UD2
Introducción al lenguaje JavaScript

1. Sintaxis Básica

Antes de comenzar a desarrollar programas y utilidades con JavaScript, es necesario conocer los **elementos básicos** con los que se construyen las aplicaciones. En esta unidad se explica en detalle y comenzando desde cero los conocimientos básicos necesarios para poder comprender la sintaxis básica de Javascript.

1.1. Espacios en blanco

No se tienen en cuenta los espacios en blanco y las nuevas líneas. El intérprete de JavaScript ignora cualquier espacio en blanco sobrante, por lo que el código se puede ordenar de forma adecuada para entenderlo mejor (tabulando las líneas, añadiendo espacios, creando nuevas líneas, etc.)

```
var that = this;
```

→ Aquí el espacio en blanco entre var y that no puede ser eliminado, pero el resto sí.

1.2. Comentarios

JavaScript ofrece dos tipos de comentarios, **de bloque** gracias a los caracteres **/* */** y **de línea** comenzando con **//**. El formato **/* */** de comentarios puede causar problemas en ciertas condiciones, como en las expresiones regulares, por lo que hay que tener cuidado al utilizarlo.

Por ejemplo:

```
/*
    var rm_a = /a*/.match(s);
*/
```

provoca un error de sintaxis. Por lo tanto, suele ser recomendable utilizar únicamente los comentarios de línea, para evitar este tipo de problemas.

1.3. Variables

Las variables en JavaScript se crean mediante la palabra reservada **var**. De esta forma, podemos declarar variables de la siguiente manera:

```
var numero_1 = 3;
var numero_2 = 1;
var resultado = numero_1 + numero_2;
```

La palabra reservada **var** solamente se debe indicar al declarar por primera vez la variable.

Inicializar: En JavaScript **no es obligatorio inicializar las variables**, ya que se pueden declarar por una parte y asignarles un valor posteriormente. Por tanto, el ejemplo anterior se puede rehacer de la siguiente manera:

```
var numero_1;
var numero_2;
numero_1 = 3;
numero_2 = 1;
 var resultado = numero_1 + numero_2;
```

Declarar: Una de las características más sorprendentes de JavaScript para los programadores habituados a otros lenguajes de programación es que **tampoco es necesario declarar las variables**. En otras palabras, se pueden utilizar variables que no se han definido anteriormente mediante la palabra reservada var. El ejemplo anterior también es correcto en JavaScript de la siguiente forma:

```
var numero_1 = 3;
var numero_2 = 1;
resultado = numero_1 + numero_2;
```

La variable resultado no está declarada, por lo que JavaScript crea una variable global (más adelante se verán las diferencias entre variables locales y globales) y le asigna el valor correspondiente. De la misma forma, también sería correcto el siguiente código:

```
numero_1 = 3;
numero_2 = 1;
resultado = numero_1 + numero_2;
```

→ **Nota:** En cualquier caso, se recomienda declarar todas las variables que se vayan a utilizar.

ECMASCRIPT 6 - LET y CONST

En la última revisión de EcmaScript se ha realizado una especial revisión sobre el **ámbito de una variable**, en inglés denominado **“scope”.** Vamos a ver con un ejemplo como funciona la declaración de variables con “var”:

```
        var a = 1;
        if (1 === a) {
            var b = 2;
        }

        for (var c=0; c < 3; c++)
        {

        }

        function letsDeclareAnotherOne() {
            var d=4;
```

```
}



console.log(a);
console.log(b);
console.log(c);
console.log(d);
```

Como se puede ver, la única estructura que identifica o implementa un “ámbito” propio es la función, es decir, la variable “d” está indefinida. En las demás, al inicializarse las variables con “var” estas serán globales.

LET

Se utiliza igual que “var” para la declaración de variables pero **con restricción de ámbito** (block scoped)

```
let aa = 1;

if (1 === aa) {
    let bb = 2;
}

for (let cc=0; cc < 3; cc++)
{

}

function letsDeclareAnotherOne() {
    let dd=4;
}


console.log(aa);
console.log(bb);
console.log(cc);
console.log(dd);
```

La única variable que está definida es “aa”.

CONST

Se utiliza para declarar variables “una sola vez”. Si se intenta reinicializar, javascript lanzará un error. Muy importante resaltar que al igual que “let” cuando declaramos una variable con “const” tenemos “restricción de ámbito”. Importante: ¡¡¡ Es obligatorio inicializar la constante !!!

```
const PI = 3.141593;
```

```
PI = 3.14   →   Lanzará un error ya que la variable ya estaba inicializada y declarada como constante.
```

1.3.1. Nombres de variables

El nombre de una variable también se conoce como **identificador** y debe cumplir las siguientes normas:

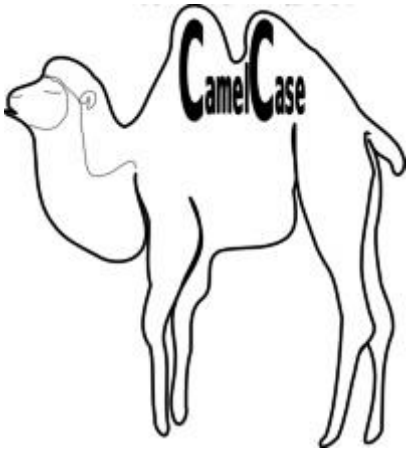
- **Sólo puede estar formado por letras, números y los símbolos $ (dólar) y _ (guión bajo).**
- **El primer carácter no puede ser un número.**

Por tanto, las siguientes variables tienen nombres correctos:

```
var $numero1;
var _$letra;
var $$$otroNumero;
var $_a__$4;
```

Sin embargo, las siguientes variables tienen identificadores incorrectos:

```
var 1numero;        // Empieza por un número
var numero;1_123;   // Contiene un carácter ";"
```



A continuación se indica el **listado de palabras reservadas en JavaScript**, y que no podremos utilizar para nombrar nuestras variables, parámetros, funciones, operadores o etiquetas:

- abstract
- boolean break byte
- case catch char class const continue
- **debugger** default delete do double
- else enum export extends
- false final finally float for function
- goto
- if implements import in instanceof int interface
- let long
- **native** new null
- package private protected public
- return
- short static super switch synchronized
- this throw throws **transient** true try typeof
- var volatile void
- while with

1.3.2. Tipos de variables

JavaScript divide los distintos tipos de variables en dos grupos: **tipos primitivos** y **tipos de referencia o clases**.

Tipos primitivos

JavaScript define cinco tipos primitivos: **undefined, null, boolean, number y string**. Además de estos tipos, JavaScript define el operador **typeof** para averiguar el tipo de una variable.

- **Variables de tipo undefined:** El tipo undefined corresponde a las variables que han sido definidas y todavía no se les ha asignado un valor:

```
var variable1;
console.log(typeof variable1);  // devuelve "undefined"
```

- **Variables de tipo null:** Se trata de un tipo similar a undefined, y de hecho en JavaScript se consideran iguales (undefined == null). El tipo null se suele utilizar para representar objetos que en ese momento no existen.

```
var nombreUsuario = null;
```

- **Variables de tipo boolean:** Se trata de una variable que sólo puede almacenar uno de los dos valores especiales definidos y que representan el valor *"verdadero"* y el valor *"falso"*.

```
var variable1 = true;
var variable2 = false;
```

Los valores **true** y **false** son valores especiales, de forma que no son palabras ni números ni ningún otro tipo de valor. Este tipo de variables son esenciales para crear cualquier aplicación, tal y como se verá más adelante.

Cuando es necesario convertir una variable numérica a una variable de tipo boolean, JavaScript aplica la siguiente conversión: **el número 0 se convierte en false y cualquier otro número distinto de 0 se convierte en true.**

- **Variables de tipo numérico:** **En JavaScript únicamente existe un tipo de número.** Internamente, es representado como un dato de **64 bits** en coma flotante. A diferencia de otros lenguajes de programación, **no existe una diferencia entre un número entero y otro decimal**, por lo que 1 y 1.0 son el mismo valor.

  Si el número es entero, se indica su valor directamente.

```
var variable1 = 10;
```

  Si el número es decimal, se debe utilizar el punto (.) para separar la parte entera de la decimal.

```
var variable2 = 3.14159265;
```

→ JavaScript define tres valores especiales muy útiles cuando se trabaja con números: Infinity, -Infinity, y NaN.

En primer lugar se definen los valores Infinity y –Infinity para representar números demasiado grandes (positivos y negativos) y con los que JavaScript no puede trabajar.

```
var variable1 = 3, variable2 = 0;
console.log(variable1/variable2);  // muestra "Infinity"
```

El otro valor especial definido por JavaScript es **NaN, que es el acrónimo de *"Not a Number"***. De esta forma, si se realizan operaciones matemáticas con variables no numéricas, el resultado será de tipo NaN.

Para manejar los valores **NaN**, se utiliza la función relacionada **isNaN()**, que devuelve true si el parámetro que se le pasa no es un número:

```
var variable1 = 3;
var variable2 = "hola";
isNaN(variable1);  // false
isNaN(variable2);  // true
isNaN(variable1 + variable2);  // true
```

- **Variables de tipo cadena de texto:** Las variables de tipo cadena de texto permiten almacenar cualquier sucesión de caracteres, por lo que se utilizan ampliamente en la mayoría de aplicaciones JavaScript. Cada carácter de la cadena se encuentra en una posición a la que se puede acceder individualmente, siendo el primer carácter el de la posición 0.

  El valor de las cadenas de texto se indica encerrado entre **comillas simples o dobles**:

```
var variable1 = "hola";
var variable2 = 'mundo';
var variable3 = "hola mundo, esta es una frase más larga";
```

  JavaScript define un mecanismo para incluir de forma sencilla **caracteres especiales** (ENTER, Tabulador) y problemáticos (comillas). Esta estrategia se denomina **"mecanismo de escape"**, ya que se sustituyen los caracteres problemáticos por otros caracteres seguros que siempre empiezan con la **barra** \:

Si se quiere incluir...	Se debe sustituir por...
Una nueva línea	`\n`
Un tabulador	`\t`
Una comilla simple	`\'`
Una comilla doble	`\"`
Una barra inclinada	`\\`

Algunos ejemplos utilizando el mecanismo de escape:

```
var variable = "hola mundo, esta es \n una frase más larga";
var variable3 = "hola 'mundo', esta es una \"frase\" más larga";
```

Tipos de referencia

Aunque **JavaScript no define el concepto de clase**, **los tipos de referencia se asemejan a las clases** de otros lenguajes de programación. Los objetos en JavaScript se crean mediante la palabra reservada **new y el nombre de la clase que se va a instanciar**. De esta forma, para crear un objeto de tipo String se indica lo siguiente (los paréntesis solamente son obligatorios cuando se utilizan argumentos, aunque se recomienda incluirlos incluso cuando no se utilicen):

```
var variable1 = new String("hola mundo");
```

JavaScript define una clase para cada uno de los tipos de datos primitivos. De esta forma, existen objetos de tipo **Boolean** para las variables booleanas, **Number** para las variables numéricas y **String** para las variables de cadenas de texto. Las clases Boolean, Number y String almacenan los mismos valores de los tipos de datos primitivos y añaden propiedades y métodos para manipular sus valores.

```
var longitud = "hola mundo".length;
```

La propiedad **length** sólo está disponible en la clase String, por lo que en principio no debería poder utilizarse en un dato primitivo de tipo cadena de texto. Sin embargo, JavaScript convierte el tipo de dato primitivo al tipo de referencia String, obtiene el valor de la propiedad length y devuelve el resultado. Este proceso se realiza de forma automática y transparente para el programador.

En realidad, con una variable de tipo String no se pueden hacer muchas más cosas que con su correspondiente tipo de dato primitivo. Por este motivo, no existen muchas diferencias prácticas entre utilizar el tipo de referencia o el tipo primitivo, **salvo en el caso del resultado del operador typeof y en el caso de la función eval(), como se verá más adelante**.

La principal diferencia entre los tipos de datos es que los datos primitivos se manipulan por valor y los tipos de referencia se manipulan, como su propio nombre indica, por referencia. En realidad, no solo se aplica a los tipos de referencia; **se aplica a todos los tipos "NO PRIMITIVOS" que son "Objetos, Arrays y Funciones"**. Los conceptos *"por valor"* y *"por referencia"* son iguales que en el resto de lenguajes de programación, aunque existen diferencias importantes (no existe por ejemplo el concepto de puntero).

Cuando un dato se manipula por valor, lo único que importa es el valor en sí. Cuando se asigna una variable por valor a otra variable, se copia directamente el valor de la primera variable en la segunda. Cualquier modificación que se realice en la segunda variable es independiente de la primera variable.

De la misma forma, cuando se pasa una variable por valor a una función (como se explicará más adelante) sólo se pasa una copia del valor. Así, cualquier modificación que realice la función sobre el valor pasado no se refleja en el valor de la variable original.

En el siguiente ejemplo, una variable se asigna por valor a otra variable:

```
var variable1 = 3;
var variable2 = variable1;
variable2 = variable2 + 5;
// Ahora variable2 = 8 y variable1 sigue valiendo 3
```

La variable1 se asigna por valor en la variable1. Aunque las dos variables almacenan en ese momento el mismo valor, son independientes y cualquier cambio en una de ellas no afecta a la otra. El motivo es que los tipos de datos primitivos siempre se asignan (y se pasan) por valor.

Sin embargo, en el siguiente ejemplo, se utilizan tipos de datos de referencia:

```
// variable1 = 25 diciembre de 2009
var variable1 = new Date(2009, 11, 25);
// variable2 = 25 diciembre de 2009
var variable2 = variable1;
// variable2 = 31 diciembre de 2010
variable2.setFullYear(2010, 11, 31);
// Ahora variable1 también es 31 diciembre de 2010
```

Más ejemplos:

- Asignando valores primitivos:

```
let a = 'hola';
let b = a; // asignamos valor de `a` a `b`.
a += '!'; // cambiamos valor de `a` añadiendo ! al final
console.log(a); // hola!
console.log(b); // hola
```

- Asignando valores por referencia:

```
const a = [1, 2, 3];
const b = a;
```

```
a.push(4);
console.log(a); // [ 1, 2, 3, 4 ]
console.log(b); // [ 1, 2, 3, 4 ]
```

¿Y en funciones? ¿Seguiremos teniendo el mismo comportamiento?

```
//Sin usar el operador arrow =>
function foo(str)
      {
            str = "otra cosa";
            return str;
      }

const aString = "a";
console.log(foo(aString));
console.log(aString);
```

```
//Usando el operador arrow =>
const foo = (str) => {
  str = 'otra cosa';
  return str;
};

const aString = 'a';
console.log(foo(aString)); //=> 'otra cosa'
console.log(aString); // => 'a'
```

Ahora vamos a probar el mismo caso con funciones pero usando un objeto por referencia como puede ser un array:

```
function foo (arr) {
  let results = arr;
  while (results.length != 0) {
      results.pop();
  }
  return results;
};

let arr = [1, 2, 3, 4];
foo(arr);
console.log(arr); // []
```

```
//Usando el operador arrow =>
```

2. Operadores

Los operadores permiten manipular el valor de las variables, realizar operaciones matemáticas con sus valores y comparar diferentes variables. De esta forma, los operadores permiten a los programas realizar cálculos complejos y tomar decisiones lógicas en función de comparaciones y otros tipos de condiciones.

2.1. Asignación

El operador de asignación es el más utilizado y el más sencillo. Este operador se utiliza para guardar un valor específico en una variable. El símbolo utilizado es = (no confundir con el operador == que se verá más adelante):

```
var numero1 = 3;
```

A la izquierda del operador, siempre debe indicarse el nombre de una variable.

2.2. Aritméticos

JavaScript permite realizar manipulaciones matemáticas sobre el valor de las variables numéricas. Los operadores definidos son: suma (+), resta (-), multiplicación (*), división (/) y módulo (%). Ejemplo:

```
var numero1 = 10;
var numero2 = 5;
resultado = numero1 / numero2;  // resultado = 2
resultado = 3 + numero1;        // resultado = 13
resultado = numero2 - 4;        // resultado = 1
resultado = numero1 * numero 2; // resultado = 50
resultado = numero1 % numero2;  // resultado = 0
```

Los operadores matemáticos también se pueden combinar con el operador de asignación para abreviar su notación:

```
var numero1 = 5;
numero1 += 3;   // numero1 = numero1 + 3 = 8
numero1 -= 1;   // numero1 = numero1 - 1 = 4
numero1 *= 2;   // numero1 = numero1 * 2 = 10
numero1 /= 5;   // numero1 = numero1 / 5 = 1
numero1 %= 4;   // numero1 = numero1 % 4 = 1
```

Existen dos operadores especiales que solamente son válidos para las variables numéricas y se utilizan para incrementar o decrementar en una unidad el valor de una variable (++ y --).

2.3. Lógicos

Los operadores lógicos son imprescindibles para realizar aplicaciones complejas, ya que se utilizan para tomar decisiones sobre las instrucciones que debería ejecutar el programa en función de ciertas condiciones. El resultado de cualquier operación que utilice operadores lógicos siempre es un valor lógico o booleano.

2.3.1. Negación

Uno de los operadores lógicos más utilizados es el de la negación. Se utiliza para obtener el valor contrario al valor de la variable:

```
var visible = true;
console.log(!visible);  // Muestra "false" y no "true"
```

2.3.2. AND

La operación lógica AND obtiene su resultado combinando dos valores booleanos. El operador se indica mediante el símbolo && y su resultado solamente es true si los dos operandos son true:

variable1	variable2	variable1 && variable2
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

2.3.3. OR

La operación lógica OR también combina dos valores booleanos. El operador se indica mediante el símbolo || y su resultado es true si alguno de los dos operandos es true:

variable1	variable2	variable1 || variable2
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`true`
`false`	`false`	`false`

2.4. Relacionales

Los operadores relacionales definidos por JavaScript son idénticos a los que definen las matemáticas: mayor que (>), menor que (<), mayor o igual (>=), menor o igual (<=), igual que (==) y distinto de (!=).

Los operadores relacionales también se pueden utilizar con variables de tipo cadena de texto:

```
var texto1 = "hola";
var texto2 = "hola";
var texto3 = "adios";
resultado = texto1 == texto3; // resultado = false
resultado = texto1 != texto2; // resultado = false
resultado = texto3 >= texto2; // resultado = false
```

Cuando se utilizan cadenas de texto, los operadores "mayor que" (>) y "menor que" (<) siguen un razonamiento no intuitivo: se compara letra a letra comenzando desde la izquierda hasta que se encuentre una diferencia

entre las dos cadenas de texto. Para determinar si una letra es mayor o menor que otra, las mayúsculas se consideran menores que las minúsculas y las primeras letras del alfabeto son menores que las últimas (a es menor que b, b es menor que c, A es menor que a, etc.)

2.5. typeof

El operador **typeof** se emplea para determinar el tipo de dato que almacena una variable. Su uso es muy sencillo, ya que sólo es necesario indicar el nombre de la variable cuyo tipo se quiere averiguar:

```
var myFunction = function() {
  console.log('hola');
};
var myObject = {
    foo : 'bar'
};
var myArray = [ 'a', 'b', 'c' ];
var myString = 'hola';
var myNumber = 3;

typeof myFunction;   // devuelve 'function'
typeof myObject;     // devuelve 'object'
typeof myArray;      // devuelve 'object' -- tenga cuidado
typeof myString;     // devuelve 'string'
typeof myNumber;     // devuelve 'number'
typeof null;         // devuelve 'object' -- tenga cuidado
```

Los posibles valores de retorno del operador son: undefined, boolean, number, string para cada uno de los tipos primitivos y object para los valores de referencia y también para los valores de tipo null.

2.6. instanceof

El operador **typeof** no es suficiente para trabajar con tipos de referencia, ya que devuelve el valor **object** para cualquier objeto independientemente de su tipo. Por este motivo, JavaScript define el operador **instanceof** para determinar la clase concreta de un objeto.

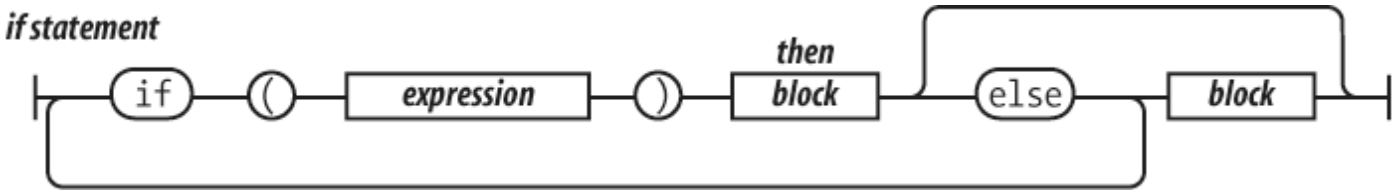
```
var variable1 = new String("hola mundo");
typeof variable1;              // devuelve "object"
variable1 instanceof String;  // devuelve true
```

El operador instanceof sólo devuelve como valor true o false. De esta forma, instanceof no devuelve directamente la clase de la que ha instanciado la variable, sino que se debe comprobar cada posible tipo de clase individualmente.

3. Estructuras de Control

3.1. Estructura if...else

La estructura más utilizada en JavaScript y en la mayoría de lenguajes de programación es la **estructura if**. Se emplea para tomar decisiones en función de una condición. Su definición formal es:




```
if(condicion) {
  ...
}
else {
  ...
}
```

Si la condición se cumple (es decir, si su valor es true) se ejecutan todas las instrucciones que se encuentran dentro del bloque {...}. Si la condición no se cumple (es decir, si su valor es false) no se ejecuta ninguna instrucción contenida en {...} y el programa continúa ejecutando el resto de instrucciones del script.

Los operadores AND y OR permiten encadenar varias condiciones simples para construir condiciones complejas:

```
var mostrado = false;
var usuarioPermiteMensajes = true;

if(!mostrado && usuarioPermiteMensajes) {
  console.log("Es la primera vez que se muestra el mensaje");
}
```

Para este segundo tipo de decisiones, existe una variante de la estructura if llamada if...else. Su definición formal es la siguiente:

Si la condición se cumple (es decir, si su valor es true) se ejecutan todas las instrucciones que se encuentran dentro del if(). Si la condición no se cumple (es decir, si su valor es false) se ejecutan todas las instrucciones contenidas en else { }. Ejemplo:

```
var edad = 18;

if(edad >= 18) {
  console.log("Eres mayor de edad");
```

```
} else {
  console.log("Todavía eres menor de edad");
}
```

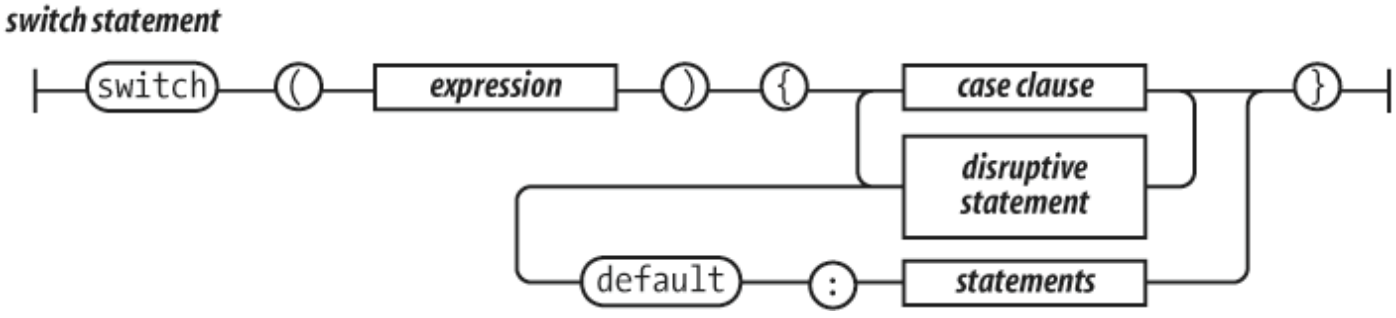
No es obligatorio que la combinación de estructuras if...else acabe con la instrucción else, ya que puede terminar con una instrucción de tipo else if().

3.2. Estructura switch

La estructura **switch** es muy útil cuando la condición que evaluamos puede tomar muchos valores. Si utilizáramos una sentencia if...else, tendríamos que repetir la condición para los distintos valores.

```
if(dia == 1) {
  console.log("Hoy es lunes.");
} else if(dia == 2) {
  console.log("Hoy es martes.");
} else if(dia == 3) {
  console.log("Hoy es miércoles.");
} else if(dia == 4) {
  console.log("Hoy es jueves.");
} else if(dia == 5) {
  console.log("Hoy es viernes.");
} else if(dia == 6) {
  console.log("Hoy es sábado.");
} else if(dia == 0) {
  console.log("Hoy es domingo.");
}
```

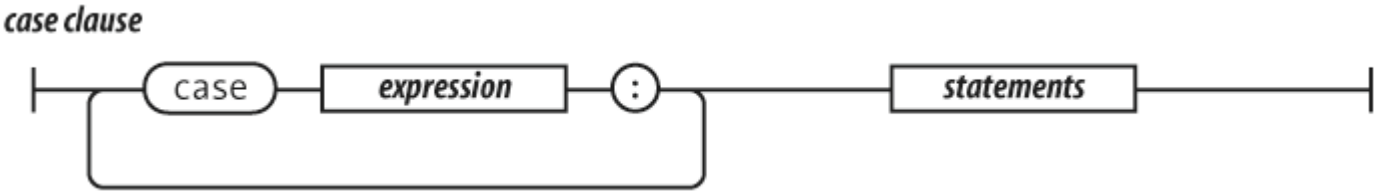
En este caso es más conveniente utilizar una estructura de control de tipo switch, ya que permite ahorrarnos trabajo y producir un código más limpio. Su definición formal es la siguiente:




```
switch(dia) {
  case 1: console.log("Hoy es lunes."); break;
  case 2: console.log("Hoy es martes."); break;
  case 3: console.log("Hoy es miércoles."); break;
  case 4: console.log("Hoy es jueves."); break;
  case 5: console.log("Hoy es viernes."); break;
  case 6: console.log("Hoy es sábado."); break;
  default : console.log("Hoy es domingo."); break;
```

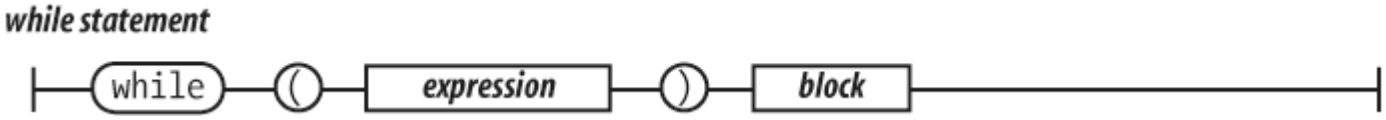
```
}
```

La cláusula **case** no tiene por qué ser una constante, sino que puede ser una expresión al igual que en la estructura **if**. El comportamiento por defecto de la estructura **switch** es seguir evaluando el resto de cláusulas, aún cuando una de ellas haya cumplido la condición. Para evitar ese comportamiento, es necesario utilizar la sentencia **break** en las cláusulas que deseemos.



3.3. Estructura while

La estructura **while** ejecuta un simple bucle, mientras se cumpla la condición. Su definición formal es la siguiente:




```
var veces = 0;

while(veces < 7) {
  console.log("Mensaje " + veces);
  veces++;
}
```

La idea del funcionamiento de un **bucle while** es la siguiente: "mientras la condición indicada se siga cumpliendo, repite la ejecución de las instrucciones definidas dentro del while. Es importante modificar los valores de las variables incluidas dentro de la condición, ya que otra manera, el bucle se repetirá de manera indefinida, perjudicando la ejecución de la página y bloqueando la ejecución del resto del script.

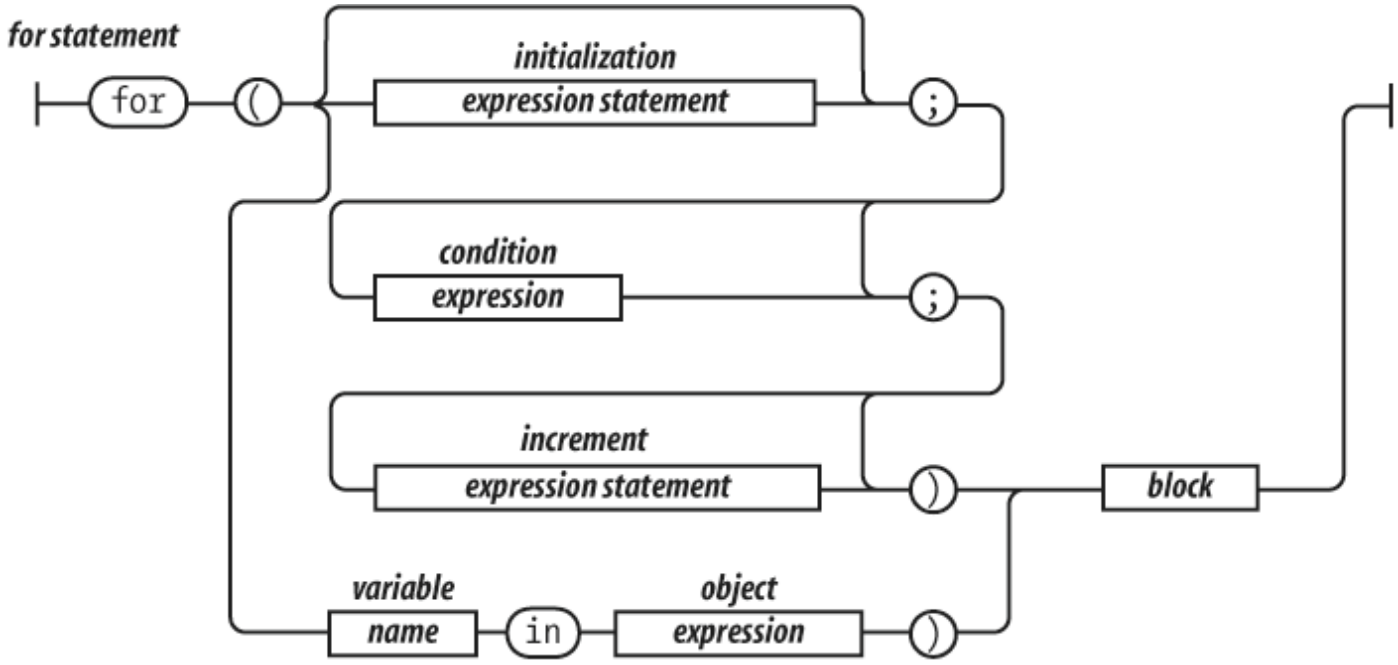
```
var veces = 0;

while(veces < 7) {
  console.log("Mensaje " + veces);
  veces = 0;
}
```

En este ejemplo, se mostraría de manera infinita una alerta con el texto "Mensaje 0".

3.4. Estructura for

La estructura **for** permite realizar bucles de una forma muy sencilla. Su definición formal es la siguiente:




```
for(inicializacion; condicion; actualización) {
  ...
}
```

La idea del funcionamiento de un bucle for es la siguiente: *"mientras la condición indicada se siga cumpliendo, repite la ejecución de las instrucciones definidas dentro del for. Además, después de cada repetición, actualiza el valor de las variables que se utilizan en la condición"*.

- La "inicialización" es la zona en la que se establece los valores iniciales de las variables que controlan la repetición.
- La "condición" es el único elemento que decide si continúa o se detiene la repetición.
- La "actualización" es el nuevo valor que se asigna después de cada repetición a las variables que controlan la repetición. var mensaje = "Hola, estoy dentro de un bucle";

```
for(var i = 0; i < 5; i++) {
  console.log(mensaje);
}
```

Un ejemplo que muestre los días de la semana contenidos en un array se puede hacer utilizando la estructura **for**:

```
var dias = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado",
"Domingo"];

for(var i=0; i<7; i++) {
  console.log(dias[i]);
}
```

3.5. Estructura for...in

Una estructura de control derivada de **for** es la **estructura for...in**. Su definición exacta implica el uso de objetos, permitiendo recorrer las propiedades de un objeto. En cada iteración, un nuevo nombre de propiedad del objeto es asignada a la variable:

```
for(propiedad in object) {
  if (object.hasOwnProperty(propiedad)) {
    ...
  }
}
```

Si se quieren **recorrer todos los elementos que forman un array**, la estructura **for...in** es la forma más eficiente de hacerlo, como se muestra en el siguiente ejemplo:

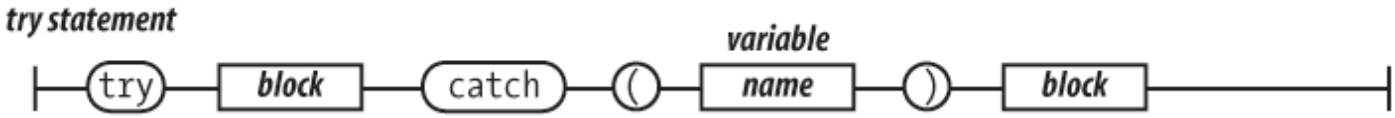
```
var dias = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado",
"Domingo"];

for(i in dias) {
  console.log(dias[i]);
}
```

Esta estructura de control es la más adecuada para recorrer arrays (y objetos), ya que evita tener que indicar la inicialización y las condiciones del bucle for simple y funciona correctamente cualquiera que sea la longitud del array. De hecho, sigue funcionando igual aunque varíe el número de elementos del array.

3.6. Estructura try

La estructura try consiste en un bloque de código que se ejecuta de manera normal, y captura cualquier excepción que se pueda producir en ese bloque de sentencias. Su definición formal es la siguiente:




```
try {
    funcion_que_no_existe();
} catch(ex) {
    console.log("Error detectado: " + ex.description);
}
```

En este ejemplo, llamamos a una función que no está definida, y por lo tanto provoca una excepción en JavaScript. Este error es *capturado* por la cláusula catch, que contiene una serie de sentencias que indican que acciones realizar con esa excepción que acaba de producirse. Si no se produce ninguna excepción en el bloque try, no se ejecuta el bloque dentro de catch.

3.6.1 La cláusula finally

La cláusula finally contiene las sentencias a ejecutar después de los bloques try y catch. Las sentencias incluidas en este bloque se ejecutan siempre, se haya producido una excepción o no. Un ejemplo clásico de utilización de la cláusula finally, es la de *liberar recursos que el script ha solicitado*.

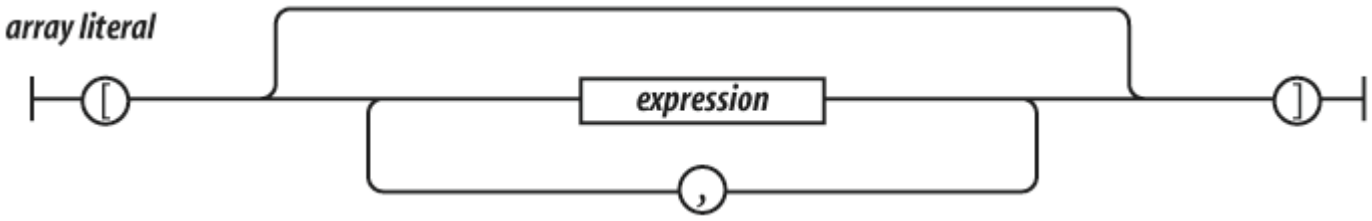
```
abrirFichero()
try {
   escribirFichero(datos);
} catch(ex) {
    // Tratar la excepción
} finally {
   cerrarFichero(); // siempre se cierra el recurso
}
```

4. Arrays

Un array es una asignación lineal de memoria donde los elementos son accedidos a través de índices numéricos, siendo además una estructura de datos muy rápida. Desafortunadamente, JavaScript no utiliza este tipo de arrays. En su lugar, JavaScript ofrece un objeto que dispone de características que le hacen parecer un array. Internamente, **convierte los índices del array en strings** que son utilizados como nombres de propiedades, haciéndolo sensiblemente **más lento** que un array.

4.1. Representación de un array

JavaScript ofrecen una manera muy cómoda para crear y representar arrays. Una representación de un array consiste en una **pareja de corchetes ([ ]) que contienen cero o más expresiones**. **El primer valor recibe la propiedad de nombre '0', la segunda la propiedad de nombre '1', y así sucesivamente.** Se define de la siguiente manera:



Algunos ejemplos de declaración de arrays:

```
var empty = [];
 var numbers = [
    'zero', 'one', 'two', 'three', 'four',
    'five', 'six', 'seven', 'eight', 'nine'
];
empty[1]        // undefined
numbers[1]      // 'one'
empty.length    // 0
numbers.length  // 10
```

Si representamos el array como un **objeto**

```
var numbers_object = {
    '0': 'zero', '1': 'one',    '2': 'two',
    '3': 'three', '4': 'four', '5': 'five',
    '6': 'six', '7': 'seven', '8': 'eight',
    '9': 'nine'
];
```

se produce un efecto similar. Ambas representaciones contienen 10 propiedades, y estas propiedades tienen exactamente los mismos nombres y valores. La diferencia radica en que **numbers hereda de Array.prototype**, mientras que **numbers_object lo hace de Object.prototype**, por lo que **numbers** hereda una serie de métodos que *convierten* a numbers en un array.

En la mayoría de los lenguajes de programación, se requiere que todos los elementos de un array sean del mismo tipo, pero en JavaScript eso no ocurre. Un array puede contener una mezcla valores:

```
var misc = [
    'string', 98.6, true, false, null, undefined,
    ['nested', 'array'], {object: true}, NaN,
    Infinity
];
misc.length // 10
```

4.2. Propiedad length

Todo array tiene una propiedad **length**. **A diferencia de otros lenguajes, la longitud del array no es fija, y podemos añadir elementos de manera dinámica.** Esto hace que la propiedad length varíe, y tenga en cuenta los nuevos elementos. La propiedad length hace referencia al mayor índice presente en el array, más uno. Esto es:

```
var myArray = [];
myArray.length              // 0
myArray[1000000] = true;
myArray.length              // 1000001
// myArray contiene un elemento!
```

4.3. Borrado

Como los arrays de JavaScript son realmente objetos, **podemos utilizar el operador delete para eliminar elementos de un array:**

```
delete numbers[2];
// numbers es ['zero', 'one', undefined, 'shi', 'go']
```

Desafortunadamente, **esto deja un espacio en el array**. Esto es porque los elementos a la derecha del elemento eliminado conservan sus nombres. Para este caso, **JavaScript incorpora una función splice, que permite eliminar y reemplazar elementos de un array. El primer argumento indica la posición por la que empezar a cortar, y el segundo argumento indica el número de elementos a eliminar.**

```
numbers.splice(2, 1);
// numbers es ['zero', 'one', 'shi', 'go']
```

Nota: ¿Sabes que con el método splice también se pueden añadir valores? Para más información puedes echarle un vistazo a este artículo sobre **splice** que es la leche (no es obligatorio): https://catonmat.net/splice-push-pop-shift-unshift

5. Funciones y propiedades básicas

JavaScript incorpora una serie de herramientas y utilidades (llamadas funciones y propiedades, como se verá más adelante) para el manejo de las variables. De esta forma, muchas de las operaciones básicas con las variables, se pueden realizar directamente con las utilidades que ofrece JavaScript.

5.1. Funciones útiles para cadenas de texto

A continuación se muestran algunas de las funciones más útiles para el manejo de cadenas de texto:

length, calcula la longitud de una cadena de texto (el número de caracteres que la forman)

```
var mensaje = "Hola Mundo";
var numeroLetras = mensaje.length; // numeroLetras = 10
```

Operador +, se emplea para concatenar varias cadenas de texto

```
var mensaje1 = "Hola";
var mensaje2 = " Mundo";
var mensaje = mensaje1 + mensaje2; // mensaje = "Hola Mundo"
```

Además del **operador +**, también se puede utilizar la función **concat()**

```
var mensaje1 = "Hola";
var mensaje2 = mensaje1.concat(" Mundo"); // mensaje2 = "Hola Mundo"
```

Las cadenas de texto también se pueden unir con variables numéricas:

```
var variable1 = "Hola ";
var variable2 = 3;
var mensaje = variable1 + variable2; // mensaje = "Hola 3"
```

Cuando se unen varias cadenas de texto es habitual olvidar añadir un espacio de separación entre las palabras:

```
var mensaje1 = "Hola";
```

```
var mensaje2 = "Mundo";
var mensaje = mensaje1 + mensaje2; // mensaje = "HolaMundo"
```

Los espacios en blanco se pueden añadir al final o al principio de las cadenas y también se pueden indicar forma explícita:

```
var mensaje1 = "Hola";
var mensaje2 = "Mundo";
var mensaje = mensaje1 + " " + mensaje2; // mensaje = "Hola Mundo"
```

toUpperCase(), transforma todos los caracteres de la cadena a sus correspondientes caracteres en mayúsculas:

```
var mensaje1 = "Hola";
var mensaje2 = mensaje1.toUpperCase(); // mensaje2 = "HOLA"
```

toLowerCase(), transforma todos los caracteres de la cadena a sus correspondientes caracteres en minúsculas:

```
var mensaje1 = "HolA";
var mensaje2 = mensaje1.toLowerCase(); // mensaje2 = "hola"
```

charAt(posicion), obtiene el carácter que se encuentra en la posición indicada:

```
var mensaje = "Hola";
var letra = mensaje.charAt(0); // letra = H
letra = mensaje.charAt(2);     // letra = l
```

indexOf(caracter), calcula la posición en la que se encuentra el carácter indicado dentro de la cadena de texto. Si el carácter se incluye varias veces dentro de la cadena de texto, se devuelve su primera posición empezando a buscar desde la izquierda. Si la cadena no contiene el carácter, la función devuelve el valor -1:

```
var mensaje = "Hola";
var posicion = mensaje.indexOf('a'); // posicion = 3
posicion = mensaje.indexOf('b');     // posicion = -1
```

Su función análoga es **lastIndexOf**():

lastIndexOf(caracter), calcula la última posición en la que se encuentra el carácter indicado dentro de la cadena de texto. Si la cadena no contiene el carácter, la función devuelve el valor -1:

```
var mensaje = "Hola";
var posicion = mensaje.lastIndexOf('a'); // posicion = 3
posicion = mensaje.lastIndexOf('b');     // posicion = -1
```

La función lastIndexOf() comienza su búsqueda desde el final de la cadena hacia el principio, aunque la posición devuelta es la correcta empezando a contar desde el principio de la palabra.

substring(inicio, final), extrae una porción de una cadena de texto. El segundo parámetro es opcional. Si sólo se indica el parámetro inicio, la función devuelve la parte de la cadena original correspondiente desde esa posición hasta el final:

```
var mensaje = "Hola Mundo";
var porcion = mensaje.substring(2); // porcion = "la Mundo"
porcion = mensaje.substring(5);     // porción = "Mundo"
porcion = mensaje.substring(7);     // porción = "ndo"
```

Si se indica un inicio negativo, se devuelve la misma cadena original:

```
var mensaje = "Hola Mundo";
var porcion = mensaje.substring(-2); // porcion = "Hola Mundo"
```

Cuando se indica el inicio y el final, se devuelve la parte de la cadena original comprendida entre la posición inicial y la inmediatamente anterior a la posición final (es decir, la posición inicio está incluida y la posición final no):

```
var mensaje = "Hola Mundo";
var porcion = mensaje.substring(1, 8); // porcion = "ola Mun"
porcion = mensaje.substring(3, 4);     // porcion = "a"
```

Si se indica un final más pequeño que el inicio, JavaScript los considera de forma inversa, ya que automáticamente asigna el valor más pequeño al inicio y el más grande al final:

```
var mensaje = "Hola Mundo";
var porcion = mensaje.substring(5, 0); // porcion = "Hola "
porcion = mensaje.substring(0, 5);     // porcion = "Hola "
```

split(separador), convierte una cadena de texto en un array de cadenas de texto. La función parte la cadena de texto determinando sus trozos a partir del carácter separador indicado:

```
var mensaje = "Hola Mundo, soy una cadena de texto!";
var palabras = mensaje.split(" ");
// palabras = ["Hola", "Mundo,", "soy", "una", "cadena", "de", "texto!"];
```

Con esta función se pueden extraer fácilmente las letras que forman una palabra:

```
var palabra = "Hola";
var letras = palabra.split(""); // letras = ["H", "o", "l", "a"]
```

Nuevas funciones para Strings incluidas en EcmaScript 6:

Normalmente para encontrar una cadena dentro de un texto se utiliza la función “indexOf” pero en la última revisión se ha incluido 3 nuevos métodos que nos puede ayudar a la hora de encontrar cadenas:

- **include(“cadena_ha_buscar”)** : Nos devuelve true o false si se ha encontrado la cadena buscada.

- **startsWith()** y **endWith()**: Nos devuelven true o false si se ha encontrado la cadena buscada al principio o al final de la cadena.

Veamos un ejemplo de su utilización:

```
let msg = "Hello world!";
console.log(msg.startsWith("Hello"));       // true
console.log(msg.endsWith("!"));             // true
console.log(msg.includes("o"));             // true

console.log(msg.startsWith("o"));           // false
console.log(msg.endsWith("world!"));        // true
console.log(msg.includes("x"));             // false

console.log(msg.startsWith("o", 4));        // true
console.log(msg.endsWith("o", 8));          // true
console.log(msg.includes("o", 8));          // false
```

5.2. Funciones útiles para arrays

A continuación se muestran algunas de las funciones más útiles para el manejo de arrays:

length, calcula el número de elementos de un array

```
var vocales = ["a", "e", "i", "o", "u"];
var numeroVocales = vocales.length; // numeroVocales = 5
```

concat(), se emplea para concatenar los elementos de varios arrays

```
var array1 = [1, 2, 3];
array2 = array1.concat(4, 5, 6);   // array2 = [1, 2, 3, 4, 5, 6]
array3 = array1.concat([4, 5, 6]); // array3 = [1, 2, 3, 4, 5, 6]
```

join(separador), es la función **contraria a split()**. Une todos los elementos de un array para formar una cadena de texto. Para unir los elementos se utiliza el carácter separador indicado

```
var array = ["hola", "mundo"];
var mensaje = array.join(""); // mensaje = "holamundo"
mensaje = array.join(" ");    // mensaje = "hola mundo"
```

pop(), elimina el último elemento del array y lo devuelve. El array original se modifica y su longitud disminuye en 1 elemento.

```
var array = [1, 2, 3];
var ultimo = array.pop();
// ahora array = [1, 2], ultimo = 3
```

push(), añade un elemento al final del array. El array original se modifica y aumenta su longitud en 1 elemento. (También es posible añadir más de un elemento a la vez).

```
var array = [1, 2, 3];
array.push(4);
// ahora array = [1, 2, 3, 4]
```

shift(), elimina el primer elemento del array y lo devuelve. El array original se ve modificado y su longitud disminuida en 1 elemento.

```
var array = [1, 2, 3];
var primero = array.shift();
// ahora array = [2, 3], primero = 1
```

unshift(), añade un elemento al principio del array. El array original se modifica y aumenta su longitud en 1 elemento. (También es posible añadir más de un elemento a la vez)

```
var array = [1, 2, 3];
array.unshift(0);
// ahora array = [0, 1, 2, 3]
```

reverse(), modifica un array colocando sus elementos en el orden inverso a su posición original:

```
var array = [1, 2, 3];
array.reverse();
// ahora array = [3, 2, 1]
```

5.3. Funciones y propiedades útiles para números

A continuación se muestran algunas de las funciones y propiedades más útiles para el manejo de números.

NaN, (del inglés, "Not a Number") JavaScript emplea el valor NaN para indicar un valor numérico no definido (por ejemplo, la división 0/0).

```
var numero1 = 0;
var numero2 = 0;
console.log(numero1/numero2); // se muestra el valor NaN
```

isNaN(), permite proteger a la aplicación de posibles valores numéricos no definidos

```
var numero1 = 0;
var numero2 = 0;
if(isNaN(numero1/numero2)) {
  console.log("La división no está definida para los números indicados");
} else {
  console.log("La división es igual a => " + numero1/numero2);
}
```

Infinity, hace referencia a un valor numérico infinito y positivo (también existe el valor –Infinity para los infinitos negativos)

```
var numero1 = 10;
var numero2 = 0;
console.log(numero1/numero2); // se muestra el valor Infinity
```

toFixed(digitos), devuelve el número original con tantos decimales como los indicados por el parámetro digitos y realiza los redondeos necesarios. Se trata de una función muy útil por ejemplo para mostrar precios.

```
var numero1 = 4564.34567;
numero1.toFixed(2); // 4564.35
numero1.toFixed(6); // 4564.345670
numero1.toFixed(); // 4564
```