

UT02: FUNDAMENTOS DEL LENGUAJE JAVASCRIPT

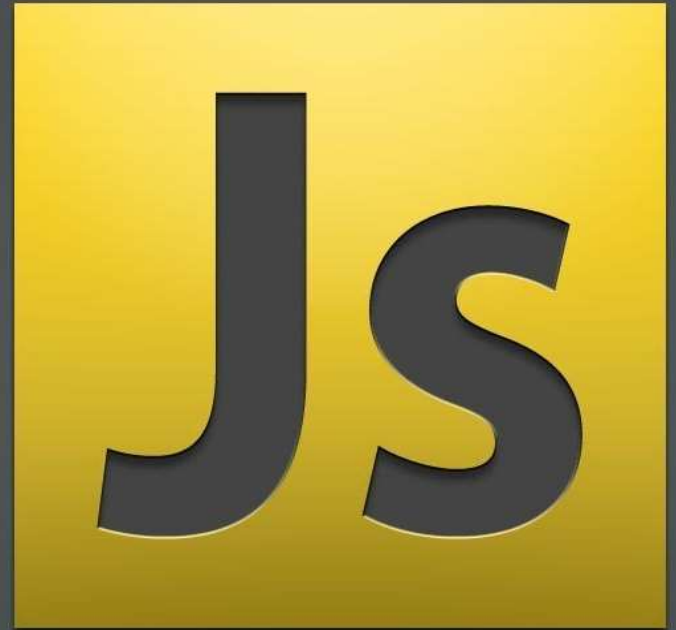
ÍNDICE

- 1.- Estructura del código
- 2.- Variables y tipos de datos
- 3.- Interacción con el usuario
- 4.- Operadores básicos y comparaciones
- 5.- Ejecución condicional
- 6.- Bucles
- 7.- La sentencia switch
- 8.- Funciones



1

ESTRUCTURA DEL CÓDIGO



Un programa está compuesto por una serie de **sentencias**.

Las sentencias se escriben en líneas separadas, pudiendo tener opcionalmente **el símbolo punto y como final de sentencia**.

```
>> console.log("Hola mundo")
```

```
Hola mundo
```

```
← undefined
```

```
>> console.log("Hola mundo");
```

```
Hola mundo
```

```
← undefined
```

Aunque el punto y coma sea opcional, lo recomendable es **ponerlo siempre** ya que el intérprete puede interpretar que se trata de una misma sentencia en diferentes líneas y mostrar errores difíciles de rastrear.

```
>> console.log("aaa")  
[1, 2].includes(2)
```

```
aaa
```

```
! ▶ Uncaught TypeError: console.log(...) is undefined  
   <anonymous> debugger eval code:2  
   [Saber más]
```

```
>> console.log("aaa");  
[1,2].includes(2)
```

```
aaa
```

```
← true
```

Javascript admite dos tipos de comentarios.

Comentarios de una línea

Se indican con **dos caracteres de barra diagonal** y se pueden poner en una línea o después de una sentencia.

Comentarios multilínea

El comentario se rodea de los símbolos `/*` y `*/`, pudiendo abarcar varias líneas

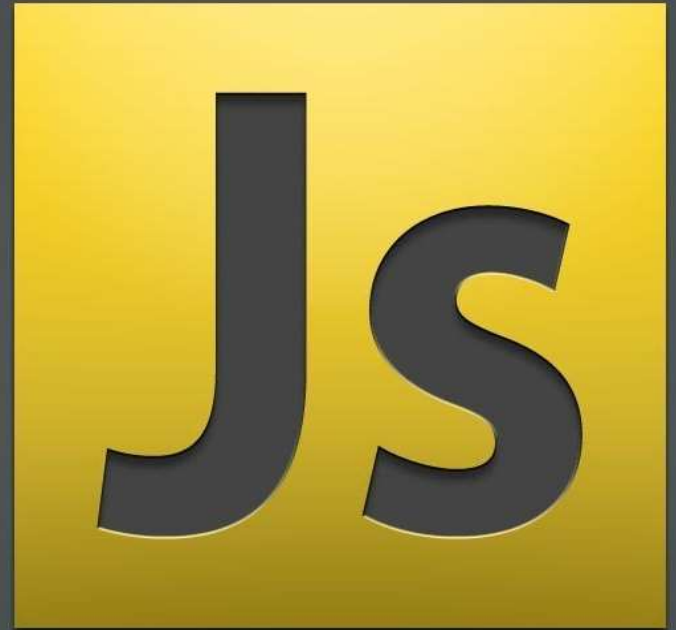
```
>> // Soy un comentario
← undefined

>> /* Y yo un comentario
    multilínea */
← undefined
```

USE STRICT

2

VARIABLES Y TIPOS DE DATOS



Una **variable** es un espacio de memoria en el que guardar datos al que se le asigna un nombre.

Declaración de la variable: es cuando se crea y se le asigna el nombre. Se hace con la palabra clave **let**.

Asignación de la variable: es cuando se le asigna un valor con el operador **=**.

```
// Declaración y asignación de variable  
let myName;  
myName = 'Victor';  
// Declaración y asignación en la misma sentencia  
let surname = 'González';  
// Varias declaraciones y asignaciones en la misma sentencia  
let surname1 = 'González', surname2 = 'Rodríguez';  
// Varias declaraciones y asignaciones en la misma sentencia  
// varias líneas. Mejora visibilidad  
let name = 'Víctor',  
    group = 'DAW2',  
    subject = 'DWEC';
```

Nomenclatura de las variables

- Solo pueden contener caracteres **alfanuméricos** o los símbolos \$ y _
- El primer carácter no puede ser un dígito
- Javascript es sensible a mayúsculas
- Permite letras de cualquier alfabeto, pero es muy recomendable **evitar cualquier carácter no anglosajón**.
- No se puede utilizar palabras reservadas como *let* o *class*
- Si no se utiliza *use strict* se puede omitir la declaración, pero es **muy mal hábito**.

Como asignar nombre a las variables

- Las variables **deben tener un nombre claro**, que describa el dato que almacena.
- Evitar nombres cortos, por ejemplo ***nc*** en lugar de ***nombreCliente***.
- Lo ideal son nombres muy descriptivos a la vez que concisos.

Hay tres formas de nombrar las variables:

- **camelCase:** todas las letras en minúsculas salvo la primera letra de cada palabra a partir de la segunda. Ej: **thingsToDo.**
- **snake_case:** todas las letras en minúsculas y utilizando el carácter guion bajo como separador de palabras. Ej.: **things_to_do**
- **PascalCase:** igual que camelCase pero la primera letra también es mayúscula: Ej.: **ThingsToDo**

Es importante ser consistente con la forma que utilicemos.

Habitualmente, en Javascript:

- **camelCase** para nombres de variables, funciones, métodos, parámetros o propiedades.
- **PascalCase** para constructores y clases.
- **snake_case** no se suele utilizar en Javascript.

```
let camelCase;  
let snake_case;  
let PascalCase;
```

Una **constante** es una variable cuyo valor no va a cambiar después de su asignación inicial.

Se declaran con la palabra clave **const**.

```
const ip = '10.0.0.5';  
ip = '192.168.1.1'; // Invalid assignment to const
```


También se pueden utilizar las constantes para almacenar valores difíciles de recordar. En ese caso se suelen nombrar con mayúsculas y guiones.

```
const PI = 3.141592;  
const EULER_NUMBER = 2.7182818;  
const COLOR_RED = '#FF0000';  
const COLOR_ORANGE = '#FF7F00';
```

¿Cuándo usar mayúsculas y cuándo no?

- Mayúsculas si el valor es conocido antes de la ejecución (***hard coded***)
- Minúsculas si el valor se asigna durante la ejecución pero no va a cambiar.

```
const COLOR_ORANGE = '#FF7F00';  
const userName = prompt('Indique su nombre:');
```

Hay una tercera forma de declarar variables utilizando la palabra clave **`var`**.

```
var saludo = 'Hola mundo!!';  
alert(saludo);
```

Es una forma **obsoleta y no debería utilizarse nunca**, aunque sí la podemos ver en código antiguo.

El problema es que **no tiene visibilidad de bloque**, sino que se visibilidad es a nivel de función.

```
// Forma CORRECTA
if (true) {
    let a = 'Hola!!';    // Ámbito de bloque
}
alert(a);                // ERROR. Fuera de ámbito

// Forma INCORRECTA
if (true) {
    var b = 'Hola!!';    // Variable GLOBAL
}
alert(b)                 // Muestra el mensaje
```

Además, **`var`** tolera redeclaraciones.

```
var user = 'Victor';  
var user = 'Pepe';           // No hay error  
  
let user = 'Victor';  
let user = 'Pepe';           // Syntax Error
```

También permite declarar la variable **después** de su asignación.

```
i = 'Hola';  
console.log(i);  
var i;
```

Este comportamiento se denomina **hoisting** (elevamiento), porque todos los var son elevados al tope de la función, incluso aunque el código del var no se llegue a ejecutar.

```
i = 'Hola';  
console.log(i);  
if (false) {  
    var i;  
}
```

Si la variable se declara dentro de una función, entonces **el ámbito se limita a la función.**

```
function saludo() {  
    userName = 'Victor';  
    console.log('Hola ' + userName);  
    var userName;  
}  
saludo();  
console.log(userName); // ERROR
```

Hay 8 tipos de datos básicos en Javascript:

- Number
- BigInt
- String
- Boolean
- El valor null
- El valor undefined
- Object y Symbol

Los valores que contienen las variables siempre pertenecen a uno de estos tipos de datos.

Al ser un lenguaje **dinámicamente tipado** la misma variable puede contener diferentes tipos de datos en diferentes momentos.

```
let message;           // Undefined
message = 'Hola!!'     // String
message = 1234;        // Number
```

Number

Representa un número **tanto entero como de punto flotante**.

Incluye también **valores numéricos especiales**:

- **Infinity y -Infinity**: representa el valor infinito.

```
let a = Infinity;  
let b = 1/0;  
let c = -7/0;  
console.log(a);      // Infinity  
console.log(b);      // Infinity  
console.log(c);      // -Infinity
```

- **NaN**: representa un error de cálculo. Cualquier operación incorrecta daría este valor. Esto quiere decir que las **operaciones matemáticas son seguras**: nunca se interrumpirá la ejecución del programa.

```
let a = 'Hola' / 2;  
console.log(a);           // NaN  
console.log(a + 7);       // NaN  
console.log(a + undefined); // NaN
```

Las operaciones con NaN son pegajosas, cualquier operación que lo involucre devuelve NaN.

BigInt

Los valores de tipo Number tienen que estar entre $-(2^{53}-1)$ y $2^{53}-1$.

Si necesitamos número mayores que estos valores debemos utilizar valores BigInt. Se indica añadiendo una **n** tras el número.

```
// La 'n' al final significa BigInt  
const bigint =  
1234567890123456789012345678901234567890n;
```

String

Son cadenas, cuyo valor se indica entre comillas.

Tres tipos de comillas:

- Comillas dobles
- Comillas simples
- Backticks (comillas invertidas)

```
let a = "Hola";      // Comillas dobles
let b = 'Hola';      // Comillas simples
let c = `Hola`;      // Backtick
```

Las comillas dobles y simples no tienen diferencias entre ellas.

El tener dos tipos permite incluirlas de forma segura en una cadena.

```
let a = "Hola, me llamo 'Victor'";  
let b = 'Hola, me llamo "Victor"';  
console.log(a);      // Hola, me llamo 'Victor'  
console.log(b);      // Hola, me llamo "Victor"
```

Los **backticks** permiten evaluar expresiones y variables dentro de una cadena encerrándolas en `${...}`

```
let myName = 'Victor';  
let surname = 'González';  
let message = `Hola, me llamo ${myName} ${surname}`;  
console.log(message);  
  
const PI = 3.141592;  
let radius = 7;  
console.log(`El perímetro es ${ 2*PI*radius } cm.`);
```

Boolean

Solo tiene dos posibles valores: true y false

```
let a = false; // false  
let b = 4 > 1; // true
```


El valor null (nulo)

Es un valor especial que representa *nada*, *vacío* o *valor desconocido*.

Indica que la variable es desconocida o está vacía por alguna razón.

```
let age = null;
```

El valor undefined

Es un valor especial que representa *valor no asignado*.

Por ejemplo, una variable contiene undefined después de su declaración hasta que le es asignado un valor.

```
let value;  
console.log(value); // undefined  
value = 7;  
console.log(value); // 7
```

Object y Symbol

Mientras que todos los tipos que hemos visto solo pueden contener una “cosa”, los objetos pueden contener colecciones de datos.

El tipo **Symbol** se utiliza para crear identificadores únicos.

El operador typeof

```
let a;  
console.log( typeof a);           // undefined  
console.log( typeof 0);           // number  
console.log( typeof 10n);         // bigint  
console.log( typeof true);        // boolean  
console.log( typeof "hola");      // string  
console.log( typeof Symbol("id")); // symbol  
console.log( typeof Math);        // object  
console.log( typeof null);        // object *  
console.log( typeof alert);       // function **  
/*  
 *   Es un error reconocido de las primeras versiones de  
 *   typeof mantenido por retrocompatibilidad.  
 **  Las funciones en realidad son objetos, pero typeof  
 *   las identifica como funciones  
 */
```

Algo importante cuando hablamos de tipos de datos son las **conversiones de tipos**.

Mientras que en otros lenguajes hay que hacerlas explícitamente, en Javascript esta conversión se hace automáticamente si es necesario, aunque también se puede realizar manualmente.

Esta conversión sigue unas **convenciones** que es muy importante conocer.

Conversión a cadena (string)

Ocurre automáticamente cuando necesitamos la forma de texto de un valor.

Ejemplo: cuando se pasa un número a la función `alert`.

También se puede forzar con la función **`String()`**

```
let a = 100;  
console.log(typeof a ); // Number  
a = String(a);  
console.log(typeof a);  // String
```

Los tipos de datos booleanos, *undefined* y *null* se convierten literalmente a la cadena.

```
console.log(String(undefined)); // 'undefined'  
console.log(String(false));    // 'false'  
console.log(String(null));     // 'null'
```

Conversión a número (number)

Ocurre automáticamente en funciones matemáticas y expresiones.

```
let a = "100";  
let b = "5";  
console.log(a / b);    // 20
```

Hay que tener cuidado porque si uno de los valores es una cadena el operador + realizará una **concatenación**

```
console.log( '1' + 5 );           // '15'  
console.log( 1 + '5' );           // '15'  
console.log( 1 + 2 + '3' + 4 );   // '334'
```


Se puede forzar la conversión con la función **Number()**

```
let a = "100";  
console.log(typeof a);           // string  
a = Number(a);  
console.log(typeof a);           // number
```

Si la cadena no es un número válido el resultado de la conversión será **NaN**.

```
let a = "a100";  
console.log( Number(a ));        // NaN
```

Otras reglas de conversión:

Valor	Se convierte en...
undefined	NaN
null	0
true and false	1 y 0
string	Se eliminan los espacios (incluye espacios, tabs <code>\t</code> , saltos de línea <code>\n</code> , etc.) al inicio y final del texto. Si el string resultante es vacío, el resultado es <code>0</code> , en caso contrario el número es "leído" del string. Un error devuelve <code>NaN</code> .

```
let a = "100";  
console.log(typeof a);           // string  
a = Number(a);  
console.log(typeof a);           // number
```

```
console.log( Number(undefined));           // NaN
console.log( Number(null));                 // 0
console.log( Number(true) );                // 1
console.log( Number(false) );               // 0
console.log( Number("  34  ") );            // 34
console.log( Number("") );                  // 0
console.log( Number('25Y') );               // NaN
let a = 2;
console.log( Number(`1${a}3`) );             // 123
```

Conversión a booleano (boolean)

Ocurre automáticamente en operaciones lógicas y manualmente con la operación **Boolean()**

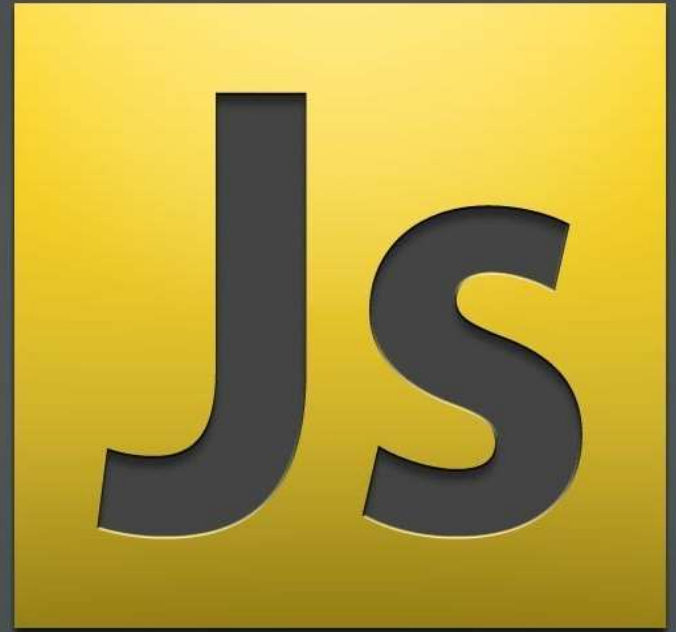
Los valores intuitivamente vacíos (0, "", null, undefined y NaN) se convierten a false.

El resto de valores se convierte a true.

```
console.log( Boolean(125) );           // true
console.log( Boolean("hola") );        // true
console.log( Boolean(null) );          // false
console.log( Boolean(undefined) );     // false
console.log( Boolean(NaN) );           // false
console.log( Boolean('') );            // false
console.log( Boolean(0) );              // false
console.log( Boolean('0') );           // true  <-
```

3

INTERACCIÓN CON EL USUARIO

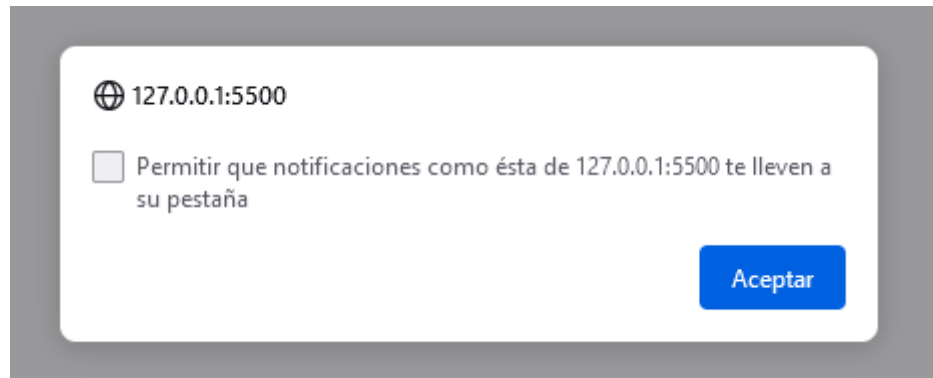


Javascript dispone de varias funciones para interactuar con el usuario: **alert**, **prompt** y **confirm**, así como la posibilidad de enviar mensajes a la **consola**.

alert

La función **alert** muestra una ventana modal que muestra un mensaje y espera a que el usuario pulse OK.

```
alert('Hola mundo!!!');
```

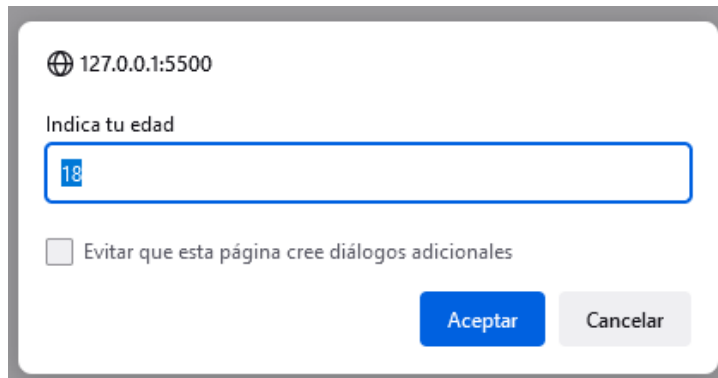


prompt

Esta función sirve para solicitar un dato al usuario. Tiene dos parámetros:

- Mensaje que se mostrará
- (opcional) Valor inicial del campo de entrada.

```
prompt('Indica tu edad', 18);
```

A screenshot of a web browser dialog box. At the top, it shows a globe icon and the address '127.0.0.1:5500'. Below that, the title 'Indica tu edad' is displayed. There is a text input field with a blue border containing the number '18'. Below the input field, there is a checkbox labeled 'Evitar que esta página cree diálogos adicionales'. At the bottom right, there are two buttons: 'Aceptar' (blue) and 'Cancelar' (gray).

El valor introducido es devuelto por la función.

Si el usuario **no introduce valor** devolverá la cadena vacía.

Si el usuario **cancela** devuelve null.

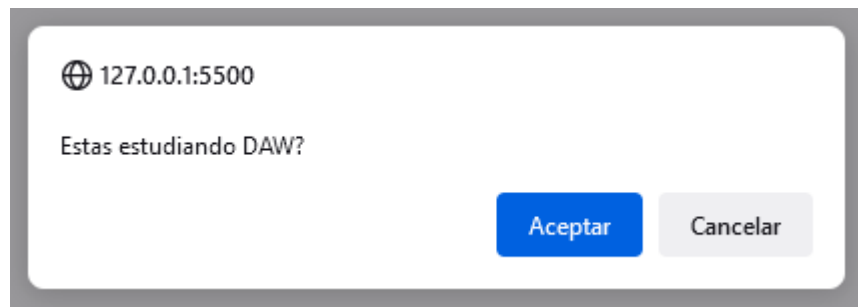
```
let age = prompt('Indica tu edad');  
console.log(`Tienes ${age} años.`);
```

confirm

Muestra una ventana modal con los botones OK y CANCELAR.

Devuelve true si se pulsa OK y false si se pulsa CANCELAR.

```
let isStudent = confirm('Estás estudiando DAW?');  
console.log(isStudent);
```



El objeto console

Este objeto provee acceso a la consola de depuración de los navegadores a través de los métodos de que dispone.

Se pueden ver todos en:

<https://developer.mozilla.org/es/docs/Web/API/Console>

```
console.log ('Mensaje de registro.');
```

```
console.info ('Mensaje de información.');
```

```
console.warn ('Mensaje de advertencia');
```

```
console.error('Mensaje de error');
```

Mensaje de registro.

📄 Mensaje de información.

⚠ Mensaje de advertencia

❗ ▶ Mensaje de error

Otra utilidad de console es medir el tiempo que se tarda en ejecutar diferentes fragmentos de código.

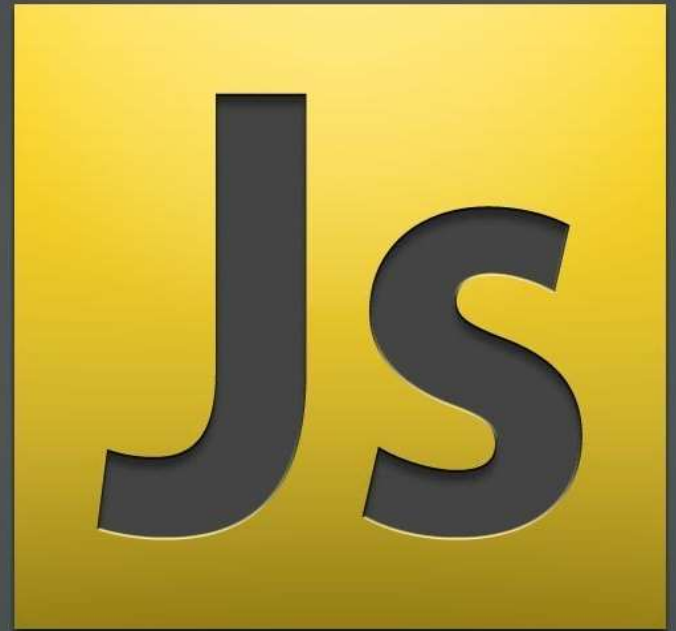
```
async function fetchData() {  
  const url = 'https://swapi.dev/api/people/1';  
  console.time('timer');      // Iniciamos cronómetro  
  let response = await fetch(url);  
  let people = await response.json();  
  console.timeEnd('timer');    // Mostramos tiempo  
  console.log(people);  
}  
fetchData();
```

```
timer: 79ms - temporizador finalizado
```

```
► Object { name: "Luke Skywalker", height: "172", mass: "77", hair_color: "blond", skin_color: "fair", e  
[...], ... }
```

4

OPERADORES BÁSICOS Y COMPARADORES



Como todos los lenguajes, Javascript dispone de una gran número de operadores.

A grandes rasgos, estos se dividen en:

- **Operadores unarios**, que tienen un único operando
- **Operadores binarios**, que tienen dos operandos.

```
let a = -7;      // Operador unario negación
let b = 4*3;     // Operador binario producto
let c = 4-3;     // Operador binario resta
```

Operadores matemáticos

Javascript soporta las operaciones de suma (+), resta (-), multiplicación (*), división (/), resto (%) y exponenciación (**)

```
console.log( 25*3 );           // 75
console.log( 25/3 );           // 8.333333333333334
console.log( 25%3 );           // 1
console.log(25**3);            // 15625
console.log( 25** $(1/2)$  );    // 5 (Raíz cuadrada)
console.log( 8** $(1/3)$  );    // 2 (Raíz cúbica)
```


Concatenación de cadenas

Si uno de los operandos del operador `+` es una cadena pasa a ser el operador de **concatenación**.

```
console.log( 'Hola ' + 'Mundo' );    // HoLa Mundo
```

Si hay operandos que no son cadenas las convertirá.

```
console.log( 'a ' + 7 );              // HoLa Mundo
console.log( 3 + 'a' );               // 3a
console.log( null + 'a' );            // nulla
console.log( NaN + 'a' );             // NaNa
```

Hay que tener cuidado con las conversiones automáticas porque el resultado en ocasiones puede ser contraintuitivo.

```
console.log( 4 + 5 + 'b' ) // 9b  
console.log( 'b' + 4 + 5 ); // b45
```

Conversión numérica, unario +

El operador + puede ser binario o unario. En el primer caso es el operador suma, en el segundo **convierte el valor a número** (luego es equivalente a *Number()*)

```
let a='75';  
console.log( a+5 );    // 755  
console.log( +a + 5 ); // 80
```

Puede ser útil para forzar la conversión cuando se piden datos al usuario por teclado, que siempre se leen como cadenas.

```
let a = prompt('Cuántos alumnos hay en DAW1?'); // Pe. 15
let b = prompt('Cuántos alumnos hay en DAW2?'); // Pe. 23
alert(`Hay un total de ${ a+b } alumnos en DAW`); // 1523
alert(`hay un total de ${ +a + +b } alumnos en DAW`); // 38
```

El problema del uso de este operador es que **se pierde legibilidad**, siendo preferible utilizar la función `Number()`.

Precedencia de operadores

Cada operador en Javascript tiene una precedencia.

Cuando se combinan varios operadores en una misma expresión se operan primero aquellos con mayor precedencia.

En caso de igual precedencia se operan de izquierda a derecha.

A continuación se muestra la tabla de precedencia de los operadores más comunes, aunque **lo más recomendable es utilizar siempre paréntesis** para dejar claro el orden en que se realizarán las operaciones.

Precedencia	Nombre	Signo
...
14	suma unaria	+
14	negación unaria	-
13	exponenciación	**
12	multiplicación	*
12	división	/
11	suma	+
11	resta	-
...
2	asignación	=
...

Tabla completa:

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/Operator_Precedence

El operador =

Al contrario que en otros lenguajes de programación, **en Javascript la asignación también es un operador.**

Su operación es almacenar un valor en una variable y **devuelve dicho valor.**

```
let x = 5;  
console.log( x=7 );    // 7
```

Esto quiere decir que podemos realizar asignaciones en cualquier punto de una expresión.

```
let a = 1;  
let b = 2;  
  
let c = 3 - (a = b + 1);  
  
console.log( a ); // 3  
console.log( c ); // 0
```

Por supuesto, esto no es nada recomendable ya que contribuye a hacer un código más ilegible.

Otra posibilidad es **encadenar asignaciones**, tal como se ve en el siguiente código.

```
let a, b, c;  
  
a = b = c = 7;  
  
console.log(a);    // 7  
console.log(b);    // 7  
console.log(c);    // 7
```

Nuevamente es algo poco legible, por lo que su uso no es aconsejado.

Operadores de asignación y operación

En ocasiones queremos realizar una operación sobre una variable y guardar el resultado en la misma.

En esas ocasiones disponemos de los operadores `+=`, `-=`, ...

```
let a = 7;  
a += 3;  
console.log(a);      // 10  
a *= 2;  
console.log(a);      // 20  
a %= 6;  
console.log(a);      // 2
```

Operadores incremento y decremento

Incrementan o decrementan en 1 el valor de una variable y devuelve el valor de la misma

```
let a = 7;  
a++;  
console.log(a);      // 8  
console.log(++a);    // 9
```

Es un operador que puede ir en forma de prefijo o de sufijo.

```
let a = 5;  
let b = 5;  
++a;  
b++;  
console.log(a);    // 6  
console.log(b);    // 6
```

Diferencia:

- **Prefijo:** incrementa el valor y lo devuelve.
- **Sufijo:** devuelve el valor y lo incrementa.

Diferencia:

- **Prefijo:** incrementa el valor y lo devuelve.
- **Sufijo:** devuelve el valor y lo incrementa.

```
let a = 5;  
let b = 5;  
console.log(++a);           // 6  
console.log(b++);           // 5  
console.log(a);              // 6  
console.log(b);              // 6
```

Operadores a nivel de bit

Los operadores a nivel de bit tratan los argumentos como números enteros de 32 bits y trabajan sobre su representación binaria. Son:

- AND (`&`)
- OR (`|`)
- XOR (`^`)
- NOT (`~`)
- LEFT SHIFT (`<<`)
- RIGHT SHIFT (`>>`)
- ZERO-FILL RIGHT SHIFT (`>>>`)

Operadores de comparación

Los operadores de comparación son:

```
console.log( 4>5 );           // false
console.log( 4<5 );           // true
console.log( 4<=5 );          // true
console.log( 4==5 );           // false
console.log( 4!=5 );           // false
console.log( 'Z'>'A' );        // true
console.log( 'Z'>'a' );        // false
console.log( 'DAW'>'DWECC' ); // false
```

Si realizamos comparaciones sobre diferentes tipos, Javascript convierte los valores a números.

```
console.log( true == 1 );    // true.  
                             // Convierte true en 1  
  
console.log( false > 0 );   // false.  
                             // Convierte false en 0
```


Igualdad estricta

El problema del operador de igualdad (==) es que realiza conversión de tipos, que en ocasiones puede que no sea lo que necesitamos.

Para solucionar este problema tenemos el operador de **igualdad estricto**, representados por ===

```
console.log( 0==false );    // true
console.log( 0===false );   // false
console.log( ''==false );   // true
console.log( ''===false );  // false
```

null y *undefined* son casos especiales.

```
console.log( null===undefined );    // false  
console.log( null==undefined );     // true
```

Un caso extraño:

```
console.log( null > 0 );           // false
console.log( null >= 0 );          // true
console.log( null == 0 );          // false
```

En comparaciones ($>$ y \geq del ejemplo) se convierten a 0.

En el operador de igualdad ($==$) no se realiza esta conversión, de forma que *null* y *undefined* solo son iguales a sí mismos.

El valor *undefined* no debe compararse con otros operadores:

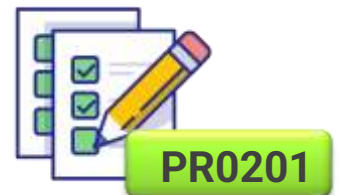
```
console.log( undefined > 0 );    // false
console.log( undefined < 0 );    // false
console.log( undefined == 0 );   // false
```

En los dos primeros se debe a que en comparaciones *undefined* se convierte a *NaN*, y ***NaN* devuelve falso en todas las comparaciones.**

En la igualdad ***undefined* solo equivale a *null*.**

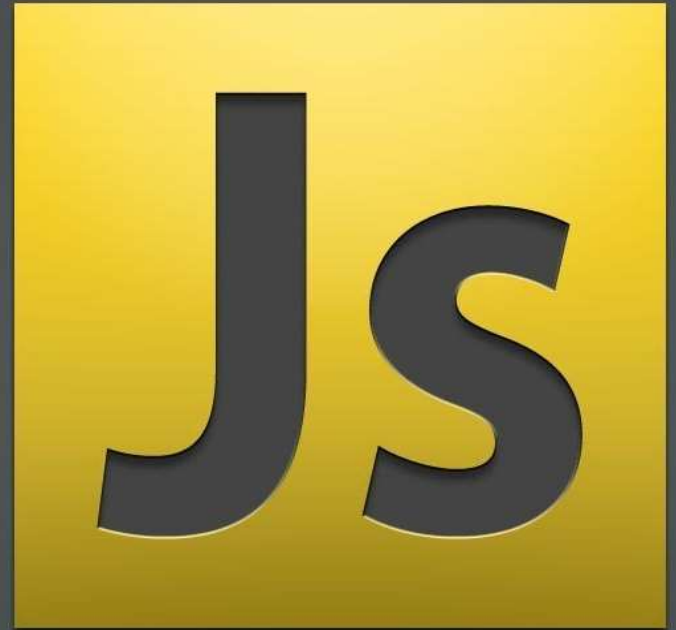
Conclusiones:

- Hay que tratar cualquier comparación con *undefined* o *null* con mucho cuidado.
- Si una variable puede tener un valor *null* o *undefined*, es mejor verificarla antes de realizar comparaciones.



5

EJECUCIÓN CONDICIONAL



La sentencia **if .. else** permite evaluar una expresión y ejecutar un código en función de si el valor de esa expresión es verdadero o falso.

```
let age = prompt('Cuántos años tienes?');  
if ( age=>18 ) {  
    alert('Eres mayor de edad.');
```

```
} else {  
    alert('Eres menor de edad.');
```

```
}
```

Si el resultado de la expresión no es un booleano realizará la conversión a booleano como vimos en el apartado anterior.

```
if ( 0 ) {  
    ...  
}
```


También se pueden enlazar diferentes *else*.

```
let year = prompt('¿En qué año fue publicada la  
especificación ECMAScript-2015?', '');  
  
if (year < 2015) {  
    alert( 'Muy poco...' );  
} else if (year > 2015) {  
    alert( 'Muy Tarde' );  
} else {  
    alert( '¡Exactamente!' );  
}
```

Javascript dispone de un **operador ternario** que permite asignar valor a una variable dependiendo de una condición.

```
let calificacion = ( nota >= 5 ) ? 'Aprobado' : 'Suspenso';
```

También se puede anidar:

```
let age = prompt('¿edad?', 18);

let message = (age < 3) ? '¡Hola, bebé!' :
  (age < 18) ? '¡Hola!' :
  (age < 100) ? '¡Felicidades!' :
  '¡Qué edad tan inusual!';

alert( message );
```

Hay otro operador que permite realizar evaluaciones condicionales denominado **nullish coalescing (fusión del null)** y que se identifica con el símbolo **??**.

El resultado de **a ?? b** será:

- Si a está definida será a
- Si a no está definida será b

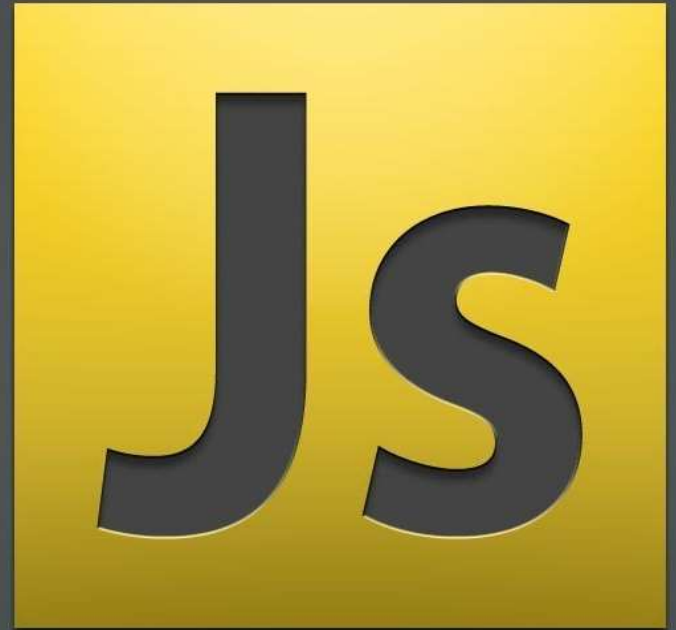
```
let user;  
console.log( user ?? 'Invitado' );
```

También se puede encadenar para devolver el primer valor definido de una serie:

```
let firstName = null;  
let lastName = null;  
let nickName = "Supercoder";  
  
// Muestra el primer valor definido:  
alert(firstName ?? lastName ?? nickName ?? "Anonymous");  
// Supercoder
```

6

BUCLES



Javascript dispone de tres constructores básicos para realizar bucles:

- `while`
- `do ... while`
- `for (...;...;...)`

Más adelante ya veremos otros dos específicos para iterar sobre objetos (`for...in`) y sobre arrays (`for...of`)

El bucle while

El bucle **while** tiene la siguiente sintaxis:

```
while ( condicion ) {  
    // Cuerpo del bucle  
}
```

El cuerpo del bucle repetidas veces mientras la condición sea true.

```
let i=0;  
while ( i<3 ) {  
    console.log(i);    // Imprimirá 0 1 2  
    i++;  
}
```

El bucle `do .. while`

En el caso del bucle **`do..while`**, el cuerpo del bucle se ejecuta **por lo menos una vez** y luego se volverá a ejecutar mientras la condición sea verdadera.

```
do {  
    // Cuerpo del bucle  
} while (condicion);
```


El bucle for

El bucle **for** se ejecuta un número fijo de veces.

```
for ( comienzo; condición; paso ) {  
    // Cuerpo del bucle  
}
```

Comienzo: se ejecuta al comienzo del bucle y se utiliza para inicializar las variables.

Condición: mientras sea verdadera se ejecutará el bucle.

Paso: se ejecuta después del cuerpo en cada iteración. Se utiliza la incrementar o decrementar la variable de forma que se acerque a la condición.

```
function factorial(num) {  
  let fact=1;  
  for ( let i=1; i<=num; i++ ) {  
    fact *= i;  
  }  
  return fact;  
}  
  
console.log( factorial(5) );
```

Se puede inicializar la variable en la propia sentencia for.

Cualquier parte del for se puede omitir siempre y cuando mantengamos los separadores (;)

```
let i=0;

for ( ; i<4; i++ ) {
  console.log(i);
}

for ( let j=6; j>=0; ) {
  console.log( j++ )
}
```

Interrupción del bucle

Se puede interrumpir la ejecución de un bucle con las sentencias `break` y `continue`

La sentencia **`break`** finaliza el bucle, pasando a ejecutar la siguiente instrucción

```
let i=0;

while ( i<10 ) {
    if (i==4) break;
    console.log(i++);    // Muestra 0 1 2 3
}
```

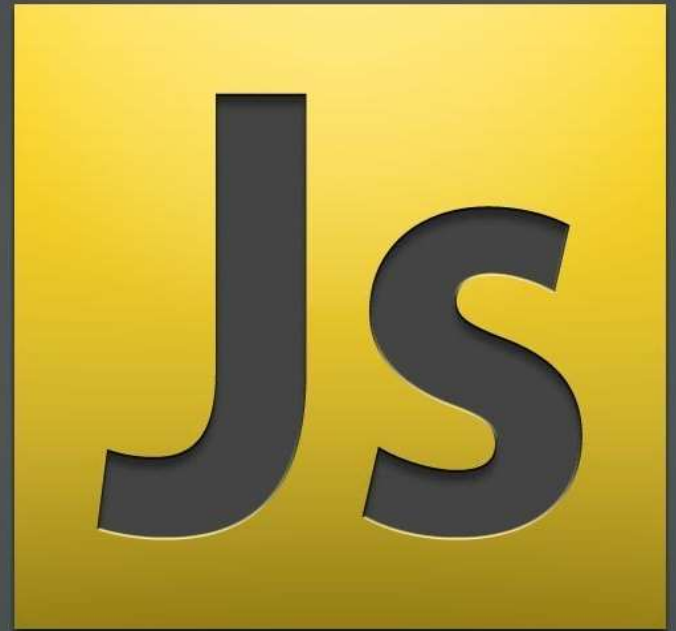
En cambio **continue** no finaliza el bucle sino que pasa a la siguiente iteración

```
let i=0;

while ( i<10 ) {
    ++i;
    if ( i == 4 ) continue;
    console.log(i);    // Muestra 0 1 2 3 5 6 7 8 9 10
}
```

7

LA SENTENCIA SWITCH



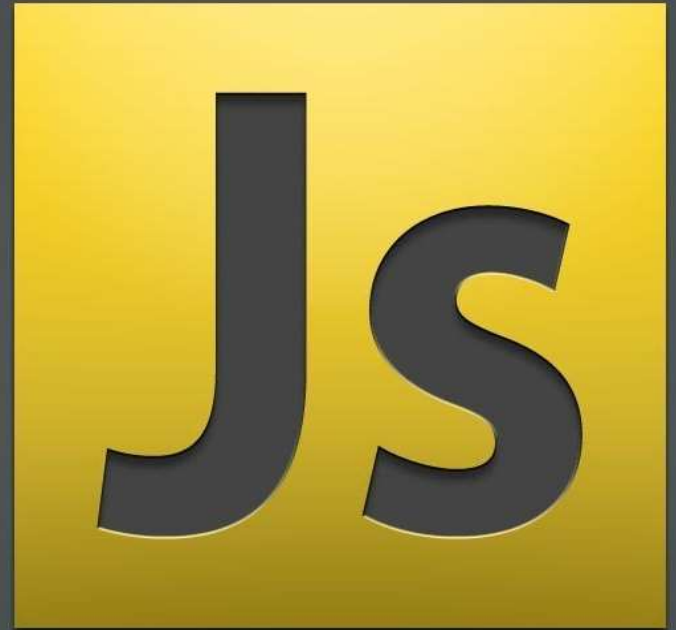
La sentencia **switch** permite evaluar una expresión y ejecutar diferentes bloques de código según el valor de la misma.

```
switch (exp) {  
    case 'valor1':  
        // Se ejecuta si exp===valor1  
        break  
    case 'valor2':  
        // Se ejecuta si exp===valor2  
        break  
    default:  
        // Se ejecuta si no se cumple ninguna de las  
        //condiciones anteriores  
}
```

```
let cod = prompt('Introduce el código del ciclo');
let ciclo;
switch (cod) {
  case 'DAW':
    ciclo='Desarrollo de Aplicaciones Web';
    break;
  case 'DAM':
    ciclo='Desarrollo de Aplic. Multiplataforma';
    break;
  case 'ASIR':
    ciclo='Admon. de Sistemas Informáticos y Redes';
    break;
  default:
    ciclo='Desconocido';
}
```


8

FUNCIONES



La sintaxis de las **funciones** en JavaScript es:

```
function showMessage() {  
    alert('Hola mundo!');  
}  
  
showMessage();
```

También se le pueden pasar parámetros:

```
function showMessage( msg, userName) {  
    alert(`${msg}, ${userName}`);  
}  
  
showMessage( 'Hola', 'Victor' );
```

Si al invocar una función no se le pasa un parámetro se le asigna el valor *undefined*.

```
function showMessage( msg ) {  
    console.log (msg);  
}  
  
showMessage( 'Hola' ); // Hola  
showMessage();        // undefined
```

Es posible indicar un **valor predeterminado** para los parámetros en caso de que el usuario no los indique.

```
function showMessage( msg ) {  
    console.log (msg);  
}  
  
showMessage( 'Hola' ); // Hola  
showMessage();         // undefined
```

En código antiguo (cuando no había parámetros por defecto) se suplía esta carencia de esta forma:

```
function showMessage( userName ) {  
    userName = userName || 'Invitado';  
    console.log ( `Bienvenido, ${userName}` );  
}  
  
showMessage( 'Victor' ); // Bienvenido, Victor  
showMessage();          // Bienvenido, Invitado
```

Aunque en esos casos sería mejor utilizar el operador de fusión del nulo (??) cuando el valor de 0 debe ser considerado normal.

Las funciones siempre devuelven un valor que se indica con la sentencia **return**. Si no hay return devolverán *undefined*.

```
function sum( a, b ) {  
    return a + b;  
}  
  
console.log( sum( 2, 3 ) ); // 5
```

Consejos con las funciones:

- Utiliza nombres representativos. Puedes usar prefijos para describir lo que hacen:
 - `get...: devuelven un valor` `-> getNif()`
 - `calc...: calculan algo` `-> calcNifLetter()`
 - `create...: crean algo` `-> createIndex()`
 - `check...: revisan algo` `-> checkPasswd()`
 - `is...: es algo. Devuelve bool` `-> isValidUser()`

- Cada funci3n debería realizar una ́nica acci3n.
- Una funci3n no debería modificar variables externas.

La forma explicada para crear funciones se llama **declaración de función**, pero no es la única forma para crearlas.

La otra forma es mediante **Expresiones de función**.

```
let saluda = function() {  
    alert("Hola");  
}  
saluda();
```

En este caso se asigna la función a una variable, pudiendo invocarla haciendo referencia a dicha variable.

Observa que para invocarla debemos utilizar la misma sintaxis que en el caso que si la declaramos, con los paréntesis.

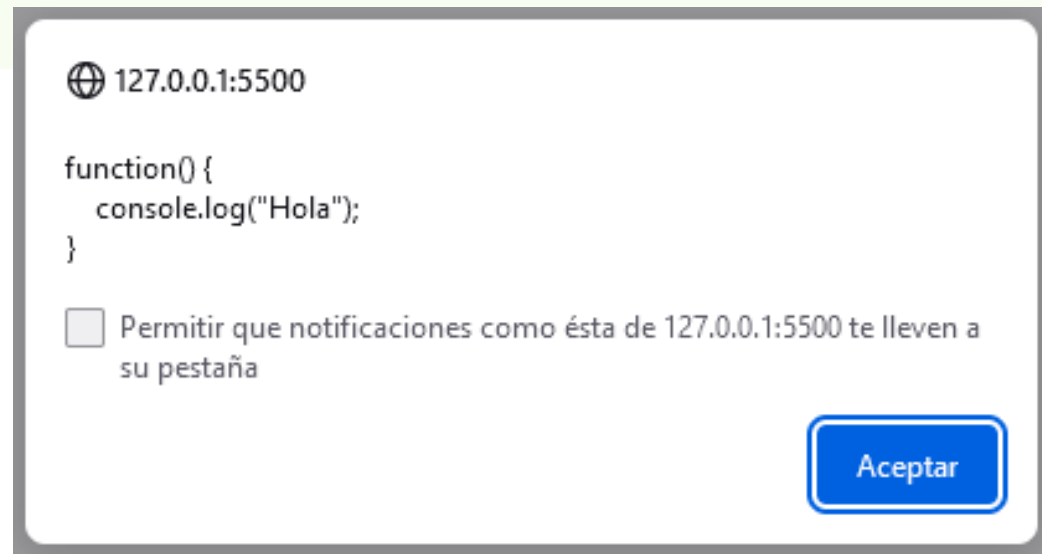
Sin embargo, también podemos mostrar la variable que contiene la función.

```
let saluda = function() {  
    console.log("Hola");  
}  
console.log(saluda);
```

```
▼ function saluda() ↗  
  arguments: null  
  caller: null  
  length: 0  
  name: "saluda"  
  ► prototype: Object { ... }  
  ► <prototype>: function ()
```

Si intentamos imprimir la función con *alert* nos la convertirá automáticamente a cadena.

```
let saluda = function() {  
  console.log("Hola");  
}  
alert(saluda);
```



Un uso muy común de las funciones en JavaScript es pasarlas como parámetro a otra función. En estas situaciones se denominan **callbacks**.

```
function ask( question, yes, no ) {  
    if ( confirm(question) ) yes()  
    else no();  
}  
function showOk() {  
    alert("Estás de acuerdo");  
}  
function showCancel() {  
    alert("No estás de acuerdo");  
}  
ask( "Estás de acuerdo?", showOk, showCancel );
```

El ejemplo anterior se puede simplificar utilizando las expresiones de función.

```
function ask( question, yes, no ) {  
    if ( confirm(question) ) yes()  
    else no();  
}  
  
ask(  
    "Estás de acuerdo?",  
    function() { alert("Estás de acuerdo"); },  
    function() { alert("No estás de acuerdo"); }  
);
```

Este tipo de funciones se denominan **funciones anónimas**.

En ES6 hay otra forma más sencilla de crear expresiones de función, y es mediante las **funciones flecha**.

Su sintaxis es la siguiente:

```
let func = ( arg1, arg2 ) => expresion;
```

La función recoge los parámetros indicados y devuelve el resultado de la expresión.

Veamos un ejemplo:

```
let sum = ( a, b ) => a+b;  
console.log( sum(5, 6) );           // 11
```

Si solo hay un argumento se pueden obviar los paréntesis.

```
let double = a => a*2;  
console.log( sum(5) );              // 10
```

El uso de funciones flecha simplifica bastante la creación de funciones anónimas

```
function ask( question, yes, no ) {  
    if ( confirm(question) ) yes()  
    else no();  
}  
  
ask(  
    "Estás de acuerdo?",  
    () => alert("Estás de acuerdo"),  
    () => alert("No estás de acuerdo")  
);
```


Se pueden crear funciones flecha multilínea rodeando el código entre llaves. En ese caso se debe indicar explícitamente al sentencia return.

```
let sum = ( a, b ) => {  
  let result = a+b;  
  return result;  
}  
  
alert( sum(2, 3) );
```