Buenas Prácticas.

1. Declarar las variables antes.

2. No declarar Numbers, Strings o Booleans como objetos ({ }).

3. No usar new Object(). Se usa {} para objetos, “” para Strings, 0..9 para Numbers, false-true para Booleans, [] para Arrays, function(){} para funciones.

4. No escribir Numbers como Strings (”5”).

5. No se pueden restar Strings, genera NaN.

   (“Hello”-”World” = NaN).

6. Usar el triple igual para comparar (===).

7. No usar eval().

Qué es Javascript?

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

. *JavaScript del lado del cliente* extiende el núcleo del lenguaje proporcionando objetos para controlar el browser.

. *Javascript del lado del servidor* extiende el núcleo del lenguaje proporcionando objetos relevantes para el servidor.

En contraste con el sistema de clases construidas por declaraciones en tiempo de compilación de Java, Javascript soporta un sistema en tiempo de ejecución basado en un pequeño número de tipos de datos que representan valores numéricos, booleanos, y cadenas. Javascript tiene un modelo de objetos basado en prototipos en lugar del más común modelo de objetos basado en clases.

Javascript

1. Orientado a objetos.
2. Tipos de datos de las variables no son declarados.

Java

1. Basado en clases.
2. Tipos de datos de las variables deben ser declarados.

Programación Orientada a Objetos.

Los objetos son entidades que tienen un determinado *estado*, *comportamiento (método)* e *identidad*:

1. El *estado* está compuesto de datos o informaciones; serán uno o varios atributos a los que se habrán asignado unos valores concretos (datos).

2. El comportamiento está definido por los métodos o mensajes a los que sabe responder dicho objeto, es decir, qué operaciones se pueden realizar con él.

3. La identidad es una propiedad de un objeto que lo

diferencia del resto; dicho con otras palabras, es su identificador (concepto análogo al de identificador de una variable o una constante).

Los métodos (comportamiento) y atributos (estado) están estrechamente relacionados por la propiedad de conjunto. Esta propiedad destaca que una clase requiere de métodos para poder tratar los atributos con los que cuenta. El programador debe pensar indistintamente en ambos conceptos, sin separar ni darle mayor importancia a alguno de ellos.

En la programación estructurada solo se escriben funciones que procesan datos. Los programadores que emplean Programación Orientada a Objetos, en cambio, primero definen objetos para luego enviarles mensajes solicitándoles que realicen sus métodos por sí mismos

Origen

Los conceptos de la programación orientada a objetos tienen origen en Simula 67, un lenguaje diseñado para hacer simulaciones; en el Centro de Cómputo Noruego, en Oslo.

En este centro se trabajaba en simulaciones de naves, que fueron confundidas por la explosión combinatoria de cómo las diversas cualidades de diferentes naves podían afectar unas a las otras. La idea surgió al agrupar los diversos tipos de naves en diversas clases de objetos,

siendo responsable cada clase de objetos de definir sus propios datos y comportamientos.

Fueron refinados más tarde en Smalltalk, pero diseñado para ser un sistema completamente dinámico en el cual los objetos se podrían crear y modificar "sobre la marcha" (en tiempo de ejecución) en lugar de tener un sistema basado en programas estáticos.

Las características de orientación a objetos fueron agregadas a muchos lenguajes existentes durante ese tiempo, incluyendo Ada, BASIC, Lisp mas Pascal, entre otros.

Los lenguajes orientados a objetos "puros", carecían de las características de las cuales muchos programadores habían venido a depender. Para saltar este obstáculo, se hicieron muchas tentativas para crear nuevos lenguajes basados en métodos orientados a objetos, pero permitiendo algunas características imperativas de maneras "seguras". El Eiffel de Bertrand Meyer fue un temprano y moderadamente acertado lenguaje con esos objetivos, pero ahora ha sido esencialmente reemplazado por Java, en gran parte debido a la aparición de Internet y a la implementación de la máquina virtual de Java en la mayoría de navegadores.

Conceptos fundamentales

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

Herencia
Es la facilidad mediante la cual la clase D hereda en ella cada uno de los atributos y operaciones de C. Por lo tanto, puede usar los mismos métodos y variables públicas declaradas en C. Los componentes registrados como "privados" también se heredan, pero como no pertenecen a la clase, se mantienen escondidos al programador y sólo pueden ser accedidos a través de otros métodos públicos.

Objeto
Instancia de una clase. Entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o

funcionalidad (métodos), los mismos que consecuentemente reaccionan a eventos. Es una instancia a una clase.

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

Evento
Es un suceso en el sistema (tal como una interacción del usuario con la máquina, o un mensaje enviado por un objeto). El sistema maneja el evento enviando el mensaje adecuado al objeto pertinente. También se puede definir como evento la reacción que puede desencadenar un objeto; es decir, la acción que genera.

Atributos
Características que tiene la clase.

Mensaje

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

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

Estado interno
Es una variable que se declara privada, que puede ser únicamente accedida y alterada por un método del objeto. No es visible al programador que maneja una instancia de la clase.

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

Identificación de un objeto
Un objeto se representa por medio de una tabla o entidad que esté compuesta por sus atributos y funciones correspondientes.
En comparación con un lenguaje imperativo, una "variable" no es más que un contenedor interno del atributo del objeto o de un estado interno.

Motor de Renderizado

Un **motor del navegador web** (a veces llamado **motor de diseño** o **motor de renderizado**) es un software de componente que tiene marcado contenido (como HTML , XML , imágenes de archivos, etc.) y la información de formato (como CSS , XSL , etc.) y muestra el contenido formateado en la pantalla. Se basa en el área de contenido de una ventana, que se muestra en un monitor de o una impresora . Un motor de diseño es típicamente incrustado en los navegadores web , clientes de correo electrónico , lectores de libros electrónicos , en línea los sistemas de ayuda u otras aplicaciones que requieren la visualización (y edición) de contenido web.

Ejemplos:

Safari - WebKit

Chrome y Opera (versiones actuales) - Blink (Basado en Webkit) + V8

Mozilla Firefox - Gecko

Internet Explorer - Trident

Operacion Técnica

El **motor** hace la mayoría del trabajo. En esencia, toma una URL y un conjunto de ventana del área de contenido rectángulo coordina como argumentos. A continuación, recupera el documento correspondiente a la URL y pinta una representación gráfica de ello en el rectángulo dado.

La **aplicación host** proporciona la barra de menú, barra de direcciones, barra de estado, administrador de marcadores, historial y preferencias, funcionalidad, etc. Se incrusta el motor y sirve como una interfaz entre el usuario, el motor, y el sistema operativo subyacente. Dado que proporciona los elementos gráficos que rodean el área en la que el motor pinta documentos.

Interpretador de JS

Javascript fue impulsado gracias a que Microsoft creó ajax con el objeto xmlhttprequest. Empezaron a surgir muchas aplicaciones con Js y surgió la necesidad de mejorar su velocidad.Los motores interpretadores de Javascript de última generación nacieron gracias a que Macromedia liberó a la comunidad una parte de su código de

interpretación de action script.

De esto nacieron: Nitro es el motor de js de alta velocidad de Safari. Firefox utiliza **Spidermonkey** como motor de javascript. Chrome utiliza **V8** y permite node.js

Programación del lado del cliente y del lado del servidor

El Servidor: Es la parte encargada de servir las páginas web.

El Cliente: Solicita las páginas al servidor y las muestra al usuario. En la mayoría de los casos el cliente es un navegador web.

El Usuario: Utiliza al **Cliente** con el fin de navegar por la web.

Programación del lado del Servidor

Es el nombre que se da para los programas que se ejecutan en el servidor.

Usos

1. Procesos de entrada del usuario.
2. Páginas de pantalla.
3. Estructura de páginas web.
4. Interactuar con el almacenamiento permanente (SQL. archivos).

Idiomas

1. PHP.
2. ASP.net en C#, C++, o Visual Basic.

Programación del lado del Cliente.

Es el nombre que se da para los programas que se ejecutan en el cliente.

Usos

1. Hacer páginas web interactivas.

2. Asegurarse de que las cosas sucedan dinámicamente en la página web.
3. Interactuar con el almacenamiento temporal y local (cookies, localStorage).
4. Enviar solicitudes al servidor y recuperar datos de él.
5. Brindar servicio remoto para aplicaciones del lado del cliente.

Idiomas

1.Javascript. 2.HTML. 3.CSS.

Scripting Language

Un **lenguaje de programación** o **lenguaje de script** es un lenguaje de programación que admite **secuencias de comandos** , los programas escritos para un especial entorno de tiempo de ejecución que puede interpretar (en lugar de compilar) y automatizar la ejecución de tareas que, alternativamente, podría ser ejecutado de una en una por un ser humano operador. Los ambientes que se pueden automatizar mediante scripts incluyen aplicaciones de software , páginas web dentro de un navegador web , las conchas de los sistemas operativos (OS) y sistemas embebidos .

Tipos de Scripting Languages

Idiomas Pegamento

Es un lenguaje de programación (por lo general una interpretado lenguaje de script) que está diseñado o adaptado para la escritura de código de unión - código para conectar componentes de software . Son especialmente útiles para la escritura y el mantenimiento de:

1. Los comandos personalizados para una consola de comandos
2. Los programas más pequeños que los que están mejor implementados en un lenguaje compilado
3. Programas de "contenedor" para ejecutables, como un archivo por lotes que se mueve o manipula archivos y hace otras cosas con el sistema operativo antes o después de ejecutar una aplicación como un procesador de textos, hoja de cálculo, base de datos, ensamblador, compilador, etc.
4. Los scripts que pueden cambiar
5. Prototipos rápidos de una solución eventualmente implementado en otro, generalmente se compilan, el lenguaje.

Algunos ejemplos:

1. Windows PowerShell.
2. JScript y Javascript.
3. Applescript.
4. Python.

Lenguajes de control de empleo y conchas.

Una clase importante de lenguajes de script ha crecido fuera de la automatización de control de trabajo , que se refiere a iniciar y controlar el comportamiento de los programas del sistema. (En este sentido, se podría pensar en conchas de ser descendientes de JCL de IBM, o Job Control Language , que se utiliza para este propósito.) Muchos de los intérpretes dobles como estas lenguas ' intérpretes de línea de comandos como el shell de Unix o el MS-DOSCOMMAND.COM . Otros, como AppleScript ofrecen el uso de inglés-como mandatos para crear scripts.

GUI scripting

Con el advenimiento de interfaces gráficas de usuario, un tipo especializado de lenguaje de script surgió para controlar un ordenador. Estas lenguas interactuar con los mismos gráficos de ventanas, menús, botones, etc. que un usuario humano haría. Lo hacen mediante la

simulación de las acciones de un usuario.Estos lenguajes se utilizan normalmente para automatizar las acciones del usuario. Estas lenguas son también llamados " macros "cuando el control es a través de pulsaciones de teclas simuladas o clics del ratón.

Algunos lenguajes de scripting de GUI se basan en el reconocimiento de objetos gráficos de la pantalla de visualización de píxeles . Estos lenguajes de scripting de GUI no dependen del apoyo del sistema operativo o la aplicación.

Lenguajes específicos de la aplicación

Muchos programas de aplicación grandes incluyen un lenguaje de scripting idiomática adaptado a las necesidades del usuario de la aplicación. Del mismo modo, muchos juegos de ordenador sistemas utilizan un lenguaje de script personalizado para expresar las acciones programadas de personajes no jugadores y el entorno del juego. Idiomas de este tipo están diseñados para una sola aplicación; y, aunque superficialmente pueden parecerse a un lenguaje de propósito general específica (por ejemplo QuakeC , siguiendo el modelo C), tienen características personalizadas que los distinguen. Emacs Lisp , mientras que un dialecto del totalmente formada y capaz Lisp , contiene muchas características especiales que lo hacen más útil para ampliar las funciones de edición de Emacs. Un lenguaje de programación específico de la aplicación puede ser visto

como un lenguaje de programación específico de dominio especializado para una sola aplicación.

Extensión / idiomas embebibles

Un número de idiomas se han diseñado con el propósito de reemplazar los lenguajes de script específicos de la aplicación por ser integrable en los programas de aplicación. El programador de aplicaciones (que trabajan en C o en otro idioma sistemas) incluye "ganchos" en el lenguaje de script puede controlar la aplicación. Estas lenguas pueden ser técnicamente equivalente a un lenguaje de extensión específica de la aplicación, pero cuando una aplicación incorpora un lenguaje "común", el usuario obtiene la ventaja de ser capaz de transferir habilidades de aplicación en aplicación. Una alternativa más genérico es simplemente proporcionar una biblioteca (a menudo una biblioteca C) que una lengua de uso general se puede utilizar para controlar la aplicación, sin modificar el idioma para el dominio específico.

Operadores Lógicos

Son típicamente usados con booleanos

&& : AND lógico. Retorna TRUE si ambas condiciones se cumplen.

|| : OR lógico. Retorna TRUE si al menos una condición se

cumple.

! : NOT lógico. Se usa para negar una condición.

Typeof

Typeof undefined = undefined

Typeof null = object

null === undefined = false

null == undefined = true

Operator	Description	Examples returning true

Equal (==)	Returns true if the operands are equal.	`3 == var1`
		`"3" == var1`
		
		`3 == '3'`

Not equal (!=)	Returns true if the operands are not equal.	`var1 != 4` `var2 != "3"`
Strict equal (===)	Returns true if the operands are equal and of the same type. See also `Object.is` and sameness in JS.	`3 === var1`
Strict not equal (!==)	Returns true if the operands are not equal and/or not of the same type.	`var1 !== "3"` `3 !== '3'`
Greater than (>)	Returns true if the left operand is greater than the right operand.	`var2 > var1` `"12" > 2`
Greater than or equal (>=)	Returns true if the left operand is greater than or equal to the right operand.	`var2 >= var1` `var1 >= 3`
Less than (<)	Returns true if the left operand is less than the right operand.	`var1 < var2` `"2" < "12"`
Less than or equal (<=)	Returns true if the left operand is less than or equal to the right operand.	`var1 <= var2` `var2 <= 5`

Delete

El operador delete elimina un objeto, una propiedad de un objeto, o un elemento en un campo específico de un array.

```
delete objectName;
delete objectName.property;
delete objectName[index];
delete property; // legal only within a with statement
```

Operadores aritméticos

Toman valores numéricos como sus operandos y retornan un único valor. Los estándares son suma (+), resta (-), multiplicación (*), y división (/).

```
console.log(1 / 2); /* prints 0.5 */
console.log(1 / 2 == 1.0 / 2.0); /* also this is true */
```

También tenemos los que se muestran en la siguiente tabla.

Table 3.3 Arithmetic operators

Operator	Description	Example
% (Modulus)	Binary operator. Returns the integer remainder of dividing the two operands.	12 % 5 returns 2.
++ (Increment)	Unary operator. Adds one to its operand. If used as a prefix operator (++x), returns the value of its operand after adding one; if used as a postfix operator (x++), returns the value of its operand before adding one.	If x is 3, then ++x sets x to 4 and returns 4, whereas x++ returns 3 and, only then, sets x to 4.
-- (Decrement)	Unary operator. Subtracts one from its operand. The return value is analogous to that for the increment operator.	If x is 3, then --x sets x to 2 and returns 2, whereas x-- returns 3 and, only then, sets x to 2.
- (Unary negation)	Unary operator. Returns the negation of its operand.	If x is 3, then -x returns -3.

Edit This Code: See Result »

```
<html>
<body>

<p>The internal clock in JavaScript starts at midnight
January 1, 1970.</p>
<p>Click the button to display the number of milliseconds
since midnight, January 1, 1970.</p>

<button onclick="myFunction()">Try it</button>

<p id="demo"></p>

<script>
function myFunction() {
    var d1 = new Date();
    var mil1 = d1.getTime();

    for(var i=0;i<10000000;i++){
         var r = 5+5;
    }
    var d2 = new Date();
    var mil2 = d2.getTime();
     document.getElementById("demo").innerHTML = mil2 +"<br>
"+mil1;
}
</script>

</body>
</html>
```

Result:

The internal clock in JavaScript starts at midnight January 1, 1970.

Click the button to display the number of milliseconds since midnight, January 1, 1970.

Try it

1422121422378
1422121422367

HTML

```
<body>
    <button onclick="alert('Inline Scripting')">Click Me</button>
    <button ondblclick="alert('doble Click')">Doble</button>


    <button id="btn-conEvento">Tengo un evento agregado</button>
    <input id="in-test" placeholder="Input vacio"/>
</body>
```

JavaScript

```
document.getElementById('btn-conEvento').addEventListener("click",
function(event) {
     alert('algo');
});

document.getElementById('in-test').addEventListener("blur",
function(event) {
     alert('fuera de foco');
});
```

```javascript
function getType(valor){
    return typeof valor;

}
var sinValor;
var conValor = "CETAV";

//limpiar/ borrar
delete conValor;

console.log('34 ->'+getType(34));
console.log('true ->'+getType(true));
console.log('sinvalor ->'+getType(sinValor));
console.log('null ->'+getType(null));
console.log('objeto ->'+getType({'nombre':'JavaScript Dev'}));
console.log('conValor ->'+getType(conValor));
```

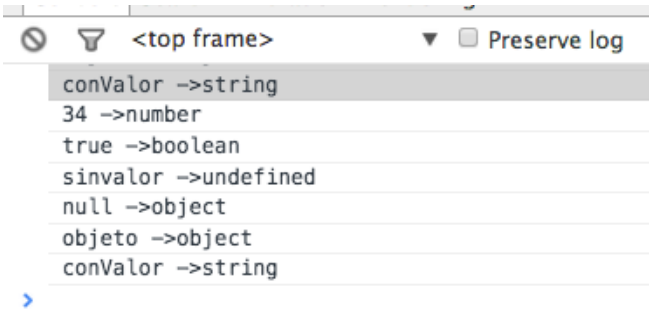


Tabla de Precedencia

Precedence	Operator Type	Associativity	Individual Operators
19	Grouping	n/a	(…)
18	Member Access	left-to-right	… . …
	Computed Member Access	left-to-right	… […]
	new(with	n/a	new … (…)

	argument list)		
17	Function call	left-to-right	… (…)
	new(with argument list)	right-to-left	new ...
16	Postfix Increment	n/a	… ++
	postfix Decrement	n/a	… --

15	Logical NOT	right-to-left	! …
	Bitwise NOT	right-to-left	~ …
	Unary Plus	right-to-left	+ …
	Unary Negation	right-to-left	- …
	Prefix Increment	right-to-left	++ …
	Prefix Decrement	right-to-left	-- …
	typeof	right-to-left	typeof …
	void	right-to-left	void …
	delete	right-to-left	delete …
14	Multiplication	left-to-right	… * …
	Division	left-to-right	… / …
	Remainder	left-to-right	… % …

13	Addition	left-to-right	… + …
	Subtraction	left-to-right	… - …
12	Bitwise Left Shift	left-to-right	… << …
	Bitwise Right Shift	left-to-right	… >> …
	Bitwise Unsigned Right Shift	left-to-right	… >>> …
11	Less Than	left-to-right	… < …
	Less Than or Equal	left-to-right	… <= …
	Greater Than	left-to-right	… > …
	Greater Than or Equal	left-to-right	… >= …
	in	left-to-right	… in …
	instanceof	left-to-right	… instanceof …

10	Equality	left-to-right	… == …
	Inequality	left-to-right	… != …
	Strict Equality	left-to-right	… === …
	Strict Inequality	left-to-right	… !== …
9	Bitwise AND	left-to-right	… & …

8	Bitwise XOR	left-to-right	… ^ …
7	Bitwise OR	left-to-right	… | …
6	Logical AND	left-to-right	… && …
5	Logical OR	left-to-right	… || …
4	Conditional	right-to-left	… ? … : …

3	Assignment	right-to-left	… = …
			… += …
			… -= …
			… *= …
			… /= …
			… %= …
			… <<= …
			… >>= …
			… >>>= …
			… &= …
			… ^= …
			… |= …

2	yield	right-to-left	yield ...
1	Spread	n/a	... …
0	Comma /	left-to-right	… , …

	Sequence		

String.prototype.indexOf()

El método **`indexOf()`** devuelve el índice, dentro del objeto `String` que realiza la llamada, de la primera ocurrencia del valor especificado, comenzando la búsqueda desde `indice Busqueda`; o -1 si no se encuentra dicho valor.

Sintaxis

```
cadena.indexOf(valorBusqueda[, indiceDesde]);
```

Parámetros

`valorBusqueda`

Una cadena que representa el valor de búsqueda.

`indiceDesde`

La localización dentro de la cadena llamada desde la que empezará la búsqueda. Puede ser un entero entre 0 y la

longitud de la cadena. El valor predeterminado es 0.

Descripción

Los caracteres de una cadena se indexan de izquierda a derecha. El índice del primer carácter es 0, y el índice del último carácter de una cadena llamada `nombreCadena` es`nombreCadena.length - 1`.

```
"Blue Whale".indexOf("Blue")   // returns 0
"Blue Whale".indexOf("Blute")  // returns -1
"Blue Whale".indexOf("Whale",0) // returns 5
"Blue Whale".indexOf("Whale",5) // returns 5
"Blue Whale".indexOf("",9)     // returns 9
"Blue Whale".indexOf("",10)    // returns 10
"Blue Whale".indexOf("",11)    // returns 10
```

El método `indexOf` es sensible a mayúsculas. Por ejemplo, la siguiente expresión devuelve -1:

```
"Ballena Azul".indexOf("azul");
```

Ejemplos

```
var cualquierCadena="Brave new world"


document.write("<P>The index of the first w from the beginning is " +
   cualquierCadena.indexOf("w"))        // Muestra 8

document.write("<P>The index of the first w from the end is " +
   cualquierCadena.lastIndexOf("w"))   // Muestra 10

document.write("<P>The index of 'new' from the beginning is " +
   cualquierCadena.indexOf("new"))     // Muestra 6

document.write("<P>The index of 'new' from the end is " +
   cualquierCadena.lastIndexOf("new")) // Muestra 6
```

Ejemplo: Usando `indexOf` para contar ocurrencias de una letra en una cadena

```
cuenta = 0;
posicion = miCadena.indexOf("x");
while ( posicion != -1 ) {
   cuenta++;
   posicion = miCadena.indexOf("x",posicion+1);
}
```

Hoisting

JavaScript Declarations are Hoisted

En JS una variable puede ser declarada después de ser usada.

En otras palabras, una variable puede ser usada antes de ser declarada.

```
x = 5; // Assign 5 to x

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x;                     // Display x in the element

var x; // Declare x
```

Para entender esto, se tiene que entender el término Hoisting.

Es el comportamiento predeterminado de JS de mover todas las declaraciones al top del actual scope (al inicio del script actual o de la función actual).

JavaScript Initializations are Not Hoisted

Js solo eleva declaraciones, no inicializaciones.

```
var x = 5; // Initialize x
var y = 7; // Initialize y

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y;           // Display x and y
```

```
var x = 5; // Initialize x

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y;           // Display x and y


var y = 7; // Initialize y
```

JavaScript Use Strict

Define que el código de JS es ejecutado en "strict mode".

Declaring Strict Mode

```
"use strict";
x = 3.14;       // This will cause an error
myFunction();   // This will also cause an error

function myFunction() {
      x = 3.14;
}
```

```
x = 3.14;       // This will not cause an error.
myFunction();   // This will cause an error
```

```
function myFunction() {
   "use strict";
      x = 3.14;
}
```

Sintaxis

La sintaxis fué diseñada para ser compatible con versiones viejas de JS

Errores Comunes

1. Uso de = en vez de === ó ==.
2. Confundir adición y concatenación.
3. Escribir strings en más de una línea.
4. No poner punto y coma (;).

Accesando Arrays con índices nombrados.

```
var person = [];
person[0] = "John";
person[1] = "Doe";
person[2] = 46;
var x = person.length;         // person.length will return 3
var y = person[0];             // person[0] will return "John"
```

El último elemento de un Array no lleva coma(,). Al igual que las definiciones de objetos.

Undefined is Not Null

Con JavaScript, null es para objetos, undefined es para variables, propiedades, y métodos.

Para ser null, un objeto tiene que ser definido, de otra manera será undefined.

Incorrect:

```
if (myObj !== null && typeof myObj !== "undefined")
```

Because of this, you must test typeof() first:

Correct:

```
if (typeof myObj !== "undefined" && myObj !== null)
```

JSON

JSON es un formato para almacenar y transportar datos.

JSON es usualmente cuando los datos son enviados desde un servidor a una página web.

```
{"employees":[
      {"firstName":"John", "lastName":"Doe"},
      {"firstName":"Anna",    "lastName":"Smith"},
      {"firstName":"Peter", "lastName":"Jones"}
]}
```

Objetos de JSON

```
{"firstName":"John", "lastName":"Doe"}
```

Arrays de JSON

```
"employees":[

      {"firstName":"John", "lastName":"Doe"},

      {"firstName":"Anna", "lastName":"Smith"},

      {"firstName":"Peter", "lastName":"Jones"}

]
```

Convertir strings a number.

```
Number("3.14")    // returns 3.14
Number(" ")       // returns 0
Number("")        // returns 0
Number("99 88")   // returns NaN
```

In the chapter Number Methods, you will find more methods that can be used to convert strings to numbers:

Method	Description
parseFloat()	Parses a string and returns a floating point number
parseInt()	Parses a string and returns an integer

Automatic Type Conversion

When JavaScript tries to operate on a "wrong" data type, it will try to convert the value to a "right" type.

The result is not always what you expect:

```
5 + null    // returns 5         because null is converted to 0
"5" + null  // returns "5null"   because null is converted to "null"
"5" + 1     // returns "51"      because 1 is converted to "1"
"5" - 1     // returns 4         because "5" is converted to 5
```

Converting Dates to Numbers

The global method **Number()** can be used to convert dates to numbers.

```
d = new Date();
Number(d)          // returns 1404568027739
```

The date method **getTime()** does the same.

```
d = new Date();
d.getTime()        // returns 1404568027739
```

Javascript Timing Events

Método setInterval()

El método setInterval esperará un número espécifico de milisegundos, y luego ejecuta una función específica, y continuará ejecutando la función, una vez a cada intervalo de tiempo dado.

Sintaxis

`window.setInterval("javascript function", milliseconds);`

El método window.setInterval() puede ser escrito sin el prefijo window.

El primer parámetro de setInterval() debería ser una función.

El segundo parámetro indica el largo del intervalo de tiempo entre cada ejecución.

Nota: hay 1000 milisegundos en cada segundo.

Example

Alert "hello" every 3 seconds:

```
setInterval(function () {alert("Hello")}, 3000);
```

Como detener la ejecución?

El método clearInterval() se utiliza para detener la ejecución de la función especificada en el método setInterval().

Sintaxis

`window.clearInterval(intervalVariable)`

El método window.clearInterval() puede ser escrita sin el prefijo window.

Para ser capaz de utilizar el método clearInterval(), debes usar una variable global cuando creas el método interval:

```
myVar=setInterval("javascript function", milliseconds);
```

Después necesitarás ser capaz de detener la ejecución llamando el método clearInterval().

Example

Same example as above, but we have added a "Stop time" button:

```
<p id="demo"></p>

<button onclick="clearInterval(myVar)">Stop time</button>

<script>
var myVar = setInterval(function () {myTimer()}, 1000);
function myTimer() {
      var d = new Date();
      document.getElementById("demo").innerHTML = d.toLocaleTimeString();
}
</script>
```

El método setTimeout()

Syntaxis

```
window.setTimeout("javascript function", milliseconds);
```

El window.setTimeout() puede ser escrito sin el prefijo window.

El método setTimeout() esperará el número especificado de milisegundos, y luego ejecutar la función especificada.

El primer parámetro de setTimeout() debería ser una función.

El segundo parámetro indica el número de milisegundos, a partir del cual quieres

que se ejecute el primer parámetro.

Example

Click a button. Wait 3 seconds. The page will alert "Hello":

```
<button onclick = "setTimeout(function(){alert('Hello')},3000)">Try
it</button>
```

Como detener la ejecución?

El método clearTimeout() se utiliza para detener la ejecución de la función especificada en el método setTimeout().

Syntaxis

`window.clearTimeout(timeoutVariable)`

El método window.clearTimeout() puede ser escrito sin el prefijo window.

Para ser capaz de utilizar el método clearTimeout(), debes usar una variable global cuando creas el método interval:

`myVar=setTimeout("javascript function", milliseconds);`

Después, si la función no ha sido ejecutada, serás capaz de detener la ejecución llamando el método the clearTimeout().

```
<button onclick="myVar=setTimeout(function(){alert('Hello')},3000)">Try
it</button>

<button onclick="clearTimeout(myVar)">Try it</button>
```

JavaScript Errors - Throw and Try to Catch

The try statement lets you test a block of code for errors.

The catch statement lets you handle the error.

The throw statement lets you create custom errors.

The finally statement lets you execute code, after try and catch, regardless of the result.

```
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
try {
      adddlert("Welcome guest!");
}
catch(err) {
      document.getElementById("demo").innerHTML = err.message;
}
</script>

</body>
</html>
```

JavaScript try and catch

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The JavaScript statements try and catch come in pairs:

```
try {
    Block of code to try
}
catch(err) {
    Block of code to handle errors
}
```

JavaScript Throws Errors

When an error occurs, JavaScript will normally stop, and generate an error message.

The technical term for this is: JavaScript will throw an error.

The throw Statement

The throw statement allows you to create a custom error.

The technical term for this is: throw an exception.

The exception can be a JavaScript String, a Number, a Boolean or an Object:

```
throw "Too big";    // throw a text
```

```
throw 500;          // throw a number
```

If you use throw together with try and catch, you can control program flow and generate custom error messages.

Input Validation Example

```
<!DOCTYPE html>
<html>
<body>

<p>Please input a number between 5 and 10:</p>

<input id="demo" type="text">
<button type="button" onclick="myFunction()">Test Input</button>
<p id="message"></p>

<script>
function myFunction() {
    var message, x;
    message = document.getElementById("message");
    message.innerHTML = "";
    x = document.getElementById("demo").value;
    try {
        x = Number(x);
        if(x == "") throw "empty";
        if(isNaN(x)) throw "not a number";
        if(x > 10) throw "too high";
        if(x < 5) throw "too low";
    }
    catch(err) {
        message.innerHTML = "Input is " + err;
    }
}
</script>

</body>
</html>
```

The finally Statement

The finally statement lets you execute code, after try and catch, regardless of the result:

```
try {
    Block of code to try
}
catch(err) {
    Block of code to handle errors
}
finally {
    Block of code to be executed regardless of the try / catch result
}
```

Example

```
function myFunction() {
    var message, x;
      message = document.getElementById("message");
      message.innerHTML = "";
    x = document.getElementById("demo").value;
    try {
      x = Number(x);
        if(x == "") throw "is empty";
        if(isNaN(x)) throw "is not a number";
        if(x > 10) throw "is too high";
        if(x < 5) throw "is too low";
      }
    catch(err) {
      message.innerHTML = "Error: " + err + ".";
      }
    finally {
          document.getElementById("demo").value = "";
      }
}
```

Another very cool shortcut notation is the ternary notation for conditions. So, instead of the following...

```
var direction;
if(x < 200){
  direction = 1;
} else {
  direction = -1;
}
```

... You could write a shorter version using the ternary notation:

```
var direction = x < 200 ? 1 : -1;
```

The `true` case of the condition is after the question mark, and the other case follows the colon.

7. FUNCTIONS CAN EXECUTE THEMSELVES

There's no denying it:

```
(function() { alert('hello'); })(); //alerts 'hello'
```

The syntax is simple enough: we declare a function and immediately call it just as we call other functions, with `()` syntax. You might wonder why we would do this. It seems like a contradiction in terms: a function normally contains code that we want to execute later, not now, otherwise we wouldn't have put the code in a function.

One good use of self-executing functions (SEFs) is to **bind the current values of variables** for use inside delayed code, such as callbacks to events, timeouts and intervals. Here is the problem:

```
var someVar = 'hello';
setTimeout(function() { alert(someVar); }, 1000);
var someVar = 'goodbye';
```

Newbies in forums invariably ask why the `alert` in the `timeout` says `goodbye` and not `hello`. The answer is that the `timeout` callback function is precisely that—a callback—so it doesn't evaluate the value of `someVar` until it runs. And by then, `someVar` has long since been overwritten by `goodbye`.

SEFs provide a solution to this problem. Instead of specifying the timeout callback implicitly as we do above, we return it from an SEF, into which we pass the current value of `someVar` as arguments. Effectively, this means **we pass in and isolate the current value of `someVar`, protecting it from whatever happens to the actual variable `someVar` thereafter**. This is like taking a photo of a car before you respray it; the photo will not update with the resprayed color; it will forever show the color of the car at the time the photo was taken.

```
var someVar = 'hello';
setTimeout((function(someVar) {
    return function()  { alert(someVar); }
})(someVar), 1000);
var someVar = 'goodbye';
```

This time, it alerts `hello`, as desired, because it is alerting the isolated version of `someVar` (i.e. the function argument, *not* the outer variable).

Summary

The `do...while` statement creates a loop that executes a specified statement until the test condition evaluates to false. The condition is evaluated after executing the statement, resulting in the specified statement executing at least once.

Syntax

```
do
   statement
while (condition);
```

`statement`

A statement that is executed at least once and is re-executed each time the condition evaluates to true. To execute multiple statements within the loop, use a `block` statement (`{ ... }`) to group those statements.

`condition`

An expression evaluated after each pass through the loop. If `condition` evaluates to true, the `statement` is re-executed. When `condition` evaluates to false, control passes to the statement following the `do...while`.

Examples

Example: Using `do...while`

In the following example, the `do...while` loop iterates at least once and reiterates until `i` is no longer less than 5.

```
var i = 0;
do {
   i += 1;
   console.log(i);
} while (i < 5);
```

Summary

The `do...while` statement creates a loop that executes a specified statement until the test condition evaluates to false. The condition is evaluated after executing the statement, resulting in the specified statement executing at least once.

Syntax

```
do
   statement
while (condition);
```

`statement`

A statement that is executed at least once and is re-executed each time the condition evaluates to true. To execute multiple statements within the loop, use a `block` statement (`{ ... }`) to group those statements.

`condition`

An expression evaluated after each pass through the loop. If `condition` evaluates to true, the `statement` is re-executed. When `condition` evaluates to false, control passes to the statement following the `do...while`.

Examples

Example: Using `do...while`

In the following example, the `do...while` loop iterates at least once and reiterates until `i` is no longer less than 5.

```
var i = 0;
do {
   i += 1;
   console.log(i);
} while (i < 5);
```

Example: Using `for`

The following `for` statement starts by declaring the variable `i` and initializing it to `0`. It checks that `i` is less than nine, performs the two succeeding statements, and increments `i` by 1 after each pass through the loop.

```
for (var i = 0; i < 9; i++) {
   console.log(i);
   // more statements
}
```

Example: Optional `for` expressions

All three expressions in the head of the `for` loop are optional.

For example, in the *initialization* block it is not required to initialize variables:

```
var i = 0;
for (; i < 9; i++) {
    console.log(i);
    // more statements
}
```

Like the *initialization* block, the *condition* block is also optional. If you are omitting this expression, you must make sure to break the loop in the body in order to not create an infinite loop.

```
for (var i = 0;; i++) {
   console.log(i);
   if (i > 3) break;
   // more statements
}
```

Factory Constructor Pattern

This pattern is special, because it doesn't use `"new"`.

The object is created by a simple function call, similar to *Python-style*:

```
var animal = Animal("fox")
```

```
var rabbit = Rabbit("rab")
```

`Declaration`

The constructor is defined as a function which returns a new object:

```
function Animal(name) {

  return {
    run: function() {
      alert(name + " is running!")
    }
  }

}
```

usage:

```
var animal = Animal("fox")
animal.run()
```

Inheritance

`Rabbit` is made by creating an `Animal`, and then mutating it:

```
function Rabbit(name) {

  var rabbit = Animal(name) // make animal

  rabbit.bounce = function() { // mutate
    this.run()
    alert(name + " bounces to the skies! :)")
  }

  return rabbit // return the result
}

var rabbit = Rabbit("rab")
rabbit.bounce()
```

Private/protected methods (encapsulation)

Local variables and functions become private:

```
01 function Bird(name) {
02
03   var speed = 100                        // private prop
04   function openWings() { /* ... */ } // private method
05
06   return {
07     fly: function() {
08       openWings()
09       this.move()
10     },
11     move: function() { /*...*/ }
12   }
13 }
```

The code above looks simple, but still there is a gotcha.

A public method can be called as `this.move()` from another public method, but *not* from a private method.

A private method like `openWings` can't reference `this`. There's no reference to the new object in a local function.

One way to solve that is to bind the new object to a local variable prior to returning:

```
01 function Bird(name) {
02
03   function doFly() {
04     openWings()
05     self.move()
06   } // private method
07
08
09   var self ={
10     fly: function() { doFly() },
11     move: function() { /*...*/ }
12   }
13   return self
14 }
```

Summary

- The *factory constructor* uses a function which creates an object on it's own without `new`.
- Inheritance is done by creating a parent object first, and then modifying it.
- Local methods and functions are private. The object must be stored in closure prior to

returning if we want to access it’s public methods from local ones.

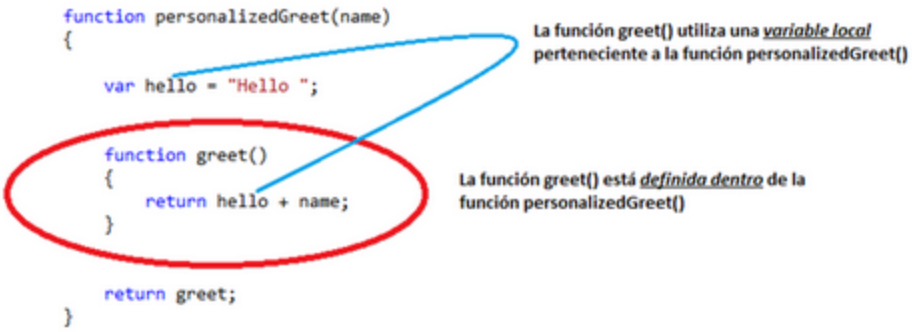
Whenever you see the function keyword within another function, the inner function has access to variables in the outer function.

```
function foo(x) {
  var tmp = 3;

  function bar(y) {
    alert(x + y + (++tmp)); // will alert 16
  }

  bar(10);
}

foo(2);
```



Javascript The Good Parts

POO(Programacion Orientada a Objetos)

Simula67 fue el programa de simulaciones de naves en que nació la POO creado por Ole-Johan Dahl y Kristen Nygaard, del Centro de Cómputo Noruego en Oslo.

Objeto: Instancia de una clase, con sus comportamientos y datos que reaccionan a eventos.

Los objetos se identifican por medio de tablas compuesta por sus atributos (data) y funciones correspondientes(métodos)

Los objetos tienen:

Estado: información, data. Uno o mas atributos.

Comportamiento: en que se puede utilizar, que se puede hacer con el en operaciones.

Identidad: La diferencia.

Con normalidad se separa el estado del comportamiento del objeto, en POO se crean objetos al los cuales se les pide que cambien su métodos por si solos. La variable en POO es un simple contenedor de dará y la función es lo que se puede hacer con la data de este objeto

(Métodos = comportamiento. Atributos = estado)

DEFINICIONES IMPORTANTES

Clase: Define los métodos y propiedades de un tipo de objeto.

Instanción: Lectura de método y propiedad para creación de objetos tras lectura.

Herencia: Traspaso de atributos, métodos inclusive variables publicas de clase a clase, componentes privados son invisibles para el lector con naturalidad aunque si son traspasados también.

Método: Comportamiento de un objeto(s) tras la llegada de un mensaje. Este puede cambiar la data de un objeto y generar un evento, direccionado a otro objeto.

Evento: Reacción posible que se puede desencadenar por un objeto.

Atributos: Características de la clase, diferencias de otras.

Mensaje: Medio por el cual Se le pide a un objeto que ejecute métodos con parámetros cambiables, asociados a el evento que genero el evento mismo.

Propiedad/artibuto:Contenedor de datos de un objeto(s) contribuye a la extracción de tales datos que pueden ser cambiados por métodos.

Estado interno:Variable privada alcansable y cambiable por metodos, indica situacione posibles de un objeto(clase de obj).

Java ¡= Javascript

JAVASCRIPT es un lenguaje multiplataformico, orientado a objetos, diseñado para trabajar con otros productos y crear control programático sobre estos. JS tiene tanto objetos predeterminados, como math y date, y se presta para declarar otros objetos.Netscape invento js, que obviamente fue utilizado primero en este browser.

LiveConnect permite interacción entre java y js para poder interactuar con objetos de un lenguaje por medio del otro.

JS soporta el sistema en tiempo de ejecución, representando val numéricos, val bloanos y strings.

Propositos:

1.JS del lado del cliente proporciona objetos para controlar el browser,que sea interactivo, y su Modelo de Documentos Orientados a Objeto (DOM).

2.JS del lado del servidos,permite interacion con base de datos, ya sean pedidos o cambias de este.

JS=/=JAVA:

JS carece de typing estricto y comprobaciones tipos fuerte, o deberes al programar, contiene sintaxis, convención de nombre y básicos controles similares a los de java, razón por la cual cambia de live- a javascript.

JS esta basado en prototipos, mas que en clases, lo cual le brinda herencia dinámica, permite funciones sin requerimientos declarativos especiales, o deberes.

Java es programación basada en clases, y exclusivamente se basa en clases y sus métodos, y su requerimientos fueres, o deberes, lo hacen mas complejo que JS.

Diferencias paso por paso:

Orientación a Objetos: Igual tipo de objetos, la herencia es por el mecanismo de prototipo, añadimiento de propiedades y métodos es dinámico.

Orientación a clases: Los objetos son divididos en clases con toda la herencia que su clase dicta, la cual no es cambiada dinámicamente.

En JS el tipo de data no tiene que ser declarada, no es un deber, por lo tanto es typing dinámico, no estático como java.

JS y espicificaciones del ECMAscript

Netscape trabaja con ECMa intenational, quienes estandarizan tomando en cuenta la base de JS, js estandarizado se llama ECMscript, documentado en las especificaciones ECMA-262, aprobada tambien por ISO (international for for standardization).

Este no describe el DOM, el cual es un destandar de W3C, el cual define la forma que los documentos HTML son expuestos al ¨script¨.

Relaciones entre las ediciones de js y ECMAscript

JS1: La primer edición de ECMAscript esta basada en js vs1

JS 1.2 ECMAscript vs1 no fue completada para este momento, JS1.2 no era completamente compatible con esta por las siguientes razones:

JS1.2 tenia features adicionales.

ECMAscript tenía dos nuevos features: Internacionalismo usando Unicode y comportamiento uniforme.

JS1.3 Completamente compatible con ECMA262 vs1

Se resuelve todas las inconsistencias de relación que la versión pasada, mantiene todas las features adicionales, mas que == y != para bien con el ECMA-262

JS1.4 Compatible también con ECMA262 vs1, aunque vs3 no fue finalizada antes de JS1.4

JS1.5 Completamente compatible con EMCAv3

(JS siempre tendrá features que ECMA262 no tendrá, mientras es compatible.)

JavaScript Documentation versus the ECMAScript Specification

Especificaciones del EMCAScript :

Set de reglas que especifican que se puede hacer en JS que servirá en el ECMAscript, dado a que JS suele tener mas aplicaciones, aunque cada una tiene individualiades, Js suele cubrir toda funcionalidad de las especificaciones de ECMAscript.

A diferencia de la documentación de JS, esta no le dice al programador como escribir JS.

Ni la globalización de objetos, como el no-parametro (zero-arguement), no se encuentran en la documentación de JS por falta de uso o por uso indirecto.

Que es scripting language?

Un lenguaje de scripting, es el que pide a otra aplicación actuar con sus elementos de ciertas maneras, mas que crear una aplicación especifica para lograr esto.

La línea entre scripting y programing languages es muy delgada, y tiende a depender del área de trabajo y su fin.

Que es V8?

Motor que Google Chrome utiliza para la lectura de documentos JS, lo compila y lo transforma en código para maquina, en lugar de solo interpretarlo.

Que es JScript?

Programa para web por excelencia, y el fallo de Java en el campo, que a diferencia de este, es de simple escritura, osea, no requiere un nivel de detalleo grande o fuerte como el que un programa como JAVA tiene. JScript es fácil de usar, pero no de masterizar.

El estándar del programa es estipulado por ECMAScript 262, aunque no abarca su plenitud.

Comentarios.

Se recomienda el uso de // para comentar dado a que /* */ puede complicar las cosas al no limitar código, que podría ser analizado, pero no como comentario.

CAPITULO 2

Nombres.

No empezar con nada mas que una letra, después de esta se pueden usar caracteres y números.

Palabras restringidas:

abstract boolean break byte case catch char class const continue debugger default delete do double else enum export extends false final finally float for function goto if implements import in instanceof int interface long native new null package private protected public return short static

super switch synchronized this throw throws transient true try typeof var volatile void while with.

Numeros.

La diferencia que se marca en diferentes programas con el num. num es nula.

1 y 1.0 son lo mismo.

NaM es igual a nada, incluyéndose, isNaN(number) function lo demuestra.

- Representa valores negativos

infinity equivale a cualquier numero mayor que 1.79769313486231570e+308.

Var Strings.

Estas están entre ¨string¨, no contienen caracteres, por lo cual hay un catalogo de formas de incluirlas, aunque mejor, no provocar mas muros por delante.

strings tienen el .lengt, para checkear su largo.

Estas se pueden sumar con otras strings para formar una total, y tienen métodos.

Statements

Estas empiezan de arriba para abajo, hay varios tipos de estas que pueden diferenciar el flujo, como:

Las condicionales (if and switch)

Las infinitas, looping (while, for, and do)

Las de ruptura (break, return, and throw),

Y la invocación de una funcion

Por su naturaleza con el bloque, las var se especifican al inicio del código, no por bloque,

Expresion

Se utilizan para asignar valor a una o mas var, invocar un método, borrar

una propiedad de un objeto.

Literals

Parece ser la forma mas fácil de crear objetos, los nombres, la propiedad de el objeto se puede especificar como nombre o string, los nombres son tratados literalmente como tal y no cambian a vars

Funcions

Funciones literales, lo que hacen es crear parámetros con los cuales se juegan como vars dentro de su cuerpo. Estas pueden llevar nombres opcionales para ser llamadas y trabajadas de formas especificas.

CAPITULO 3

La simpleza de js cae en sus strings, numbers, booleans, y undefineds, el resto es una colección de objetos, estos ¨mutables¨ a diferencia de los demás.

Los objetos contienen propiedades, y estas valores, al tener un nombre, que puede ser tratado como var, es todo aceptable menos el var undefined.

tienen la propiedad prototype, capas de traspasar herencia de objeto a objeto.

Object literals

Apararecen donde una expresión puede aparecer, sirven para asignar valores a un objeto, de 0 para arriba. (can be empty, but not undefined).

Las comillas en los nombres de un objeto son opcionales mientras no se use una palabra reservada.

Retrieval

Los valores de un objeto son extraíble, se puede de dos maneras:

nameOfobject[property] // value

nameOfobject.property // value

Si se intenta llamar a una propiedad inexistente, el valor sera undefined.

Update

El value de la propiedad de un objeto, puede ser remplazado tan solo asignano otro value encima.

nameOfobject.property = “a new one”;

Prototype

Todos los objetos tienen el prototype object por usar, todos los objects provenientes de un literal objects están linkeados a prototype.

La propiedad prototype, se mantiene incambiada, inclusive al hacer cambios en el objeto que la contiene.

El link de propiedades solo se usa para extraer info. Si no se encuentra un nombre de la propiedad buscada, se busca en prototype, si no se encuentra allí, se guarda en esa posición.

cuando no se encuentra la propiedad en la cadena de prototype, el resultado es undefined, proceso llamado delegation.

Reflecting

Nos ayuda a checkear el tipo de propiedad con la que se trata, se espera no tener funciones de vuelta al usar typeOf object.property, usualmente se busca data y no funciones, la cual se soluciona asi:

flight.hasOwnProperty('number') // true

flight.hasOwnProperty('constructor') // false

Enumeration

El for in statement puede hacer un loop, al enumerar todas las propiedades de un objeto, incluyendo prototype y funciones, asi que es

necesario hacer un filtro, el mas usado es el hasOwnProperty usando typeof

```
var name; for (name in another_stooge) {
if (typeof another_stooge[name] !== 'function') {
document.writeln(name + ': ' + another_stooge[name]);
}
 }
```

No habra un orden de propiedades de esta manera, razon por la cual, un array se puede usar

```
 var i;
var properties = [ 'first-name', 'middle-name', 'last-name', 'profession' ]
; for (i = 0; i < properties.length; i += 1) {
document.writeln(properties[i] + ': ' + another_stooge[properties[i]]);
}
```

Delete

Permite borrar una propiedad de un objeto, un objeto, o un espacio de un array, para que una propiedad del linkage de prototype aparezca.

Para borrar una propiedad se necesita un statemante

Globalización

Se espera tener una sola var que mantenga toda la aplicación, para mantener mas seguridad y mejor run-time.

Datos extras:

El typeOf null =/= undifined, es un objeto

No más iso de New object{}, just, no.

null se transforma al typeOf de lo que sea que se está añadiendo, ejem:

5+null // 5

“5”+null // “5null”

“5”+1 // “51”

“5”-1// 4

para cambiar un string a number se usa Number() //

CAPITULO 4

Argument

Parametro extra añadido al invocar una funcion, contiene una cantidad de parametros indefinidos, por un error de diseño es parecido a un array sin serlo.

Return

funciona para retornar algo, antes de salir de la funcion,

Si no se espefifica lo que se quiere retornar, undefined es el valor mostrado.

Exception.

Objeto contenido en un throw, que se enviara a el catch de un try, como parametro.

El bloque de catch manejara todas las exceptions, asi que deberemos fijarnos en el name property para diferenciarlas

Augmenting

Es algo complicado, pero basicamente el añadimiento de metodos para facilitat 'bugs' que ja tiene, como el redondeo de numeros o espacios al final de toda string, aunque hacer esto es un statement publico, se puede utilizat este metodo para incluirle SOLO si no existe ya.

Recursive

La funcion que se llama a a

Si misma para dividir los problemas en subproblrmas que resulven el total.

Scope

controla la visibilidad de las var.

Se recomienda declarar previo al uso de tales var

Closure

Acceso a parametros para funciones de funciones padre, para privatisar info

Callback

Una forma de responder hasta tener lo ocupado, para optimizar tiempo.

Module

Creado por closures y funciones, es un objeto

No lo entendi

Cascade

No todo metodo tiene un valor para retornar. Permite escribir como en un objeto de ajax

Curry

Nos permite producor una nueva funcion al combinat una funcion con un

argumento

Memoization

Las funciones pueden guardar en objetos resultados de operaciones previas

CAPITULO 5

Inheritance.

En js tiene muchas mas opciones para reuso de código, las cuales se extienden desde las clásicas a nuevas y creativas.

Pseudoclasical

Usa la function constructor como formas de clases , lo cual puede traer problemas.

Object specifiers

A veces, cuando un constructor tiene muchos parámetros, se les puede ordenar, para bien, especificando un nombre de parámetro

Prototypal

Una forma más simple de tratar clases al hacerlo pura base de objetos. Al diferenciar un objeto de quien viene.

Functional

Por falta de privacidad en las previas formas, podemos crear privacidad al

1. Crear un objeto (puede ser un object literal, un constructor, o usar object.create)

2. Se definen vars y methods privados

3. Se aumenta el objeto creado con métodos que tendrán acceso a parámetros y variables ya definidos

4. Se retorna tal objeto.

También nos da una forma de lidiar con super methods.

Parts

Object.create method creates new instances for a new object

CAPITULO 6

Array literal:

Arrays pueden contener data, inclusive de diferentes tipos dentro de si, puede ser producida con un 0 también.

Hay diferencias entre un objeto y un array, empezando con la capacidad de .length

Lenght

0 o vacío por si solo es una propiedad.

Este cuenta la cantidad de propiedades del array.

Delete

Como los arrays funcionan como objetos en js, se puede usar el delete para borrar espacios, aunque esto serán cambiados por un 'undefined' asi es mejor usar splice, dictando el lugar del array y luego la cantidad de propiedades por remover

Enumeration

Un array puede incrementar su tamaño por el método for.

Confusion

Los array son muy parecidos a los objetos, para poder distinguibles se puede chequear si se tiene un lenght, splice y si es verdadera.

Array.methods

Los arrays también tienen la capacidad de ser aumentados.

No combinar el object.create en un array puesto a que puede que se hereden propiedades, el objeto no tendrá el .lenght

Dimensions

Como preparar un array cuando está 'vacío' para actuar como si realmente lo estuviese, o no.

Se pueden meter arrays en arrays, o bi-dimensiones arrays

CAPITULO 7

Regular Expressions

Una expresión regular es la especificación de la sintaxis de un lenguaje sencillo. regular

expresiones se usan con los métodos para buscar, reemplazar y extraer información de

cuerdas. Los métodos que funcionan con expresiones regulares son

regexp.exec, regexp.test, string.match, string.replace, string.search, and string.split.

Ejemplo

var parse_url = /^(?:([A-Za-z]+):)?(\/{0,3})([0-9.\-A-Za-z]+)(?::(\d+))?(?:\/([^?#]*))?(?:\?([^#]*))?(?:#(.*))?$/;

var url = “http://www.ora.com:80/goodparts?q#fragment”;

Vamos a llamar al metodo de ejecucion de parse_url

Si coincide con éxito la cadena que se pasa

que, devolverá una matriz que contiene piezas extraídas de la url:

var url = "http://www.ora.com:80/goodparts?q#fragment";

var result = parse_url.exec(url); var names = ['url', 'scheme', 'slash', 'host', 'port', 'path', 'query', 'hash']; var blanks = ' ';

var i;

for (i = 0; i < names.length; i += 1) { document.writeln(names[i] + ':' + blanks.substring(names[i].length), result[i]);

}

This produces: url: http://www.ora.com:80/goodparts?q#fragment scheme: http

slash: //

host: www.ora.com

port: 80

path: goodparts

query: q

hash: fragment

Las expresiones regulares no pueden romperse en pedazos más pequeños la forma en que las funciones pueden,

por lo que la pista que representa parse_url es larga.

El carácter ^ indica el comienzo de la cadena. Es un ancla que impide

exe de saltándose un prefijo-como no-URL:

(?:([A-Za-z]+):)?

Este factor coincide con un nombre de esquema, pero sólo si es seguido por un: (colon). The (?:...) indica un grupo que no captura. El sufijo ? indica que el grupo es opcional.

CAPITULO 8

Metodos

Array

array.concat(item…)

El método concat produce una nueva matriz que contiene una copia superficial de esta matriz con el

artículos anexan a la misma. Si un artículo es una matriz, entonces cada uno de sus elementos se adjunta por separado.

array.join(separator)

El método join hace una cadena de una matriz. Esto se hace haciendo una cadena de cada uno de

elementos de la matriz y, a continuación, la concatenación de todos ellos junto con un separador entre

ellos. El separador predeterminado es ','. Para unirse sin separación, utilice una cadena vacía como la

separador.

array.pop( )

Los métodos pop y push hacen un trabajo conjunto como una pila. El método pop elimina y

devuelve el último elemento de este vector. Si la matriz está vacía, devuelve indefinido.

var a = ['a', 'b', 'c'];

var c = a.pop (); // A es ['a', 'b'] y c es 'c'

pop puede ser implementado como esto:

```
Array.method ('pop', function () {
  return this.splice (this.length - 1, 1) [0];
```

array.push(item…)

El método push añade elementos al final de una matriz. A diferencia del método concat, modifica

la matriz y anexar elementos de matriz entera. Se devuelve la nueva longitud de la matriz:

});

array.reverse( )

El método modifica la matriz inversa invirtiendo el orden de los elementos. devuelve

la matriz.

array.shift( )

El método de shift elimina el primer elemento de una matriz y devuelve. Si la matriz es

vacía, devuelve undefined. desplazamiento suele ser mucho más lento que el pop.

array.slice(start,end)

El método slice hace una copia superficial de una porción de una matriz. El primer elemento copiado

será array [Inicio]. Se detendrá antes de copiar array [final]. El parámetro

final es opcional,

y el valor predeterminado es Array.length. Si alguno de los parámetros es negativo, se añadirá Array.length

a ellos en un intento de hacerlos no negativo. Si inicio es mayor que o igual a la matriz.

longitud, el resultado será una nueva matriz vacía. No se debe confundir slice con splice.

array.sort(comparefn)

El método sort ordena el contenido de una matriz en su lugar. Se clasifica series de números

incorrectamente:

JavaScript función de comparación por defecto asume que los elementos a ser ordenados son cadenas.

No es lo suficientemente inteligente como para probar el tipo de los elementos antes de compararlos, lo que convierte

los números a cadenas como las compara, asegurando un resultado sorprendentemente incorrecta.

Afortunadamente, es posible reemplazar la función de comparación con el suyo propio. Su comparación

función debe tomar dos parámetros y devuelve 0 si los dos parámetros son iguales, un negativo

número si el primer parámetro debe ser lo primero, y un número positivo si el segundo

parámetro debe ser lo primero. (Los veteranos podrían ser recordados de la aritmética FORTRAN II

IF.)

array.splice(start, deleteCount, item…)

El método de splice elimina elementos de una matriz, reemplazándolos con nuevos elementos. el

parámetro de inicio es el número de una posición dentro de la matriz. El parámetro es deleteContent

el número de elementos para eliminar a partir de esa posición. Si hay parámetros adicionales,

esos elementos se insertan en la posición. Devuelve una matriz que contiene el borrado

elementos.

El uso más popular de splice es borrar elementos de una matriz. No confunda splice

con slice:

array.unshift(item…)

El método unshift es como el método push excepto que empuja a los elementos en la parte frontal

de esta matriz en lugar de al final. Devuelve la matriz nueva longitud:

function.apply(thisArg, argArray)

El método de aplicación invoca una función, pasando el objeto que se une a esto y

un conjunto opcional de argumentos. El método de aplicación se utiliza en el patrón de aplicar invocación

number.toExponential(fractionDigits)

El método toExponential convierte este número en una cadena en forma exponencial. la

fractionDigits parámetro opcional controla el número de decimales. Debería ser

entre 0 y 20:

number.toFixed(fractionDigits)

El método toFixed convierte este número en una cadena en forma decimal. el opcional

fractionDigits parámetro controla el número de decimales. Debe estar entre 0

y 20. El valor predeterminado es 0:

number.toPrecision(precision)

El método toPrecision convierte este número en una cadena en forma decimal. el opcional

parámetro de precisión controla el número de dígitos de precisión. Debe estar entre 1 y

21:

number.toString(radix

El método toString convierte este número en una cadena. El parámetro opcional radix

controla radix, o base. Debe estar entre 2 y 36. Los radix defecto es base 10. El

parámetro radix se utiliza más comúnmente con números enteros, pero se puede utilizar en cualquier número.

El caso más común, Number.toString (), se puede escribir más simplemente como String (número):

object.hasOwnProperty(name)

El método hasOwnProperty devuelve true si el objeto contiene una propiedad que tiene el nombre.

No se examina la cadena de prototipo. Este método es inútil si el nombre es hasOwnProperty:

regexp.exec(string)

El método exec es el más poderoso (y más lento) de los métodos que utilizan regularmente

expresiones. Si coincide con éxito la expresión regular y la cadena, devuelve una matriz. el 0

elemento de la matriz contendrá la subcadena que coincide con la expresión regular. El elemento 1 es

el texto capturado por el grupo 1, el elemento 2 es el texto capturado por el grupo 2, y así sucesivamente. si

la comparación falla, devuelve nulo.

regexp.test(string)

El método de prueba es el más sencillo (y más rápido) de los métodos que utilizan expresiones regulares.

Si luego regex coincide con la cadena, devuelve true; de lo contrario, devuelve false. No utilice el

g, marca con este método:

string.charAt(pos)

El método charAt devuelve el carácter en la posición pos en esta cadena. Si pos es menos de

cero o mayor que o igual a String.length, devuelve la cadena vacía. JavaScript hace

no tener un tipo de carácter. El resultado de este método es una cadena:

string.charCodeAt(pos)

El método charCodeAt es el mismo que charAt excepto que en lugar de devolver una cadena,

devuelve una representación entera del valor de punto de código del carácter en la posición pos en

esa cadena. Si pos es menor que cero o mayor que o igual a string.length, devuelve NaN:

string.concat(string…)

El método concat hace una nueva cadena concatenando otras cadenas juntas. Rara vez es

usado porque el operador + es más conveniente:

string.indexOf(searchString, position)

El método indexOf Busca una cadena de búsqueda dentro de una cadena. Si se encuentra, devuelve

la posición del primer carácter emparejado; de lo contrario, devuelve -1. La posición opcional

parámetro hace que la búsqueda para comenzar en alguna posición especificada en la cadena:

string.lastIndexOf(searchString, position)

El método lastIndexOf es como el método indexOf, excepto que busca desde el final de

la cadena en lugar de la parte delantera:

string.localeCompare(that)

El método localeCompare compara dos cadenas. Las reglas de cómo las cadenas son

comparación no se especifican. Si esta cadena es menor que la cadena, el resultado es negativo. si

son iguales, el resultado es cero. Esto es similar a la convención para la array.sort

función de comparación:

string.match(regexp)

El método partido coincide con una cadena y una expresión regular. Cómo lo hace esto depende

la bandera g. Si no hay ningún indicador g, entonces el resultado de la

llamada String.match (regexp) es el mismo que

regexp.exec llamando al (cadena). Sin embargo, si la expresión regular tiene el indicador g, entonces se produce una matriz

de todos los partidos, pero excluye a los grupos de captura:

string.replace(searchValue, replaceValue)

El método replace hace una búsqueda y reemplazar operación en esta cadena, la producción de un nuevo

cadena. El argumento de búsqueda de valor puede ser una cadena o un objeto de expresión regular. Si es una

cadena, sólo se reemplaza la primera aparición del valor de búsqued

string.search(regexp)

El método de búsqueda es como el método indexOf, excepto que toma una expresión regular

objeto en lugar de una cadena. Devuelve la posición del primer carácter del primer partido, si

hay uno, o -1 si la búsqueda falla. El indicador g se ignora. No hay parámetro de posición:

string.slice(start,end)

El método slice hace una nueva cadena copiando una parte de otra cadena. Si el inicio

parámetro es negativo, se añade string.length a ella. El parámetro final es opcional, y su

valor predeterminado es string.length. Si el parámetro final es negativo, entonces se añade string.length

a la misma. El parámetro final es uno mayor que la posición del último carácter. Para obtener n caracteres

comenzando en la posición p, u se string.slice (p, p + n). También vea String.substring y

Array.slice, después y antes en este capítulo, respectivamente.

string.split(separator, limit)

El método split crea una matriz de cadenas mediante el fraccionamiento de esta cadena en trozos. la

parámetro de límite opcional puede limitar el número de piezas que se pueden dividir. el separador

parámetro puede ser una cadena o una expresión regular.

Si el separador es la cadena vacía, una serie de personajes individuales se produce:

string.substring(start,end)

El método subcadena es el mismo que el método rebanada excepto que no controla el

ajuste de parámetros negativos. No hay ninguna razón para utilizar el método subcadena. uso

cortar en su lugar.

string.toLocaleLowerCase( )

El método toLocaleLowerCase produce una nueva cadena que se hizo mediante la conversión de este

cadena a minúsculas utilizando las reglas para la configuración regional. Esto es principalmente para el beneficio de

Turco porque en ese idioma "yo" se convierte en I, no 'i'.

string.toLocaleUpperCase( )

El método toLocaleUpperCase produce una nueva cadena que se hizo mediante la conversión de este

cadena a mayúsculas utilizando las reglas para la configuración regional. Esto es principalmente para el beneficio de

Turco, porque en ese idioma 'i' se convierte en ", no 'I'.

string.toLowerCase( )

El método toLowerCase produce una nueva cadena que se hizo mediante la conversión de esta cadena para

minúsculas.

string.toUpperCase( )

El método toUpperCase produce una nueva cadena que se hizo mediante la conversión de esta cadena para

mayúsculas.

String.fromCharCode(char…)

La función String.fromCharCode produce una cadena a partir de una serie de números.