

JavaScript

Avanzado

Adolfo Sanz De Diego

Mayo 2015

1 Acerca de

1.1 Autor

- **Adolfo Sanz De Diego**
 - Blog: asanzdiego.blogspot.com.es
 - Correo: asanzdiego@gmail.com
 - GitHub: github.com/asanzdiego
 - Twitter: twitter.com/asanzdiego
 - LinkedIn: in/asanzdiego
 - SlideShare: slideshare.net/asanzdiego

1.2 Licencia

- **Este obra está bajo una licencia:**
 - Creative Commons Reconocimiento-CompartirIgual 3.0
- **El código fuente de los programas están bajo una licencia:**
 - GPL 3.0

2 JavaScript

2.1 Historia

- Lo crea **Brendan Eich en Netscape en 1995** para hacer páginas web dinámicas
- Aparece por primera vez en Netscape Navigator 2.0
- Cada día más usado (clientes web, videojuegos, windows 8, servidores web, bases de datos, etc.)

2.2 El lenguaje

- Orientado a objetos
- Basado en prototipos
- Funcional
- Débilmente tipado
- Dinámico

3 Orientación a objetos

3.1 ¿Qué es un objeto?

- **Colección de propiedades** (pares nombre-valor).
- Todo son objetos (las funciones también) excepto los primitivos: **strings, números, booleans, null o undefined**
- Para saber si es un objeto o un primitivo hacer **typeof variable**

3.2 Propiedades (I)

- Podemos acceder directamente o como si fuese un contenedor:

```
objeto.nombre === objeto[nombre] // true
```

3.3 Propiedades (II)

- Podemos crearlas y destruirlas en tiempo de ejecución

```
var objeto = {};  
objeto.nuevaPropiedad = 1; // añadir  
delete objeto.nuevaPropiedad; // eliminar
```

3.4 Objeto iniciador

- Podemos crear un objeto así:

```
var objeto = {  
  nombre: "Adolfo",  
  twitter: "@asanzdiego"  
};
```

3.5 Función constructora

- O con una función constructora y un new.

```
function Persona(nombre, twitter) {  
  this.nombre = nombre;  
  this.twitter = twitter;  
};  
var objeto = new Persona("Adolfo", "@asanzdiego");
```

3.6 Prototipos (I)

- Las funciones son objetos y tienen una propiedad llamada **prototype**.
- Cuando creamos un objeto con `new`, la referencia a esa propiedad **prototype** es almacenada en una propiedad interna.
- El prototipo se utiliza para compartir propiedades.

3.7 Prototipos (II)

- Podemos acceder al objeto prototipo de un objeto:

```
// Falla en Opera o IE <= 8  
Object.getPrototypeOf(objeto);  
  
// No es estandar y falla en IE  
objeto.__proto__;
```

3.8 Herencia

- Ejemplo:

```
function ConstructorA(p1) {  
  this.p1 = p1;  
}  
  
function ConstructorB(p1, p2) {  
  // llamamos al super para que no se pierda p1.  
  ConstructorA.call(this, p1);  
  this.p2 = p2;  
}  
  
// Hacemos que B herede de A  
// Prototipo de Función Constructora B apunta al  
// Prototipo de Función Constructora A  
ConstructorB.prototype = Object.create(ConstructorA.prototype);
```


3.9 Cadena de prototipos (I)

- Cuando se invoca una llamada a una propiedad, **JavaScript primero busca en el propio objeto, y si no lo encuentra busca en su prototipo**, y sino en el prototipo del prototipo, así hasta el prototipo de Object que es null.

3.10 Cadena de prototipos (II)

- En el ejemplo anterior:

```
instanciaB.__proto__ == ConstructorB.prototype // true  
instanciaB.__proto__.__proto__ == ConstructorA.prototype // true  
instanciaB.__proto__.__proto__.__proto__ == Object.prototype // true ins
```

3.11 Operador instanceof

- La expresión **instanciaB instanceof ConstructorA** devolverá true, si el prototipo de la Función ConstructorA, se encuentra en la cadena de prototipos de la instanciaB.
- En el ejemplo anterior:

```
instanciaB instanceof ConstructorB; // true  
instanciaB instanceof ConstructorA; // true  
instanciaB instanceof Object; // true
```

3.12 Extensión

- Con los prototipos podemos extender la funcionalidad del propio lenguaje.
- Ejemplo:

```
String.prototype.hola = function() {  
    return "Hola " + this;  
}  
  
"Adolfo".hola(); // "Hola Adolfo"
```

3.13 Eficiencia (I)

- Si queremos que nuestro código se ejecute una sola vez y que prepare en memoria todo lo necesario para generar objetos, la mejor opción es usar una **función constructora solo con el estado de una nueva instancia, y el resto (los métodos) añadirlos al prototipo.**

3.14 Eficiencia (II)

- Ejemplo:

```
function ConstructorA(p1) {  
  this.p1 = p1;  
}  
  
// los métodos los ponemos en el prototipo  
ConstructorA.prototype.metodo1 = function() {  
  console.log(this.p1);  
};
```

3.15 Propiedades y métodos estáticos (I)

- Lo que se define dentro de la función constructora va a ser propio de la instancia.
- Pero como hemos dicho, en JavaScript, una función es un objeto, al que podemos añadir tanto atributos como funciones.
- **Añadiendo atributos y funciones a la función constructora obtenemos propiedades y métodos estáticos.**

3.16 Propiedades y métodos estáticos (II)

- Ejemplo:

```
function ConstructorA() {  
    ConstructorA.propiedadEstatica = "propiedad estática";  
}  
  
ConstructorA.metodoEstatico = function() {  
    console.log("método estático");  
}
```


3.17 Propiedades y métodos privados (I)

- La visibilidad de objetos depende del contexto.
- Los contextos en JavaScript son bloques de código entre dos `{ }` y en general, desde uno de ellos, solo tienes acceso a lo que en él se defina y a lo que se defina en otros contextos que contengan al tuyo.

3.18 Propiedades y métodos privados (II)

- Ejemplo:

```
function ConstructorA(privada, publica) {  
  var propiedadPrivada = privada;  
  this.propiedadPublica = publica;  
  var metodoPrivado = function() {  
    console.log("-->propiedadPrivada", propiedadPrivada);  
  }  
  this.metodoPublico = function() {  
    console.log("-->propiedadPublica", this.propiedadPublica);  
    metodoPrivado();  
  }  
}
```

3.19 Polimorfismo

- Poder llamar a métodos sintácticamente iguales de objetos de tipos diferentes.
- Esto se consigue mediante herencia.

4 Técnicas avanzadas

4.1 Funciones

- Son objetos con sus propiedades.
- Se pueden pasar como parámetros a otras funciones.
- Pueden guardarse en variables.
- Son mensajes cuyo receptor es **this**.

4.2 This

- Ejemplo:

```
var nombre = "Laura";

var alba = {
  nombre: "Alba",
  saludo: function() {
    return "Hola " + this.nombre;
  }
}

alba.saludo(); // Hola Alba

var fn = alba.saludo;

fn(); // Hola Laura
```

4.3 call y apply

- Dos funciones permiten manipular el this: **call** y **apply** que en lo único que se diferencian es en la llamada.

```
fn.call(thisArg [, arg1 [, arg2 [...]]])
```

```
fn.apply(thisArg [, arglist])
```

4.4 Número variable de argumentos

- Las funciones en JavaScript aunque tengan especificado un número de argumentos de entrada, **pueden recibir más o menos argumentos** y es válido.

4.5 Arguments

- Es un objeto que **contiene los parámetros** de la función.

```
function echoArgs() {  
  console.log(arguments[0]); // Adolfo  
  console.log(arguments[1]); // Sanz  
}  
echoArgs("Adolfo", "Sanz");
```

4.6 Declaración de funciones

- Estas 2 declaraciones son **equivalentes**:

```
function holaMundo1() {  
    console.log("Hola Mundo 1");  
}  
holaMundo1();  
  
var holaMundo2 = function() {  
    console.log("Hola Mundo 2");  
}  
holaMundo2();
```

4.7 Transfiriendo funciones a otras funciones

- Hemos dicho que las funciones son objetos, así que **se pueden pasar como parámetros**.

```
function saluda() {  
  console.log("Hola")  
}  
function ejecuta(func) {  
  func()  
}  
ejecuta(saluda);
```

4.8 Funciones anónimas (I)

- Hemos dicho que las funciones se pueden declarar.
- Pero también **podemos no declararlas y dejarlas como anónimas.**

4.9 Funciones anónimas (II)

- Una función anónima así declarada **no se podría ejecutar**.

```
function(nombre) {  
  console.log("Hola "+nombre);  
}
```

4.10 Funciones anónimas (III)

- Pero una función puede devolver una función anónima.

```
function saludador(nombre) {  
  return function() {  
    console.log("Hola "+nombre);  
  }  
}  
  
var saluda = saludador("mundo");  
saluda(); // Hola mundo
```

4.11 Funciones autoejecutables

- Podemos autoejecutar funciones anónimas.

```
(function(nombre) {  
  console.log("Hola "+nombre);  
})("mundo")
```

4.12 Clousures

- Un closure **combina una función y el entorno en que se creó.**

```
function creaSumador(x) {  
  return function(y) {  
    return x + y;  
  };  
}  
  
var suma5 = creaSumador(5);  
var suma10 = creaSumador(10);  
  
console.log(suma5(2)); // muestra 7  
console.log(suma10(2)); // muestra 12
```


4.13 El patrón Modulo

- Se trata de una función que actúa como contenedor para un contexto de ejecución.

```
miModulo = (function() {  
  
    var propiedadPrivada;  
  
    function metodoPrivado() { };  
  
    // API publica  
    return {  
        metodoPublico1: function () {  
        },  
  
        metodoPublico2: function () {  
        }  
    }  
} ());
```

4.14 Eficiencia (I)

- Si se ejecuta desde el navegador, **se suele pasar como parámetro el objeto window para mejorar el rendimiento.** Así cada vez que lo necesitemos el intérprete lo utilizará directamente en lugar de buscarlo remontando niveles.
- Y también **se suele pasar el parámetro undefined,** para evitar los errores que pueden darse si la **palabra reservada ha sido reescrita** en alguna parte del código y su valor no corresponda con el esperado.

4.15 Eficiencia (II)

```
miModulo = (function(window, undefined) {  
    // El código va aquí  
})( window );
```

4.16 El patrón Modulo Revelado (I)

- El problema del patrón Modulo es pasar un método de privado a público o viceversa.
- Por ese motivo lo que se suele hacer es definir todo en el cuerpo, y luego **referenciar solo los públicos en el bloque return.**

4.17 El patrón Modulo Revelado (II)

```
miModulo = (function() {  
  
    function metodoA() { };  
  
    function metodoB() { };  
  
    function metodoC() { };  
  
    // API publica  
    return {  
        metodoPublico1: metodoA,  
        metodoPublico2: metodoB  
    }  
} ());
```

4.18 Espacios de nombres (I)

- Para simular espacios de nombre, en JavaScript se anidan objetos.

```
miBiblioteca = miBiblioteca || {};  
  
miBiblioteca.seccion1 = miBiblioteca.seccion1 || {};  
  
miBiblioteca.seccion1 = {  
  propiedad: p1,  
  metodo: function() { },  
};  
  
miBiblioteca.seccion2 = miBiblioteca.seccion2 || {};  
  
miBiblioteca.seccion2 = {  
  propiedad: p2,  
  metodo: function() { },  
};
```

4.19 Espacios de nombres (II)

- Se combinar lo anterior con módulos autoejecutables:

```
miBiblioteca = miBiblioteca || {};  
  
(function(namespace) {  
  
    var propiedadPrivada = p1;  
  
    namespace.propiedadPublica = p2;  
  
    var metodoPrivado = function() { };  
  
    namespace.metodoPublico = function() { };  
  
}(miBiblioteca));
```

5 Document Object Model

5.1 ¿Qué es DOM?

- Acrónimo de **Document Object Model**
- Es un conjunto de utilidades específicamente diseñadas para **manipular documentos XML, y por extensión documentos XHTML y HTML.**
- DOM transforma internamente el archivo XML en una estructura más fácil de manejar formada por una jerarquía de nodos.

5.2 Tipos de nodos

- Los más importantes son:
 - **Document**: representa el nodo raíz.
 - **Element**: representa el contenido definido por un par de etiquetas de apertura y cierre y puede tener tanto nodos hijos como atributos.
 - **Attr**: representa el atributo de un elemento.
 - **Text**: almacena el contenido del texto que se encuentra entre una etiqueta de apertura y una de cierre.

5.3 Recorrer el DOM

- JavaScript proporciona **funciones** para recorrer los nodos:

```
getElementById(id)  
getElementsByName(name)  
getElementsByTagName(tagname)  
getElementsByClassName(className)  
getAttribute(attributeName)  
querySelector(selector)  
querySelectorAll(selector)
```

5.4 Manipular el DOM

- JavaScript proporciona **funciones** para la manipulación de nodos:

```
createElement(tagName)
createTextNode(text)
createAttribute(attributeName)
appendChild(node)
insertBefore(newElement, targetElement)
removeAttribute(attributename)
removeChild(childreference)
replaceChild(newChild, oldChild)
```

5.5 Propiedades Nodos (I)

- Los nodos tienen algunas **propiedades** muy útiles:

```
attributes[]  
className  
id  
innerHTML  
nodeName  
nodeValue  
style  
tabIndex  
tagName  
title
```

5.6 Propiedades Nodos (II)

- Los nodos tienen algunas **propiedades** muy útiles:

```
childNodes[]  
firstChild  
lastChild  
previousSibling  
nextSibling  
ownerDocument  
parentNode
```

5.7 jQuery

- **jQuery** puede ser muy util en ciertos casos, pero en muchos otros es **matar moscas a cañonados**.

6 Enlaces

6.1 General (ES)

- <http://developer.mozilla.org/es/docs/Web/JavaScript/G>
- <http://cevicejs.com/>
- <http://www.arkaitzgarro.com/javascript/>
- <http://www.etnassoft.com/category/javascript/>

6.2 General (EN)

- <http://www.javascriptkit.com/>
- <http://javascript.info/>
- <http://www.howtcreate.co.uk/tutorials/javascript/>

6.3 Orientación Objetos (ES) (I)

- <http://www.programania.net/disenio-de-software/entendiendo-los-prototipos-en-javascript/>
- <http://www.programania.net/disenio-de-software/creacion-de-objetos-eficiente-en-javascript/>
- <http://blog.amatiasq.com/2012/01/javascript-conceptos-basicos-herencia-por-prototipos/>

6.4 Orientación Objetos (ES) (II)

- <http://albertovilches.com/profundizando-en-javascript-parte-1-funciones-para-todo>
- <http://albertovilches.com/profundizando-en-javascript-parte-2-objetos-prototipos-herencia-y-namespaces>
- <http://www.arkaitzgarro.com/javascript/capitulo-9.html>
- <http://www.etnassoft.com/2011/04/15/concepto-de-herencia-prototipica-en-javascript/>

6.5 Orientación Objetos (EN)

- <http://www.codeproject.com/Articles/687093/Understanding-JavaScript-Object-Creation-Patterns>
- <http://javascript.info/tutorial/object-oriented-programming>
- <http://www.howtocreate.co.uk/tutorials/javascript/object-oriented-programming>

6.6 Técnicas avanzadas (ES) (I)

- <http://www.etnassoft.com/2011/03/14/funciones-autoejecutables-en-javascript/>
- <http://www.etnassoft.com/2012/01/12/el-valor-de-this-javascript-como-manejarlo-correctamente/>
- <https://developer.mozilla.org/es/docs/Web/JavaScript/>
- <http://www.variablenotfound.com/2012/10/closures-en-javascript-entiendelos-de.html>

6.7 Técnicas avanzadas (ES) (II)

- <http://www.webanalyst.es/espacios-de-nombres-en-javascript/>
- <http://www.etnassoft.com/2011/04/11/el-patron-de-modulo-en-javascript-en-profundidad/>
- <http://www.etnassoft.com/2011/04/18/ampliando-patron-modulo-javascript-submodulos/>
- <http://notasjs.blogspot.com.es/2012/04/el-patron-modulo-en-javascript.html>

6.8 DOM (ES)

- <http://cevicejs.com/3-dom-cssom#dom>
- <http://www.arkaitzgarro.com/javascript/capitulo-13.html>

6.9 DOM (EN)

- <http://www.javascriptkit.com/domref/>
- <http://javascript.info/tutorial/dom>

6.10 ES6 (ES)

- <http://rlbisbe.net/2014/08/26/articulo-invitado-ecmascript-6-y-la-nueva-era-de-javascript-por-ckgrafico/>
- <http://carlosazaustre.es/blog/ecmascript-6-el-nuevo-estandar-de-javascript/>

6.11 ES6 (EN)

- <http://es6-features.org/>
- <http://kangax.github.io/compat-table/es5/>