



Bucles: sentencias while, for y do/while

Bucle

```
mod01 — bash — 56x5
venus:mod01 jq$
venus:mod01 jq$
venus:mod01 jq$ node 44-veces_do.js
La frase "la casa roja" tiene 4 veces la letra a
venus:mod01 jq$
```

- ◆ Un bucle es una **secuencia o bloque de instrucciones**
 - que **se repite** mientras se cumple una **condición** de permanencia
- ◆ Un bucle se controla con 3 elementos,
 - normalmente asociados a una variable(s) de control del bucle
 - ◆ **Inicialización:** fija los valores de arranque del bucle
 - ◆ **Permanencia en bucle:** indica si se debe volver a ejecutar el bloque
 - ◆ **Acciones de final bloque:** actualiza en cada repetición las variables de control
- ◆ Ilustraremos los bucles (**while**, **for** y **do/while**) con la **función veces**
- ◆ **veces(..)** acepta **letra** y **frase** como primer y segundo parámetros
 - Y devuelve el número de veces que la frase contiene la letra

función veces(..): bucle while

El ejemplo ilustra el bucle while con la **función veces**. En este, variables "i" y "n" se inicializan a cero antes del bucle.

La condición de permanencia en el bucle (expresión entre paréntesis después de palabra while) determina si se permanece en el bucle o se sale, según esté sea true o false.

El bucle recorre el array con el índice "i" e incrementa la variable "n" si la letra del string coincide con la de referencia.

La variable "i" se incrementa durante la evaluación de la condición de la sentencia if con **post-incremento**: "i++".

Al finalizar se devuelve "n" (numero de coincidencias).

```
mod01 — bash — 50x5
venus-5:mod01 jq$
venus-5:mod01 jq$
venus-5:mod01 jq$ node 40-veces_while.js
La frase "la casa roja" tiene 4 veces la letra a
venus-5:mod01 jq$
```

```
40-veces_while.js
function veces (letra, frase) {
  var i = 0, n = 0;
  while ( i < frase.length ) {
    if ( letra === frase[i++] ) { ++n; };
  }
  return n;
};

var l='a', f='la casa roja';
console.log('La frase "' + f + '" tiene '
+ veces(l,f) + ' veces la letra ' + l);
```

función veces(..): bucle while + continue

```
mod01 — bash — 50x5
venus-5:mod01 jq$
venus-5:mod01 jq$
venus-5:mod01 jq$ node 41-veces_while_continue.js
La frase "la casa roja" tiene 4 veces la letra a
venus-5:mod01 jq$
```

En este ejemplo se ilustra el uso de la sentencia **continue**, dentro de un bucle **while** es similar al del ejemplo anterior. El bucle recorre el array con el índice "i" e incrementando la variable "n" si la letra indexada en el string coincide con la de referencia.

La condición de la **sentencia if** es aquí la negación de la del ejemplo anterior, de forma que se invoque la **sentencia continue** para volver al principio del bucle sin incrementar "n", si no hay que incrementarla. En el tutoría de JS pueden verse otros usos de **continue**.

La variable "i" se incrementa durante la evaluación de la condición de **if** con post-incremento: **i++**.

```
function veces (letra, frase) {
  var i = 0, n = 0;           // inicialización del bucle
  while ( i < frase.length ) { // condición de permanencia
    if ( letra !== frase[i++] ) // compara e incrementa índice
      { continue };           // vuelve a comienzo del bucle
    ++n;                       // acción del bucle
  }
  return n;
};

var l='a', f='la casa roja';
console.log('La frase "' + f + '" tiene '
+ veces(l,f) + ' veces la letra ' + l);
```

función veces(..): bucle while + break

```
mod01 — bash — 50x5
venus-5:mod01 jq$
venus-5:mod01 jq$
venus-5:mod01 jq$ node 42-veces_while_break.js
La frase "la casa roja" tiene 4 veces la letra a
venus-5:mod01 jq$
```

En este ejemplo se ilustra el uso de la sentencia `break`, dentro de un bucle `while`, que se ha transformado en un **bucle infinito al definir la condición de permanencia como `true`**.

La primera sentencia **sentencia `if`** define la permanencia en el bucle, porque en cuanto se cumpla, se ejecuta la sentencia `break`, que finaliza la ejecución del bucle. La sentencia `break` tiene mas posibilidades, que pueden ver e en el tutoría de JS:

En la segunda sentencia se incrementa la variable "`i`" durante la evaluación de la condición con post-incremento (`i++`) y si la letra indexada en el string coincide con la de referencia, se incrementa "`n`".

```
function veces (letra, frase) {
  var i = 0, n = 0; // inicialización del bucle
  while (true) {
    if ( i >= frase.length ) { break; }; // condición de salida de bucle con break
    if ( letra === frase[i++] ) { ++n; }; // incrementa índice, y n si coincide con letra
  }
  return n;
};

var l='a', f='la casa roja';
console.log('La frase "' + f + '" tiene '
  + veces(l,f) + ' veces la letra ' + l);
```

función veces(..): bucle for

La **función veces** ilustra aquí a **sentencia for** de gestión del bucle. La gestión del bucle (entre paréntesis) consta de tres partes separadas por ";":

- 1) **Inicialización**: define e inicializa las variables "i" y "n"
- 2) **Condición de permanencia en el bucle**: se evalúa a true o false para determinar si se permanece en el bucle (true) o se sale (false).
- 3) **Acción final del bucle**: se ejecuta al final de cada iteración en la ejecución del bloque de código del bucle.

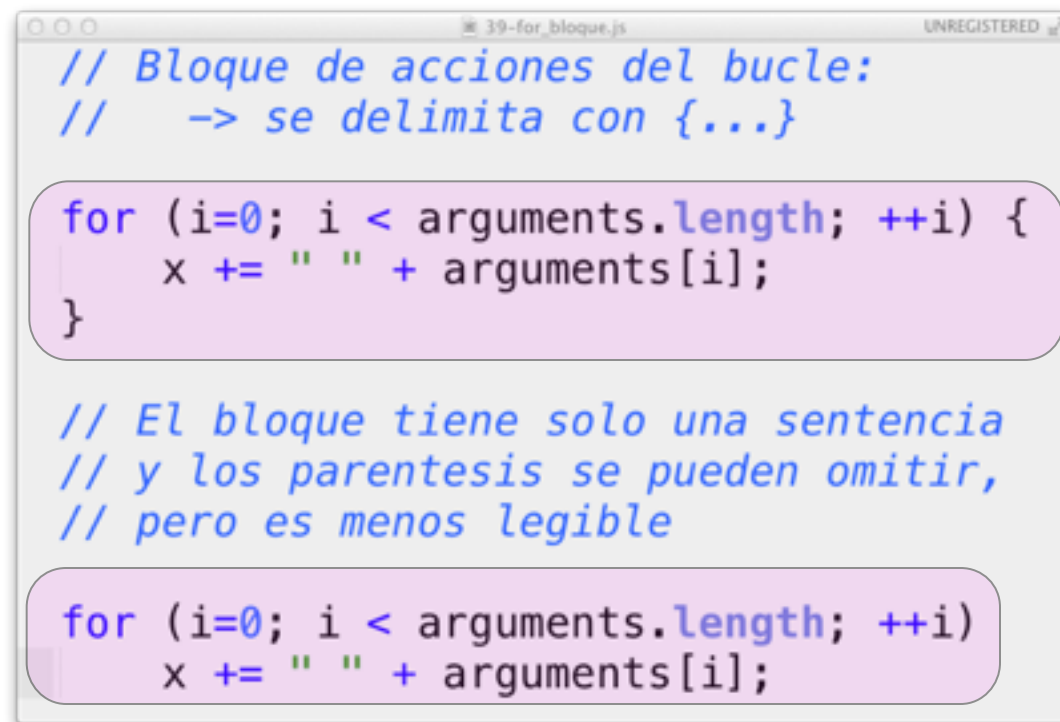
La sentencia if del bloque de código incrementa la variable "n" si la letra del string coincide con la letra de la frase que referencia i.

```
1 function veces (letra, frase) {  
2     // inicialización; condición; acciones final  
3     for ( var i=0, n=0; i < frase.length; ++i ) {  
4         if ( letra === frase[i] ) { ++n; };  
5     }  
6     return n;  
7 };  
8  
9 var l='a', f='la casa roja';  
10 console.log('La frase "' + f + '" tiene '  
11             + veces(l,f) + ' veces la letra ' + l);
```

```
mod01 — bash — 50x5  
venus-5:mod01 jq$  
venus-5:mod01 jq$  
venus-5:mod01 jq$ node 43-veces_for.js  
La frase "la casa roja" tiene 4 veces la letra a  
venus-5:mod01 jq$
```

Sintaxis de la sentencia for

- ◆ La sentencia comienza con **for**
- ◆ sigue la condición (con 3 partes)
 - **(i=0; i < arguments[i]; i++)**
 - ◆ **Inicialización:** i=0,
 - ◆ **Permanencia en bucle:** i < arguments.length
 - ◆ **Acción final bloque:** ++i, ...
- ◆ La sentencia termina con un bloque que debe delimitarse con {...}
- ◆ Bloques que contengan solo 1 sentencia
 - pueden omitir {...}, pero se mejora la legibilidad delimitandolos con {...}



```
// Bloque de acciones del bucle:
//    -> se delimita con {...}

for (i=0; i < arguments.length; ++i) {
    x += " " + arguments[i];
}

// El bloque tiene solo una sentencia
// y los parentesis se pueden omitir,
// pero es menos legible

for (i=0; i < arguments.length; ++i)
    x += " " + arguments[i];
```

función veces(..): bucle do-while

El ejemplo ilustra el bucle do-while con la **función veces**. La primera sentencia inicializa variables "i" y "n".

La condición de permanencia en el bucle (expresión entre paréntesis después de palabra while) se sitúa al final del bucle. Esto implica que el bucle siempre se ejecuta al menos una vez.

El bucle recorre el array con el índice "i" e incrementa la variable "n" si la letra del string coincide con la de referencia. La variable "i" se incrementa en la evaluación de la condición de la sentencia i.

Al finalizar se devuelve "n" (numero de coincidencias).

```
mod01 — bash — 50x5
venus-5:mod01 jq$
venus-5:mod01 jq$
venus-5:mod01 jq$ node 44-veces_do.js
La frase "la casa roja" tiene 4 veces la letra a
venus-5:mod01 jq$
```

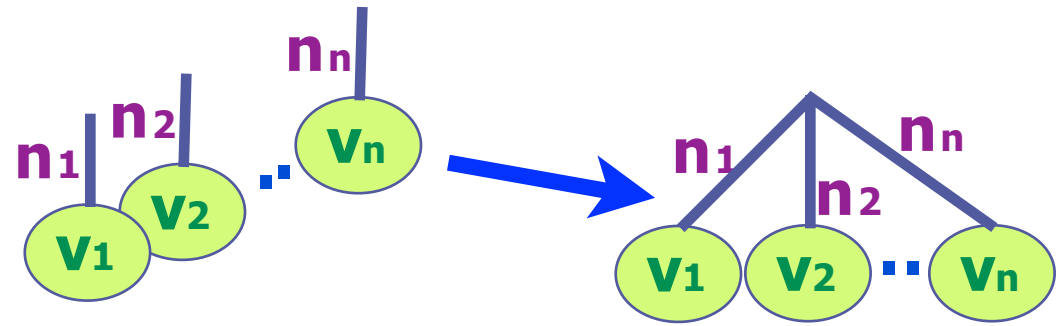
```
function veces (letra, frase) {
  var i = 0, n = 0; // inicialización del bucle
  do {
    if ( letra === frase[i++] ) { ++n; }; // acción del bucle e incremento de índice
  } while ( i < frase.length ) // condición de permanencia en el bucle
  return n;
};

var l='a', f='la casa roja';
console.log('La frase "' + f + '" tiene '
+ veces(l,f) + ' veces la letra ' + l);
```




Objetos, propiedades y métodos

Objetos



◆ Los objetos son colecciones de variables

- agrupadas como un elemento estructurado que llamamos **objeto**
 - ◆ Las variables de un objeto se denominan **propiedades**
 - ◆ Doc: https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Trabajando_con_objetos

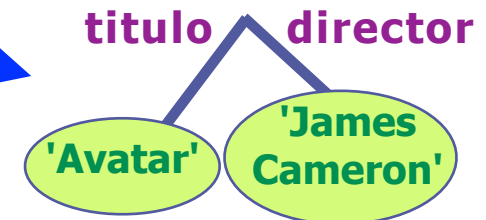
◆ Una **propiedad** es un par **nombre:valor** donde

- los **nombres** deben ser **todos diferentes** en un mismo objeto

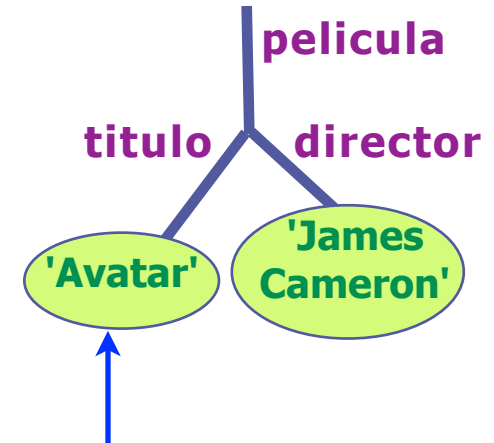
◆ Se definen con el literal: **{ nombre:valor, ... }**

- **Por ejemplo:** **{titulo: 'Avatar', director: 'James Cameron'}**

- ◆ crea un objeto con 2 propiedades:
 - **titulo:'Avatar'**
 - **director:'James Cameron'**

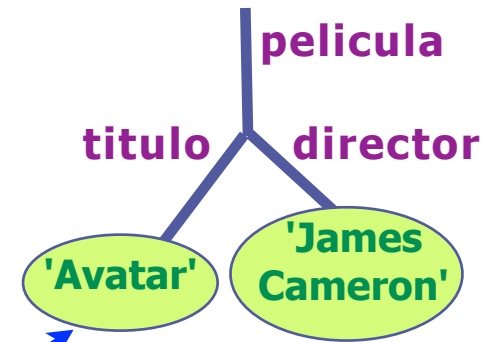


Propiedades



- ◆ El acceso a propiedades utiliza el operador punto
 - **obj.propiedad**
- ◆ Por ej. en: `var pelicula = {titulo: 'Avatar', director: 'James Cameron'}`
 - **pelicula.titulo** => "Avatar"
 - **pelicula.director** => "James Cameron"
 - **pelicula.fecha** => undefined // la propiedad fecha no existe
- ◆ Aplicar el operador punto sobre **undefined** o **null**
 - Provoca un **Error_de_ejecución** y aborta la ejecución del programa
- ◆ La notación punto solo acepta nombres de propiedades
 - Con la sintaxis de variables: **a, _method, \$1, ...**
 - ◆ No son utilizables: **"#43", "?a=1",**

Notación array



- ◆ La notación array es equivalente a la notación punto
 - **`pelicula["titulo"]`** es equivalente a **`pelicula.titulo`**
 - ◆ Al acceder a: **`var pelicula = {titulo: 'Avatar', director: 'James Cameron'}`**
- ◆ La notación array permite utilizar **strings arbitrarios** como nombres
 - por ejemplo, **`objeto["El director"]`**, **`pelicula[""]`** o **`a["%43"]`**
 - ◆ **OJO!** es conveniente utilizar siempre nombres compatibles con notación punto
- ◆ Nombres (strings) arbitrarios son posibles también en un literal de objeto:
 - Por ejemplo, **`{"titulo": 'Avatar', "El director": 'James Cameron'}`**

Nombres de propiedades como variables

- ◆ La notación array permite acceder también a propiedades
 - cuyo nombre esta en una variable en forma de string
 - ◆ Esto no es posible con la notación punto

```
var x = {titulo: 'Avatar', director: 'James Cameron'}
```

```
x.titulo;      => 'Avatar'
```

```
x['titulo'];   => 'Avatar'
```

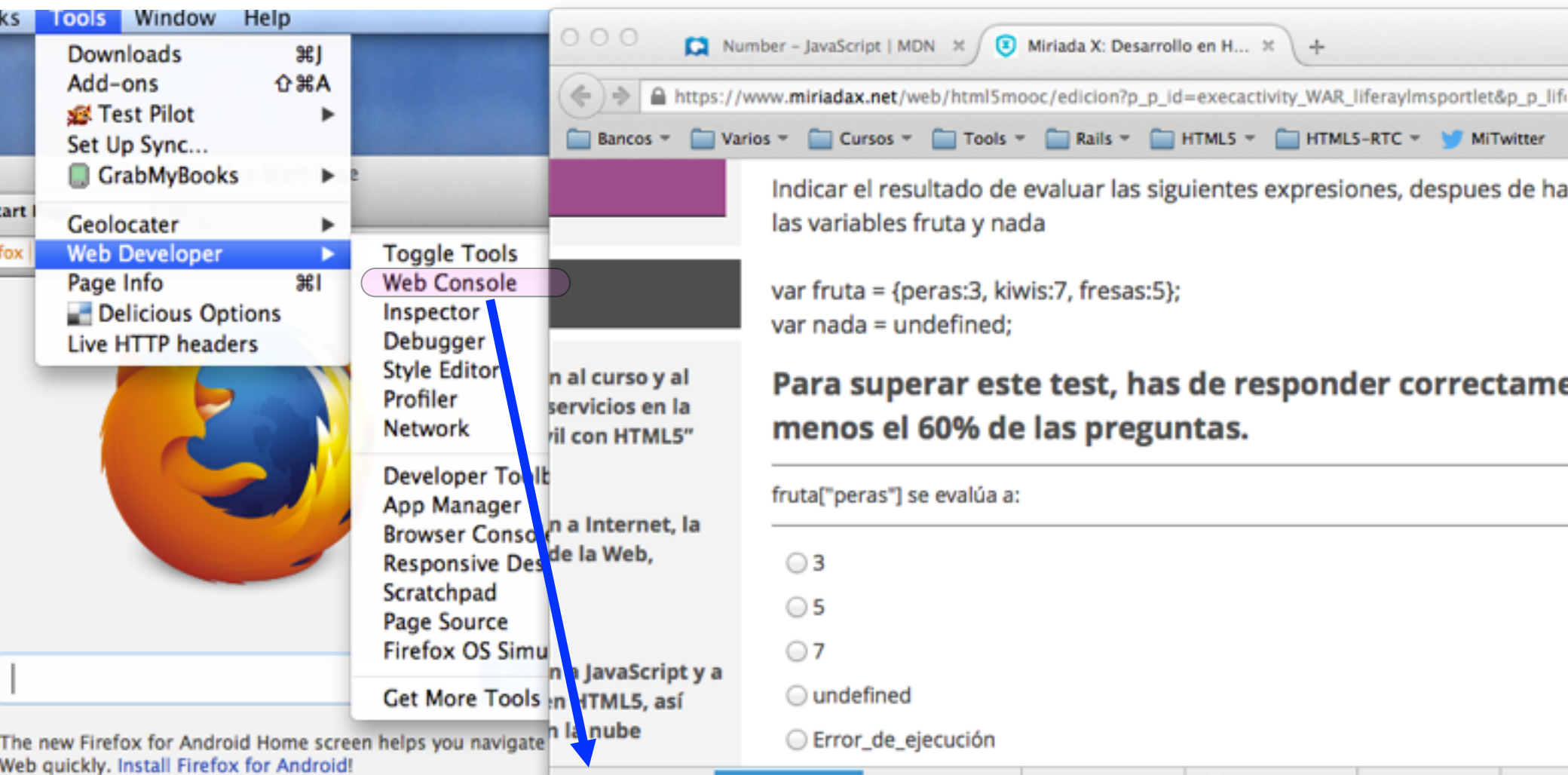
```
var p = 'titulo'; // inicializada con string 'titulo'
```

```
x[p];          => 'Avatar'
```

```
x.p;           => undefined
```

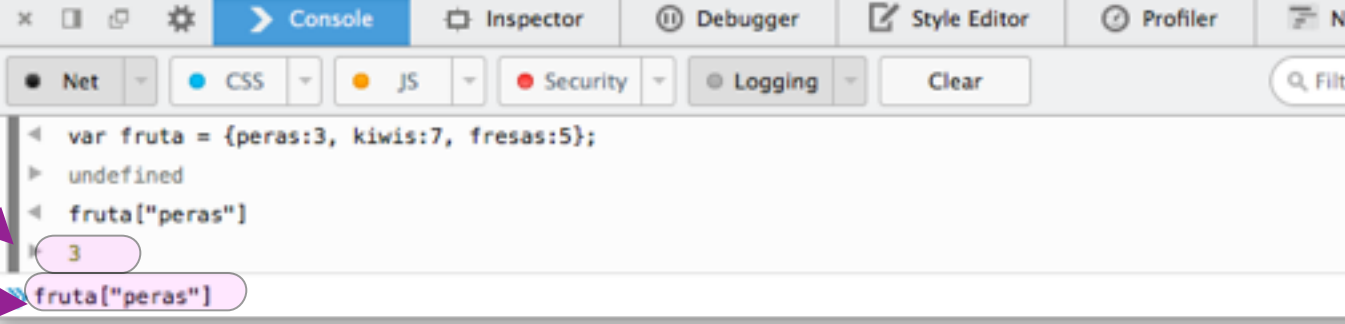
x tiene una propiedad de nombre **'titulo'**, que es el string que contiene **p**

El **objeto x** no tiene ninguna propiedad de nombre **p** y devuelve **undefined**



La consola nos va mostrando
el resultado de ejecutar las
sentencias JavaScript

Aquí se introduce la sentencia



Clases y herencia

- ◆ Todos los objetos de JavaScript pertenecen a la **clase Object**
 - Javascript posee mas clases predefinidas que derivan de Object
 - ◆ **Date, Number, String, Array, Function,**
 - ◆ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Predefined_Core_Objects
 - Un objeto hereda los métodos y propiedades de su clase
- ◆ Un **método** es una operación (~función) invocable sobre un objeto
 - Se invoca con la notación punto: **objeto.metodo(..params..)**
- ◆ Todas las clases tienen un constructor con el nombre de la clase
 - que permite crear objetos con el operador **new**
 - ◆ Por ejemplo, **new Object()** crea un objeto vacío equivalente a **{}**

Métodos de la clase

- ◆ Un objeto **hereda** métodos de su **clase**, por ejemplo
 - los objetos de la clase **Date** heredan métodos como
 - ◆ **toString(), getDay(), getFullYear(), getHours(), getMinutes(),** (ver ejemplo)
 - ◆ https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Date
- ◆ En un objeto solo se puede invocar métodos heredados o definidos
 - Invocar un método **no heredado ni definido en un objeto**
 - ◆ provoca un **error_de_ejecución**

```
var fecha = new Date();
```

```
fecha.toString()    => Fri Aug 08 2014 12:34:36 GMT+0200 (CEST)
fecha.getHours()    => 12
fecha.getMinutes()  => 34
fecha.getSeconds()  => 36
```


Definición de un nuevo método de un objeto

- ◆ Los métodos se pueden definir también directamente en un objeto
 - El nuevo método solo se define para ese objeto (no es de la clase)
- ◆ Invocar un método cambia el **entorno de ejecución** de JavaScript
 - pasando a ser el **objeto invocado**, que se referencia con **this**
 - ◆ **this.titulo** referencia la propiedad **titulo** del objeto **pelicula**

```
var pelicula = {  
  titulo:'Avatar',  
  director:'James Cameron',
```

```
  resumen:function () {  
    return "El director de " + this.titulo + " es " + this.director;  
  }  
}
```

```
pelicula.resumen()    =>    "El director de Avatar es James Cameron"
```

Algunas clases predefinidas (Core)

◆ Object

- Clase raíz, suele usarse el literal: `{a:3, b:"que tal"}`

◆ Array

- Colección indexable, suele usarse el literal: `[1, 2, 3]`

◆ Date

- Hora y fecha extraída del reloj del sistema: `new Date()`

◆ Function

- Encapsula código, suele usarse literal o def.: `function (x) {....}`

◆ RegExp

- Expresiones regulares, suele usarse el literal: `/(hola)+$/`

◆ Math

- Modulo con **constantes** y **funciones matemáticas**

◆ Number, String y Boolean

- Clases que encapsulan valores de los tipos **number**, **string** y **boolean** como objetos
 - Sus métodos se aplican a los tipos directamente, la conversión a objetos es automática

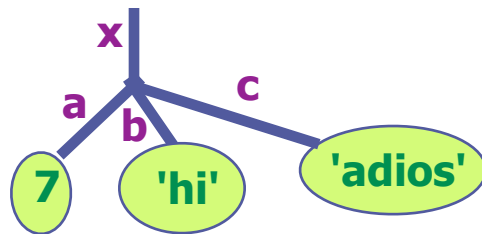
◆ Doc: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Predefined_Core_Objects



Sentencia for/in de JavaScript

Sentencia for/in

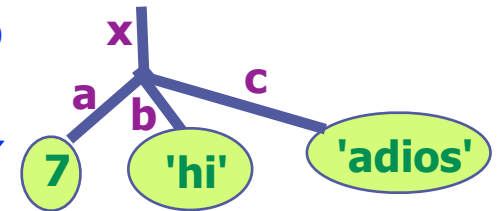
- ◆ **for (i in x) {..bloque de instrucciones..}**
 - itera en todas las propiedades del objeto **x**
- ◆ El **nombre** de propiedad y su **contenido** se referencian con **"i"** y **"x[i]"**
 - **"i"** contiene el nombre de la propiedad en cada iteración
 - **"x[i]"** representa el valor de la propiedad **"i"**
 - ◆ Dentro de la sentencia for debe utilizarse la notación array



Sentencia for/in

```
mod01 — bash — 41x5
venus-5:mod01 jq$ node 55-for_in.js
Propiedad a = 7
Propiedad b = hi
Propiedad c = adios
venus-5:mod01 jq$
```

- ◆ En el ejemplo se utiliza **for (i in x) {...}**
 - para mostrar el contenido de las propiedades de un objeto
 - ◆ utilizando la notación array: **x[i]**



```
55-for_in.js UNREGISTERED
1  var x = {a:7, b:'hi', c:'adios'};
2
3  for (var i in x) {
4      console.log("Propiedad " + i + " = " + x[i]);
5  }
```

Sintaxis de la sentencia for/in

- ◆ La sentencia comienza por **for**
- ◆ Sigue la condición (**i in obj**)
 - debe ir entre **paréntesis (...)**
- ◆ Los bloques de más de 1 sentencia
 - deben delimitarse con {...}
- ◆ Bloques de 1 sentencia
 - pueden omitir {...}, pero mejoran la legibilidad delimitados con {..}

```
14-for_in_bloque.js  UNREGISTERED

// Utilizar notacion array para
// acceder a propiedades: obj[i]

for (i in obj) {
    z = z + obj[i];
    obj[i] = "inspected";
}

// En bloques de solo 1 sentencia
// {...} es opcional
//     -> pero se recomienda usarlo

for (i in obj) {
    z = z + obj[i];
}

// Estas 2 formas son equivalentes
// pero menos legibles

for (i in obj)    z = z + obj[i];

for (i in obj)
    z = z + obj[i];
```



Objetos:

Propiedades dinámicas y anidadas

Objetos anidados: árboles

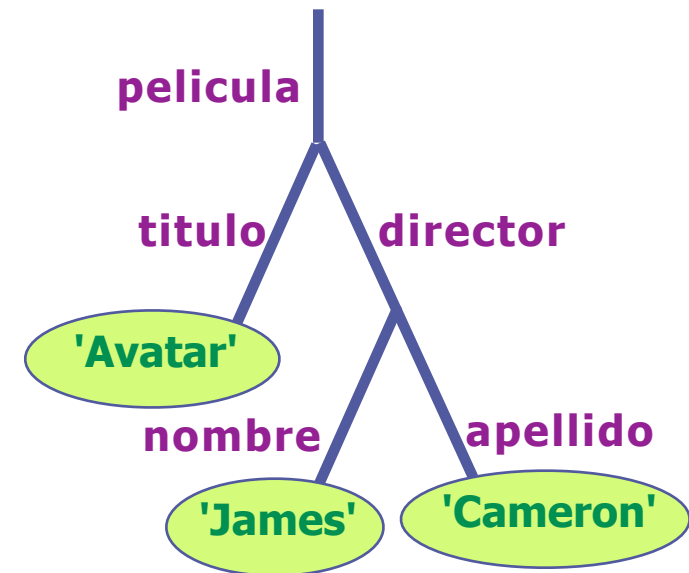
```
var pelicula = {  
  titulo: 'Avatar',  
  director: {  
    nombre: 'James',  
    apellido: 'Cameron'  
  }  
};
```

- ◆ Los objetos pueden **anidarse** entre si
 - Los objetos anidados representan **árboles**

- ◆ La notación punto o array puede **encadenarse**
 - Representando un **camino en el árbol**

- ◆ Las siguientes expresiones se evalúan así:

- `pelicula.director.nombre` \Rightarrow 'James'
- `pelicula['director']['nombre']` \Rightarrow 'James'
- `pelicula['director'].apellido` \Rightarrow 'Cameron'
- `pelicula.estreno` \Rightarrow undefined
- `pelicula.estreno.año` \Rightarrow Error_de_ejecución



Propiedades dinámicas

◆ Las propiedades de objetos

- Pueden **crearse**
- Pueden **destruirse**

◆ Operaciones sobre propiedades

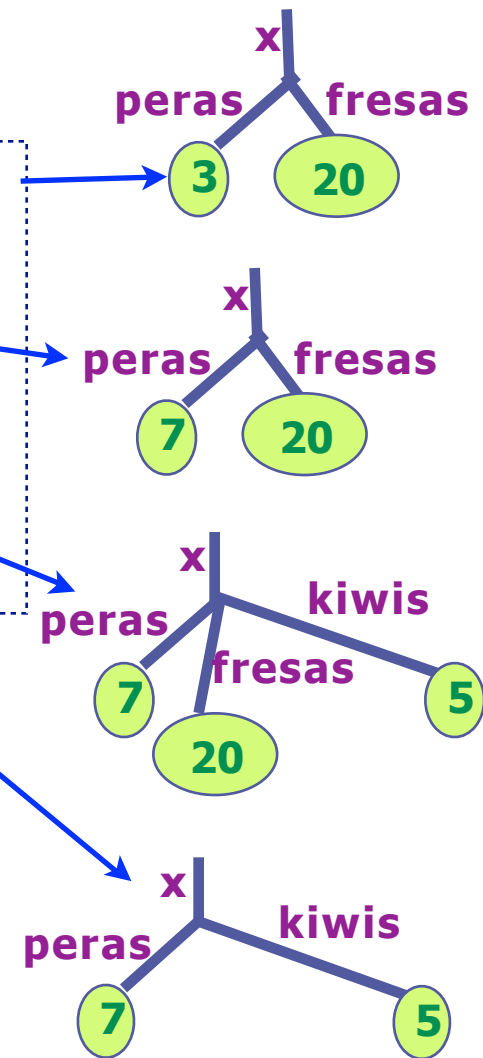
- **x.c = 4** ¡¡OJO: sentencia compleja!!
 - ◆ si propiedad **x.c** existe, le asigna **4**;
 - si **x.c** no existe, crea **x.c** y le asigna **4**
- **delete x.c**
 - ◆ si existe **y.c**, la elimina; si no existe, no hace nada
- **"c" in x**
 - ◆ si **x.c** existe, devuelve **true**, sino devuelve, **false**

```
var x = { peras:3, fresas:20};
```

```
x.peras = 7;
```

```
x.kiwis = 5;
```

```
delete x.fresas;
```



Usar propiedades dinámicas

- ◆ Las propiedades dinámicas de JavaScript
 - son muy útiles si se utilizan bien
- ◆ Un objeto solo debe definir las propiedades
 - que contengan información conocida
 - ◆ añadirá mas solo si son necesarias
- ◆ La información se puede consultar con
 - **prop1 && prop1.prop2**
 - ◆ para evitar errores de ejecución
 - ◆ si las propiedades no existen

// Dado un objeto **pel** definido con

```
var pel = {  
  titulo: 'Avatar',  
  director: 'James Cameron'  
};
```

// se puede añadir **pel.estreno** con

```
pel.estreno = {  
  año: '2009',  
  cine: 'Tivoli'  
}
```

// Una expresión muy util es

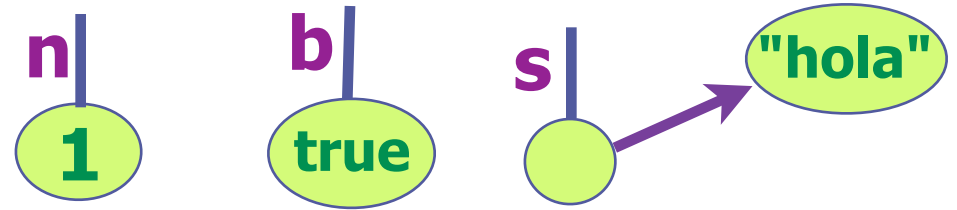
pel.estreno && pel.estreno.año

// devuelve **pel.estreno** o **undefined**,
// evitando **ErrorDeEjecución**, si
// **pel.estreno** no se hubiese creado



Referencias a objetos

Valores y referencias

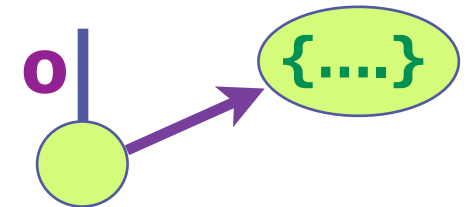


◆ Los tipos JavaScript se gestionan por valor o por referencia

- **number, boolean o undefined** se gestionan por valor
 - ◆ **string** se gestiona por referencia, pero es a todos los efectos un valor,
- **object** se gestiona por referencia

◆ La **asignación** copia el contenido de la variable

- Copia el valor o el puntero según sea el contenido



◆ La **identidad** y la **igualdad** también se ven afectadas

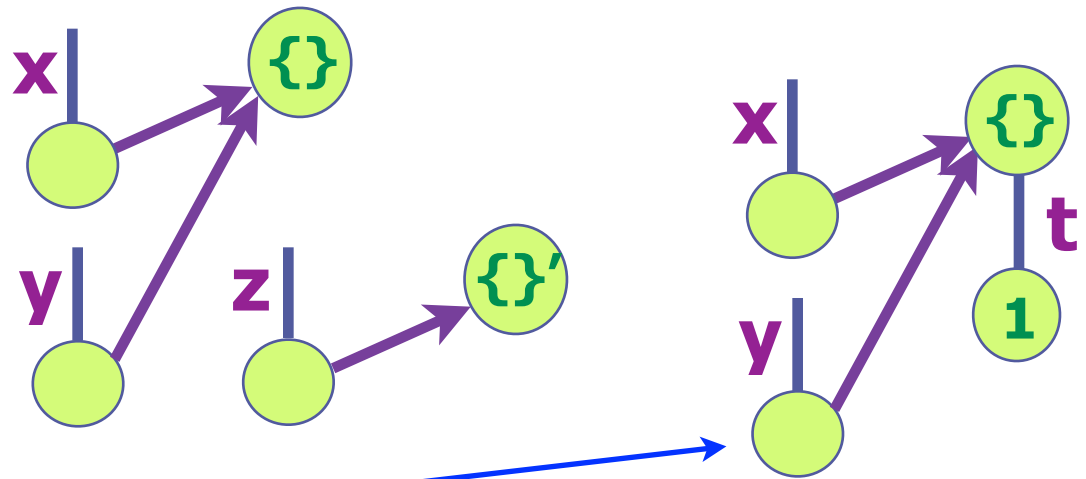
- Comparan el valor o el puntero según sea el contenido
 - ◆ Salvo con **strings** donde se comparan los valores (string apuntado) y no los punteros

```
var x = {}; // x e y tienen la  
var y = x; // misma referencia
```

```
var z = {}; // la referencia a z  
// es diferente de  
// la de x e y
```

```
y.t = 1;
```

```
x.t => 1 // x accede al mismo  
y.t => 1 // objeto que y  
z.t => undefined
```

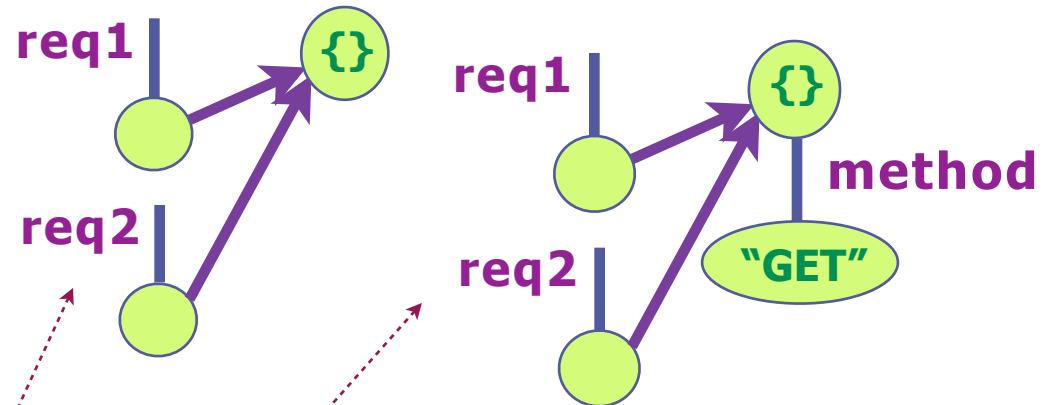


- ◆ Las variables que contienen objetos
 - solo contienen la referencia al objeto
- ◆ En objeto esta en otro lugar en memoria
 - indicado por la referencia
- ◆ Al asignar una variable se copia el puntero
 - si se modifica el objeto de una de ellas
 - ◆ Se modificarán los objetos de las variables que contengan el mismo puntero
- ◆ Los **parámetros de funciones** tienen el mismo efecto lateral, cuando son objetos

Efectos laterales de las referencias a objetos

Parámetros por referencia

```
71_func_reference.js  UNREGISTERED
1  var req = {};
2
3  function set(req1) {
4    req1.method = "GET";
5  }
6
7  function answer(req2) {
8    if (req2.method === "GET") {
9      return "Ha llegado: " + req2.method;
10   } else {
11     return "-> Error 37" ;
12   }
13 }
14
15 answer(req); // => "-> Error 37"
16
17 set(req);
18 answer(req); // => "Ha llegado: GET"
19
```



- ◆ Cuando pasamos objetos como parámetro
 - solo se pasa la referencia al objeto
- ◆ Si varias funciones modifican el mismo objeto
 - las modificaciones se verán en todas ellas
- ◆ Los objetos son un mecanismo muy eficaz
 - para comunicar funciones entre sí

Identidad e igualdad de objetos

◆ Las referencias a objetos afectan a la identidad

- porque **identidad de objetos**
 - ◆ es **identidad de referencias**
- La identidad en objetos significa
 - ◆ que se esta compartiendo el mismo objeto

◆ Para tener igualdad semántica se debe redefinir la **identidad**

- En los **strings** la identidad si es semántica

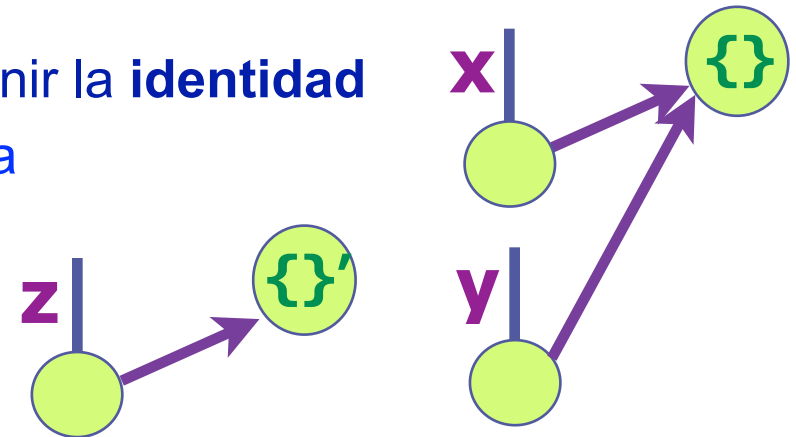
◆ Igualdad (debil) de objetos == y !=

- no tiene utilidad tampoco con objetos
 - ◆ no se debe utilizar

```
var x = {}; // x e y contienen la  
var y = x; // misma referencia
```

```
var z = {} // la referencia a z  
           // es diferente de x e y
```

```
x === y      => true  
x === {}    => false  
x === z      => false
```





La Clase Array

Arrays

- ◆ **Array:** lista ordenada de
 - elementos **heterogéneos**
 - ◆ accesibles a través de un índice
 - de **0** a **length-1**

- ◆ **Tamaño máximo:** $2^{32}-2$ elementos

- ◆ **Elementos**

- **a[0]** es el primer elemento
-
- **a[a.length-1]** último elemento

```
var x = [1, 2, 3];
```

```
a[0]      => 1
```

```
a[1]      => 2
```

```
a[2]      => 3
```

```
a.length  => 3
```

- ◆ https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array

Elementos de un Array

- ◆ Elementos del array pueden contener cualquier valor u objeto
 - **undefined**
 - otro **array**
 - **objetos**
 - ...
- ◆ Indexar elementos que no existen
 - devuelve **undefined**
 - ◆ por ejemplo con índices mayores que **length**

```
var a = [1, undefined, 'a', , [1, 2]];
```

```
a[3];           => undefined
```

```
a[4];           => [1, 2]
```

```
a[9];           => undefined
```

```
a[4][1];        => 2
```

Tamaño del Array

- ◆ Los arrays son dinámicos
 - pueden crecer y encoger
- ◆ Asignar un elemento fuera de rango
 - incrementa el tamaño del array
- ◆ El tamaño del array se puede modificar
 - con la propiedad **a.length**
 - ◆ **a.length = 3;**
 - modifica el tamaño del array
 - Que pasa a ser 4

```
var a = [1, 3, 1];  
  
a;           => [1, 3, 1]  
  
a[4] = 2;    => 2  
a;           => [1, 3, 1, , 2]  
  
// el array se reduce  
a.length = 2 => 2  
  
a            => [1, 3]
```

Métodos de Array

Array hereda métodos de su clase

- ◆ **sort()**: devuelve array ordenado
- ◆ **reverse()**: devuelve array invertido
- ◆ **push(e1, ..., en)**
 - añade **e1, ...,en** al final del array
- ◆ **pop()**
 - extrae último elemento del array

```
var a = [1, 5, 3];  
  
a.sort()      => [1, 3, 5]  
  
a.reverse()   => [5, 3, 1]  
  
a.push(false) => 4  
a             => [5, 3, 1, false]  
  
a.pop()       => false  
a             => [5, 3, 1]
```

Más métodos

◆ **join(<separador>):**

- devuelve string uniendo elementos
 - ◆ introduce <separador> entre elementos

◆ **slice(i,j):** devuelve una rodaja

- Índice negativo (j) es relativo al final
 - ◆ índice “-1” es igual a a.length-2

◆ **splice(i, j, e1, e2, .., en)**

- sustituye j elementos desde i en array
 - ◆ por e1, e2, ..,en
- Devuelve elementos eliminados

```
var a = [1, 5, 3, 7];
```

```
a.join(';')      => '1;5;3;7'
```

```
a.slice(1, -1)    => [5, 3]
```

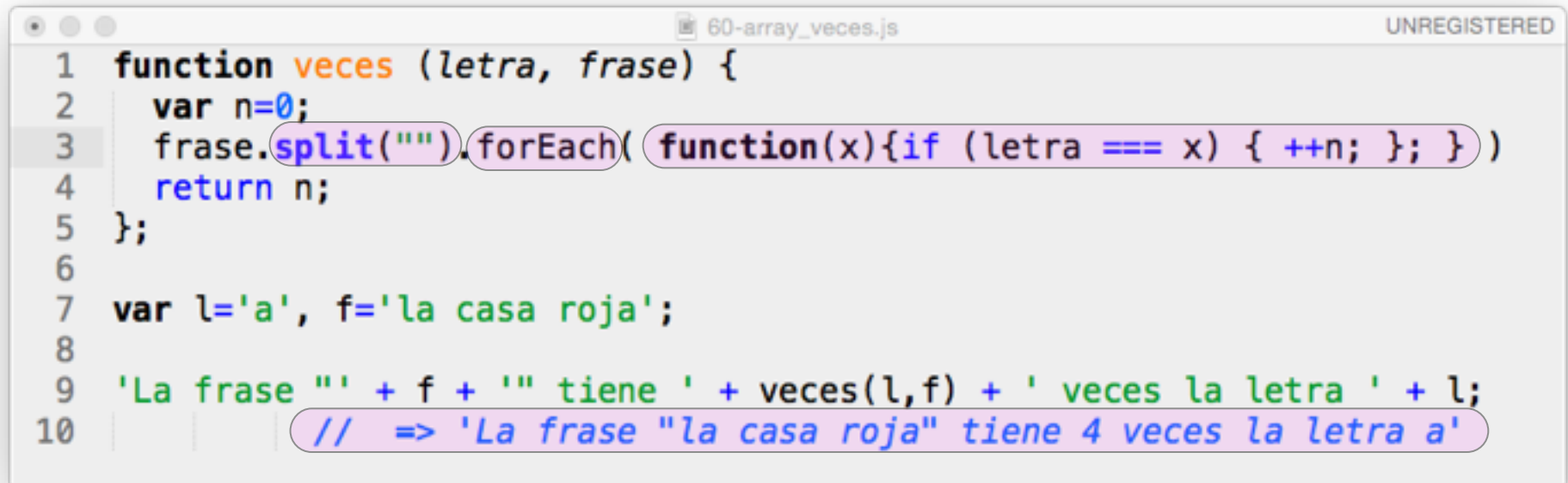
```
a.splice(1,2,true) => [5, 3]
```

```
a                => [1, true, 7]
```

Iteradores JavaScript 1.5

JavaScript 1.5 introduce nuevos métodos, de tipo iterador, de gran interés

- ◆ **forEach(function(elem, index, array){...}):** iterador en los elementos del array
 - ejecuta la función para cada elemento del array de forma secuencia
 - ◆ Equivale a un bucle, pero es mas compacto (ver ejemplo equivalente a función veces(...))
- ◆ **map(function(elem){...}):** mapea los elementos del array de acuerdo a la función
- ◆ **filter(function(elem){...}):** filtra los elementos del array de acuerdo a la función
- ◆ Y muchas otras: https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array



```
60-array_veces.js UNREGISTERED
1 function veces (letra, frase) {
2   var n=0;
3   frase.split("").forEach( function(x){if (letra === x) { ++n; }; } )
4   return n;
5 };
6
7 var l='a', f='la casa roja';
8
9 'La frase "' + f + '" tiene ' + veces(l,f) + ' veces la letra ' + l;
10 // => 'La frase "la casa roja" tiene 4 veces la letra a'
```



JSON: JavaScript Object Notation

JSON

- ◆ JSON: formato textual de representación de tipos y objetos JavaScript
 - <http://json.org/json-es.html>
- ◆ Un **objeto JavaScript** se transforma a un **string JSON** con
 - **JSON.stringify(object)**
- ◆ Un **string JSON** se transforma en el **objeto original** con
 - **JSON.parse(string_JSON)**

```
var x = {a:1, b:{y:[false, null, ""]}}, y, z;
```

```
y = JSON.stringify(x);      => '{"a":1, "b":{"y":[false, null, ""]}}'
```

```
z = JSON.parse(y);         => {a:1, b:{y:[false, null, ""]}}
```


Serialización de datos

◆ Serialización:

- transformación **reversible** de un tipo u objeto (en memoria) en un **string equivalente**

◆ La serialización es un formato de intercambio de datos

- **Almacenar datos** en un fichero
- **Enviar datos** a través de una línea de comunicación
- **Paso de parámetros** en interfaces REST

◆ En JavaScript se realiza desde ECMAScript 5 con

- **JSON.stringify(...)** y **JSON.parse(...)**

◆ Otros formatos de serialización: XML, HTML, XDR(C), ...

- Estos formatos están siendo desplazados por JSON, incluso XML
 - ◆ Hay bibliotecas de JSON para los lenguajes más importantes

Características de JSON

◆ JSON puede serializar

- objetos, arrays, strings, números finitos, true, false y null
 - ◆ NaN, Infinity y -Infinity se serializan a null
 - ◆ Objetos Date se serializan a formato ISO
 - la reconstrucción devuelve un string y no el objeto original
- No se puede serializar
 - ◆ Funciones, RegExp, errores, undefined

◆ Admite filtros para los elementos no soportados

- ver doc de APIs JavaScript

```
JSON.stringify(new Date())    => '"2013-08-08T17:13:10.751Z"'
```

```
JSON.stringify(NaN)          => 'null'
```

```
JSON.stringify(Infinity)     => 'null'
```



Prototipos y clases JavaScript

Prototipo

- ◆ La **herencia** en JavaScript se basa en **prototipos**
 - Todo **objeto** JavaScript **posee un prototipo**
 - ♦ del cual hereda sus propiedades y métodos
- ◆ El **prototipo** es un objeto como otro cualquiera
 - al que se pueden añadir o quitar propiedades y métodos
 - ♦ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance_Revisited
- ◆ Una modificación del prototipo trascenderá a todos los objetos asociados
 - si borramos un método,
 - ♦ este ya no se podrá invocar sobre los objetos enlazados
 - si añadimos un método
 - ♦ este se podrá invocar sobre todos los objetos asociados al prototipo
- ◆ Podemos obtener el prototipo de un objeto con el método de la clase Object
 - **Object.getPrototypeOf(obj)**

Tipado JavaScript

◆ JavaScript implementa Tipado de Patos

- “Si anda y grazna como un pato, es un pato”

◆ Las **clases son funciones** que actúan como constructores

- El nombre de un constructor empieza por mayúscula (convención)
 - ◆ Los objetos de una clase se crean invocando el constructor con el **operador new**
 - **new Object(), new Date(), new Array(),**
 - ◆ Los literales de los tipos predefinidos crean objetos igual que los constructores
- Al crear un objeto se le asigna el prototipo del constructor

◆ El prototipo de una clase esta accesible en la propiedad prototype

- **Object.prototype, Array.prototype, ...**

Prototipos, clases y herencia

◆ Una **clase** en JavaScript es

- El conjunto de objetos que tienen el mismo prototipo (el de la clase)
 - ◆ Los objetos de la clase heredan métodos y propiedades del prototipo de la clase

◆ Si una clase deriva de otra sus prototipos están enlazados

- Se hereda de toda la cadena de prototipos
 - ◆ hasta llegar a la raíz del árbol

◆ La clase **Object** es la clase raíz del árbol de herencia

- Su prototipo es el único que no tiene prototipo
 - ◆ Las clases predefinidas **Array**, **RegExp**, **Date**, ... derivan directamente de **Object**

◆ Las clases predefinidas tienen además literales (**{..}**, **[..]**, **/../**,)

- permiten construir objetos de forma mucho mas legible y eficaz
 - ◆ Los literales deben utilizarse siempre que sea posible (en vez de los constructores)

Añadir método integer a clase Number

- ◆ La clase **Number** encapsula números del tipo primitivo number
 - permite añadir nuevos métodos, como el **método integer()** que vamos a añadir
 - ◆ **integer()** extrae la parte entera de un número
- ◆ La parte entera se calcula con **Math.floor(n)** si **n** es positivo y con **Math.ceil(n)** si **n** es negativo
 - **this** referencia en un método el objeto sobre el que se está invocando dicho método
 - ◆ En el ejemplo referencia el número sobre el que se invoca integer

```
01_integer.js UNREGISTERED
1 // La propiedad integer no debe existir en Number.prototype!
2 Number.prototype.integer; // => undefined
3
4 // Añadimos método "integer()" a Number
5
6 Number.prototype.integer = function () {
7     return Math[this < 0 ? "ceil" : "floor"](this);
8 }
9
10 7.3.integer(); // => 7
11 -7.3.integer(); // => -7
```

Propiedades heredadas y propias

◆ Los **objetos** de una **clase**

- Heredan las propiedades y métodos comunes del prototipo
 - ◆ Los métodos heredados pueden invocarse sobre cualquier objeto de la clase

◆ Las clases se pueden modificar a través del prototipo

- Añadir, cambiar o quitar métodos y propiedades
 - ◆ las modificaciones afectan a todos los objetos de la clase inmediatamente

◆ Un objeto tiene además propiedades y métodos propios (own)

- creados directamente en el propio objeto
 - ◆ **obj.hasOwnProperty("prop")** determina si una propiedad es propia o heredada

Objeto y prototipo

- ◆ **Object.prototype**: es el prototipo de object
- ◆ **Object.prototype.x**
 - es la propiedad **x** del prototipo de **Object**
- ◆ **obj.x** es la propiedad **x** de **obj**
 - Si **obj** no tiene una propiedad **x** propia
 - ◆ Buscará en su prototipo
 - El prototipo **obj** es el de la clase **Object**
- ◆ **delete obj.x**
 - borra la propiedad **x** de **obj**
- ◆ **delete Object.prototype.x**
 - borra la propiedad **x** del prototipo de **Object**
- ◆ **delete** borra solo las propiedades propias
 - no borra nada del prototipo

The screenshot shows a Node.js REPL window with the following code and output:

```
> Object.prototype
{}
> Object.prototype.x = 7
7
> Object.prototype
{ x: 7 }
> var obj = {}
undefined
> obj.x
7
> obj.x = 'hola'
'hola'
> obj.x
'hola'
> Object.prototype.x
7
> delete obj.x
true
> obj.x
7
> delete Object.prototype.x
true
> obj.x
undefined
>
```

To the right of the code, a diagram illustrates the prototype chain:

- A yellow oval labeled **obj** has a dashed blue arrow pointing up to a yellow oval containing **{ }**.
- A label **prototipo** is placed next to the arrow.
- Text above the top oval reads **Prototipo de Object**.

Ejemplo: Clase Contador

```
1 function Contador(inicial) {  
2     this.cont = inicial;  
3 }  
4  
5 Contador.prototype = {  
6     contador: function(){ return this.cont;},  
7     incr:     function(){ return ++this.cont;}  
8 }  
9  
10 var cont_1 = new Contador(0);  
11 var cont_2 = new Contador(7);  
12  
13 cont_1.contador() // => 0  
14 cont_1.incr()    // => 1  
15  
16 cont_2.contador() // => 7  
17 cont_2.incr()    // => 8
```

- ◆ El constructor de la clase **Contador** es una función como otra cualquiera
 - Los nombres de los constructores **empiezan con mayúscula** (convención)
 - **new Contador(5)** crea un objeto de la clase Contador inicializando cont con 5
- ◆ **this** referencia en un constructor y en un método el **objeto asociado**
 - **this** se puede omitir: **this.cont** y **cont** referencing la misma propiedad
- ◆ Los métodos asignados al prototipo serán heredados por los objetos de la nueva clase
 - **contador: function () { return cont; }** // devuelve el valor de la variable cont
 - **incr: function () { return ++cont; }** // incrementa cont y devuelve su valor

Función instanceof

- ◆ La sentencia **instanceof** determina
 - si un objeto o valor pertenece a una clase
- ◆ Los objetos de una clase derivada pertenecen también a la clase padre
 - Un array o una función pertenecen a la clase Object

```
{ } instanceof Object    => true    // { } es un objeto aunque este vacío
{ } instanceof Array     => false    // { } no es un Array, pertenece solo a Object

[] instanceof Array      => true     // [] es un array aunque este vacío
[] instanceof Object     => true     // pertenece a la clase Object,
                                   // porque Array deriva de Object

function(){} instanceof Function => true // function(){} es una función vacía
function(){} instanceof Object  => true // pertenece a la clase Object,
                                   // porque Function deriva de Object


"" instanceof String      => false   // "" es un string y los tipos no son objetos
new String("") instanceof String => true // new String("") si pertenece a clase String
```

Espacios de nombres y cierres (closures)

Ordenar el espacio de nombres

- ◆ En un programa JavaScript un nombre identifica un elemento y puede representar
 - una **variable**, una **función**, una **propiedad**, una **clase** (pseudo), ...
 - ◆ En un programa debemos minimizar y estructurar los nombres utilizados
- ◆ **Espacio de nombres**: conjunto de nombres visibles en un lugar del programa
 - **Espacio de nombres global**: nombres visibles en el ámbito global
 - **Espacio de nombres locales**: nombres visibles en algún ámbito local
- ◆ JavaScript no posee mecanismos para aislar los espacios de nombres entre sí, pero
 - Las propiedades de objetos permiten ordenar y estructurar espacios de nombres:
 - ◆ **blog.titulo** y **blog.texto** representan el título y el texto de un blog
 - ◆ **libro.titulo** y **libro.texto** representan el título y el texto de un libro
 - ◆ **pelicula.titulo** y **pelicula.director** representan título y director de una película
 - ◆

Cierre o closure



```
70_closure.js  UNREGISTERED
function funcion_exterior( ...) {
  var var_local ...; // variables internas
  .....
  // funciones internas
  function funcion_local(..) {...};
  .....
  // interfaz exterior
  return parametro_de_retorno;
}
```

- ◆ **Cierre o closure:** función que encapsula un conjunto de definiciones locales
 - que solo son accesibles a través del objeto interfaz retornado por dicha función
 - ◆ Las variables y funciones locales no se pueden acceder ni ver desde el exterior de una función
 - **OJO!** Un cierre no instancia sus variables hasta que no se ejecuta (invocar función)
- ◆ La **interfaz** de un cierre con el exterior es el parámetro de retorno de la función
 - Suele ser un objeto JavaScript que da acceso a las variables y funciones locales
- ◆ Las **variables locales de un cierre siguen existiendo** si existen referencias a ellas
 - aunque la función que las engloba haya finalizado su ejecución
 - ◆ Un cierre permite crear un objeto independiente cada vez que se invoca (factoría de objetos)
 - ◆ Cada objeto tiene sus propias variables y funciones internas, y su propio interfaz de acceso

Ejemplo: contador

```
81_contador_closure1.js  UNREGISTERED
1  function contador(inic) {
2      var _cont = inic; // variable interna
3
4      function contador() { return _cont;};
5      function incr()      { return ++_cont;};
6
7      return {contador: contador,
8              incr:      incr
9              };
10 }
11
12 var cont_1 = contador(0);
13 var cont_2 = contador(7);
14
15 cont_1.contador() // => 0
16 cont_1.incr()    // => 1
17
18 cont_2.contador() // => 7
19 cont_2.incr()    // => 8
```

- ◆ En este ejemplo, la función exterior (**contador**) permite crear objetos “contador”
 - El parámetro de esta función inicializa el contador (**variable _cont**)
 - ♦ `_cont` utiliza el convenio de comenzar por “_” las definiciones no visibles fuera (muy habitual)
- ◆ La interfaz del cierre son las 2 funciones incluidas en el parámetro de retorno
 - Estas funciones tienen acceso a la **variable cont** al invocarlas en `cont_1` o `cont_2`
 - ♦ **contador:** `function () { return cont; }` // devuelve el valor de la variable `cont`
 - ♦ **incr:** `function () { return ++cont; }` // incrementa `cont` y devuelve su valor
 - ♦ Se podrían añadir mas funciones al interfaz. p. e. decrementar, incrementar n unidades, etc.

Ejemplo contador II

```
1 function contador(inic) {  
2   var _cont = inic; // variable interna  
3  
4   return {contador: function(){ return _cont;},  
5           incr:     function(){ return ++_cont;}  
6 }  
7  
8  
9 var cont_1 = contador(0);  
10 var cont_2 = contador(7);  
11  
12 cont_1.contador() // => 0  
13 cont_1.incr()    // => 1  
14  
15 cont_2.contador() // => 7  
16 cont_2.incr()    // => 8
```

- ◆ El ejemplo es equivalente al anterior, pero con un patrón diferente
 - El código es mas conciso, aunque menos legible
 - ◆ A veces las closures se construyen utilizando este patrón
- ◆ Los métodos el interfaz están definidos directamente en el objeto de retorno

Objetos como diccionarios: cierres, clases y algo de metodología

Diccionarios clave-valor

- ◆ Los objetos de JavaScript son **diccionarios clave-valor**
 - El <nombre> de una propiedad es la clave, el valor está guardado en la propiedad
 - ♦ La clave debe ser única y puede ser un **string** o **número** de acuerdo a la sintaxis de JavaScript
 - Los diccionarios clave-valor se utilizan a menudo en programas
 - ♦ Equivalen a lo que en programación o informática se ha denominado tablas de hash
- ◆ A continuación se ilustra un diccionario JavaScript con una agenda de teléfonos
 - los **nombres de las propiedades** serán los **nombres de las personas**
- ◆ Como los nombres de personas no se ajustan a la sintaxis de la notación punto
 - utilizamos la notación array, p.e. **tf["Javier García"]**
- ◆ El literal de objetos admite ambas sintaxis, incluso mezcladas, por ejemplo
 - **{ "Javier García": 913278561, "José Jimenez": 957845123, Pepe: 913333333 }**

La agenda telefónica

- ◆ Los ejemplos siguientes de agenda telefónica ilustran el uso de cierres y clases
 - Así como sus similitudes y sus diferencias
- ◆ Desde el punto de vista sintáctico definir una clase o un cierre es muy parecido
 - El constructor de la clase cumple el mismo cometido que la función del cierre
 - ◆ La función del cierre es una factoría de objetos, devuelve un objeto cada vez que se invoca
 - El objeto que se asigna al prototipo de la clase es igual al interfaz del cierre
- ◆ La agenda es un objeto con dos propiedades, **_titulo** y **_contenido**
 - **_contenido** representa las personas de la agenda en los nombres de propiedades
 - ◆ El acceso a nombres con strings arbitrarios usará la notación array, por ej.
 - ◆ **meter: function(nombre, tf) {_contenido[nombre]=tf;}**
- ◆ La agenda ilustra también como utilizar JSON como formato de intercambio de objetos

```
function agenda (titulo, inic) {
  var _titulo = titulo;
  var _contenido = inic;

  return {
    titulo: function() { return _titulo; },
    meter: function(nombre, tf) { _contenido[nombre]=tf; },
    tf: function(nombre) { return _contenido[nombre]; },
    borrar: function(nombre) { delete _contenido[nombre]; },
    toJSON: function() { return JSON.stringify(_contenido); }
  }
}
```

```
var amigos = agenda ("Amigos",
  { Pepe: 913278561,
    José: 957845123
  });
amigos.meter("Jesús", 978512355);
```

```
var trabajo = agenda ("Trabajo",
  { "Javier García": 913278561,
    "José Jimenez": 957845123
  });
```

```
console.log('Agenda: ' + amigos.titulo());
console.log('Teléfono de Jesús: ' + amigos.tf("Jesús"));
console.log('Teléfono de José: ' + amigos.tf("José"));
amigos.borrar("José");
console.log('Tf de José borrado: ' + amigos.tf("José"));
console.log();
```

```
console.log('Agenda: ' + trabajo.titulo());
console.log('Tf de Javier García: ' + trabajo.tf("Javier García"));
console.log('Trabajo: ' + trabajo.toJSON());
```

Agenda como cierre

```
mod01 — bash — 64x10
venus-5:mod01 jq$ node 70-agenda_closure.js
```

```
Agenda: Amigos
Teléfono de Jesús: 978512355
Teléfono de José: 957845123
Tf de José borrado: undefined
```

```
Agenda: Trabajo
Tf de Javier García: 913278561
Trabajo: {"Javier García":913278561,"José Jimenez":957845123}
venus-5:mod01 jq$
```

Agenda como clase

```
function Agenda (titulo, inic) {  
  this.tituloAgenda = titulo;  
  this.contenido = inic;  
};
```

```
Agenda.prototype = {  
  titulo: function() { return this.tituloAgenda; },  
  meter: function(nombre, tf) { this.contenido[nombre]=tf; },  
  tf: function(nombre) { return this.contenido[nombre]; },  
  borrar: function(nombre) { delete this.contenido[nombre]; },  
  toJSON: function() { return JSON.stringify(this.contenido); }  
}
```

```
var amigos = new Agenda ("Amigos",  
  { Pepe: 913278561,  
    José: 957845123  
  });  
amigos.meter("Jesús", 978512355);
```

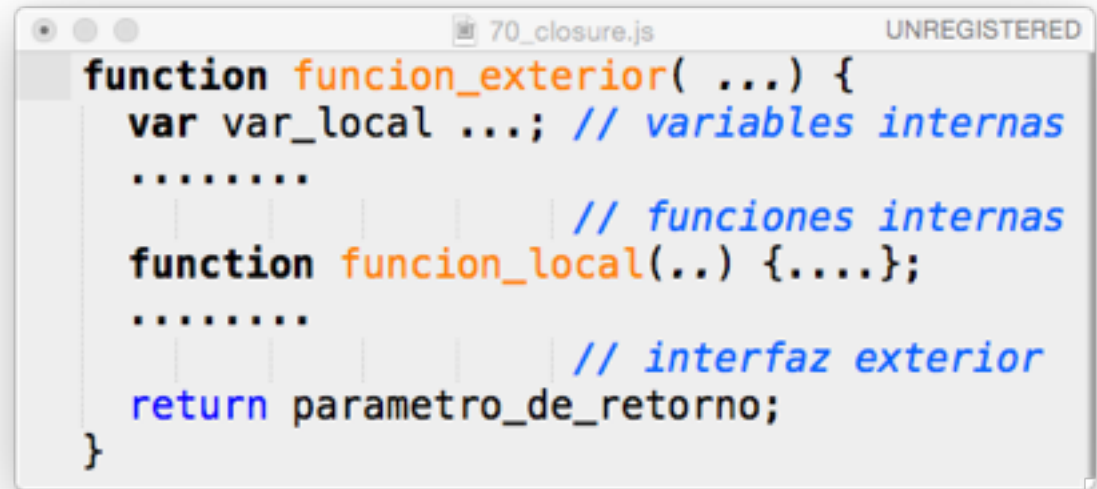
```
var trabajo = new Agenda ("Trabajo",  
  { "Javier García": 913278561,  
    "José Jimenez": 957845123  
  });
```

```
console.log('Agenda: ' + amigos.titulo());  
console.log('Teléfono de Jesús: ' + amigos.tf("Jesús"));  
console.log('Teléfono de José: ' + amigos.tf("José"));  
amigos.borrar("José");  
console.log('Tf de José borrado: ' + amigos.tf("José"));  
console.log();
```

```
console.log('Agenda: ' + trabajo.titulo());  
console.log('Tf de Javier García: ' + trabajo.tf("Javier García"));  
console.log('Trabajo: ' + trabajo.toJSON());
```

```
mod01 -- bash -- 64x10  
venus-5:mod01 jq$ node 75-agenda_class.js  
Agenda: Amigos  
Teléfono de Jesús: 978512355  
Teléfono de José: 957845123  
Tf de José borrado: undefined  
  
Agenda: Trabajo  
Tf de Javier García: 913278561  
Trabajo: {"Javier García":913278561,"José Jimenez":957845123}  
venus-5:mod01 jq$
```

Cierres o clases



```
70_closure.js  UNREGISTERED
function funcion_exterior( ...) {
  var var_local ...; // variables internas
  .....
  // funciones internas
  function funcion_local(...) {.....};
  .....
  // interfaz exterior
  return parametro_de_retorno;
}
```

- ◆ Un **cierre (closure)** se utiliza para aislar un bloque de código del resto del programa
 - El espacio de nombres local (var. y func.) queda totalmente aislado del exterior
 - ◆ Haciendo accesible dicho código a través del objeto interfaz retornado por la función del cierre
- ◆ En cambio, las **propiedades de un objeto son accesibles** en el exterior del objeto
 - No crean un espacio de nombres local aislado del exterior del objeto
 - ◆ También pueden ser modificadas por otros objetos o instrucciones exteriores al objeto
- ◆ Se recomienda por estas razones **utilizar cierres en vez de clases**
 - Para crear objetos que encapsulen variables y funciones locales en el interior

Estructuración de un programa

- ◆ JavaScript es un lenguaje orientado a objetos, funciones y prototipos
 - Usando bien estos elementos podemos paliar sus principales deficiencias:
 - ◆ Programas monolíticos basados en el uso de variables globales
 - ◆ Falta de módulos que separen el código en ficheros con espacios de nombres locales
 - ◆ “Tipado de patos” en la construcción de nuevas clases
- ◆ Para que los programas JavaScript sean claros y bien estructurado conviene:
 - Utilizar propiedades de objetos para tener variables mas claras y legibles
 - ◆ Por ejemplo, **blog.titulo**, **blog.descripcion**, ..., **libro.titulo**, **libro.indice**,
 - No utilizar variables globales, salvo para incluir módulos importados
 - ◆ Utilizando objetos para estructurar los nombres del interfaz de acceso a dichos módulos
 - Usar preferentemente cierres frente a clases para crear módulos
 - ◆ Aislando las definiciones locales de variables y funciones del resto del programa
 - ◆ Los cierres encapsulan la implementación mejor que una clase
 - Enriquecer clases existentes a través de sus prototipos
 - ◆ Cuando se necesiten propiedades o métodos comunes a clases o grupos de objetos



Final del tema