



# Desarrollo Web en Entorno Cliente

Alicia García Martín  
CIFP Camino de la Miranda – 2024/25



**02 Manejo de la sintaxis del lenguaje JavaScript**



# Contenidos

- Etiquetas y ubicación del código.
- Tipos de datos. Conversiones entre tipos de datos. Literales.
- Variables. Declaración y convenciones. Tipos especiales y definidas por el usuario. Ámbito.
- Asignaciones.
- Operadores.
- Expresiones.
- Comentarios al código.
- Sentencias.
- Bloques de código.
- Decisiones.
- Bucles.



**Etiquetas y ubicación del código**

# Etiquetas y ubicación del código

index.html

```
<script>
document.getElementById('elemento').style.color = "red";

function buttonOnClick() {
    const languages = ['HTML4', 'HTML5', 'Common Lisp', 'C++', 'JavaScript', 'C#', 'Haskell'];
    const random = Math.floor(Math.random() * languages.length);
    document.getElementById('elemento').innerHTML = languages[random];
}
</script>
```

Sirve para breves fragmentos de código, pero no es deseable en general, ni se considera el mejor método

# Etiquetas y ubicación del código

index.html

```
<script src="js/main.js"></script>
```

```
document.getElementById( elementId: 'elemento').style.color = "red";  
  
function buttonOnClick() {  
    const languages = ['HTML4', 'HTML5', 'Common Lisp', 'C++', 'JavaScript', 'C#', 'Haskell'];  
    const random = Math.floor( x: Math.random() * languages.length);  
    document.getElementById( elementId: 'elemento').innerHTML = languages[random];  
};
```

main.js



# Tipos de datos



# Tipos de datos primitivos

**number** 0, 1, -10, 0.7, 2.38e-14, NaN, Infinity, ...

**string** "Hola mundo", 'hola', `hola \${nombre}`, ...

**boolean** true, false

**function** function(...){ ... }

**object** [2, 3, 4], { foo: "bar" }, null, ...

**undefined** undefined

**symbol** Symbol() ...

# number

number

0, 1, -10, 0.7, 2.38e-14, NaN, Infinity, ...

JavaScript no distingue entre datos numéricos enteros o en coma flotante: internamente todos son números en coma flotante.

La constante Infinity representa cualquier valor mayor que el máximo representable en doble precisión (1.79769313486231570e+380).

- Infinity es mayor que cualquier número
- Infinity sumado o multiplicado a cualquier número es Infinity
- Cualquier número menos Infinity es -Infinity
- Cualquier número entre Infinity es cero

La constante NaN representa una condición anómala.

- NaN comparado con cualquier valor (incluido NaN) es false
- Si uno de los operandos es NaN, el resultado de la operación es NaN
- Para detectar NaN necesitamos la función isNaN()

# number

**number**      0, 1, -10, 0.7, 2.38e-14, NaN, Infinity, ...

Un **literal** es un valor que aparece de forma explícita en el programa:

```
0.2      "hola mundo"      null  
function() { return false; }
```

Podemos representar literales numéricos en JavaScript de varias formas:

- Literales enteros en base 10:      0, 1, -10
- Literales enteros en otras bases numéricas:

Hexadecimal	0xC0FFEE83
Octal	0o377
Binario	0b10010100111011
- Literales en coma flotante

Notación decimal	0.10, 40.5, -0.008
Notación exponencial	1.85e55, 3.2e-5

# string

**string** "Hola mundo", 'hola', `hola \${nombre}`, ...

Un literal de tipo cadenas de caracteres en JavaScript se puede denotar con comillas simples o dobles. Ambas formas son equivalentes, salvo en que cada una admite el otro tipo de comillas como parte de la cadena:

"Una cadena con 'comillas' dobles"  
'Una cadena con "comillas" simples'

Los caracteres especiales (tabulador, nueva línea, la comilla empleada para denotar la cadena, etc) se deben escapar con el carácter '\':

\n	nueva línea	\t	tabulador	\\" el carácter "
\"	comilla doble	\'	comilla simple	
\u02A7 carácter Unicode con código hexadecimal 02A7				

# string

```
string      "Hola mundo", 'hola', `hola ${nombre}`, ...
```

Las cadenas se pueden concatenar con el operador +

```
"Una " + "cadena" < "Una cadena"
```

Además de las cadenas mencionadas, tenemos plantillas de cadena: cadenas que contienen referencias a otras variables, y que sustituyen dicha referencia por el contenido de la variable. Se denotan con una comilla invertida ` (back-tick):

```
const nombreUsuario = "Cthulhu";
console.log(`Hola ${nombreUsuario}, disfruta de tu estancia
en la Tierra`);
```

mostraría por pantalla:

```
Hola Cthulhu, disfruta de tu estancia en la Tierra
```



# boolean

**boolean**

true, false

Los siguientes valores se consideran equivalentes a `false`:

- `null`
- `undefined`
- The empty string `""`
- `0`
- `NaN`

Cualquier otro valor se considera equivalente a `true`.

# function

`function`

`function(...){ ... }`

En JavaScript las funciones son objetos (o ciudadanos) de primera clase: esto significa que son objetos en toda regla, y por lo tanto tienen todas las capacidades de un objeto, incluyendo:

- Asignarlas a una variable
- Devolverlas como valor de retorno de una función
- Pasarlas como parámetros a otra función
- Invocar métodos en ellas

Un lenguaje en el que las funciones son ciudadanos de primera clase permite emplear patrones de programación funcional con facilidad.

Una función en JavaScript puede tener o no nombre. Si no lo tiene, se llama *función anónima*.



# function

**function**

`function(...){ ... }`

La sintaxis para definir una función es:

```
function nombreFuncion(parámetros) {  
    // Cuerpo de la función  
}
```

O bien:

```
nombreFuncion = function (parámetros) {  
    // Cuerpo de la función  
}
```

Ambas son casi equivalentes, pero no se comportan 100% igual en todos los casos.

# object

object

[2, 3, 4], { foo: "bar" }, null, ...

En JavaScript los objetos difieren de los que encontramos en Java u otros lenguajes "tradicionales" de programación orientada a objetos:

- Todos los objetos son de tipo "object"
- En JavaScript no necesito clases para definir objetos
- Un objeto en JavaScript es muy similar a una estructura de Array Asociativo (diccionario, hashmap, tabla hash...) que contiene pares (clave, valor). Se accede a cada elemento (valor) mediante una clave, que suele ser de tipo String
- Puedo añadir métodos y atributos a un objeto en tiempo de ejecución
- En JavaScript se suele hablar de propiedades en lugar de atributos

# object

```
object [2, 3, 4], { foo: "bar" }, null, ...
```

Puedo crear un nuevo objeto "de la nada" mediante la notación {}:

```
var vacio = {};
```

```
// crea un objeto vacío  
  
var casa = {  
    direccion: "Calle Foo Bar, 7",  
    propietario: "Paracelso Coscojuela",  
    precio: 125000  
};
```

Para acceder a sus propiedades puedo usar dos notaciones:

```
casa.direccion equivale a casa['direccion']
```

# object

**object** [2, 3, 4], { foo: "bar" }, null, ...

Los Arrays en JavaScript son objetos con una notación particular:

```
var vacio = [];  
// crea un Array vacío
```

```
var precios = [10, 50, 100, 200, 500, 1000];
```

Para acceder a sus elementos puedo usar varias notaciones:

```
precios[0] ⚡ precios['0'] ⚡ precios.at(0) ⚡ 10
```

Los elementos de un Array pueden ser de cualquier tipo:

```
var absurdo = [0, 1, undefined, null, NaN, Infinity];
```



# object

**object** [2, 3, 4], { foo: "bar" }, null, ...

Otros objetos predefinidos que podemos encontrar en JavaScript son:

Math, Date, RegExp, BigInt, Map, Set,  
ArrayBuffer, JSON, ...

Por ejemplo, para operar con fechas usamos Date:

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

El objeto Math contiene funciones matemáticas comunes

```
let distancia = Math.sqrt(x**2 + y**2);  
let proyeccion = longitud * Math.cos(angulo);
```



# **undefined, symbol**

**undefined**

`undefined`

Representa un valor no inicializado

**symbol**

`Symbol()` ...

Es un tipo de objeto que se emplea para generar valores únicos

# Conversiones entre tipos

Table 3-2. JavaScript type conversions

Value	to String	to Number	to Boolean
undefined	"undefined"	NaN	false
null	"null"	0	false
true	"true"	1	
false	"false"	0	
"" (empty string)	0		false
"1.2" (nonempty, numeric)	1.2	true	
"one" (nonempty, non-numeric)	NaN	true	
0	"0"		false
-0	"0"		false
1 (finite, non-zero)	"1"		true
Infinity	"Infinity"		true
-Infinity	"-Infinity"		true
NaN	"NaN"		false
() (any object)	see §3.9.3	see §3.9.3	true
[] (empty array)	""	0	true
[9] (one numeric element)	"9"	9	true
['a'] (any other array)	use join() method	NaN	true
function(){} (any function)	see §3.9.3	NaN	true

JavaScript es muy flexible en cuanto a los tipos que acepta, e intenta realizar **conversiones implícitas** cuando se le pasa un tipo distinto del esperado:

```
10 + " objects"      // => "10 objects": Number 10 converts to a string
"7" * "4"            // => 28: both strings convert to numbers
let n = 1 - "x";     // n == NaN; string "x" can't convert to a number
n + " objects"       // => "NaN objects": NaN converts to string "NaN"
```

Algunas conversiones implícitas, no obstante, pueden dar resultados inesperados.

La conversión implícita de objetos a tipos primitivos es más compleja y la trataremos aparte.

# Conversiones entre tipos

Podemos hacer **conversiones explícitas** con el fin de tener mayor control sobre cómo se realizan:

```
Number("3")    // => 3  
String(false)  // => "false": Or use false.toString()  
Boolean([])    // => true
```

Todos los objetos, salvo null y undefined, tienen los siguientes métodos:

.toString()  
.valueOf()

Representación del objeto como cadena  
Representación del objeto como tipo primitivo (si es posible)

# Conversiones entre tipos

## Conversiones number → string

Number.toString(base=10)	Conversión genérica
Number.toFixed(decimales=0)	Conversión con coma fija
Number.toExponential(precision)	Conversión a notación exponencial
Number.toPrecision(precision)	Conversión con precisión especificada

```
let n = 123456.789;  
n.toFixed(0)           // => "123457"  
n.toFixed(2)           // => "123456.79"  
n.toFixed(5)           // => "123456.78900"  
n.toExponential(1)    // => "1.2e+5"  
n.toExponential(3)    // => "1.235e+5"  
n.toPrecision(4)       // => "1.235e+5"  
n.toPrecision(7)       // => "123456.8"  
n.toPrecision(10)      // => "123456.7890"
```

# Conversiones entre tipos

## Conversiones string ⇄ number

`parseInt(s, base=10)`  
`parseFloat(s)`

Conversión a entero en la base dada  
Conversión a coma flotante

<code>parseInt("3 blind mice")</code>	<code>// =&gt; 3</code>	<code>parseInt("11", 2)</code>	<code>// =&gt; 3: (1*2 + 1)</code>
<code>parseFloat(" 3.14 meters")</code>	<code>// =&gt; 3.14</code>	<code>parseInt("ff", 16)</code>	<code>// =&gt; 255: (15*16 + 15)</code>
<code>parseInt("-12.34")</code>	<code>// =&gt; -12</code>	<code>parseInt("zz", 36)</code>	<code>// =&gt; 1295: (35*36 + 35)</code>
<code>parseInt("0xFF")</code>	<code>// =&gt; 255</code>	<code>parseInt("077", 8)</code>	<code>// =&gt; 63: (7*8 + 7)</code>
<code>parseInt("0xff")</code>	<code>// =&gt; 255</code>	<code>parseInt("077", 10)</code>	<code>// =&gt; 77: (7*10 + 7)</code>
<code>parseInt("-0xFF")</code>	<code>// =&gt; -255</code>		
<code>parseFloat(".1")</code>	<code>// =&gt; 0.1</code>		
<code>parseInt("0.1")</code>	<code>// =&gt; 0</code>		
<code>parseInt(".1")</code>	<code>// =&gt; NaN: integers can't start with "."</code>		
<code>parseFloat("\$72.47")</code>	<code>// =&gt; NaN: numbers can't start with "\$"</code>		

• • •

**Variables**

# Variables

Declaración de variables: `let`

```
let i;  
let sum;  
let i, sum;  
let message = "hello";  
let i = 0, j = 0, k = 0;  
let x = 2, y = x*x;
```

Un identificador en JavaScript puede contener:

- Letras mayúsculas [A-Z]
- Letras minúsculas [a-z]
- Los caracteres \$ y \_ [\$\_]
- Números [0-9]
- Caracteres Unicode alfabéticos

No obstante, sólo puede comenzar por una letra, \$, o \_

# Variables

Declaración de constantes: const

```
const H0 = 74;           // Hubble constant (km/s/Mpc)
const C = 299792.458;   // Speed of light in a vacuum (km/s)
const AU = 1.496E8;     // Astronomical Unit: distance to the sun (km)
```

El valor de una constante ha de ser inicializado en la declaración.

Cualquier intento de modificarlo después provocará una excepción de tipo TypeError.

Hay dos "escuelas" sobre el uso de const :

- Una dice que todo lo que no cambie ha de ser const
- La otra, que sólo aquello que *no puede cambiar* ha de ser const

La segunda es históricamente más frecuente, la primera hereda de las convenciones modernas en lenguajes compilados, que pueden optimizar más agresivamente el código declarado como inmutable.



# Convenciones

No existen convenciones 100% universales, pero algunas de las más habituales son:

- Utilizamos `camelCase` para los nombres de variables
- Utilizamos `SCREAMING_SNAKE_CASE` para los nombres de constantes
- Los nombres de variables comienzan por letra minúscula
- Los campos privados de un objeto comienzan por `_`
- El carácter `$` no se utiliza demasiado a menudo

Nota:

- **camelCase**: sepáramos las palabras indicando la primera letra de cada una (salvo de la primera) con una letra mayúscula
- **snake\_case**: sepáramos las palabras mediante el carácter `_`

# Ámbito

Las variables declaradas con `let` o `const` tienen ámbito de bloque: existen desde el punto en el que se declaran hasta el final del bloque de código:

```
{  
  console.log(x);    // => TypeError: x no se ha declarado aquí  
  let x = 5;  
  console.log(x);    // => 5  
}  
console.log(x);    // => TypeError: x ha dejado de ser válida
```

Recordemos que un bloque es un fragmento de código delimitado por `{...}`.



# Ámbito: ocultación

Una declaración `let` o `const` no puede declarar de forma repetida una variable que está dentro del ámbito actual. Sí puede ocultar un nombre definido en el bloque que contiene al actual.

```
const x = 1;          // Declare x as a global constant
if (x === 1) {
  let x = 2;          // Inside a block x can refer to a different value
  console.log(x);    // Prints 2
}
console.log(x);      // Prints 1: we're back in the global scope now
let x = 3;           // ERROR! Syntax error trying to re-declare x
```

# Ámbito: ámbito global

Un identificador declarado en el nivel superior del archivo (fuera de todo bloque) tiene **ámbito global**:

- Si estamos usando Node.js, o si estamos dentro de un módulo en entorno cliente, el ámbito global corresponde al archivo en ejecución.
- Si estamos en entorno cliente tradicional, el ámbito global corresponde con todo el archivo HTML que contiene el código en ejecución.

Por tanto, una variable declarada dentro de un archivo JavaScript o fragmento `<script></script>` como variable global puede ser accedida desde un script o fragmento `<script></script>` diferente, produciendo errores inesperados e incompatibilidades entre librerías JavaScript.

Usar una variable no declarada la crea como variable global, sin importar dónde se emplea por primera vez.

# Ámbito: let vs var

Anteriormente a ES6 la única forma de declarar variables era con `var`.

Hay varias diferencias de comportamiento inesperadas en un identificador declarado con `var` frente a uno declarado con `let`:

- El ámbito es de ámbito de ejecución, no de bloque. El ámbito de ejecución de una sentencia JavaScript es la función en que se ejecuta, o global, si no está dentro de una función. La forma de simular ámbito de bloque antes de ES6 era mediante el patrón:

```
(function () { ... })();
```

- Redeclarar una variable ya declarada no produce error.
- Las variables declaradas con `var` sufren de **elevación (hoisting)**: la variable está disponible en todo su ámbito, incluso antes de la declaración, como si su declaración se hubiera elevado al comienzo de dicho ámbito. No obstante, su inicialización permanece en el lugar donde se ha escrito.

Dado que algunos de estos comportamientos son inesperados, pueden ser fuente de errores (bugs) difíciles de localizar.



# Expresiones

# Expresiones

Una **expresión** es un fragmento de JavaScript que puede ser evaluado para producir un valor. Por ejemplo:

- Un literal es una expresión cuyo valor es el del propio literal.
- Un nombre de variable es una expresión cuyo valor es el de la variable.
- Podemos construir expresiones complejas a partir de expresiones simples.

④ ¿Cuántas expresiones hay en este fragmento de código, y qué valor tienen?

```
let i = 3;  
numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];  
let numero = numeros.at(i * 2 - 1);
```

# Expresiones

```
let i = 3;  
numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];  
let numero = numeros.at(i * 2 - 1);
```

⑧ ¿Cuál es el valor de cada una de ellas?

# Operadores

La forma más común de unir expresiones simples para producir expresiones más complejas es mediante **operadores**.

Un operador combina los valores de sus operandos (típicamente dos, pero puede ser uno, o más de dos) para producir un valor nuevo.

Ejemplos:

- El operador ...\*... devuelve el producto de sus dos operandos
- El operador ...() devuelve el resultado de evaluar la función que le precede
- El operador [...] devuelve un Array cuyos elementos se indican en su interior

Puedo combinar varias de estas expresiones así formadas para producir expresiones más complejas. Cuando en una expresión aparecen varios operadores, tendremos que tener en cuenta las **reglas de precedencia** para determinar cuáles se evalúa primero.



# Expresiones

Expresiones primarias:

- Literales
- Las palabras reservadas `true`, `false`, `null`, `this`
- Referencias a variables, constantes, o propiedades del objeto global

Inicializadores de Arrays y objetos

Expresiones de definición de función

Expresiones de acceso a propiedades

Invocación

Expresión de creación de objeto

Operadores y asignaciones

# Expresiones

## Inicializadores de Arrays y objetos

```
let numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
let matrix = [[1,2,3], [4,5,6], [7,8,9]];
let sparseArray = [1, , , , 5];
let names = [
    ["Pepe", "Gotera"],
    ["Mortadelo"],
];
let p = { x: 2.3, y: -1.2 }; // An object with 2 properties

let q = {};
q.x = 2.3; q.y = -1.2; // Now q has the same properties as p

let rectangle = {
    upperLeft: { x: 2, y: 2 },
    lowerRight: { x: 4, y: 5 }
};
```

# Expresiones

## Expresiones de definición de función

```
// Devuelve el cuadrado del argumento
```

```
let square = function(x) { return x * x; };
```

Una expresión de definición de función también puede asignar un nombre:

```
let square = function sq (x) { return x * x; };
```

Se puede invocar inmediatamente sin asignar:

```
( function(x) { return x * x; } )(4); // => 16
```

No obstante, normalmente emplearemos **sentencias** para definir funciones.

# Expresiones

## Expresiones de acceso a propiedades

```
let o = {x: 1, y: {z: 3}}; // An example object
let a = [o, 4, [5, 6]];   // An example array that contains the object

o.x                      // => 1: property x of expression o
o.y.z                    // => 3: property z of expression o.y
o["x"]                   // => 1: property x of object o
a[1]                     // => 4: element at index 1 of expression a
a[2]["1"]                 // => 6: element at index 1 of expression a[2]
a[0].x                   // => 1: property x of expression a[0]
```

expression . identifier  
expression [ expression ]



# Expresiones

## Invocación

```
f(0)          // f es una expresión que evalúa a una función;  
             // 0 es la expresión argumento.  
  
Math.max(x,y,z) // Math.max es una función; x, y, z son los argumentos.  
  
a.sort()       // a.sort es la función; no hay argumentos.
```



# Expresiones

## Expresión de creación de objeto

```
new Object()  
new Point(2,3)
```

```
new Object  
new Date
```

# Expresiones

## Operadores y asignaciones

Table 4-1. JavaScript operators

Operator	Operation	A	N	Types
<code>++</code>	Pre- or post-increment	R	1	<code>lval</code> → <code>num</code>
<code>--</code>	Pre- or post-decrement	R	1	<code>lval</code> → <code>num</code>
<code>-</code>	Negate number	R	1	<code>num</code> → <code>num</code>
<code>+</code>	Convert to number	R	1	<code>any</code> → <code>num</code>
<code>~</code>	Invert bits	R	1	<code>int</code> → <code>int</code>
<code>!</code>	Invert boolean value	R	1	<code>bool</code> → <code>bool</code>
<code>delete</code>	Remove a property	R	1	<code>lval</code> → <code>bool</code>
<code>typeof</code>	Determine type of operand	R	1	<code>any</code> → <code>str</code>
<code>void</code>	Return undefined value	R	1	<code>any</code> → <code>undef</code>

# Expresiones

**	Exponentiate	R 2	num,num→num
*, /, %	Multiply, divide, remainder	L 2	num,num→num
+, -	Add, subtract	L 2	num,num→num
+	Concatenate strings	L 2	str,str→str
<<	Shift left	L 2	int,int→int
>>	Shift right with sign extension	L 2	int,int→int
>>>	Shift right with zero extension	L 2	int,int→int
<, <=,>, >=	Compare in numeric order	L 2	num,num→bool
<, <=,>, >=	Compare in alphabetical order	L 2	str,str→bool
instanceof	Test object class	L 2	obj,func→bool
in	Test whether property exists	L 2	any,obj→bool
==	Test for non-strict equality	L 2	any,any→bool
!=	Test for non-strict inequality	L 2	any,any→bool
===	Test for strict equality	L 2	any,any→bool

Operator	Operation	A	N	Types
!==	Test for strict inequality	L 2		any,any→bool
&	Compute bitwise AND	L 2		int,int→int
^	Compute bitwise XOR	L 2		int,int→int
	Compute bitwise OR	L 2		int,int→int
&&	Compute logical AND	L 2		any,any→any
	Compute logical OR	L 2		any,any→any
??	Choose 1st defined operand	L 2		any,any→any
: :	Choose 2nd or 3rd operand	R 3		bool,any,any→any
=	Assign to a variable or property	R 2		lval,any→any
**=, *=, /=, %=,	Operate and assign	R 2		lval,any→any
+=, -=, &=, ^=,  =,				
<<=, >>=, >>>=				
,	Discard 1st operand, return 2nd	L 2		any,any→any

# Sentencias

Un programa en JavaScript está compuesto de una o más **sentencias**, u órdenes que se han de ejecutar. Una sentencia es una unidad completa de ejecución.

Una expresión se evalúa para producir un valor.

Una sentencia se ejecuta para realizar una acción.

Possibles acciones que realiza una sentencia son:

Un *side effect* (efecto secundario / lateral / colateral) es un efecto de una expresión que tiene consecuencias más allá de su mero valor. Por ejemplo: una expresión de asignación cambia el valor de la variable asignada.

Control de flujo: condicionales, bucles, saltos.

# Sentencias

La sentencia más simple consiste en una única expresión.

```
saludo = "Hola " + nombreUsuario;  
i *= 3;
```

Una de las sentencias de este tipo más habituales es llamar a una función.

```
console.log(debugMessage);
```

Si la función llamada no tiene efectos secundarios, no tiene mucho sentido descartar el valor de retorno:

```
Math.cos(x);
```

Pero sí es habitual en este caso que la llamada a la función forme parte de una asignación:

```
cx = Math.cos(x);
```

# Bloques de código

Una sentencia compuesta o *bloque* consiste en cero o más sentencias rodeadas por {...}:

```
{  
    x = Math.PI;  
    cx = Math.cos(x);  
    console.log("cos(π) = " + cx);  
}
```

Por contra, la sentencia vacía consta únicamente de un terminador de sentencia:

```
;
```

Puede ser útil cuando tenemos un bucle cuyo cuerpo no necesita ejecutar nada:

```
// Initialize an array a  
for(let i = 0; i < a.length; a[i++] = 0) ;
```



# Decisiones

La sentencia condicional `if` básica tiene el siguiente aspecto:

```
if (expresión)
    sentencia
```

Primeramente evalúa *expresión*. Si el resultado, interpretado como boolean, evalúa a `true`, ejecuta *sentencia*, que puede ser una sentencia vacía, simple, o un bloque de código. De lo contrario, ignora dicha sentencia.

en lugar de...

```
if (condicion == true) hacerAlgo();
if (condicion == false) hacerAlgo();
```

se prefiere...

```
① if (condicion) hacerAlgo();
① if (!condicion) hacerAlgo();
```



# Decisiones

La sentencia condicional `if` puede tener una rama alternativa `else`:

```
if (expresión)
    sentencia1
else
    sentencia2
```

Primeramente evalúa `expresión`. Si el resultado, interpretado como boolean, evalúa a `true`, ejecuta `sentencia1`, que puede ser una sentencia vacía, simple, o un bloque de código. De lo contrario, ejecuta `sentencia2`.



# Decisiones

La sentencia condicional `if` puede tener una rama alternativa `else`:

```
if (expresión)
    sentencia1
else
    sentencia2
```

Primeramente evalúa `expresión`. Si el resultado, interpretado como boolean, evalúa a `true`, ejecuta `sentencia1`, que puede ser una sentencia vacía, simple, o un bloque de código. De lo contrario, ejecuta `sentencia2`.

# Decisiones

Hay que tener cuidado al escribir condicionales anidados:

```
i = j = 1;  
k = 2;  
  
if (i === j)  
    if (j === k)  
        console.log("i equals k");  
else  
    console.log("i doesn't equal j"); // WRONG!!
```

En este ejemplo, la rama `else` pertenece al segundo `if`, no al primero, aunque la indentación del código nos sugiera lo contrario.

Una buena forma de evitar este tipo de errores es usando **siempre** bloques de código para cada rama, aunque sólo tengan una sentencia a ejecutar.



# Decisiones

## Decisiones encadenadas

```
if (n === 1) {  
    // Execute code block #1  
} else if (n === 2) {  
    // Execute code block #2  
} else if (n === 3) {  
    // Execute code block #3  
} else {  
    // If all else fails, execute block #4  
}
```

equivale a:

```
if (n === 1) {  
    // Execute code block #1  
}  
else {  
    if (n === 2) {  
        // Execute code block #2  
    }  
    else {  
        if (n === 3) {  
            // Execute code block #3  
        }  
        else {  
            // If all else fails, execute block #4  
        }  
    }  
}
```



# Decisiones

```
switch(expresion) {  
    case 1:  
        // Ejecuta si expresion === 1  
        break;  
    case 2:  
        // Ejecuta si expresion === 2  
        break;  
    case 'foo':  
        // Ejecuta si expresion === 'foo'  
        break;  
    .  
    .  
    .  
    default:  
        // Opcionalmente, si todo falla, se ejecuta el bloque default  
        break;  
}
```

# Bucles

Los bucles básicos while y do/while funcionan exactamente igual que en Java:

```
while(count < 10) {           do {  
    console.log(count);        console.log(a[i]);  
    count++;                  } while(++i < len);  
}  
}
```

Al igual que el bucle for:

```
for(inicializador ; test ; incremento)  
    sentencia
```

que equivale (aproximadamente) a:      *inicializador*;

```
while(test) {  
    sentencia
```

*incremento*;

```
}
```

# Bucles

JavaScript además nos ofrece el bucle `for/in` para recorrer las propiedades de un objeto:

```
for(let propiedad in objeto) {  
    // Asigna a propiedad los nombres de las propiedades  
    // de objeto, como string, sucesivamente en cada iteración  
  
    console.log(objeto[propiedad]);  
}
```

# Bucles

Para objetos **iterables** (arrays, maps, sets...) es frecuente que queramos recorrer su contenido en un bucle `for`. JavaScript nos ofrece el bucle `for/of` para ello:

```
let data = [1, 2, 3, 4, 5, 6, 7, 8, 9]

let sum = 0;
for(let element of data) {
    sum += element;
}

sum           // => 45
```

# Bucles

Una cadena también es iterable por caracteres:

```
let frequency = {};
for(let letter of "mississippi") {
    if (frequency[letter]) {
        frequency[letter]++;
    } else {
        frequency[letter] = 1;
    }
}

frequency // => {m: 1, i: 4, s: 4, p: 2}
```



# Declaraciones

const, let, var, function, class, import, export



## Modo estricto

"use strict";

- ④ Primera línea de cada archivo JavaScript
- ④ Evitaremos errores usando modo estricto **siempre**

# Bibliografía

O'REILLY®

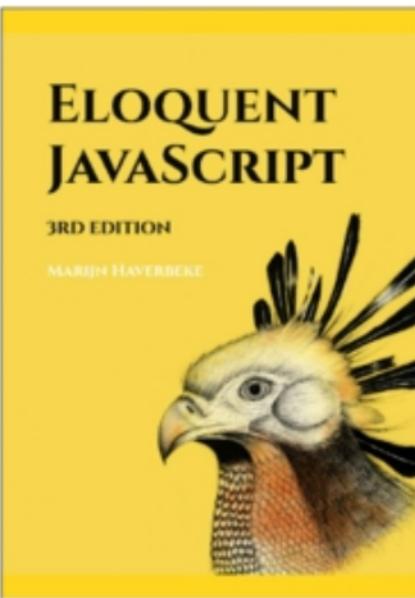
Seventh  
Edition

## JavaScript The Definitive Guide

Master the World's Most-Used  
Programming Language



David Flanagan



- è Materiales de clase en Teams
- è Material adicional en Teams



# GRACIAS!

Preguntas?

alicew.garmar@educa.jcyl.es

CREDITS: This presentation template was created by [Slidesgo](#), including icons by [Flaticon](#), infographics & images by [Freepik](#).  
This presentation contains illustrations by [Storyset](#)