



# Introducción a JavaScript, a sus tipos y valores

# JavaScript



## ◆ JavaScript

- Diseñado por Netscape en 1995 para ejecutar en un Navegador
  - ◆ Hoy se ha convertido en el **lenguaje del Web y Internet**

## ◆ Norma ECMA (European Computer Manufacturers Association)

- Versión actual: **ES5: ECMAScript 5**, Dic. 2009, (JavaScript 1.5)
- Versión en desarrollo: **ES6: ECMAScript 6**, (JavaScript 1.6)
  - ◆ [https://developer.mozilla.org/en-US/docs/Web/JavaScript/New\\_in\\_JavaScript/ECMAScript\\_6\\_support\\_in\\_Mozilla](https://developer.mozilla.org/en-US/docs/Web/JavaScript/New_in_JavaScript/ECMAScript_6_support_in_Mozilla)

◆ Guía: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide>

◆ Referencia: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>

◆ Repositorio MDN: <https://developer.mozilla.org/en-US/docs/Web/JavaScript>

◆ Libro: “*JavaScript Pocket Reference*”, D. Flanagan, O’Reilly 2012, 3rd Ed.

◆ Libro: “*JavaScript: The Good Parts*”, D. Crockford, O’Reilly 2008, 1st Ed.



# Tipos, objetos y valores

## ◆ Tipos de JavaScript

### ■ **number**

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



### ■ **boolean**

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

**FALSE**  
**true**

### ■ **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**

### ■ **undefined**

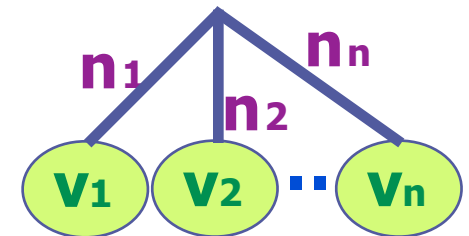
- ◆ **undefined**: representa **indefinido**

UNDEFINED



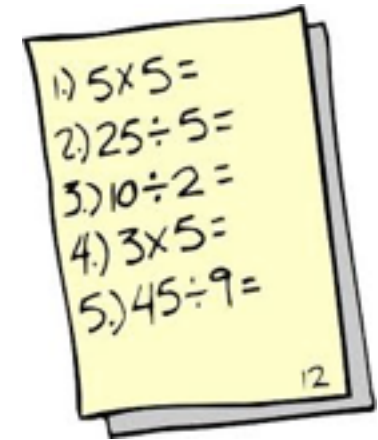
## ◆ **Objetos**: agregaciones estructuradas de valores

- Se agrupan en **clases**: **Object**, **Array**, **Date**, ...
  - ◆ Objeto **null**: valor especial que representa objeto nulo



[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)

# Operadores y expresiones



- ◆ JavaScript incluye **operadores** de tipos y objetos
  - Los **operadores** permiten formar **expresiones**
    - ◆ componiendo **valores** con dichos operadores
  - Una expresión representa un valor, que es el resultado de evaluarla
- ◆ Por ejemplo, las operaciones aritméticas +, -, \*, / permiten formar las expresiones numéricas del ejemplo, que se evalúan a los valores indicados
  - Expresiones con sintaxis errónea abortan la ejecución del programa

13 + 7	=>	20	// Suma de números
13 - 7	=>	6	// Resta de números
(8*2 - 4)/3	=>	4	// Expresión con paréntesis
8 / * 3	=>	Error_de_ejecución	// Ejecución se interrumpe
8 3	=>	Error_de_ejecución	// Ejecución se interrumpe

# Sobrecarga de operadores

- ◆ Los operadores sobrecargados tienen varias semánticas
  - dependiendo del contexto en que se usan en una expresión
- ◆ Por ejemplo, el operador **+** tiene 3 semánticas diferentes
  - **Suma de enteros** (operador binario)
  - **Signo de un número** (operador unitario)
  - **Concatenación de strings** (operador binario)



**13 + 7                      =>    20                      // Suma de números**

**+13                          =>    13                      // Signo de un número**

**"Hola " + "Pepe"   =>   "Hola Pepe"   // Concatenación de strings**



# Conversión de tipos en expresiones

- ◆ JavaScript realiza conversión automática de tipos

- cuando hay ambigüedad en una expresión
  - ◆ utiliza las prioridades para resolver la ambigüedad

- ◆ La expresión **"13" + 7** es ambigua

- porque combina un **string** con un **number**
  - ◆ JavaScript asigna mas prioridad al **operador +** de strings, convirtiendo **7** a string

- ◆ La expresión **+"13"** también necesita conversión automática de tipos

- El **operador +** solo esta definido para **number**
  - ◆ JavaScript debe convertir el **string "13"** a **number** antes de aplicar operador **+**

13 + 7	=>	20
"13" + "7"	=>	"137"
"13" + 7	=>	"137"
+"13" + 7	=>	20

Los operadores están ordenados con prioridad descendente. Mas altos más prioridad.

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions\\_and\\_Operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators)

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator\\_Precedence](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence)

<b>.</b>	<b>Acceso a propiedad o invocar método; índice a array</b>
<b>new</b>	<b>Crear objeto con constructor de clase</b>
<b>()</b>	<b>Invocación de función/método o agrupar expresión</b>
<b>++ --</b>	<b>Pre o post auto-incremento; pre o post auto-decremento</b>
<b>! ~</b>	<b>Negación lógica (NOT); complemento de bits</b>
<b>+ -</b>	<b>Operador unitario, números. signo positivo; signo negativo</b>
<b>delete</b>	<b>Borrar propiedad de un objeto</b>
<b>typeof void</b>	<b>Devolver tipo; valor indefinido</b>
<b>* / %</b>	<b>Números. Multiplicación; división; modulo (o resto)</b>
<b>+</b>	<b>Concatenación de string</b>
<b>+ -</b>	<b>Números. Suma; resta</b>
<b>&lt;&lt; &gt;&gt; &gt;&gt;&gt;</b>	<b>Desplazamientos de bit</b>
<b>&lt; &lt;= &gt; &gt;=</b>	<b>Menor; menor o igual; mayor; mayor o igual</b>
<b>instanceof in</b>	<b>¿objeto pertenece a clase?; ¿propiedad pertenece a objeto?</b>
<b>== != === !==</b>	<b>Igualdad; desigualdad; identidad; no identidad</b>
<b>&amp;</b>	<b>Operacion y (AND) de bits</b>
<b>^</b>	<b>Operacion ó exclusivo (XOR) de bits</b>
<b> </b>	<b>Operacion ó (OR) de bits</b>
<b>&amp;&amp;</b>	<b>Operación lógica y (AND)</b>
<b>  </b>	<b>Operación lógica o (OR)</b>
<b>?:</b>	<b>Asignación condicional</b>
<b>=</b>	<b>Asignación de valor</b>
<b>OP=</b>	<b>Asig. con operación: += -= *= /= %= &lt;&lt;= &gt;&gt;= &gt;&gt;&gt;= &amp;= ^=  =</b>
<b>,</b>	<b>Evaluación múltiple</b>

## Operadores JavaScript

**+"3" + 7 => 10**

“+” unitario tiene mas prioridad y se evalúa antes que “+” binario

Los operadores están ordenados con prioridad descendente. Mas altos más prioridad.

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions\\_and\\_Operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators)

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator\\_Precedence](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence)

<b>.</b>	<b>Acceso a propiedad o invocar método; índice a array</b>
<b>new</b>	<b>Crear objeto con constructor de clase</b>
<b>()</b>	<b>Invocación de función/método o agrupar expresión</b>
<b>++ --</b>	<b>Pre o post auto-incremento; pre o post auto-decremento</b>
<b>! ~</b>	<b>Negación lógica (NOT); complemento de bits</b>
<b>+ -</b>	<b>Operador unitario, números. signo positivo; signo negativo</b>
<b>delete</b>	<b>Borrar propiedad de un objeto</b>
<b>typeof void</b>	<b>Devolver tipo; valor indefinido</b>
<b>* / %</b>	<b>Números. Multiplicación; división; modulo (o resto)</b>
<b>+</b>	<b>Concatenación de string</b>
<b>+</b>	<b>Números. Suma; resta</b>
<b>&lt;&lt; &gt;&gt; &gt;&gt;&gt;</b>	<b>Desplazamientos de bit</b>
<b>&lt; &lt;= &gt; &gt;=</b>	<b>Menor; menor o igual; mayor; mayor o igual</b>
<b>instanceof in</b>	<b>¿objeto pertenece a clase?; ¿propiedad pertenece a objeto?</b>
<b>== != === !==</b>	<b>Igualdad; desigualdad; identidad; no identidad</b>
<b>&amp;</b>	<b>Operacion y (AND) de bits</b>
<b>^</b>	<b>Operacion ó exclusivo (XOR) de bits</b>
<b> </b>	<b>Operacion ó (OR) de bits</b>
<b>&amp;&amp;</b>	<b>Operación lógica y (AND)</b>
<b>  </b>	<b>Operación lógica o (OR)</b>
<b>?:</b>	<b>Asignación condicional</b>
<b>=</b>	<b>Asignación de valor</b>
<b>OP=</b>	<b>Asig. con operación: += -= *= /= %= &lt;&lt;= &gt;&gt;= &gt;&gt;&gt;= &amp;= ^=  =</b>
<b>,</b>	<b>Evaluación múltiple</b>

## Operadores JavaScript

**8\*2 - 4 => 12**

“\*” tiene mas prioridad y se evalúa antes que “-”. Es equivalente a:

**(8\*2) - 4 => 12**



Los operadores están ordenados con prioridad descendente. Mas altos más prioridad.

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions\\_and\\_Operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators)

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator\\_Precedence](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence)

<b>.</b>	<b>Acceso a propiedad o invocar método; índice a array</b>
<b>new</b>	<b>Crear objeto con constructor de clase</b>
<b>()</b>	<b>Invocación de función/método o agrupar expresión</b>
<b>++ --</b>	<b>Pre o post auto-incremento; pre o post auto-decremento</b>
<b>! ~</b>	<b>Negación lógica (NOT); complemento de bits</b>
<b>+ -</b>	<b>Operador unitario, números. signo positivo; signo negativo</b>
<b>delete</b>	<b>Borrar propiedad de un objeto</b>
<b>typeof void</b>	<b>Devolver tipo; valor indefinido</b>
<b>* / %</b>	<b>Números. Multiplicación; división; modulo (o resto)</b>
<b>+</b>	<b>Concatenación de string</b>
<b>+</b>	<b>Números. Suma; resta</b>
<b>&lt;&lt; &gt;&gt; &gt;&gt;&gt;</b>	<b>Desplazamientos de bit</b>
<b>&lt; &lt;= &gt; &gt;=</b>	<b>Menor; menor o igual; mayor; mayor o igual</b>
<b>instanceof in</b>	<b>¿objeto pertenece a clase?; ¿propiedad pertenece a objeto?</b>
<b>== != === !==</b>	<b>Igualdad; desigualdad; identidad; no identidad</b>
<b>&amp;</b>	<b>Operacion y (AND) de bits</b>
<b>^</b>	<b>Operacion ó exclusivo (XOR) de bits</b>
<b> </b>	<b>Operacion ó (OR) de bits</b>
<b>&amp;&amp;</b>	<b>Operación lógica y (AND)</b>
<b>  </b>	<b>Operación lógica o (OR)</b>
<b>?:</b>	<b>Asignación condicional</b>
<b>=</b>	<b>Asignación de valor</b>
<b>OP=</b>	<b>Asig. con operación: += -= *= /= %= &lt;&lt;= &gt;&gt;= &gt;&gt;&gt;= &amp;= ^=  =</b>
<b>,</b>	<b>Evaluación múltiple</b>

## Operadores JavaScript

**8\*(2 - 4) => -16**

El paréntesis tiene más prioridad y obliga a evaluar primero “-” y luego “\*”

# 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"

**typeof 7**

=> "number"



**typeof "hola"**

=> "string"



**typeof true**

=> "boolean"

~~FALSE~~  
true

**typeof undefined**

=> "undefined"

UNDEFINED

**typeof null**

=> "object"

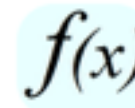
**typeof new Date()**

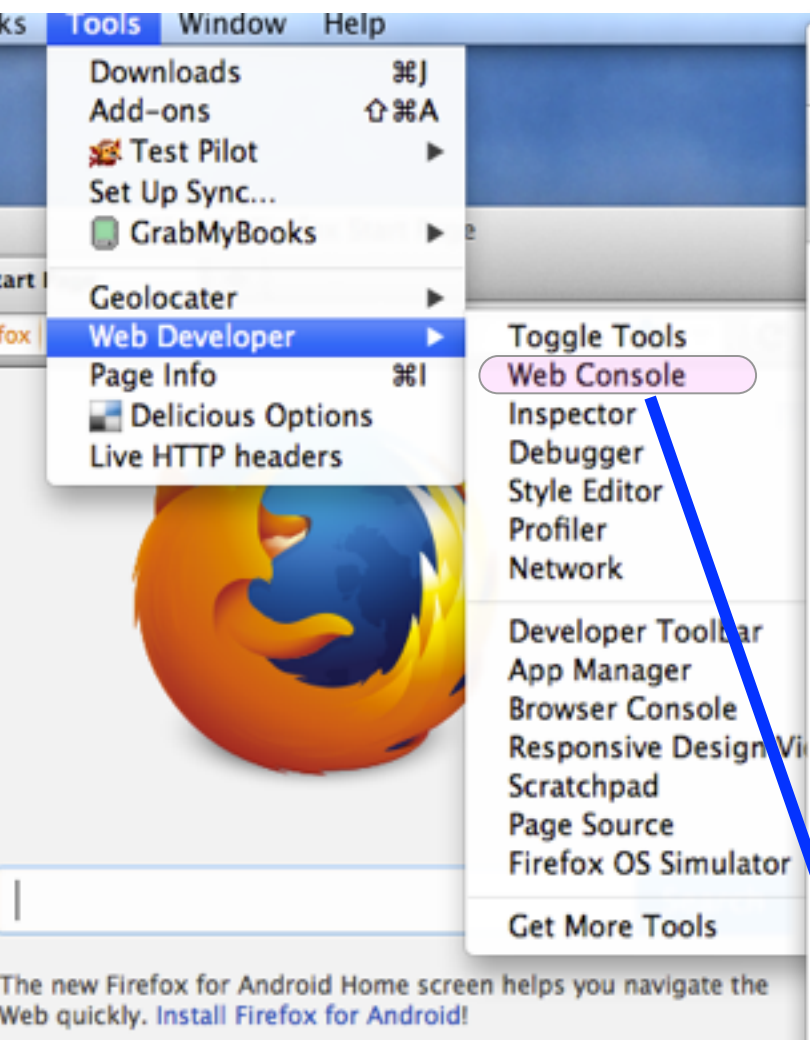
=> "object"



**typeof new Function()**

=> "function"





Number - JavaScript | MDN x Miriada X: Desarrollo en ... x

https://www.miriadax.net/web/html5mooc/edicion?p\_p\_id=e

Bancos Varios Cursos Tools Rails HTML5

✎ X ⬇ ⬆

- Modulo 1: Introducción a Internet, la nube, la arquitectura de la Web, HTML5 y CSS
- Modulo 2: Introducción a JavaScript y a las aplicaciones Web en HTML5, así como la publicación en la nube

✎ X ⬇ ⬆

Añadir

Transparencias y ejemplos del modulo

✎ X ⬇ ⬆

Tema 2.1: Tipos y valores de JavaScript

✎ X ⬇ ⬆

typeof ("10"+23)

☐ 10

☐ 23

☐ 33

☐ "23"

☐ "1023"

☐ "1033"

☐ "number"

☐ "string"

☐ "boolean"

typeof (10+23)

Net CSS JS Security Logging Clear

typeof ("10"+23)

"string"

>> typeof ("10"+23)

Juan Quemada, DIT, UPM

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

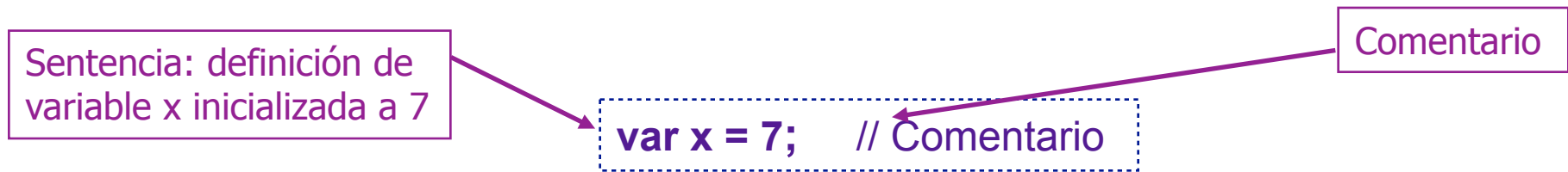
Aquí se introduce la sentencia



# Programa, sentencia, variable y comentario

# Programa, sentencias y comentarios

- ◆ Un programa es una secuencia de sentencias
  - que se ejecutan en el orden en que han sido definidas
- ◆ Las sentencias realizan tareas al ejecutarse en un ordenador
  - Son los elementos activos de un programa
- ◆ Los comentarios solo tienen valor informativo
  - para ayudar a entender o recordar como funciona el programa



# Comentarios

## ◆ Los comentarios son mensajes informativos

- Deben ser claros, concisos y explicar todo lo importante del programa
  - ◆ Incluso el propio autor los necesita con el tiempo para recordar detalles del programa

## ◆ En JavaScript hay 2 tipos de comentarios

- Monolínea: Delimitados por `//` y **final de línea**
- Multilínea: Delimitados por `/*` y `*/`
  - ◆ **OJO!** Los comentarios multi-línea tienen problemas con las expresiones regulares

`/* Ejemplo de comentario multilínea que ocupa 2 líneas  
-> al tener ambigüedades, se recomienda utilizarlos con cuidado */`

`var x = 1; // Comentario monolínea que acaba al final de esta línea`

# Variables y estado del programa

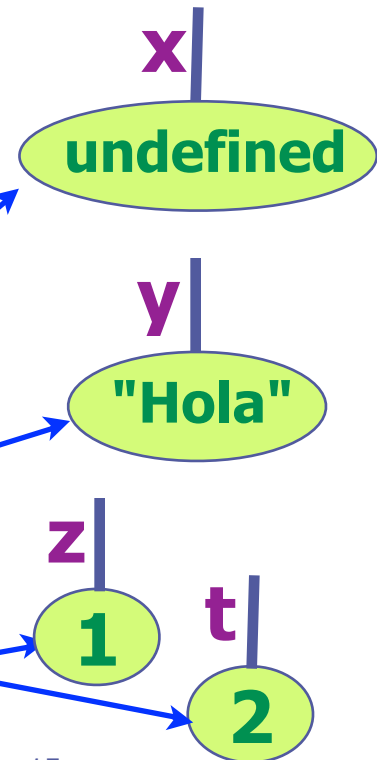
## ◆ Las variables se crean con la sentencia de definición de variables

- Comienza con la palabra reservada **var**
  - ◆ Seguida de la variable, a la que se puede asignar un valor inicial
- Se pueden crear varias variables en una sentencia
  - ◆ separando las definiciones por comas

## ◆ Estado del programa

- conjunto de valores contenido en todas sus variables

```
var x;           // crea la variable x y asigna undefined  
  
var y = "Hola";  // crea y, asignandole el valor "Hola"  
  
var z = 1, t = 2; // crea x e y, asignandosles 1 y 2 respectivamente
```



# Sentencia de asignación de variables

- ◆ Una variable es un contenedor de valores
  - La sentencia de asignación introduce un nuevo valor en la variable
    - ◆ Modificando, por tanto, el estado del programa
- ◆ Las variables de JavaScript son **no tipadas**
  - pueden contener **valores de cualquier tipo** de JavaScript

`var x = 5;` // Crea la variable x y le asigna el valor inicial 5

`x = "Hola";` // Asigna el string (texto) "hola" a la variable x

`x = new Date();` // Asigna objeto Date a la variable x

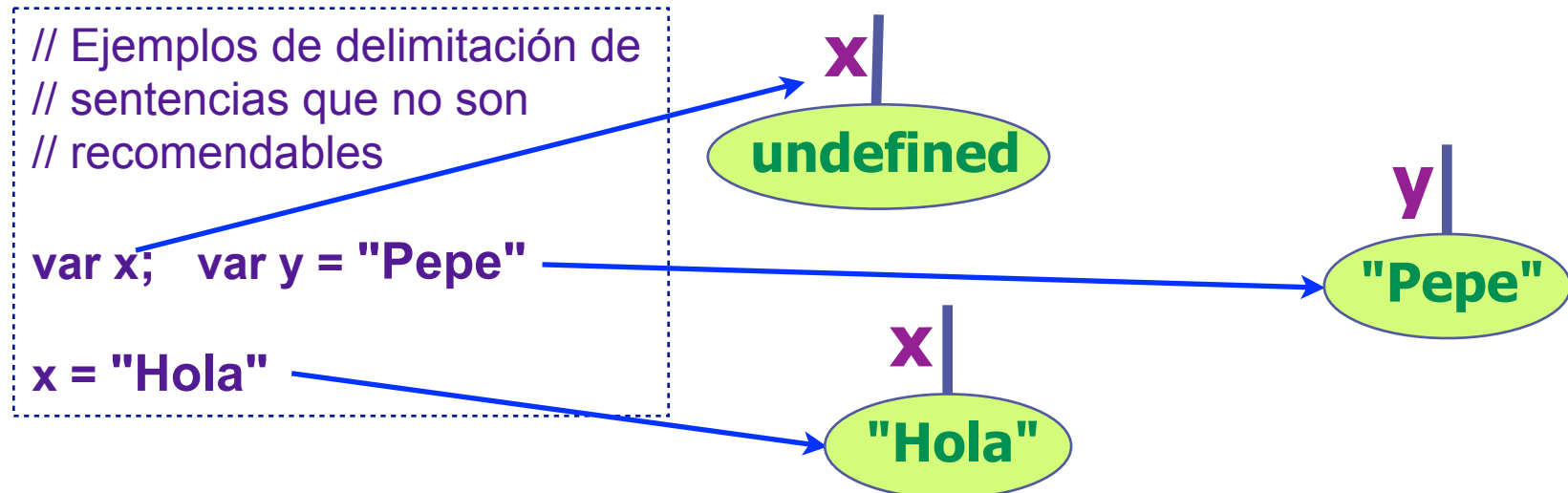




SENTENCIA	SINTAXIS	DESCRIPCIÓN DE LA SENTENCIA JAVASCRIPT
block	{ statements };	Agrupar un bloque de sentencias como 1 sentencia
break	break [label];	Salir del bucle o switch o sentencia etiquetada
case	case expression:	Etiquetar sentencia dentro de sentencia switch
continue	continue [label];	Salto a sig. iteración de bucle actual/etiquetado
debugger	debugger;	Punto de parada (breakpoint) del depurador
default	default:	Etiquetar sentencia default en 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 (incluye asignación a variable)
for	for(init; test; incr) statement	Bucle sencillo. "init": inicialización; "test": condición; "incr": acciones final bucle
for/in	for (var in object) statement	Enumerar 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 inicializar una o mas variables
while	while (expression) statement	Bucle básico con condición al principio
with	with (object) statement	Extender cadena de ámbito (no recomendado)

# Delimitación de sentencias

- ◆ “;” delimita el final de una sentencia
- ◆ El final de sentencia también puede delimitarse con nueva línea
  - Pero hay ambigüedades y no se recomienda hacerlo
- ◆ **Recomendación:** cada sentencia **ocupa una línea y termina con “;”**  
-> es mas legible y seguro



# Nombres de variables

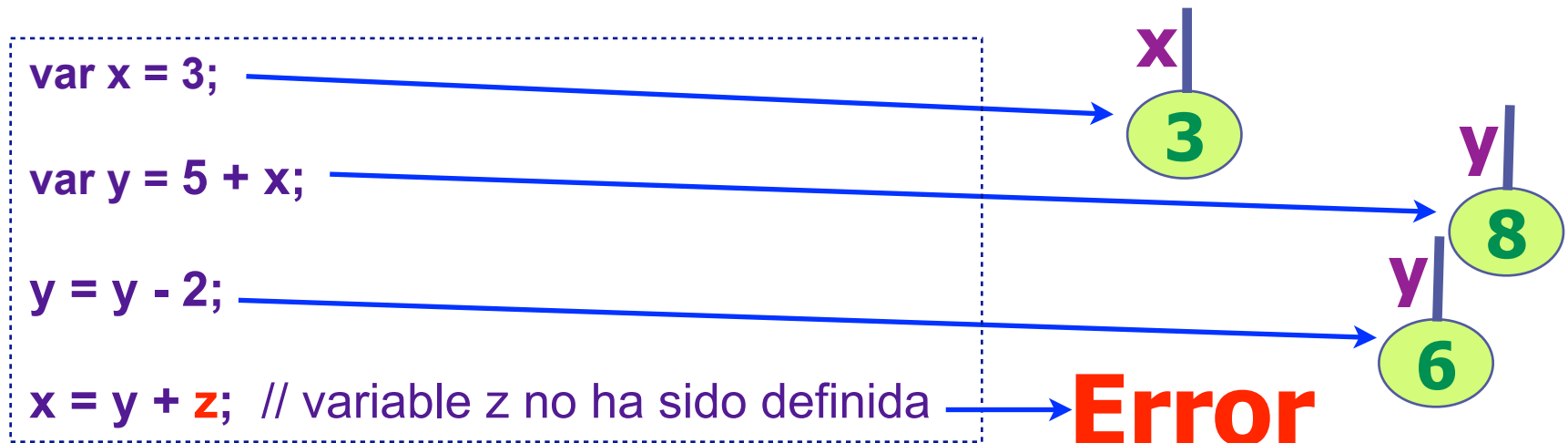
- ◆ El **nombre** (o identificador) de una variable debe comenzar por:
  - **letra**, **\_** o **\$**
    - ◆ El nombre pueden contener además **números**
  - Nombres **bien contruidos**: **x**, **ya\_vás**, **\$A1**, **\$**, **\_43dias**
  - Nombres **mal contruidos**: **1A**, **123**, **%3**, **v=7**, **a?b**, **..**
    - ◆ Nombre incorrecto: da error\_de\_ejecución e interrumpe el programa
- ◆ Un nombre de variable
  - **no** debe ser una **palabra reservada** de JavaScript
- ◆ Las variables son sensibles a **mayúsculas**
  - **mi\_var** y **Mi\_var** son variables distintas



# Expresiones con variables

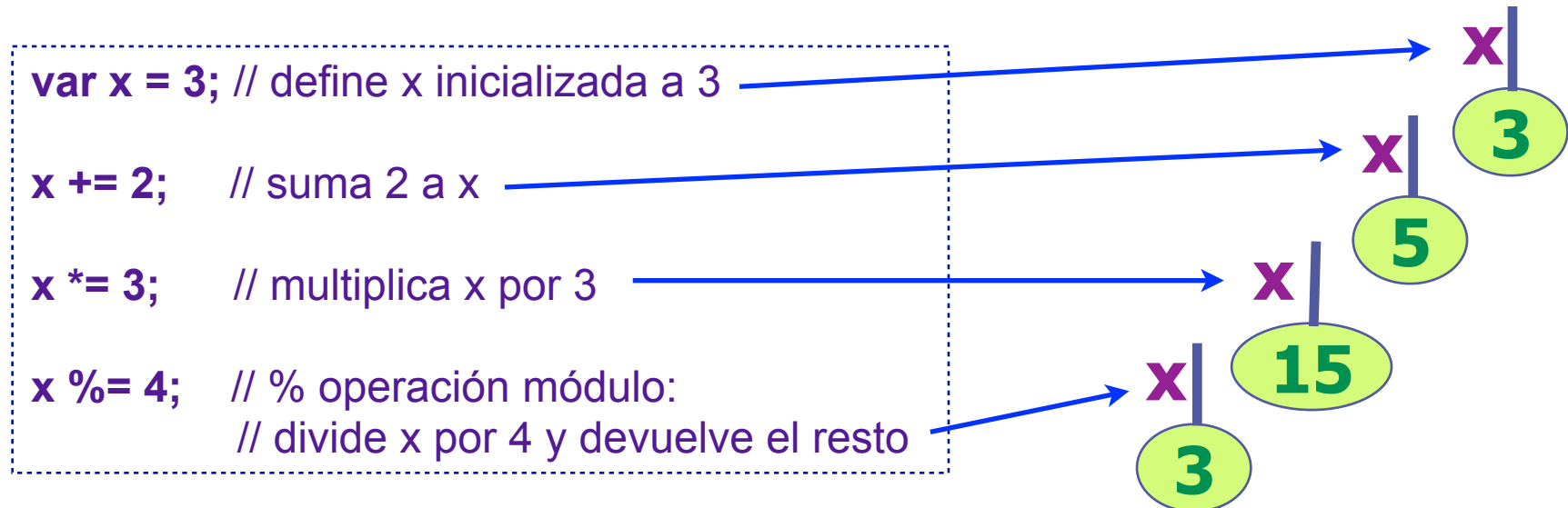
# Expresiones con variables

- ◆ Una variable representa el valor que contiene
  - Puede ser usada en expresiones como cualquier otro valor
- ◆ Una variable puede utilizarse en la expresión que se asigna a ella misma
  - La parte derecha usa el valor anterior a la ejecución de la sentencia
    - ◆ En  $y = y - 2$ ,  $y$  tiene el valor 8 antes de ejecutarse, asignándose a  $y$  el valor 6 (8-2)
- ◆ Usar una **variable no definida** en una expresión
  - provoca un **error** y la ejecución del **programa se interrumpe**



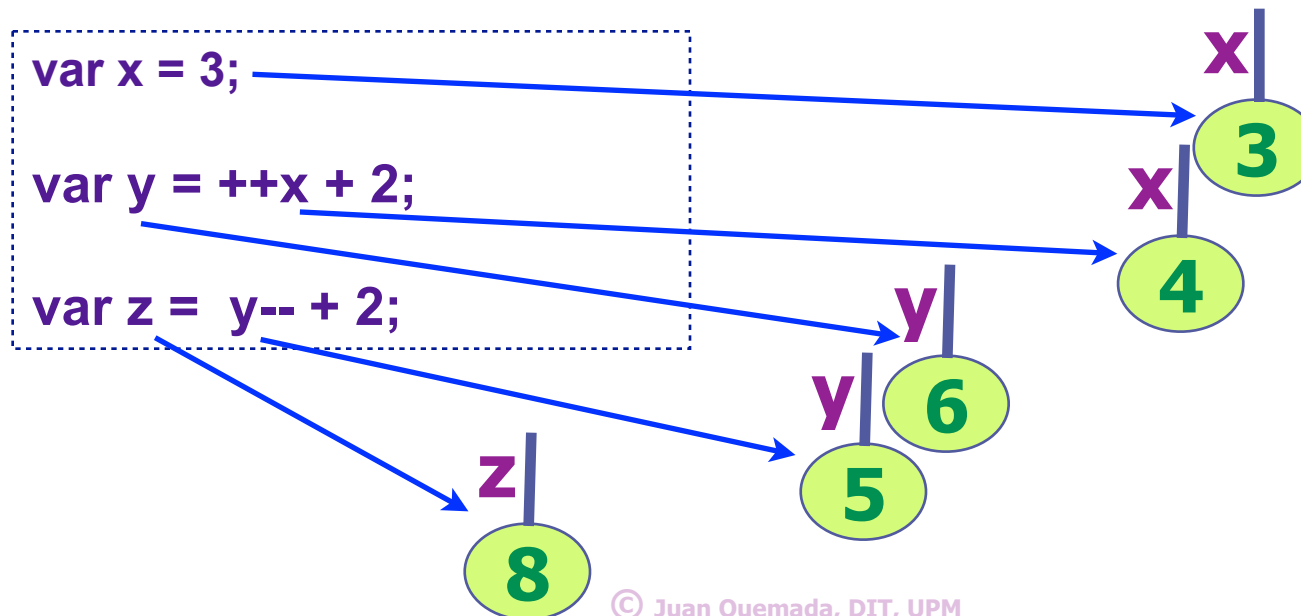
# Operadores de asignación

- ◆ Es muy común modificar el valor de una variable
  - sumando, restando, .... algún valor
    - ♦ Por ejemplo,  $x = x + 7$ ;  $y = y - 3$ ;  $z = z * 8$ ; .....
- ◆ JavaScript tiene operadores de asignación especiales para estos casos
  - $+=$ ,  $-=$ ,  $*=$ ,  $/=$ ,  $\% =$ , .....(y para otros operadores del lenguaje)
    - ♦  $x += 7$ ; será lo mismo que  $x = x + 7$ ;
    - ♦ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators>



# Pre y post incremento o decremento

- ◆ JavaScript posee los operadores ++ y -- de auto-incremento/decremento
  - auto-incremento (++) suma 1 a la variable a la que se aplica
  - auto-decremento (--) resta 1 a la variable a la que se aplica
- ◆ ++ y -- se aplican a la izquierda o derecha de una variable en una expresión
  - modificando el valor antes o después de evaluar la expresión
    - ◆ **Ojo!** Al producir efectos laterales y programas críticos, se recomienda un uso limitado
    - ◆ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators>



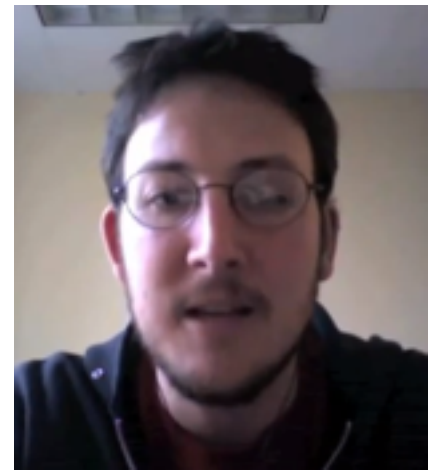


# Introducción a node.js

Juan Quemada, DIT - UPM



# node.js



## ◆node.js

- Intérprete de consola de JavaScript
  - Permite desarrollar servidores Web
    - Procesan HTTP ~1000% mas rapido que con hebras

## ◆Desarrollado en 2009 por Ryan Dahl

- Usando motor V8 de JavaScript (Google 2008)
  - V8: Compilador al vuelo de JavaScript muy eficiente



## ◆Historia: Estudiante de doctorado en matemáticas

- Se aburrió (2006) y se fue a viajar por Sudamérica
  - Se dedicó a programar Webs y conoció **Ruby on Rails**
    - No pudo acelerar Rails y desarrollo **node.js** al aparecer **V8**

## ◆Instalación e info: <http://nodejs.org/>

- Node up and running, T. Hughes-Croucher y M. Wilson, O'Reilly 2012, 1st Ed.

# comando node

## ◆node: comando UNIX

- Arranca un proceso UNIX con el intérprete de JavaScript
- Tiene dos modos básicos de funcionamiento
  - Modo comando
  - Modo interprete -> REPL

```
venus-2:~ jq$ node --help
Usage: node [options] [ -e script | script.js ] [arguments]
       node debug script.js [arguments]

Options:
  -v, --version          print node's version
  -e, --eval script      evaluate script
  -p, --print            print result of --eval
  -i, --interactive      always enter the REPL even if stdin
                        does not appear to be a terminal
  --no-deprecation       silence deprecation warnings
  --trace-deprecation    show stack traces on deprecations
  --v8-options           print v8 command line options
  --max-stack-size=val  set max v8 stack size (bytes)

Environment variables:
NODE_PATH               ':'-separated list of directories
                        prefixed to the module search path.
NODE_MODULE_CONTEXTS    Set to 1 to load modules in their own
                        global contexts.
NODE_DISABLE_COLORS     Set to 1 to disable colors in the REPL

Documentation can be found at http://nodejs.org/
venus-2:~ jq$
```

# Entrada/Salida y comunicación con la consola

- ◆ **console.log("Texto");** muestra **"Texto"** por la consola de usuario
  - Es una sentencia de evaluación de expresiones con efecto lateral
- ◆ Ejemplos de sentencia de **evaluación de expresiones**
  - **"Hola" + "Pepe", 3+2;**    o    **console.log("Texto");**
- ◆ Una expresión sin efecto lateral, solo genera un valor
  - Si ese valor no se guarda en una variable la instrucción es inútil
    - ◆ No tiene ningún efecto en el programa y solo consume recursos

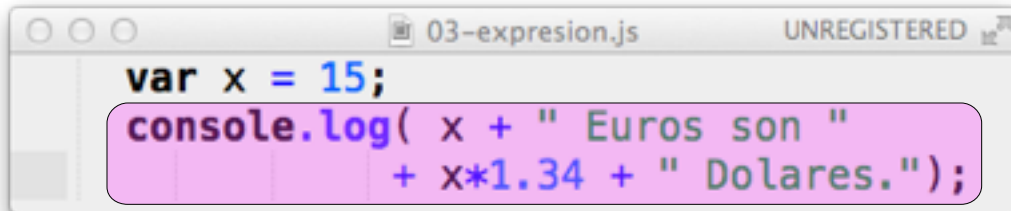
```
var x = "Hola";    // definición e inicialización de variable
```

```
x + "Pepe"; // es una expresión correcta, pero inútil al no guardar el resultado
```

```
x = x + "Pepe";    // asignación, es una expresión útil porque guarda el resultado
```

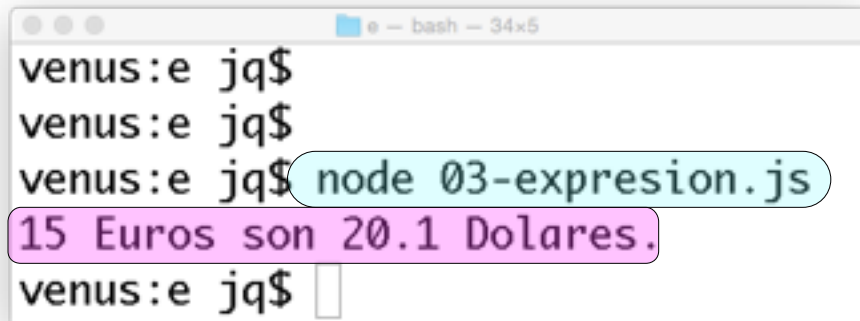
```
console.log(x);    // Expresión útil, envía "Hola Pepe" a la consola (exterior)
```

# Interprete node.js en modo comando



A screenshot of a code editor window titled '03-expresion.js' with 'UNREGISTERED' in the top right corner. The code is as follows:

```
var x = 15;  
console.log( x + " Euros son "  
            + x*1.34 + " Dolares.");
```



A screenshot of a terminal window titled 'e - bash - 34x5'. It shows the following sequence of commands and output:

```
venus:e jq$  
venus:e jq$  
venus:e jq$ node 03-expresion.js  
15 Euros son 20.1 Dolares.  
venus:e jq$
```

node.js permite ejecutar JavaScript de 2 formas diferentes

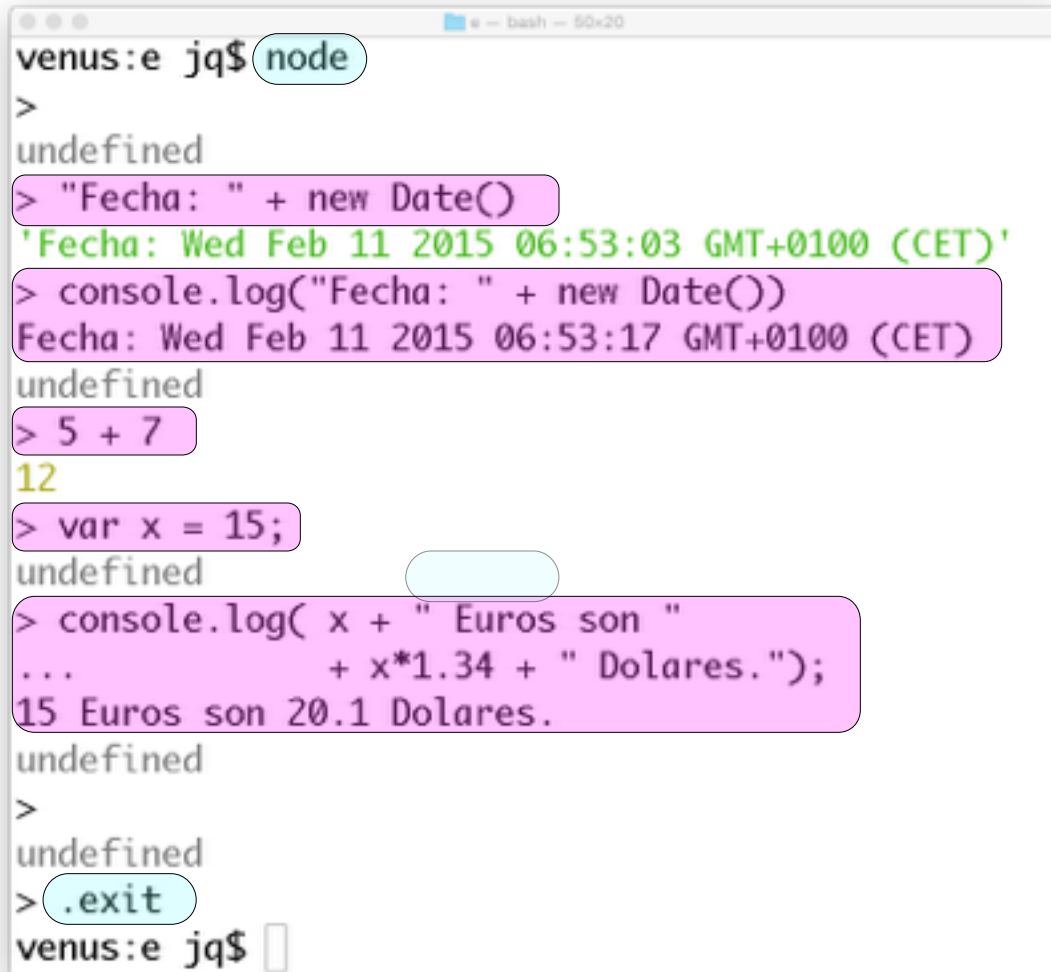
- 1) Ejecución en modo comando
- 2) Ejecución de instrucciones paso a paso

Esta transparencia muestra la ejecución del fichero en modo comando.

Para ejecutar un comando se pasa al interprete de node.js el nombre del fichero (comando) como parámetro, por ejemplo:

...> node 03-expresion.js

# Interprete interactivo



```
venus:e jq$ node
>
undefined
> "Fecha: " + new Date()
'Fecha: Wed Feb 11 2015 06:53:03 GMT+0100 (CET)'
> console.log("Fecha: " + new Date())
Fecha: Wed Feb 11 2015 06:53:17 GMT+0100 (CET)
undefined
> 5 + 7
12
> var x = 15;
undefined
> console.log( x + " Euros son "
...           + x*1.34 + " Dolares.");
15 Euros son 20.1 Dolares.
undefined
>
undefined
> .exit
venus:e jq$
```

El comando `node` permite además teclear y ejecutar JavaScript de forma interactiva en la consola, ejecutando las instrucciones **paso a paso**.

Este tipo de ejecución se denomina **REPL** (Read-Eval-Print-Loop) y esta descrito en el módulo `repl` de `node.js`:

<http://nodejs.org/api/repl.html>

El interprete interactivo se lanza invocando `node` sin parametros. Y su ejecución finaliza al teclear `".exit"`.

Todas las instrucciones JavaScript son expresiones que devuelven el valor al que se evalúan. El interprete REPL devuelve siempre este valor después de ejecutar la instrucción.

# Aplicaciones de servidor

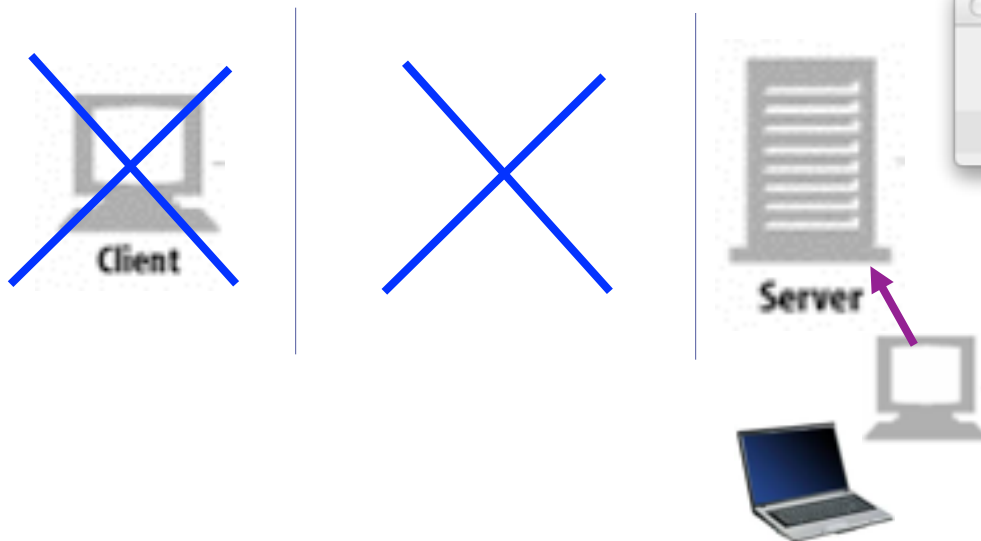
## ◆ Aplicaciones que **residen en un servidor**

### ■ ¡y se ejecutan en el servidor!

- ◆ Pueden realizar tareas locales como un programa C, C++, Java, ....

## ◆ Nuestro terminal hace de consola del servidor

- No hay comunicación con clientes remotos
  - ni por Sockets, ni por HTTP



```
03-expresion.js  UNREGISTERED
var x = 15;
console.log( x + " Euros son "
            + x*1.34 + " Dolares.");
```

**node.js** permite ejecutar programas JavaScript como procesos del servidor. Nuestro terminal es ahora la consola del servidor.

**¡No hay comunicación HTTP con el cliente Web!**



# Booleano, igualdad y otros operadores lógicos

# Tipo booleano

FALSE  
true

◆ El tipo **boolean** (booleano) solo tiene solo 2 valores

- **true**: verdadero
- **false**: falso

<b>!false</b>	=> true
<b>!true</b>	=> false

◆ Operador unitario **negación** (negation): **! <valor booleano>**

- Convierte un valor lógico en el opuesto, tal y como muestra la tabla

◆ Las expresiones booleanas, también se denominan expresiones lógicas

- Se evalúan siempre a verdadero (true) o falso (false)
  - ◆ Se utilizan para **tomar decisiones en sentencias condicionales**: if/else, bucles, ....
    - ◆ Por ejemplo: **if (expresión booleana) {Acciones\_A} else {Acciones\_B}**



# Conversión a booleano

FALSE  
true

◆ La regla de conversión de otros tipos a booleano es

- **false:** 0, -0, NaN, null, undefined, "", "
- **true:** resto de valores

◆ Cuando el operador negación ! se aplica a otro tipo

- convierte el valor a su equivalente booleano
  - ◆ y obtiene el valor booleano opuesto

◆ El operador negación aplicado 2 veces permite obtener

- el booleano equivalente a un valor de otro tipo, por ejemplo **!!1 => true**

!4	=> false
!"4"	=> false
!null	=> true
!0	=> true
!!""	=> false
!!4	=> true

# Operadores de identidad e igualdad

- ◆ Identidad o igualdad estricta: **<v1> === <v2>**
  - igualdad de tipo y valor:
    - ◆ aplicable a: **number, boolean y strings**
  - En objetos es **identidad de referencias**
- ◆ Desigualdad estricta: **<v1> !== <v2>**
  - negación de la igualdad estricta
- ◆ Igualdad y desigualdad débil: **== y !=**
  - **No utilizar!** Conversiones impredecibles!

// Identidad de tipos básicos

0 === 0	=> true
0 === 0.0	=> true
0 === 0.00	=> true
0 === 1	=> false
0 === false	=> false
'2' === "2"	=> true
'2' === "02"	=> false
" === ""	=> true
" === " "	=> false

◆ Mas info: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Sameness>

# Operadores de comparación

## ◆ JavaScript tiene 4 operadores de comparación

- Menor: `<`
- Menor o igual: `<=`
- Mayor: `>`
- Mayor o igual: `>=`

## ◆ Utilizar comparaciones solo con números (number)

- poseen una relación de orden bien definida

## ◆ No se recomienda utilizar con otros tipos: **string**, **boolean**, **object**, ..

- Las relación de orden en estos tipos existe, pero es muy poco intuitiva

- ◆ [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison\\_Operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators)

`1.2 < 1.3`  $\Rightarrow$  true

`1 < 1`  $\Rightarrow$  false

`1 <= 1`  $\Rightarrow$  true

`1 > 1`  $\Rightarrow$  false

`1 >= 1`  $\Rightarrow$  true

`false < true`  $\Rightarrow$  true

`"a" < "b"`  $\Rightarrow$  true

`"a" < "aa"`  $\Rightarrow$  true

# Operador y de JavaScript: &&

## ◆ Operador lógico y

- operador binario: **<valor\_1> y <valor\_2>**
  - ◆ Verdadero solo si ambos valores son verdaderos

## ◆ && es más que un operador lógico

- devuelve **<valor\_1>** o **<valor\_2>** sin modificarlos

## ◆ Semántica del operador **<valor\_1> && <valor\_2>**

- si **<valor\_1>** es equivalente a **false**
  - ◆ devuelve **<valor\_1>**, sino devuelve **<valor\_2>**

true && true	=> true
false && true	=> false
true && false	=> false
false && false	=> false

0 && true	=> 0
1 && "5"	=> "5"

# Operador o de JavaScript: ||

```
true || true  => true
false || true  => true
true  || false => true
false || false => false
```

```
undefined || 0    => 0
13 || 0           => 13
```

```
// Asignar valor por defecto
// si x es undefined o null
```

```
x = x || 0;
```

## ◆ Operador lógico o

- operador binario: **<valor\_1> o <valor\_2>**
  - ◆ Falso solo si ambos valores son falsos

## ◆ || es más que un operador lógico

- devuelve **<valor\_1> o <valor\_2>** sin modificarlos

## ◆ Semántica del operador **<valor\_1> || <valor\_2>**

- si **<valor\_1>** es equivalente a **true**
  - ◆ devuelve **<valor\_1>**, sino devuelve **<valor\_2>**

# Operador de asignación condicional: “?:”

- ◆ El operador de asignación condicional
  - devuelve un valor en función de una **condición** lógica
    - ◆ La condición lógica va siempre entre paréntesis
- ◆ Semántica de la asignación condicional: **(condición) ? <valor\_1> : <valor\_2>**
  - si **condición** se evalúa a **true**
    - ◆ devuelve **<valor\_1>**, sino devuelve **<valor\_2>**

(true) ? 0 : 7	=> 0
(false)? 0 : 7	=> 7

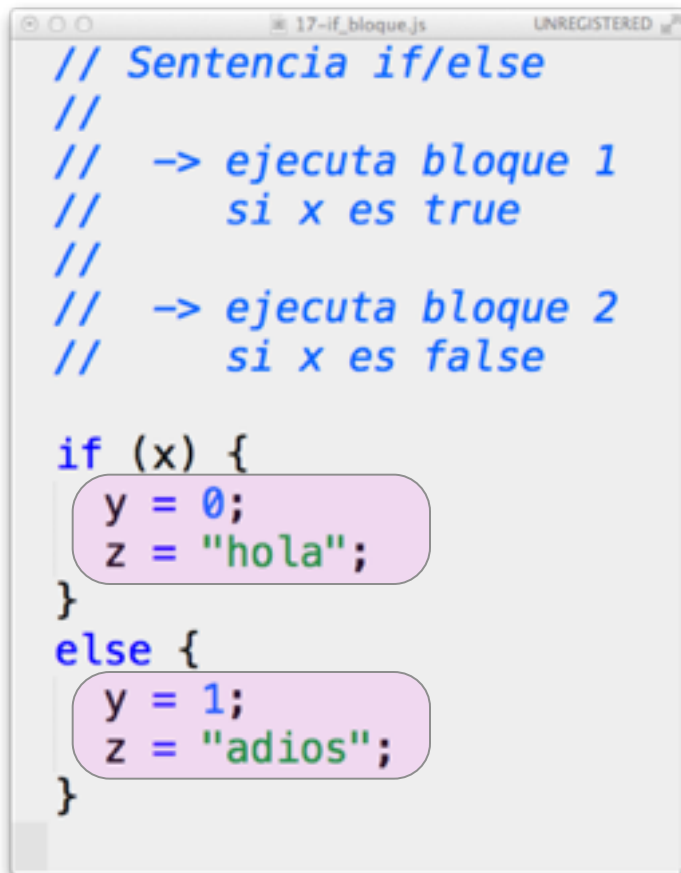
(7) ? 0 : 7	=> 0
("") ? 0 : 7	=> 7



# Sentencia if/else

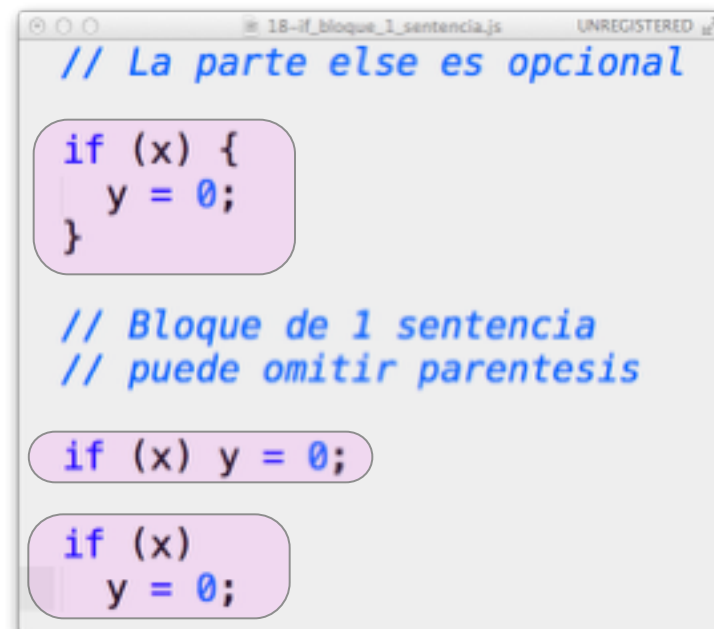
# Sentencia if/else

- ◆ **if/else** permite ejecución condicional de
  - bloques de instrucciones
- ◆ Comienza por la palabra reservada **if**
  - La condición va después entre paréntesis
    - ◆ Si evalúa a true se ejecuta bloque if, sino bloque else
- ◆ **Bloque:** sentencias delimitadas por **{..}**
  - Bloque de 1 sentencia puede omitir **}**
- ◆ La parte **else** es opcional
  - Si no existe es equivale a un bloque vacío



```
// Sentencia if/else
//
// -> ejecuta bloque 1
//     si x es true
//
// -> ejecuta bloque 2
//     si x es false

if (x) {
    y = 0;
    z = "hola";
}
else {
    y = 1;
    z = "adios";
}
```



```
// La parte else es opcional

if (x) {
    y = 0;
}

// Bloque de 1 sentencia
// puede omitir parentesis

if (x) y = 0;

if (x)
    y = 0;
```



## Ejemplo con sentencia if/else

```
mod01 — bash — 38x6
venus-5:mod01 jq$
venus-5:mod01 jq$ node 10-if-else.js
0.04083862667903304 MENOR que 0,5

venus-5:mod01 jq$
```

```
06-if-else.js UNREGISTERED
// Math.random() devuelve número aleatorio entre 0 y 1.
var numero = Math.random();

if (numero <= 0.5){
  console.log('\n' + numero + ' MENOR que 0,5 \n');
}
else {
  console.log('\n' + numero + ' MAYOR que 0,5 \n');
}
```

## Ejemplo con sentencia if

```
venus-5:mod01 jq$  
venus-5:mod01 jq$ node 11-if-only.js  
0.8151861219666898 MAYOR que 0,5  
  
venus-5:mod01 jq$
```

```
11-if-only.js UNREGISTERED  
// Math.random() devuelve un  
// número aleatorio entre 0 y 1.  
var numero = Math.random();  
var str = ' MAYOR que 0,5';  
  
if (numero <= 0.5){  
    str = ' MENOR que 0,5';  
}  
  
console.log('\n' + numero + str + '\n');
```

## Ejemplo con sentencia if/else-if

```
venus-5:mod01 jq$  
venus-5:mod01 jq$ node 12-if-else-if.js  
  
0.37068058364093304 MENOR que 0,66  
  
venus-5:mod01 jq$
```

```
12-if-else-if.js UNREGISTERED  
  
// Math.random() devuelve número aleatorio entre 0 y 1.  
var numero = Math.random();  
  
if (numero <= 0.33){  
    console.log('\n' + numero + ' MENOR que 0,33 \n');  
}  
else if (numero <= 0.66){  
    console.log('\n' + numero + ' MENOR que 0,66 \n');  
}  
else {  
    console.log('\n' + numero + ' MAYOR que 0,66 \n');  
}
```



# Números

# Números: tipo number

◆ Se representan sintácticamente en JavaScript con literales de números:

- **Números decimales:** 32, 32.11
- **Núm. hexadecimales:** 0xFF, 0X10
- **Coma flotante:** 3.2e1 (3,2x10<sup>1</sup>)
  - ◆ <mantisa>e<exponente>

◆ Cualquiera de estos literales representa un valor del tipo **number**

- Todos se codifican igual internamente
  - ◆ Coma flotante doble precisión (64bits)

```
10 + 4  => 14    // sumar
10 - 4  => 6     // restar
10 * 4  => 40    // multiplicar
10 / 4  => 2.5   // dividir
10 % 4  => 2     // operación resto
```

```
0xA + 4  => 14  // A es 10 en base 16
0x10 + 4  => 20 // 10 es 16 en base 16
```

```
3e2      => 300  // 3x10^2
3e-2     => 0,03 // 3x10^-2
```

```
// los decimales dan error de redondeo
// por la representación en coma flotante
0.1 + 0.2  => 0,3000000000000004
```

# Infinity y NaN

- ◆ El tipo **number** incluye 2 valores especiales
  - **Infinity y NaN**
- ◆ **Infinity** representa
  - El infinito matemático
  - Desborda la capacidad de representación en coma flotante con doble precisión
    - ◆ Rango real:  $\sim 1,797 \times 10^{308}$  ---  $5 \times 10^{-324}$
- ◆ **NaN** (Not a Number)
  - representa un resultado no numérico
    - ◆ strings que no son números
    - ◆ números complejos
    - ◆ .....

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

// Números demasiado grandes
5e500  => Infinity //desborda
-5e500 => -Infinity //desborda

// Número demasiado pequeño
5e-500 => 0        // desborda

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

# Conversión a number

- ◆ JavaScript **convierte tipos** según necesita
  - al evaluar expresiones
- ◆ JavaScript **convierte tipos** según necesita
  - **boolean**: convierte true a 1 y false a 0
  - **string**: convierte a valor o NaN
    - ◆ parseInt() y parseFloat() realizan la conversión
  - **null**: convierte a 0
  - **undefined**: convierte a NaN

```
'67' + 13      => '6713'
```

```
// string convertible a número
```

```
+'67'    + 13  => 80
```

```
+'6.7e1' + 13  => 80
```

```
// string no convertible a núm.
```

```
+'xx' + 13     => NaN
```

```
'xx' + 13      => 'xx13'
```

```
13 + true      => 14
```

```
13 + false     => 13
```

# parseInt(..) y parseFloat(..)

- ◆ JavaScript posee 2 funciones para convertir strings a números equivalentes
  - **parseInt(...)** analiza si es un literal de número entero y lo convierte a tal
  - **parseFloat(...)** analiza si es un literal de coma flotante y lo convierte a tal

## ◆ **parseInt(string, base)**

- Devuelve número equivalente a string o NaN
  - ◆ Analiza si string tiene un prefijo no vacío que es literal de entero en la base indicada, si no hay base se utiliza 10

## ◆ **parseFloat(string):**

- Devuelve número equivalente a string o NaN
  - ◆ analiza si string tiene un prefijo no vacío que es un literal de coma flotante con todos los campos en base 10

```
parseInt("1010") => 1010
```

```
parseInt("1010",2) => 10
```

```
parseInt("12",8) => 10
```

```
parseInt("10",10) => 10
```

```
parseInt("a",16) => 10
```

```
parseFloat("1e2") => 100
```

```
parseInt("xx",2) => NaN
```

```
parseInt("01xx") => 1
```

```
parseInt("01+4") => 1
```



# Modulo Math

- ◆ El Modulo Math contiene
  - constantes y funciones matemáticas
- ◆ Constantes
  - Números: E, PI, SQRT2, ...
  - ...
- ◆ Funciones
  - sin(x), cos(x), tan(x), asin(x), ....
  - log(x), exp(x), pow(x, y), sqrt(x), ....
  - abs(x), ceil(x), floor(x), round(x), ....
  - min(x,y,z,...), max (x,y,z,...), ...
  - random()

**Math.PI** => 3.141592653589793

**Math.E** => 2.718281828459045

// numero aleatorio entre 0 y 1

**Math.random()** => 0.7890234

**Math.pow(3,2)** => 9 // 3 al cuadrado

**Math.sqrt(9)** => 3 // raíz cuadrada de 3

**Math.min(2,1,9,3)** => 1 // número mínimo

**Math.max(2,1,9,3)** => 9 // número máximo

**Math.floor(3.2)** => 3

**Math.ceil(3.2)** => 4

**Math.round(3.2)** => 3

**Math.sin(1)** => 0.8414709848078965

**Math.asin(0.8414709848078965)** => 1

Mas 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)

# Clase Number

- ◆ La clase Number encapsula números
  - como objetos equivalentes
- ◆ Number define algunos métodos útiles
  - **toFixed(n)** devuelve string eq. a número
    - ◆ redondeando a n decimales
  - **toExponential(n)** devuelve string
    - ◆ redondeando mantisa a n decimales
  - **toString(base)** convierte número
    - ◆ a string equiv. en la **base** especificada
- ◆ JS convierte una expresión a objeto al
  - aplicar el método a una expresión
    - ◆ **Ojo!** literales dan error sintáctico

Tiene muchos más métodos, ver documentación en:

[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)

```
var x = 1.1;
```

```
x.toFixed(0)      => "1"
```

```
x.toFixed(2)      => "1.10"
```

```
(1).toFixed(2)    => "1.00"
```

```
1.toFixed(2)      => Error sintáctico
```

```
Math.PI.toFixed(4) => "3.1416"
```

```
(0.1).toExponential(2) => "1.00e-1"
```

```
x.toExponential(2)    => "1.10e+0"
```

```
(15).toString(2)      => "1111"
```

```
(15).toString(10)     => "15"
```

```
(15).toString(16)     => "f"
```



# Strings y UNICODE



# El tipo string

- ◆ El código UNICODE representa muchas lenguas diferentes
  - JavaScript utiliza UNICODE para codificar los caracteres de un string
- ◆ Literales de string: textos delimitados por **comillas** o **apóstrofes**
  - **"hola, que tal", 'hola, que tal', 'Γεια σου, ίσως' o '嗨，你好吗'**
    - ◆ en la línea anterior se representa "hola, que tal" en castellano, griego y chino
  - String vacío: **""** o **"**
  - **"texto 'entrecomillado' "**
    - ◆ comillas y apóstrofes se pueden anidar: **'entrecomillado'** forma parte del texto
- ◆ Operador de concatenación de strings: **+**
  - **"Hola" + " " + "Pepe"      =>    "Hola Pepe"**

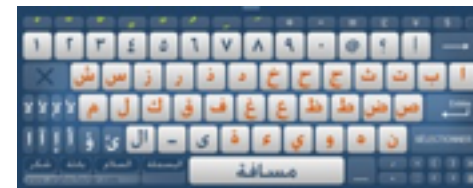


# UNICODE



Teclado chino

- ◆ UNICODE es un consorcio internacional: <http://www.unicode.org/>
  - Define normas de internacionalización (I18N), como el código UNICODE
    - ◆ UNICODE puede representar muchas lenguas: <http://www.unicode.org/charts/>
- ◆ JavaScript utiliza solo el **Basic Multilingual Plane** de UNICODE
  - Caracteres codificados en 2 octetos (16 bits), similar a BMP
    - ◆ UNICODE tiene otros planos que incluyen lenguas poco frecuentes
- ◆ **Teclado:** suele incluir solo las lenguas de un país
  - Los caracteres de lenguas no incluidas
    - ◆ solo se pueden representar con caracteres escapados
      - ◆ por ejemplo, `'\u55e8'` representa el ideograma chino '嗨'
- ◆ **Pantalla:** es gráfica y puede representar cualquier carácter



Teclado arabe



# Caracteres escapados

氷	𠩺	𠩺	𠩺	𠩺
2EA2	2EB2	2EC2	2ED2	2EE2

## ◆ Los **caracteres escapados**

- son caracteres no representables dentro de un string
  - ◆ comienzan por la barra inclinada (\) y la tabla incluye algunos de los más habituales

## ◆ Además podemos representar cualquier carácter UNICODE o ISO-LATIN-1:

- **\uXXXX** carácter UNICODE de código hexadecimal **XXXX** 
- **\xXX** carácter ISO-LATIN-1 de código hexadecimal **XX** 

## ◆ Algunos ejemplos

- "Comillas dentro de \"comillas\""
  - ◆ " debe ir escapado dentro del string
- "Dos \n líneas"
  - ◆ retorno de línea delimita sentencias
- "Dos \u000A líneas"

### CARACTERES ESCAPADOS

NUL (nulo):	\0, \x00, \u0000
Backspace:	\b, \x08, \u0008
Horizontal tab:	\t, \x09, \u0009
Newline:	\n, \x0A, \u000A
Vertical tab:	\v, \x0B, \u000B
Form feed:	\f, \x0C, \u000C
Carriage return:	\r, \x0D, \u000D
Comillas (dobles):	\", \x22, \u0022
Apóstrofe :	\', \x27, \u0027
Backslash:	\\, \x5C, \u005C

# Clase String

ciudad

[0] [1] ..... [5]

## ◆ La clase String

- incluye métodos y propiedades para procesar strings
  - [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String)

## ◆ Un string es un array de caracteres

- un índice entre **0 y número\_de\_caracteres-1** referencia cada carácter

◆ Propiedad con tamaño: **'ciudad'.length** ==> **6**

◆ Acceso como array: **'ciudad'[2]** ==> **'u'**

◆ Método: **'ciudad'.charCodeAt(2)** ==> **117**

- devuelve código UNICODE de tercer carácter

◆ Método: **'ciudad'.indexOf('da')** ==> **3**

- devuelve posición de substring

◆ Método: **'ciudad'.substring(2,5)** ==> **'uda'**

- devuelve substring entre ambos índices

# Ejemplo I18N

```
venus:modulo_05 jq$ node 11-unicode_hola.js
"hola, que tal": hola, que tal
'hola, que tal': hola, que tal
```

En griego (Γεια σου, ίσως): Γεια σου, ίσως

'hola, que tal' en chino (嗨, 你好吗): 嗨, 你好吗

Caracteres escapados (`\u55e8\u2013\u4f60\u597d\u5417`): 嗨, 你好吗

El caracter escapado `\u55e8` representa: 嗨

```
venus:modulo_05 jq$
```

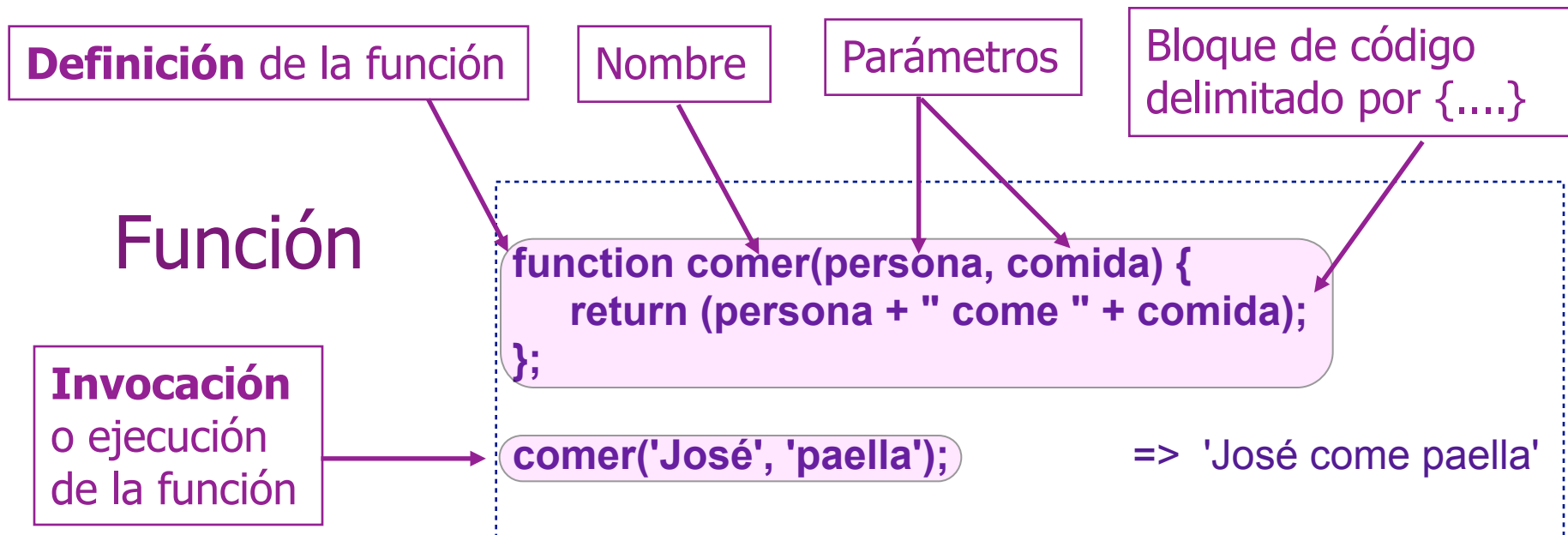
```
console.log('"hola, que tal": ' + "hola, que tal");
console.log("'hola, que tal': " + 'hola, que tal');
console.log();
console.log("En griego (Γεια σου, ίσως): " + 'Γεια σου, ίσως');
console.log();
console.log("'hola, que tal' en chino (嗨, 你好吗): " + '嗨, 你好吗');
console.log("Caracteres escapados (\\u55e8\\u2013\\u4f60\\u597d\\u5417): "
+ "\\u55e8\u2013\u4f60\u597d\u5417");
console.log();

var x = '嗨, 你好吗'.charAt(0).toString(16); // conversión char a string hexadec.
var y = String.fromCharCode(parseInt(x, 16)); // conversión hexadecimal a string
console.log('El caracter escapado \\u' + x + ' representa: ' + y);
```





# Funciones



## ◆ Función:

- bloque de código con parámetros, invocable (ejecutable) a través del nombre
  - ◆ La ejecución finaliza con la sentencia “**return expr**” o al **final** del bloque
- Al acabar la ejecución, devuelve un resultado: **valor de retorno**
- Doc: <https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Funciones>

## ◆ Valor de retorno

- resultado de evaluar **expr**, si se ejecuta la sentencia “**return expr**”
- **undefined**, si se alcanza final del bloque sin haber ejecutado ningún **return**

# Parámetros de una función

- ◆ Los parámetros de la función son variables utilizables en el cuerpo de la función
  - Al invocarlas se asignan los valores de la invocación
- ◆ La función se puede invocar con un **número variable de parámetros**
  - Un **parámetro inexistente** está **undefined**

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

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

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

# El array de argumentos

- ◆ Los parámetros de la función están accesibles también a través del
  - array de argumentos: **arguments[....]**
    - ◆ Cada parámetro es un elemento del array
- ◆ En: **comer('José', 'paella')**
  - **arguments[0]** => 'José'
  - **arguments[1]** => 'paella'

```
function comer() {  
    return (arguments[0] + " come " + arguments[1]);  
};
```

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

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

# Parámetros por defecto

- ◆ Funciones invocadas con un número variable de parámetros
  - Suelen definir parámetros por defecto con
    - ◆ `"x || <parámetro_por_defecto>"` // no funciona si x es "", 0, null, ..
    - ◆ `"(x!==undefined) ? x : <parámetro_por_defecto>"`
- ◆ Si x es "undefined", será false y devolverá **parámetro por defecto**
- ◆ Los parámetros son variables y se les puede asignar un valor

```
function comer (persona, comida) {  
    persona = (persona || 'Alguién');  
    comida = (comida || 'algo');  
    return (persona + " come " + comida);  
};
```

```
comer('José');    => 'José come algo'  
comer();          => 'Alguien come algo'
```



# Funciones como objetos y cierres

# Ambito y definiciones locales de una función

```
var ambito = "global";
```

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

```
ambito      => "global"  
ambitoLocal() => "local"
```

- ◆ 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**

# 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](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()**”
- ◆ 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 de 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'
```

# Funciones anidadas

- ◆ 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**
- ◆ Aplicando el operador paréntesis a la función exterior devuelve la función interior (código)
  - Aplicando 2 veces el operador paréntesis invoca, en cambio, la función interior
    - ◆ La función interior ve las variables exteriores s1 y s2 y puede concatenar s1+s2+s3

```
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"
```

# Cierres o closures

- ◆ Un cierre o closure es una 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 el interfaz es la **función contar()**
  - la **función contar** devuelve el valor de la **variable contador**, que luego incrementa
    - ◆ El **cierre encapsula “contador”** y la **función contar** es la única que tiene acceso a esta variable
      - ◆ Ninguna instrucción fuera del cierre puede modificar la variable contador
  - En la **variable enteroUnico** se carga la invocación del cierre (paréntesis del final)
    - ◆ Así nos devuelve la **función contar**, de forma que invocar **enteroUnico()** es lo mismo que invocar **contar()**

```
var enteroUnico = function () {  
  var contador = 0;  
  function contar () { return contador++; };  
  return contar;  
} ();
```

```
enteroUnico()      => 0  
enteroUnico()      => 1
```

```
// definición equivalente a la anterior  
var enteroUnico = function () {  
  var contador = 0;  
  return function () { return contador++; };  
} ();
```

```
enteroUnico()      => 0  
enteroUnico()      => 1
```



# Final del tema