



# JavaScript en el navegador - Modulo 5:

Tipos primitivos, Objetos, Clases, Multi-asignación, Spread/Rest y Ejecución de JS

Juan Quemada, DIT - UPM  
(31-Enero-2020)

# Índice

## **MODULO 5 - Tipos primitivos, Objetos, Clases, Multi-asignación, Spread-Rest y Ejecución de JS**

1. Tipos primitivos, clases predefinidas, typeof y belongsto y métodos propios y heredados .....	3
2. Number: Literales de decimal, hex., oct. y bin., NaN, Infinity, módulo Math y clase Number .....	11
3. Arrays ES6: Asignación múltiple y spread/rest (...x) .....	14
4. Objetos ES6+: literal, multi-asignación y spread/rest (...x) .....	17
5. Referencias a objetos: comparación, compartición y clonacion de objetos .....	21
6. Clases en ES6 y Prototipos .....	27
7. Herencia de Clases en ES6 .....	33
8. Ejecución de programas: Eventos, bucle, cola, manejadores, timers, .. .....	39



# Tipos primitivos, clases predefinidas, typeof y belongsto y métodos propios y heredados

Juan Quemada, DIT - UPM

# Tipos primitivos y objetos

## ◆ Tipos primitivos

### ■ number

- ◆ Literales de números: **32**, **1000**, **3.8**



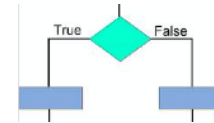
### ■ string

- ◆ Los literales de string son caracteres delimitados entre comillas o apóstrofes
  - "Hola, que tal", 'Hola, que tal',
- ◆ Internacionalización con Unicode: 'Γεια σου, ίσως', '嗨, 你好吗'



### ■ boolean

- ◆ Los literales son los valores **true** y **false**



### ■ symbol (nuevo en ES6)

- ◆ Representan un valor único diferente de cualquier otro y se crean con **Symbol()**



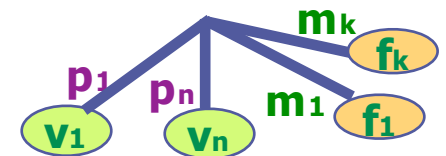
### ■ undefined

- ◆ **undefined**: valor único que representa algo no **definido** **UNDEFINED**

## ◆ Clase Object: un objeto es una agregación de propiedades y métodos

### ■ Se agrupan en **clases**: Object, Array, Date, **Function**, ...

- ◆ Objeto **null**: valor especial que representa objeto nulo



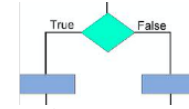



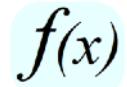


[https://developer.mozilla.org/en-US/docs/Web/JavaScript/A\\_re-introduction\\_to\\_JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript)

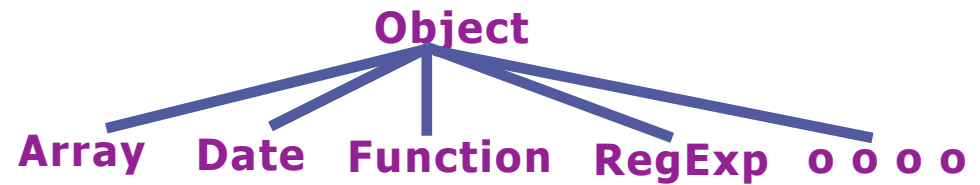
[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Values,\\_variables,\\_and\\_literals](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Values,_variables,_and_literals)

# Operador typeof

- ◆ El operador **typeof** permite conocer el tipo de un valor
  - Devuelve un **string** con el **nombre del tipo**
    - ◆ "number", "string", "boolean", "undefined", "object" y "function"
    - ◆ **Todos los objetos** (de cualquier clase) devuelven **"object"**, salvo las **funciones**

typeof 7	=> "number"	
typeof "hola"	=> "string"	
typeof true	=> "boolean"	
typeof Symbol()	=> "symbol"	
typeof undefined	=> "undefined"	
typeof {}	=> "object"	
typeof null	=> "object"	
typeof new Date()	=> "object"	
typeof function (){}	=> "function"	

# Clases predefinidas



◆ JavaScript tiene las siguientes **clases predefinidas**

◆ **Object**

- Clase raíz de la cual derivan todas las demás clases.

◆ **Array**

- Define colecciones ordenadas de valores.

◆ **Date**

- Define objetos con hora y fecha del reloj del sistema.

◆ **Function**

- Define código parametrizado.

◆ **RegExp**

- Define expresiones regulares para reconocer y procesar patrones de texto.

◆ **Error**

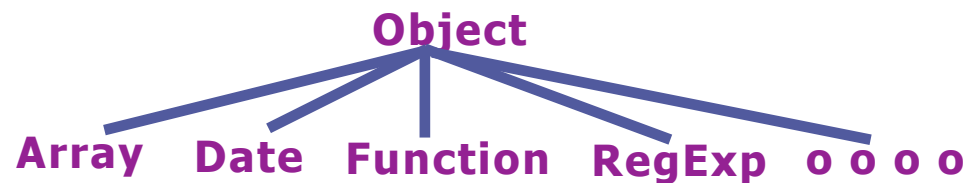
- Errors de ejecución lanzados por el interprete de JavaScript.

◆ **Number, String y Boolean**

- Encapsulan valores (1, 2, 3, ....., "hola", ....., true, ..) en objetos para poder aplicar métodos en ellos.

◆ **Promise, Map, Set, Typed Arrays, ...** nuevas clases introducidas por ES6+

# Jerarquía de clases, constructores y literales



◆ La clase **Object** es la raíz de la que **derivan** todas las **clases** de JavaScript

- Las demás clases predefinidas **extienden** la clase **Object**
  - ◆ Una clase **hereda** los **métodos** y **propiedades** de la clase que extiende y añade otros nuevos
- JavaScript permite definir **nuevas clases**, además de las clases predefinidas existentes

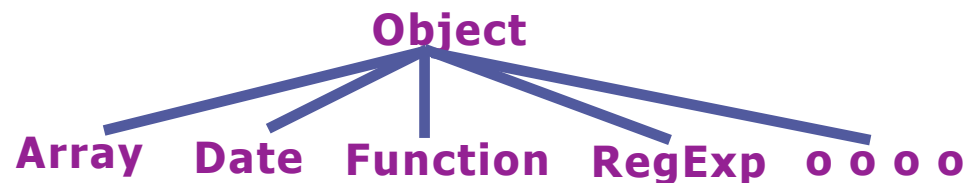
## ◆ Clase

- Una clase es el conjunto de objetos del mismo tipo creados con su **constructor**
  - ◆ El **constructor** tiene el mismo nombre de la clase y crea objetos con el operador **new**
    - Por ejemplo **new Object()**, **new Date ()**; **new Array()**, **new Function()**, ..
- Algunas clases predefinidas tienen además **literales** de gran eficacia
  - ◆ Literal de objetos: **{a:3, b:"que tal"}** o **{}** que crea un objeto vacío igual que **new Object()**
  - ◆ Literales de arrays: **[5, 2, 3]** o **[]** que crea un array vacío igual que **new Array()**
  - ◆ Literales de función: **function (x) {....}** o **(x) => {....}** el literal de funciones arrow de ES6
  - ◆ Literales de Regexp: **/(hola)+\$/**
  - ◆ Valores de los tipos primitivos **number**, **string** y **boolean** (1, 2, 3, ....., "hola", ....., true, ..) se traducen automáticamente a objetos de **Number**, **String** o **Boolean** al invocar métodos en dichos valores.

◆ Mas detalles sobre las clases predefinidas de JavaScript

- [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects)

# Herencia y operador instanceof



- ◆ Todas las **clases** de JavaScript **derivan** de la **clase Object** (la extienden)
  - Todos los **objetos** de JavaScript **pertenecen** a la **clase Object**
    - ◆ Los **objetos** de una clase **pertenecen** también a la **clase padre** de la que derivan
- ◆ El operador **instanceof** determina si un **valor** pertenece a una clase

<code>({}) instanceof Object</code>	<code>=&gt; true</code>	<code>// {} es un objeto aunque este vacío</code>
<code>({}) instanceof Array</code>	<code>=&gt; false</code>	<code>// {} no es un Array, pertenece solo a Object</code>
<code>[] instanceof Array</code>	<code>=&gt; true</code>	<code>// [] es un array aunque este vacío</code>
<code>[] instanceof Object</code>	<code>=&gt; true</code>	<code>// Array deriva de Object</code>
<code>(function(){}).</code>	<code>instanceof Function =&gt; true</code>	<code>// function(){} es una función vacía</code>
<code>(() =&gt; {})</code>	<code>instanceof Function =&gt; true</code>	<code>// ()=&gt;{} es una función vacía definida con =&gt;</code>
<code>""</code>	<code>instanceof String =&gt; false</code>	<code>// "" es un tipo primitivo (no son objetos)</code>
<code>new String("")</code>	<code>instanceof String =&gt; true</code>	<code>// new String("") pertenece a la clase String</code>



# Métodos heredados

- ◆ **Método:** función invocable sobre un objeto con el operador punto: "."
  - Por ejemplo, **new Date().toString()**
- ◆ Un objeto **hereda** las propiedades y métodos de su **clase**
  - Por ejemplo, los objetos de la **clase Date heredan** métodos como
    - ◆ **toString(), getDay(), getFullYear(), getHours(), getMinutes(), .....**
      - [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date)
      - <https://javascript.info/date>

```
var fecha = new Date(); // The object created contains hour-date of creation
```

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

```
fecha.toTimeString()  => 12:34:36 GMT+0200 (CEST)
```

```
fecha.getHours()      => 12
```

```
fecha.getMinutes()    => 34
```

```
fecha.getSeconds()    => 36
```

```
.....
```

# Métodos propios

## ◆ Una **función** asignada a una **propiedad** de un objeto, crea un **método**

- Este método, denominado **propio**, solo se puede invocar el **objeto** que lo contiene
  - ◆ Ese método no existe en los demás objetos de la clase y da error al invocarlo en ellos

## ◆ **this** es una referencia al objeto sobre el que se invoca un método

- En el ejemplo, **this.count** referencia la propiedad **count** del objeto sobre el que se invoque
  - ◆ **this** puede omitirse si no hay ambigüedad y en el ejemplo se podría utilizar **count**, en vez de **this.count**
    - <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this>

```
let person = {  
  name: "John",  
  age: 33,  
  show: function () { return `${this.name} is ${this.age} years old` }  
}
```

Método propio existente solo en el objeto **person**.

```
let person_2 = {  
  name: "Eve",  
  age: 26  
}
```

```
person.show(); // => "John is 33 years old"
```

Error: el método show no existe en el objeto **person\_2**.

```
person_2.show(); // => Uncaught TypeError: person_2.show is not a function
```



JavaScript

# Number:

Literales de dec., hex., oct. y bin., NaN, Infinity, mod. Math y clase Number

Juan Quemada, DIT - UPM

# El tipo number

## ◆ Literales de números

- **1, 32, ....** Enteros
- **1.2, 32.1, ....** Números decimales
- **3.2e1 (3,2x10<sup>1</sup>)** Coma flotante
- **0xFF, 0X10ff, ...** Enteros en hexadecimal
- **0b01101000, 0B1010, ...** Enteros en binario (ES6)
- **0o7123, 0O777, ...** Enteros en octal (ES6)

- **NaN (Not a Number)** strings y otros elementos no convertibles
- **Infinity y -Infinity** representa infinito o desbordamiento

## ◆ number: IEEE 754 coma flotante doble precisión (64 bits)

- Reales máximo y mínimo: **~1,797x10<sup>308</sup> y 5x10<sup>-324</sup>**
- Entero máximo: **9007199254740992**

Mas info: <https://javascript.info/number>,  
[https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First\\_steps/Math](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Math)

### // Operadores aritméticos

```
10 + 4  => 14    // suma
10 - 4  => 6     // resta
10 * 4  => 40    // multiplica
10 / 4  => 2.5   // divide
10 % 4  => 2     // resto
10 ** 2 => 100   // potencia (ES6)

3e2 + 1  => 301  // 3x102
3e-2 + 1 => 1,03 // 3x10-2
```

```
0xA + 4  => 14 // 0xA es 10
0x10 + 0o10 => 24 // 0x10 es 16
                // 0o10 es 8
0b1000 + 4  => 12 // 0b1000 es 8
```

```
// no representable
+'xx'  => NaN

// Infinito matemático
+10/0  => Infinity
-10/0  => -Infinity
```

```
// Desbordamiento
5e500  => Infinity
-5e500 => -Infinity

// Aproximación a cero
5e-500 => 0
```

# Funciones, métodos y módulos de number

## ◆ **Number(a)** (función de conversión a number)

```
Number('60')    => 60  
Number('1e2')   => 100  
Number('1.3e2') => 130
```

```
Number('01xx')  => NaN  
Number('5e500') => Infinity  
Number('5e-500') => 0
```

```
(111).toFixed(2)    => "111.00"  
(111).toPrecision(2) => "1.1e+2"  
Math.PI.toFixed(4)  => "3.1416"
```

## ◆ Métodos de la clase **Number**

- **toFixed(n)** devuelve string equivalente con n decimales
- **toPrecision(n)** devuelve string equivalente con n dígitos
- **toString([base])** convierte a string equivalente en base

- Doc: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Number](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number)

```
111.toFixed(2)      => Error  
  
(31).toString(2)    => "11111"  
(31).toString(10)   => "31"  
(31).toString(16)   => "1f"  
(10.75).toString(16) => "a.c"
```

## ◆ El módulo **Math** contiene

- Constantes matemáticas: **E**, **PI**, **SQRT2**, ...
- Funciones matemáticas: **sin(x)**, **cos(x)**, **asin(x)**, **log(x)**, **pow(x, y)**, **sqrt(x)**, **abs(x)**, **ceil(x)**, **floor(x)**, **round(x)**, **min(x,y,z,...)**, **max(x,y,z,...)**, **random()**, .....
- Más info: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Math](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math)

```
// numero aleatorio entre 0 y 1  
Math.random()  => 0.7890234  
Math.PI        => 3.141592653589793  
Math.pow(3,2)   => 9 // 3^2  
Math.sqrt(9)    => 3 // raíz cuadr.
```

```
Math.min(2,1,9,3)  => 1 // mínimo  
Math.max(2,1,9,3)  => 9 // máximo  
Math.sin(1)        => 0.8414709848078965  
Math.asin(0.8414709848078965) => 1
```

```
Math.floor(3.2)    => 3  
Math.ceil(3.2)     => 4  
Math.round(3.2)    => 3
```



# Arrays ES6:

## Asignación múltiple y spread/rest (...x)

Juan Quemada, DIT - UPM

# Asignación múltiple en arrays (ES6)

## ◆ La **asignación múltiple** asigna valores de un **array** a distintas **variables**

- Se puede utilizar en la **definición de variables** o en la **asignación**
  - ◆ Las variables deben agruparse entre corchetes y se relacionan con los valores por posición
- La asignación múltiple puede utilizar **valores por defecto**

## ◆ La asig. múltiple se denomina también **desestructuradora** (destructuring)

- Permite hacer programas más cortos y legibles

// Valores iniciales

```
let [x, y, z] = [5, 1, 3, 4];
```

x	=>	5
y	=>	1
z	=>	3

// Intercambiar contenidos

```
let x = 5, y = 1;
```

```
[x, y] = [y, x];
```

x	=>	1
y	=>	5

// Valores por defecto/indefinidos

```
let [x, y, z=1, t=2, v] = [5, , ,10]
```

x	=>	5
y	=>	undefined
z	=>	1
t	=>	10
v	=>	undefined

Buen tutorial sobre destructuring assignment y spread/rest: <https://javascript.info/destructuring-assignment>

Destructuring assignment: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring\\_assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)

Spread/rest syntax: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread\\_operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator)

# Operador spread/rest: ...x

## ◆ ES6 añade el operador spread/rest (...x)

- Tiene semántica spread (esparcir) o rest (resto) dependiendo del contexto

## ◆ Operador spread (...x) **esparce** los elementos de un **array** en otro

- Puede utilizarse en el constructor de array, al invocar una función, ...

## ◆ Operador rest (...x) agrupa el **resto** de valores en un **array**

- Agrupa en un array el resto de los elementos asignados de una lista
  - ♦ El operador rest debe ir al final y agrupa los últimos elementos de la lista

```
let a = [2, 3];  
let b = [0, 1, ...a];  
b => [0, 1, 2, 3]
```

```
f(0, ...a) => f(0, 2, 3)
```

```
let [x, ...rest1] = [0, 2, 3];  
x           => 0  
rest1      => [2, 3]  
  
[x, ...rest2] = [4, x, ...rest1];  
x           => 4  
rest2      => [0, 2, 3]
```

```
const f = (x, ...rest) => {  
  return rest.sort();  
}  
  
f(4, 3, 5, 0); => [0, 3, 5]
```

Buen tutorial sobre destructuring assignment y spread/rest: <https://javascript.info/destructuring-assignment>

Destructuring assignment: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring\\_assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)

Spread/rest syntax: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread\\_operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator)





# Objetos ES6+:

## Literal, multi-asignación y spread/rest (...x)

Juan Quemada, DIT - UPM  
Santiago Pavón, DIT - UPM

# El literal de objetos ES6+: agrupar variables

- ◆ El literal de objetos de ES5 agrupa variables en propiedades así
  - **var obj = {a:a, b:b, c:c}** agrupa las variables a, b y c en un objeto con propiedades de igual nombre (es muy frecuente en algunas aplicaciones)
- ◆ El literal de objetos de **ES6+** permite una sintaxis simplificada
  - **var obj = {a, b, c}** es equivalente en **ES6+** a lo anterior

```
let a=5, c=3, d=4;
// ES5: agrupar variables en un objeto con
let obj_ES5 = {a:a, c:c, d:d}; // propiedades de igual nombre a las variables
obj_ES5      => {a:5, c:3, d:4}

let obj_ES6 = {a, c, d};      // ES6: Las mismas variables se agrupan así
obj_ES6      => {a:5, c:3, d:4}
```

# Asignación múltiple o desestructuradora

## ◆ La multi-asignación de ES6+ se puede aplicar también a objetos

- En este caso asigna varias propiedades a variables del mismo nombre
  - ◆ En inglés se denomina 'destructuring', que se ha traducido por desestructuradora

## ◆ Variables y valores asignados se **relacionan por nombre**

- **Variables asignadas:** deben agruparse con llaves
  - ◆ Pueden usar valores por defecto

```
let {a, c=1, d, e} = {a:5, e:3, f:4};
```

a	=>	5
c	=>	1
d	=>	undefined
e	=>	3

```
let a, c, d;
```

```
({a, c=1, d} = {a:5, e:3});
```

a	=>	5
c	=>	1
d	=>	undefined

La multi-asignación debe ir aquí entre paréntesis por un problema de análisis sintáctico de JavaScript.

Buen tutorial sobre destructuring assignment y spread/rest: <https://javascript.info/destructuring-assignment>

Destructuring assignment: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring\\_assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)

Spread/rest syntax: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread\\_operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator)

# Operador rest/spread (...x) para objetos

◆ El operador **rest** (...x) de ES6+ puede combinarse con la asignación múltiple

- Por ejemplo `let {a, ...x} = {a:5, b:1, c:2}` o `({a, ...x} = {a:1, b:2})`

```
let {a, ...x} = {a:5, b:1, c:2};
```

<b>a</b>	=>	5
<b>x</b>	=>	{b:1, c:2}

```
let a, x;
```

```
({a, ...x} = {a:1, b:2});
```

<b>a</b>	=>	1
<b>x</b>	=>	{b:2}

La multi-asignación debe ir aquí entre paréntesis por un problema de análisis sintáctico de JavaScript.

◆ El operador **spread** (...x) de ES6 esparce propiedades en un objeto

- Por ejemplo `let x = {a:5, b:1}` y `let y = {...x, c:6, d:7}`

```
let x = {a:5, b:1};
```

```
let y = {...x, c:6, d:7};
```

<b>y</b>	=>	{a:5, b:1, c:6, d:7}
----------	----	----------------------

Buen tutorial sobre destructuring assignment y spread/rest: <https://javascript.info/destructuring-assignment>

Destructuring assignment: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring\\_assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)

Spread/rest syntax: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread\\_operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator)

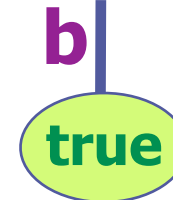


# Referencias a objetos:

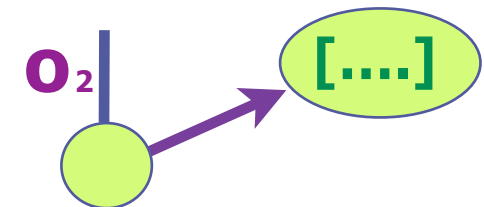
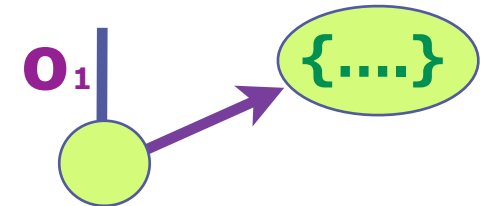
## comparación, compartición y clonación de objetos

Juan Quemada, DIT - UPM  
Santiago Pavón, DIT - UPM

# Valores y referencias



- ◆ Los tipos JavaScript se gestionan por valor o por referencia
  - Los tipos primitivos **number**, **string**, **boolean** o **undefined** se manejan por valor
  - Los **objetos** se manejan con una **referencia** (contenida en las variables)
    - ◆ Por ejemplo **Object**, **Array**, **Function**, **Date**, ...
- ◆ La **asignación** copia el contenido de la variable
  - En los tipos primitivos se copia el **valor**
  - En los objetos se copia solo la **referencia** (contenida en las variables)
    - ◆ Cuando queremos **copiar** el objeto debemos **clonarlo**.
- ◆ La **identidad** y la **igualdad** también se ven afectadas
  - En los tipos primitivos se comparan los **valores**
  - En los objetos se comparan las **referencias**



# Identidad e igualdad 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

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

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

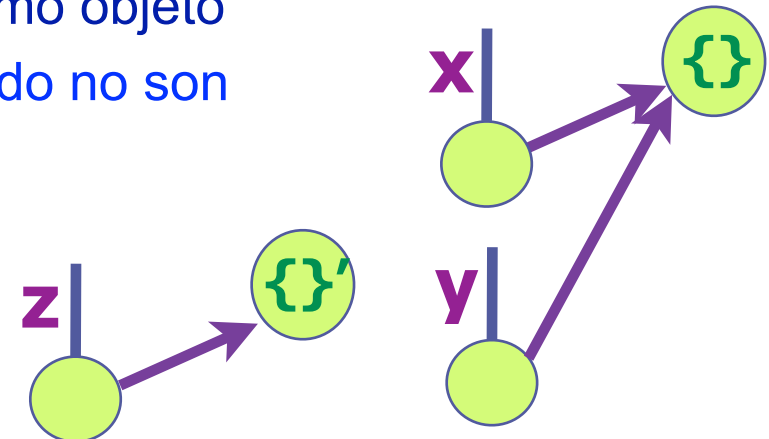
<code>x === y</code>	<code>=&gt; true</code>
<code>x === {}</code>	<code>=&gt; false</code>
<code>x === z</code>	<code>=&gt; false</code>

## ◆ La **identidad de objetos** indica que son el mismo objeto

- Dos **objetos** distintos con el mismo contenido no son idénticos

## ◆ Igualdad (débil) de objetos `==` y `!=`

- No tiene utilidad con objetos
  - ◆ Se recomienda no utilizarla

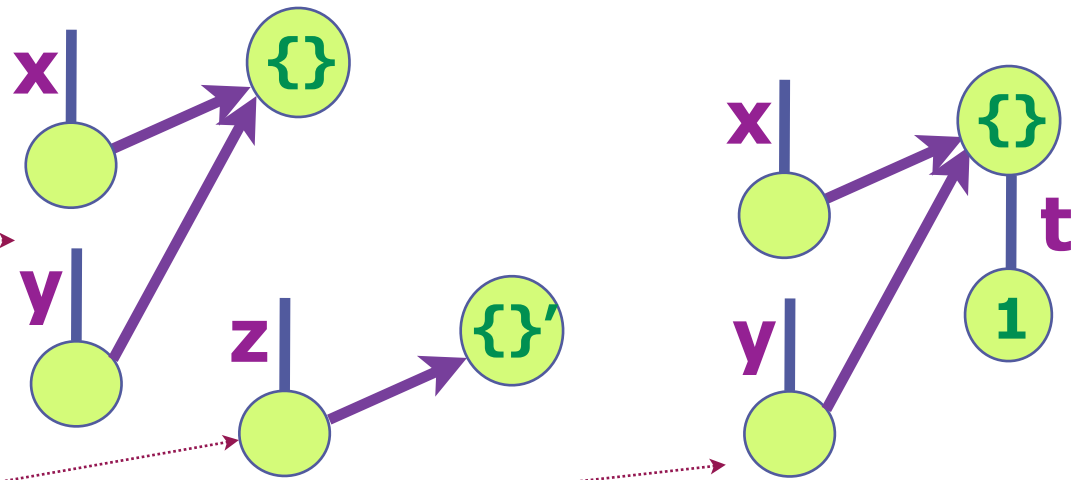


```
let x = {}; // x e y tienen la  
let y = x; // misma referencia
```

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

```
y.t = 1; // Añade la propiedad t a y
```

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



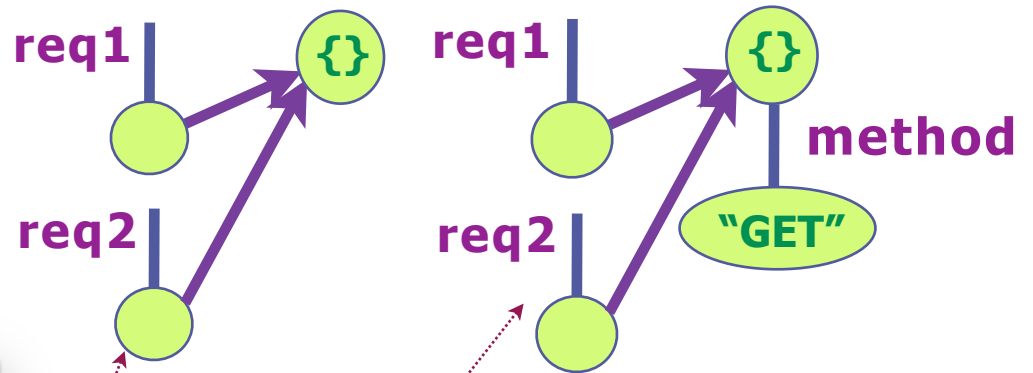
- ◆ Las variables que contienen objetos solo guardan la referencia al objeto
  - El objeto está alojado en un lugar de la memoria apuntado por la referencia
- ◆ Al asignar una variable se copia la referencia
  - si se modifica el objeto de una de ellas
    - ◆ Todas las variables que contengan la misma referencia verán el objeto modificado
- ◆ Los **parámetros y valores de retorno de funciones** de objetos también acceden por referencia y tienen el mismo efecto

## 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
```



- ◆ Parámetros de una función
  - Los tipos primitivos se pasan por valor
  - Los objetos se pasan por referencia
- ◆ Si la función modifica el objeto
  - esta modificación se verá a través de todas las referencias al objeto
    - ◆ Es decir, los **cambios** realizados **dentro** de la función, se ven **fuera** de la función

# Clonar objetos y arrays

```
var x = {a:1, b:2, c:[1, 2]};  
var y = ...x; // y es una "shallow copy" de x  
  
x === y;      => false // x e y tienen referencias diferentes  
x.a === y.a;  => true  // Se copia el valor (1)  
x.b === y.b;  => true  // Se copia el valor (2)  
x.c === y.c;  => true  // Se copia la referencia al array en c  
  
x.a = 5;      // Se modifica solo x.a y no y.a  
x.c[0]=5;     // Se modifican x.c[0] e y.c[0] (misma referencia)
```

## ◆ Clon de un objeto o array

- Copia del objeto (u array) (tiene las mismas propiedades que el original)
  - ◆ Al asignar un objeto a una variable **solo se copia la referencia** y no el objeto

## ◆ Copia superficial (shallow):

- Objeto con las mismas propiedades, que incluyen copias de los **valores** primitivos o copias de las **referencias** a objetos
  - ◆ Se suele realizar con **Object.assign(destino, origen)** o con spread **...obj**

## ◆ Copia profunda (deep):

- Copia completa que incluye todo el árbol de objetos (no se comparte nada)
  - ◆ Es **costosa** en recursos (recomendable usar librería **lodash** <https://lodash.com>)
    - ◆ Se puede utilizar **JSON.parse(JSON.stringify(obj))** si **obj** es representable en JSON



JavaScript

# Clases en ES6 y Prototipos

Sonsoles Lopez Pernas, DIT - UPM  
Juan Quemada Vives, DIT - UPM

# Clases

- Una clase es un modelo que define un conjunto de variables (atributos) y métodos apropiados para operar con dichos datos.
- Cada instancia que se crea de la clase es un objeto.

```
let hoy = new Date();
```

Para crear un objeto de una clase se utiliza la palabra clave `new`

```
hoy.getMinutes();
```

`getMinutes` es un método heredado de la clase `Date`

# Definir una clase en ES6

```
class User {
```

El constructor permite crear objetos de la clase

```
  constructor(name, age) {
```

```
    this.name = name;
```

```
    this.age = age;
```

`name` y `age` son los atributos de instancia que el constructor crea para cada objeto de la clase

```
  }
```

Una clase puede incluir métodos que heredarán todos las instancias de la clase

```
  show() {
```

```
    console.log(` ${this.name} is ${this.age} years old`);
```

```
  }
```

Para acceder a los atributos de cada objeto desde un método se utiliza la palabra clave `this`, aunque se puede omitir (igual a `` ${name} is ${age} years old``)

```
// Uso:
```

```
let user = new User("John", 33);
```

```
user.show();           // John is 33 years old
```

Para crear un objeto de la clase con el constructor se utiliza la palabra clave `new`

Los métodos heredados de la clase se invocan a partir del objeto creado

# Definir una clase utilizando prototipos

JavaScript simula las clases utilizando funciones, objetos y prototipos.

La clase anterior se podría haber definido así:

```
function User(name, age) {  
  this.name = name;  
  this.age = age;  
}
```

El constructor es una función que por convenio suele comenzar por mayúscula

```
let user = new User("John", 33);
```

Una instancia de la clase es un objeto JavaScript creado con el operador `new` y el constructor (los atributos son propiedades)

```
User.prototype.show = function() {  
  console.log(`${this.name} is ${this.age} years old`);  
}
```

El prototipo es un objeto JavaScript con los métodos heredados de la clase. Cualquier objeto tiene una propiedad `prototype` que da acceso al prototipo y permite añadir métodos heredados.

```
user.show(); // John is 33 years old
```

Cualquier instancia de la clase puede invocar el método añadido al prototipo

# Prototipos II

Los prototipos también permiten modificar clases ya existentes

```
String.prototype.words = function(){  
    return this.trim().split(" ")  
}
```

Extensión del  
prototipo String para  
añadir una nueva  
función que devuelve  
las palabras que  
forman una frase

```
let frase = " Buenos días";  
  
frase.words();    // ["Buenos", "días"]
```

Podemos utilizar la  
nueva función en  
cualquier String de  
nuestro programa

# Observaciones

Tanto si utilizamos la sintaxis de clase como la de prototipo, las clases JavaScript son funciones y sus instancias, objetos:

```
typeof User    // "function"
```

```
let user = new User("John", 33)
```

```
typeof user    // "object"
```

Si queremos comprobar si un objeto es instancia de una clase, utilizamos `instanceof`

```
user instanceof User    // true
```





# Herencia de Clases en ES6

Sonsoles Lopez Pernas, DIT - UPM  
Juan Quemada Vives, DIT - UPM

# Herencia

- Cuando una clase **X** hereda de otra clase **Y** quiere decir que la clase hija **X** dispone de los mismos métodos y atributos que la clase padre **Y**, además de los atributos y métodos nuevos de **X**.
- Para implementar la herencia utilizamos la palabra clave **extends**

```
class X extends Y {}
```

# Super

- La palabra clave **super** se usa para llamar funciones de la clase padre
- Se suele emplear cuando queremos mantener la funcionalidad del método heredado y añadir lógica adicional en la clase hija

# Ejemplo de herencia

```
class Worker extends User {  
  constructor(name, age) {  
    super(name, age);  
  }  
  isRetired() {  
    return this.age >= 65;  
  }  
}
```

```
let alice = new Worker("Alice", 67);  
alice.show();           // Alice is 67 years old  
alice.isRetired();      // true
```

```
let bob = new User("Bob", 18);  
bob.show();             // Bob is 18 years old  
bob.isRetired();        // Uncaught TypeError: bob.isRetired is not a function
```

# Sobrescribir métodos

- A veces queremos que la clase hija tenga un comportamiento distinto a la clase padre al llamar a un método heredado
- Para ello podemos redefinir el método heredado en la clase hija

# Ejemplo de sobreescritura de métodos

```
class Employee extends Worker{  
  constructor(name, age, earlyRetirement) {  
    super(name, age);  
    this.earlyRetirement = earlyRetirement;  
  }  
  isRetired() {  
    if (this.earlyRetirement) {  
      return true;  
    } else {  
      return super.isRetired();  
    }  
  }  
}
```

```
let chris = new Employee("Chris", 43, true);  
chris.show();      // Chris is 43 years old  
chris.isRetired(); // true
```



# Ejecución de programas:

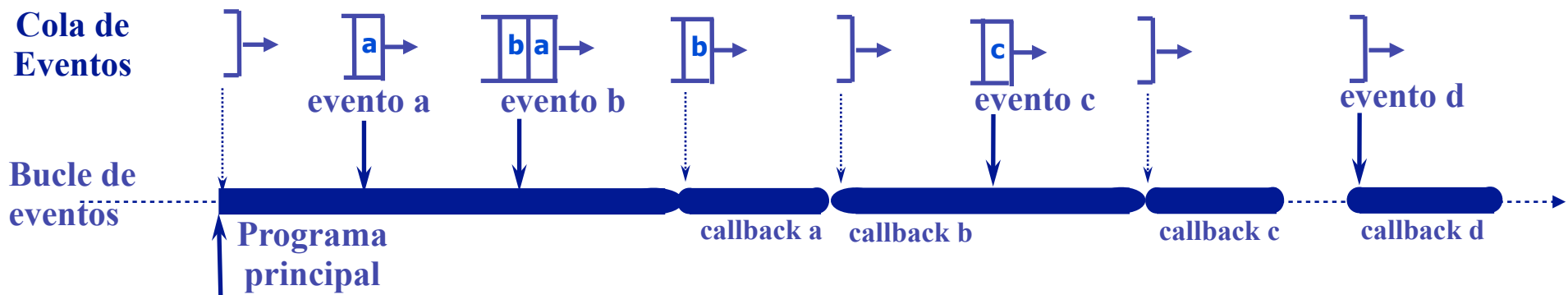
Eventos, bucle, cola, manejadores, timers, ..

Juan Quemada, DIT - UPM

# El bucle y la cola de eventos



- ♦ JavaScript se ejecuta en un **hilo (thread)** del proceso asignado al navegador
  - Primero se ejecuta el **programa principal** (scripts de la página)
    - Después se **atienden los eventos** ejecutando sus **manejadores** (o callbacks)
- ♦ La **cola de eventos** guarda los eventos pendientes de ejecutar
  - Al **finalizar** el programa en **ejecución**, se **atiende al primer evento** de la cola
    - Los nuevos eventos **se guardan** en la **cola**, si se esta **atendiendo otro**
- ♦ JavaScript se queda en espera pasiva si no hay eventos que atender
  - Los recursos del procesador solo se consumen cuando se atiende a eventos





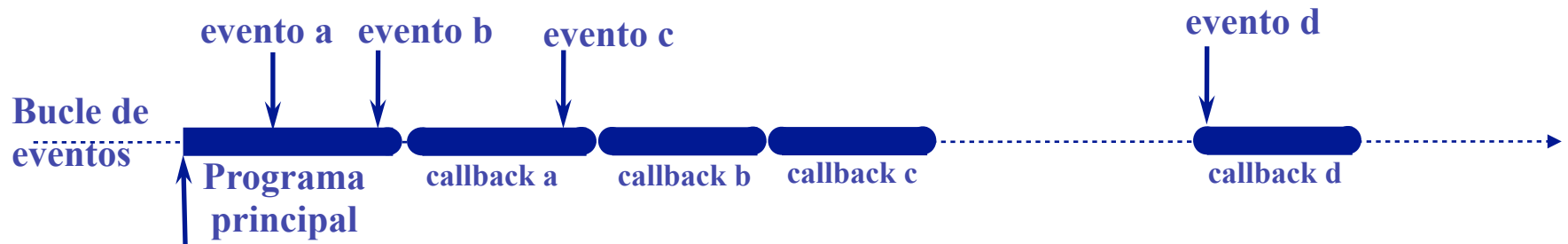
# JavaScript garantiza exclusion mutua

## ♦ La ejecución de JavaScript es sencilla de entender

- Los eventos se introducen al final de la cola de eventos y se ejecutan en serie empezando por el mas antiguo

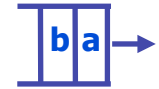
## ♦ La gestión de la cola de eventos

- garantiza exclusión mutua en el acceso a variables y objetos
  - No se necesitan mecanismos de exclusión mutua: zonas críticas, monitores, ...



# Bloqueo en JavaScript

Cola de Eventos

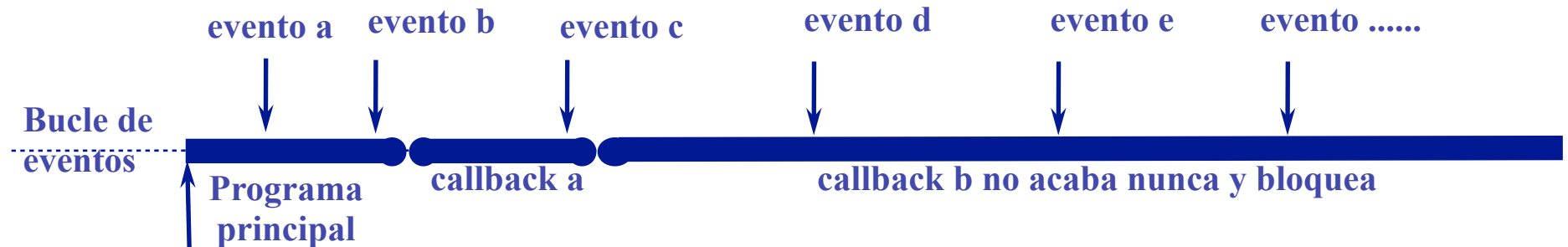


## ◆ Bloqueo

- **Problema importante** de la programación concurrente
  - Un programa, o parte de él, **deja de ejecutarse**, esperando que otro acabe

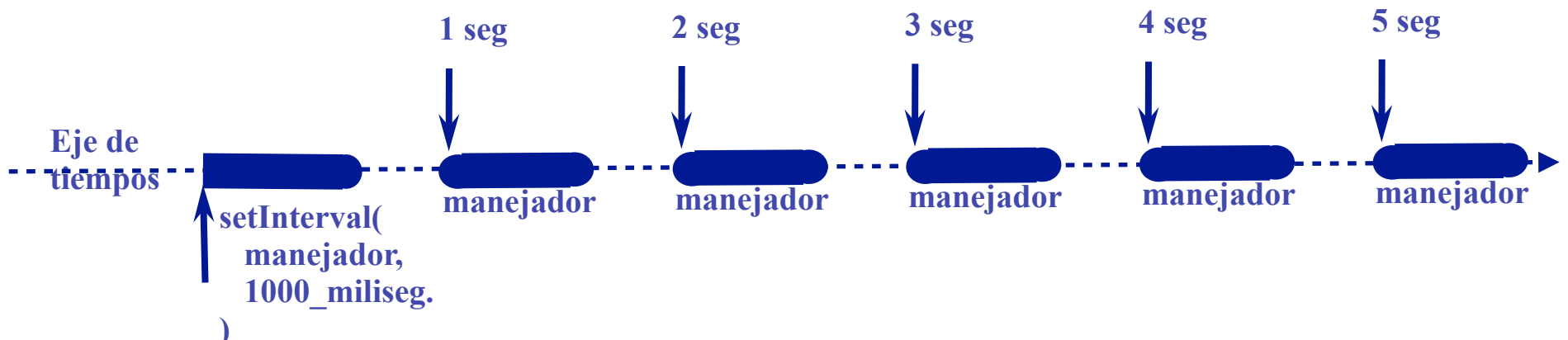
## ◆ Programa principal y manejadores (o "callbacks")

- pueden bloquear al resto **solo por inanición ("starvation")**
  - Si un manejador **no finaliza**, **no se atienden** mas eventos y el servidor **se bloquea**
- Un manejador debe **finalizar lo mas rápidamente posible**
  - Así se garantiza una pronta atención a los siguientes eventos



# Eventos periódicos con setInterval(...)

- ◆ JavaScript tiene funciones para programar eventos temporizados
  - **setTimeout (..)**: programa un único evento temporizado
  - **setInterval (..)**: programa eventos internos periódicos
    - ◆ Mas info: <https://javascript.info/settimeout-setinterval>
- ◆ **setInterval (manejador, periodo\_en\_milisegundos)**
  - tiene 2 parámetros
    - ◆ **manejador o "callback"**: función ejecutada al ocurrir el evento
    - ◆ **periodo\_en\_milisegundos**: tiempo entre eventos periódicos



# Reloj

- ◆ El reloj utiliza un evento periódico:
  - cada segundo actualiza fecha y hora
    - ◆ La muestra en un bloque <div>
- ◆ Evento periódico: se programa con
  - **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](https://developer.mozilla.org/en-US/Add-ons/Code_snippets/Timers)

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

```
<script type="text/javascript">
```

```
const mostrar_fecha = ( ) =>
  document
    .getElementById("fecha")
    .innerHTML = new Date( );
```

Mostrar  
fecha en  
bloque  
<div>

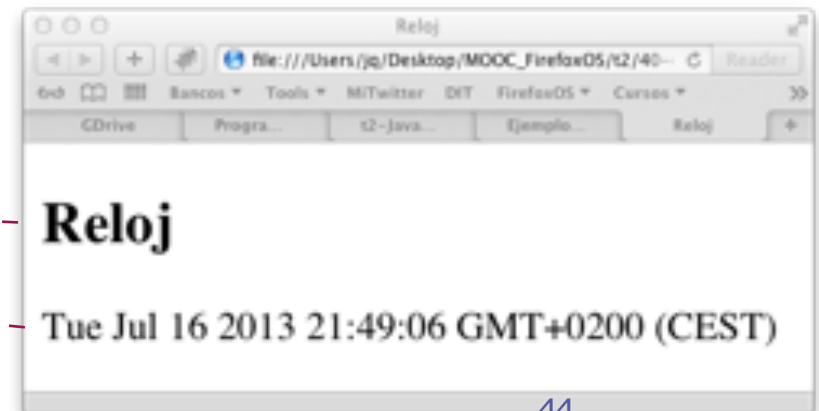
```
document.addEventListener(
  'DOMContentLoaded',
  ( ) => {
    setInterval(mostrar_fecha, 1000);
    mostrar_fecha();
  }
)
```

Define un  
evento que  
actualiza la  
hora cada  
segundo

```
</script>
</head>
<body>
```

Muestra la hora al  
cargar la página Web

```
<h2>Reloj</h2>
<div id="fecha"><div>
</div>
</html>
```





JavaScript

# Final