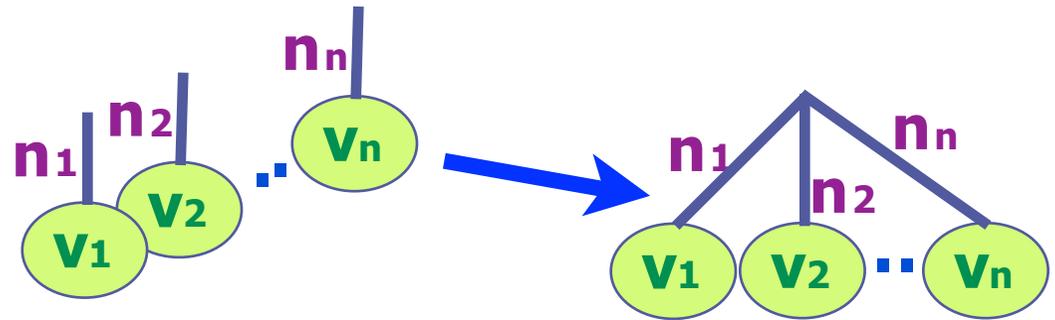




# Objetos, propiedades y métodos

# Objetos



- ◆ Los objetos son colecciones de variables
  - agrupadas como un elemento estructurado que llamamos objeto
    - ◆ Las variables de un objeto se denominan **propiedades**

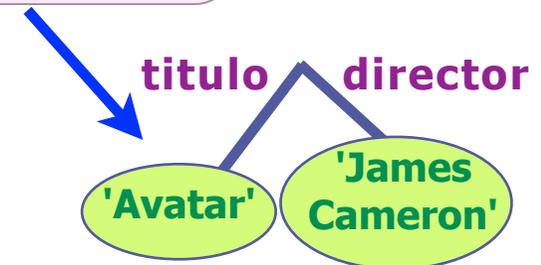
- ◆ Una **propiedad** es un par **nombre:valor** donde
  - los **nombres** deben ser **todos diferentes** en un mismo objeto

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

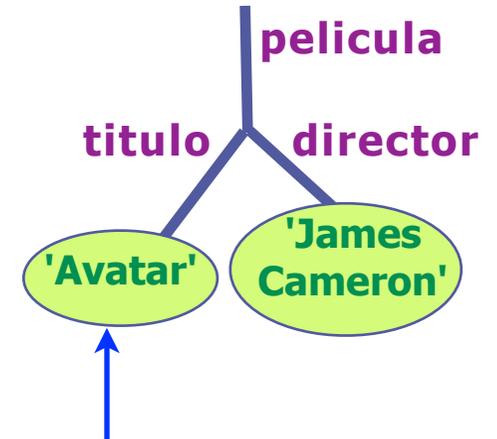
- **Por ejemplo:** { titulo: 'Avatar', director: 'James Cameron' }

- ◆ crea un objeto con 2 propiedades:

- titulo: 'Avatar'
- director: 'James Cameron'

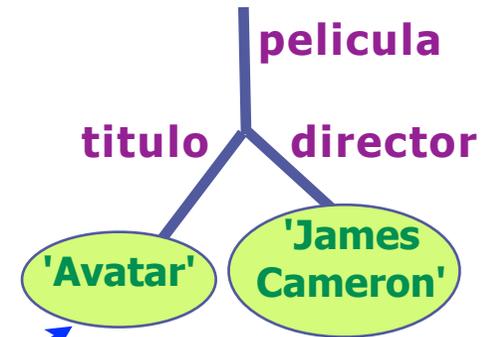


# Propiedades



- ◆ 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'
```

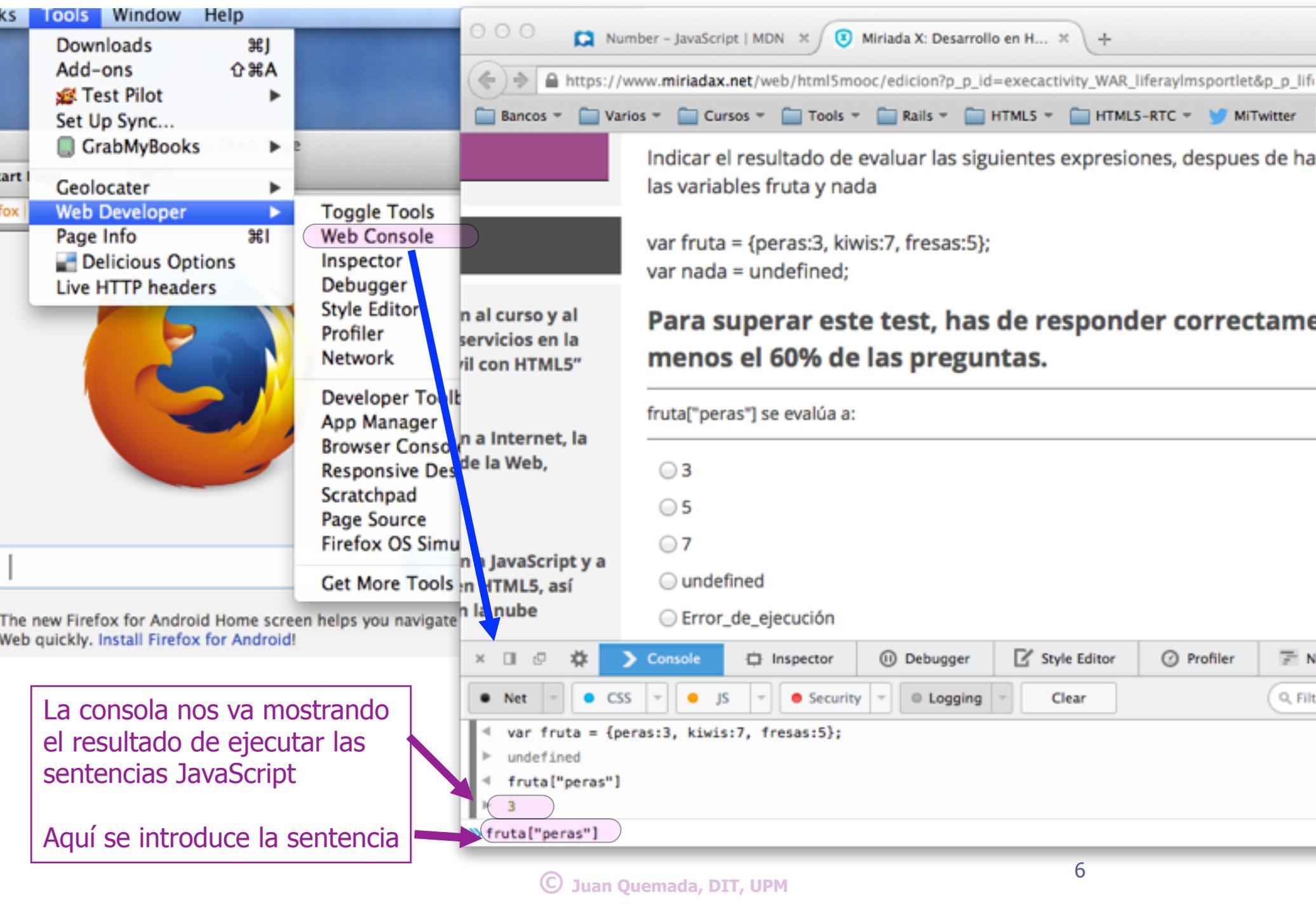
```
var p = 'titulo'; // inicializada con string 'titulo'
```

```
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](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](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Date)
- ◆ Solo se puede invocar métodos heredados o definidos en un objeto
  - 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

## ◆ 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



# Objetos: características avanzadas

# Propiedades dinámicas

## ◆ Las propiedades de objetos

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

## ◆ Operaciones sobre propiedades

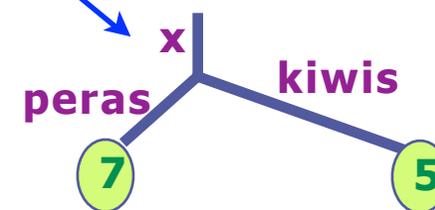
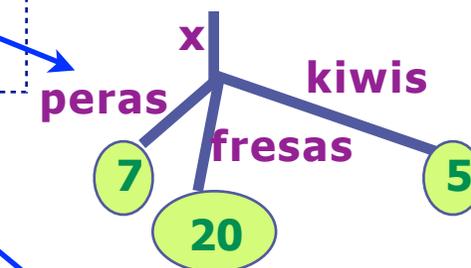
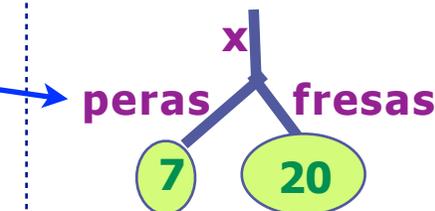
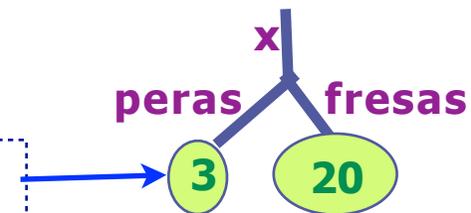
- **x.c = 4** ¡¡OJO: sentencia compleja!!
  - ◆ si propiedad **x.c** existe, le asigna **4**;
  - si **x.c** no existe, crea **x.c** y le asigna **4**
- **delete x.c**
  - ◆ si existe **y.c**, la elimina; si no existe, no hace nada
- **"c" in x**
  - ◆ si **x.c** existe, devuelve **true**, sino devuelve,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

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

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

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

```
delete x.fresas;
```



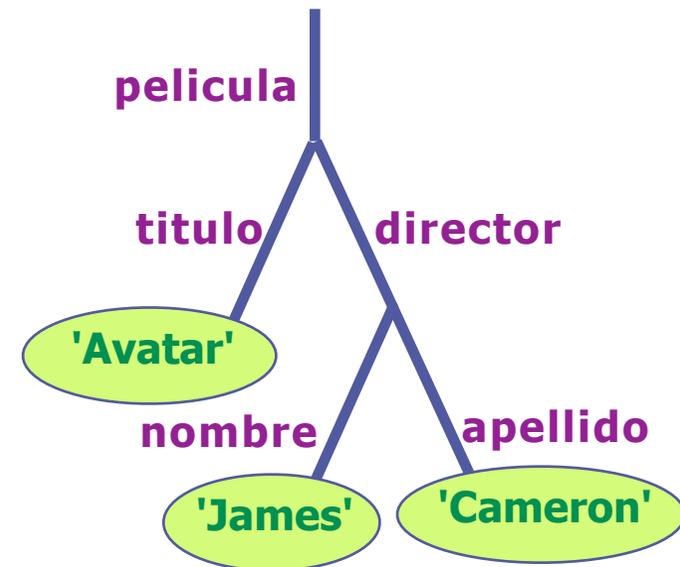
# Objetos anidados: árboles

```
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` => 'James'
- `pelicula['director']['nombre']` => 'James'
- `pelicula['director'].apellido` => 'Cameron'
- `pelicula.estreno` => undefined
- `pelicula.estreno.año` => Error\_de\_ejecución



# 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'  
}
```

```
// La expresión
```

```
pel.estreno && pel.estreno.año
```

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

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

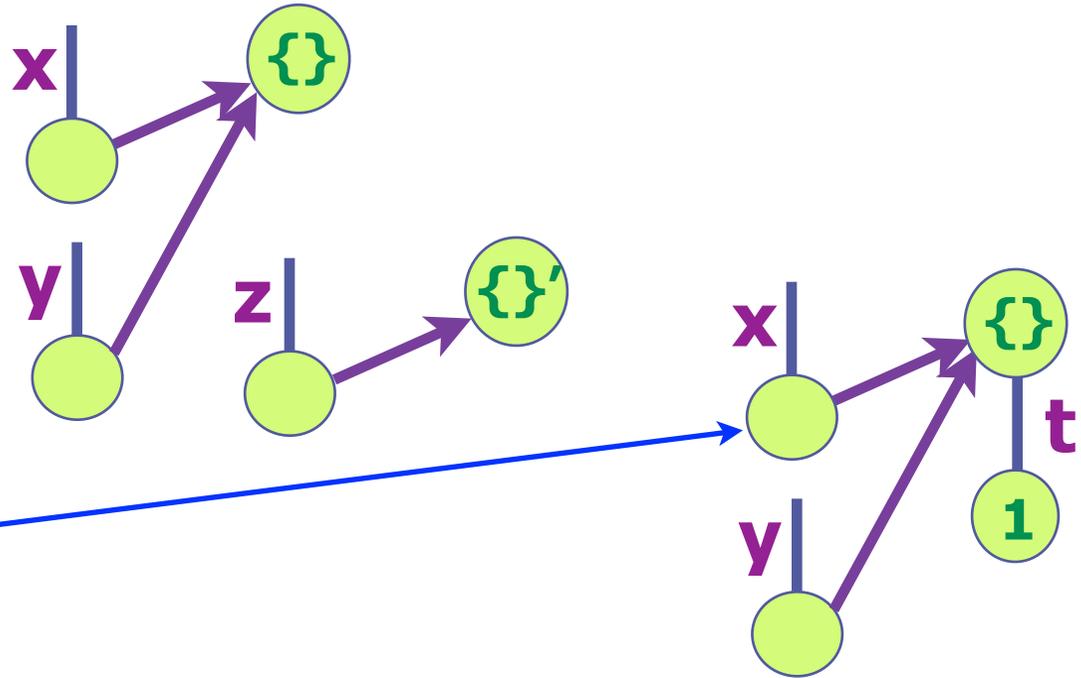
```
var z = {}; // la referencia a z  
// es diferente de  
// la de x e y
```

```
x.t = 1;
```

```
x.t => 1 // x accede al mismo
```

```
y.t => 1 // objeto que y
```

```
z.t => undefined
```



## Referencias a objetos

- ◆ Las variables que contienen objetos
  - solo contienen la referencia al objeto
- ◆ El objeto esta en otro lugar en memoria
  - indicado por la referencia
- ◆ Esto produce efectos laterales
  - como ilustra el ejemplo

# Identidad de objetos

## ◆ Las referencias a objetos afectan a la identidad

- porque identidad de objetos
  - ◆ es identidad de referencias
- los objetos no se comparan
  - ◆ se comparan solo las referencias
- es poco util si no se redefine

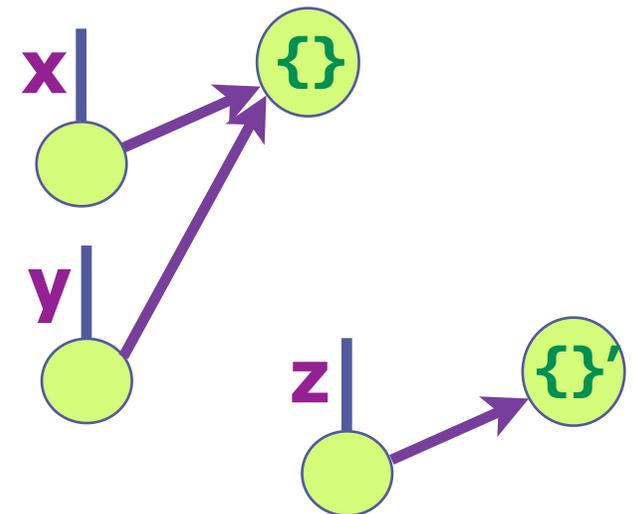
## ◆ Igualdad (debil) de objetos == y !=

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

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

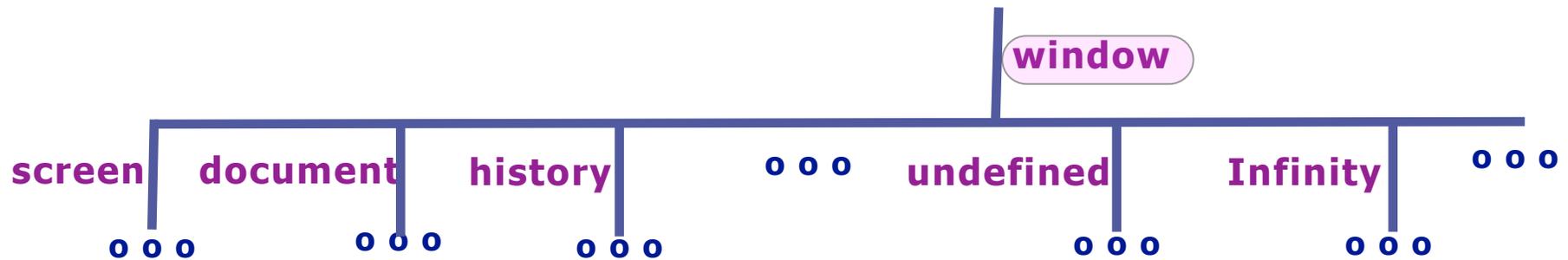
```
var z = {} // la referencia a z  
// es diferente de x e y
```

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



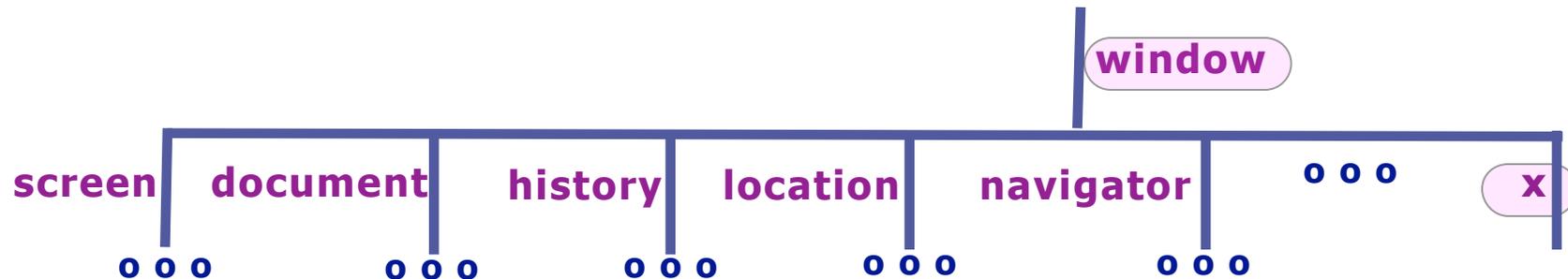
# Objeto window o this

```
Script: genera HTML al vue
Fecha: Sun Jan 19 2014 18:34:26 GMT+0100
pelicula.titulo => Avatar
pelicula['titulo'] => Avatar
```



- ◆ El entorno de ejecución de JavaScript es el **objeto global window**
  - El **objeto global window** tiene propiedades con información sobre
    - ◆ Objetos predefinidos de JavaScript, el navegador, el documento HTML, .....
- ◆ **window** se referencia también como **this** en el entorno global
  - La **propiedad document** de **window** se referencia como
    - ◆ **window.document**, **this.document** o **document**
- ◆ Documentación: <https://developer.mozilla.org/en-US/docs/Web/API/Window>

# Variables globales y el entorno de ejecución



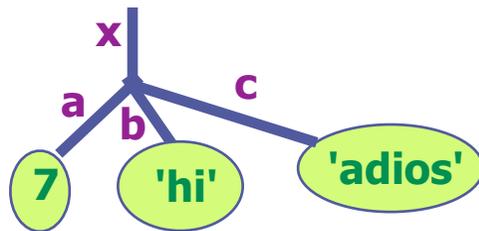
- ◆ Un programa JavaScript se ejecuta con el objeto **window** como entorno
  - una asignación a una variable no definida como **x = 1;**
    - ◆ Crea una nueva **propiedad de window** de nombre **x**, porque
      - **x = 1;** es equivalente a **this.x = 1;** y a **window.x = 1;**
- ◆ Olvidar definir una variable, es un error muy habitual
  - y al asignar un valor a la variable no definida, JavaScript no da error
    - ◆ sino que crea una nueva **propiedad de window**
  - Es un error de diseño de JavaScript y hay que tratar de evitarlo



# Sentencia for/in de JavaScript

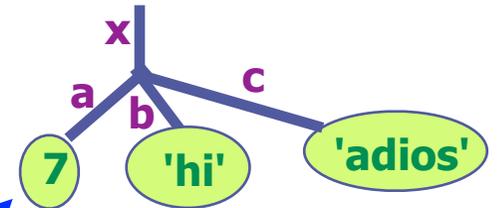
# Sentencia for/in

- ◆ **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

- ◆ En el ejemplo se utiliza **for (i in x) {...}**
  - para mostrar en una página Web
    - ◆ el contenido de las propiedades de un objeto



```
13-for_in.htm
<!DOCTYPE html><html>
<head><meta charset="UTF-8"></head>
<body>
<h3>Sentencia for/in:</h3>

<script type="text/javascript">
  var x = {a:7, b:'hi', c:'adios'};

  var i;
  for (i in x) {
    document.write("Propiedad " + i + " = " + x[i] + "<br>");
  }
</script>
</body>
</html>
```

UNREGISTERED

Sentencia for/in

File:///Users/jq/Desktop/MOOC\_FirefoxOS/t2/24

Sentencia for/in

**Sentencia for/in:**

Propiedad a = 7  
Propiedad b = hi  
Propiedad c = adios

# 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];
```

# window.screen

```
<!DOCTYPE html>
<html>
<head>
<title>DOM</title>
<meta charset="UTF-8">
</head>
<body>
```

```
<h2> Screen </h2>
```

```
    <!-- tabla con propiedades de screen -->
<table id="tabla">
  <tr><th> Propiedad </th><th> Valor </th></tr>
</table>
```

```
<script>
var i, tabla = document.getElementById("tabla");
```

```
for (i in screen){ //cada iteración genera una fila de la tabla
  tabla.innerHTML+="<tr><td>" + i + "</td><td> = " + screen[i] + "</td></tr>";
}
```

```
</script>
</body>
</html>
```

## Screen

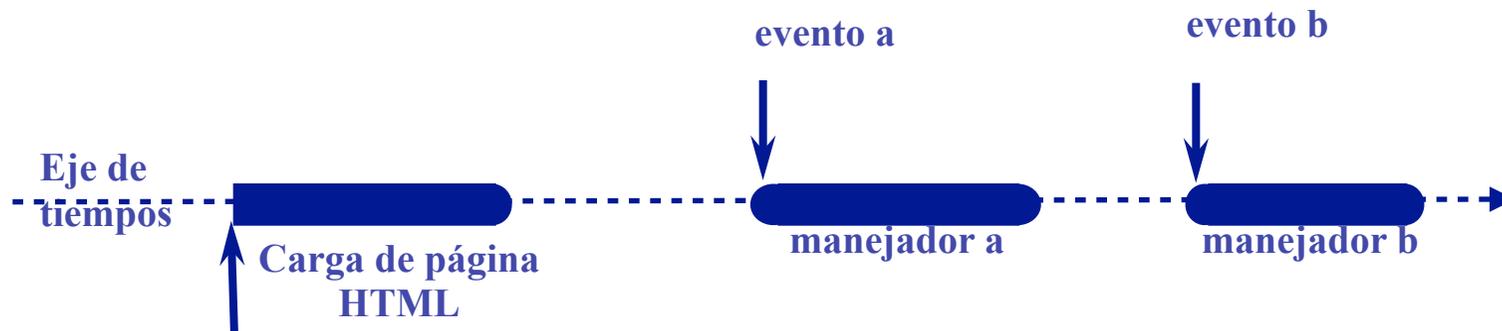
Propiedad	Valor
availWidth	= 2514
availHeight	= 1418
availTop	= 22
availLeft	= 46
pixelDepth	= 24
colorDepth	= 24
width	= 2560
height	= 1440



# Eventos Javascript

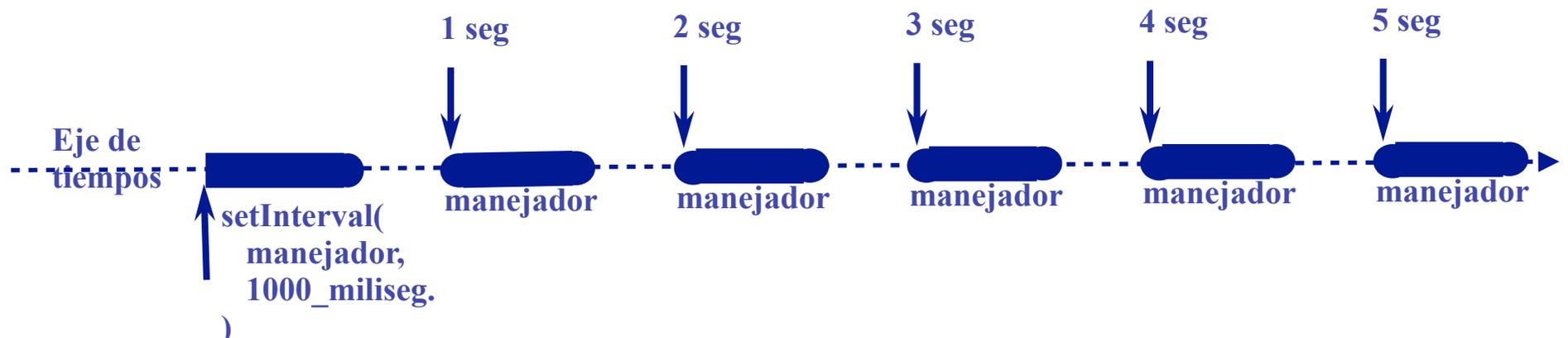
# Eventos y Manejadores

- ◆ JavaScript utiliza eventos para interactuar con el entorno
  - Hay eventos de muchos tipos
    - ◆ Temporizadores, clicks en boton, tocar en pantalla, pulsar tecla, ...
- ◆ Manejador (callback) de evento
  - función que se ejecuta al ocurrir el evento
- ◆ El script inicial debe configurar los manejadores (callbacks)
  - a ejecutar cuando ocurra cada evento que deba ser atendido



# Eventos periódicos con setInterval(...)

- ◆ JavaScript tiene una función **setInterval** (..)
  - para programar eventos periódicos
- ◆ **setInterval (manejador, periodo\_en\_milisegundos)**
  - tiene 2 parámetros
    - ◆ **manejador**: función que se ejecuta al ocurrir el evento
    - ◆ **periodo\_en\_milisegundos**: tiempo entre eventos periódicos



```
<!DOCTYPE html>
<html>
<head><title>Reloj</title>
  <meta charset="UTF-8">

<script type="text/javascript">

function mostrar_fecha( ) {
  var cl = document.getElementById("fecha");
  cl.innerHTML = new Date( );
}
</script>

</head>

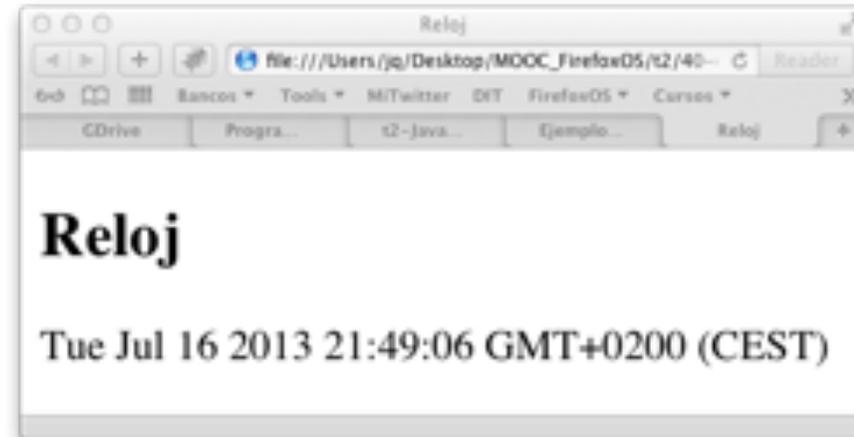
<body>
<h2>Reloj</h2>

<div id="fecha"><div>

<script type="text/javascript">
  mostrar_fecha();// muestra fecha al cargar
                    // actualiza cada segundo
  setInterval(mostrar_fecha, 1000);
</script>
</body>
</html>
```

# Reloj

- ◆ Utilizamos la función
  - `setInterval(manejador, T)`
    - ◆ para crear un reloj
- ◆ Cada segundo se muestra
  - El valor de reloj del sistema



# Eventos DOM

## ◆ Los eventos DOM se asocian a elementos HTML

- como atributos: 'onclick', 'ondblclick', 'onload', ....
  - ◆ donde el manejador es el valor asignado al atributo

## ◆ Ejemplo:

- ``
  - ◆ Código del manejador: `"this.src='img2.png'"` (valor del atributo)
    - `this` referencia el objeto DOM asociado al manejador

## ◆ Tutorial:

- [https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Event\\_attributes](https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Event_attributes)

# Eventos en HTML

- ◆ Definimos **2 manejadores** de evento en elem. `<img .... >`
  - Atributo **onclick**: muestra el icono enfadado
  - Atributo **ondblclick**: muestra el icono pasivo
- ◆ **this.src** referencia **atributo src** de `<img ..>`
  - **this** referencia objeto DOM asociado: `<img ..>`



```
05-event.htm UNREGISTERED
<!DOCTYPE html>
<html><head><meta charset="UTF-8"></head>
<body>

  <h4> Evento HTML</h4>

  
</body>
</html>
```

# Eventos definidos directamente en Javascript

- ◆ Los manejadores de eventos se pueden definir con
  - **objeto.addEventListener(evento, manejador)**
- ◆ También se pueden definir como propiedades
  - **objeto.evento = manejador**
    - ◆ objeto: objeto DOM al que se asocia el evento
    - ◆ evento: nombre (onload, onclick, onmouseover, etc. )
    - ◆ manejador: función ejecutada al ocurrir un evento
- ◆ Ejemplos
  - `img.addEventListener("onclick", function() {... código ...})`
  - `img.onclick=function() {... código...}`

- ◆ Los manejadores de evento se definen ahora en un script separado
  - El objeto `<img ...>` se identifica desde JavaScript con `getElementById(..)`
  - manejador se añade con método: `object.addEventListener(event, manejador)`
    - Actualmente se recomienda usar siempre `addEventListener(...)`

## Evento como propiedad

```
07-event_listener.htm UNREGISTERED
<!DOCTYPE html>
<html>
<head><meta charset="UTF-8"></head>
<body>

  <h4>Evento JS</h4>

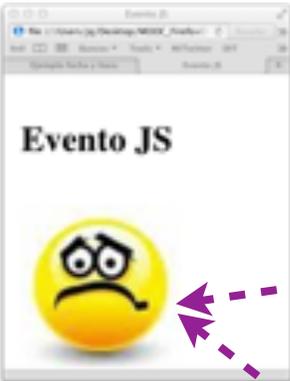
  

  <script type="text/javascript">
    var i=document.getElementById('i1');

    i.addEventListener('dblclick', function(){i.src='wait.png'});

    i.addEventListener('click', function(){i.src='scare.png'});

  </script>
</body>
</html>
```



- ◆ Los manejadores de evento se definen también así en un script separado
  - El objeto `<img ...>` se identifica desde JavaScript con `getElementById(..)`
  - Sintaxis de los manejadores: `object.event= manejador`

## Evento como propiedad

```
06-event_id.htm UNREGISTERED
<!DOCTYPE html>
<html>
<head><meta charset="UTF-8"></head>
<body>

  <h4>Evento JS</h4>

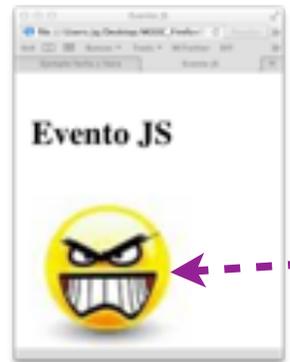
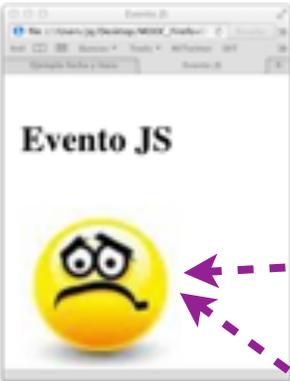
  <script type="text/javascript">

    var i=document.getElementById('i1');

    i.ondbclick = function(){ i.src='wait.png'; };

    i.onclick   = function(){ i.src='scare.png'; };

  </script>
</body>
</html>
```



- ◆ El script pasa a la cabecera, se **separa del documento HTML**
  - El código se mete en la función **inicializar()**, que se ejecuta al ocurrir **onload**
    - ◆ **onload** ocurre con la página HTML ya cargada y el objeto DOM construido



```
<!DOCTYPE html>
<html>
<head><title>Evento onload</title><meta charset="UTF-8">

<script type="text/javascript">
    function inicializar() {
        var i=document.getElementById('i1');
        i.ondblclick = function () {i.src='wait.png'};
        i.onclick    = function () {i.src='scare.png'};
    }
</script>
</head>

<!-- El arbol DOM ya esta construido al ocurrir onload -->
<body onload="inicializar()">

    <h4>Evento onload</h4>

</body>
</html>
```

Evento onload

Evento onload



# Botones y formularios en JavaScript

# Entradas y botones

- ◆ **Entrada:** un cajetín en pantalla para introducir texto en una aplicación
  - Se define con `<input type=text ..>`
    - ◆ el atributo `value="texto"` representa en texto dentro del cajetin
- ◆ **Botón:** elemento gráfico que invita a hacer clic
  - Se define con `<buton type=button ...>nombre</button>`

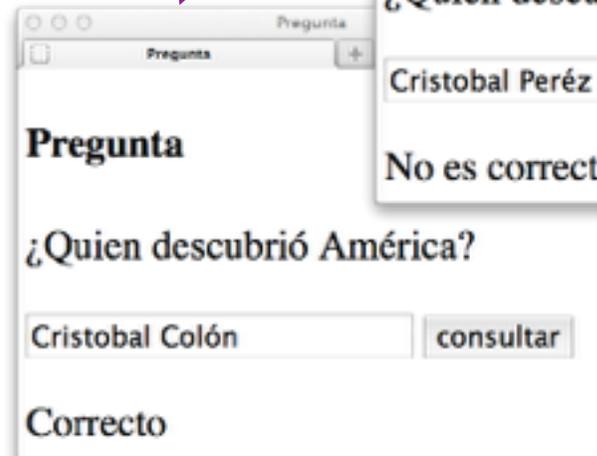
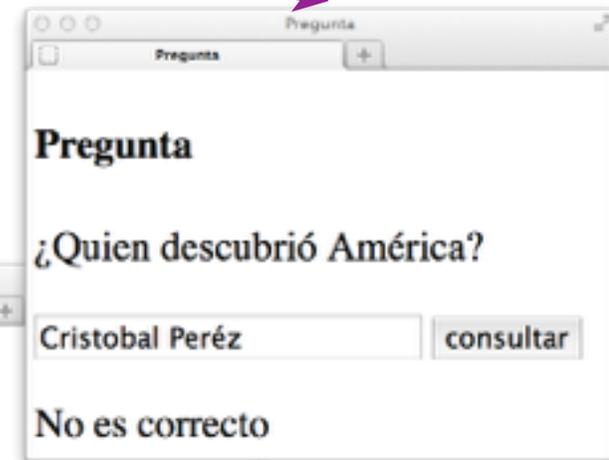
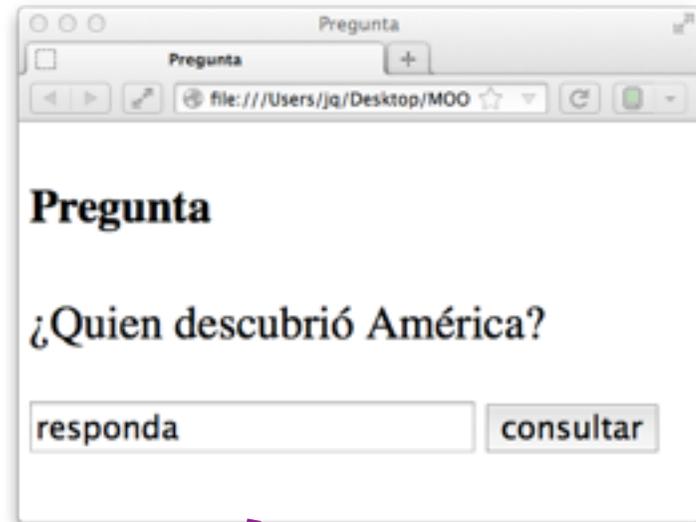
The image shows a code editor on the left and a browser window on the right. The code editor displays the following HTML code:

```
<!DOCTYPE html>
<html>
<head><meta charset="UTF-8"></head>
<body>
  <h4> Input y Button </h4>
  <input type="text" value="responda aquí"/>
  <button type="button">consultar</button>
</body>
</html>
```

The browser window, titled "Input y Button", shows the rendered output. It features a text input field containing the text "responda aquí" and a button labeled "consultar". Dashed arrows point from the corresponding HTML code in the editor to the rendered elements in the browser.

# Ejemplo Pregunta

- ◆ Esta WebApp plantea la pregunta
  - ¿Quien descubrió América?
    - ◆ para ilustrar como interaccionar
      - a través de formularios y botones
- ◆ Escribir la respuesta en el cajetín
  - y pulsar el boton “**consultar**”
    - ◆ para saber si es correcto
- ◆ Según sea la respuesta se responde
  - “**Correcto**” o “**No es correcto**”



# Pregunta

```
<!DOCTYPE html>
<html><head><title>Pregunta</title><meta charset="UTF-8">
<script type="text/javascript">
```

```
function res() {
  var respuesta = document.getElementById('respuesta');
  var resultado = document.getElementById('resultado');

  if (respuesta.value === "Cristobal Colón")
    resultado.innerHTML = "Correcto";
  else resultado.innerHTML = "No es correcto";
}
```

```
</script>
</head>
<body>
  <h4> Pregunta </h4>
  <p> ¿Quien descubrió América? </p>
```

```
<input type="text" id="respuesta" value="responda aquí">
```

```
<button type="button" onclick="res()">consultar</button>
```

```
<p><div id="resultado" /></p>
```

```
</body>
</html>
```

