



JavaScript 6-7-9 ... (2019)

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

JavaScript 6 - Índice

1.	Introducción a los tipos primitivos y operadores: números, strings, ...	3
2.	Números (dec., hex., oct., bin), NaN, Infinity, módulo Math y Number	16
3.	Strings, UNICODE, literal, plantillas, códigos escapados y String	21
4.	Programas, sentencias, comentarios, constantes, variables, var, let y const	29
5.	Expresiones con variables, operadores ++, --, +=, *=, -=, /=, %=, etc.	38
6.	Funciones, array args, parámetros por defecto y operador spread	42
7.	Funciones como objetos, notación flecha, ámbitos de visibilidad y cierres	49
8.	Boolean, operadores lógicos (!, &&,), de comparación (===, !==, <, >, >=, <=) y operador .. ? .. :	55
9.	Decisiones: if...else y switch...case	62
10.	Bucles: while, for, do...while, break y continue	68
11.	Arrays, spread y métodos sort, reverse, concat, join, indexOf, slice, splice, push y pop	75
12.	Arrays, spread/rest (...x) y asignación múltiple (destructuring assigment)	80
13.	Iteradores de arrays (forEach, find, findIndex, filter, map y reduce), bucles for...of y for...in	83
14.	Objetos, propiedades, métodos propios y this	88
15.	Objetos: Literal de ES6, multi-asignación, spread/rest (...x), for...in y Object.keys(..)	95
16.	JSON: JavaScript Object Notation	101
17.	Clases predefinidas de JavaScript: propiedades y métodos heredados	106
18.	Clase, prototipo y herencia: métodos de instancia o estáticos y this	111
19.	Estructurar el espacio de nombres con objetos, cierres (closures) y clases	117
20.	Referencias a objetos: comparación y compartición de objetos	124
21.	RegExp I: Búsqueda de patrones	129
22.	RegExp II: Repetición y alternativa	134
23.	RegExp III: Subpatrones y sustituciones	138



JavaScript

Introducción a los tipos primitivos y operadores: números, strings, ...

Juan Quemada, DIT - UPM

Tipos primitivos y objetos (ES5 y ES6)

◆ Tipos primitivos

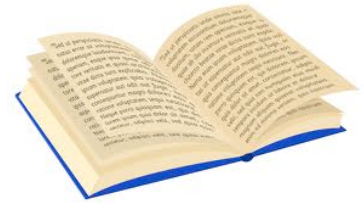
■ **number**

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



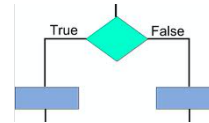
■ **string**

- ◆ 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()**



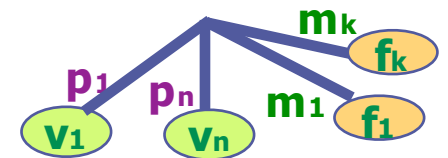
■ **undefined**

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

◆ **Objeto**: agregación estructurada 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/Guide/Values,_variables,_and_literals

Operador typeof (ES3 y ES5)

- ◆ 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"



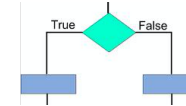
typeof "hola"

=> "string"



typeof true

=> "boolean"



typeof Symbol()

=> "symbol"



typeof undefined

=> "undefined"

UNDEFINED

typeof null

=> "object"

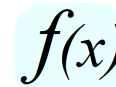
typeof new Date()

=> "object"

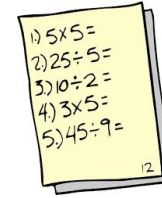


typeof new Function()

=> "function"



Números y operadores



◆ JavaScript permite construir expresiones con **operadores** predefinidos

- **+**, **-**, ***** y **/** son operadores binarios (2 valores) infijos (operador en el centro)
 - ◆ La evaluación de la expresión genera otro valor como resultado

13 + 7 => **20** // Suma de números

13 - 1.5 => **11.5** // Resta de números

(8*2 - 4)/3 => **4** // Expresión con paréntesis

=> significa
se evalúa a

En JavaScript
los decimales se
separan con un
punto, igual que
en inglés.

◆ Una expresión es una sentencia de JavaScript con una sintaxis estricta

- Al ejecutarla se comprueba que la sintaxis es correcta y se evalúa el resultado
 - ◆ Expresiones mal construidas dan error que interrumpe la ejecución (si no se atiende)

Expresión incorrecta: 2 operadores seguidos

8 / * 3 => ...?

Expresión incorrecta: dos números sin operador entre ellos

8 3 => ...?



Intérprete node: ejecutar expresiones

Consola Unix con intérprete **node.js**. Se suele invocar con el comando **node** (o **nodejs**)

```
venus:~ jq$  
venus:~ jq$ node  
> 13 + 7  
20  
> 13 + 2.5  
15.5  
> (8*2 - 4)/3  
4  
>  
> 8 / * 3  
...  
>  
> .exit  
venus:~ jq$
```

El comando "**node**" (sin parámetros) arranca el intérprete interactivo en un terminal de comandos. El intérprete analiza y ejecuta el texto introducido al teclear nueva línea (Enter).

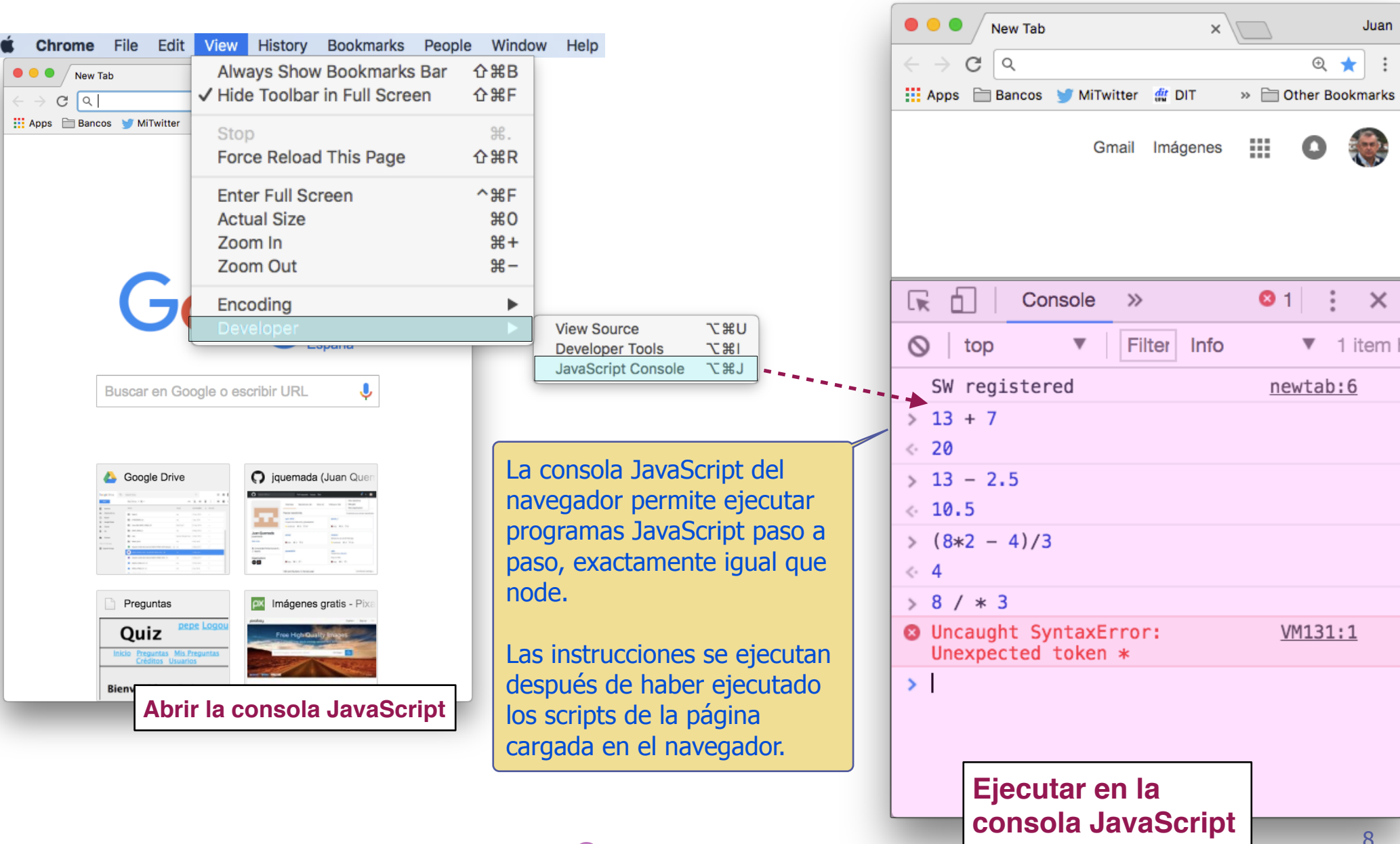
Los literales son las formas sintácticas que representan en JavaScript valores concretos. Por ejemplo los números se teclean igual que en expresiones matemáticas convencionales.

La captura del terminal adjunta muestra como evalúa JavaScript las expresiones de la transparencia anterior.

Teclear algo erróneo bloquea el intérprete, **^C** lo recupera.

El comando "**.exit**" finaliza la ejecución del intérprete y devuelve control a la shell de UNIX.

Consola del navegador (Chrome): ejecutar expr.



La consola JavaScript del navegador permite ejecutar programas JavaScript paso a paso, exactamente igual que node.

Las instrucciones se ejecutan después de haber ejecutado los scripts de la página cargada en el navegador.

Abrir la consola JavaScript

Ejecutar en la consola JavaScript

```
SW registered newtab:6
> 13 + 7
< 20
> 13 - 2.5
< 10.5
> (8*2 - 4)/3
< 4
> 8 / * 3
Uncaught SyntaxError: Unexpected token * VM131:1
> |
```


Texto: strings

◆ El texto escrito se representa en JavaScript con strings

- Un string delimita el texto con **comillas** o **apóstrofes**, por ej.
 - ◆ Frases: `"hola, que tal"` o `'hola, que tal'`
 - ◆ String vacío: `""` o `"`

◆ Ejemplo de "texto 'entrecorillado' "

- comillas y apóstrofes se pueden anidar
 - ◆ 'entrecorillado' forma parte del texto

◆ Operador `+` concatena strings, por ejemplo

- `"Hola" + " " + "Pepe"` \Rightarrow `"Hola Pepe"`

◆ La **propiedad length** de un string indica su longitud (Número de caracteres)

- `"Pepe".length` \Rightarrow `4`
- `"Hola Pepe".length` \Rightarrow `9`



Ejemplos de string

Consola Unix con interprete **node.js**

```
venus:~ jq$  
venus:~ jq$ node  
> "Eva"  
'Eva'  
> 'Eva'  
'Eva'  
> " "  
' '  
> "black" + "bird"  
'blackbird'  
> "Eva" + " " + "Brown"  
'Eva Brown'  
>  
> "Eva".length  
3  
> .exit  
venus:~ jq$
```

Los strings **"Eva"** y **'Eva'** son literales de string, que representan exactamente el mismo string o texto.

El string **" "** o **' '** representa el carácter **espacio** (space) o **blanco** (blank), que separa palabras en un texto.

El operador **+** aplicado a strings los concatena o une, generando un nuevo string con la unión de los dos. Es asociativo y permite concatenar más de 2 strings.

"Eva".length devuelve el contenido de la propiedad **length** del string: número de caracteres del string. **node** muestra un string en verde y un número en color crema.

Sobrecarga de operadores



◆ Los operadores tienen asignados caracteres especiales

- Por ejemplo, el operador suma tiene asignado el carácter **+**
 - ◆ El operador **+** está sobrecargado con 3 semánticas (significados) diferentes
 - El intérprete utiliza las reglas sintácticas de JavaScript para determinar la semántica

◆ Suma de números

- Operador binario de números (tipo number), infijo (operador en medio)

◆ Signo de un número

- Operador prefijo unario de número que define el signo positivo de un número

$$13 + 7 \Rightarrow 20$$



◆ Concatenación de strings

- Operador binario de cadenas de caracteres (tipo string), infijo (op. en medio)

$$+13 \Rightarrow 13$$



$$\text{"Hola "} + \text{"Pepe"} \Rightarrow \text{"Hola Pepe"}$$



Conversión de tipos en expresiones

◆ JavaScript realiza conversión automática de tipos

- La ambigüedad de una expresión se resuelve
 - ♦ con las reglas sintácticas y la prioridad entre operadores
 - Concatenación de strings tiene más prioridad que suma de números

◆ La expresión "13" + 7 es ambigua

- porque combina un string con un number
 - ♦ Si hay ambigüedad, JavaScript interpreta el operador + como concatenación de strings, convirtiendo 7 a string y concatenando ambos strings

ABCDE

◆ La expresión +"13" realiza conversión automática de tipos

- El operador + solo esta definido para number (no hay ambigüedad)
 - ♦ JavaScript debe convertir el string "13" a number antes de aplicar operador +



```
jq - node - 20x10
>
> 13 + 7
20
> "13" + "7"
'137'
> "13" + 7
'137'
> +"13" + 7
20
>
```

La prioridad de los operadores es descendente y de izquierda a derecha. (Mayor si más arriba o más a izq.)

[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)

. [] new	Acceso a propiedad o invocar método; Índice de array; Crear objeto
()	Invocación de función/método o evaluar expresión
++ --	Pre o post auto-incremento; Pre o post auto-decremento
! ~	Negación lógica (NOT); complemento de bits
+ -	Operador unitario, números. Signo positivo; Signo negativo
delete	Borrar propiedad de un objeto
typeof void	Devolver tipo; Valor indefinido (operador de escasa utilidad)
* / %	Números. Multiplicación; División; Modulo (o resto)
+ + -	Concatenación de strings; Números: Suma y Resta
<< >> >>>	Desplazamientos de bit
< <= > >=	Menor; Menor o igual; Mayor; Mayor o igual
instanceof in	¿Objeto pertenece a clase?; ¿Propiedad pertenece a objeto?
== != === !==	Igualdad; Desigualdad; Identidad; No identidad
&	Operación y (AND) de bits
^	Operación ó exclusivo (XOR) de bits
 	Operación ó (OR) de bits
&&	Operación lógica y (AND)
 	Operación lógica o (OR)
?:	Asignación condicional
=	Asignación de valor
OP=	Asignación con operación: += -= *= /= %= <<= >>= >>>= &= ^= =
yield (ES6)	Generar nuevo elemento de un "iterable"
... (ES6)	Distribuir (Spread) elems (array, obj, ..) o agrupar (Rest) parámetros (función)
,	Evaluación múltiple

8*2 - 4 => 12

* tiene más prioridad que -, pero (..) obliga a evaluar antes - en:

8*(2 - 4) => -16

Operadores ES5 y ES6

La prioridad de los operadores es descendente y de izquierda a derecha. (Mayor si más arriba o más a izq.)

[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)

. [] new	Acceso a propiedad o invocar método; Índice de array; Crear objeto
()	Invocación de función/método o evaluar expresión
++ --	Pre o post auto-incremento; Pre o post auto-decremento
! ~	Negación lógica (NOT); complemento de bits
+ -	Operador unitario, números. Signo positivo; Signo negativo
delete	Borrar propiedad de un objeto
typeof void	Devolver tipo; Valor indefinido (operador de escasa utilidad)
* / %	Números. Multiplicación; División; Modulo (o resto)
+ + -	Concatenación de strings; Números: Suma y Resta
<< >> >>>	Desplazamientos de bit
< <= > >=	Menor; Menor o igual; Mayor; Mayor o igual
instanceof in	¿Objeto pertenece a clase?; ¿Propiedad pertenece a objeto?
== != === !==	Igualdad; Desigualdad; Identidad; No identidad
&	Operación y (AND) de bits
^	Operación ó exclusivo (XOR) de bits
 	Operación ó (OR) de bits
&&	Operación lógica y (AND)
 	Operación lógica o (OR)
?:	Asignación condicional
=	Asignación de valor
OP=	Asignación con operación: += -= *= /= %= <<= >>= >>>= &= ^= =
yield (ES6)	Generar nuevo elemento de un "iterable"
... (ES6)	Distribuir (Spread) elems (array, obj, ..) o agrupar (Rest) parámetros (función)
,	Evaluación múltiple

+"3" + 7 => 10

+ unitario (signo) tiene mas prioridad que **+ binario** (suma) y se evalúa antes

Operadores ES5 y ES6



JavaScript

Números (dec., hex., oct., bin), NaN, Infinity, módulo Math y Number

Juan Quemada, DIT - UPM

Literales de números

◆ Los valores numéricos

- se representan con **literales de números**
 - ◆ Los literales permiten representar números en varios formatos

◆ Números decimales en coma fija

- Enteros: **1, 32,**
- Números con decimales: **1.2, 32.1,**

◆ Enteros hexadec., binarios y octales

- Hexadecimal: **0xFF, 0X10ff, ...**
- Binarios (ES6): **0b01101000, 0B1010, ...**
- Octal (ES6): **0o7123, 0O777, ...**

◆ Coma flotante (decimal)

- Coma flotante: **3.2e1 (3,2x10¹)**
 - ◆ sintaxis: **<mantisa>e<exponente>**

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

0b1000 + 4	=> 12	// 0b1000 es 8 en dec.
0o10 + 4	=> 12	// 0o10 es 8 en dec.

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

Infinity y NaN

◆ El tipo number posee **valores especiales**

- **NaN**
- **Infinity** y **-Infinity**

◆ **NaN** (Not a Number)

- representa un **valor no numérico**
 - ◆ números complejos
 - ◆ strings no convertibles a números

◆ **Infinity** representa

- El **infinito matemático**
- **Desbordamiento**

◆ El tipo number utiliza el formato

- IEEE 754 coma flotante doble precisión (64 bits)
 - ◆ Reales máximo y mínimo: **$\sim 1,797 \times 10^{308}$** y **$5 \times 10^{-324}$**
 - ◆ Entero máximo: **9007199254740992**
- Cualquier literal se convierte a este formato

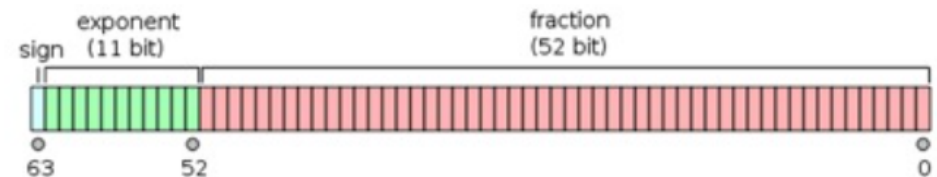
```
+ 'xx'    => NaN      // no representable

+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         // redondeo

// los decimales dan error de redondeo
0.1 + 1.2 => 1,30000000000004
```



Modulo Math

◆ El Modulo Math contiene

- constantes y funciones matemáticas

◆ Constantes

- Números: **E**, **PI**, **SQRT2**, ...

◆ Funciones

- $\sin(x)$, $\cos(x)$, $\tan(x)$, $\text{asin}(x)$,
- $\log(x)$, $\exp(x)$, $\text{pow}(x, y)$, $\text{sqrt}(x)$,
- $\text{abs}(x)$, $\text{ceil}(x)$, $\text{floor}(x)$, $\text{round}(x)$,
- $\text{min}(x,y,z,...)$, $\text{max}(x,y,z,...)$, ...
- $\text{random}()$
-

◆ Lista de todas las constantes y funciones, incluyendo las nuevas de ES6:

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math

```
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
```

Métodos de Number

◆ La clase Number define propiedades y métodos

- estos procesan valores de tipo number

◆ Algunos métodos de Number son

- **toFixed(n)** devuelve string equiv. a número
 - ◆ redondeando a n decimales
- **toPrecision(n)** devuelve string equiv. a número
 - ◆ redondeando número a n dígitos
- **toExponential(n)** devuelve string eq. a número
 - ◆ redondeando mantisa a n decimales
- **toString([base])** convierte un número a
 - ◆ string con el número en la base indicada
 - ◆ [base] es opcional, por defecto base 10

◆ Los métodos se invocan con el operador punto: "."

- **OJO!** los literales de números deben estar entre paréntesis (ver ejemplos), sino da error

◆ Lista de propiedades y métodos de la clase Number, incluyendo las nuevas de ES6:

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number

(1).toFixed(2)	=> "1.00"
(1).toPrecision(2)	=> "1.0"
1.toFixed(2)	=> Error
Math.PI.toFixed(4)	=> "3.1416"
Math.E.toFixed(2)	=> "2.72"
(1).toExponential(2)	=> "1.00e+0"
(31).toString(2)	=> "11111"
(31).toString(10)	=> "31"
(31).toString(16)	=> "1f"
(10.75).toString(2)	=> "1010.11"
(10.75).toString(16)	=> "a.c"

Conversión a number:

Number, parseInt y parseFloat

◆ **Number(<valor>)**: convierte cualquier valor a su equivalente en Number()

- **Number(..)** debe utilizarse para convertir porque corrige los errores de parseInt y parseFloat

◆ **parseInt(string, [base])**: función predefinida que convierte **string** a **number**

- **string** se interpreta como un entero en la **base** indicada (por defecto base 10)
 - ♦ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseInt
- **OJO!** Si un substring tiene sintaxis de número, genera **número** y no **NaN**, p. e. 01xx

◆ **parseFloat(string)**: función predefinida de JS que convierte **string** a **number**

- **string** se interpreta como un número en coma flotante
 - ♦ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseFloat

Number('60')	=> 60
Number('01xx')	=> NaN
parseInt('60')	=> 60
parseInt('60.45')	=> 60
parseInt('xx')	=> NaN
parseInt('01xx')	=> 1
parseInt('01+4')	=> 1

parseInt('1010')	=> 1010
parseInt('1010',2)	=> 10

parseInt('12',8)	=> 10
parseInt('10',10)	=> 10
parseInt('a',16)	=> 10

Number('1e2')	=> 100
Number('1.3e2')	=> 130
Number('01xx')	=> NaN

parseFloat('1e2')	=> 100
parseFloat('1.3e2')	=> 130
parseFloat('xx1e2')	=> NaN



Strings, UNICODE, literal, plantillas, códigos escapados y String

Juan Quemada, DIT - UPM



El tipo string

- ◆ Los literales de string se utilizan para representar textos
 - Puede representar lenguas diferentes porque utiliza el código UNICODE
 - ◆ Lenguas y símbolos soportados en UNICODE: <http://www.unicode.org/charts/>
 - Los strings se denominan también cadenas de caracteres
- ◆ Literal 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 'entrecorillado'"
 - ◆ comillas y apóstrofes se pueden anidar: 'entrecorillado' forma parte del texto
- ◆ Operador de concatenación de strings: +
 - "Hola" + " " + "Pepe" ==> "Hola Pepe"
- ◆ ES6 introduce **plantillas de strings** delimitados por **comillas invertidas** `...`
 - Son strings multi-línea y pueden contener expresiones delimitadas por **`${expr}`** que evalúan la expresión y la sustituyen por el valor correspondiente
 - ◆ Por ejemplo: `Un día tiene `${24*60}` minutos y `${24*60*60}` segundos`

Ejemplos de string

Consola Unix con intérprete **node.js** (se podrían ejecutar igual en la consola del navegador).

```
venus:ej jq$ node
> "Eva"
'Eva'
> 'Eva'
'Eva'
> " "
' '
> "corta" + "plumas"
'cortaplumas'
> "Eva"+" " + "Moya"
'Eva Moya'
> "Eva".length
3
> `Un día son ${24*60*60} segundos`
'Un día son 86400 segundos'
> `Primera línea
... segunda línea`
'Primera línea\nsegunda línea'
> .exit
venus:ej jq$
```

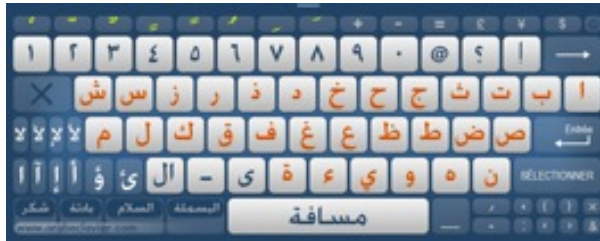
Los strings **"Eva"** y **'Eva'** son literales de string, que representan exactamente el mismo string o texto.

El string **" "** o **' '** representa el carácter **espacio** (space) o **blanco** (blank), que separa palabras en un texto.

El operador **+** aplicado a strings los concatena o une, generando un nuevo string. Es un operador asociativo que permite concatenar más de 2 strings.

"Eva".length devuelve el número de caracteres del string. **length** es una propiedad del string.

Las plantillas de strings (ES6) se delimitan por comillas invertidas **`...`**, son multi-línea y pueden incluir expresiones (**\${<expr>}**).



Ejemplos de
teclados



Entrada y Salida de Caracteres

- ◆ JavaScript utiliza el plano básico multi-lingüe (BMP) de UNICODE
 - Caracteres codificados en 2 octetos (16 bits): **65536 combinaciones**
- ◆ El **texto se introduce** en el ordenador **tecleando en el teclado**
 - Los **teclados** suelen incluir solo las teclas del **alfabeto(s) de un país**
 - ◆ Tecleando solo es posible introducir un **conjunto muy pequeño de símbolos**
 - Caracteres **no incluidos en el teclado** pueden añadirse por "**corta y pega**"
 - ◆ por ejemplo, de strings traducidos con Google translate: <https://translate.google.es>
 - O también pueden añadirse con los **códigos escapados**
 - ◆ Permiten introducir caracteres no existentes en el teclado con códigos especiales
- ◆ **Pantalla:** es gráfica y **puede mostrar cualquier carácter**

Códigos escapados

◆ Definen caracteres a través de códigos

- Comienzan con barra inversa \... y hay 3 tipos
 - ◆ Códigos escapados ASCII (de control)
 - ◆ Caracteres UNICODE
 - ◆ Caracteres ISO-8859-1

◆ Códigos escapados ASCII

- La tabla incluye los caracteres ASCII más habituales en formato escapado
 - ◆ Primero viene el formato ASCII tradicional, seguido de los códigos ISO-8859-1 y UNICODE equivalentes

CARACTERES DE CONTROL ASCII

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

◆ Caracteres UNICODE o ISO-8859-1 se definen con **punto de código**, así:

- UNICODE: \uHHHH donde HHHH es el punto del código (4 dígitos hex), p.e. \u005C
- ISO-8859-1: \xHH donde HH es el punto del código (2 dígitos hex), p.e. \x5C



◆ Algunos ejemplos

- Las "Comillas dentro de \"comillas\"" deben ir escapadas dentro del string.
- En "Dos \n líneas" el retorno de línea (\n) separa las dos líneas.
- En "Dos \u000A líneas" las líneas se separan con el código UNICODE \u000A equivalente a \n.

Métodos de String

ciudad
[0] [1] [5]

◆ Algunos métodos y elementos útiles de String

- Más info: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

◆ Un string es un array de caracteres y puede accederse como tal

- Un índice, de **0** a **length-1**, permite acceder a los caracteres individualmente

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

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

- La propiedad **length** contiene el número de caracteres del string

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

- devuelve el substring comprendido entre ambos índices

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

- devuelve el número (decimal) con código UNICODE del tercer carácter

◆ Método: **String.fromCharCode(117)** **=> 'u'**

- devuelve el string con el carácter correspondiente al código numérico (decimal)

Ejemplos de String

```
venus:~ jq$ node
> "La Ñ en ISO-8859-1 es: \xd1"
'La Ñ en ISO-8859-1 es: Ñ'
> "La á en ISO-8859-1 es: \xe1"
'La á en ISO-8859-1 es: á'
> "Backslash \\ debe escaparse"
'Backslash \\ debe escaparse'
> "El Euro: \u20ac"
'El Euro: €'
> "El Yen Japonés: \xa5"
'El Yen Japonés: ¥'
> "Ciudad"[1]
'i'
> "Ciudad".charCodeAt(1)
105
> String.fromCharCode(105)
'i'
> "Ciudad".substring(3,5)
'da'
> "Ciudad".substring(3,5).length
2
> .exit
venus:~ jq$
```

La Ñ existe en el código ISO-8859-1 y su código numérico en hexadecimal es d1, por lo que se puede incluir en un string tecleando Ñ o como \xd1.

La á existe también en el código ISO-8859-1 y la introducimos tecleando el acento y luego la letra a o con el código numérico en hexadecimal \xe1.

La barra inclinada (backslash) debe escaparse (\\) para que se visualice.

EL Euro no existe en ISO-8859-1 porque este código se creó antes de existir el Euro. Unicode se actualizó al aparecer el Euro añadiendo el símbolo € con el código \u20ac.

EL Yen Japonés si existe en ISO-8859-1: código hexadecimal \xa5.

Un string es un array (matriz) de caracteres, numerados de 1 a n-1 (último-1). **"Ciudad"[1]** indexa el segundo carácter del string, el primero será **"Ciudad"[0]**.

Al invocar el método **charCodeAt(1)** con el operador "." sobre el string **"Ciudad"** nos devuelve el valor numérico decimal del **punto del código** del 2º carácter ("i").

String.fromCharCode(105) realiza la operación inversa, devuelve un string con el carácter cuyo valor (punto del código) se pasa como parámetro.

"Ciudad".substring(3,5) devuelve la subcadena entre 3 y 5: **"da"**

"Ciudad".substring(3,5).length devuelve la longitud de la subcadena devuelta ("da"), que tiene 2 caracteres.

Conversión a String

◆ JavaScript posee además la función **String(<valor>)**

- **String(<valor>)** transforma a string cualquier valor
 - ◆ String(...) debe utilizarse para hacer conversiones explícitas (es mas legible y completo)

◆ Los tipos y objetos suelen tener un método **toString()**

- **toString()** transforma un valor u objeto en un string equivalente
 - ◆ toString() suele mostrar el contenido de los objetos de forma legible
- JavaScript utiliza el método toString() en conversiones automáticas

String(4)	=> "4"
String(-4)	=> "-4"
String(NaN)	=> "NaN"
String(Infinity)	=> "Infinity"
String(undefined)	=> "undefined"

(4).toString()	=> "4"
(-4).toString()	=> "-4"
(NaN).toString()	=> "NaN"
(Infinity).toString()	=> "Infinity"
(undefined).toString()	=> "undefined"



JavaScript

Programas, sentencias, comentarios,
constantes, variables, var, let y const

Juan Quemada, DIT - UPM

Programa, sentencia y código fuente

◆ Programa: secuencia de sentencias

- Se ejecutan en el orden que tienen en la secuencia
 - ◆ Hay excepciones: sentencias de bucles y saltos

◆ Comentario: solo tienen valor informativo

- Documenta el programa y ayuda a entenderlo mejor
 - ◆ Hay dos tipos de comentarios: mono-línea y multi-línea

◆ Sentencia: orden al procesador

- Especifica una **tarea** a realizar por el procesador
 - ◆ Se recomienda terminar la sentencia con punto y coma (;)
- Cada una tiene una sintaxis (estructura) diferente
 - ◆ Parecida a la sintaxis de español, inglés,...

◆ Código fuente: texto con las sentencias y comentarios de un programa

- Se edita con **editor de texto plano**: nano, notepad, vi, vim, sublime-text, atom,
 - ◆ **Fichero fuente**: fichero que contiene un programa JavaScript ejecutable

Sentencia 1: define la **variable x** con **valor 7**.

Comentario multi-línea: delimitado con `/* */`

```
/* Ejemplo de  
programa JavaScript */
```

```
var x = 7; // Define variable
```

```
// muestra x*1,13 por consola  
console.log(x * 1.13);
```

Sentencia 2: Sentencia de salida. Muestra por consola el **valor x * 1,13**

Comentario mono-línea: empieza con `//` y acaba al final de la línea.

Intérprete node: modo comando

El fichero fuente **01-conversor.js** contiene el siguiente programa JavaScript ejecutable con node.js tal y como se muestra más abajo.

El intérprete node permite ejecutar también programas en **modo comando** (de una vez), tal y como ilustrarnos aquí.

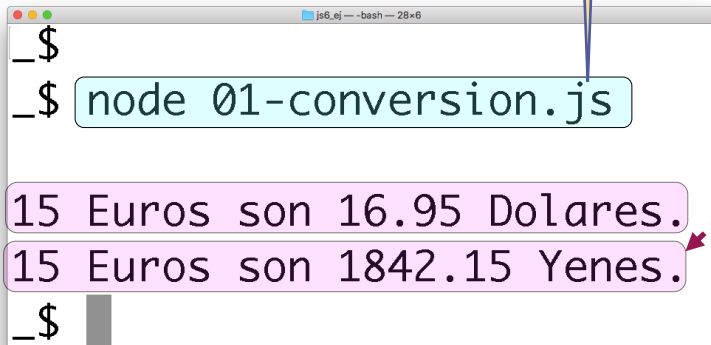
El ejemplo, en el fichero fuente "**01-conversor.js**", define primero la variable x y le asigna el valor 15. A continuación calcula su equivalente en dólares y yenes japoneses.

```
var x = 15;  
console.log();  
console.log(x + " Euros son " + x*1.13 + " Dolares.");  
console.log(x + " Euros son " + x*122.81 + " Yenes.");
```

console.log() envía a consola una línea en blanco.

console.log(expresión)
envía el resultado de
evaluar la expresión a
consola.

El comando node ejecuta
un programa cuando este
se pasa como parámetro.



A terminal window titled 'js6_ej' with a prompt '_\$'. The command 'node 01-conversion.js' is entered and executed. The output shows two lines: '15 Euros son 16.95 Dolares.' and '15 Euros son 1842.15 Yenes.' followed by a new prompt '_\$'.

El nombre de la variable representa el
valor contenido y puede utilizarse en
expresiones como otro valor.

Un programa se ejecuta en modo
comando pasando la ruta al fichero
como parámetro al intérprete node.js:

...\$ node 01-conversor.js

Creación y asignación de variables (ES5)

- ◆ Una variable es un **contenedor de valores**, cuyo contenido puede variar
 - Las variables se declaran hasta **ES5** con la palabra reservada **var**
 - ◆ Los valores (incluido el inicial) se asignan con el operador **=**
- ◆ Las variables de JavaScript son **no tipadas**
 - Pueden contener cualquier tipo de valor
 - ◆ Una variable puede contener un número, un string, undefined, un objeto, ..
- ◆ **Estado de un programa:** variables creadas con los valores guardados
 - El estado varía a medida que se **ejecutan las instrucciones**

```
var x = 5;           // Creates variable x with initial value 5
x = "Hello";        // Assigns string "Hello" to variable x
x = undefined;      // Assigns undefined to variable x
x = new Date();     // Assigns Date object to variable x
```

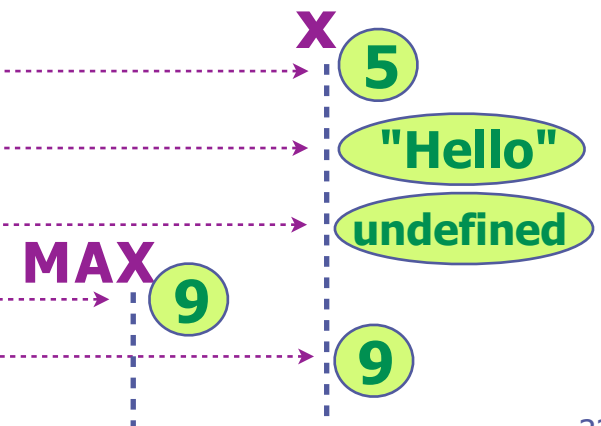
Evolución del estado en función de la instrucción ejecutada



Constantes y variables (ES6)

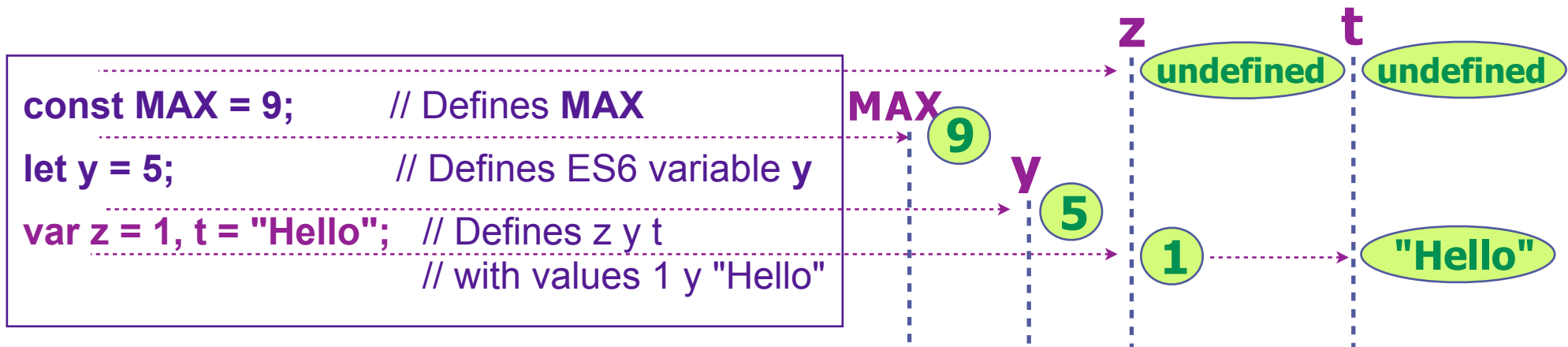
- ◆ En **ES6** las variables también se declaran con la palabra reservada **let**
 - Los valores se asignan a las variables también con el operador: **=**
- ◆ Características de las variables ES6
 - También son **no tipadas** y pueden contener cualquier valor como en ES5
 - Modifican el estado de forma similar a ES5
 - Su ámbito de visibilidad está limitado al bloque que la contiene
 - ♦ Solo existen desde su definición hasta el final del bloque
- ◆ ES6 permite definir además **constantes** con la palabra reservada **const**
 - Al definir una constante es obligatorio asignar un valor, que ya no podrá modificarse
 - ♦ Se suele utilizar un convenio por el que las constantes suelen utilizar letras mayúsculas

```
let x = 5;           // Creates variable x with initial value 5
x = "Hello";        // Assigns string "Hello" to variable x
x = undefined;      // Assigns undefined to variable x
const MAX = 9;      // Creates constant MAX equal to 9
x = MAX;            // Assigns MAX (9) to variable x
```



Declaración de variables

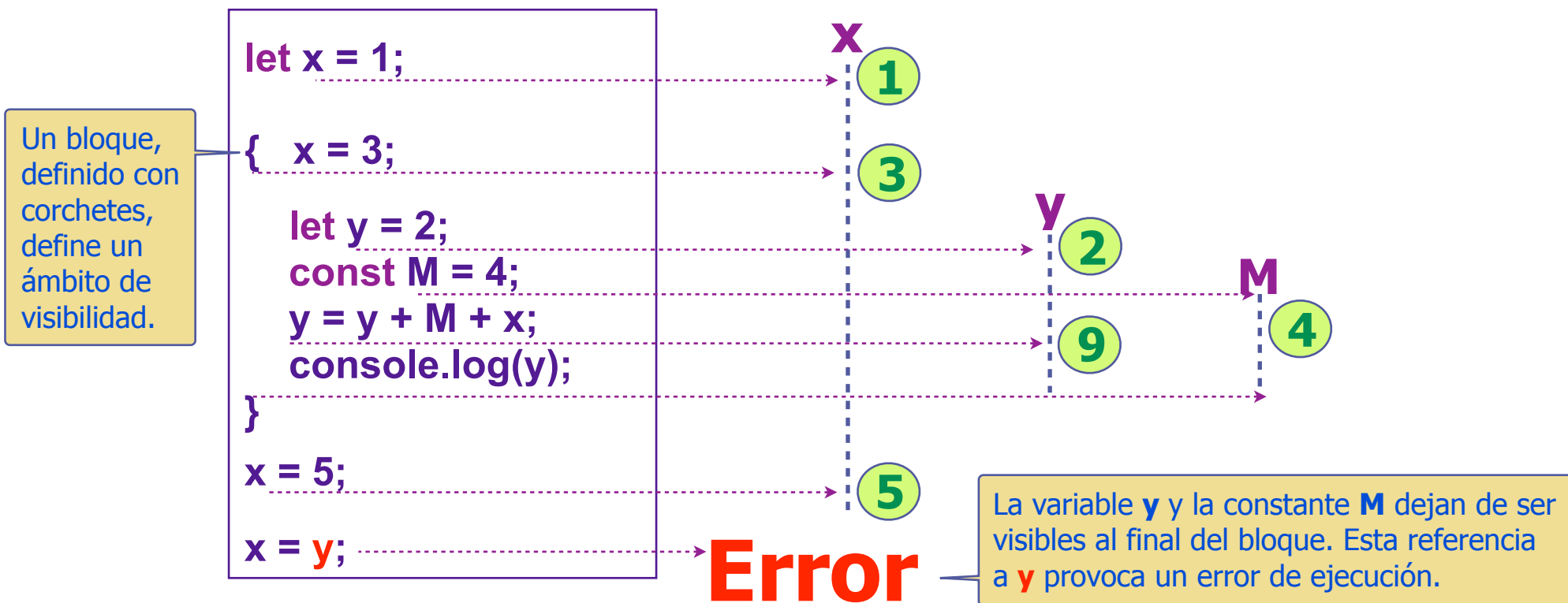
- ◆ La declaración de variables con **var** pertenece a las partes malas de JavaScript
 - son visibles en todo el ámbito del programa, incluso antes de haberlas definido
 - ◆ Esta característica se denomina hoisting y es muy engañosa
 - Por ejemplo, las variables **z** y **t** del ejemplo son visibles antes de declararse
 - ◆ Las variables **z** y **t** contienen **undefined** antes de declararlas y el **valor asignado** después
- ◆ **var**, **let** y **const** pueden mezclarse en un programa JavaScript
 - variables declaradas con **var** se mantienen por compatibilidad hacia atrás
 - ◆ Pero solo se deben utilizar variables o constantes declaradas con **let** o **const**



Ámbito de visibilidad de variables ES6

◆ Variables y constantes ES6

- Solo existen desde el momento en que se declaran
 - ◆ Invocarlas antes de ser declaradas provoca un error de ejecución
- Son visibles solo en los bloques donde se declaran
 - ◆ En el exterior dejan de ser visibles

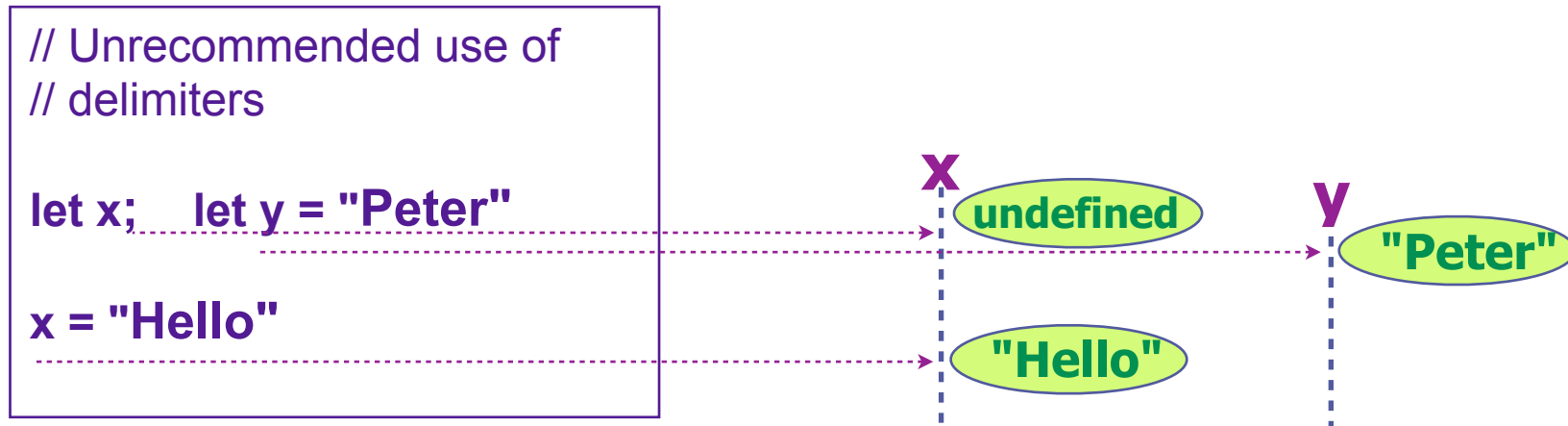


Sintaxis: variables

- ◆ 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_sintaxis e interrumpe el programa
- ◆ Un nombre de variable **no** debe ser una **palabra reservada** de JavaScript
 - por ejemplo: **var**, **function**, **return**, **if**, **else**, **while**, **for**, **new**, **typeof**, ...
- ◆ Las variables son sensibles a **mayúsculas**
 - **mi_var** y **Mi_var** son variables distintas

Recomendaciones sobre sintaxis

- ◆ La delimitación de final de sentencia JavaScript se considera mal diseñada
 - El final de una sentencia se indica en JavaScript con **punto y coma (;)**
 - ◆ JavaScript permite terminar sentencias con nueva línea con **nueva línea (\n)**, si no provoca ambigüedad
 - Para evitar ambigüedades **se recomienda** terminar **todas** las sentencias con **punto y coma (;)**
 - Recomendaciones de uso de punto y coma (;): <https://www.codecademy.com/blog/78>
 - ◆ **JSLint** (o **ESLint**) verifican que se hace el uso recomendado: <http://www.jshint.com/> <http://eslint.org/>
- ◆ Una regla sencilla (aunque simplista) es:
 - Las sentencias deben **ocupar una línea** y acabar con **punto y coma (;)**
 - ◆ salvo las sentencias con **bloques de código** (if/else, while, for, definición de funciones, ...)
 - O expresiones muy largas que no caben en una línea





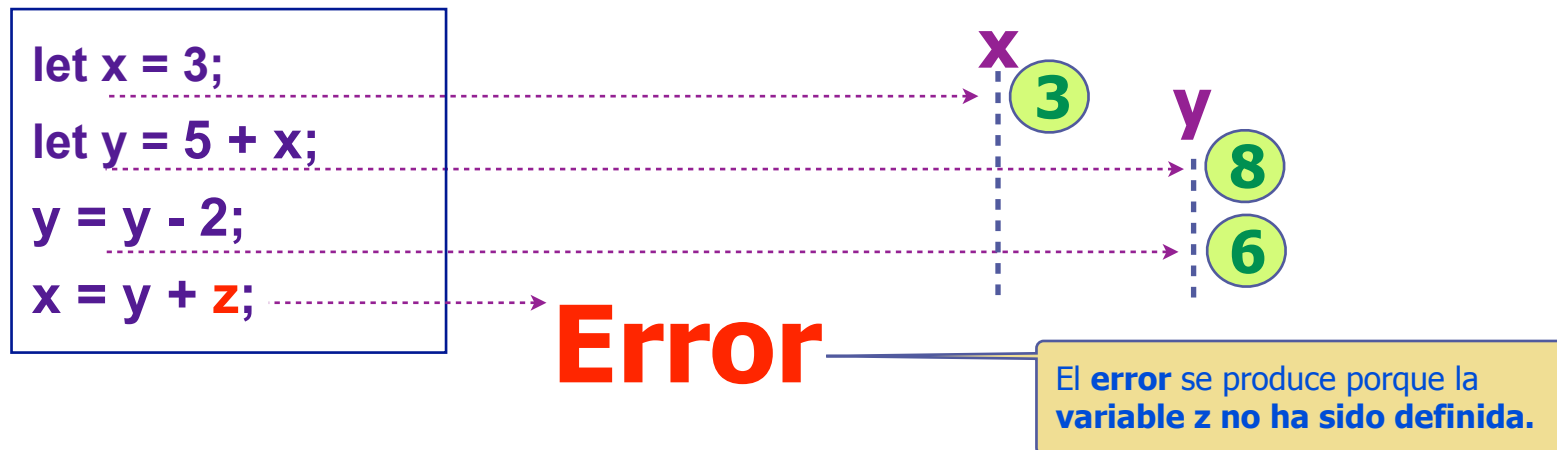
JavaScript

Expresiones con variables, operadores
++, --, +=, *=, -=, /=, %=, etc.

Juan Quemada, DIT - UPM

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 asignada a ella misma
 - La parte derecha usa el valor anterior a la ejecución de la sentencia
 - ◆ Si **y** tiene un valor **8**, entonces **y = y - 2** asigna el valor **6 (8-2)** a la variable **y**
- ◆ Usar una variable no definida en una expresión
 - provoca un **error** y la ejecución del programa se **interrumpe**



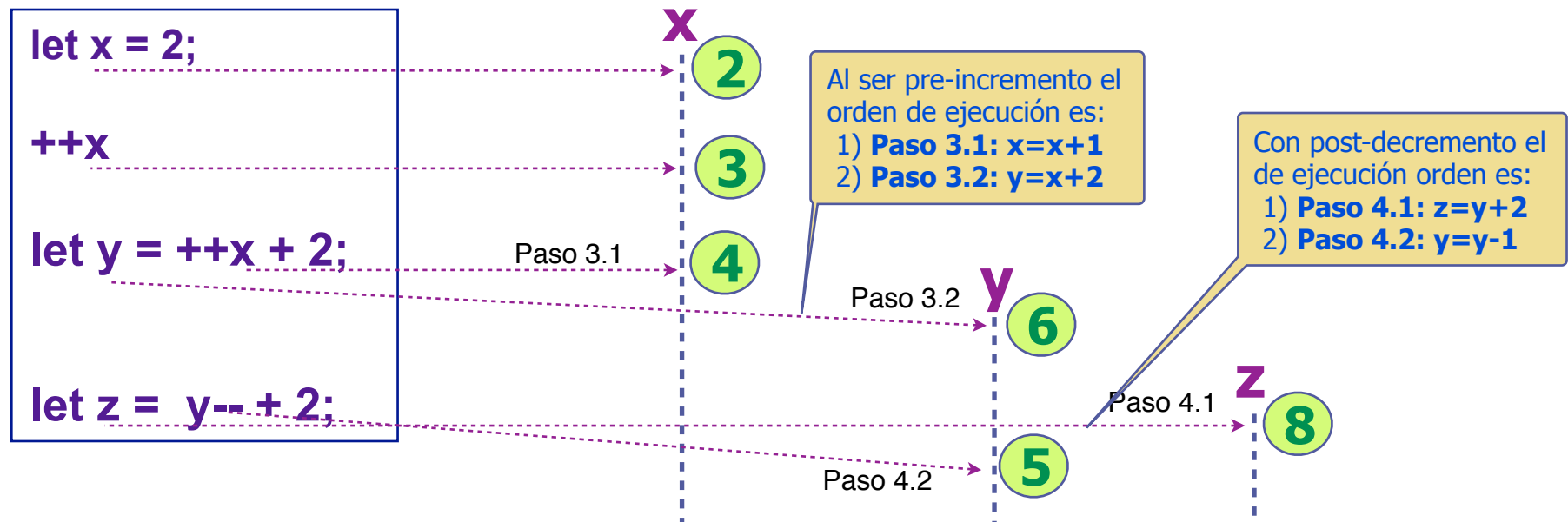
Pre y post auto incremento o decremento

◆ JavaScript posee los operadores ++ y -- de **auto-incremento** o **decremento**

- ++ suma 1 y -- resta 1 a la variable a la que se aplica
 - ◆ ++ y -- se pueden aplicar por la derecha o por la izquierda a las variables de una expresión
 - Si ++/-- se aplica por la **izquierda** a la variable (**pre**), el incremento/decremento se realiza **antes** de evaluar la expresión
 - Si ++/-- se aplica por la **derecha** (**post**) se incrementa/decrementa **después** de evaluarla
- **Ojo!** Usar con cuidado, sus efectos laterales llevan a programas difíciles de entender.

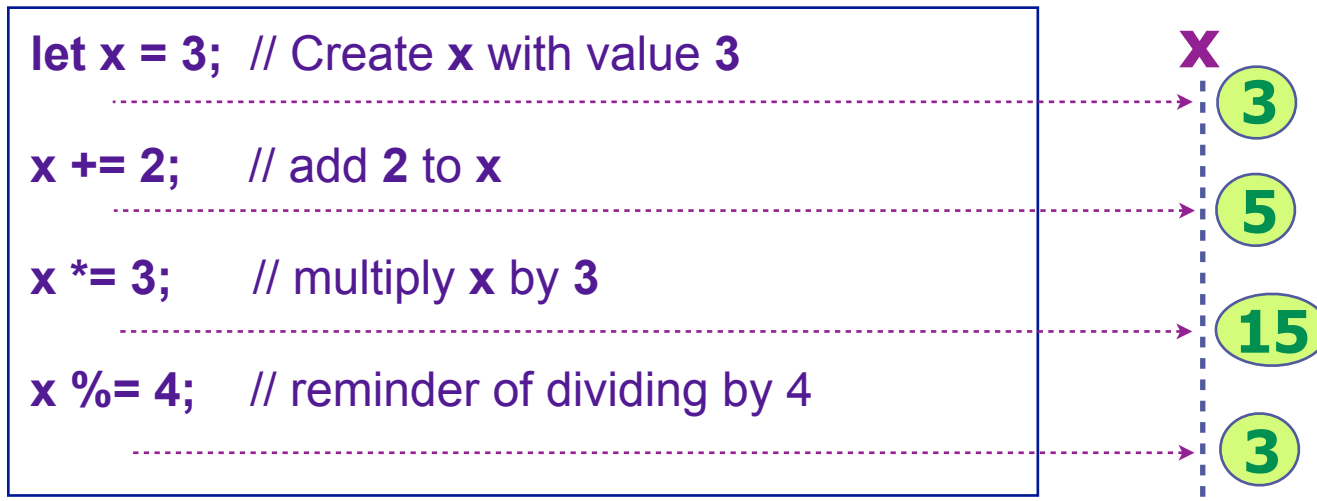
◆ Documentación adicional

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



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>





JavaScript

Funciones, array arguments, valores por defecto y operador spread

Juan Quemada, DIT - UPM

Función

Definición de la función

```
function my_preferred_movies () {  
  console.log();  
  console.log("My preferred movies:");  
  console.log(" - Jurassic Park by Steven Spielberg (1993)");  
  console.log(" - King Kong by Merian C. Cooper (1933)");  
  console.log(" - Citizen Kane by Orson Wells (1941)");  
  console.log();  
}
```

my_preferred_movies();

Invocación (ejecución) de la función

```
_ $  
_ $ node 10_function_movies.js  
  
My preferred movies:  
- Jurassic Park by Steven Spielberg (1993)  
- King Kong by Merian C. Cooper (1933)  
- Citizen Kane by Orson Wells (1941)  
  
_ $
```

Ejecución del programa
10_function_movies.js
con node.

◆ Una **función** encapsula código y lo representa por un **nombre**

- Una función debe definirse primero, para poder invocarla (ejecutarla) posteriormente
 - ◆ Documentación: <https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Funciones>

◆ La **definición** de la función comienza por la palabra reservada: **function**

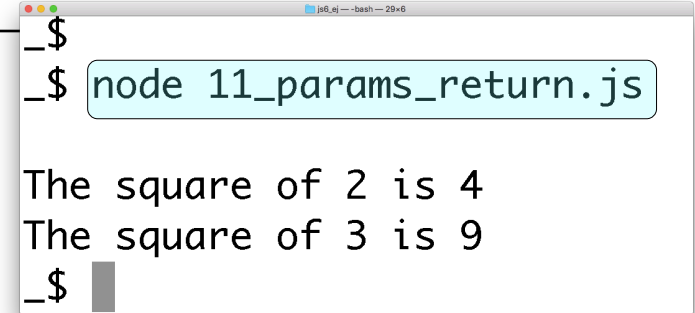
- A continuación viene el **nombre** de la función, que debe ser único en el programa
 - ◆ En tercer lugar vienen los parámetros entre paréntesis: () indica sin parámetros en este ejemplo
 - Por último viene el bloque de código, entre corchetes { }

◆ La **invocación** de la función ejecuta el bloque de código de la función

- Se invoca con el nombre y el operador paréntesis (), por ej. **my_preferred_movies()**

Parámetros de invocación y de retorno

```
function square (x) {  
  return x*x ;  
}  
  
console.log();  
console.log("The square of " + 2 + " is " + square(2));  
console.log("The square of " + 3 + " is " + square(3));
```



A terminal window titled 'js@ej: ~ — bash — 20x6' showing the execution of a Node.js script. The prompt is '_\$'. The command 'node 11_params_return.js' is entered and executed. The output is 'The square of 2 is 4' followed by 'The square of 3 is 9'. The prompt returns to '_\$'.

- ◆ Una **función** recibe parámetros de entrada (parámetro **x** del ejemplo)
 - Y devuelve un valor con la sentencia: **return <expr>**
 - ♦ Esta sentencia finaliza la ejecución de la función y devuelve el valor resultante de evaluar **<expr>**
 - Si la **función** llega a final del bloque **sin ejecutar** return, finaliza y devuelve **undefined**
- ◆ Un **parámetro** de una función es similar a la definición de una **variable**
 - El parámetro solo es **visible** dentro del **bloque de la función**
 - ♦ El parámetro se **inicia** con el **valor pasado al invocar la función**, en el ejemplo con los valores 2 y 3
- ◆ Una función puede usarse en expresiones como otro valor más
 - La función se ejecutará y se sustituirá por el valor devuelto en la expresión
 - ♦ En el ejemplo (return **x*x** ;) devuelve el **cuadrado** del valor pasado en el parámetro **x**

Número de parámetros de una función

Antes de ES6 los strings se concatenaban así:
greeting + " " + person + ", how are you?"
(es prácticamente equivalente, pero menos compacto)

```
function greet (greeting, person) {  
  return `${greeting} ${person}, how are you?`;  
};  
  
greet ("Good morning", "Peter");    // => "Good morning Peter, how are you?"  
  
greet ("Hi", "Peter");              // => "Hi Peter, how are you?"  
  
greet ("Hi", "Peter", "Bill");      // => "Hi Peter, how are you?"  
greet ("Hi");                       // => "Hi undefined, how are you?"  
greet ();                           // => "undefined undefined, how are you?"
```

- ◆ Una función **se puede invocar** con un **número variable de parámetros**
 - Un parámetro definido, pero **no pasado** en la invocación, toma el valor **undefined**
 - ◆ Un parámetro pasado en la invocación, pero **no utilizado**, no tiene utilidad
- ◆ La función **greet(..)** genera un saludo utilizando 2 parámetros
 - El ejemplo ilustra como procesa JavaScript parámetros no pasados o no utilizados

arguments: el array con los parámetros

```
function greet () {  
    return `${arguments[0]} ${arguments[1]}, how are you?`;  
};  
  
greet ("Good morning", "Peter");    // => "Good morning Peter, how are you?"  
  
greet ("Hello", "Peter");           // => "Hello Peter, how are you?"
```

- ◆ Una función tiene predefinida un array de nombre **arguments**
 - **arguments** contiene los valores asignados a los parámetros en la invocación
 - ◆ Aquí se define la función **greet** utilizando **arguments** en vez de parámetros explícitos
 - Por último viene el bloque de código, entre corchetes {...}
- ◆ Una función se puede invocar con un número variable de parámetros
 - El array **arguments** permite saber su número y acceder a todos

Resto de parametros en ES6: ...x

- ◆ Operador spread (...x) da acceso al resto de los parámetros de una función en ES6
 - Los parámetros están accesibles a través del array asociado al operador
 - ♦ Parámetros explícitos y operador rest pueden mezclarse entre sí, por ejemplo: **function f1 (x, y, ...resto) {....}**
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator
- ◆ los ejemplos muestran 2 definiciones equivalentes de la función greet

```
function greet (...args) {  
  return `${args[0]} ${args[1]}, how are you?`;  
};
```

```
greet ("Good morning", "Peter"); // => "Good morning Peter, how are you?"  
greet ("Hello", "Peter");       // => "Hello Peter, how are you?"
```

```
function greet (greeting, ...more) {  
  return `${greeting} ${more[0]}, how are you?`;  
};
```

```
greet ("Good morning", "Peter"); // => "Good morning Peter, how are you?"  
greet ("Hello", "Peter");       // => "Hello Peter, how are you?"
```

Valores por defecto de parámetros (ES6)

```
function greet (greeting = "Hi", person = "my friend") {  
    return `${greeting} ${person}, how are you?` ;  
};  
  
greet ("Hello");           // => "Hello my friend, how are you?"  
greet ();                  // => "Hi my friend, how are you?"
```

- ◆ **ES6** permite asignar valores por defecto a parámetros de funciones
 - Los valores por defecto se asignan al parámetro en la definición
 - ◆ utilizando el operador =, como en las definiciones de variables
- ◆ El valor por defecto se utiliza si la invocación no incluye ese parámetro



Funciones como objetos, notación flecha, ámbitos de visibilidad y cierres

Juan Quemada, DIT - UPM

Funciones como objetos

◆ Las **funciones** son objetos de pleno derecho

- pueden **asignarse a variables**, a **propiedades**, pasarse como **parámetros**,

◆ **Literal de función:** function (<argumentos>){<sentencias>}

- Construye un objeto de tipo función que no tiene nombre
 - ◆ Puede guardarse en variables o parámetros como cualquier otro valor
 - Se invoca aplicando el operador paréntesis: ()

◆ **El operador paréntesis, (),** invoca un objeto function ejecutando su código

- Este operador solo es aplicable a funciones (objetos de la clase Function), sino da error
 - ◆ Se pueden incluir parámetros explícitos separados por coma, accesibles en el código de la función

```
let greet = function (greeting, person) {  
    return `${greeting} ${person}, how are you?` ;  
};  
  
greet ("Hi", "Peter");           // => "Hi Peter, how are you?"  
  
let x = greet;  
  
x ("Hi", "Peter");               // => "Hi Peter, how are you?"
```

Notación flecha (arrow) de ES6

◆ ES6 añade la notación flecha a JavaScript, por ej.: **const add = (x,y) => x+y**

- Esta notación permite crear funciones sin nombre, con las siguientes diferencias
 - ♦ El objeto **this** tiene vinculación léxica (y no dinámica) con su entorno, como las variables
 - ♦ No pueden ser constructores de objetos
 - ♦ No tienen la variable predefinida **arguments**, que accede a los parámetros de la invocación
 - El operador **spread** (...) de ES6 hace **arguments** innecesario, como se ve más adelante
- La notación flecha es concisa y además corrige el problema de ámbito de **this**
 - ♦ Dado el uso tan habitual de la programación funcional en JavaScript es conveniente utilizarla
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

```
const greet = function (greeting, person) {           // defined with function literal
  return `${greeting} ${person}, how are you?` ;
};
```

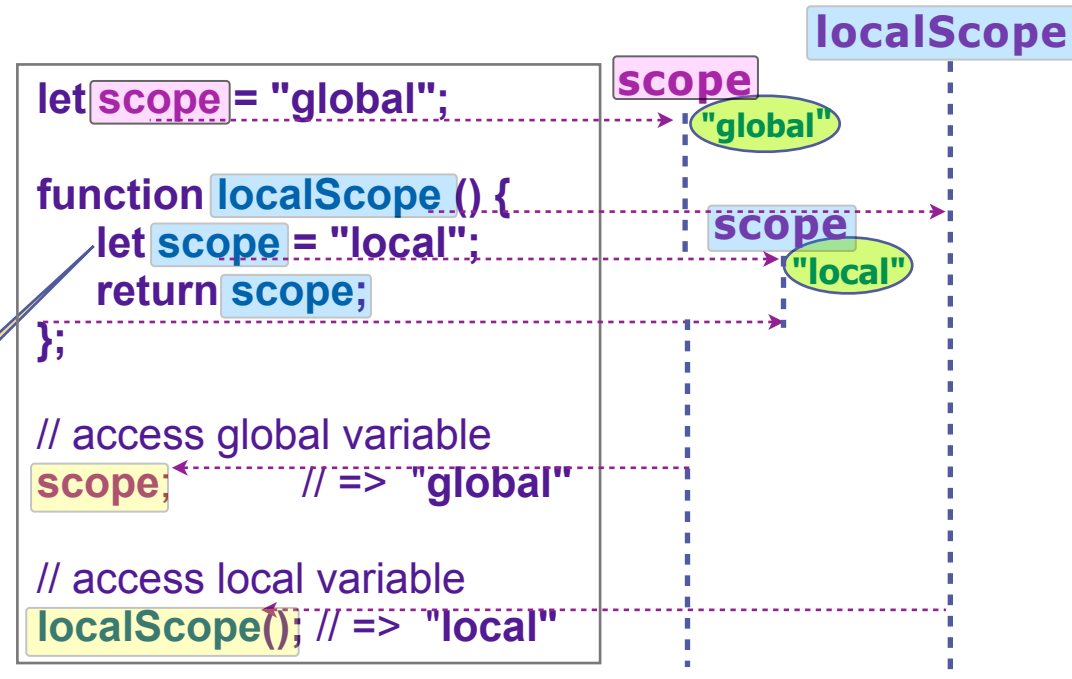
```
                                     // The same function defined with arrow notation
const greet = (greeting, person) => {
  return `${greeting} ${person}, how are you?` ;
};
```

```
                                     // Parenthesis may be omitted with only one parameter
let square = x => x*x; // Blocks with one instruction may omit curly brackets and return
```

```
const say_hi = () => "Hi, how are you?";           // function without parameters
```

Declaraciones locales de una función y ámbito

La variable local **scope** tapa la variable global del mismo nombre dentro de la función.



- ◆ Una **función** puede tener **declaraciones locales** de variables y funciones
 - Las declaraciones son **visibles solo dentro de la función**
- ◆ Las **variables y funciones** tienen **visibilidad sintáctica** en JavaScript
 - Son **visibles solo dentro del bloque de código** de la función donde se declaran
 - ◆ **OJO!** Si están declaradas con **var** serán visibles en el bloque de la función antes de su declaración
- ◆ **Variables y funciones globales** son **visibles** también **dentro de la función**
 - Siempre que no sean **tapadas** por otras declaraciones locales del **mismo nombre**
 - ◆ Una declaración **local tapa** a una **global** del **mismo nombre**

Funciones anidadas

◆ Una función puede tener **declaraciones locales** definidas en su **interior**

- La variable **s2** y la función **inner** son declaraciones **locales** de la función **external**
 - ◆ Son visibles solo dentro de la función **external**

◆ La variable **s3** es local de la función **inner**

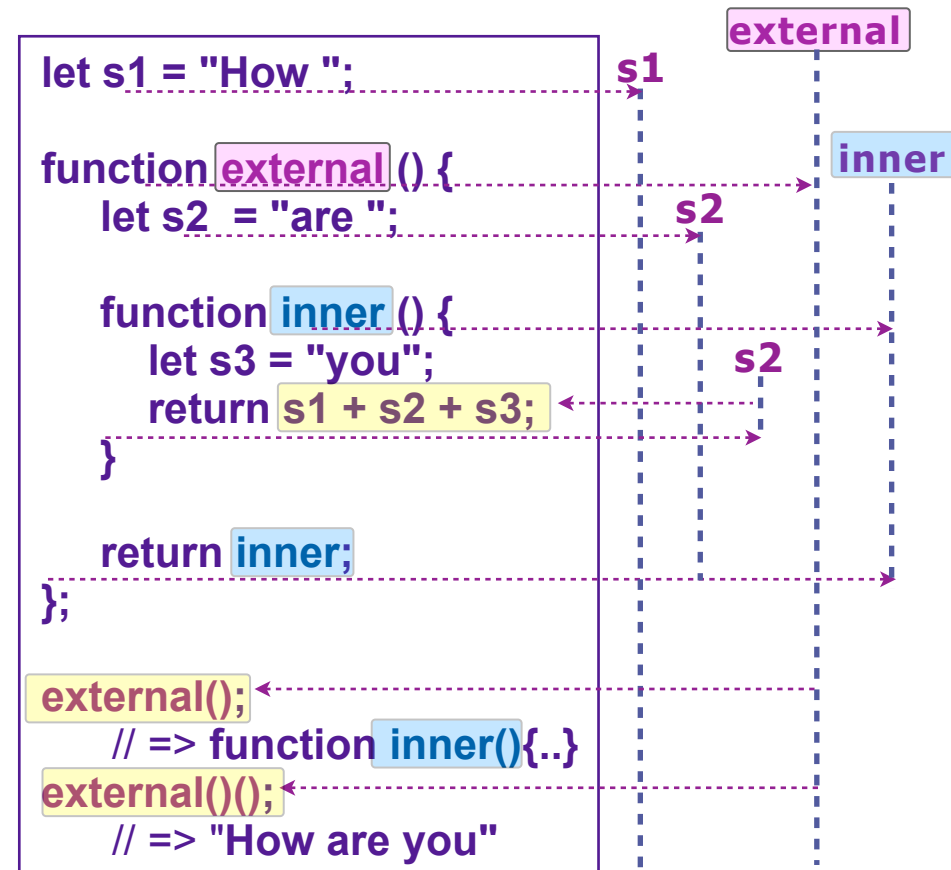
- **s3** solo es visible dentro de la función **inner**
 - ◆ En cambio, debido a la visibilidad sintáctica **s1**, **s2** y **s3** son visibles dentro de la función **inner**

◆ Como una **función es un objeto**

- también puede **devolverse** como **retorno** de otra función
 - ◆ La función **external** devuelve la función **inner** como retorno

◆ Como la función **external** devuelve la función **inner**

- **external** es una referencia a la función exterior
- **external()** devuelve una referencia a la función interior **inner**
- **external()()** ejecuta la función interior **inner** (equivale a **inner()**)
 - ◆ Como las **variables s1, s2 y s3** son visibles dentro de **inner**, esta puede concatenar **s1+s2+s3**



Cierres (o closures)

◆ Cierre (o closure)

- Un cierre crea un **entorno de ejecución aislado** del exterior utilizando funciones
 - ◆ Estos entornos son parecidos a **módulos** o a **abstracciones de datos**
- El **cierre** es la **función** que aísla en su interior el **entorno de ejecución**
 - ◆ La **función cierre** devuelve el **objeto interfaz** que da acceso a dicho entorno
- El **estado de las variables locales** de la función cierre se **mantiene** entre usos
 - ◆ La ejecución de la función cierre crea las variables y funciones locales que permanecen

◆ **uniqueInteger** (función cierre del ejemplo) devuelve como interfaz la función **count**

- El entorno de ejecución es accesible por la función **count**
 - ◆ La variable **_counter** solo puede ser modificada y leída por la función **count()**
 - Invocar **unique()** es invocar **count()**, que incrementa **_counter** y devuelve su valor al exterior
 - ◆ El **cierre** y su variable **_counter** existirá mientras **función interfaz count** este guardada en alguna variable

```
function uniqueInteger () {  
  var _counter = 0;  
  function count () { return _counter++; };  
  return count;  
};
```

// assigns return value: function count
var **unique** = uniqueInteger **()**;

unique() // => 0 invoques count()
unique() // => 1 invoques count()

// Equivalent definition to previous one,
// without giving name to interface function

```
var unique = function () {  
  var _counter = 0;  
  return function () { return _counter ++; };  
} ();
```

unique() // => 0
unique() // => 1

() invoca el cierre, que devuelve la función **count()** que se asigna a **unique**.



Boolean, operadores lógicos (!, &&, ||), de comparación (===, !==, <, >, >=, <=) y operador .. ? .. : ..

Juan Quemada, DIT - UPM

Tipo booleano

◆ El tipo **boolean** tiene 2 valores

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

◆ Los booleanos permiten **tomar decisiones**

- En sentencias condicionales: **If/else**, **bucles**, etc.

`((x % 2) === 0)` comprueba si **x** es par (resto de dividir 2 es 0).

```
const even = x => (x % 2) === 0;

console.log('12 is even? => ' + even(12));
console.log('5 is even? => ' + even(5));
```

```
_ $
_ $ node 20-bool_even.js
12 is even? => true
5 is even? => false
_ $
```

```
const between_0_4 = x => 0 <= x && x <= 4;

console.log('3 between 0 and 4? => ' + between_0_4(3));
console.log('8 between 0 and 4? => ' + between_0_4(8));
```

Las comparaciones resultan en un booleano:

- **comparación** de **orden**

menor: < menor_o_igual: <=

mayor: > mayor_o_igual: >=

- **comparación** de **identidad**

identidad: === no_identidad: !==

Los **operadores lógicos** booleanos son:

negación: ! !true => false
 !false => true

operador y: && true && true => true
 true && false => false
 false && true => false
 false && false => false

operador o: || true || true => true
 true || false => true
 false || true => true
 false || false => false

```
_ $
_ $ node 21-bool_between.js
3 between 0 and 4? => true
8 between 0 and 4? => false
_ $
```


Conversión a boolean

- ◆ Los tipos primitivos se convierten a boolean así:
 - **0, -0, NaN, null, undefined, "", "", ``** se convierten a **false**
 - **resto de valores** se convierten a **true**
- ◆ El operador negación (!) convierte primero a booleano
 - Una vez convertido a booleano calcula la negación
- ◆ La función Boolean(...) y la doble negación **!!<valor>**
 - convierten otros valores a booleanos

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

!!""	=> false
!!4	=> true
Boolean("")	=> false
Boolean(4)	=> true

Operadores de identidad e igualdad

◆ Identidad o igualdad estricta: **===**

- determina si **2 valores** son **exactamente los mismos**
 - ◆ Es igualdad semántica solo en: **number**, **boolean**, **strings** y **undefined**
 - ◆ **OJO!** En objetos es identidad de referencias (punteros)
- La identidad determina igualdad de tipo y de valor

◆ Desigualdad estricta: **!==**

- negación de la igualdad estricta

◆ Igualdad y desigualdad débil: **==** y **!=**

- **OJO!** No debe utilizarse
 - ◆ Realiza conversiones difícilmente predecibles

◆ Mas info:

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

Tipos básicos: identidad

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

Operadores de comparación

◆ JavaScript tiene cuatro operadores de comparación

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

◆ Las comparaciones se suelen utilizar:

- con **números** y con **strings**
 - ◆ La relación de orden está bien definida en ambos casos

◆ No se recomienda utilizar con otros tipos: **function**, **object**, ..

- La relación de orden es muy poco intuitiva en estos casos
 - ◆ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators

1.2 < 1.3 => true

1 < 1 => false

1 <= 1 => true

1 > 1 => false

1 >= 1 => true

false < true => true

"a" < "b" => true

"a" < "aa" => true

Operadores y (&&) y o (||)

◆ Operador lógico y (and): **<v1> && <v2>**

- devuelve <v1> o <v2> **sin modificarlos**
 - ◆ **<v1> && <v2>**
 - ◆ devuelve <v1> -> si <v1> es equivalente a **false**
 - ◆ devuelve <v2> -> en caso contrario

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

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

◆ Operador lógico o (or): **<v1> || <v2>**

- devuelve <v1> o <v2> **sin modificarlos**
 - ◆ **<v1> || <v2>**
 - ◆ devuelve <v1> -> si <v1> es equivalente a **true**
 - ◆ devuelve <v2> -> en caso contrario

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

```
undefined || 1   => 1
13 || 1          => 13
```

Operador condicional: ?:

◆ El operador condicional: ?:

- devuelve un valor en función de una condición lógica
 - ◆ Es una **versión más funcional** del operador **if/else**

◆ Sintaxis: **condición ? <v1> : <v2>**

- devuelve **<v1>** -> si **condición** es equivalente a **true**
- devuelve **<v2>** -> en caso contrario

true ? 1 : 7	=> 1
false ? 1 : 7	=> 7

7 ? 1 : 7	=> 1
"" ? 1 : 7	=> 7

◆ En ES5 los parámetros por defecto se definen con este operador

- También con **"x || <param_por_defecto>"**, pero es incorrecto porque aplicaría a **""**, **0**, **null**, ..

```
function greet (greeting = "Hi", person = "my friend") {           // Parámetros por defecto en ES6
  return `${greeting} ${person}, how are you?` ;
}

function greet (greeting, person) {                                  // Parámetros por defecto en ES5
  greeting = (greeting !== undefined) ? greeting : 'Hi';           // Ambos son equivalentes
  person    = (person    !== undefined) ? person : 'my friend';
  return `${greeting} ${person}, how are you?` ;
};

greet ("Hello");           // => "Hello my friend, how are you?"
greet ();                  // => "Hi my friend, how are you?"
```



Decisiones: if...else y switch...case

Juan Quemada, DIT - UPM

Decisiones: if...else y switch...case

◆ Sentencia **if...else**

- permite ejecutar código condicionalmente
 - ◆ en función de condiciones de tipo lógico
 - ◆ Es una sentencia muy versátil y muy usada
- Documentación
 - ◆ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else>

◆ Sentencia **switch...case**

- Ejecuta bloques de código asociados al resultado de una expresión
 - ◆ Una sentencia **switch...case** puede sustituirse por un **if...else** encadenado
- Documentación
 - ◆ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/switch>

Sentencia if...else

La **condición (hora <12)** va entre paréntesis y según se evalúe a **true** o **false**, decide si se ejecuta el primer o el segundo bloque.

new Date().getHours() devuelve la hora del día (0-23).

El **primer bloque** de sentencias va después de la condición, delimitado entre llaves: **{ }**

La sentencia **if/else** comienza con la palabra reservada **if**

```
let hour = new Date().getHours();  
  
if (hour < 12) {  
    console.log(`\n Good morning, it's ${hour} hours`);  
} else {  
    console.log(`\n Good afternoon, it's ${hour} hours`);  
}
```

\n representa el carácter nueva línea en un string.

El **segundo bloque** de sentencias va precedido por la palabra reservada **else** y delimitado entre llaves: **{ }**

```
.$  
.$ node 25-if-else.js  
  
Good afternoon, it's 20 hours  
.$
```


Sentencia if

Este programa es equivalente al anterior, pero con diferente estructura. No utiliza la parte **else** (opcional). En cambio añade la variable **var saludo = "\n Buenas tardes"**

La **sentencia if** tiene ahora solo la primera parte. Esta cambia el contenido asignado a la variable por **saludo = "\n Buenos días"**

```
.$  
.$ node 26-if-only.js  
  
Good afternoon, its 20 hours  
.$
```

```
let hour = new Date().getHours();  
let greeting = "\n Good afternoon";  
  
if (hour < 12) {  
  greeting = "\n Good morning";  
}  
  
console.log(`${greeting}, its ${hour} hours`);
```

El mensaje enviado a consola se genera con las variables **saludo** y **hora**.

Sentencias if...else encadenadas

Las **sentencias if...else** pueden encadenarse para comprobar múltiples condiciones en cascada (de las cuales solo se ejecutará una), tal y como se hace en este ejemplo donde también se comprueba si se debe decir "Buenas noches".

```
js6_ej -- bash -- 30x5
.$
.$ node 27-if-else-if.js
Good afternoon, its 20 hours
.$
```

```
let hour = new Date().getHours();
let greeting;

if (hour < 12) {
  greeting = "\n Good morning";
} else if (hour < 21) {
  greeting = "\n Good afternoon";
} else {
  greeting = "\n Good night";
}

console.log(` ${greeting}, its ${hour} hours`);
```

Sentencia switch...case

Math.round(Math.random()*10) genera un número aleatorio entre 0 y 9.

La sentencia **switch...case** evalúa la expresión asociada (result) y pasa a ejecutar la sentencia justo después del "**case**" que coincide con el valor resultante de evaluar la expresión.

La sentencia **break** finaliza la ejecución de la sentencia **switch...case**. Es necesario ponerla para finalizar cada **case** (o conjunto de **case**), porque sino continuará ejecutando las sentencias del siguiente **case**.

Cuando el valor resultante de evaluar la expresión no coincide con ninguno asociado a un **case**, se pasa a ejecutar las sentencias asociadas a **default**:

```
// Math.round(Math.random()*10):  
// entero aleatorio entre 0 y 9
```

```
let result = Math.round(Math.random()*10);
```

```
switch (result) {
```

```
  case 9:
```

```
    console.log("\n You win the first prize!");
```

```
    break;
```

```
  case 8:
```

```
  case 7:
```

```
    console.log("\n You win the second prize!");
```

```
    break;
```

```
  default:
```

```
    console.log("\n Sorry, no prize!");
```

```
}
```

Varios **case** se pueden agrupar si comparten el mismo código. **case** es solo un punto de comienzo de ejecución.

```
$  
$ node 28-case.js  
  
You win the first prize!  
$  
$ node 28-case.js  
  
Sorry, no prize!  
$  
$ node 28-case.js  
  
Sorry, no prize!  
$
```



Bucles: while, for, do...while, break
y continue

Juan Quemada, DIT - UPM

Bucle while

Iniciación del bucle:

n contiene el exponente

res se inicia a 1, al finalizar el bucle contendrá el resultado.

Condición de permanencia,
delimitada por **()**

Bloque de acciones, delimitado por **{ }**

```
let n = 3;    // 2^n
let res = 1;
```

```
while (0 < n) {
    res = res * 2;
    n = n - 1;
}
```

```
let z = res - 2;
```

◆ Bucle: **bloque de instrucciones que se repite**

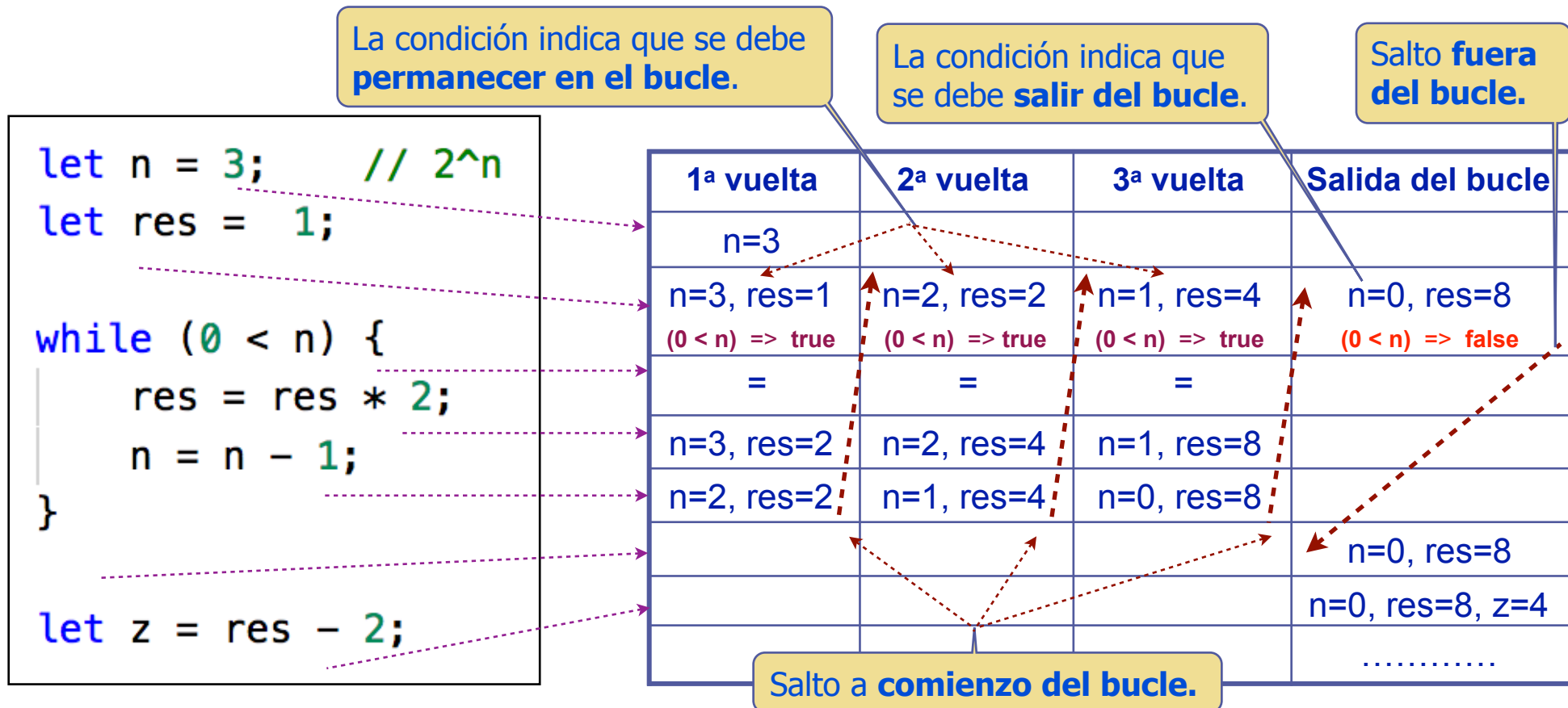
- mientras se cumple una **condición** de permanencia
 - ◆ Las variables que controlan el bucle deben iniciarse antes de comenzar
- Lo ilustramos con un bucle **while** que calcula 2^n ($2*2*...*2$ n veces)
 - ◆ Además existen otros tipos de bucles que no vemos aquí: **for**, **for/in**, **do/while**, ...
 - ◆ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration

◆ Un bucle tiene 3 partes

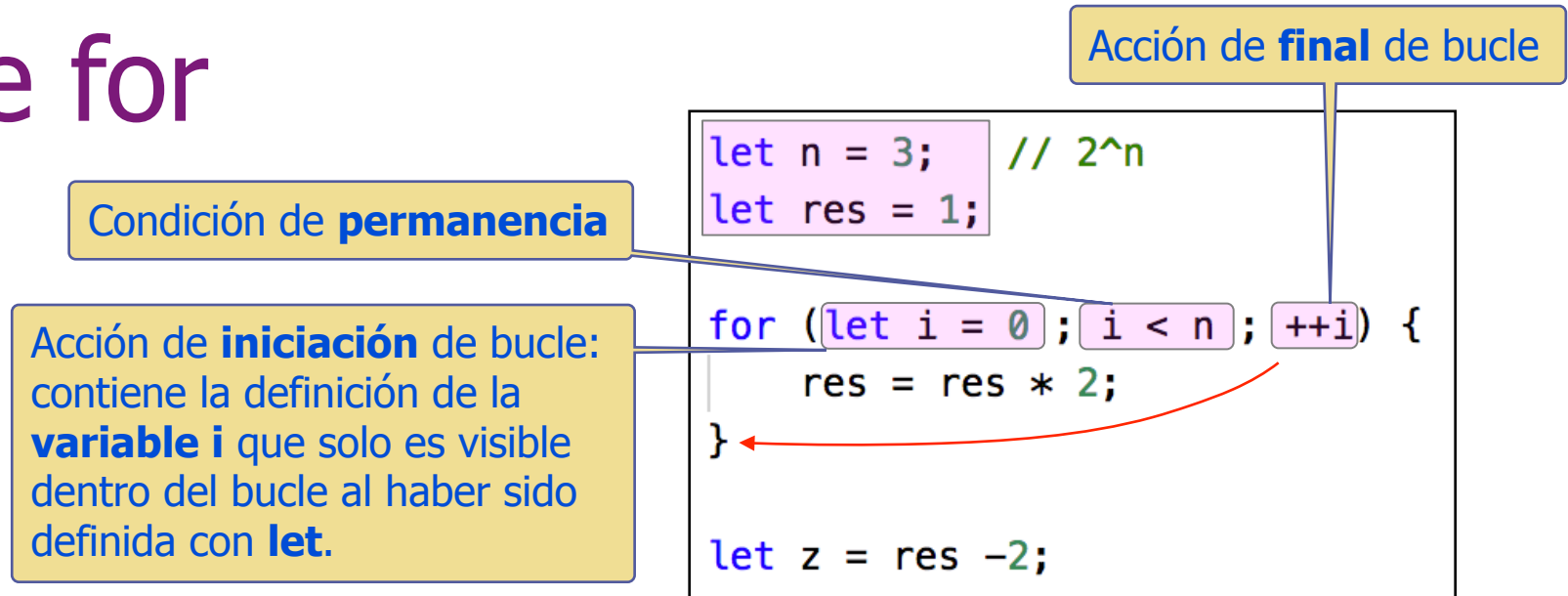
- **Iniciación:** fija los valores de arranque del bucle en la 1ª iteración
 - ◆ La iniciación se realiza aquí en instrucciones anteriores a la sentencia del bucle
- **Condición de permanencia:** controla la finalización del bucle
 - ◆ El bucle se ejecuta mientras la condición sea **true**
- **Bloque de acciones:** acciones realizadas en cada iteración del bucle
 - ◆ Realiza el cálculo de forma iterativa hasta que la condición de permanencia indica salir del bucle

Ejecución del bucle: evolución del estado

- ◆ Ejecución del bucle: se controla por la condición de permanencia: ($0 < n$)
 - Permanecerá en el bucle, si n es mayor que 0 y si no saldrá
 - ◆ La condición de permanencia depende del estado del programa (valor de la variable n)



Bucle for

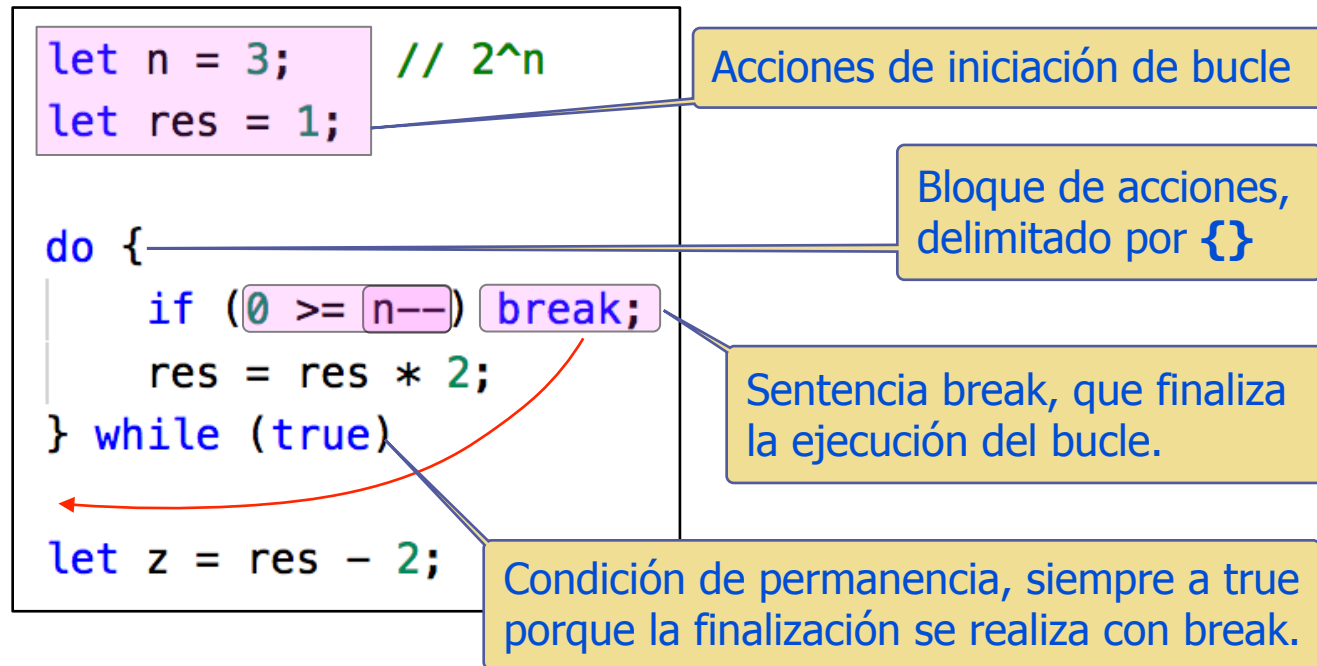


Este programa es equivalente al anterior, pero utilizando **sentencia for** que es mas compacta que while. La gestión del bucle (entre paréntesis) va detrás de la palabra reservada **for** y consta de tres partes separadas por ";":

- 1) **Iniciación**: define e inicia la variable "i" con una sentencia let, para que no sea visible fuera del bucle.
- 2) **Condición de permanencia**: el bucle se ejecuta mientras la condición sea **true**. Se sale del bucle cuando la condición pase a **false** (`i < n`). equivale a la condición de permanencia de while.
- 3) **Acción final del bucle**: se ejecuta al final de cada ejecución del bloque de código, después de la última instrucción. `++i` incrementa `i` y 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, como en cualquier otro bloque que contenga solo una sentencia.

Bucle do...while y sentencia break



La sentencia **do...while** comprueba la condición de permanencia al final del bucle, por lo que el bloque se ejecuta por lo menos una vez. La sintaxis de esta sentencia comienza por la palabra reservada **do**, continua con el **bloque** delimitado por {...} y finaliza con la palabra **while**, seguida por la condición de permanencia entre paréntesis.

La estructura de este programa es similar al del bucle while anterior, pero la condición de permanencia es siempre verdadera. Esto es así, porque la finalización de bucle se controla con una sentencia **if** con la **condición de salida** de bucle, que ejecuta la **sentencia break**, cuando esta se cumple. La sentencia **break** finaliza la ejecución de un bucle, pasando a ejecutar la sentencia siguiente al bucle.

Todos los bloques de acciones de sentencias JavaScript pueden omitir las llaves {}, cuando solo contienen una sentencia. Por eso la sentencia if omite las llaves.

Etiquetas y sentencia continue

```
_ $  
_ $ node 32-while_while.js  
2^0 = 1  
2^1 = 2  
2^2 = 4  
2^3 = 8  
2^4 = 16  
2^5 = 32  
2^6 = 64  
2^7 = 128  
2^8 = 256  
2^9 = 512  
2^10 = 1024  
_ $
```

```
let i, n = 0, res;  
const MAX = 10;  
  
while (n <= MAX) {  
  res = 1;  
  i = n;  
  while (0 < i--) {  
    res *= 2;  
  }  
  console.log("2^" + n++ + " = " + res);  
}
```

Esta transparencia presenta dos programas equivalentes que muestran una tabla de potencias de dos. El primer ejemplo realiza el calculo con dos bucles while convencionales. El bucle exterior muestra una nueva fila en cada iteración y el interior calcula la potencia de dos correspondiente.

El segundo ejemplo utiliza la etiqueta num asociada al bucle exterior para forzar directamente desde el bucle interior, tanto la finalización del bucle interior, como una nueva iteración del bucle exterior.

```
let i, n = 0, res;  
const MAX = 10;  
  
num: while (n <= MAX) {  
  res = 1;  
  i = n;  
  while (true) {  
    if (0 >= i--) {  
      console.log("2^" + n++ + " = " + res);  
      continue num;  
    }  
    res *= 2;  
  }  
}
```

Etiqueta que identifica bucle

Sentencia **continue** que arranca nueva iteración de bucle exterior identificado por la **etiqueta num**.

STATEMENT SINTAX

block	{ statements ;}
break	break [label];
case	case expression:
const	const name [= expr] [,...];
continue	continue [label];
debugger	debugger;
default	default:
do...while	do statement while(expression);
empty	;
expression	expression;
for	for(init; test; incr) statement
for...in	for (var in obj) statement
for...of	for (var in obj) statement
function	function name([param[,...]]) { body }
func_arrow	([param[,...]]) => { body }
if...else	if (expr) statement1 [else statement2]
label	label: statement
let	let name [= expr] [,...];
return	return [expression];
switch	switch (expression) { statements }
throw	throw expression;
try	try {statements} [catch { statements }] [finally { statements }]
strict	"use strict";
var	var name [= expr] [,...];
while	while (expression) statement
with	with (object) statement

DESCRIPCIÓN DE LA SENTENCIA JAVASCRIPT

Agrupar un bloque de sentencias (como 1 sentencia)
Salir del bucle o switch o sentencia etiquetada
Etiquetar sentencia dentro de sentencia switch
Declarar e inicializar una o mas constantes (ES6)
Salto a sig. iteración de bucle actual/etiquetado
Punto de parada (breakpoint) del depurador
Etiquetar sentencia default de sentencia switch
Alternativa al bucle while con condición al final
Sentencia vacía, no hace nada
Evaluar expresión (incluyendo asignación a variables)
Bucle: init: iniciación; test: condición; incr: acciones final bucle
Bucle for...in: Itera en los nombres de propiedades de obj
Bucle for...of: Itera en los elementos del objeto iterable obj (ES6)
Declarar una función llamada "name"
Definir un literal de función con visibilidad léxica (ES6)
Ejecutar statement1 o statement2
Etiquetar sentencia con nombre label
Declarar e inicializar una o mas variables (ES6)
Devolver un valor desde una función
Multi-opción con etiquetas "case" o "default"
Lanzar una excepción o error
Define un manejador de excepciones con el bloque catch que procesa las exceptions lanzadas (throw) en el bloque try, el bloque finally, si existe, se ejecuta siempre
Activar restricciones strict a scripts o funciones
Declarar e inicializar una o mas variables
Bucle básico con condición al principio
Extender cadena de ámbito (no recomendado)



Arrays, spread y métodos sort,
reverse, concat, join, indexOf, slice,
splice, push y pop

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

Arrays

◆ Array

- Es una colección ordenada de elementos
 - ◆ Se suele crear con el literal de array: `[7, 4, 2, 23]`
 - El operador corchetes agrupa elementos en arrays
- **`toString()`** devuelve un string con los elementos

◆ Los elementos de un array de tamaño n

- se acceden con un índice entre 0 y n-1
 - ◆ **`a[k]`** accede al elemento **k+1**

◆ **`a.length`** indica el tamaño del array

- Un array tiene un máximo de $2^{32} - 2$ elementos

◆ Cambiar **`length`** cambia el tamaño del array

- Por ejemplo, **`a.length = 2`** reduce el tamaño de **a** a 2
 - ◆ Quedando solo los dos primeros elementos

◆ El **operador spread** (**`...x`**) nuevo en ES6

- Inserta los elementos de un array en otro array

```
let a = [7, 4, 1, 23];
```

```
a[0]      => 7
```

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

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

```
a[3]      => 23
```

```
a.toString() => "7,4,1,23"
```

```
a.length   => 4
```

```
let a = [7, 4, 1, 23];
```

```
a.length = 2   => 2
```

```
a              => [7, 4]
```

```
let a = [7, 4, 1];
```

```
let b = [0, 0, ...a];
```

```
b              => [0, 0, 7, 4, 1]
```

```
b.length       => 5
```

Arrays con elementos heterogéneos

◆ Un array puede contener elementos de tipos diferentes

- números, strings, undefined, objetos, arrays, ...

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

a[0]	=> 1
a[1]	=> 'a'
a[2]	=> undefined
a[4]	=> [1, 2]

◆ Indexar elementos inexistentes devuelve undefined

- Esto incluye acceso a índices mayores que a.length
 - ◆ Cambiando la longitud del array reducimos su tamaño

```
let a = [0, 'a', undefined, , [1, 2]];
```

a[2]	=> undefined
a[3]	=> undefined
a[9]	=> undefined

◆ Un array puede contener arrays como elementos

- El operador indexación se puede aplicar n veces
 - ◆ Por ejemplo **a[2][1]** accede al elemento 1 del elemento 2 de a

```
let b = [0, [ 'a', [4, 3]], [1, 2]];
```

b[0]	=> 0
b[1][0]	=> 'a'
b[1][1][0]	=> 4
b[1][1][1]	=> 3
b[2][0]	=> 1
b[2][1]	=> 2

◆ Más documentación

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Métodos para ordenar, invertir, concatenar o buscar

◆ **sort()**

Estos métodos no modifican el array original, solo devuelven el resultado como parámetro retorno.

- devuelve el array ordenado

```
[1, 5, 3].sort() // => [1, 3, 5]
```

◆ **reverse()**

- devuelve el array invertido

```
[1, 5, 3].reverse() // => [3, 5, 1]
```

◆ **concat(e1, ..., en)**

- devuelve un nuevo array con **e1, ..., en** añadidos al final

```
[1, 5, 3].concat(9) // => [1, 5, 3, 9]  
[1, 5, 3].concat(9, 3) // => [1, 5, 3, 9, 3]
```

◆ **join(<separador>)**

- concatena elementos en un string
 - ◆ introduce <separador> entre elementos

```
[1, 5, 3, 7].join(';') // => '1;5;3;7'  
[1, 5, 3, 7].join('') // => '1537'
```

◆ **indexOf(elem, offset)**

- devuelve índice de primer **elem**
 - ◆ **offset**: comienza búsqueda (por defecto 0)

```
[1, 5, 3, 5, 7].indexOf(5) // => 1  
[1, 5, 3, 5, 7].indexOf(5, 2) // => 3
```

```
[1, 5, 3].concat(2).sort().reverse() // => [5, 3, 2, 1]
```

Los métodos encadenados aplican el segundo método sobre retorno del primero.

Extraer, modificar o añadir elementos al array

- ◆ **slice(i,j)**: devuelve la rodaja entre **i** y **j**
 - Índice negativo (**j**) es relativo al final
 - ◆ índice "-1" es igual a `a.length-2`
 - No modifica el array original
- ◆ **splice(i, j, e1, e2, ..., en)**
 - sustituye **j** elementos desde **i** en array
 - ◆ por **e1, e2, ..., en**
 - Devuelve rodaja eliminada
- ◆ **push(e1, ..., en)**
 - añade **e1, ..., en** al final del array
 - ◆ devuelve el tamaño del array (`a.length`)
- ◆ **pop()**
 - elimina último elemento y lo devuelve

```
[1, 5, 3, 7].slice(1, 2) => [5]
[1, 5, 3, 7].slice(1, 3) => [5, 3]
[1, 5, 3, 7].slice(1, -1) => [5, 3]
```

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

```
a.splice(1, 2, 9) => [5, 3]
a                  => [1, 9, 7]
```

```
a.splice(1,0,4,6) => []
a                  => [1, 4, 6, 9, 7]
```

```
let b = [1, 5, 3];
```

```
b.push(6, 7)      => 5
b                  => [1, 5, 3, 6, 7]
```

```
b.pop()           => 7
b                  => [1, 5, 3, 6]
```



Arrays, spread/rest (...x) y asignación múltiple (destructuring assignment)

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

Asignación múltiple en arrays (ES6)

- ◆ **ES6** añade una sentencia que asigna los elementos de un array a variables individuales
 - Se puede utilizar para asignar valores iniciales en definiciones de variables o en la asignación
 - ♦ Las variables deben agruparse entre corchetes y se relacionan por posición
 - Por ejemplo `let [x, y, z] = [5, 1, 3]` o `[y, z] = [4, 5]`
- ◆ La asignación múltiple puede utilizar valores por defecto
 - Por ejemplo `let [x, y, z=3] = [5, 1]`
- ◆ Se denomina asignación múltiple o también asignación desestructuradora (destructuring)
 - Permite hacer programas más cortos y legibles

```
// Inicializar con los 3  
// primeros elementos del  
// array
```

```
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
```

```
// Con valores por defecto e  
// 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

Spread/rest syntax: 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 del **array x** en otro array

- Actúa así cuando se aplica al constructor de array o en la invocación de una función
 - ♦ Por ejemplo, `[x, ...y, z, ...t]` o `mi_funcion(x, ...y, z, ...t)`

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

- Agrupa en un array el resto de los elementos asignados de una lista
 - ♦ Por ejemplo, `[x, y, ...resto] = [1, 2, 3, 4, 5]` o `function f(x, y, ...resto) {..}`
 - La variable agrupadora debe ir al final y agrupa los últimos elementos de la lista

```
....  
const a = [2, 3];  
const b = [0, 1, ...a];  
b => [0, 1, 2, 3]  
  
f(0, 1, ...a) => f(0, 1, 2, 3)
```

```
let [x, y, ...rest] = [0, 1, 2, 3, 4];  
x      => 0  
y      => 1  
rest   => [2, 3, 4]  
  
function f(x, y, ...z) {....}  
f(0, 1, 2, 3) => f(0, 1, [2, 3])
```

```
let x, y, z;  
[y, z, ...x] = [2, 3, 4, 5];  
x      => [4, 5];  
y      => 2  
z      => 3
```

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

Spread/rest syntax: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator



Iteradores de arrays (forEach, find, findIndex, filter, map y reduce), bucles for...of y for...in

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

Métodos iteradores de Array: forEach

◆ Método iterador

- Métodos que ejecutan una función para cada elemento de un array (u objeto iterable)
 - ◆ La **función** recibe como parámetro los **elementos** del array que debe procesar en esa invocación
 - Empiezan por el elemento de índice **0** y lo van incrementando hasta llegar a **length-1**

◆ Los métodos iteradores equivalen a **bucles**

- Ejecutan cíclicamente la función iterando en cada elemento de un array (u objeto iterable)

◆ **forEach**(function(element, index, array){...}) o **forEach**((element, index, array)=>{...})

- Invoca la función con 3 parámetros: elemento actual, su índice y el array
 - ◆ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach
- Estos dos ejemplos son equivalentes: ambos suman los elementos del array

```
let n = [7, 4, 1, 23];  
let add = 0;  
  
for (let i=0; i < n.length; ++i){  
  add += n[i];  
}  
  
add      // => 35 (7+4+1+23)
```

```
let n = [7, 4, 1, 23];  
let add = 0;  
  
n.forEach(elem => add += elem)  
  
add      // => 35
```

Otros métodos iteradores de Array

◆ Estos métodos invocan la función también con los mismos 3 parámetros

- **elem**: elemento del array accesible en la invocación en curso
- **i**: índice al elemento del array accesible en la invocación en curso
- **a**: array completo sobre el que se invoca el método

◆ **find**(function(elem, i, a){...})

```
[7, 4, 1, 23].find(elem => elem < 3); // => 1
```

- devuelve el 1^{er} elemento donde la función retorna true
 - ♦ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find

◆ **findIndex**(function(elem, i, a){...})

```
[7, 4, 1, 23].findIndex(elem => elem < 3); // => 2
```

- devuelve el índice del 1^{er} elem. donde la función retorna true
 - ♦ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/findIndex

◆ **filter**(function(elem, i, a){...})

```
[7, 4, 1, 23].filter(elem => elem > 5); // => [7, 23]
```

- elimina los elementos del array donde la función retorna false
 - ♦ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

◆ **map**(function(elem, i, a){...})

```
[7, 4, 1, 23].map(elem => -elem); // => [-7, -4, -1, -23]
```

- sustituye cada elemento del array por el resultado de invocar la función
 - ♦ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

Método reduce

◆ El método **reduce** añade el parámetro **accumulator** a element, index y array

- **accumulator**: variable con valor retornado por invocación anterior de la función
 - ◆ además están los 3 parámetros típicos de los métodos iteradores: **element**, **index** y **array**

◆ **reduce**(function(accumulator, element, index, array){...}), initial_value)

- Inicializa accumulator con **initial_value** e itera de **0** a **array.length-1**
 - ◆ **accumulator** recibe en cada nueva iteración el valor de retorno de la función
 - https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/reduce
- si **initial_value** se omite inicia accumulator con **array[0]** e itera de **1** a **array.length-1**

```
// Example of addition of numbers with reduce
[7, 4, 1, 23].reduce((acc, elem) => acc += elem, 0); // => 35
```

```
// Example which orders first the array and eliminates then duplicated numbers
[4, 1, 4, 1, 4].sort().reduce((ac, el, i, a) => el !== a[i-1] ? ac.concat(el) : ac, []); // => [1, 4]
```

```
// sort(..) and reduce(..) are composed in series, where each one performs the following
[4, 1, 4, 1, 4].sort(); // => [1, 1, 4, 4, 4]
[1, 1, 4, 4, 4].reduce((ac, el, i, a) => el !== a[i-1] ? ac.concat(el) : ac, []); // => [1, 4]
```

Bucles for...in y for...of

◆ JavaScript incluye el bucle **for...in** que itera en las propiedades de un objeto

- El bucle **for...of** de ES6 itera con una función generadora en los elementos de un objeto iterable
 - ◆ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of>
 - ◆ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in>
- Los arrays son objetos y son iterables por lo que pueden procesarse con ambos bucles

◆ La sentencia **for (prop in object) {..bloque..}** (**object** es un array u objeto)

- Ejecuta el bloque para cada **propiedad** (accesible en la variable **prop**) del objeto o array
 - ◆ Los **índices** de los elementos de un array son **propiedades** especiales (su nombre es el número)

◆ La sentencia **for (elem of object) {..bloque..}** (**object** debe ser un obj. o array iterable)

- Ejecuta el bloque para cada **elemento** (accesible en la variable **elem**) del objeto o array
 - ◆ **object** debe ser un iterable que define el orden del recorrido
 - Por ejemplo, en un array el recorrido empieza en el elemento de índice **0** y termina en el de **length-1**
- Estos 2 ejemplos de suma de elementos de Array con los nuevos bucles equivalen a los 3 ya vistos

```
let n = [7, 4, 1, 23];  
let add = 0;
```

```
for (let i in n){  
  add += n[i];  
}
```

```
add // => 35
```

```
let n = [7, 4, 1, 23];  
let add = 0;
```

```
for (let elem of n){  
  add += elem;  
}
```

```
add // => 35
```

```
let n = [7, 4, 1, 23];  
let add = 0;
```

```
for (let i=0; i < n.length; ++i){  
  add += n[i];  
}
```

```
add // => 35
```

```
let n = [7, 4, 1, 23];  
let add = 0;
```

```
n.forEach(elem => add += elem)
```

```
add // => 35
```

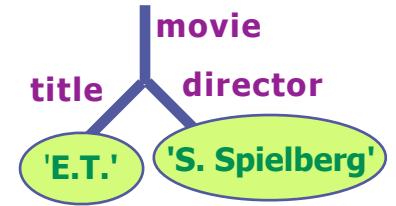
```
[7, 4, 1, 23].reduce((acc, elem) => acc += elem, 0); // => 35
```



Objetos, propiedades, métodos propios y this.

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

Propiedades de un objeto



◆ Objeto (solo con propiedades)

- **Agregación de variables** (denominadas **propiedades**)
- Suelen crearse con el literal de objeto
 - ◆ { propiedad_1:valor_1,, propiedad_n:valor_n }

◆ Los nombres de propiedades de un objeto

- deben ser **todos diferentes**
- deben tener la misma **sintaxis que las variables**
 - ◆ a, _method, \$1, una_piña,

◆ Operador punto

- **objeto.propiedad**
 - ◆ Accede al contenido de propiedades por **nombre**

◆ Operador array

- **objeto["propiedad"]**
 - ◆ La propiedad puede ser un string en una variable
 - **ES6** permite incluir expresiones arbitrarias

◆ Notación **array** extiende la notación **punto**

◆ Propiedades inexistentes devuelven **undefined**

- Pero el operador punto (.) aplicado a **undefined**
 - ◆ provoca **error de ejecución**

```
var movie = {title:'E.T.', director:'S. Spielberg'};
```

```
// Access to properties
```

```
movie.title           // => 'E.T.'
```

```
movie.director        // => 'S. Spielberg'
```

```
movie['title']         // => 'E.T.'
```

```
movie['director']      // => 'S. Spielberg'
```

```
// Access by means of variables with [..]
```

```
var t = 'title';      // contains string 'title'
```

```
movie[t]              // => 'E.T.'
```

```
movie['ti' + 'tle']    // => 'E.T.'
```

```
movie.t               // => undefined
```

```
// nonexistent properties are undefined
```

```
movie.premiere        // => undefined
```

```
movie['premiere']      // => undefined
```

```
// Execution errors
```

```
undefined.t           // => error, program stops
```

```
undefined[t]          // => error, program stops
```

Nombres extendidos de propiedades

◆ Nombre extendido de propiedad

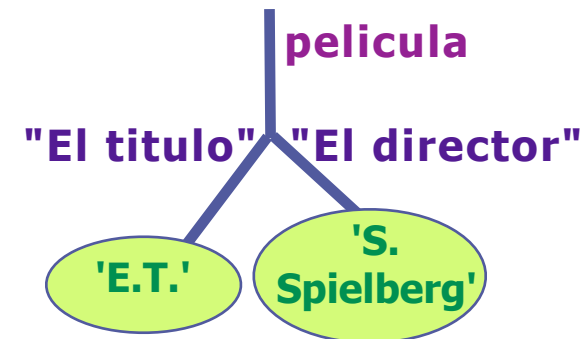
- Es un string arbitrario que **no sigue las reglas sintácticas de las variables**, es decir
 - ◆ Empezar por **letra**, **_** o **\$** y continuar por alguno de estos caracteres o **dígitos decimales**

◆ Utilizando literales de objeto y notación array

- Es posible manejar objetos con **nombres extendidos** de propiedades
 - ◆ La notación punto ('.') solo permite nombre con sintaxis de variable

◆ El literal de objeto permite crear objetos

- Utilizando strings con **nombres extendidos** de propiedades
 - ◆ {"El titulo": 'E.T.', "El director": 'S. Spielberg'}



◆ La notación array es otra forma de referenciar propiedades

- Puede utilizar **nombres extendidos** de propiedades
 - ◆ película["El director"], objeto[""] o a["%43"]
- Los **índices de arrays** son nombres especiales de propiedades de un objeto array
 - ◆ Por ejemplo, el elemento de índice 2 de un array se referencia como: a[2] o a["2"]

◆ OJO! normalmente es conveniente utilizar nombres para notación punto

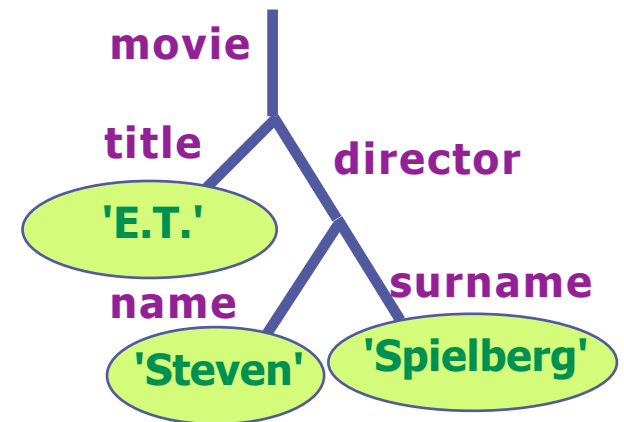
- Strings arbitrarios pueden ser útiles en objetos tipo diccionario o similares

Objetos anidados: árboles

- ◆ Los objetos pueden **anidarse** entre sí
 - Los objetos anidados representan **árboles**
- ◆ La notación punto o array puede **encadenarse**
 - Representando un **camino en el árbol**
 - ◆ Las siguientes expresiones se evalúan así:

■ <code>movie.title</code>	=> <code>'E.T.'</code>
■ <code>movie.director.name</code>	=> <code>'Steven'</code>
■ <code>movie['director']['name']</code>	=> <code>'Steven'</code>
■ <code>movie['director'].surname</code>	=> <code>'Spielberg'</code>
■ <code>movie.director</code>	=> <code>{name:'Steven', surname: 'Spielberg'}</code>
■ <code>movie.premiere</code>	=> <code>undefined</code>
■ <code>movie.premiere.year</code>	=> Error_de_ejecución

```
var movie = {  
  title: 'E.T.',  
  director: {  
    name: 'Steven',  
    surname: 'Spielberg'  
  }  
};
```



Propiedades dinámicas

◆ Las **propiedades** se pueden **crear** y **destruir**

- Para ello se utilizan 3 sentencias
 - ◆ Asignación de valores
 - ◆ Borrado de propiedades
 - ◆ Comprobar si existe una propiedad

◆ Asignar a (**y crear**) propiedades: **x.c = 4**

- **asigna 4** -> si la propiedad **c** **existe**
- **crea c y le asigna 4** -> si la propiedad **c** **no existe**

◆ Borrar propiedades:

- **elimina x.c** -> si la propiedad **x.c** **existe**
- **no hace nada** -> si la propiedad **x.c** **no existe**

◆ ¿Existe la propiedad?:

- devuelve **true** -> si la propiedad **x.c** **existe**
- devuelve **false** -> si la propiedad **x.c** **no existe**

La propiedad ya existe y **solo cambia el valor a 7**

La propiedad no existe y **se crea** con el valor 5

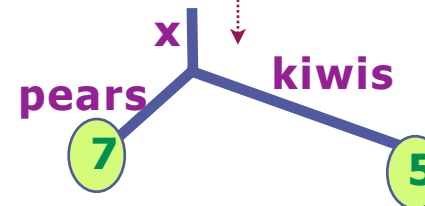
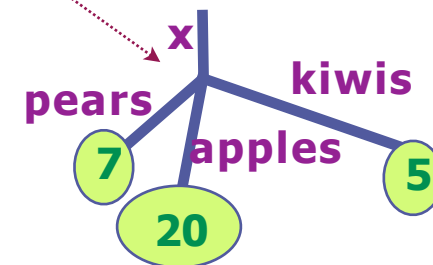
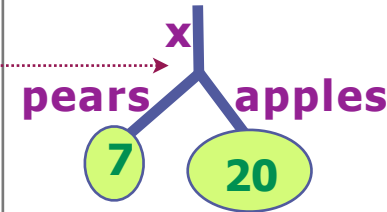
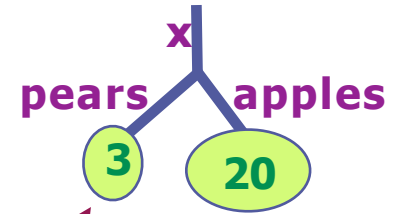
```
var x = { pears:3, apples:20};
```

```
x.pears = 7;
```

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

```
delete x.apples;
```

La propiedad se destruye

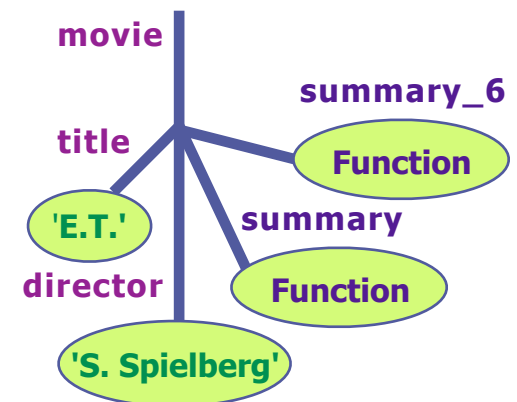


Definición de métodos propios

- ◆ Un **método propio** es una función que se guarda en una propiedad del objeto
 - Un método propio **solo existe** en el **objeto en el que ha sido definido**
 - ◆ Se invoca sobre ese objeto con los operadores punto y paréntesis, por ej. `movie.summary()`
- ◆ **this** es una referencia al objeto sobre el que se invoca el método
 - En el ejemplo, **this.title** referencia la propiedad **title** del objeto **movie**
 - ◆ **this** puede omitirse si no hay ambigüedad y en el ejemplo podría utilizarse solo **title** o **director**
 - <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this>
- ◆ **ES6** añade una sintaxis simplificada que omite **":"** y **"function"**
 - Por ejemplo, **summary_6** define un método equivalente a **summary** con sintaxis **ES6**

```
var movie = {  
  title: 'E.T.',  
  director: 'S. Spielberg',  
  summary: function() {  
    return "The director of " + this.title + " is " + this.director;  
  },  
  summary_6 () {  
    return "The director of " + this.title + " is " + this.director;  
  }  
}
```

```
movie.summary() // => "The director of E.T. is S. Spielberg"  
movie.summary_6() // => "The director of E.T. is S. Spielberg"
```



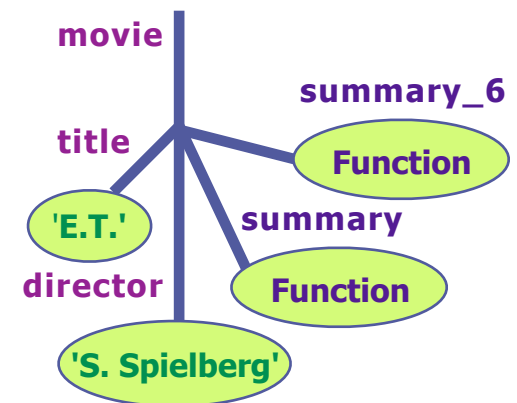
Estos dos métodos se denominan **propios** porque se han definido directamente en un objeto y solo se pueden invocar en él.

Creación dinámica de métodos

- ◆ Un **método propio** se puede añadir también dinámicamente a un objeto
 - Debe añadirse una propiedad con la función correspondiente, por ejemplo
 - ◆ `movie.summary = function(){ return "The director of " + this.title + " is " + this.director};`
- ◆ Los métodos `summary` y `summary_6` del objeto `movie` son similares
 - La única diferencia es que se han creado dinámicamente y no con el literal

```
var movie = {  
  title: 'E.T.',  
  director: 'S. Spielberg',  
}  
  
movie.summary = function() {  
  return "The director of " + this.title + " is " + this.director;  
};
```

```
movie.summary() // => "The director of E.T. is S. Spielberg"
```



Estos dos métodos siguen siendo métodos **propios** porque se han definido sobre este objeto y solo se pueden invocar en él.



Objetos: Literal de ES6, multi- asignación, spread/rest (...x), for...in y Object.keys(..)

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

El literal de objetos ES6: agrupar variables

- ◆ La agrupación de variables en ES5 se realiza con el literal de objetos
 - Por ejemplo, `var obj = {a:a, b:b, c:c}` agrupa en un objeto las variables a, b y c
- ◆ ES6 también permite agrupar o estructurar objetos de forma mas concisa
 - Por ejemplo, `var obj = {a, b, c}` es equivalente en ES6 a lo anterior
 - ◆ El literal de objeto permite incluir solo el nombre de la variable, cuando esta inicializa una propiedad del mismo nombre con la variable en un objeto

```
let a=5, c=3, d=4;
```

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

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


Asignación múltiple o destructuración

◆ 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 destructor

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

- Las variables a asignar se agrupan con llaves y pueden llevar valores por defecto
 - ◆ Por ejemplo `let {a, b} = {a:5, b:1}` o `({a, b} = {a:1, b:2})`

```
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, c, d, e:3});
```

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

La multi-asignación debe ir entre paréntesis por un problema del 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

Spread/rest syntax: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator

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

◆ El operador **rest** (...x) también puede utilizarse con la asignación múltiple de objetos

- 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};
```

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

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

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

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

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

◆ El operador **spread** (...x) también puede utilizarse para esparcir 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};
```

y	=>	{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

Spread/rest syntax: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator

Multi-asignación con objetos anidados

- ◆ La multi-asignación permite también **objetos anidados**, así como **cambiar nombres**
 - El **nombre de la variable** creada o asignada es el de la **propiedad** que casa en la parte izquierda

```
let {x:a, y} = {x:1, y:10}
```

```
a      => 1
```

```
y      => 10
```

```
let {x, y:{y}} = {x:1, y:{t:10, y:11}}
```

```
x      => 10
```

```
y      => 11
```

- ◆ Formato general permite además **cambios de nombre** y valor por defecto
 - El **nuevo nombre de la variable** debe añadirse como **valor de la propiedad** a la izquierda
 - Y el **valor por defecto** debe asignarse con = al **nombre de la variable** a asignar

```
let {x:a, y:{t=7}, y:{y:b}, z:c=4} = {x:1, y:{t:10, y:11}, u:3}
```

```
a      => 1
```

```
t      => 10
```

```
b      => 11
```

```
c      => 4
```

Sentencias `for...in` y `Object.keys(obj)`

◆ Sentencia `for (let p in obj) {..bloque de instrucciones..}`

- Itera en todas las propiedades de `obj`, siguiendo el orden de inserción de propiedades
 - ♦ En cada iteración `p` contiene el nombre (string) de la propiedad para acceso con `obj[p]`
- `for (let elem of obj) {...}` solo permite objetos iterables y no se utilizar con `{a:7, b:4, c:1, d:23}`
- La sentencia itera en las propiedades enumerables del objeto y de sus prototipos
 - ♦ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in>

◆ `Object.keys(obj)` extrae un array con los nombres de las **propiedades**

- Permite utilizar los iterados de arrays con objetos
 - ♦ Normalmente conviene utilizar `Object.keys(...)`, en vez de `for (let p in obj) {...}`, porque el array devuelto por `Object.keys` contiene solo las propiedades enumerables propias del objeto, no incluye las de sus prototipos

```
let obj = {a:7, b:4, c:1, d:23};  
let add = 0;
```

```
for (let p in obj) {  
    add += obj[p];  
}
```

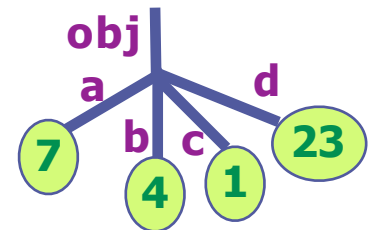
```
add // => 35
```

```
let obj = {a:7, b:4, c:1, d:23};  
let add = 0;
```

```
Object.keys(obj); // => ["a", "b", "c", "d"]
```

```
Object.keys(obj).forEach(p => add += obj[p]);
```

```
add // => 35
```





JSON: JavaScript Object Notation

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

Serialización de datos: JSON

◆ Serialización de datos

- transformación **reversible** de valores en un string equivalente
- Facilita el almacenamiento y envío de datos, por ejemplo
 - ◆ Almacenar datos en un fichero
 - ◆ Enviar datos a través de una línea de comunicación
 - ◆ Paso de parámetros en interfaces REST

◆ JSON - JavaScript Object Notation

- Formato de serialización de valores y objetos JavaScript
 - ◆ Cubre las partes más importantes de los objetos JavaScript
 - <http://json.org/json-es.html>

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

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

Objeto global JSON

◆ JavaScript tiene el objeto global JSON con métodos de conversión

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON

◆ **JSON.stringify(<object>)**

- El método **stringify** transforma un objeto (<object>) en un string JSON equivalente

◆ **JSON.parse(<string>)**

- El método **parse** transforma un <string> JSON en el objeto o valor equivalente

JSON.stringify(null)	// => 'null'
JSON.parse('null')	// => null
JSON.stringify(127)	// => '127'
JSON.stringify('hola')	// => '"hola"'
JSON.stringify([1, 2, 3])	// => '[1, 2, 3]'
JSON.stringify({a:27, b:"hola"})	// => '{"a":27,"b":"hola"}'

Características de JSON

◆ JSON puede serializar

- objetos, arrays, strings, números finitos, true, false y null
 - ◆ NaN, Infinity y -Infinity se serializan por defecto a null
 - ◆ Los objetos Date se serializan como un string en formato ISO 8601
 - la reconstrucción devuelve un string y no el objeto original
- No se puede serializar
 - ◆ Funciones, RegExp, errores, undefined

◆ parse y stringify admiten filtros para los elementos no soportados

- ver doc de APIs JavaScript:
 - ◆ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON

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

JSON.stringify(NaN) => 'null'

JSON.stringify(Infinity) => 'null'

Ejemplo de datos en JSON

- ◆ **JSON** es un formato flexible y legible de datos muy utilizado
 - permite insertar espacios en blanco y retorno de línea entre los símbolos
 - ◆ El siguiente ejemplo muestra un array con 4 objetos en JSON

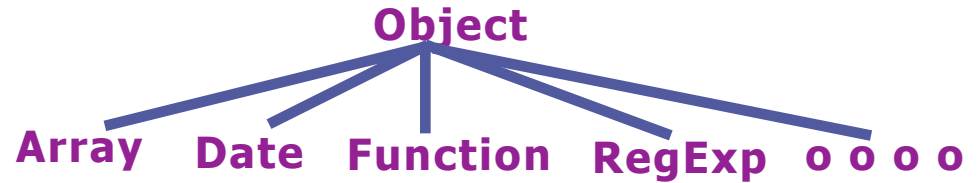
```
'[ { "title": "E.T.",  
    "director": "Steven Spielberg"  
},  
  { "title": "Star Wars",  
    "director": "George Lucas"  
},  
  { "title": "Psicosis",  
    "director": "Alfred Hitchcock"  
},  
  { "title": "Placido",  
    "director": "Luis García Berlanga"  
}  
']
```



Clases predefinidas de JavaScript: propiedades y métodos heredados

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

Clases y herencia



◆ Todos los objetos de JavaScript pertenecen a la **clase Object**

- Javascript posee otras **clases predefinidas** que derivan de Object
 - ◆ **Array, Date, Function, RegExp, Number, String, Boolean, Map, Set,**
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript
- Un objeto **hereda** los métodos y propiedades de su clase
 - ◆ Un objeto puede tener además propiedades y métodos **propios**
 - Los métodos propios solo existen para un objeto particular y no para el resto de la clase

◆ El acceso a métodos y propiedades utiliza la **notación punto o array**

- Una propiedad se accede como: **objeto.propiedad** o **objeto["propiedad"]**
- Un método se invoca como: **objeto.metodo(...)** o **objeto["metodo"](...)**

◆ Cada clase tiene un **constructor** que se invoca con el **nombre de la clase**

- El constructor permite **crear objetos** con el **operador new**
 - ◆ Pero los literales suelen ser más eficientes creando objetos, arrays, etc.
 - Por ejemplo, **new Object()** crea un objeto vacío, equivalente a **{}**

Clases predefinidas

◆ Object

- Clase raíz que define colecciones de propiedades y métodos. Literal de objeto: **{a:3, b:"que tal"}**

◆ Array

- Define colecciones ordenadas de valores. Literal de array: **[1, 2, 3]**

◆ Date

- Define objetos con hora y fecha del reloj del sistema. Solo constructor: **new Date(...)**

◆ Function

- Define código parametrizado. Literales de función: **function (x) {....}** o **(x) => {....}** (ES6)

◆ RegExp

- Define expresiones regulares para reconocer y procesar patrones de texto. Literal: **/(hola)+\$/**

◆ Error

- Errors de ejecución lanzados por el interprete de JavaScript. Solo constructor: **new Error(...)**

◆ Number, String y Boolean

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

◆ ES6 introduce nuevas clases

- **Promises, Map, Set, Typed Arrays,**

◆ Más información:

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

Operador instanceof

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

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

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

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

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

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(),** (ver ejemplo)
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/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
```



Clase, prototipo y herencia: métodos de instancia o estáticos y this

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

Clases JavaScript

◆ JavaScript utiliza "Tipado de Patos"

- "Si anda y grazna como un pato, debe ser un pato"
 - ◆ JavaScript simula clases utilizando **funciones** y **prototipos**

◆ Clase

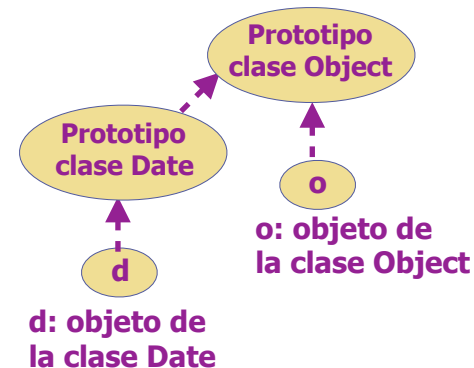
- Conjunto de objetos creados con el mismo **constructor** y que comparten un **prototipo**

◆ Prototipo

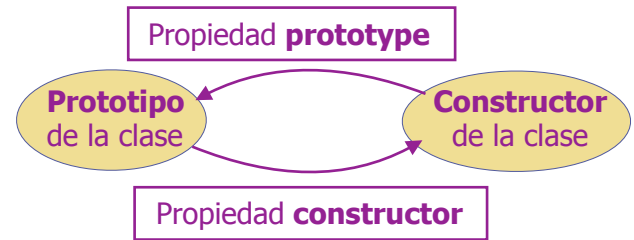
- Objeto del que objetos de una clase **heredan** los métodos y propiedades de clase
 - ◆ Estos se denominan **heredados** y existen en todos los objetos de la clase
- Los prototipos de clases derivadas están enlazados y se hereda de toda la cadena
 - ◆ La clase Object es la clase raíz del árbol de herencia y su prototipo es el único que no está enlazado
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain
 - ◆ Si dos prototipos de la cadena tienen una propiedad/método con el mismo nombre, se hereda la más cercana

◆ Constructor

- **Función** que crea objetos de la clase al invocarse con el operador **new**, por ejemplo
 - ◆ **new Object()** crea un objeto similar a {}
 - ◆ **new Array()** crea un array similar a []
 - ◆ **new Date()** crea un objeto Date con la fecha y la hora de su creación (Date no tiene literal)
- El **nombre del constructor** y de la **clase** son los mismos
 - ◆ Existe la convención de utilizar los nombres que comienzan por mayúscula solo para constructores de clase, aunque cualquier función puede utilizarse como constructor de una clase



Algunas propiedades y métodos



◆ Propiedades del **constructor**

■ **prototype**

- ◆ Devuelve el prototipo de la clase del objeto
- ◆ Ejemplos de métodos de instancia:
 - **Object.prototype.toString()**
 - **Array.prototype.forEach()**

■ **name**

- ◆ Devuelve un string con el nombre del constructor o de la clase asociada

```
Object.prototype; // => {}  
Object.name;      // => "Object"  
  
Array.prototype;  // => []  
Array.name;       // => "Array"  
  
Date.prototype;   // => Date {}  
Date.name;        // => "Date"
```

◆ Propiedades del **prototipo**

■ **constructor**

- ◆ Devuelve el constructor de la clase

◆ **Object.getPrototypeOf(<obj>)**

■ **Método estático** de la clase **Object**

- ◆ Da acceso al prototipo de la clase asociada al objeto <obj>
 - Equivale a: **obj.constructor.prototype**

```
{}.constructor; // => [Function: Object]  
{}.constructor.name; // => "Object"  
  
[].constructor; // => [Function: Array]  
[].constructor.name; // => "Array"  
  
new Date().constructor; // => [Function: Date]  
new Date().constructor.name; // => "Date"
```

Ejemplo de definición de clase: Counter

◆ Constructor

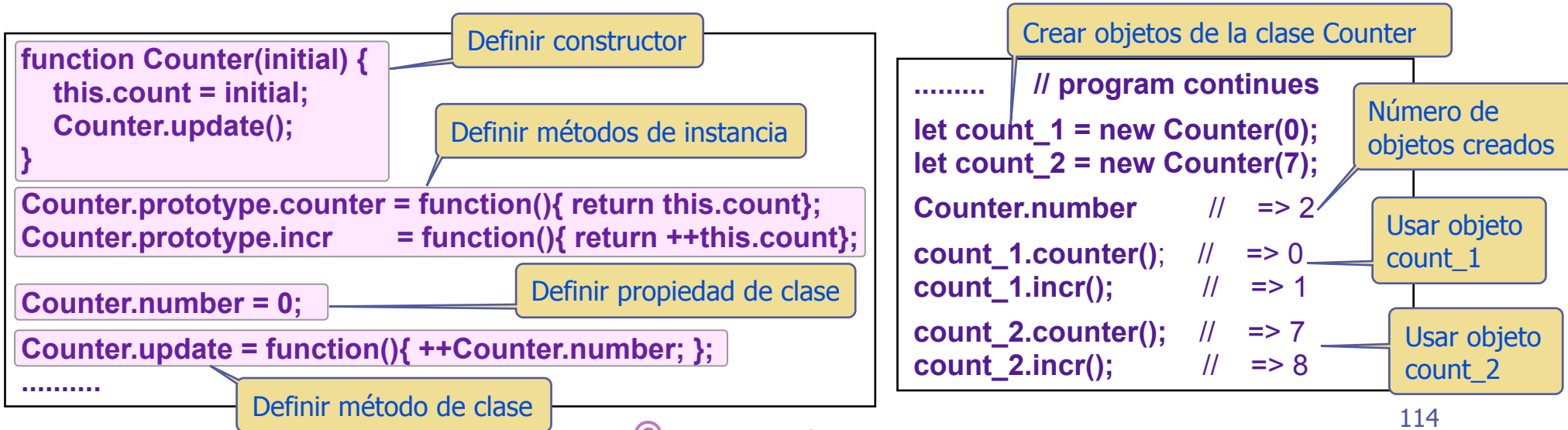
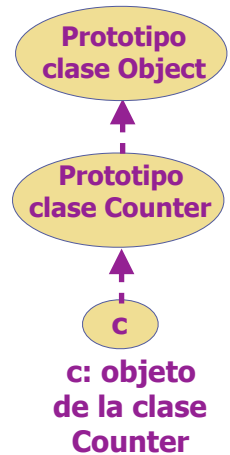
- El primer paso para crear una clase es definir el constructor: **Counter(..){..}**
 - this** referencia el objeto que está creando el constructor, cuando se invoca con **new**
 - this.count** crea dinámicamente la propiedad **count** con el contador del nuevo objeto

◆ Métodos de instancia: counter() e incr()

- Los métodos de instancia se añaden al prototipo de la función creado al definirla
 - El prototipo es un objeto JavaScript al que se le puede añadir propiedades y métodos como a los demás
 - obj.counter()** devuelve el estado del contador y **obj.incr()** incrementa el contador y devuelve su contenido

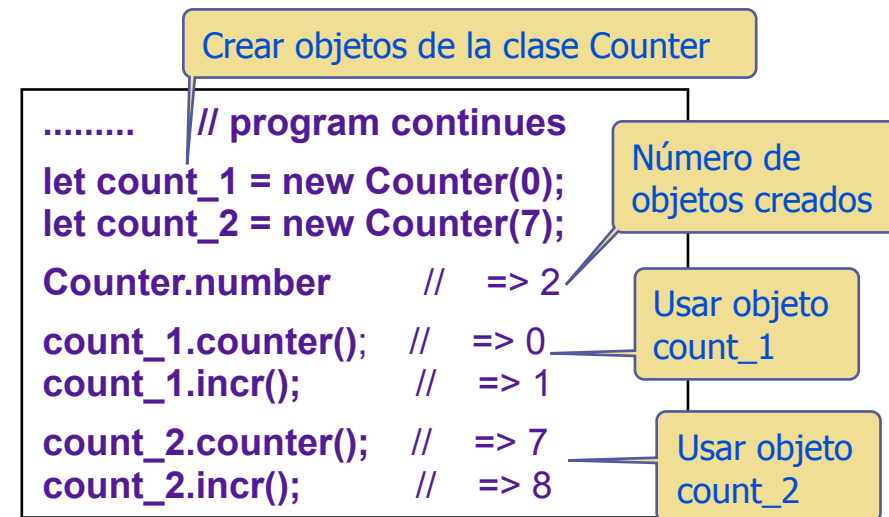
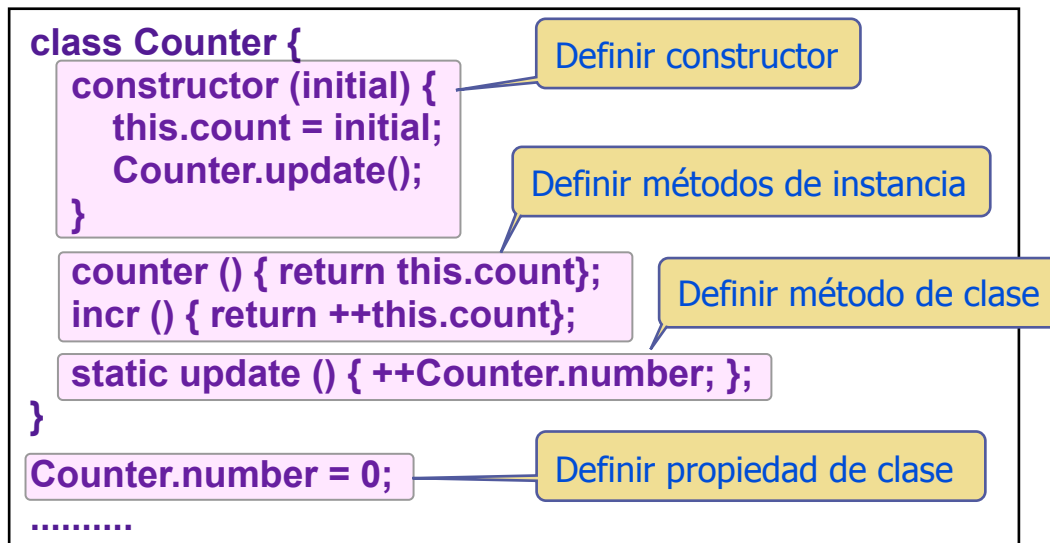
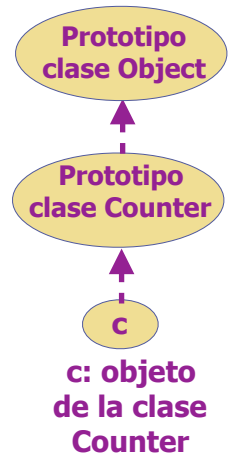
◆ Métodos y propiedades de clase: number y update

- Se deben añadir al constructor, por ejemplo **Counter.number** o **Counter.update**
 - Counter.update()** incrementa la cuenta de objetos creados en **Counter.number**

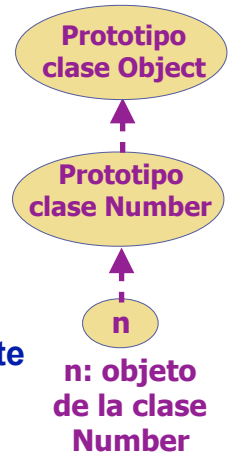


Definición de clase en ES6: Counter

- ◆ **ES6** añade nueva sintaxis para definir clases de forma mas legible y concisa
 - Es azúcar sintáctico: las clases se construyen con funciones y prototipos como en ES5
 - ◆ Mas información en: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes>
- ◆ El ejemplo muestra la misma clase Counter de la transparencia anterior
 - La sintaxis incluye la clase, los métodos de instancia y los métodos estáticos o de clase
 - ◆ Las propiedades de clase (estáticas) no están soportadas y se definen como en ES5 (en constructor)
- ◆ La sintaxis de **ES6** permite también extender clases y crear jerarquías
 - Aquí no lo vemos, pero se puede encontrar mas información en:
 - ◆ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/extends>
 - ◆ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super>



Añadir método integer a la clase Number



◆ La clase **Number** encapsula métodos del tipo primitivo number

- Esta clase no posee un **método integer()** que calcule la parte entera del número
 - ♦ Para añadir este nuevo método a la clase se debe añadir **al prototipo**
 - Como **la propiedad puede existir**, hay que comprobarlo primero y lanzar un **error si existe**
- Documentación:
 - ♦ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain

◆ La parte entera de **n** es **Math.floor(n)** si es positivo o **Math.ceil(n)** si es negativo

- **this** referencia el objeto sobre el que se invoca el método (el número aquí)
 - ♦ Se utiliza notación array para invocar **Math.floor(..)** o **Math.ceil(..)**

Si ya existe una propiedad **integer** se lanza un error.

```
if("integer" in Number.prototype){
  throw new Error("Number.prototype.integer already exists!");
}

// Añadimos método "integer()" a Number

Number.prototype.integer = function () {
  return Math[this > 0 ? "floor" : "ceil"](this);
}

console.log("' 7.3.integer()' se evalúa a: " + 7.3.integer());
console.log("' -7.3.integer()' se evalúa a: " + -7.3.integer());
```

this > 0 ? "floor" : "ceil" devuelve el nombre de la función que calcula el entero más próximo: **"floor"** o **"ceil"**. Este se aplicaría, por ejemplo si el número es positivo como: **Math["floor"](this)**. **this** referencia el número al que se aplica el método **integer()**.

El nuevo método **integer** se añade al prototipo de la clase **Number**.

```
_$
_$ node 50_integer.js
' 7.3.integer()' se evalúa a: 7
' -7.3.integer()' se evalúa a: -7
_$
```



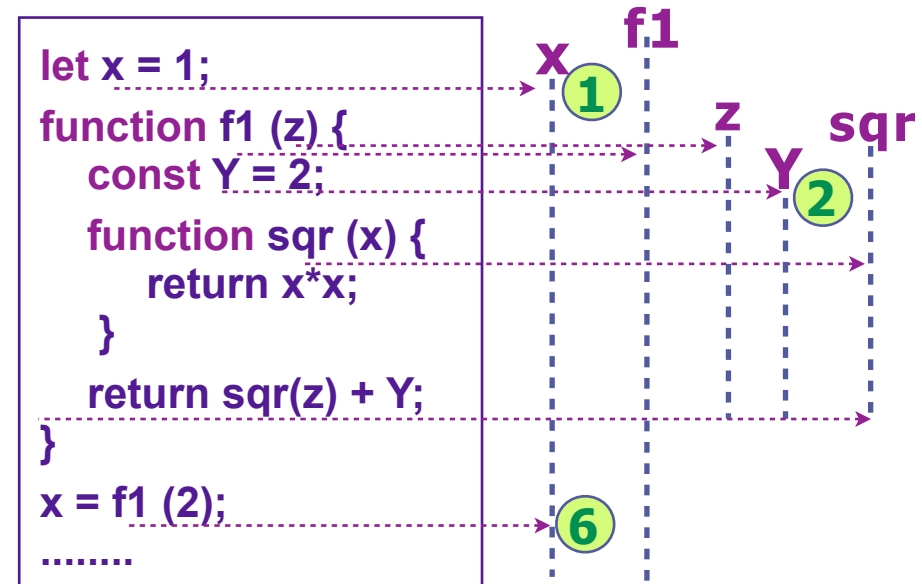
Espacio de nombres, cierres y clases

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

Espacio de nombres, ámbito de visibilidad y funciones

◆ Espacio de nombres de un programa

- Son particiones usadas para organizar los elementos declarados en un programa (variables, constantes, funciones, etc) en ámbitos de visibilidad
 - ◆ Los nombres permiten referenciar los elementos en expresiones y sentencias de cada ámbito



◆ Ámbito de visibilidad

- Es la parte del programa donde una declaración es visible y puede ser utilizada, por ejemplo
 - ◆ **Variables y funciones globales:** tienen como **ámbito** de visibilidad **todo el programa**
 - ◆ **Variables y funciones locales** (de otra función): tienen como **ámbito** de visibilidad la **función** que las contiene
 - Existe un convenio extendido por el cual el nombre variables y funciones locales suele comenzar con `_`
- Hasta **ES6** la **función** era el único mecanismo para limitar el **ámbito** de visibilidad en JavaScript
- En **ES6** es posible limitar el **ámbito** de visibilidad también con **módulos ES6** y con **bloques** (delimitados por `{}`)

◆ Funciones y encapsulación

- Las funciones son el mecanismo principal para crear ámbitos de visibilidad y encapsular código
 - ◆ Las funciones permiten limitar la visibilidad las definiciones locales de variables y funciones

Cierres

◆ Un **cierre** encapsula una parte del código de un programa aislándolo del resto

- El cierre mantiene un estado interno
- Se implementa definiendo una función
- La función **cierre** retorna un **objeto** con el **interfaz** de uso del cierre
 - ◆ Este objeto se denominará **objeto interfaz**
- Los **módulos** son cierres en JavaScript

◆ La función cierre del ejemplo

- **Encapsula las declaraciones locales** (variables y funciones) aislándolas de accesos exteriores
 - ◆ Existe un **convenio** por el que los nombres de las **declaraciones locales** suelen empezar por _
- Retorna un **objeto interfaz**, aislando el **entorno de ejecución interno**
 - ◆ El objeto interfaz puede tener todas las propiedades y métodos necesarias

◆ El cierre del ejemplo es una factoría que genera **objeto_1** y **objeto_2**

- Las variables locales de **objeto_1** y **objeto_2** son diferentes
 - ◆ Cada objeto generado por la factoría tiene su propio estado, igual que los objetos generados con constructores
- El estado del cierre permanece mientras existan referencias a **objeto_1** u **objeto_2**

```
function mi_cierre (.. parámetros ..) {  
  let _var_local_1 ...;  
  let _var_local_2 ...;  
  .....  
  function _funcion_local_1 (..) { .... }  
  function _funcion_local_2 (..) { .... }  
  .....  
  return {  propiedad_interfaz_1: ...,  
            propiedad_interfaz_2: ...,  
            .....  
            funcion_interfaz_1: function (..) {..},  
            funcion_interfaz_2: function (..) {..},  
            .....  
          }  
}  
  
let objeto_1 = mi_cierre (.. parám_iniciales_1 ..);  
let objeto_2 = mi_cierre (.. parám_iniciales_2 ..);
```

Ejemplo: Agenda telefónica

- ◆ La **agenda telefónica** ilustra como modularizar con cierres y clases
 - La **primera** agenda es un **cierre** que crea una **factoría** de objetos agenda
 - ◆ El nombre de la propiedad será el nombre de la persona
 - La **segunda** es una **clase** donde el constructor permite crear objetos agenda
 - La **tercera** es una **clase**, como la segunda, pero con la nueva **sintaxis ES6**
- ◆ La agenda utiliza un **objeto** para guardar **pares clave-valor**
 - El nombre de la propiedad es la **clave** (persona) y el **valor** el teléfono
 - Para acceder a propiedades debe utilizarse la notación array
 - ◆ El nombre de las propiedades es un string arbitrario con el nombre de la persona
 - Por ejemplo **agenda["Pedro Ruíz"]**
- ◆ El ejemplo ilustra también el uso de JSON para transferir agendas
 - El formato JSON es string sencillo de pasar como parámetro o guardar en fichero


```
function agenda (title, init) {
```

```
  let _title = title;  
  let _content = init;
```

El cierre protege las variables locales **_title** y **_content**, que no son accesibles desde el exterior

Agenda como cierre

```
  return {  
    title: function() { return _title; },  
    add: function(nombre, tf) { _content[nombre]=tf; },  
    tf: function(nombre) { return _content[nombre]; },  
    remove: function(nombre) { delete _content[nombre]; },  
    toJSON: function() { return JSON.stringify(_content); },  
    fromJSON: function(agenda) { Object.assign(_content, JSON.parse(agenda)); }  
  }  
}
```

El estado del cierre, **_title** y **_content**, solo es accesible a través de los métodos del interfaz.

```
let friends = agenda ("friends",  
  { Peter: 913278561,  
    John: 957845123  
  });  
friends.add("Mary", 978512355);
```

```
let work = agenda ("Work", {});  
work.fromJSON('{"Peter Tobb": 913278561, "Paul Smith": 957845123}');
```

```
console.log('Peter: ' + friends.tf("Peter"));  
console.log('Mary: ' + friends.tf("Mary"));  
console.log('Edith: ' + friends.tf("Edith"));  
console.log();  
console.log('Peter Tobb: ' + work.tf("Peter Tobb"));  
console.log('Work agenda: ' + work.toJSON());
```

```
venus:ej jq$  
venus:ej jq$ node 70-agenda_closure.js  
Peter: 913278561  
Mary: 978512355  
Edith: undefined  
  
Peter Tobb: 913278561  
Work agenda: {"Peter Tobb":913278561,"Paul Smith":957845123}  
venus:ej jq$
```

```
function Agenda (title, init) {
```

```
  this.title = title;  
  this.content = init;
```

```
};
```

Las propiedades del objeto **title** y **content** no están protegidas, están accesibles desde el exterior

Agenda como clase

```
Agenda.prototype = {  
  title: function()      { return this.title; },  
  add:   function(name, tf) { this.content[name]=tf; },  
  tf:    function(name)    { return this.content[name]; },  
  remove: function(name)  { delete this.content[name]; },  
  toJSON: function()      { return JSON.stringify(this.content);},  
  fromJSON: function(agenda) { Object.assign(this.content, JSON.parse(agenda));}  
}
```

```
let friends = new Agenda ("friends",  
                           { Peter: 913278561,  
                             John: 957845123  
                           });  
friends.add("Mary", 978512355);
```

```
let work = new Agenda ("Work", {});  
work.fromJSON('{"Peter Tobb": 913278561, "Paul Smith": 957845123}');
```

```
console.log('Peter: ' + friends.tf("Peter"));  
console.log('Mary: ' + friends.tf("Mary"));  
console.log('Edith: ' + friends.tf("Edith"));  
console.log();
```

```
console.log('Peter Tobb: ' + work.tf("Peter Tobb"));  
console.log('Work agenda: ' + work.toJSON());
```

```
venus:ej jq$
```

```
venus:ej jq$ node 71-agenda_class.js
```

```
Peter: 913278561
```

```
Mary: 978512355
```

```
Edith: undefined
```

```
Peter Tobb: 913278561
```

```
Work agenda: {"Peter Tobb":913278561,"Paul Smith":957845123}
```

```
venus:ej jq$
```

```
class Agenda {
```

```
  constructor (title, init) {  
    this.title = title;  
    this.content = init;  
  };
```

Las propiedades del objeto **title** y **content** no están protegidas, están accesibles desde el exterior

Agenda como clase ES6

```
  title()      { return this.title; };  
  add(name, tf) { this.content[name]=tf; };  
  tf(name)     { return this.content[name]; };  
  remove(name) { delete this.content[name]; };  
  toJSON()     { return JSON.stringify(this.content);};  
  fromJSON(agenda) { Object.assign(this.content, JSON.parse(agenda));};  
}
```

```
let friends = new Agenda ("friends",  
                           { Peter: 913278561,  
                             John: 957845123  
                           });  
friends.add("Mary", 978512355);
```

```
let work = new Agenda ("Work", {});  
work.fromJSON('{ "Peter Tobb": 913278561, "Paul Smith": 957845123 }');
```

```
console.log('Peter: ' + friends.tf("Peter"));  
console.log('Mary: ' + friends.tf("Mary"));  
console.log('Edith: ' + friends.tf("Edith"));  
console.log();
```

```
console.log('Peter Tobb: ' + work.tf("Peter Tobb"));  
console.log('Work agenda: ' + work.toJSON());
```

```
venus:ej jq$  
venus:ej jq$ node 72-agenda_class_ES6.js  
Peter: 913278561  
Mary: 978512355  
Edith: undefined  
  
Peter Tobb: 913278561  
Work agenda: {"Peter Tobb":913278561,"Paul Smith":957845123}  
venus:ej jq$
```

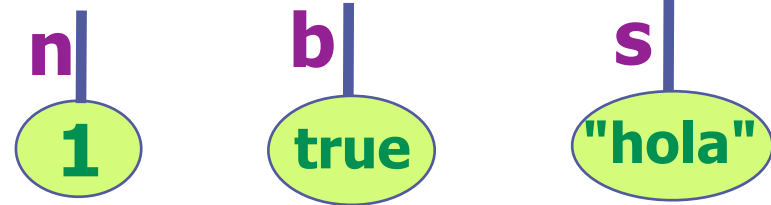


JavaScript

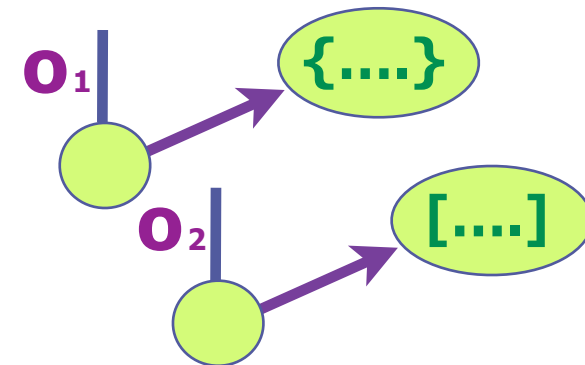
Referencias a objetos: comparación y compartición de objetos

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 gestionan por valor
 - Los **objetos** se gestionan por referencia
 - ◆ 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 la **referencia**
- ◆ 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

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

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

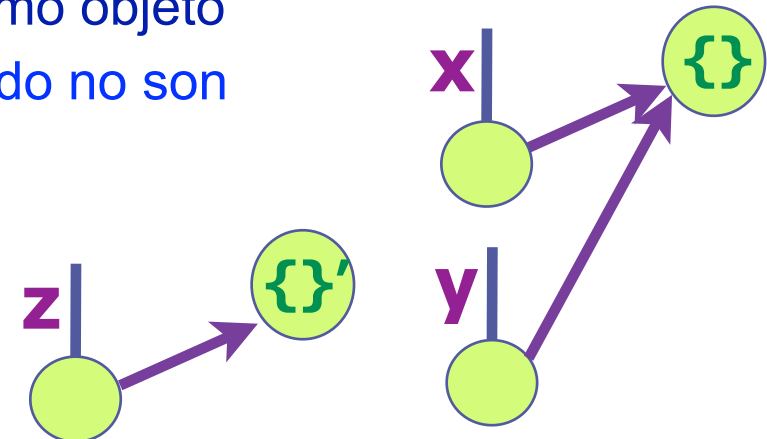
<code>x === y</code>	<code>=> true</code>
<code>x === {}</code>	<code>=> false</code>
<code>x === z</code>	<code>=> false</code>

◆ 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

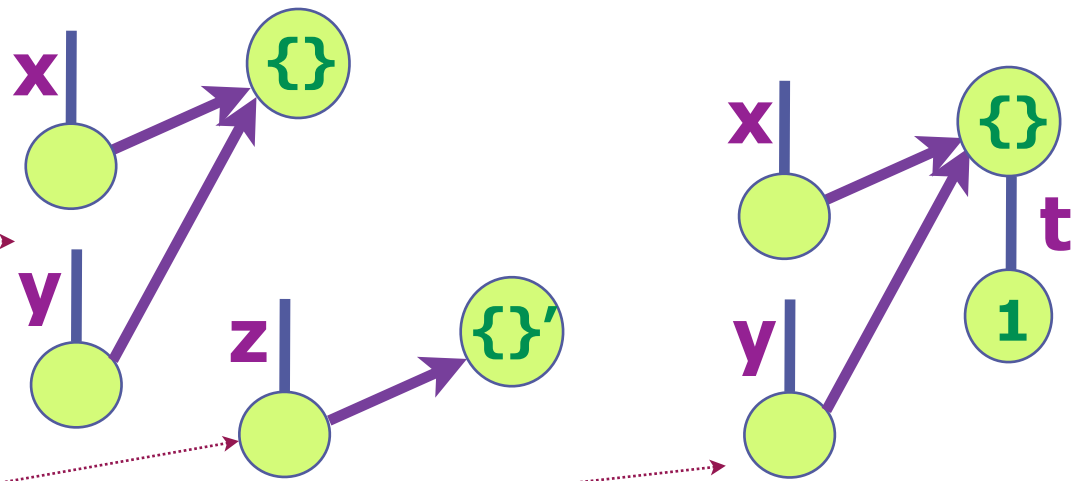


```
var x = {}; // x e y tienen la  
var y = x; // misma referencia
```

```
var 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
```

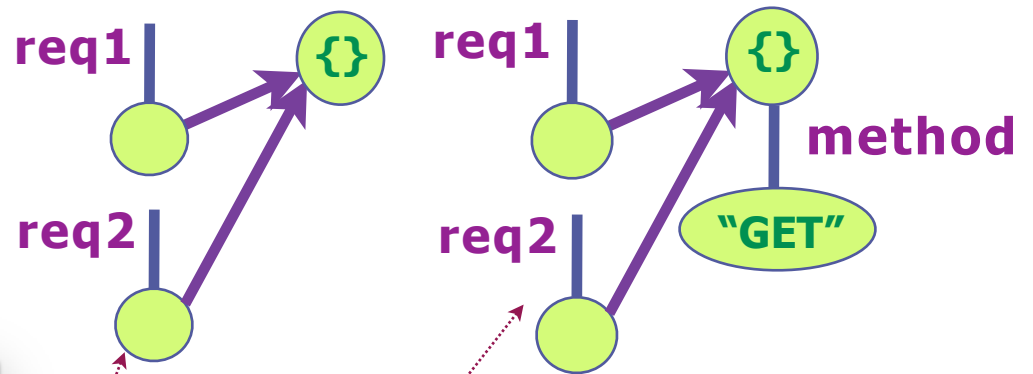


- ◆ Las variables que contienen objetos solo guardan la referencia al objeto
 - El objeto está alojado en un lugar de la memoria apuntado por la referencia
- ◆ Al asignar una variable se copia la referencia
 - si se modifica el objeto de una de ellas
 - ◆ Todas las variables que contengan la misma referencia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

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"
```



- ◆ 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 desde todas las referencias al objeto
- ◆ Hay que tener presente que los objetos pasados como parámetros a una función
 - pueden ser modificados por esta



RegExp I: Búsqueda de patrones

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

Expresiones regulares: RegExp

◆ Expresiones regulares:

- ▶ Mecanismo muy eficaz para procesar strings
 - Soportado por muchos lenguajes y herramientas (UNIX)
 - » Emacs, vi, awk, grep, PERL, Java, Javascript, Ruby, ...
- ▶ Definen patrones que reconocen cadenas de caracteres específicas
 - Si un patrón reconoce una cadena, se dice que casa (match) con el patrón

◆ En JS se definen con la clase **RegExp** y se pueden crear con

- ▶ Constructor: **RegExp("expresion-regular")**
 - El string puede ser cualquier expresión regular
- ▶ RegExp literal: **/expresion-regular/**
- ▶ Info: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions

Búsqueda de patrones

◆ `string.match(/patrón/)` busca `/patrón/` en `string`

- ▶ Devuelve la primera ocurrencia del patrón en un array (`[match]`), y si no casa devuelve `null`

◆ Algunos patrones básicos

- ▶ caracter: `/a/` reconoce solo el caracter “a”
- ▶ secuencia: `/abc/` reconoce la secuencia “abc”
- ▶ principio de string: `/^hoy/` reconoce “hoy” al principio del string
- ▶ final de string: `/hoy$/` reconoce “hoy” al final del string
- ▶ cualquier caracter: `/./` reconoce cualquier caracter

<code>"Es hoy".match(/hoy/)</code>	<code>=> ['hoy']</code>
<code>"Es hoy".match(/hoy/)[0]</code>	<code>=> 'hoy'</code>
<code>"Es hoy".match(/hoy\$/)</code>	<code>=> ['hoy']</code>
<code>"Es hoy".match(/^hoy/)</code>	<code>=> null</code>
<code>"Es hoy".match(/^..../)</code>	<code>=> ['Es h']</code>

Clases y rangos de caracteres

- ◆ **Clase de caracteres:** patrón con varios caracteres alternativos entre corchetes
 - ▶ Ejemplo de clase de caracteres: `/[aeiou]/` cualquier vocal (minúscula)
 - ▶ Ejemplo de clase negada: `/[^aeiou]/` no debe ser vocal (minúsc.)
 - ▶ Patrón `\s`: reconoce separadores `[\f\n\r\t\v\u00a0\u1680]`
- ◆ **Rango de caracteres:** Patrón con un rango de caracteres de ASCII alternativos
 - ▶ Rango de caracteres: `/[a-z]/` rango “a-z” de letras ASCII
 - ▶ Patrón `\w`: equivale a `/[a-zA-Z0-9_]/`
 - ▶ Patrón `\d`: equivale a `/[0-9]/`

```
"canciones".match(/[aeiou]/)    => ['a']  
"canciones".match(/c[aeiou]/)   => ['ca']  
"canciones".match(/n[aeiou]/)   => ['ne']
```

Controles y **match()**

◆ **match()** admite controles: **i**, **g** y **m**

- **i**: insensible a mayúsculas
- **g**: devuelve array con **todos los “match”**
- **m**: multilínea, **^** y **\$** representan principio y fin de línea

```
"canciones".match(/[aeiou]/g)    => ['a', 'i', 'o', 'e']
```

```
"canciones".match(/c[aeiou]/g)   => ['ca', 'ci']
```

```
"Hoy dice hola".match(/ho/i)     => ['Ho']
```

```
"Hoy dice hola".match(/ho/ig)    => ['Ho', 'ho']
```

```
"Hola Pepe\nHoy vás".match(/^Ho/g)  => ['Ho']
```

```
"Hola Pepe\nHoy vás".match(/^ho/gim) => ['Ho', 'Ho']
```



RegExp II: Repetición y alternativa

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

Operadores de Repetición

◆+ (una o más veces):	/a+/	casa con:	“a”, “aa”, “aaa”, ..
◆? (cero o una vez):	/a?/	casa solo con:	“” y “a”
◆* (cero o más veces):	/a*/	casa con:	“”, “a”, “aa”, “aaa”, ...
◆{n} (n veces):	/a{2}/	casa solo con:	“aa”
◆{n,} (n o más veces):	/a{2,}/	casa con:	“aa”, “aaa”, “aaaa”, ...
◆{n,m} (entre n y m veces):	/a{2,3}/	casa solo con:	“aa” y “aaa”

```
"tiene".match(/[aeiou]+/g) => ['ie', 'e'] // cadenas no vacías de vocales
```

```
"tiene".match(/[aeiou]?/g) => ['', 'i', 'e', '', 'e', ''] // vocal o nada
```

```
"tiene".match(/[aeiou]*/g) => ['', 'ie', '', 'e', ''] // cadenas de vocales (incluyendo “”)
```

```
"Había un niño.".match(/[a-zñáéíóú]+/ig) => [ 'Había', 'un', 'niño' ]  
// casa con palabras en castellano: ascii extendido con ñ, á, é, í, ó, ú
```

Repetición ansiosa o perezosa

- ◆ Los operadores de repetición son “ansiosos” y reconocen
 - ▶ la **cadena más larga posible que casa con el patrón**
- ◆ Pueden volverse “perezosos” añadiendo “?” detrás
 - ▶ Entonces reconocen la **cadena más corta posible**

<code>"aaabb".match(/a+/)</code>	<code>=> ['aaa']</code>
<code>"aaabb".match(/a+?/)</code>	<code>=> ['a']</code>
<code>"ccaaccbccaa".match(/.+cc/)</code>	<code>=> ['ccaaccbcc']</code>
<code>"ccaaccbccaa".match(/.+?cc/)</code>	<code>=> ['ccaacc']</code>

Patrones alternativos

- ◆ “|” define dos patrones alternativos, por ejemplo
 - ▶ `/[a-z]+/` casa con palabras escritas con caracteres ASCII
 - ▶ `/[0-9]+/` casa con números decimales
 - ▶ `/[a-z]+|[0-9]+/` casa con palabras o números

```
"canciones".match(/ci|ca/)      => [ 'ca' ]
```

```
"canciones".match(/ci|ca/g)     => [ 'ca','ci' ]
```

```
"1 + 2 --> tres".match(/[a-z]+|[0-9]+/g)  => ['1', '2', 'tres']
```



RegExp III: Subpatrones y sustituciones

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

Subpatrones

- ◆ Dentro de un patrón podemos delimitar subpatrones
 - ▶ Un subpatrón es una parte del patrón delimitada entre paréntesis
- ◆ Por ejemplo `/(c)([aeiou])` tiene dos subpatrones
 - ▶ `(c)` es el primer subpatrón
 - ▶ `([aeiou])` es el segundo subpatrón y así sucesivamente
- ◆ **"string".match(/patrón/)** busca patrón y subpatrones en el string
 - ▶ Devuelve array: `[match, match_subpatron_1, match_subpatron_2, ...]`

```
"canciones".match(/(c)([aeiou])/)    => ['ca','c', 'a']
```

```
"canciones".match(/c([aeiou])n/)    => ['can', 'a']
```

```
"canciones".match(/(..)(..)(..)/)    => ['cancio', 'ca', 'io']
```

Sustitución de patrones

- ◆ La clase String tiene el **método replace()** para sustituir patrones
- ◆ La expresión: `"string".replace(/patrón/, x)`
 - ▶ devuelve "string" sustituyendo el primer match de "patrón" por x
- ◆ El patrón también puede tener controles i, g y m
 - ▶ i: insensible a mayúsculas
 - ▶ g: devuelve string con todos los “match” sustituidos
 - ▶ m: multilínea, ^ y \$ representan principio y fin de línea

`"Número: 142719".replace(/1/, 'x')` \Rightarrow `'Número: x42719'`

`"Número: 142719".replace(/1/g, 'x')` \Rightarrow `'Número: x427x9'`

`"Número: 142719".replace(/[0-9]+/, '<número>')` \Rightarrow `'Número: <número>'`

Sustitución con subpatrones

- ◆ Dentro de un patrón podemos delimitar subpatrones
 - ▶ Un subpatrón se delimita con paréntesis
- ◆ Por ejemplo `/([ae]+)([iou]*)/` tiene dos subpatrones
 - ▶ `$1` representa el match del primer subpatrón
 - ▶ `$2` el match del segundo y así sucesivamente
 - ▶ `$3`

```
"Número: 142,719".replace(/([0-9]+)(,[0-9]*)?/, '$1')    => 'Número: 142'
```

```
"Número: 142,719".replace(/([0-9]+)(,[0-9]*)?/, '0$2')    => 'Número: 0,719'
```

```
"Número: 142,719".replace(/([0-9]+)(,[0-9]*)?/, '$1.$2')    => 'Número: 142.719'
```



Final del tema
Muchas gracias!