tipo hijo.

- Ideado en 2006 por [Douglas Crockford](#), este método no usa constructores definidos como tal.
- Su premisa es que los *prototypes* te permiten crear nuevos objetos basados en objetos existentes sin la necesidad de definir tipos custom.
- Introdujo la siguiente función:

```
function object(o){  
    function F(){}
    F.prototype = o;  
    return new F();  
}
```

- La función `object` crea un constructor temporal.
- Asigna el objeto pasado por parámetro como prototype del constructor.
- Retorna una nueva instancia del tipo temporal.
- ES5 formaliza este concepto con la función `Object.Create()`.

```
var person = {  
    name: 'Nicholas',  
    friends: ['Shelby', 'Court', 'Van']  
};
```

```
var anotherPerson = object(person);  
anotherPerson.name = 'Greg';  
anotherPerson.friends.push('Rob');
```

```
var yetAnotherPerson = object(person);  
yetAnotherPerson.name = 'Linda';  
yetAnotherPerson.friends.push('Barbie');  
console.log(person.friends); //['Shelby,Court,Van,Rob,Barbie'
```

- La función *Object.Create()* acepta un segundo argumento usando el formato de *Object.defineProperties()*.
- Esto nos permite crear propiedades de instancia específicas.
- Recordad que las propiedades que contienen valores de referencias comparten esos valores.

```
var person = {  
    name: 'Nicholas',  
    friends: ['Shelby', 'Court', 'Van']  
};
```

```
var anotherPerson= Object.create(person, {  
    name: {  
        value: 'Linda'  
    }  
});  
console.log(anotherPerson.name); //Linda
```

- Otro patrón popularizado por Crockford.
- La idea es similar a los patrones de Parasitic Constructor y Factory Pattern.
- Se crea una función que haga la herencia, aumente el objeto y retorne el objeto.

```
function createAnother(original){  
    var clone = Object.create(original);  
    clone.sayHi= function(){ //No hay reutilización de las funciones  
        console.log('hi');  
    }  
    subType.prototype = prototype;  
}
```

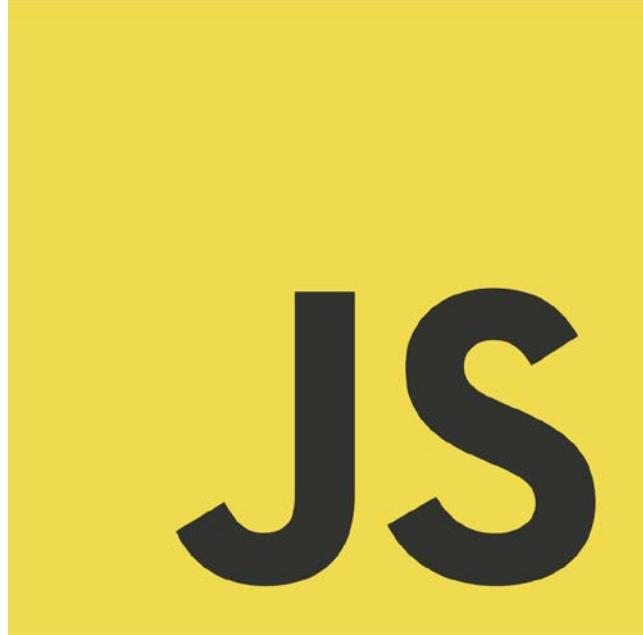
- Este patrón usa *Robo de Constructor* para heredar las propiedades pero usa un híbrido del patrón *Prototype Chaining* para heredar los métodos.
- La idea es que en vez de instanciar un tipo padre para el *prototype* del tipo hijo, copiemos el *prototype* del tipo padre.
- Para ello usamos la función *object* (introducida al hablar de *Prototype Inheritance*) o la función *Object.create()*.

```
function inheritPrototype(subType, superType){  
    var prototype = Object.create(superType.prototype);  
    prototype.constructor = subType;  
    subType.prototype = prototype;  
}
```

- Este es el patrón optimo de herencia para tipos de referencia.

```
function SuperType(name) {  
    this.name = name;  
    this.colors = ['red', 'blue', 'green'];  
}  
SuperType.prototype.sayName = function () {  
    console.log(this.name);  
}  
  
function SubType(name, age) {  
    SuperType.apply(this, arguments);  
    this.age = age;  
}  
inheritPrototype(SubType, SuperType);  
SubType.prototype.sayAge = function () {  
    console.log(this.age);  
}
```

```
var instance1 = new SubType('John', 29);  
instance1.colors.push('black');  
console.log(instance1.colors); // 'red,blue,green,black'  
instance1.sayName(); // John;  
instance1.sayAge(); // 29  
  
var instance2 = new SubType('Stuart', 15);  
console.log(instance2.colors); // 'red,blue,green'  
instance2.sayName(); // Stuart;  
instance2.sayAge(); // 15
```



# JavaScript Desmitificado

Bibliografía

東京' TOKIOTA



# Bibliografía

JS

ZAKAS, Nicholas C. *Professional JavaScript for Web Developers*; 3<sup>a</sup> ed. Indianapolis: John Wiley & Sons, Inc, 2012. 964p. ISBN: 978-1-118-02669-4

ZAKAS, Nicholas C. *Object-Oriented JavaScript*; 1<sup>a</sup> ed. San Francisco: No Starch Press, 2014. 122p. ISBN-10: 1-59327-540-4

ZAKAS, Nicholas C. *High Performance JavaScript*; 1<sup>a</sup> ed. USA: O'Reilly, 2010. 232p. ISBN: 978-0-596-80279-0

STEFANOV, Stoyan. *JavaScript Patterns*; 1<sup>a</sup> ed. USA: O'Reilly, 2010. 236p. ISBN: 978-0-596-80675-0

FOGUS, Michael. *Functional JavaScript*; 1<sup>a</sup> ed. USA: O'Reilly, 2013. 260p. ISBN: 978-1-449-36072-6

ZAKAS, Nicholas C. *Maintainable JavaScript*; 1<sup>a</sup> ed. USA: O'Reilly, 2012. 240p. ISBN: 978-1-449-32768-2