



Introducción a JavaScript 6-7-8-9 (ES6 o ES2015, ES7 o ES2016, ..)

Índice

- [Tipos primitivos y operadores](#)
- [Expresiones con variables, operadores](#)
- [Números y módulos Math y Number](#)
- [String y UNICODE](#)
- [Boolean, operadores lógicos y Decisiones](#)
- [Bucles](#)

JavaScript

- ◆ Lenguaje de programación diseñado en 1995 por Brendan Eich
 - Para animar páginas Web y realizar aplicaciones en el Navegador Netscape
 - ◆ Hoy se ha convertido en **el lenguaje de programación** más utilizado en **Internet**
- ◆ JavaScript tiene pocos elementos, pero muy genéricos y potentes
 - Tiene literales muy expresivos, y funciones, objetos y tipado débil muy potentes
 - ◆ JavaScript tiene además algunas partes mal diseñadas, que se recomienda no utilizar
- ◆ Este **curso** se centra en la partes buenas (**Good parts**) de JavaScript
 - Libro: **JavaScript: The Good Parts**, Douglas Crockford, O'Reilly Media, 2008
- ◆ Existen herramientas para comprobar el buen uso:
 - **jslint**: <http://www.jslint.com/>
 - **eslint**: <http://eslint.org/>
 - ◆ Muchos editores incluyen estas herramientas y avisan cuando no se utilizan las partes buenas
 - Por ejemplo, ATOM, Sublime Text 3, Visual Studio, Webstorm, Eclipse, ..
- ◆ JavaScript ha sido portado a otros entornos
 - Aplicaciones de servidor, de escritorio, sistemas empotrados, etc.
 - ◆ Ryan Dahl crea **node.js** en 2009 para crear **aplicaciones de servidor** con gran éxito

ECMAScript

◆ JavaScript sigue la norma ECMA-262

- Publicada por European Computer Manufacturers Association
 - <https://www.ecma-international.org/publications/standards/Ecma-262.htm>
- El lenguaje ha pasado por múltiples versiones desde que apareció
 - Las siglas ESversión o ESaño identifican las versiones, por ejemplo ES6 o ES2015 para JavaScript 6

◆ ECMA-262 está en fase de transición de **ES5** a **ES6, ES7, ES8, ES9**

- **ES5** - ECMAScript 5.1, ES2011 o JavaScript 5 (Jun. 2011)
 - Se desarrolló junto con HTML5 y está soportada por node.js y todos los navegadores actuales
- **ES6** - ECMAScript 6, ES2015 o JavaScript 6 (Jun. 2015)
 - Incluye muchas mejoras y está soportada por node.js y en fase de transición en los navegadores
- **ES7** (2016), **ES8** (2017), **ES9** (2018), ... se añaden mejoras anualmente

◆ Documentación

- La documentación de JavaScript 6-7-8-9 utilizada para preparar este curso es
 - **Buenos tutoriales de JavaScript 6:** <https://javascript.info> <https://t.co/nLNMwtgHV5>
 - **Listados de mejoras:** <http://es6-features.org/>, <https://github.com/lukehoban/es6features#readme>
 - **Libro online:** Exploring ES6, Axel Rauschmayer, (<http://exploringjs.com/es6/>)
- La documentación más completa de JavaScript es de Mozilla Developer Network (MDN)
 - **Guía:** <https://developer.mozilla.org/es/docs/Web/JavaScript>
 - **Guía en inglés** (mas completa): <https://developer.mozilla.org/en/docs/Web/JavaScript>

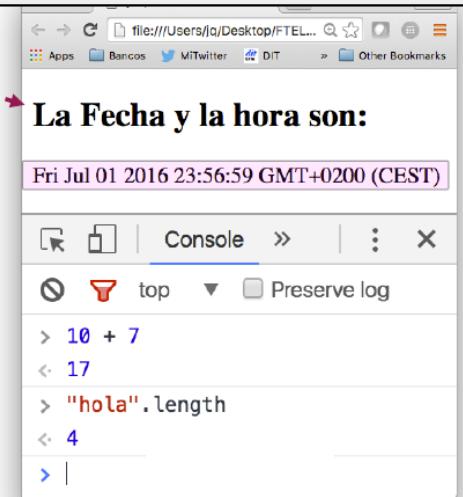
JavaScript en el Navegador

- ◆ JavaScript se extiende al navegador con
 - Objetos de interacción con el entorno y el usuario
 - ◆ **window**
 - objeto de acceso a los elementos del navegador
 - ◆ **DOM** (Document Object Model)
 - objeto **document** de acceso a elementos la página HTML
- ◆ Los navegadores ejecutan programas de 2 formas
 - Ejecutan **scripts** JavaScript en una página HTML
 - Ejecutan sentencias paso a paso en la **consola**
- ◆ La adaptación de los navegadores a ES6 es lenta
 - Todos los navegadores existentes deben adaptarse
 - ◆ Ver tabla de soporte: <https://kangax.github.io/compat-table/es6/>
 - Existen librerías que traducen ES6 para el browser
 - ◆ Babel: <https://babeljs.io>
 - ◆ Traceur: <https://github.com/google/traceur-compiler>
 - ◆ etc.

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

<body>
<h2> La Fecha y la hora son:</h2>
<div id="fecha"></div>

<script type="text/javascript">
  document.getElementById("fecha")
    .innerHTML = new Date();
</script>
</body>
</html>
```



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

Tipos primitivos y objetos (ES5 y ES6)

◆ Tipos primitivos

■ number

- Literales de números: 32, 1000, 3.8



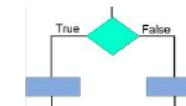
■ string

- 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()



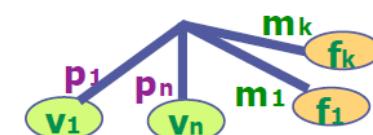
■ undefined

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

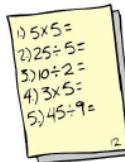
◆ Objeto: agregación estructurada de propiedades y métodos

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

- Objeto null: valor especial que representa objeto nulo



Números y operadores



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 'entrecomillado'** "

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

◆ Operador + concatena strings, por ejemplo

- "Hola" + " " + "Pepe" => "Hola Pepe"

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

- "Pepe".length => 4
- "Hola Pepe".length => 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)

$$13 + 7 \Rightarrow 20$$



◆ Signo de un número

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

$$+13 \Rightarrow 13$$



◆ Concatenación de strings

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

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

A B C D E

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

A B C D E

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

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



```
> 13 + 7
20
> "13" + "7"
'137'
> "13" + 7
'137'
> +"13" + 7
20
>
```

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.

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**

Evolución del estado en función
de la instrucción ejecutada

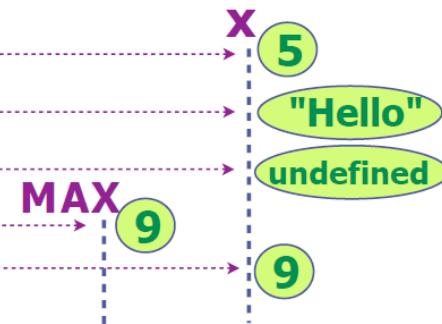
```
var x = 5;          // Creates variable x with initial value 5
x = "Hello";       // Assigns string "Hello" to variable x
x = undefined;     // Assigns undefined to variable x
x = new Date();    // Assigns Date object to variable x
```



Constantes y variables (ES6)

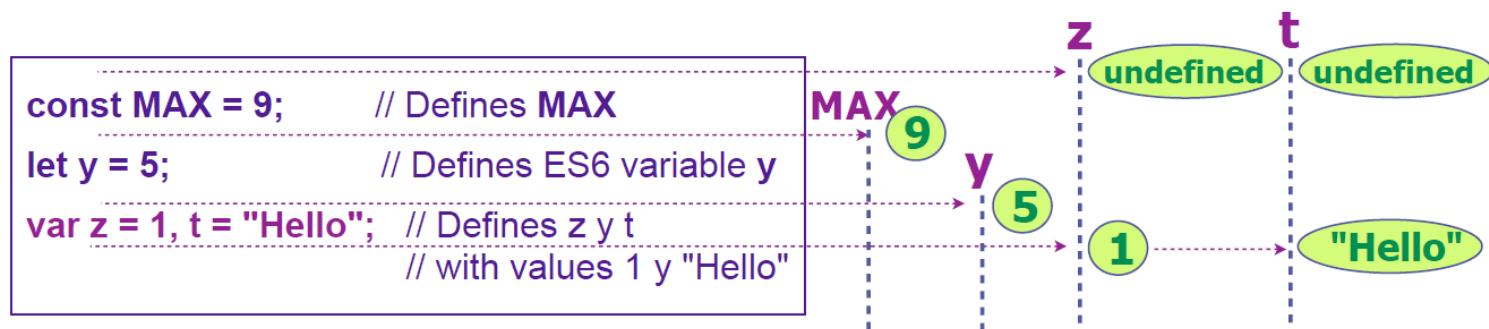
- ◆ En **ES6** las variables también se declaran con la palabra reservada **let**
 - Los valores se asignan a las variables también con el operador: **=**
- ◆ Características de las variables ES6
 - También son **no tipadas** y pueden contener cualquier valor como en ES5
 - Modifican el estado de forma 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

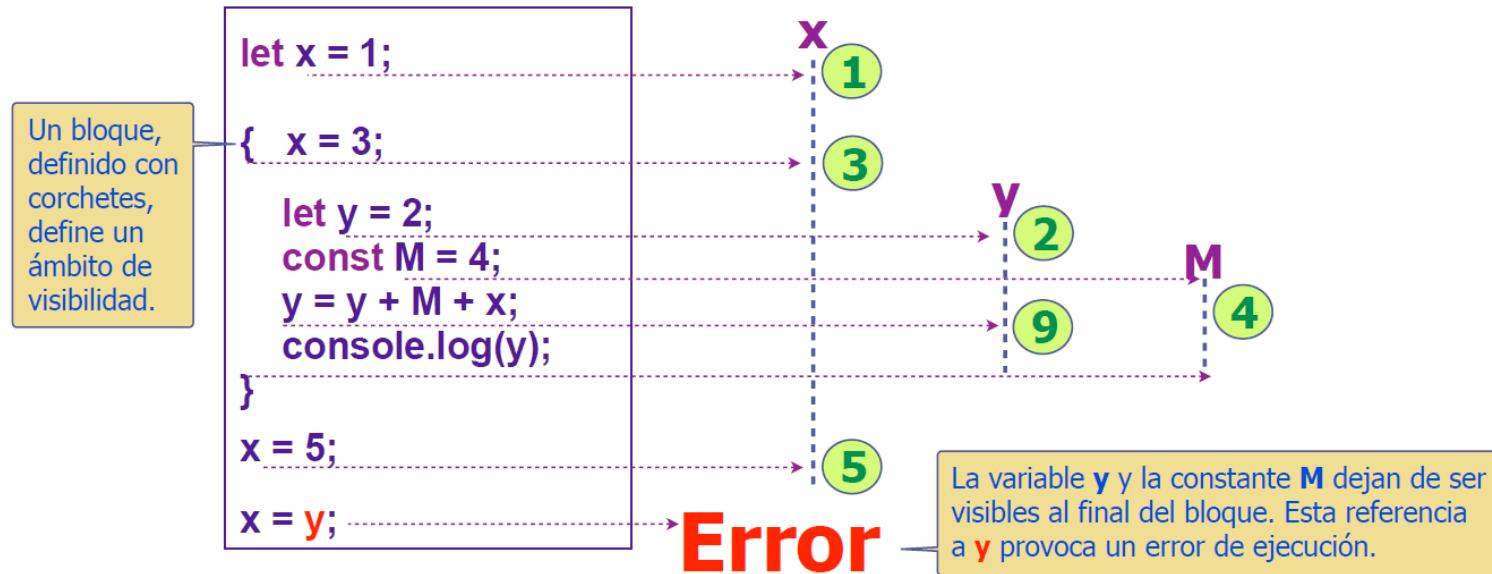
- ◆ La declaración de variables con **var** pertenece a las partes malas de JavaScript
 - son visibles en todo el ámbito del programa, incluso antes de haberlas definido
 - ◆ Esta característica se denomina hoisting y es muy engañosa
 - Por ejemplo, las variables z y t del ejemplo son visibles antes 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 construidos**: x, ya_vás, \$A1, \$, _43dias
 - Nombres **mal construidos**: 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

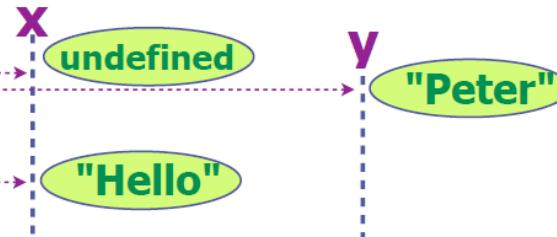
Recomendaciones sobre sintaxis

- ◆ 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 linea (\n)**, si no provoca ambigüedad
 - Para evitar ambigüedades **se recomienda** terminar **todas** las sentencias con **punto y como** (;
 - 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.jslint.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

```
// Unrecommended use of  
// delimiters
```

```
let x; let y = "Peter"
```

```
x = "Hello"
```



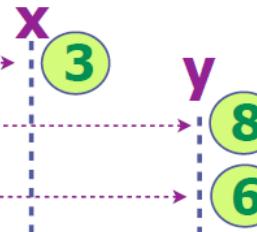
Expresiones con variables, operadores
++, --, +=, *=, -=, /=, %=, etc.

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**

```
let x = 3;  
let y = 5 + x;  
y = y - 2;  
x = y + z;
```

Error



El **error** se produce porque la **variable z** no ha sido definida.

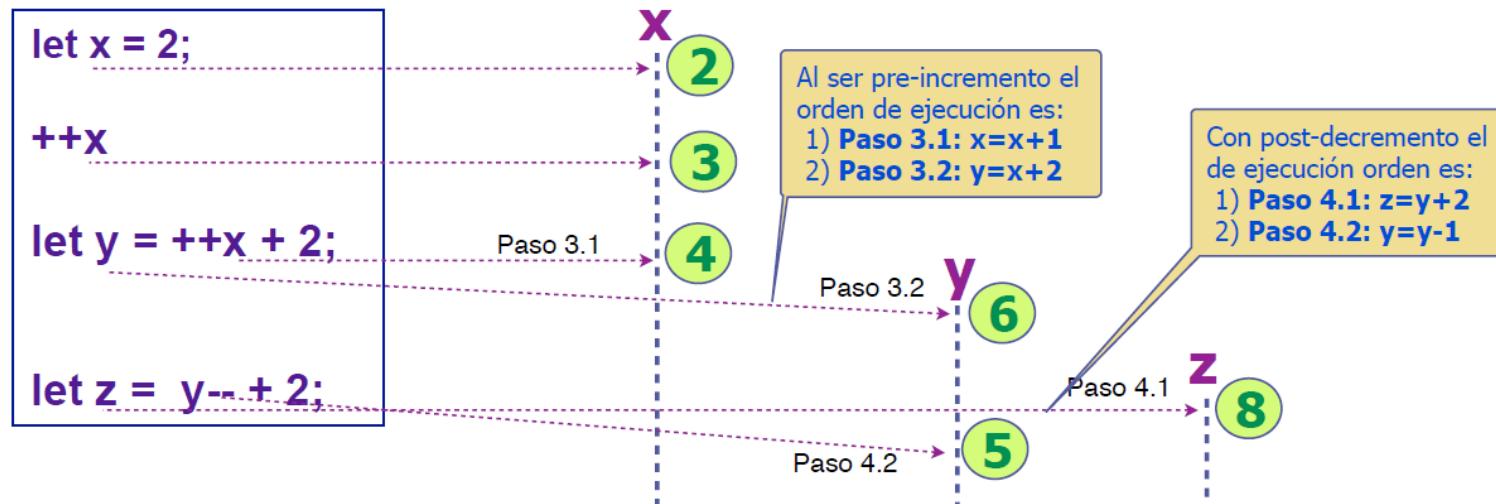
Pre y post auto incremento o decremto

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

- **++** 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/decremto se realiza **antes** de evaluar la expresión
 - Si **++/-** se aplica por la **derecha** (**post**) se incrementa/decremto **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>

```
let x = 3; // Create x with value 3  
-----  
x += 2; // add 2 to x  
-----  
x *= 3; // multiply x by 3  
-----  
x %= 4; // remainder of dividing by 4
```



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

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 5×10^{-324}
 - ◆ Entero máximo: 9007199254740992
- Cualquier literal se convierte a este formato

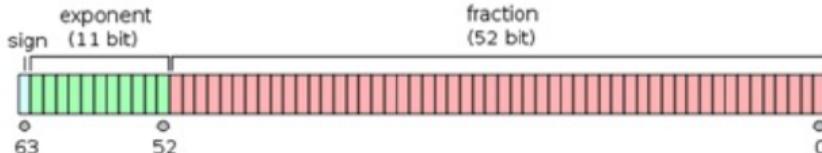
```
+''xx' => NaN      // no representable
+10/0  => Infinity // Infinito matemático
-10/0  => -Infinity // Infinito matemático

                                // 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

- $\sin(x)$, $\cos(x)$, $\tan(x)$, $\text{asin}(x)$,
- $\log(x)$, $\exp(x)$, $\text{pow}(x, y)$, \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

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

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 'entrecomillado' "
 - ◆ comillas y apóstrofes se pueden anidar: 'entrecomillado' 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`

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 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 lineas" el retorno de línea (\n) separa las dos líneas.
- En "Dos \u000A lineas" las líneas se separan con el código UNICODE \u000A equivalente a \n.

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

Métodos de String



- ◆ 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

```
jq - bash - 36x23
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

<code>String(4)</code>	=> "4"
<code>String(-4)</code>	=> "-4"
<code>String(NaN)</code>	=> "NaN"
<code>String(Infinity)</code>	=> "Infinity"
<code>String(undefined)</code>	=> "undefined"

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

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

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 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

<code>0 === 0</code>	=> true
<code>0 === 0.0</code>	=> true
<code>0 === 0.00</code>	=> true

<code>0 === 1</code>	=> false
<code>0 === false</code>	=> false

<code>'2' === "2"</code>	=> true
<code>'2' === "02"</code>	=> false

<code>" === ""</code>	=> true
<code>" === " "</code>	=> 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**, ..

- Las 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

<code>true && true</code>	=> true
<code>false && true</code>	=> false
<code>true && false</code>	=> false
<code>false && false</code>	=> false

<code>0 && true</code>	=> 0
<code>1 && "5"</code>	=> "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

<code>true true</code>	=> true
<code>false true</code>	=> true
<code>true false</code>	=> true
<code>false false</code>	=> false

<code>undefined 1</code>	=> 1
<code>13 1</code>	=> 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

◆ 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, ...

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

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

```
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

◆ 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.

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

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

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

```
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`);  
}
```

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

`\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: `{}`

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".

```
$.  
$. 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!");
```

```
}
```

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

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



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

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

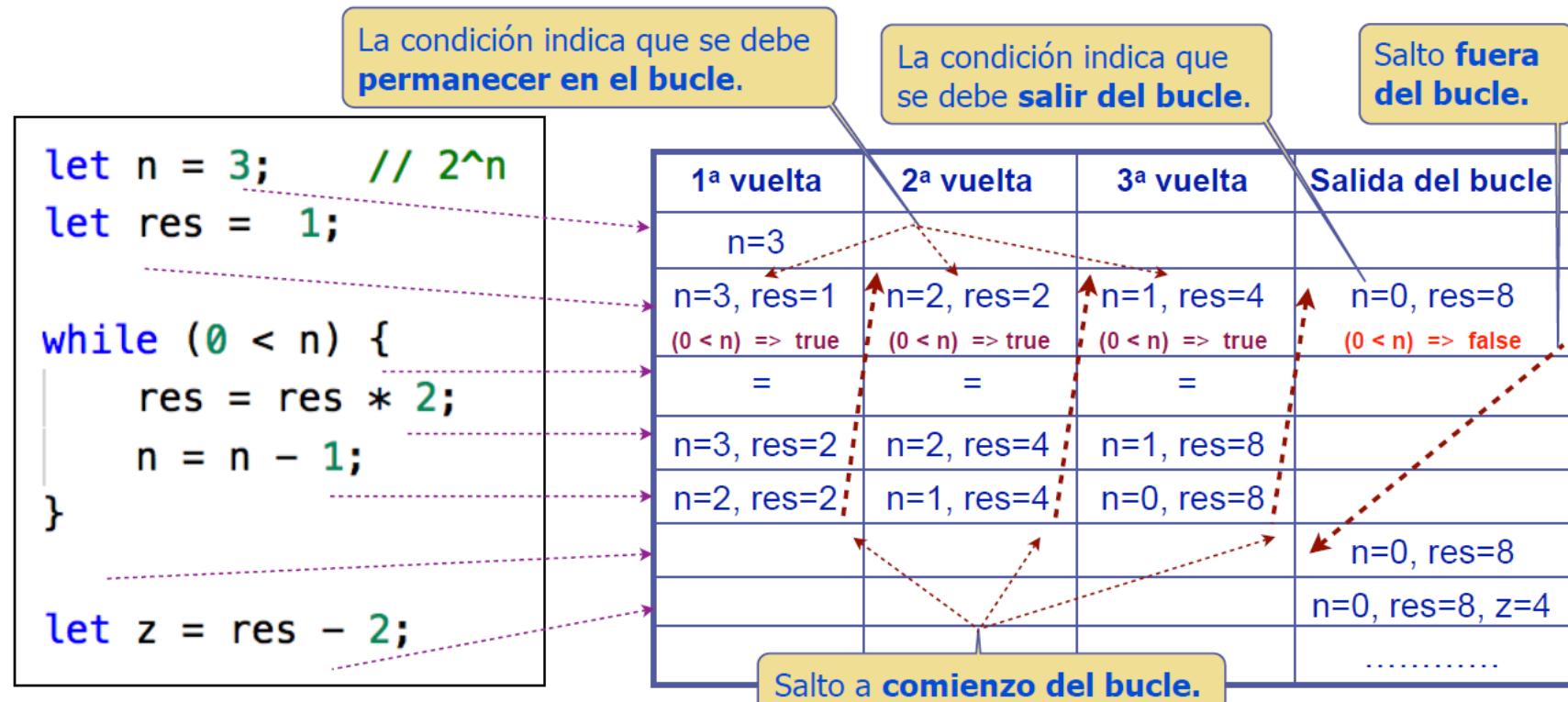
- mientras se cumple una **condición** de permanencia
 - Las variables que controlan el bucle deben iniciarse antes de comenzar
- Lo ilustramos con un bucle **while** que calcula d **2ⁿ** ($2 \times 2 \times \dots \times 2$ n veces)
 - Además existen otros tipos de bucles que no vemos aquí: **for**, **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 1^a iteración
 - La iniciación se realizar aquí en instrucciones anteriores a la sentencia del bucle
- **Condición de permanencia**: controla la finalización del bucle
 - El bucle se ejecuta mientras la condición sea **true**
- **Bloque de acciones**: acciones realizadas en cada iteración del bucle
 - Realiza el cálculo de forma iterativa hasta que la condición de permanencia indica 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

Condición de **permanencia**

Acción de **iniciación** de bucle:
contiene la definición de la
variable i que solo es visible
dentro del bucle al haber sido
definida con **let**.

Acción de **final** de bucle

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

for (let i = 0; i < n; ++i) {
    res = res * 2;
}

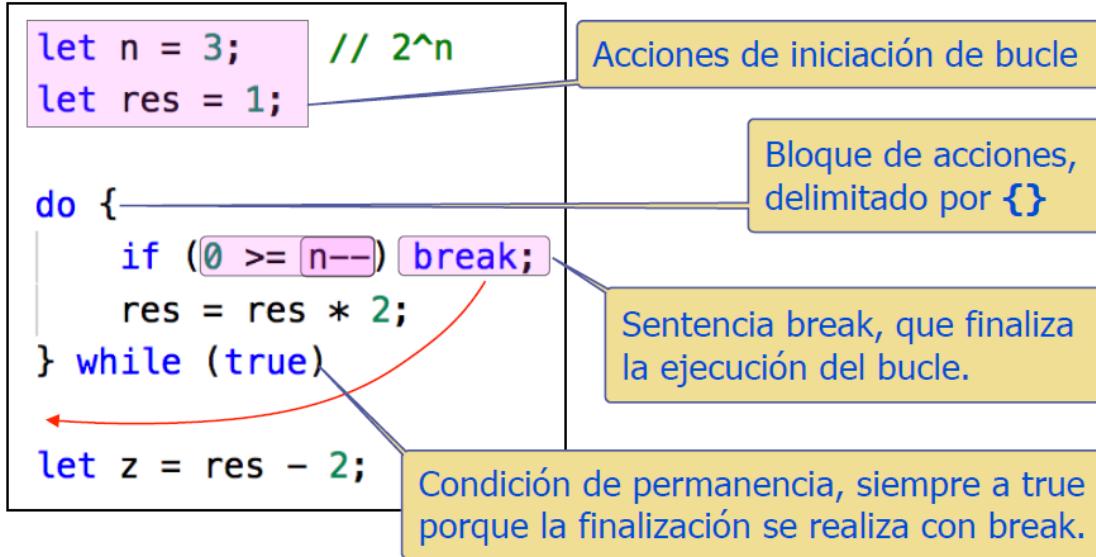
let z = res -2;
```

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 cálculo 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**.