# Tema 2. Manejo de la sintaxis del lenguaje

## Contenido

## Sintaxis básica

- Javascript es sensible a mayúsculas
- Los nombres de variables y funciones pueden contener letras, números y símbolos como $ y _, pero no pueden empezar por número. Tampoco pueden usarse las palabras reservadas del lenguaje.
- Es recomendable usar nombres semánticos (significativos)
- Nomenclatura: mejor *camelCase*, ya que es el que el propio Javascript utiliza, que *snake_case* o *kebab-case* (aunque se suele usar PascalCase para clases y CONSTANT_CASE para constantes).
- // Pueden incluirse comentarios de una línea
- /* Pero
  también
  multilínea */
- Finalizar sentencias siempre con punto y coma como buena práctica
- Indentar el código (usar 2 o 4 espacio por cada nivel)
- Delimitar bloques de código com llaves {}
- Las cadenas de texto se delimitan con comillas dobles o simples.

# Variables y constantes

En los lenguajes de programación es fundamental la existencia de variables que permiten almacenar un valor para su uso en un ámbito determinado del programa. En este sentido JavaScript es un lenguaje débilmente tipado porque no requiere la declaración expresa del tipo de dato de la variable y permite asimismo que el tipo pueda variar durante la ejecución.

Las variables tienen dos acciones fundamentales: **declaración** y **asignación**. La declaración permite reservar un espacio en memoria, aunque aún no hayamos dado un valor a la variable, que podremos asignar después. En JavaScript no es necesario declarar una variable para poder usarla, es decir, podemos asignar directamente valor a una variable sin declararla previamente.

En el uso de las variables es también fundamental tener en cuenta su ámbito. En este sentido debimos distinguir entre **variables globales** y **locales.** Las variables globales tienen como ámbito el objeto window del navegador y por tanto su existencia coincide con el tiempo de ejecución del programa, mientras que las variables locales tienen un ámbito más reducido, por ejemplo, el de ejecución de una función. Podríamos distinguir por tanto inicialmente como variables globales a aquellas declaradas y/o asignadas fuera de cualquier función u otro bloque de código delimitado por llaves y las variables locales como aquellas declaradas en el interior de bloques delimitados por llaves.

Para declarar variables se ha usado tradicionalmente la palabra reservada VAR, pero desde ECMAScript 6 se prefiere el uso de la palabra reservada LET. Podemos seguir usando var ya que por compatibilidad hacia atrás no se ha desechado su uso y, de hecho, si copiamos código de manuales o de internet veremos que los ejemplos funcionan, pero hay diferencias entre var y let que debemos tener en cuenta., Por ejemplo, al declarar variables dentro de bloques delimitados por llaves, la variable declarada con var tiene como ámbito no el bloque delimitado por llaves, sino la función en la que se encuentra, mientras que let sí tiene como ámbito su propio bloque de llaves. Esto hace que por ejemplo un bucle for declarado con var nos permita seguir accediendo al valor de la variable una vez acabado el bucle, cosa que no sucede si declaramos con let:

| | |
|---|---|
| for (var i=1;i<10;i++){<br><br>console.log(i)<br><br>}<br><br>console.log(i) //Nos devolerá 10 | for (let a=1;a<10;a++){<br><br>console.log(a)<br><br>}<br><br>console.log(a) //Nos devolverá error |

Esto hace que se prefiera el uso de let para delimitar el carácter local de la variable, y que var no se recomiende usar. Igualmente, en el uso para declarar variables globales hay también alguna diferencia pues la variable creada con var pertenece al objeto global Windows y la declarada con let, no).

Por tanto, una variable se considera global si es declarada o asignada fuera de un bloque de llaves o si estando dentro de un bloque delimitado por llaves no está declarada, así por ejemplo:

```
<script type="text/javascript">
let a= 1; //variable global por ser declarada fuera de cualquier ámbito de bloque
b=2 // variable global por tener asignado un valor sin declarar fuera de bloques
function ejemplo(){
let c=3; //variable local por estar declarada dentro un bloque de código
d=4; //variable global a pesar de tener asignado el valor en un bloque de código al no estar declarada.
}
</script>
```

Si ejecutamos este código, veremos que una vez ejecutado seguimos teniendo acceso a las variables *a* y *b* por ser globales, por lo que podríamos recuperar su valor en siguientes líneas de código si fuera necesario en nuestro programa. Igualmente podríamos acceder al valor de la variable *d*, pero en cambio no podríamos acceder a la variable *c* una vez ejecutada la función, pues su ámbito está limitado a la ejecución de esta.

El nombre de las variables debe comenzar por letra o guion bajo o $. No pueden, por tanto, empezar por número ni por otros caracteres y tampoco pueden usarse como nombre de variables palabras reservadas del propio lenguaje Javascript.

Si declaramos en un bloque una variable local que tiene el mismo nombre que una variable global, durante la ejecución del bloque se usará el valor de la variable local, como si la global no existiera, pero una vez finalizada la ejecución del bloque volverá a existir la variable global con su correspondiente valor.
**Los parámetros de una función son tratados como variables locales**

| BLOQUE | DECLARADA | TIPO |
|---|---|---|
| Fuera | Sí | Global |
| Fuera | No | Global |
| Dentro | Sí | Local |
| Dentro | No | Global |

**RECUERDA**

*«Solo son locales las variables declaradas dentro de un bloque de código y los parámetros de la función»*

## Hoisting de variables

Las variables se asignan en memoria durante la fase de compilación, lo que a efectos de ejecución es como si estuvieran al principio siempre de su bloque, por eso se denomina a este efecto hosting, porque es como si se «subieran» al inicio, aunque estén en otro lugar del bloque. Lo que sucede por tanto es que cualquier variable declarada en el bloque se declara de facto al principio. Por ello podemos usar una variable antes de declararla:

```
function ejemplo(){
alert("Primera "+miVariable); //miVariable no ha sido aún declarada en el código, pero por hoisting
sí lo estará aunque con valor undefined
var miVariable =1;
alert("Segunda: "+miVariable); //miVariable ya tiene valor asignado y por tanto mostrará 1
}
```

La declaración se produce al principio por *hoisting* pero la asignación del valor 1 se hace en el lugar correspondiente del código.
En el caso de let sucede lo mismo, pero no se le asigna el valor *undefined*, y por tanto, aunque se suba al principio la declaración, dará error si se quiere usar.

## CONSTANTES

Además de las variables que están pensadas precisamente para que su valor pueda variar durante la ejecución del programa tenemos también la posibilidad de declarar **constantes**, lo que nos permite guardar un valor de forma invariable para la ejecución de nuestro programa, para ello utilizamos la palabra reservada *const*, lo que indica al programa que el valor asignado no puede variarse, por ejemplo:

```
const PESO=15;
```

En el caso de las constantes no podemos declararlas sin asignarles un valor y hacerlo después, pues al ser de valor constante, al declararlas se le asigna ya un valor, por lo que en caso de no hacerlo tendrá el valor *undefined*. Por otra parte, es una convención usar letras mayúsculas para el nombre de las constantes.

Si la constante apunta a un objeto este sí puede cambiar, por ejemplo, un array puede variar, dado que la constante apunta al mismo objeto (eso no cambia), así, por ejemplo:

```
const MiArray=[1,2,3];

MiArray.push(4);
```

## Tipos de datos

Javascript es un lenguaje débilmente tipado pues al contrario que otros lenguajes no declara expresamente el tipo de las variables y además permite que este cambie durante la ejecución del programa (conversión o coerción de tipo de variables), es decir, una variable declarada inicialmente con un valor numérico puede más tarde albergar un valor de cadena de texto, es decir, en Javascript esto es correcto:

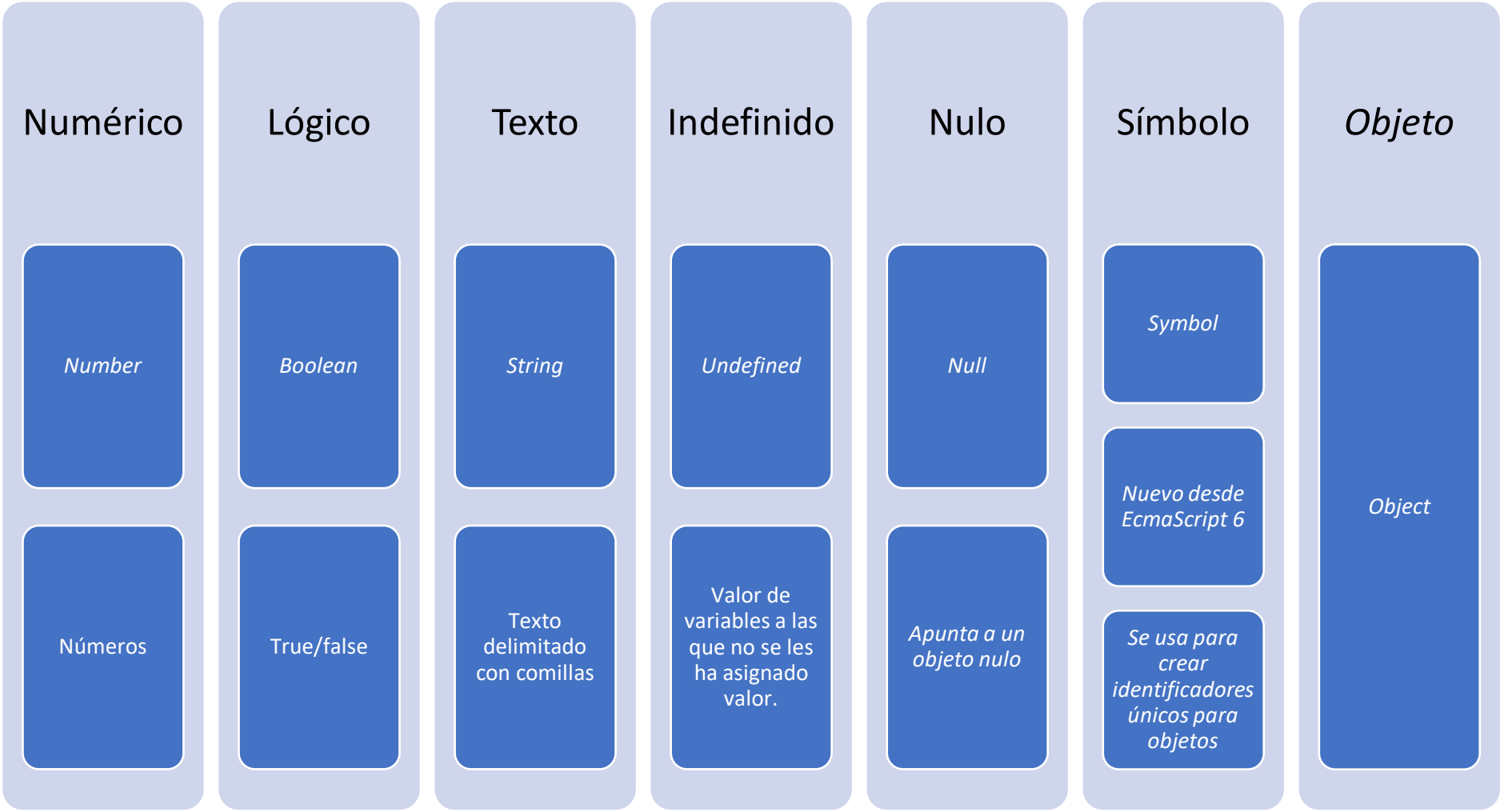
```
let a=1;

a="Hola mundo";
```

Pero también se producen en algunos casos conversiones explícitas debido a operaciones, por ejemplo:

```
let a=”2”
a++;
```

**En Javascript existen diferentes tipos de datos:**



### Number

- Son variables con un valor numérico. Por tanto tras el igual encontraremos un número.

### String

- Son variables que acogen un texto como valor. Por tanto tras el igual encontraremos siempre un contenido entre comillas (que pueden ser dobles o simples). Es el uso de las comillas y no el contenido de las mismas lo que delimita que la variable sea de texto, es decir, un número entre comillas es una cadena de texto.

### Boolean

- Estas son variables de tipo lógico, es decir, solo tienen dos valores posibles: verdadero (*true*) o falso (*false*).

### Undefined

- Este tipo de variable corresponde con aquellas a las que no se ha asignado todavía ningun valor. Si declaramos una variable y preguntamos por su tipo, al no haberle asignado aún ningún valor se considera de tipo indefinida, pues carece aún de otro tipo.

### Null

- En este caso estamos ante variables que apuntan a objetos. En el caso de *null* se trata de un objeto nulo. Es decir, no es que no tenga valor (que sería el caso de *undefined*), sino que o no existe el objeto o bien se le ha asignado explícitamente el valor nulo.

### Symbol

- Como en el caso de *null*, las variables de tipo *symbol* apuntan a un objeto y es un tipo de dato nuevo a partir de EcmaScript 6. Se utiliza para crear identificadores únicos para un objeto.

### Object

- Es directamente el tipo de dato objeto. Apunta igualmente como *null* y *symbol* a un objeto.

Para saber el tipo de una variable *number*, *boolean*, *string*, y *undefined* podemos usar el operador *typeof,* que nos devolverá el tipo de la variable:

**typeof nombreVariable;**

Pero tanto *null* como *Symbol* y *Object* devolverán el mismo tipo (Object), pues los tres apuntan en realidad a un objeto. Para saber el tipo en estos casos necesitaremos hacer una comprobación explícita.

# Operadores

**Operadores aritméticos.** Como hemos visto en la función sumar, podemos usar operadores aritméticos para realizar operaciones con números que devuelven un nuevo resultado numérico, así podemos hacer las operaciones matemáticas de suma (a+b), resta (a-b), multiplicación (a*b), división (a/b), resto (a%b), exponente(a**b).

Los operadores aritméticos pueden ser binarios, cuando se utilizan con dos números, como en los ejemplos indicados, pero algunos también tienen un uso unitario, en cuyo caso su funcionamiento es distinto, por ejemplo el operador + puede emplearse de forma unitaria para forzar convertir a número: +"1", o para incrementar el valor de una variable, por ejemplo a++ incrementara en 1 el valor de *a* (pero devolverá primero el valor de *a* antes del incremento, mientras que ++a hará el incremento devolviendo ya el valor con el incremento). El operador - puede usarse igualmente de forma unitaria para decrementar el valor de una variable (--a o a--)

> El operador + tiene también un valor de concatenación cuando se emplea con cadenas de texto, por ejemplo: "Hola "+ "mundo" dará como resultado una nueva cadena "Hola mundo".

Debemos tener cuidado al usar operadores aritméticos con tipos distintos, pues al ser JavaScript un lenguaje débilmente tipado en los que las variables pueden cambiar de tipo durante la ejecución, puede producirse una conversión implícita de tipos en operaciones aritméticas, pues para evitar un error el valor predomina sobre el tipo. Así, al intentar multiplicar una cadena o restarla, dado que esa operación no es posible con cadenas, JavaScript las convierte a número. Y lo mismo sucede con variables booleanas que, dado que en valor son 0 y 1, cuando operamos aritméticamente con ellas se pueden convertir implícitamente en números: 1+true devolverá 2.

En el caso de operación con cadena y números, la  cadena tendrá preferencia así, por ejemplo "1"+3 no devolverá 4, sino "13". Pero si la operación no es posible, se hará una conversión de tipos, por ejemplo “4”-1 devolverá 3. Igualmente, como hemos visto, podemos convertir previamente la cadena a número usando el operador unitario +, pues dado que este solo opera con números, para que funcione Javascript realiza la conversión de tipos, de forma que +"1"+3 sí devolverá 4.

**Operadores relacionales.** Estos operadores nos sirven para consultar la relación entre dos elementos y devuelven por tanto un valor booleano; verdadero si cumplen la relación expresada o falso si no se cumple.

| | |
|---|---|
| **Mayor que** | **>** |
| **Menor que** | **<** |
| **Mayor o igual que** | **>=** |
| **Menor o igual que** | **<=** |
| **Igual** | **==** |
| **Distinto** | **!=** |

**1<2 devolverá *true*, pero 1>2 devolverá *false***

La **igualdad** en Javascript expresada con == compara los valores, pero no los tipos, por lo que debemos tener en cuenta que javascript puede forzar la igualdad, por ejemplo 1=="1" devolverá *true*, pues los valores son iguales, aunque no los tipos. Por esa razón existe la fórmula de **igualdad estricta** que usa tres signos en vez de dos. Así 1==="1" sí devolverá *false*, pues comprueba tanto el valor (que es el mismo) como el tipo (que es distinto).

**Operadores lógicos.** Los operadores lógicos se utilizan para evaluar una expresión y devuelven por tanto valores booleanos.

| | | | |
|---|---|---|---|
| AND | Devuelve *true* si ambas expresiones son verdaderas, de lo contrario devuelve *false* | a&&b | (1<2)&&(2<3) devuelve *true* pues ambas son verdaderas.<br><br>(1>2)&&(2<3) devuelve *false* porque 1>2 es false. |
| OR | Devuelve *true* si al menos una de las expresiones es verdadera, de lo contrario devuelve *false* | a\|\|b | (1>2)\|\|(2<3) devuelve true pues al menos una de las dos es verdadera |
| NOT | Devuelve *true* si la expresión es falsa, porque se evalúa la negación de la expresión. | !a | !(1>2) devuelve true, pues (1>2 es falso y por tanto su negación es verdadera. |

Dado que con los operadores lógicos podemos agrupar varias expresiones es importante el uso de paréntesis. El paréntesis indica el orden de agrupamiento de las expresiones, es decir, siempre se evalúa primero lo contenido en los paréntesis y después lo que está fuera de los paréntesis, y siempre con orden de preferencia del interior al exterior, así la expresión:

!(1>2) es verdadera, porque se evalúa antes la expresión 1>2, que es falsa, y después, con el operador NOT se invierte su valor, por lo que el resultado del conjunto de la expresión es verdadero.

Sin embargo, sin paréntesis:

!1>2 es falsa porque el operador NOT solo afecta al número 1 y la negación de un número devuelve FALSE y asimismo la expresión FALSE>2 también es falsa.

## TABLAS DE VERDAD

| TÉRMINO1 | TÉRMINO2 | OPERADOR | RESULTADO |
|---|---|---|---|
| **V** | V | AND | V |
| **V** | F | AND | F |
| **F** | V | AND | F |
| **F** | F | AND | F |
| **V** | V | OR | V |
| **V** | F | OR | V |
| **F** | V | OR | V |
| **F** | F | OR | F |

En el caso del operador NOT, que es unitario, la tabla de verdad sería esta:

| TÉRMINO | OPERADOR | RESULTADO |
|---|---|---|
| **V** | NOT | F |
| **F** | NOT | V |

**Operadores de asignación.** Son los operadores usados para asignar valores, por tanto, emplean el operador = pero este puede combinarse con otros para añadir un incremento, decremento, producto o división antes de la asignación, así:

| | | |
|---|---|---|
| a=1 | | 1 |
| a+=1 | Equivale a: a=a+1 | 2 |
| a-=1 | Equivale a: a=a-1 | 0 |
| a*=2 | Equivale a: a=a*2 | 2 |
| a/=2 | Equivale a: a=a/2 | 0.5 |

# Funciones

Las funciones son objetos creados por el programador para subdividir el programa en diferentes procedimientos que se pueden invocar a lo largo de la ejecución. Las funciones permiten almacenar valores y realizar diferentes tareas.

La forma clásica de definir una función en JavaScript es con la palabra reservada *function* seguida del nombre asignado a la función y paréntesis. Las tareas y asignaciones de la función se encontrarán a continuación entre llaves:

```
function miFuncion(){
//tareas y asignaciones
}
```

Los paréntesis pueden estar vacíos o incluir nombres de variables u otros objetos que la función recibirá al ser invocada; son los llamados **parámetros de la función**. Por ejemplo, una función que desarrolla la tarea de sumar dos números recibirá como parámetros los números que se desea sumar:

```
function sumar(num1,num2){
let resultado=num1+num2;
return resultado;
}
```

Para invocar a una función usaremos su nombre (ya sin la palabra reservada *function*), por ejemplo:

```
sumar(2,4)
```

Las funciones pueden usar la palabra reservada *return* para devolver un valor, pero también pueden realizar una serie de tareas sin devolver ningún resultado, en cuyo caso, el resultado de la función será indefinido.

Por ejemplo, si hacemos un *typeof* de sumar(3,5) nos indicará que es un número, puesto que la función devuelve mediante *return* un valor numérico. Sin embargo, si eliminamos el *return*:

```
function sumar(num1,num2){

let resultado=num1+num2

}
```

Al hacer typeof sumar(2,4) el resultado será undefined pues la función no devuelve ningún resultado.

Podemos asignar lo que devuelve la función a una variable

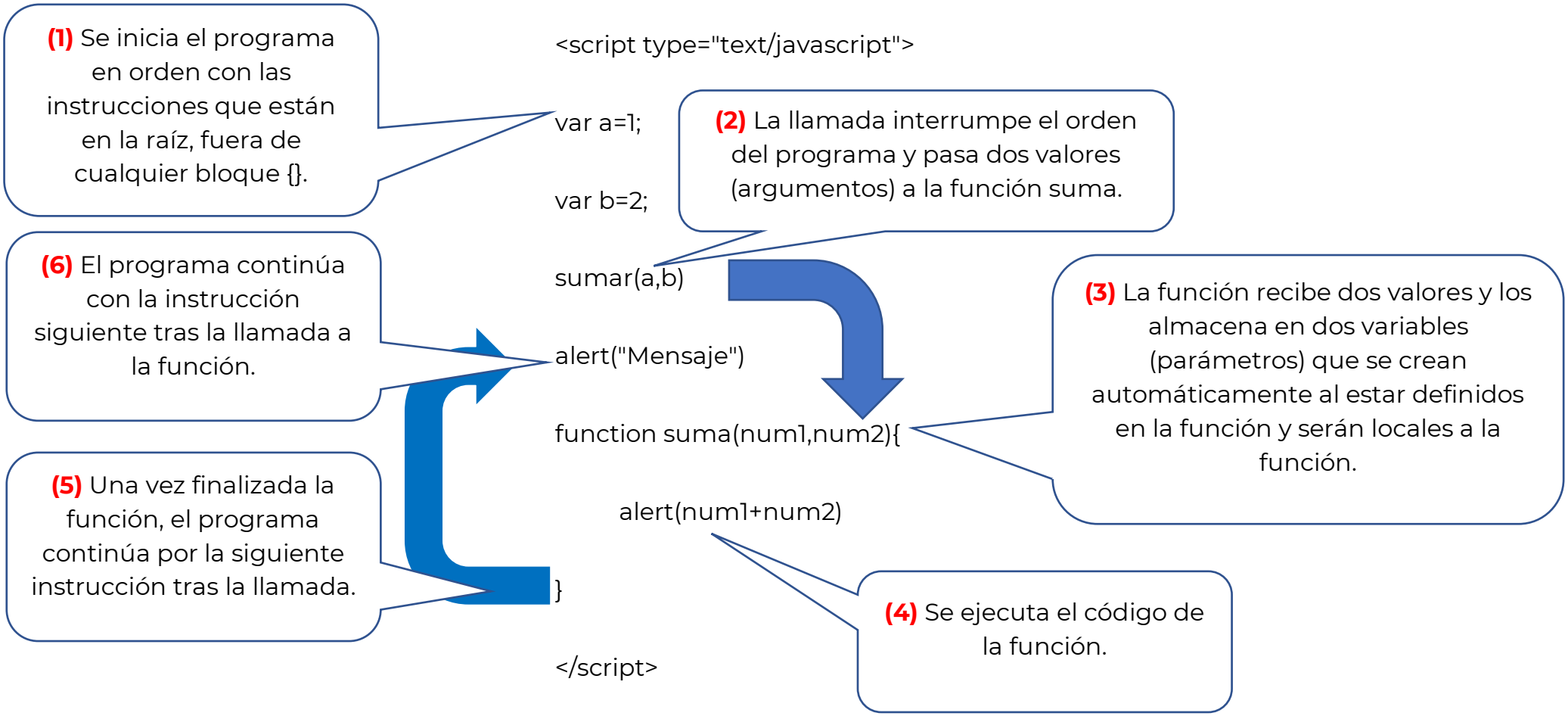
```
let miVariable = sumar(5,6)
```

Es posible anidar funciones:

```
function a(){
        alert("estoy en a")
        function b(){
                alert("estoy en b")
         }
        return b; //el resultado de a es devolver la función b, lo que permite que se pueda acceder a ella mediante a()()
        }

a()(); //Ejecutará ambas
a(); Solo ejecutara a y no b.
```

## PROGRAMA


(1) Se inicia el programa en orden con las instrucciones que están en la raíz, fuera de cualquier bloque {}.
<script type="text/javascript">
var a=1;
var b=2;
(2) La llamada interrumpe el orden del programa y pasa dos valores (argumentos) a la función suma.
sumar(a,b)
(6) El programa continúa con la instrucción siguiente tras la llamada a la función.
alert("Mensaje")
(3) La función recibe dos valores y los almacena en dos variables (parámetros) que se crean automáticamente al estar definidos en la función y serán locales a la función.
function suma(num1,num2){
alert(num1+num2)
(5) Una vez finalizada la función, el programa continúa por la siguiente instrucción tras la llamada.
}
(4) Se ejecuta el código de la función.
</script>

Podemos asignar un valor por defecto a los parámetros en el caso de que no se reciban correctamente, por ejemplo:

```
function sumar(num1=1,num2=2) {

let resultado=num1+num2;

return resultado;

}
```

En este caso, si al invocar la función no incluimos los parámetros, estos tomaran por defecto el valor indicado como alternativa.

## Funciones anónimas

Las funciones tienen un nombre que permite invocarlas cuando sea necesario ejecutar su código, pero en ocasiones necesitamos simplemente ejecutar una función una única vez, en ese caso podemos usar una función anónima, que, no tiene, por tanto, nombre.

Las funciones anónimas suelen usarse para asignar las acciones que contienen la función anónima a otro elemento, en este sentido, podemos usar la función anónima de esta forma:

```
unNombre= function(){alert("mensaje"}
```

Lo que equivale a crear en realidad una función llamada unNombre, que podemos ejecutar en la forma

```
unNombre()
```

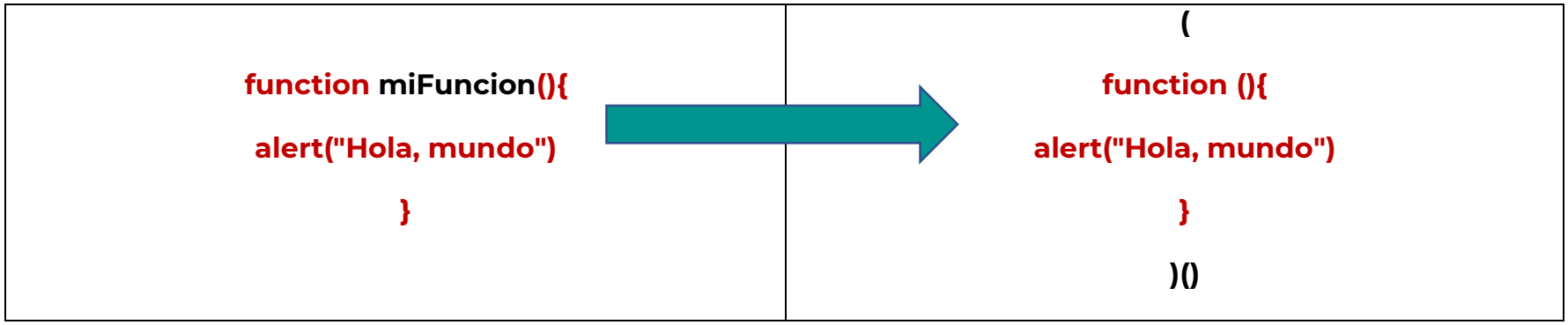
Pero también podemos ejecutar directamente una función, sin llamarla usando para ello una función anónima directamente. En este caso, debemos tener cuidado con la sintaxis por el uso de llaves y paréntesis para englobar a la función anónima:

```
( function(){

return 2%2}

) ()
```

El paso de una función normal a una anónima sería por tanto quitarle el nombre e introducir toda la función dentro de paréntesis y a continuación añadir al final otros paréntesis (lo que cambia es lo marcado en negro en cada caso):




Los segundos paréntesis son los equivalentes a los de la llamada a una función. Dado que la función anónima se llama a sí misma, en estos segundos paréntesis podemos incluir argumentos que recibe la propia función como parámetros:

```
(function (texto){
alert(texto)
}
)("Hola, mundo")
```

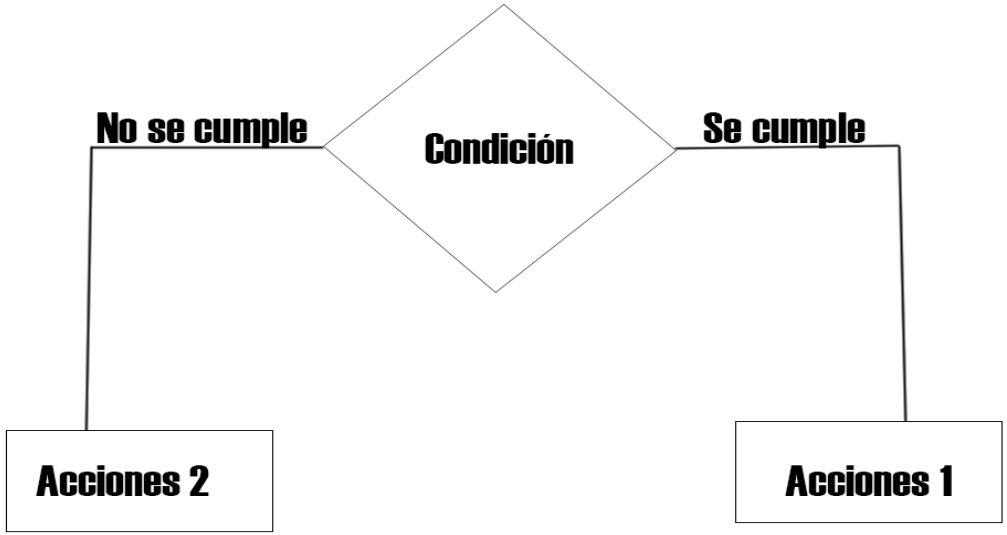
Otras formas de función que veremos más adelante son la función flecha:

```
suma=(num1,num2)=>console.log(num1+num2)
```

# Estructuras de control. Condicionales

En Javascript utilizamos expresiones condicionales para evaluar una condición y ejecutar unas tareas u otras en función del resultado.

Cuando plasmamos gráficamente un programa mediante un diagrama de flujo, las estructuras de control se representan con un rombo, así, una condición sencilla tendría esta forma:




Las diferentes estructuras condicionales en Javascript, pueden ser selectivas o iterativas.

**IF.** Es la forma clásica de condicional en JavaScript. Plantea una condición y solo si se cumple se ejecutan las acciones indicadas:

***If (edad<18) window.location="https://www.google.com";***

**IF / ELSE.** El condicional puede tener también una forma doble en la que se evalúa la condición y se plantean dos bloques de acciones, el primero se ejecuta si se cumple la condición, y el segundo si no se cumple:

```
if (a>1){
console.log("Se cumple")
}
else{
console.log("No se cumple")
}
```

Esta estructura selectiva doble puede también utilizarse mediante el operador ternario:

**Condicional ternario.** Una forma abreviada de condicional puede usarse mediante una forma ternaria:

**condicion?verdadero:falso**

Es decir, se expresa una condición y a continuación las acciones a realizar si es verdadera seguida de las acciones a realizar si es falsa, por ejemplo, el condicional anterior podría expresarse así:

**a>1?console.log("Se cumple"):console.log("No se cumple")**

Pueden agruparse diferentes acciones usando paréntesis y separando las sentencias con comas, por ejemplo:

**a<1?(a=1,console.log(a)):(a=0,console.log(a))**

E igualmente pueden anidarse condiciones, por ejemplo:

```
a >= 5
  ? a > 7
    ? alert("Sobresaliente")
    : a >= 6
    ? alert("Notable")
    : alert("Aprobado")
  : a < 4
  ? alert("Ya te vale")
  : alert("Suspenso, pero has estado cerca");
```

**IF / ELSE IF.** Cuando queremos comprobar no solo si se cumple o no una condición, sino diferentes alternativas, podemos añadir else if para expresar nuevas condiciones, es decir, si no se cumple la condición expresada con if pasaría a comprobar las deferentes condiciones expresadas en los bloques else if, e igualmente puede añadirse un bloque else para cuando no se cumpla ninguna de las condiciones expresadas:

```
if (a==1){
//acciones
}
else if(a==2){
//acciones
}
else if(a==3){
//acciones
}
else{
//acciones si no se cumple ninguna condición
}
```

**SWITCH / CASE.** Otra forma de condicional es la estructura de control múltiple switch / case que permite evaluar una expresión y realizar acciones en función del resultado o valor de esa expresión.

```
switch (variable o expresión) {
    case 1:
    //acciones
    break
    case 2:
    //acciones
    break
    default:
    //acciones
    //No es necesario break porque es el último bloque
}
```

```
switch (1<2){
    case true:
    console.log("Cierto")
     break
    case false:
    console.log("Falso")
    break
    default:
    console.log("valor por defecto")
    }
```

En esta estructura usamos *break* para interrumpir la comprobación de las condiciones en cualquier momento y también podemos incluir opcionalmente acciones a ejecutar cuando no se cumple ninguna condición, para lo cual usamos *default*.

Es posible agrupar varios valores en un mismo caso:

```
switch (a){
     case 1:
     case 3:
     case 5:
     case 7:
     case 9:
     alert("el resultado es impar");
     break;
     case 2:
     case 4:
     case 6:
     case 8:
     case 10:
     alert("el resultado es par");
     break;
}
```

# Estructuras iterativas. Bucles

Cuando es necesario repetir una tarea varias veces, podemos emplear estructuras iterativas para generar bucles:

**FOR**. Los bucles FOR se utilizan para repetir una serie de acciones un número determinado de veces, para lo cual se usa un valor inicial, una condición que mantendrá el bucle mientras se cumpla y un incremento del valor inicial, lo que permite que el ciclo se repita el número de veces deseado:

```
for (let i=1;i<20;i++){
//acciones
}
for (i=20;i!=71%8;i--){
 console.log(i)
}
```

Podemos forzar la interrupción del bucle con *break*, mientras que con *continue* podemos saltar al paso siguiente

| |
|---|
| La estructura **for** tiene también otras formas para iterar en objetos como **for in** y **for of** así como **for each** que veremos más adelante al tratar arrays y objetos. |

**WHILE**. Los bucles WHILE evalúan una condición que, si se cumple, inicia el ciclo, que se repetirá mientras la condición se siga cumpliendo:

```
while (a<5){
//acciones
}
```

**DO / WHILE.** Este caso es similar con la diferencia de que las acciones se ejecutan una primera vez antes de evaluar la condición, por lo que siempre se ejecutan al menos una vez (aunque la condición no se cumpla), y después volverán a repetirse mientras la condición se cumpla.

```
do{
//acciones
}
while(condición)
```

```
let a = 5;
do {
  a++
  console.log(a);
}
while (a < 5);
```

En este caso, la condición no se cumple, pero al menos una vez se ejecutan las acciones del bloque *do*.

Por el contrario, en un while o en un bucle for, con la misma condición no se imprimiría nunca el valor de a:

```
for (let a=5;a<5;a++){
console.log(a)
}
```

```
while (a<5){
console.log(a)
}
```

En un bucle **while** podemos introducir un ***break*** en cualquier momento (mediante un condicional) para salir del bucle.

```
let a = 0;
do {
    a++;
    if (a==3) break;
    console.log(a);
    }
while (a < 5);
```

O bien saltar un paso usando **continue**:

```
let a = 0;
do {
    a++;
    if (a==3) continue;
    console.log(a);
    }
while (a < 5);
```

También es posible interrumpir un bucle mediante una etiqueta, para ello identificamos el bucle con una etiqueta, de forma que después usamos la instrucción break seguida de la etiqueta que identifica el bucle que queremos interrumpir.

| La forma de etiquetar un bucle es **etiqueta:while(condicion)** |
|---|

```
var a=1;
buclePrincipal:while(a<6){
        let b=0;
        bucleSecundario: while (b<10){
                (b>11)?b=1:b++;
                console.log("bucle 1 con valor  "+a +" y bucle 2 con valor
"+b)
                if (b==5) break bucleSecundario
                 if(a==3) break buclePrincipal
        }
        a++
}
```

# Cuadros de diálogo

Para dialogar con el usuario, Javascript cuenta con tres cuadros de diálogo

## Alert

Ofrece una ventana con un mensaje y el usuario debe pulsar aceptar para que continúe la ejecución del programa

```
alert(“Bienvenido a mi web”);
```

## Confirm

Ofrece una ventana con un mensaje que el usuario puede aceptar o cancelar, y por tanto el valor que devuelve será *true* si el usuario acepta y *false* si cancela, que puede almacenarse en una variable:

```
let edad=confirm(“¿Eres mayor de edad?”)
```

## Prompt

Ofrece una ventana con un mensaje y una caja en la que el usuario puede escribir libremente, por lo que el valor que devuelve es una cadena con el texto introducido por el usuario.

```
let nombre=prompt(“¿Cómo te llamas?”);
```

Un ejemplo de combinación de los tres cuadros de diálogo:

```
confirm("Eres mayor de edad")==true? alert("Bienvenido, "+prompt("¿Cómo te llamas?"))
:alert("Adiós");
```