1 Tipos de datos en JavaScript

1.1 Sumario

1.2 Tipos de datos

JavaScript es un lenguaje de tipado débil o dinámico. Esto significa que no es necesario declarar el tipo de variable antes de usarla. El tipo será determinado automáticamente cuando el programa comience a ser procesado. Esto también significa que puedes tener la misma variable con diferentes tipos.

Se dispone de dos maneras de crear variables en JavaScript: una forma es usar la palabra reservada **var** (o **let** o **const** como veremos luego) seguida del nombre de la variable. Por ejemplo, para declarar una variable **edad**, el código de JavaScript será:

```
var edad;
```

Otra forma consiste en crear la variable, y asignarle un valor directamente durante la creación:

```
let edad = 38;
//Aunque no estamos obligados a declarar la variable:
edad = 38;
//pero es una buena práctica el hacerlo pues
// al no indicar 'var' (o 'let' o 'const' como veremos luego)
// el scope (ámbito) de esa variable será 'global'

// Para declarar más de una variable en la misma línea:
let altura, peso, edad;
```

Una variable de JavaScript podrá almacenar diferentes tipos de valores, y así no tendremos que decirle de qué tipo es una variable u otra. A la hora de dar nombres a las variables, tendremos que poner nombres que realmente describan el contenido de la variable. No podremos usar palabras reservadas (*return*, *for*...), ni símbolos de puntuación en el medio de la variable, ni la variable podrá contener espacios en blanco. Los nombres de las variables han de construirse con caracteres alfanuméricos y el carácter subrayado (_). No podremos utilizar caracteres raros como el signo **+**, un espacio, **%** , **$**, etc. y nunca podrán comenzar con un número. Si queremos nombrar variables con dos palabras, tenemos dos opciones: separarlas con el símbolo ?_? o bien diferenciar las palabras con una mayúscula (estilo "*camel case*"), por ejemplo:

```
var mi_peso;  // Separar con guiones bajos
var miPeso;  // Estilo camel case, mejor opción
```

La última definición del estándar ECMAScript define siete tipos de datos:

- Seis tipos de datos que son primitivos:
  - ◊ *Boolean*
  - ◊ *Null*
  - ◊ *Undefined*
  - ◊ *Number*
  - ◊ *String*
  - ◊ *Symbol* (nuevo en ECMAScript 6)
- y *Object* (que los veremos en el punto 2.5)

1.2.1 Definición de variables con *var*, *let* y *const* en JavaScript

En ES5 sólo se cuenta con **var** como palabra reservada para definir variables; pero en ES6 existen tres modos de definir variables: **var**, **let** y **const**. Veamos las diferencias entre ellas y cuándo es más adecuado utilizar una u otra.

- **var**

  var se utiliza para declarar una variable y, adicionalmente, se puede inicializar el valor de esta variable. Por ejemplo: **var i = 0**.
  Podemos decir que:
  - Las variables declaradas con **var** son procesadas antes de la ejecución del código.
  - El *scope* de una variable declarada con **var**, es su contexto de ejecución.
  - El *scope* de una variable declarada fuera de la función es global.

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta charset="UTF-8" />
    <title>Entender var</title>
  </head>
  <body>
    <script type="text/javascript">
      function entenderVar() {
        var a = 10;
        console.log(a); //output 10
        if (true) {
          var a = 20;
          console.log(a); //output 20
        }
        console.log(a); //output 20
      }
      //Llamamos la función
      entenderVar();
    </script>
  </body>
</html>
```

Se puede ver que, cuando la variable es actualizada dentro del condicional **if**, el valor de ésta se actualiza a **20** a nivel global.

Veamos este otro ejemplo:

```
<script type="text/javascript">
  var i = 60;
    function entenderVar() {
      for (var i = 0; i < 5; i++) {
        console.log(i); // Output 0, 1, 2, 3, 4
      }
    }
    //Llamamos la función
    entenderVar();
```

```
    //Mostramos el valor de i
    console.log("Despues del loop", i); // Output 60
</script>
```

Si declaramos la variable dentro de una función, el valor de nuestra variable vivirá solamente dentro del scope de esta función. Por lo tanto, fuera de entenderVar() nuestra variable tomará el valor asignado previamente. Pero, si no definimos la variable dentro de la función?

```
<script type="text/javascript">
  var i = 60;
  function entenderVar() {
    for (i = 0; i < 5; i++) {
      console.log(i); //Output 0, 1, 2, 3, 4
    }
  }
  //Llamamos la función
  entenderVar();
  //Mostramos el valor de i
  console.log("Después del loop", i); // Output 5
</script>
```

Importante:

La explicación de esta última salida sería que, si olvidas declarar una variable (no indicas ni **var**, ni **let**, ni **const**) dentro de tu función o **loop** (en este caso no declaramos **i** dentro del **for**), el intérprete de JavaScript la va a declarar de forma global. El valor de **i** a nivel global será reasignado por el *for loop*.
En JavaScript las variables son ?hoisted? o ?izadas?. Esto, como su nombre indica, quiere decir que una variable es izada (subida) hasta el tope de la función o hasta llegar al inicio del *Window Object*. Cuando no declaramos una variable, JavaScript la crea en el objeto **window** (la iza completamente). Llegando incluso a reasignar el valor de ésta sin pedirnos permiso. Así que, hay que tener mucho cuidado con esto, pues se puede terminar creando variables globales y reasignando valores sin darse uno cuenta.

- **let**

Se introduce en JavaScript versión ES6, teniendo **let** un *scope* a nivel de bloque.

```
<script type="text/javascript">
  var i = 60;
  function entenderLet() {
    for (let i = 0; i < 5; i++) {
      console.log(i); //Output 0, 1, 2, 3, 4
    }
  }
  //Llamamos la función
  entenderLet();
  //Mostramos el valor de i
  console.log("Después del loop", i); // Output 60
</script>
```

¡Funciona de acuerdo a lo esperado! Esto pasa porque **let** introduce el "block scope". La variable asignada como **let** solo será accesible dentro del *for loop*.

Veamos un par de ejemplos más. En este primero se ve un comportamiento muy parecido al de otros lenguajes:

```
function entenderLet() {
    let a = 10;
    console.log(a);        // output 10

    if(true) {
        let a = 20;
        console.log(a); }// output 20

    console.log(a);        // output 10
}
```

Y podemos repetir este ejemplo anterior cambiando **let** por **var** dentro del **if**, deduciendo así cómo funciona **let**.

Y este otro ejemplo, donde vemos que se produce un error por declarar dos veces la misma variable:

```
function entenderLet() {
    let a = 10;
    let a = 20; //error
    console.log(a);
```

```
}
```

Vemos que, el uso de **let** mantendrá las funciones limpias y claras. Se puede decir que **let** es el nuevo **var**, con ES6 no hay razones para usar **var**.

- **const**

const es igual que **let**, pero con una diferencia importante, no se puede reasignar su valor.

```
function entenderConst() {
    const x = 10;
    console.log(x);        // output 10
}
```

¿Qué pasa si reasignamos el valor de la variable **const**?

```
function entenderConst() {
    const x = 10;
    console.log(x);        // output 10
    x = 20;        // error
    console.log(x);
}
```

La consola mostrará un error cuando se trate de re-asignar el valor de una variable **const**. Así, vemos que es ideal usar **const** si no tienes que re-asignar el valor de una variable en ningún momento. Eso es, porque se quiere minimizar el estado de mutación (*mutable state*). Es importante saber que en nuestro código estarán corriendo muchos procesos a la vez. Cada vez que uses *let* es bueno analizar si realmente cambiará el valor de esta variable en algún momento. De no ser así, mejor utilizar **const**.

◊ Post de donde se sacó esta parte de los apuntes **var & let & const**.

1.2.2 Valores primitivos

Todos los tipos, excepto "los Objetos", definen **valores inmutables** (valores que no pueden ser cambiados, al modificar un valor se crea uno nuevo, con el mismo nombre pero en una dirección de memoria distinto).

1.2.2.1 Boolean

Estas variables sólo pueden contener dos valores: *true* (verdadero) o *false* (falso).

```
var valido = true;
```

1.2.2.2 Number

De acuerdo al standard ECMAScript, sólo existe un tipo numérico: el valor de doble precisión de 64-bits IEEE 754, un número entre $-(2^{53}-1)$ y $2^{53}-1$.

No existe un tipo específico para los números enteros.

También es capaz de representar números de coma flotante.

Como en todos los lenguajes, es posible operar con estos números.

```
let minumero = 2;
minumero = minumero * 25 + 3; //Salida : 53
alert(minumero);
```

Podemos crear números en distintas bases de modo sencillo:

```
let numbase10 = 1234;
let numbase8 = 0123;
let numbase16 = 0xf7;
```

- El tipo número tiene los siguientes valores simbólicos :

  -Infinity < Number.MIN_SAFE_INTEGER < Number.MIN_VALUE < 0 < Number.MAX_SAFE_INTEGER < Number.MAX_VALUE < Infinity

- También tenemos la propiedad global NaN, que es un valor que representa *Not-A-Number* (No Es Un Número).

- **Generador de números aleatorios**

  El generador de números aleatorios será de gran importancia para simular condiciones. Para esta tarea, JavaScript tiene la función Math.random() que devuelve un número en coma flotante entre 0 y 1.
  Ejemplos de uso donde se utilizan también las funciones Math.floor, Math.round y Math.ceil:

```
Math.random() * 100; //Número flotante entre 0 y 100
Math.random() * 25 + 5 // Número flotante entre 5 y 30
Math.random() * 10 - 100 // Número flotante entre -100 y -90
Math.floor(Math.random() * 100) // Entero entre 0 y 99
Math.round(Math.random() * 25) + 5 // Entero entre 5 y 30
Math.ceil(Math.random() * 10) - 100 // Entero entre -100 y -90
```

1.2.2.3 String

El tipo **String** de Javascript es usado para representar datos textuales o cadenas de caracteres. Es un conjunto de "elementos", de valores enteros sin signo de 16 bits.

Cada elemento ocupa una posición en el *string*. El primer elemento está en el índice 0, el próximo está en el índice 1, y así sucesivamente. La longitud de un *string* es el número de elementos en ella.

```
var mitexto = "Hola Mundo!";
```

Para concatenar texto podemos emplear el operador **+**.

```
var mitexto = "Mi nombre es ";
mitexto = mitexto + "Juan";
/* Si concatenamos un número,
el número se convierte en una cadena de caracteres */
var mitexto = "El número es " + 3;
```

Se pueden definir las cadenas de texto con comillas simples y dobles. Si queremos utilizarlas en el interior del texto hay que "escaparlas".

```
var mitexto = 'El \'libro\' es interesante';
```

También disponemos de caracteres de escape como **\n** para generar una nueva línea, **\r** para devolver el cursor al comienzo de la línea, **\t** para introducir un tabulado, y otros... En el siguiente código se generan dos líneas de texto.

```
var mitexto = "\"Felicidad\" no es hacer lo que uno quiere\n";
mitexto += "sino querer lo que uno hace"
```

Las cadenas se escriben siempre con comillas simples o dobles, ambas se comportan de la misma manera. La única diferencia es qué tipo de comillas necesitas para escapar una cita dentro de ellas.

Las cadenas con **comillas invertidas (`´)**, generalmente llamadas **literales de plantilla**, pueden hacer algunos trucos más. Además de poder abarcar líneas, también pueden incrustar otros valores.

Un ejemplo sería:

```
`La mitad de 100 es ${100 / 2}`
```

Cuando se escribe algo dentro de **${}** en el interior de unas comillas invertidas, su resultado se calculará, se convertirá en una cadena y se incluirá en esa posición.

El ejemplo anterior anterior produce:

```
// "La mitad de 100 es 50"
```

1.2.2.4 Symbol

El **Symbol** es un nuevo tipo en JavaScript introducido en la versión ECMAScript Edition 6. Un *Symbol* es un valor primitivo único e inmutable y puede ser usado como la clave de una propiedad de un *Object*.

Las utilidades prácticas más usadas de este tipo de datos es la de servir como identificadores para guardar las propiedades de un objeto, ya que usando los símbolos, evitamos borrar ninguna de esas propiedades al no permitir sobreescribirlas.

1.2.2.5 null y undefined

Son dos valores que utiliza JavaScript para indicar "que falta información en el *script*" (explicación).

· **undefined** significa que no hay ni un valor primitivo ni un objeto, y podemos encontrarlo en variables sin inicializar, en una falta de parámetros o en una omisión de propiedades.

· **null** significa que no hay ningún objeto, y podemos encontrarlo en aquellas partes del *script* en que se espera la existencia de un objeto, sea del tipo que sea.

Resumiendo: **undefined** indica la no existencia y **null** el vacío.

Así, para **null** :

```
Number(null) //El resultado sería 0
5 + null //El resultado sería 5
```

Y para **undefined** :

```
Number(undefined) //El resultado sería NaN
5 + undefined //El resultado sería NaN
```

Cuando se necesita saber si el valor de una variable es **null**, es necesario comparar esa variable con el valor **null** utilizando el uso del comparador "estrictamente igual" : **===**. También se puede comprobar la existencia de una variable **undefined** utilizando el mismo comparador.

```
if (typeof(cosa) === "undefined") {
alert("cosa no está definido");
}
```

> **typeof** es un operador que devuelve una cadena, o *string*, que describe el tipo de dato que corresponde al objeto, ya sea una variable, una función...: *undefined*, *boolean*, *number*, *string*, *object* (si la variable es un objeto o de tipo *null*).

1.2.3 Objetos

Los objetos son estructuras de información capaces de contener propiedades y métodos. **ECMA-262** define objeto como "una colección sin ordenar de propiedades que contienen un valor primitivo, un objeto o una función".

Todo objeto se define mediante una **clase**, que puede definirse como "la receta del objeto". Cuando un programa utiliza una clase para crear un objeto, el objeto resultante se denomina instancia de la clase.

- **Objetos nativos**

  Son cualquier objeto proporcionado por una implementación de JavaScript independiente del entorno anfitrión, se trata de los siguientes: **object, function, array, string, boolean, number, date, regexp, error, evalerror, rangeerror, referenceerror, syntaxerror, typeerror, urierror**.

- **Objetos incorporados**

  Son cualquier objeto proporcionado por una implementación de JavaScript, independientemente del entorno anfitrión, que está presente al iniciarse la ejecución de un programa de JavaScript. Son los siguientes: **global, math**.

- **Objetos anfitrión**

  Cualquier objeto que no sea nativo se considera anfitrión. Todos los objetos BOM y DOM se consideran objetos anfitrión y se analizarán en un apartado posterior.

Aunque tendremos una sección dedicada a los objetos, en el siguiente enlace podemos ver una introducción interesante.

En muchos programas emplearemos los objetos simplemente como los objetos diccionario de Python. Veamos un ejemplo sacado de la web "Developer.mozilla":

```
var myCar = {
```

```
    make: 'Ford',
    model: 'Mustang',
    year: 1969
};
//Para acceder a su contenido:
console.log(myCar.model); //'Mustang'
//O también empleando los corchetes:
console.log(myCar['year']); //1969
//Mostrarlas todas con un loop
console.log("Recorremos todas las propiedades del diccionario:")
for (const propiedad in myCar) {
    console.log(propiedad + " : " + myCar[propiedad]);
  }

//Las propiedades no asignadas de un objeto son undefined (y ?no null).
console.log(myCar.color); // undefined

//Añadir una nueva propiedad al diccionario
myCar.color = 'black';
console.log(myCar.color); //'black'

//Ver si una propiedad existe en el diccionario
('numRuedas' in myCar)?console.log('Si numRuedas'):console.log('No numRuedas');
('color' in myCar)?console.log('Hay color'):console.log('No hay color');
```

Vemos así que, los objetos, a veces son llamados **arreglos asociativos** (diccionarios), debido a que cada propiedad está asociada con un valor de cadena que se puede utilizar para acceder a ella.

1.2.3.1 Arrays

Un array es una colección ordenada de datos (tanto primitivos como otros objetos). Los arrays se emplean para almacenar multiples valores en una sola variable, frente a las variables que sólo pueden almacenar un valor (por cada variable).

```
var miarray = ["rojo", "verde", "azul"];
console.log(miarray[0]); //rojo
```

También podemos crear un array vacío:

```
var miarray = [];
var miarray = new Array(); // Otro modo de hacerlo
```

Los arrays pueden contener cualquier tipo de valor que deseemos.

```
var miarray = ["rojo", 32, "HTML5 es genial"];
console.log(miarray); //rojo, 32, HTML5 es genial
```

Si se intenta leer un valor en un índice que aún no se ha definido, JavaScript devuelve el valor ***undefined*** (indefinido). También podemos asignar este valor cuando aún no contamos con el valor para esa posición.

```
var miarray = ["rojo", undefined, 32];
console.log(miarray[1]);
```

Con el valor especial ***null*** (nulo) también se le puede indicar al sistema que no existe un valor disponible en ese momento. La diferencia, como ya se comentó antes, es: "**undefined** indica que la variable fue declarada pero ningún valor le fue asignado, mientras que **null** indica que existe un valor, pero es nulo".

Los arrays pueden incluir cualquier tipo de valores, por lo que es posible declarar *arrays* de *arrays* (**arrays multidimensionales**).

```
var miarray = [[2, 45, 31], [5, 10], [81, 12]];
alert(miarray[1][0]); //5
```

Si queremos eliminar uno de los valores, podemos declararlo como **undefined** o **null**, o declararlo como un array vacío asignando corchetes sin valores en su interior.

```
var miarray = [[2, 45, 31], [5, 10], [81, 12]];
miarray[1][0] = []
alert(miarray[1][0]); //undefined
```

Si queremos ver el número de elementos que tiene un array utilizamos la propiedad **length**:

```
var miarray = [[2, 45, 31], [5, 10], [81]];
console.log(miarray.length);        // devuelve 3
console.log(miarray[0].length);        // devuelve 3
console.log(miarray[miarray.length - 1].length); // devuelve 1
```

Para iterar en el array podemos hacerlo de los siguientes modos:

```
var array1 = [2, 45, 31];
// for (variables;condición;modificación)
for ( let i=0, len=array1.length; i<len; i++) {
    console.log(array1[i]);
}

// while
let counter=0;
while(counter<array1.length) {
    //pon aquí el código
    counter++;
}

// loop infinito con while
while(true) {
    if (breakCondition) {
        break;
    }
}
// loop infinito con for
for ( ; ;) {
    if (breakCondition) {
        break;
    }
}

// for (in)
for (let i in array1) {
    console.log(i)
}

// for (of)
for (let e in array1) {
    console.log(e)
}

// forEach()
array1.forEach(function (elemento, indice, array) {
console.log(indice + '->' + elemento);
});
```

Métodos básicos de los arrays:

```
var miarray = [2, 45, 31];
// Añadir elemento al final del array
var longitud = miarray.push(10); // ahora [2, 45, 31, 10]
console.log(longitud); //muestra 4
// Eliminar al final
var eliminado = miarray.pop(); // ahora [2, 45, 31]
console.log(eliminado); // muestra 10
// Eliminar del inicio
var eliminado = miarray.shift(); // ahora [45, 31]
console.log(eliminado); // muestra 2
// Añadir al inicio
var longitud = miarray.unshift(1); // ahora [1, 45, 31]
console.log(longitud); // muestra 3

//Vemos como va el array
console.log(miarray) // muestra [1, 45, 31]

// Encontrar el índice de un elemento
var pos = miarray.indexOf(45);
console.log(pos); // muestra 1
```

```
// ¿Y si el elemento NO está en el array?
var pos = miarray.indexOf(100);
console.log(pos) // muestra -1

// Eliminar UN elemento indicando posición por índice
// splice(inicio[,numEleBorrados[,eleAnade1[,eleAnade2,[...]]]])
var eliminado = miarray.splice(pos, 1); // ahora [1, 31]
console.log(eliminado); //muestra [45]

// Si queremos una parte del array : slice(inicio,fin)
const array = [1, 45, 56, 78, 31];
array.slice(1,2); // [42]
array.slice(2,4); // [56,78]
// Si quiero copiar el contenido de un array en otro nuevo
const arrayCopia = array.slice();
// Otro modo de hacerlo es utilizando .from()
const arrayCopia2 = Array.from(array);

// Convertir el array en un string separado por un carácter especificado ('; ')
let texto = "";
texto = array.join('; ');
console.log(array) //muestra [1, 45, 56, 78, 31]
console.log(texto) //muestra 1; 45; 56; 78; 31
```

1.2.3.2 Combinación Arrays + Objetos

Ahora que ya conocemos el funcionamiento de los Objetos básicos y de los Arrays, podemos ver como es posible combinar ambos para obtener estructuras de datos más completas y útiles. Recordar que los corchetes son utilizados para indicar que se trata de un array y las llaves para indicar que se trata de un Objeto. Así, veamos un ejemplo de un "array de objetos" como tipo de estructura que combina ambos tipos de estructuras de datos:

```
var animals = [
 {
 species: "lion",
 class: "mammalia",
 order: "carnivora",
 extinct: false,
 number: 123456
 },
 {
 species: "gorilla",
 class: "mammalia",
 order: "primates",
 extinct: false,
 number: 555234
 },
 {
 species: "octopus",
 class: "cephalopoda",
 order: "octopoda",
 extinct: false,
 number: 333421
 }
];
```

Siendo el modo de acceder a cada uno de los elementos del siguiente modo:

```
animals[0].extinct //return false
animals[2]['species'] //return "octopus"
```

También podemos observar que este tipo de estructuras de datos podríamos utilizarlo para almacenar un CSV y luego acceder de un modo rápido y dinámico a cada uno de los campos. El contenido del CSV que con la misma información anterior tendría esta forma:

```
species;class;order;extinct;number
"lion";"mammalia";"carnivora";false;123456
"gorilla";"mammalia";"primates";false;555234
"octopus";"cephalopoda";"octopoda";false;333421
```

Por todo esto, vemos que es muy útil tener una función que convierta un archivo CSV a una estructura de datos Array de Objetos:

```
//Función que convierte un CSV se parado por , ; : en un Array de Objetos
```

```
//Ojo, que no existan en el interior de los campos ningún carácter , ; :
function csvToJson(csv,separador) {
let jsonCSV = [];

const patronLinea = /^.+$/gm;
let lineas = csv.match(patronLinea);

let listaCampos = lineas[0].split(separador);
for (let i = 1; i < lineas.length; i++) {
// console.log(lineas[i]);
let tmpObjeto = {};
let valorCampo = lineas[i].split(separador);
for (let j = 0; j < listaCampos.length; j++) {
let clave = listaCampos[j].replace(/[\"]/g, "");
let valor = valorCampo[j].replace(/[\"]/g, "");
tmpObjeto[clave] = valor;
}
jsonCSV.push(tmpObjeto);
}
    //console.log(jsonCSV);
    return jsonCSV;
}
//Llamamos la función del modo
let csv = 'Objeto que representa el archivo csv a leer';
//En este caso, nuestro CSV está separado por comas (podría ser por ; o :).
let miJson = csvToJson(csv,',');
```

Podemos ahora recorrer el JSON creado:

```
for (let i in jsonCSV) {
    console.log("--------------------");
    for (let c in jsonCSV[i]) {
        console.log(`${c} : ${jsonCSV[i][c]}`);
    }
}
```

1.2.4 Maps y Sets

ES6 introduce dos nuevos tipos de datos Map y Set:

- Los tipos de datos Map, como los objetos, asocian "claves" con "valores", pero ofrecen algunas ventajas que pueden ser interesantes en ciertas situaciones.
- Los tipos de datos Set, son similares a los arrays, a excepción de que no pueden tener elementos duplicados.

1.2.4.1 Maps

Antes de ES6, cuando e necesitaban asignar claves a valores, era obligatorio recurrir a un objeto, porque los objetos permiten asignar 'claves' a 'valores' de cualquier tipo. Sin embargo, el uso de objetos para este propósito tiene muchos inconvenientes:

- No hay una forma sencilla de saber cuántas asignaciones hay en un objeto.
- Las claves deben ser cadenas o símbolos, lo que impide asignar objetos a valores.
- Los objetos no garantizan ningún orden en sus propiedades.

El objeto Map aborda estas deficiencias y es una mejor opción para asignar claves a los valores (incluso si las claves son cadenas). Por ejemplo, imagina que tienes objetos de usuario desea mapear roles:

```
const u1 = {nombre: 'Ana'};
const u2 = {nombre: 'María'};
const u3 = {nombre: 'Juan'};
const u4 = {nombre: 'Pedro'};

const userRoles = new Map();

//Asignamos los usuarios a sus roles empleando el método set()
/// y aprovechando que es 'encadenable'
userRoles
    .set(u1, 'Admin')
    .set(u2, 'User')
```

```
    .set(u3, 'User');
//Pedro no tiene asignado ningún rol
```

También lo podríamos hacer asignando los datos desde un array. Así el código anterior quedaría del siguiente modo:

```
const u1 = {nombre: 'Ana'};
const u2 = {nombre: 'María'};
const u3 = {nombre: 'Juan'};
const u4 = {nombre: 'Pedro'};

const userRoles = new Map([
    [u1, 'Admin'],
    [u2, 'User'],
    [u3, 'User']
]);
```

Para ver si existe una clave en concreto y su valor empleamos los métodos **has()** y **get()**:

```
//Para recuperar el rol de 'u2' -> get()
console.log(userRoles.get(u2)); //'User'

//Si queremos saber si una clave existe -> has()
console.log(userRoles.has(u1)); //true
console.log(userRoles.get(u1)); //'Admin'
console.log(userRoles.has(u4)); //false
console.log(userRoles.get(u4)); //undefined
```

Si se llama al método **set()** con una clave que ya existe, ese valor será reemplazado:

```
console.log(userRoles.get(u2)); //'User'
console.log(userRoles.set(u2, 'Admin'));
console.log(userRoles.get(u2)); //'Admin'
```

La propiedad **size** nos devuelve el número de elementos del objeto **map**:

```
console.log(userRoles.size); // 3
```

Utilizando el método **keys()** obtendremos las claves existentes en el objeto **map**, con el método **values()** los valores y con el método **entries()** nos devolverá un array con las entradas donde el primer elemento será la clave y el segundo elemento será el valor. Todos estos métodos devuelven un objeto iterable, que podremos recorrer empleando la sentencia **for ... of**:

```
//Las claves
for (let u of userRoles.keys()) {
    console.log(u.nombre);
}

//Los valores
for (let r of userRoles.values()) {
    console.log(r);
}

//['clave' : 'valor']
for (let ur of userRoles.entries()) {
    console.log(`${ur[0].nombre} : ${ur[1]}`);
}
//Este último método también podemos utilizarlo así
//ya que el método 'entries()' es el de por defecto de los maps
for (let [u, r] of userRoles) {
    console.log(`${u.nombre} : ${r}`);
}
```

Para borrar una entrada de un objeto **map**, se puede utilizar el método **delete()**:

```
userRoles.delete(u2);
console.log(userRoles.size); //2
```

Por último, si queremos borrar todas las entradas del objeto **map**, emplearemos el método **clear()**:

```
userRoles.clear();
```

```
console.log(userRoles.size); //0
```

1.2.4.2 Sets

Un **set** es una colección de elementos que no permite la duplicidad de ninguno de ellos.

Veamos un ejemplo que sigue con el caso anterior:

```
// Creamos la instancia del objeto 'set'
const roles = new Set();

//Creamos los tipos de roles empleando el método 'add()'
roles.add("User"); //Set [ "User" ]
roles.add("Admin"); //Set [ "User", "Admin"]

//La propiedad 'size' nos devuelve el número de elementos
console.log(roles.size); //2
```

Lo mejor de los '*sets* llega ahora, pues al intentar añadir un dato ya existente, por ejemplo "User", no pasa nada:

```
roles.add("User");
console.log(roles.size); //2
```

Para eliminar un elemento, simplemente llamamos al evento **delete()**, que devuelve *true* si el elemento estaba en el **set** y *false* si ya no estaba:

```
console.log(roles.delete("Admin")); //true
console.log(roles); //Set [ "User" ]
console.log(roles.delete("Admin")); //false
```

1.2.5 Conversiones

Como todos los lenguajes, JavaScript permite convertir variables de un tipo a otro.

1.2.5.1 Convertir cadenas

JavaScript tiene la función **toString()** para pasar cualquier valor a *string*.

```
var aExiste = false;
alert(aExiste.toString())        //devuelve "false"
var iNum1 = 10;
var iNum2 = 10.73;
alert(iNum1.toString())        //devuelve "10"
alert(iNum2.toString())        //devuelve "10.73"
```

La conversión de números a cadenas también se puede hacer de un modo más sencillo, ya que, simplemente, tendrás que concatenar una cadena vacía al principio y, de esta forma, el número será convertido a su cadena equivalente.

```
console.log(typeof(" " + 3400))// devuelve string
```

En el siguiente ejemplo podemos ver la gran potencia de la evaluación de expresiones. Los paréntesis fuerzan la conversión del número a una cadena. Una cadena de texto en JavaScript tiene una propiedad asociada con ella que es la longitud (*length*), la cual te devolverá en este caso el número 4, indicando que hay 4 caracteres en esa cadena "3400". La longitud de una cadena es un número, no una cadena.

```
console.log(" " + ("3400").length);// devuelve "4"
console.log(typeof(" " + ("3400").length)); // devuelve string
```

También podemos cambiar la base de un número utilizando la función **toString()**.

```
var iNum1 = 10;
console.log(iNum1.toString(2));        //En Binario : 1010
console.log(iNum1.toString(8));        //En Octal : 12
console.log(iNum1.toString(16));//En Hexadecimal : a
console.log(typeof(iNum1.toString(16))); //devuelve string
```

1.2.5.2 Convertir a números

Para convertir primitivos no numéricos a números utilizamos **parseInt()** y **parseFloat()**.

- Utilización de **parseInt()**.

```
var iNum1 = parseInt("1234blue")        //devuelve 1234
var iNum2 = parseInt("0xA")        //devuelve 10
var iNum3 = parseInt("22.5")        //devuelve 22
var iNum4 = parseInt("blue")        //devuelve NaN
salida = iNum1 + "\n" + iNum2 + "\n" + iNum3 + "\n" + iNum4
console.log(salida)
```

- Utilización de **parseFloat()**.

```
var fNum1 = parseFloat("1234blue")        //devuelve 1234.0
var fNum2 = parseFloat("0xA")        //devuelve 0
var fNum3 = parseFloat("22.5")        //devuelve 22.5
var fNum4 = parseFloat("22.34.5")        //devuelve 22.34
var fNum5 = parseFloat("0908")        //devuelve 908
var fNum6 = parseFloat("blue")        //devuelve NaN
salida = fNum1 + "\n" + fNum2 + "\n" + fNum3 + "\n"
salida += fNum4 + "\n" + fNum5 + "\n" + fNum6
console.log(salida)
```

1.2.5.3 Funciones específicas para convertir tipos

La conversión de tipos nos permite acceder a un valor específico como si fuera de un tipo diferente. La conversión de un valor con una de estas tres funciones crea un uevo valor que es una conversión directa del original:

- **Boolean(valor)** : Convierte en Booleano el valor indicado.

```
var fNum1 = Boolean("algo")    //true
var fNum2 = Boolean(1)         //true
var fNum3 = Boolean(0)         //false
var fNum4 = Boolean(true)      //true
var fNum5 = Boolean(false)     //false
var fNum6 = Boolean(NaN)       //false
var fNum7 = Boolean(undefined) //false
var fNum8 = Boolean(null)      //false
salida = fNum1 + "\n" + fNum2 + "\n" + fNum3 + "\n"
salida += fNum4 + "\n" + fNum5 + "\n" + fNum6 + "\n"
salida += fNum7 + "\n" + fNum8
console.log(salida)
```

- **Number(valor)** : Convierte el valor indicado en un número (entero o de coma flotante).

```
var fNum1 = Number("1234blue")    //NaN
var fNum2 = Number("0xA")        //10
var fNum3 = Number("22.5")       //22.5
var fNum4 = Number("22.34.5")    //NaN
var fNum5 = Number("0908")      //908
var fNum6 = Number("blue")      //NaN
salida = fNum1 + "\n" + fNum2 + "\n" + fNum3 + "\n"
salida += fNum4 + "\n" + fNum5 + "\n" + fNum6
console.log(salida)
```

- **String(valor)** : Convierte el valor indicado en una cadena.

```
let fNum1 = String("algo")    //true
let fNum2 = String(1)         //true
let fNum3 = String(0)         //false
let fNum4 = String(true)      //true
let fNum5 = String(false)     //false
let fNum6 = String(NaN)       //false
let fNum7 = String(undefined) //false
let fNum8 = String(null)      //false
salida = fNum1 + "\n" + fNum2 + "\n" + fNum3 + "\n"
salida += fNum4 + "\n" + fNum5 + "\n" + fNum6 + "\n"
```

```
salida += fNum7 + "\n" + fNum8
console.log(salida)
</script>
```

1.2.6 Operadores

JavaScript es un lenguaje rico en **operadores**: símbolos y palabras que realizan operaciones sobre uno o varios valores, para obtener un nuevo valor.

Cualquier valor sobre el cuál se realiza una acción (indicada por el **operador**), se denomina un **operando**. Una **expresión** puede contener un operando y un operador (denominado **operador unario**), como por ejemplo en **b++**, o bien dos operandos, separados por un operador (denominado operador binario), como por ejemplo en **a + b**.

1.2.6.1 Categorías de operadores en JavaScript

Categorías de operadores en JavaScript	
Tipo	**Qué realizan**
Comparación	Comparan los valores de 2 operandos, devolviendo un resultado de *true* o *false* (se usan extensivamente en sentencias condicionales como *if... else* y en instrucciones *loop*). **== != === !== > >= < <=**
Aritméticos	Unen dos operandos para producir un único valor que es el resultado de una operación aritmética u otra operación sobre ambos operandos. **+ - * / % ++ - - +valor -valor**
Asignación	Asigna el valor a la derecha de la expresión a la variable que está a la izquierda. **= += -= *= /= %= <<= >= >>= >>>= &= |= ? = []**
Boolean	Realizan operaciones booleanas aritméticas sobre uno o dos operandos booleanos. **&& || !**
Bit a Bit	Realizan operaciones aritméticas o de desplazamiento de columna en las representaciones binarias de dos operandos. **& | ? ? << >> >>>**
Objeto	Ayudan a los scripts a evaluar la herencia y capacidades de un objeto particular antes de que tengamos que invocar al objeto y sus propiedades o métodos. **. [] () delete in instanceOf new this**
Misceláneos	Operadores que tienen un comportamiento especial. **, ?: typeof void**

1.2.6.2 Operadores de comparación en JavaScript

Operadores de comparación en JavaScript			
Sintaxis	**Nombre**	**Tipos de operandos**	**Resultados**
==	Igualdad	Todos	Boolean
!=	Distinto	Todos	Boolean
===	Igualdad estricta	Todos	Boolean
!==	Desigualdad estricta	Todos	Boolean

Operadores de comparación en JavaScript			
>	Mayor que	Todos	Boolean
>=	Mayor o igual que	Todos	Boolean
<	Menor que	Todos	Boolean
<=	Menor o igual que	Todos	Boolean

En valores numéricos, los resultados serían los mismos que obtendríamos con cálculos algebraicos.

```
30  ==  30        //  true
30  ==  30.0        //  true
5  !=  8        //  true
9  >  13        //  false
7.29  <=  7.28        //  false
```

También podríamos comparar cadenas a este nivel.

```
"Marta"  ==  "Marta"        //  true
"Marta"  ==  "marta"        //  false
"Marta"  >  "marta"        //  false
"Mark"  <  "Marta"        //  true
```

Para poder comparar cadenas, JavaScript convierte cada carácter de la cadena de un **string**, en su correspondiente valor **ASCII**. Cada letra, comenzando con la primera del operando de la izquierda, se compara con su correspondiente letra en el operando de la derecha. Los valores **ASCII** de las letras mayúsculas, son más pequeños que sus correspondientes en minúscula, por esa razón "Marta" no es mayor que "marta". En JavaScript hay que tener muy en cuenta la sensibilidad a mayúsculas y minúsculas.

Si por ejemplo comparamos un número con su cadena correspondiente.

```
"123" == 123// true
```

JavaScript cuando realiza esta comparación, convierte la cadena en su número correspondiente y luego realiza la comparación. Como vimos, también se dispone de otra opción, que consiste en convertir mediante las funciones **parseInt()** o **parseFloat()** el operando correspondiente.

```
parseInt("123") == 123// true
```

> **Los operadores === y !== comparan tanto el dato como el tipo de dato. El operador *===* sólo devolverá true, cuando los dos operandos son del mismo tipo de datos (por ejemplo ambos son números) y tienen el mismo valor.**

Analizar con cuidado las siguientes "Igualdades" vs "Igualdad estricta"			
console.log(1 === 1);	True	console.log(1 == 1);	True
console.log(1 === "1");	false	console.log(1 == "1");	True
console.log(null === undefined);	false	console.log(null == undefined);	True
console.log(1 === true);	false	console.log(1 == true);	True
console.log("" === false);	false	console.log("" == false);	True
console.log('\n' === false);	false	console.log('\n' == false);	True
console.log("" === 0);	false	console.log("" == 0);	True
console.log('\n' === 0);	false	console.log('\n' == 0);	True

1.2.6.3 Operadores aritméticos en JavaScript

Operadores aritméticos en JavaScript			
Sintaxis	**Nombre**	**Tipos de Operando**	**Resultados**
+	Más	Integer, ?oat, string	Integer, ?oat, string
-	Menos	Integer, ?oat	Integer, ?oat
*****	Multiplicación	Integer, ?oat	Integer, ?oat
/	División	Integer, ?oat	Integer, ?oat
%	Módulo	Integer, ?oat	Integer, ?oat
++	Incremento	Integer, ?oat	Integer, ?oat
--	Decremento	Integer, ?oat	Integer, ?oat
+valor	Positivo	Integer, ?oat, string	Integer, ?oat
-valor	Negativo	Integer, ?oat, string	Integer, ?oat

Veamos algunos ejemplos:

```
var  a  =  10; //  Inicializamos  a  al valor  10
var  z  =  0; //  Inicializamos  z  al valor 0
z  =  a; //  a  es igual a 10,  por lo tanto  z  es igual a 10.
z  =  ++a; //  el valor de a se incrementa justo antes
//de ser asignado a z, por lo que a es 11 y z valdrá 11.
z  =  a++; // se asigna el valor de a (11) a z y
//luego se incrementa el valor de a (pasa a ser 12).
z  =  a++; // a vale 12 antes de la asignación,
//por lo que z es igual a 12;
//una vez hecha la asignación a valdrá 13.
```

Otros ejemplos:

```
var  x  =  2;
var  y  =  8;
var  z  =  -x;        //  z  es igual a -2,
//pero x sigue siendo igual a 2.
z  =  - (x  +  y);        // z  es igual a -10,
//x es igual a 2 e y es igual a 8.
z  =  -x  +  y;        // z  es igual a 6,
//pero x sigue siendo igual a 2 e y igual a 8.
```

1.2.6.4 Operadores de asignación en JavaScript

Operadores de asignación en JavaScript			
Sintaxis	**Nombre**	**Ejemplo**	**Significado**
=	Asignación	x = y	x = y
+=	Sumar un valor	x += y	x = x + y
+=	Substraer un valor	x -= y	x = x - y
***=**	Multiplicar un valor	x *= y	x = x * y
/=	Dividir un valor	x /= y	x = x / y
%=	Módulo de un valor	x %= y	x = x % y

Operadores de asignación en JavaScript			
<<=	Desplazar bits a la izquierda	x <<= y	x = x << y
>=	Desplazar bits a la derecha	x >= y	x = x > y
>>=	Desplazar bits a la derecha rellenando con 0	x >>= y	x = x >> y
>>>=	Desplazar bits a la derecha	x >>>= y	x = x >>> y
&=	Operación AND bit a bit	x &= y	x = x & y
=	Operación OR bit a bit	= y	y
? =	Operación XOR bit a bit	x ? = y	x = x ? y
[]=	Desestructurando asignaciones	[a,b]=[c,d]	a=c, b=d

1.2.6.5 Operadores booleanos

Debido a que parte de la programación tiene un gran componente de lógica, es por ello, que los operadores booleanos juegan un gran papel.

Los operadores booleanos te van a permitir evaluar expresiones, devolviendo como resultado **true** (verdadero) o **false** (falso).

Operadores booleanos en JavaScript			
Sintaxis	**Nombre**	**Operandos**	**Resultados**
&&	AND	Boolean	Boolean
ll	OR	Boolean	Boolean
!	NOT	Boolean	Boolean

Ejemplos:

```
!true // resultado =  false
!(10  >  5) // resultado  =  false
!(10  <  5) // resultado  =  true
!("gato"  ==  "pato") // resultado  =  true
5  >  1  &&  50  >  10 // resultado  =  true
5  >  1  &&  50  <  10 // resultado  =  false
5  <  1  &&  50  >  10 // resultado  =  false
5  <  1  &&  50  <  10 // resultado  =  false
```

Tabla de valores de verdad para el operador AND			
Operando Izquierdo	**Operador AND**	**Operando Derecho**	**Resultado**
True	&&	True	True
True	&&	False	False
False	&&	True	False
False	&&	False	False

Tabla de valores de verdad para el operador OR			
Operando Izquierdo	**Operador OR**	**Operando Derecho**	**Resultado**
True	ll	True	True
True	ll	False	True

Tabla de valores de verdad para el operador OR			
False	II	True	True
False	II	False	False

Ejemplos:

```
5  >  1  ||  50  >  10        //  resultado  =  true
5  >  1  ||  50  <  10        //  resultado  =  true
5  <  1  ||  50  >  10        //  resultado  =  true
5  <  1  ||  50  <  10        //  resultado  =  false
```

1.2.6.6 Operadores bit a bit

Para los programadores de *scripts*, las operaciones bit a bit suelen ser un tema avanzado. A menos que tú tengas que gestionar procesos externos en aplicaciones del lado del servidor, o la conexión con *applets* de Java, es raro que tengas que usar este tipo de operadores.

Los operandos numéricos, pueden aparecer en JavaScript en cualquiera de los tres formatos posibles (decimal, octal o hexadecimal). Tan pronto como el operador tenga un operando, su valor se convertirá a representación binaria (32 bits de longitud). Las tres primeras operaciones binarias bit a bit que podemos realizar son **AND**, **OR** y **XOR** y los resultados de comparar bit a bit serán:

- Bit a bit AND: 1 si ambos dígitos son 1.
- Bit a bit OR: 1 si cualquiera de los dos dígitos es 1.
- Bit a bit XOR: 1 si sólo un dígito es 1.

Tabla de operadores Bit a Bit en JavaScript			
Operador	**Nombre**	**Operando izquierdo**	**Operando derecho**
&	Desplazamiento AND	Valor integer	Valor integer
I	Desplazamiento OR	Valor integer	Valor integer
^	Desplazamiento XOR	Valor integer	Valor integer
?	Desplazamiento NOT	(Ninguno)	Valor integer
<<	Desplazamiento a la izquierda	Valor integer	Cantidad a desplazar
>>	Desplazamiento a la derecha	Valor integer	Cantidad a desplazar
>>>	Desplazamiento derecha rellenando con 0	Valor integer	Cantidad a desplazar

Por ejemplo:

```
4  <<  2  //  resultado =  16
```

La razón de este resultado es que el número decimal 4 en binario es 00000100. El operador << indica a JavaScript que desplace todos los dígitos dos lugares hacia la izquierda, dando como resultado en binario 00010000, que convertido a decimal te dará el valor 16.

1.2.6.7 Operadores de Objeto

El siguiente grupo de operadores se relaciona directamente con objetos y tipos de datos. La mayor parte de ellos fueron implementados a partir de las primeras versiones de JavaScript, por lo que puede haber algún tipo de incompatibilidad con navegadores antiguos.

. (punto)

El operador punto, indica que el objeto a su izquierda tiene o contiene el recurso a su derecha, como por ejemplo: **objeto.propiedad** y **objeto.método()**.

Ejemplo con un objeto nativo de JavaScript:

```
var s = new String('rafa');
var longitud = s.length;
var pos = s.indexOf("fa"); // resultado: pos = 2
```

[ ] (corchetes para enumerar miembros de un objeto).

Por ejemplo cuando creamos un array: **var a =["Santiago","Coruña", "Lugo"];**
Enumerar un elemento de un array: **a[1] = "Coruña";**
Enumerar una propiedad de un objeto: **a["color"]= "azul";**

Delete (para eliminar un elemento de una colección).

Por ejemplo si consideramos:

```
var oceanos = new Array("Atlantico", "Pacifico", "Indico");
```

Podríamos hacer:

```
delete oceanos[2];
// Esto eliminaría el tercer elemento del array ("Indico"),
//pero la longitud del array no cambiaría.
//Si intentamos referenciar esa posición oceanos[2]
//obtendríamos undefined.
```

In (para inspeccionar métodos o propiedades de un objeto).

El operando a la izquierda del operador, es una cadena referente a la propiedad o método (simplemente el nombre del método sin paréntesis); el operando a la derecha del operador, es el objeto que estamos inspeccionando. Si el objeto conoce la propiedad o método, la expresión devolverá true.
Ejemplo: **"write" in document**
o también: **"defaultView" in document**

instanceof (comprueba si un objeto es una instancia de un objeto nativo).

Ejemplo:

```
a = new Array(1,2,3);
a instanceof Array;// devolverá true.
```

new (para acceder a los constructores de objetos incorporados en el núcleo de JavaScript).

Ejemplo:

```
var hoy = new Date(); // creará el objeto hoy de tipo Date()
```

this (para hacer referencia al propio objeto en el que estamos localizados).

Ejemplo:

```
nombre.onchange = validateInput;
function validateInput(evt) {
  var valorDeInput = this.value;
  // Este this hace referencia al propio objeto nombre
}
```

1.2.6.8 Operadores misceláneos

El operador coma ,

Este operador, indica una serie de expresiones que van a ser evaluadas en secuencia, de izquierda a derecha. La mayor parte de las veces, este operador se usa para combinar múltiples declaraciones e inicializaciones de variables en una única línea.

Ejemplo: **var nombre, direccion, apellidos, edad;**

Otra situación en la que podemos usar este operador coma, es dentro de la expresión **loop**. En el siguiente ejemplo inicializamos dos variables de tipo contador, y las incrementamos en diferentes porcentajes. Cuando comienza el bucle, ambas variables se inicializan a 0 y a cada paso del bucle una de ellas se incrementa en 1, mientras que la otra se incrementa en 10.

```
for (var i=0, j=0 ; i < 125; i++, j+10) {
// más instrucciones aquí dentro
}
```

Nota: no confundir la coma, con el delimitador de parámetros ?;? en la instrucción **for**.

? : (operador condicional)

Este operador condicional es la forma reducida de la expresión **if ... else**.
La sintaxis formal para este operador condicional es:
condición ? expresión si se cumple la condición: expresión si no se cumple;

Si usamos esta expresión con un operador de asignación:
var = condicion ? expresión si se cumple la condición: expresión si no se cumple;

Ejemplo:

```
var a,b;
a = 3; b = 5;
var h = a > b ? a : b;        // a h se le asignará el valor 5;
```

Volver