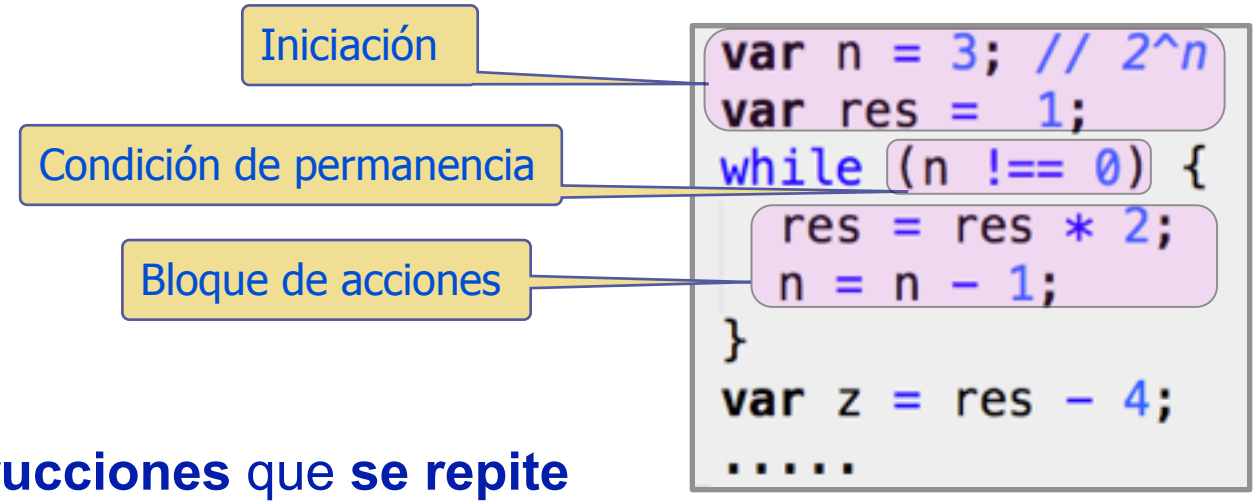
JavaScript: bucles

© Juan Quemada, DIT, UPM

Bucle



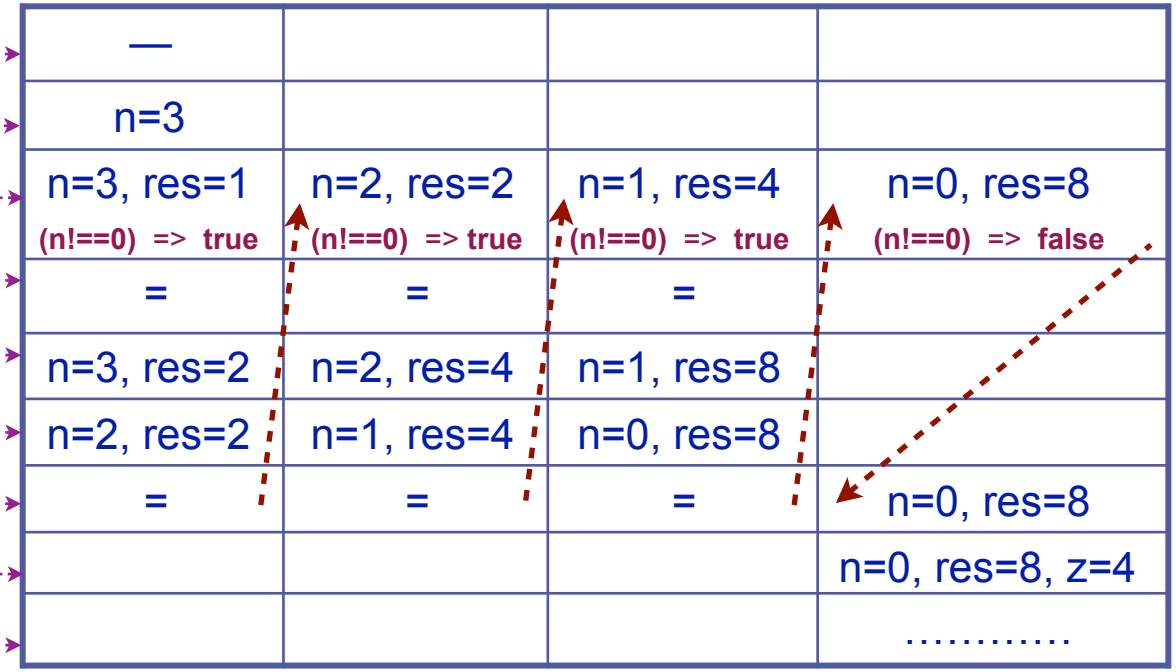
- Bucle: **bloque de instrucciones** que **se repite**
  - mientras se cumple una **condición** de permanencia
    - Lo ilustramos con el cálculo de 2 elevado a n: 2^n (2*2*…*2)
      - Además existen otros tipos de bucles que no vemos aquí: **for**, **for/in, do/while, …**
- Un bucle tiene 3 partes
  - **Iniciación**: fija los valores de arranque del bucle en la 1ª iteración
    - La iniciación se realizar aquí en instrucciones anteriores a la sentencia del bucle
  - **Condición de permanencia**: controla la finalización del bucle
    - El bucle se ejecuta mientras la condición sea **true**
  - **Bloque de acciones**: acciones realizadas en cada iteración del bucle
    - Realiza el cálculo de forma iterativa hasta que la condición de permanencia indica que se ha obtenido el resultado

© Juan Quemada, DIT, UPM

Ejecución del bucle

- La ejecución del bucle while esta controlada por la condición de permanencia: **(n !== 1)**
  - Mientras **n** sea distinto de 1 ejecutará el bucle
  - Cuando **n** sea igual a 1 saldrá del bucle
    - El estado del programa determina en cada evaluación de la condición del bucle, si **(n !== 1)** es **true** o a **false**

```
var n = 3; // 2^n
var res =  1;
while (n !== 0) {
   res = res * 2;
   n = n - 1;
}
var z = res - 4;
.....
```

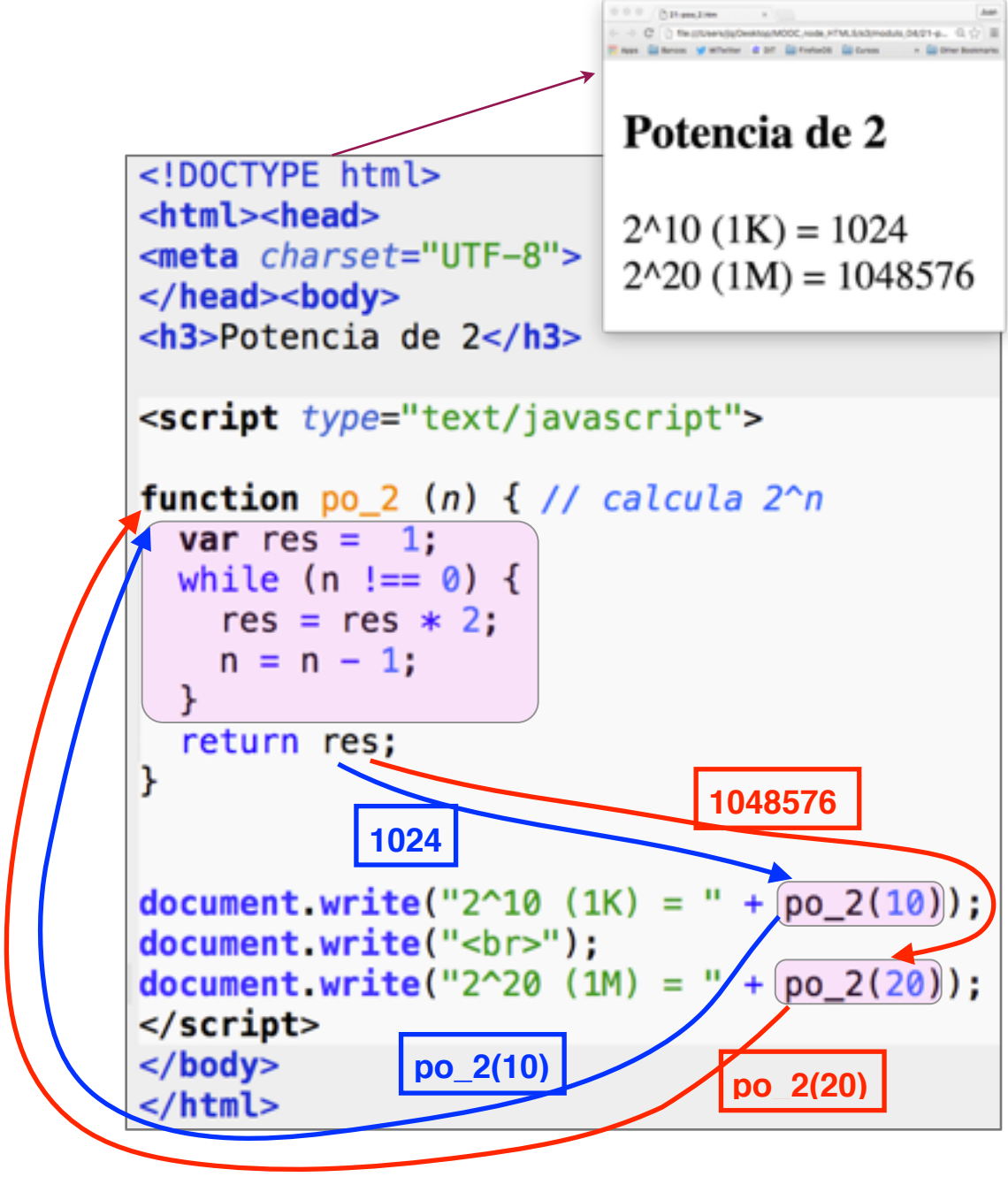


Ejemplo función po_2

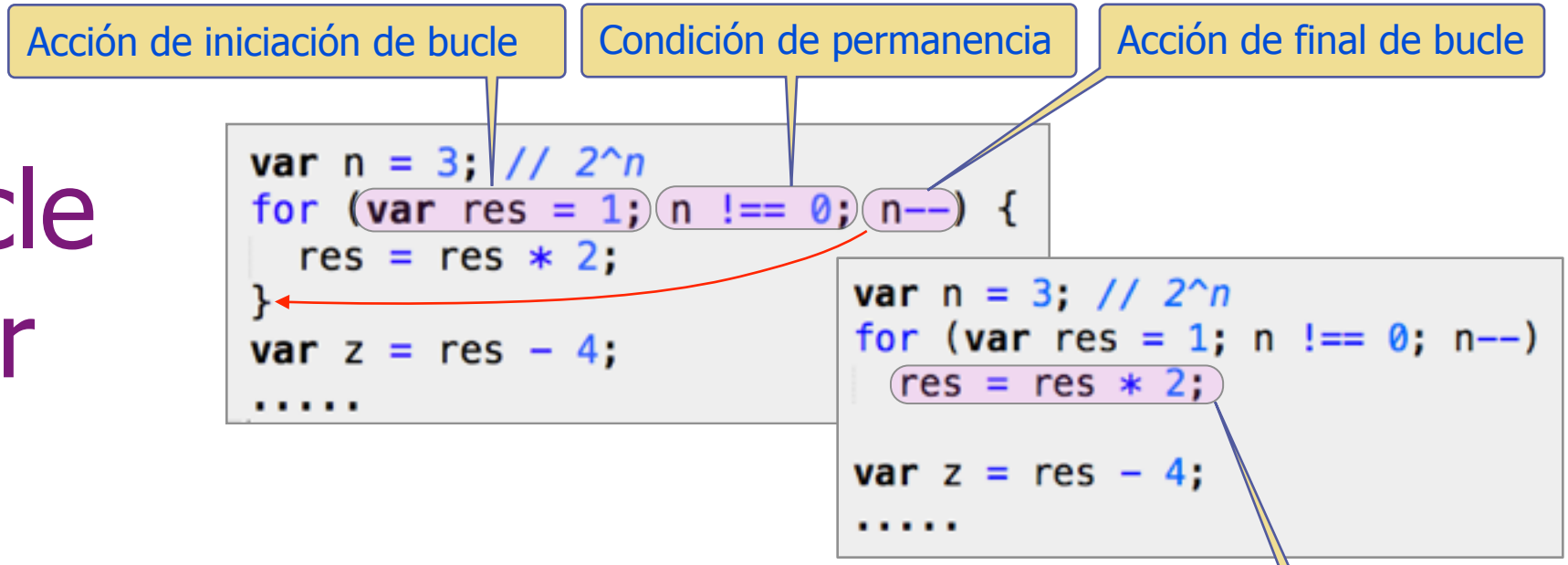
El bucle se inicia con los valores iniciales del **parámetro n** y de la **variable res**. **n** es el número cuya potencia de 2 que vamos a calcular. **res** es una variable local de la función (definida dentro de su bloque de acciones y solo es visible en su interior), cuyo valor inicial es 1.

El bloque de acciones va delimitado entre llaves: **{}** y se ejecuta en cada iteración del bucle. La primera instrucción **res = res * 2**, calcula la potencia de forma incremental. La segunda **resta 1 a n** para llevar la cuenta de multiplicaciones de **res * 2**, una en cada iteración, para multiplicar n veces.

El bucle finaliza cuando la condición de permanencia **(n !== 1)** sea **false** (n llega a cero). La condición de permanencia va siempre delimitada entre paréntesis: **()**.



bucle for



La **sentencia for** es mas compacta que while. La gestión del bucle (entre paréntesis) va detrás de la palabra reservada **for** y consta de tres partes separadas por ";":

1) **Iniciación**: define e inicia la variable "i". La variable "n" se define fuera del bucle.

2) **Condición de permanencia**: el bucle se ejecuta mientras la condición sea **true**. Se sale del bucle en cuanto pase a **false**. Similar a la condición de permanencia de while.

3) **Acción final del bucle**: se ejecuta al final de cada iteración en la ejecución del bloque de código del bucle. **n--** decrementa el número hasta llegar a 1. Lleva la cuenta del número de multiplicaciones por 2.

El bloque de acciones se delimita con llaves **{}**, pero si un bloque tiene solo una sentencia, las llaves pueden omitirse (segundo ejemplo), como en cualquier otra sentencia que incluya un bloque.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference#Statements
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Statements

STATEMENT	SINTAX	DESCRIPCIÓN DE LA SENTENCIA JAVASCRIPT
block	{ statements };	Agrupar un bloque de sentencias (como 1 sentencia)
break	break [label];	Salir del bucle o switch o sentencia etiquetada
case	case expression:	Etiquetar sentencia dentro de sentencia switch
continue	continue [label];	Salto a sig. iteración de bucle actual/etiquetado
debugger	debugger:	Punto de parada (breakpoint) del depurador
default	default:	Etiquetar sentencia default de sentencia switch
do/while	do statement while(expression);	Alternativa al bucle while con condición al final
empty	;	Sentencia vacía, no hace nada
expression	expression;	Evaluar expresión (incluyendo asignación a variable)
for	for(init; test; incr) statement	Bucle for: "init": inicialización; "test": condición;"incr": acciones final bucle
for/in	for (var in object) statement	Bucle for/in: Enumera las propiedades del objeto "object"
function	function name([param[,...]]) { body }	Declarar una función llamada "name"
if/else	if (expr) statement1 [else statement2]	Ejecutar statement1 o statement2
label	label: statement	Etiquetar sentencia con nombre "label"
return	return [expression];	Devolver un valor desde una función
switch	switch (expression) { statements }	Multiopción con etiquetas "case" o "default"
throw	throw expression;	Lanzar una excepción
try	try {statements} [catch { statements }] [finally { statements }]	Gestionar excepciones
strict	"use strict";	Activar restricciones strict a script o función
var	var name [= expr] [,...];	Declarar e initializar una o mas variables
while	while (expression) statement	Bucle básico con condición al principio
with	with (object) statement	Extender cadena de ámbito (no recomendado)

JavaScript:
Arrays

Arrays

- Array: **lista ordenada** e **indexable** de **n** elementos heterogéneos
  - Se suelen construir con el literal de array: **[1, 4, 2, 23]**
    - Sus elementos son accesibles con un índice de rango: **0** a **n-1**
      - **a[k]** es el elemento **k+1**
  - El **tamaño de un array** es su propiedad **a.length**
    - El tamaño máximo de un array es: **2^{32}-2 elementos**
  - En castellano se denomina matriz o vector
- Un array puede contener
  - números, strings, undefined, objetos, arrays, …
    - A un array dentro de un array se accede con 2 indexaciones
- Indexar elementos inexistentes devuelve undefined
  - incluyendo índices mayores que a.length
    - Cambiando la longitud del array reducimos su tamaño

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

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

a[3]            => undefined
a[9]            => undefined

a.length        => 5

a.length = 2    => 2
a               => [1, 'a']
```

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

Sumatorio: versión 1

- La calculadora ilustra aquí el uso de arrays
  - Creando la función sumatorio de n números: **Σ x**
- El sumatorio de los números 1, 2, 3, 4 es:
  - **Σ (1, 2, 3, 4) = 1 + 2 + 3 + 4 = 10**
- Formato **CSV** (Coma Separated Values)
  - String con **valores separados por comas**
    - por ejemplo: **"1, 2, 3, 4"** o **"1,2,3,4"**
  - Muy utilizado bases de datos, hojas de calculo, …
- Los números se introducen en el cajetín
  - En formato CSV (Coma Separated Values)

```
<!DOCTYPE html><html>
<head><title>Ejemplo</title><meta charset="utf-8">

<script type="text/javascript">

function vaciar () {
  document.getElementById("n1").value = "";
}

function sumatorio() {
  var num = document.getElementById("n1");

  var lista = num.value.split(",");

  var i = 0, acc = 0;
  while ( i < lista.length ) {
    acc = acc + (+lista[i]);
    i++;
  }

  num.value = acc;
}
</script>
</head><body>

  Número:
  <input type="text" id="n1" onclick="vaciar()">
  <p>
  <button onclick="sumatorio()">Σ x</button>
</body></html>
```

En este bucle se suman los números del array obtenido con split().

Convierte string a number (suma aritmética).

El método **split(",")** transforma un string en un array, donde sus elementos son las partes separadas por coma **(",")**.
Por ejemplo, **"1, 2, 3, 4". split(",") => ["1", " 2", " 3", " 4"]**
Mas info: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/split

Sumatorio: versión 2

- Ejemplo con la calculadora ligeramente modificada
  - El bloque de código que calcula **Σ x** se cambia por
    - **Una sola sentencia** equivalente al bloque anterior
- Los bloques con una sola sentencia
  - pueden **omitir las llaves {..}** (como en esta versión)
    - Aunque las llaves suelen incluirse para mejor legibilidad
- Operadores de la sentencia: **acc += +lista[i++];**
  - El operador **+=** suma a la variable la expresión asignada
  - El operador **+** de **+lista[i++]** convierte el string a number
  - **[i++]** accede al array antes de incrementar (post-incremento)

```
<!DOCTYPE html><html>
<head><title>Ejemplo</title><meta charset="utf-8">

<script type="text/javascript">

function vaciar () {
  document.getElementById("n1").value = "";
}

function sumatorio() {
  var num = document.getElementById("n1");

  var lista = num.value.split(",");

  var i = 0, acc = 0;
  while ( i < lista.length )
    acc += +lista[ i++ ];

  num.value = acc;
}
</script>
</head><body>

  Número:
  <input type="text" id="n1" onclick="vaciar()">
  <p>
  <button onclick="sumatorio()">Σ x</button>
</body></html>
```

Los bloques de una sentencia pueden omitir las llaves **{...}** que delimitan el bloque. Suelen incluirse por legibilidad.

JavaScript:
Funciones como objetos

© Juan Quemada, DIT, UPM

Clase Function y literal de función

- Las funciones son **objetos** de pleno derecho que pertenecen a la **clase Function**
  - pueden asignarse a **variables, propiedades, parámetros de funciones, …**
    - Doc: https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Function
- **function (..){..}** es un literal de función que crea un objeto función (sin nombre)
  - El objeto suele asignarse a una variable o parámetro, que le da su nombre
    - El literal "**function () {}**" crea una función vacía como la creada con el constructor "**new Function()**"
  - La definición de funciones con nombre, también crea objetos de la clase Function
- El **operador paréntesis** permite invocar (ejecutar) objetos de tipo función
  - pasando una lista de parámetros al objeto función, p.e. comer('José','paella')

```
var comer = function(persona, comida) {
    return (persona + " come " + comida);
};

comer('José','paella');          =>  'José come paella'
```

Operador invocación de una función: (...)

- El objeto función puede asignarse o utilizarse como un valor
  - el objeto función contiene el código de la función
- el operador **(...)** invoca una función ejecutando su código
  - Solo es aplicable a funciones (objetos de la clase Function), sino da error
    - Puede incluir parámetros separados por coma, accesibles en el código de la función

```
var comer = function(persona, comida) {
    return (persona + " come " + comida);
};

var x = comer;                    // asigna a x el código de la función
x('José','paella');   =>  'José come paella'
x();                  =>  'undefined come undefined'

var y = comer();                  // asigna a y el resultado de invocar la función
y;                    =>  'undefined come undefined'
```

Iteradores de ES5

JavaScript 1.5 introduce métodos iteradores de arrays que sacan partido de las funciones

- **forEach(function(elem, index, array){…})**:
  - Iterador que ejecuta la función secuencialmente para cada elemento del array
    - Parámetros: **elem** (elemento i), **index** (indice i), **array** (array sobre el que se itera)
  - Equivale a un **bucle,** que itera desde el primero al último elemento de un array
- **map(function(elem){…})**: mapa elementos del array a valor de retorno de la función
  - por ejemplo, **[1,2].map(function(e){return e+1;}) => [2,3]**
- **filter(function(elem){…})**: filtra los elementos del array de acuerdo a la función
- Y otras funciones: https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array

```
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'
```

forEach(function(x){…}) itera el bloque (código) de la función de 0 a length-1, pasando en cada iteración un nuevo elemento del array en la variable x.

El método **split("")** transforma un string en un array con todos sus caracteres separados como elementos.
Por ej., **"Mi casa". split("")** => **["M","i"," ","c","a","s","a"]**

función veces(..) con bucle while

- En este ejemplo se implementa la función veces con un bucle while
  - Es **totalmente equivalente** a la implementación anterior con el iterador **forEach()**
    - La condición de permanencia en el bucle no es necesaria en el iterador porque esta implícita,
      - El **parámetro index** de la función itera de **0** a **array.length-1**
    - La **parámetro index** de la función del iterador toma los mismos valores que el **indice i** del bucle
    - El **bloque de código de la función** itera exactamente igual que el **bloque de código del bucle**

40-veces_while.js — UNREGISTERED

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

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

JavaScript:
Ámbitos de visibilidad y cierres (closures)

© Juan Quemada, DIT, UPM

Ambito y definiciones locales de una función

```
var ambito = "global";

function ambitoLocal () {
    var ambito = "local";
    return ambito;
};

// acceso a variable global
ambito          =>  "global"

// función global
//    - accede a variable local
ambitoLocal()  =>  "local"
```

La variable local ámbito tapa a la variable global del mismo nombre en la función.

- Una **función** puede tener
  - **definiciones locales** de variables y funciones
    - Estas son **visibles** solo **dentro de la función**
- Las **variables** y **funciones locales** tienen **visibilidad sintáctica**
  - son **visibles en todo el bloque de código** de la función, incluso antes de definirse
    - **OJO!** Se recomienda definir variables y funciones al **principio de la función**
- Las **variables** y **funciones globales** son **visibles** también **dentro de la función**
  - Siempre que no sean **tapadas** por otras locales del **mismo nombre**
    - Una definición **local tapa** a una **global** del **mismo nombre**

Funciones anidadas

```
var s1 = "Hola, ";

function exterior () {
    var s2  = "que ";

    function interior () {
        var s3 = "tal";
        return s1 + s2 + s3;
    }

    return interior;
};

exterior()    => function interior()..
exterior()()  => "Hola, que tal"
```

El operador paréntesis **aplicado 2 veces** sobre la función exterior invoca la función interior.

- Las **funciones locales** pueden tener
  - otras **funciones locales** definidas en su **interior**
- Las **variables externas** a las funciones locales
  - son **visibles en el interior** de estas funciones
- Además, una **función es un objeto** y puede
  - **devolverse** como **parámetro** de otra función
    - La función exterior devuelve la función interior como parámetro
- El operador paréntesis sobre la función **exterior** devuelve la función **interior** (su código)
  - El **operador paréntesis aplicado 2 veces** en la función exterior, en cambio, invoca la función interior
    - La función **interior tiene visibilidad** sobre las **variables exteriores s1 y s2**, y puede concatenar s1+s2+s3

Cierres o closures

- Un cierre (closure): función que **encapsula un conjunto de definiciones locales**
  - que solo son **accesibles** a través de una **interfaz** (función u objeto)
- 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
- En este ejemplo la **interfaz** es la función **contar()**
  - la **función contar** devuelve el valor de la **variable contador** y lo incrementa después
    - El cierre **encapsula** la **variable contador** y la **función contar**
      - **Ninguna instrucción** fuera del cierre **puede modificar** la variable contador, solo la función contar()
  - La variable **entero_unico** contiene la invocación del cierre (**enteroUnico ()**)
    - Esta devuelve la función contar, de forma que **invocar entero_unico()** es lo mismo que **invocar contar()**
      - Al invocar el cierre, sus **variables** se crean y **siguen existiendo hasta que el objeto se destruye**

```
function enteroUnico () {
    var contador = 0;
    function contar () { return contador++; };
    return contar;
};
     // asignamos el objeto función contar
var entero_unico = enteroUnico();

entero_unico()     =>  0  // invoca contar()
entero_unico()     =>  1  // invoca contar()
```

```
// definición equivalente a la anterior
// sin dar nombres a las funciones.

var entero_unico = function () {
    var contador = 0;
    return function () { return contador++; };
} ();

entero_unico()     =>  0
entero_unico()     =>  1
```

() invoca la función exterior asignando la función retornada por esta a entero_único.

© Juan Quemada, DIT, UPM

JavaScript:
Introducción a la librería jQuery

© Juan Quemada, DIT, UPM

Librerías JavaScript: jQuery

- Las **librerías JavaScript** facilitan la programación **multi-navegador**
  - Se diseñan para funcionar en IE, Firefox, Safari, Chrome, Opera, ...
    - **Ahorran mucho tiempo** -> se deben utilizar siempre que existan
      - Por ejemplo. **jQuery**, jQuery UI, D3, Bootstrap, Prototype, PhoneGap, …
- **jQuery** es muy popular (http://jquery.com/) -> Lema: **write less, do more**
  - Se distribuye con licencia abierta (MIT) y facilita mucho la programación JavaScript de cliente
    - Procesa objetos DOM, gestiona eventos, animaciones, estilos CSS, Ajax, ..
  - **jQuery 1.x** y **2.x** son **dos versiones** de la librería con la **misma interfaz (API)**
- **jQuery 1.x** (última versión 7-1-15: **1.11.3**)
  - Fue la primera y mantiene compatibilidad desde **Explorer 6+**
    - Esta optimizada para compatibilidad legacy y es **más pesada** que jQuery 2.x
- **jQuery 2.x** (última versión 7-1-15: **2.1.4**)
  - Creada recientemente y mantiene compatibilidad desde **Explorer 9+**
    - Está optimizada para móviles y es **mucho mas ligera** que jQuery 1.x

	Internet Explorer	Chrome, Firefox	Edge	Safari	Opera	iOS	Android
jQuery 1.x	6+	(Current - 1) or Current	Current	5.1+	12.1x, (Current - 1) or Current	6.1+	2.3, 4.0+
jQuery 2.x	9+						

La función jQuery

- **Objeto jQuery**: representación **equivalente a un objeto DOM**
  - **Mas eficaz de procesar**, tanto de forma individual, como en bloque (array)
- Función jQuery: **jQuery("<selector CSS>")** o **$("<selector CSS>")**
  - devuelve el **objeto jQuery** que **casa con <selector CSS>**
    - Si **no casa ninguno**, devuelve **null** o **undefined**
  - **<selector CSS>** selecciona objetos DOM **igual que CSS**

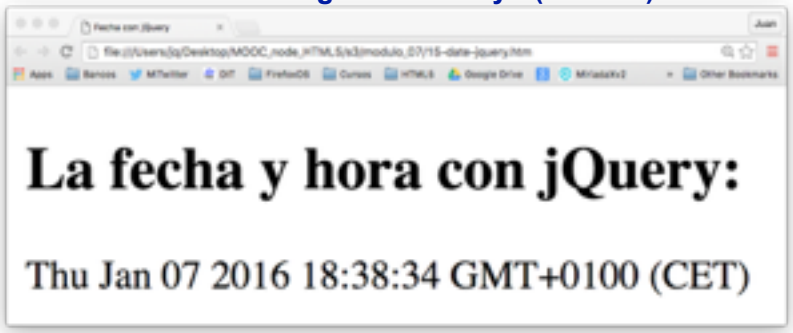
```
document.getElementById("fecha")
        // es equivalente a:
$("#fecha")
```

- Además la función jQuery **convierte objetos DOM** y **HTML** a **objetos jQuery**

```
$(objetoDOM)                          //  convierte objetoDOM a objeto jQuery
$("<ul><li>Uno</li><li>Dos</li></ul>")   //  convierte HTML a objeto jQuery
```

Fecha y hora con jQuery

- Una **librería JavaScript** externa se identifica por su **URL**:
  - <script type="text/javascript" **src="jquery-2.1.4.min.js.js"** > </script>
- **$("#fecha")** obtiene el objeto jQuery
  - del **elemento HTML** con **id="fecha"**
- **$("#fecha").html(new Date())**
  - inserta **new Date()** como **HTML interno**
    - del **objeto jQuery** devuelto por **$("#fecha")**
  - es equivalente a
    - **document.getElementById("fecha").innerHTML = new Date();**




```
<!DOCTYPE html>
<html>
<head>
<title>Fecha con jQuery</title>
<script type="text/javascript"
        src="jquery-2.1.4.min.js">
</script>
</head>

<body>
<h2>La fecha y hora con jQuery:</h2>

<div id="fecha"></div>

<script type="text/javascript">
    $('#fecha').html(new Date( ));
</script>
</body>
</html>
```

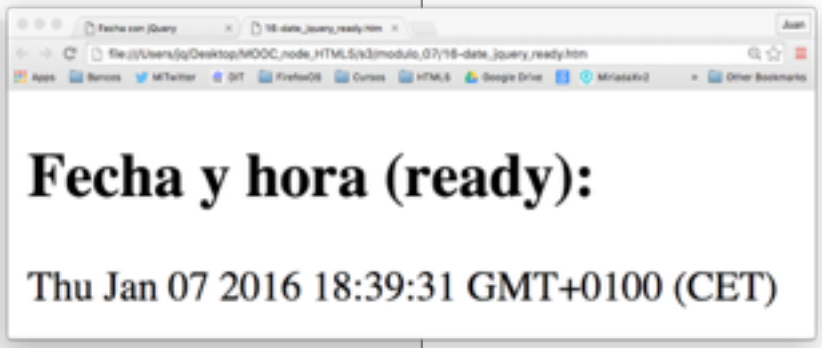
Selecciona el elemento DOM con atributo id="fecha": .

Asigna la fecha y hora a innerHTML del objeto DOM seleccionado.

Función ready: árbol DOM construido

- **$(document).ready(function() { ..código..)}**
  - Ejecuta el código (bloque) de la función cuando el **árbol DOM está construido**
    - Es decir, dicho bloque se ejecuta cuando ocurre el evento **onload** de <body>
  - Se recomienda utilizar la invocación abreviada: **$(function() { ..código.. })**

```
<!DOCTYPE html>
<html>
<head>
  <script type="text/javascript" src="jquery-2.1.4.min.js"></script>

  <script type="text/javascript">

    $(function() { $('#fecha').html(new Date( )); });
  </script>
</head>

<body>
<h2>Fecha y hora (ready):</h2>

<div id="fecha"></div>
</body>
</html>
```



Cache y CDN (Content Distribution Network)

- Cache: contiene recursos cargados anteriormente durante la navegación
  - La cache identifica los recursos por igualdad de URLs
    - Un nuevo recurso se carga de alguna cache (navegador, ..) si tiene el mismo URL que otro ya guardado
      - Cargarlo de la cache es más rápido que bajarlo del servidor, especialmente de la del navegador
- CDNs Web: utilizan el mismo URL (a Google, jQuery, …) en muchas páginas
  - Así se maximiza la probabilidad de que los recursos estén ya en la cache

```
<html>
<head>
<script type="text/javascript"
        src="https://code.jquery.com/jquery-2.1.4.min.js" >
</script>

<script type="text/javascript">
  $(function() { $('#clock').html(new Date( )); });
</script>
</head>
<body>
<h2>Fecha y hora, con CDN de jQuery</h2>

<div id="clock"></div>
</body>
</html>
```

Selectores tipo CSS de jQuery

```
SELECTORES DE MARCAS CON ATRIBUTO ID
$("h1#d83")      /* devuelve objeto con marca h1 e id="d83" */
$("#d83")        /* devuelve objeto con con id="d83" */

SELECTORES DE MARCAS CON ATRIBUTO CLASS
$("h1.princ")    /* devuelve array de objetos con marcas h1 y class="princ" */
$(".princ")      /* devuelve array de objetos con class="princ" */

SELECTORES DE MARCAS CON ATRIBUTOS
$("h1[border]")          /* devuelve array de objetos con marcas h1 y atributo border */
$("h1[border=yes]")      /* devuelve array de objetos con marcas h1 y atributo border=yes */

SELECTORES DE MARCAS
$("h1, h2, p")   /* devuelve array de objetos con marcas h1, h2 y p */
$("h1 h2")       /* devuelve array de objetos con marca h2 después de h1 en el árbol */
$("h1 > h2")     /* devuelve array de objetos con marca h2 justo después de h1 en arbol */
$("h1 + p")      /* devuelve array de objetos con marca p adyacente a h1 del mismo nivel */
.................
```

Métodos de manipulación

- Los objetos jQuery se manipulan con métodos de la librería
  - Más información en: http://api.jquery.com/category/manipulation/
- Método **html(<código html>)**
  - **$("#id3").html("Hello World!")** sustituye el **innerHTML** del elemento con **id="id3** por **"Hello World!"**
    - Es equivalente a: **document.getElementById("fecha").innerHTML = "Hello World!"**
- Método **html()**
  - **$("#id3").html()** devuelve el **innerHTML** del elemento con **"#id3"**
- Método **append("Hello World!")**
  - **$("#id3").append("Hello World!")** añade "Hello World!" al **innerHTML** del elemento con id=**"id3"**
- Método **val(<valor>)**
  - **$("#id3").val("3")** asigna el **valor "3"** al atributo **value** del elemento con id=**"id3"**
- Método **attr(<atributo>, <valor>)**
  - **$(".lic").attr("rel", "license")** asigna **"license"** al atributo **rel** a todos los elementos con **class="lic"**
    - Una **gran ventaja de jQuery** es que **puede hacer asignaciones en bloque** sin utilizar bucles como aquí!
- Método **addClass(<valor>)**
  - **$("ul").addClass("visible")** asigna el **valor "visible"** al atributo **class** de todos los elementos **<ul>**
    - Una **gran ventaja de jQuery** es que **puede hacer asignaciones en bloque** sin utilizar bucles como aquí!

Los 4 modos de la función jQuery

- **Acceso a DOM**: **$("selector CSS")**
  - Devuelve un **array** con los **objetos jQuery** que casan con **<selector CSS>**
    - Programas mas **cortos**, **eficaces** y **multi-navegador** que con JavaScript directamente
- **Transformar HTML en jQuery**: **$("
Uno
Dos
")**
  - Devuelve **objeto jQuery** equivalente al **HTML**
    - Mecanismo simple para convertir **HTML** en **jQuery**
- **Transformar DOM en jQuery**: **$(objetoDOM)**
  - Transforma objeto DOM en objeto jQuery equivalente
    - Tiene compatibilidad total con DOM y con otras librerías basadas en DOM
- **Esperar a DOM-construido**: **$(function(){..código..})**
  - **Ejecuta el código de la función** cuando el **árbol DOM está construido**
    - Equivalente a **ejecutar el código** asociado al evento **onload**

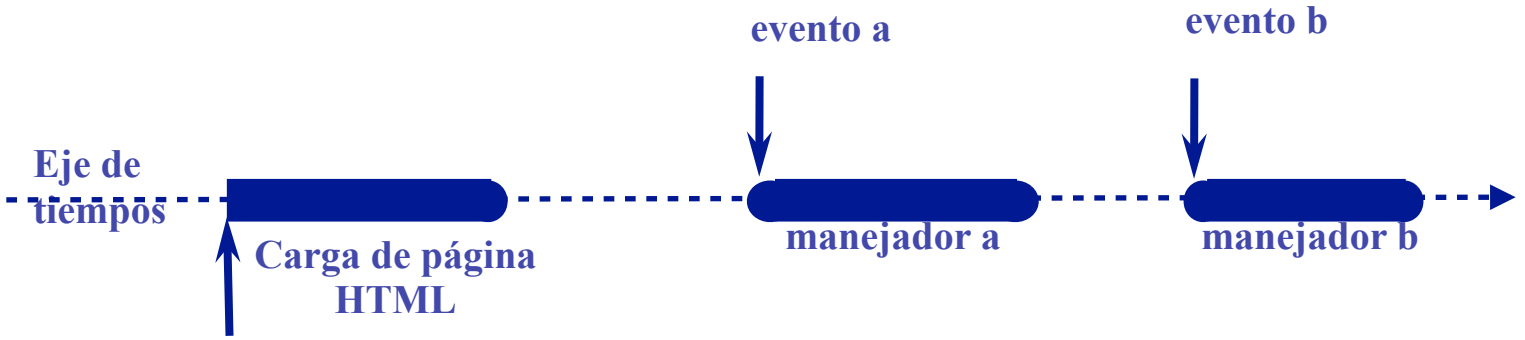
© Juan Quemada, DIT, UPM

JavaScript:
Eventos DOM y jQuery

© Juan Quemada, DIT, UPM

Eventos y Manejadores

- JavaScript utiliza eventos para interaccionar con el entorno
  - Hay eventos de muchos tipos
    - Temporizadores, clicks en boton, tocar en pantalla, pulsar tecla, ...
- Un manejador (callback) de evento
  - Es una 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 en HTML


```
<!DOCTYPE html>
<html><head><title>Evento HTML</title>
<style> body{text-align:center;} </style>
</head><body>
  <h1>Eventos HTML</h1>

  <img src="lamp_off.jpg"
       ondblclick="this.src='lamp_off.jpg';"
       onclick="this.src='lamp_on.jpg';"
       />
</body>
</html>
```

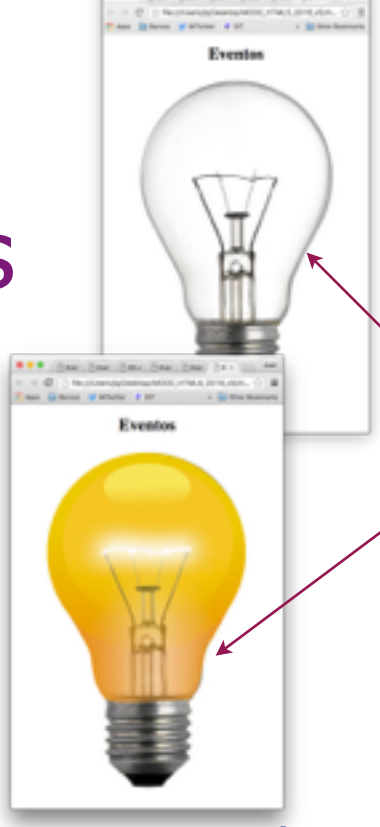
- En programas grandes es recomendable **separar HTML**, **CSS** y **JavaScript**
  - Debe estar en **ficheros diferentes** (o al menos en **partes claramente separadas)**
    - De esta forma el **cuerpo (body) solo contiene HTML** y la **cabecera** (head) incluye **CSS** y **JavaScript**
- La forma habitual de definir **eventos directamente en JavaScript** es con
  - **objetoDOM.addEventListener(evento, manejador)**
    - También existe un método **removeEventListener(..)** para **eliminar el evento**
      - https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener
  - Tradicionalmente el manejador se asignaba a una propiedad con el nombre del atributo HTML
    - **objeto.evento = manejador**
- jQuery usa la función **on()** para definir eventos y **off()** para eleminarlos
  - **objetoJQuery.on(evento, manejador)**
    - http://api.jquery.com/category/events/event-handler-attachment/

Eventos en JavaScript


```
<!DOCTYPE html>
<html><head><title>Evento</title><meta charset="UTF-8">
<style> body{text-align:center;} </style>

  <script type="text/javascript">

    function inicializar() {
       var img = document.getElementById('i1');

       img.addEventListener("dblclick", function () {img.src='lamp_off.jpg'});
       img.addEventListener("click",    function () {img.src='lamp_on.jpg'});
    }
  </script>
</head>   <!-- El arbol DOM debe estar construido al ocurrir onload -->
<body onload="inicializar()">
  <h1>Eventos</h1>
  <img src="lamp_off.jpg" id="i1"/>
</body></html>
```

- Los **eventos** se definen asociados a **onload** para que el **árbol DOM esté ya está construído**
  - El manejador del **evento onload** hay que invocarlo o definirlo o en HTML o en sino en un script al final
- La norma de JavaScript incluye **muchos eventos** diferentes
  - Se pueden ver en https://developer.mozilla.org/en-US/docs/Web/Events
    - Los **nombres de los eventos** son **diferentes** del de los atributos de eventos
  - La forma tradicional (objeto.evento = manejador) esta en desuso y no la ilustramos
- El **manejador** del evento es una **función** (objeto de la clase Función)
  - Puede pasarse directamente como un literal de función con el código del manejador (como aquí)
    - o cualquier otro objeto Function, como por ejemplo el nombre de una función definida en otro lugar
      - OJO! debe ser la función (su código) y no su invocación

Eventos en jQuery




```
<!DOCTYPE html>
<html><head><title>Evento jQuery</title><meta charset="UTF-8">
<style> body{text-align:center;} </style>

<script type="text/javascript" src="jquery-2.1.4.min.js" > </script>
<script type="text/javascript">
  $(function(){
    var img = $('#i1');

    img.on('dblclick', function(){img.attr('src', 'lamp_off.jpg')});
    img.on('click',    function(){img.attr('src', 'lamp_on.jpg')});
  });
</script>
</head>
<body>
  <h1>Eventos</h1>
  <img src="lamp_off.jpg" id="i1">
</body>
</html>
```

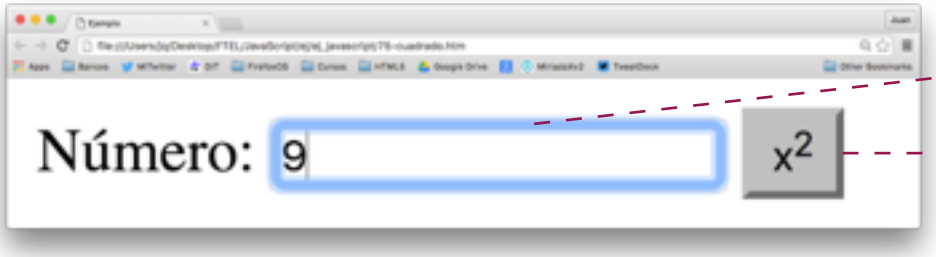
- jQuery también permite definir eventos en objetos jQuery con el método **on()**
  - **objetoJQuery.on(evento, manejador)**
    - jQuery conserva métodos asociados a eventos individuales de versiones anteriores, pero está recomendado usar solo **on()** y **off()**
      - http://api.jquery.com/category/events/
- Los **nombres y tipos de eventos** utilizados por los métodos **on(..)** y **off()**
  - son los mismos que los utilizados con el método addEventListener(..)
    - se pueden ver en https://developer.mozilla.org/en-US/docs/Web/Events
- El **manejador** del evento es una **función** (objeto de la clase Función) ejecutado al ocurrir el evento
  - Puede pasarse directamente como un literal de función con el código del manejador (como aquí)
    - o cualquier otro objeto Function, como por ejemplo el nombre de una función definida en otro lugar
      - OJO! debe ser el nombre de la función (su código) y no su invocación

Calculadora jQuery

Obtener objeto jQuery (DOM) deL cajetín: **$("#n1")**

Obtener objeto jQuery (DOM) del botón: **$("#b1")**

- jQuery simplifica la calculadora
- Modificaciones
  - Debemos **importar** la **librería jQuery**
  - Definir eventos en **función ready**
    - con **método on(..)**
      - con **árbol DOM ya construido**
  - **Obtener** objetos jQuery con **$("#...")**
  - **Obtener texto** de cajetín con **val()**
  - **Asignar texto** en cajetín con **val(texto)**




```
<!DOCTYPE html>
<html><head><title>Calculadora</title>
          <meta char set="utf-8">
  <script type="text/javascript"
          src="jquery-2.1.4.min.js">
  </script>

  <script type="text/javascript">
  $(function() {
    $("#n1").on("click",
      function(){ $("#n1").val("");}
    );

    $("#b1").on("click",
      function() {
        var num = $("#n1");
        num.val(num.val() * num.val());
      }
    );
  });
  </script>
</head>

<body>
  Número:
  <input type="text" id="n1">

  <button id="b1"> x<sup>2</sup> </button>
</body>
</html>
```

Importar librería jQuery

Evento click

Vaciar el cajetín

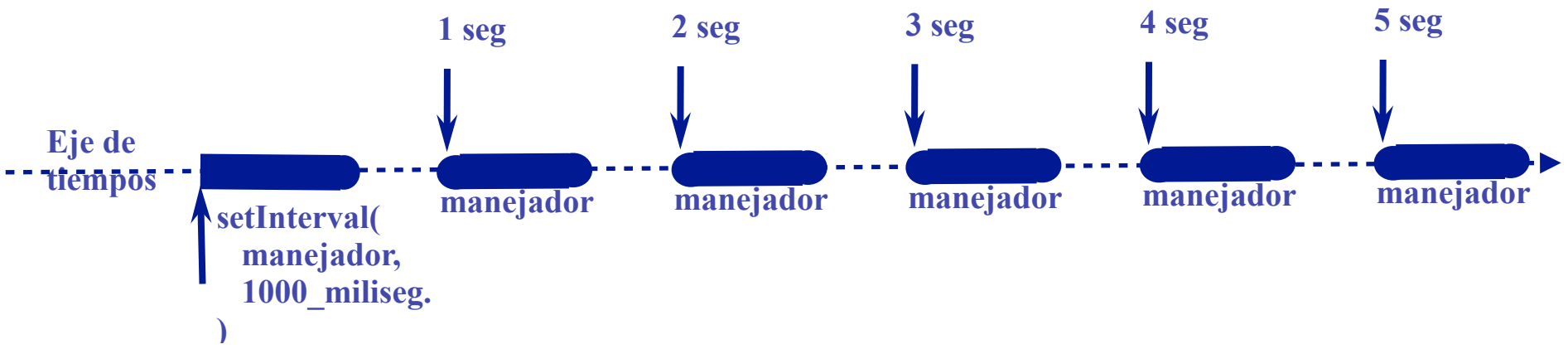
Evento click

Calcular resultado obteniendo el string tecleado en cajetín con **num.val()** y guardando el resultado con **num.val(..resultado..)**.

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



Reloj

```
<!DOCTYPE html>
<html><head><title>Reloj</title>
    <meta charset="UTF-8">
<script type="text/javascript"
        src="jquery-2.1.4.min.js" >
</script>

<script type="text/javascript">

  function mostrar_fecha( ) {
    $("#fecha").html(new Date( ));
  }

  $(function(){
      // Define evento periodico, ocurre
      // cada segundo (1000 miliseg)
    setInterval(mostrar_fecha, 1000);

          // muestra fecha al cargar
    mostrar_fecha();
  });
</script>
</head>
<body>

  <h2>Reloj</h2>

  <div id="fecha"><div>
</body>
</html>
```

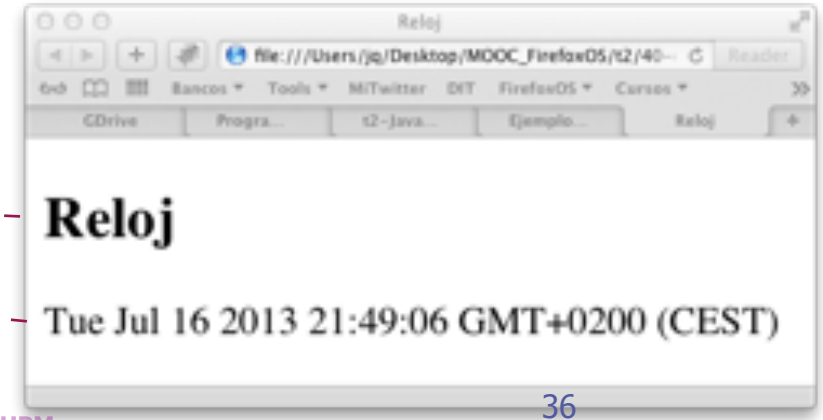
Importar librería jQuery

Mostrar fecha en bloque

Define un evento que actualiza la hora cada segundo

Muestra la hora al cargar la página Web

- El reloj utiliza un evento periódico
  - para actualizar cada segundo
    - la fecha y la hora mostrada en el bloque
- El evento periódico se programa con
  - **setInterval(..manejador.., ..periodo..)**
    - El manejador es una función
    - El periodo se da en milisegundos
      - con **árbol DOM ya construido**
  - **setInterval(mostrar_fecha, 1000)**
    - Ejecuta la función mostrar_fecha()
      - cada segundo (1000 milisegundos)
- Más información en
  - https://developer.mozilla.org/en-US/Add-ons/Code_snippets/Timers



© Juan Quemada, DIT, UPM

Final del tema

Muchas gracias!

© Juan Quemada, DIT, UPM