CURSO DE PROGRAMACIÓN JAVA FULLSTACK

JavaScript

2. Colecciones. Funciones globales. Fechas.

Arturo Bernal Mayordomo

Edición: Noviembre 2019

Contenido

1. - Colecciones

1.1 *Arrays*

En JavaScript, los arrays son un tipo de objetos. Podemos crear un array con la instancia de un objeto de clase **Array**. Estos no tienen un tamaño fijo, por tanto, podemos inicializarlo con un tamaño y luego añadirle más elementos.

El constructor puede recibir 0 parámetros (array vacío), 1 número (el tamaño del array), o en cualquier otro caso, se creará un array con los elementos recibidos. Debemos tener en cuenta que en JavaScript un array puede contener al mismo tiempo diferentes tipos de datos: number, string, boolean, object, etc.

```
let  a = new Array(); // Crea un array vacío
a[0] = 13;
console.log(a.length); // Imprime 1
console.log(a[0]); // Imprime 13
console.log(a[1]); // Imprime undefined
```

Fíjate que cuando accedes a una posición del array que no ha sido definida, devuelve undefined. La longitud de un array depende de las posiciones que han sido asignadas. Vamos a ver un ejemplo de lo que ocurre cuando asignas una posición mayor que la longitud y que no es consecutiva al último valor asignado.

```
let  a = new Array(12); // Crea un array de tamaño 12
console.log(a.length); // Imprime 12
a[20] = "Hello";
console.log(a.length); // Ahora imprime 21 (0-20). Las posiciones 0-19 tendrán el valor undefined
```

Podemos reducir la longitud del array modificando directamente la propiedad de la longitud del array (**length)**. Si reducimos la longitud de un array, las posiciones mayores a la nueva longitud serán consideradas como undefined (borradas).

```
let  a = new Array("a", "b", "c", "d", "e"); // Array con 5 valores
console.log(a[3]); // Imprime "d"
a.length = 2; // Posiciones 2-4 serán destruidas
console.log(a[3]); // Imprime undefined
```

Puedes crear un array usando corchetes en lugar de usar **new Array()**. Los elementos que pongamos dentro, separados por coma serán los elementos que inicialmente tendrá el array.

```
let  a = ["a", "b", "c", "d", "e"]; // Array de tamaño 5, con 5 valores inicialmente
console.log(typeof a); // Imprime object
console.log(a instanceof Array); // Imprime true. a es una instancia de array
a[a.length] = "f"; // Insertamos in nuevo elemento al final
console.log(a); // Imprime ["a", "b", "c", "d", "e", "f"]
```

1.2 *Recorriendo arrays*

Podemos recorrer un array con los clásicos bucles while y for, creando un contador para el índice que iremos incrementando en cada iteración. Otra versión del for, es el bucle **for..in**. Con este bucle podemos iterar los índices de un array o las propiedades de un objeto (similar al bucle foreach de otros lenguajes, pero recorriendo los índices en lugar de los valores).

```
let  ar = new Array(4, 21, 33, 24, 8);

let i = 0;
while(i < ar.length) { // Imprime 4 21 33 24 8
    console.log(ar[i]);
    i++;
}

for(let i = 0; i < ar.length; i++) { // Imprime 4 21 33 24 8
    console.log(ar[i]);
}
```

```
for (let i in ar) { // Imprime 4 21 33 24 8
    console.log(ar[i]);
}
```

Iterando las propiedades de un objeto (veremos objetos en el bloque 3):

```
let person = {
    nombre : "John",
    edad  : 45,
    telefono: "65-453565"
};

/**
 * Imprimirá:
 * nombre: John
 * edad: 45
 * telefono: 65-453565
 */
for (let field in person) {
    console.log(field + ": " + person[field]);
}
```

Desde ES2015 podemos iterar por los elementos de un array o incluso por los caracteres de una cadena sin utilizar el índice. Para ello se utiliza el bucle **for..of** (que se comporta como un bucle **foreach** de otros lenguajes).

```
let a = ["Item1", "Item2", "Item3", "Item4"];

for(let index in a) {
    console.log(a[index]);
}

for(let item of a) { // Hace lo mismo que el bucle anterior
    console.log(item);
}

let str = "abcdefg";

for(let letter of str) {
    if(letter.match(/^[aeiou]$/)) {
        console.log(letter + " es una vocal");
    } else {
        console.log(letter + " es una consonante");
    }
}
```

1.3 *Métodos de arrays*

Insertar valores al principio de un array (**unshift**) y al final (**push**).

```
let  a = [];
a.push("a"); // Inserta el valor al final del array
a.push("b", "c", "d"); // Inserta estos nuevos valores al final
console.log(a); // Imprime ["a", "b", "c", "d"]
a.unshift("A", "B", "C"); // Inserta nuevos valores al principio del array
console.log(a); // Imprime ["A", "B", "C", "a", "b", "c", "d"]
```

Ahora, vamos a ver la operación opuesta. Eliminar del principio (**shift**) y del final (**pop**) del array. Estas operaciones nos devolverán el valor que ha sido eliminado.

```
console.log(a.pop()); // Imprime y elimina la última posición → "d"
console.log(a.shift()); // Imprime y elimina la primera posición → "A"
console.log(a); // Imprime ["B", "C", "a", "b", "c"]
```

Podemos convertir un array a string usando **join()** en lugar de **toString()**. Por defecto, devuelve un string con todos los elementos separados por coma. Sin embargo, podemos especificar el separador a imprimir.

```
let  a = [3, 21, 15, 61, 9];
console.log(a.join()); // Imprime "3,21,15,61,9""
console.log(a.join(" -#- ")); // Imprime "3 -#- 21 -#- 15 -#- 61 -#- 9"
```

¿Cómo concatenamos dos arrays?. Usando **concat**.

```
let  a = ["a", "b", "c"];
let  b = ["d", "e", "f"];
let  c = a.concat(b);
console.log(c); // Imprime ["a", "b", "c", "d", "e", "f"]
console.log(a); // Imprime ["a", "b", "c"] . El array original no ha sido modificado
```

El método **slice** nos devuelve un nuevo *sub-array*.

```
let a = ["a", "b", "c", "d", "e", "f"];
let b = a.slice(1, 3); // (posición de inicio → incluida, posición final → excluida)
console.log(b); // Imprime ["b", "c"]
console.log(a); // Imprime ["a", "b", "c", "d", "e", "f"]. El array original no es modificado
console.log(a.slice(3)); // Un parámetro. Devuelve desde la posición 3 al final → ["d", "e", "f"]
```

splice elimina elementos del array original y devuelve los elementos eliminados. También permite insertar nuevos valores.

```
let  a = ["a", "b", "c", "d", "e", "f"];
a.splice(1, 3); // Elimina 3 elementos desde la posición 1 ("b", "c", "d")
console.log(a); // Imprime ["a", "e", "f"]
a.splice(1,1, "g", "h"); // Elimina 1 elemento en la posición 1 ("e"), e inserta "g", "h" en esa posición
console.log(a); // Imprime ["a", "g", "h", "f"]
a.splice(3, 0, "i"); // En la posición 3, no elimina nada, e inserta "i"
console.log(a); // Imprime ["a", "g", "h", "i", "f"]
```

Podemos invertir el orden del array usando el método **reverse**.

```
let  a = ["a", "b", "c", "d", "e", "f"];
a.reverse(); // Hace el reverse del array original
console.log(a); // Imprime ["f", "e", "d", "c", "b", "a"]
```

También, podemos ordenar los elementos de un array usando el método **sort**.

```
let  a = ["Peter", "Anne", "Thomas", "Jen", "Rob", "Alison"];
a.sort(); // Ordena el array original
console.log(a); // Imprime ["Alison", "Anne", "Jen", "Peter", "Rob", "Thomas"]
```

Pero, ¿Qué ocurre si intentamos ordenar elementos que no son string?. Por defecto, lo ordenará por su valor como string (teniendo en cuenta que si son objetos, se intentará llamar al método **toString()** para ordenarlo). Para ello, tendremos que pasar una función (de ordenación), que comparará 2 valores del array y devolverá un valor numérico indicando cual es menor (negativo si el primero es menor, 0 si son iguales y positivo si el primero es mayor).

```
let a = [20, 6, 100, 51, 28, 9];
a.sort(); // Ordena el array original
console.log(a); // Imprime [100, 20, 28, 51, 6, 9]
a.sort((n1, n2) => n1 - n2);
console.log(a); // Imprime [6, 9, 20, 28, 51, 100]
```

Veamos un ejemplo con objetos (los veremos en el bloque 3), en este caso son personas y las ordenaremos según la edad.

```
function Persona(nombre, edad) { // Constructor de la clase persona
   this.nombre = name;
   this.edad = edad;
```

```
    this.toString = function() { // Método toString()
        return this.nombre + " (" + this.edad + ")";
    }
}

let  personas = [];
personas[0] = new Persona("Thomas", 24);
personas[1] = new Persona("Mary", 15);
personas[2] = new Persona("John", 51);
personas[3] = new Persona("Philippa", 9);

personas.sort((p1, p2) => p1.age - p2.age));
console.log(personas.toString()); // Imprime: "Philippa (9),Mary (15),Thomas (24),John (51)"
```

Usando **indexOf**, podemos conocer si el valor que le pasamos se encuentra en el array o no. Si lo encuentra nos devuelve la primera posición donde está, y si no, nos devuelve -1. Usando el método **lastIndexOf** nos devuelve la primera ocurrencia encontrada empezando desde el final.

```
let a = [3, 21, 15, 61, 9, 15];
console.log(a.indexOf(15)); // Imprime 2
console.log(a.indexOf(56)); // Imprime -1. No encontrado
console.log(a.lastIndexOf(15)); // Imprime 5
```

El método **every** devolverá un boolean indicando si todos los elementos del array cumplen cierta condición. Esta función recibirá cualquier elemento, lo testeará, y devolverá cierto o falso dependiendo de si cumple la condición o no.

```
let a = [3, 21, 15, 61, 9, 54];
console.log(a.every(num =>  num < 100));  // Comprueba si cada número es menor a 100. Imprime true
console.log(a.every(num => num % 2 == 0)); // Comprueba si cada número es par. Imprime false
```

Por otro lado, el método **some** es similar a **every**, pero devuelve cierto en el momento en el que **uno de los elementos** del array cumple la condición.

```
let a = [3, 21, 15, 61, 9, 54];
console.log(a.some(num => num % 2 == 0));  // Comprueba si algún elemento del array es par. Imprime true
```

Podemos iterar por los elementos de un array usando el método **forEach.** De forma opcional, podemos llevar un seguimiento del índice al que está accediendo en cada momento, e incluso recibir el array como tercer parámetro.

```
let a = [3, 21, 15, 61, 9, 54];
let sum = 0;
a.forEach(num => sum += num);
console.log(sum); // Imprime 163

a.forEach((num, indice, array) => { // índice y array son parámetros opcionales
    console.log("Índice " + indice + " en [" + array + "] es " + num);
}); // Imprime -> Índice 0 en [3,21,15,61,9,54] es 3, Índice 1 en [3,21,15,61,9,54] es 21, ...
```

Para modificar todos los elementos de un array, el método **map** recibe una función que transforma cada elemento y lo devuelve. Este método devolverá al final un nuevo array del mismo tamaño conteniendo todos los elementos transformados.

```
let  a = [4, 21, 33, 12, 9, 54];
console.log(a.map(num => num*2)); // Imprime [8, 42, 66, 24, 18, 108]
```

Para filtrar los elementos de un array, y obtener como resultado un array que contenga sólo los elementos que cumplan cierta condición, usamos el método **filter**.

```
let  a = [4, 21, 33, 12, 9, 54];
console.log(a.filter(num => num % 2 == 0));  // Imprime [4, 12, 54]
```

El método **reduce** usa una función que acumula un valor, procesando cada elemento (segundo parámetro) con el valor acumulado (primer parámetro). Como segundo parámetro de reduce, deberías pasar un valor inicial. Si no pasas un valor inicial, el primer elemento de un array será usado como tal (si el array está vacío devolvería undefined).

```
let  a = [4, 21, 33, 12, 9, 54];
console.log(a.reduce((total, num) => total + num, 0));  // Suma todos los elementos del array. Imprime 133
console.log(a.reduce((max, num) => num > max? num : max, 0));  // Número máximo del array. Imprime 54
```

Para hacer lo mismo que **reduce** hace pero al revés, usaremos **reduceRight**.

```
let  a = [4, 21, 33, 12, 9, 154];
// Comienza con el último número y resta todos los otros números
console.log(a.reduceRight((total, num)  => total - num));
// Imprime 75 (Si no queremos enviarle un valor inicial, empezará con el valor de la última posición del array
```

1.4 *Extensiones de Array en ES2015*

Estos son los nuevos métodos que tiene el objeto global Array:

- **Array.of(value)** → Si queremos instanciar un array con un sólo valor, y éste es un número, con new Array() no podemos hacerlo, ya que crea vacío con ese número de posiciones.

```
let array = new Array(10); // Array vacío (longitud 10)
let array = Array(10); // Mismo que arriba: array vacío ( longitud 10)
let array = Array.of(10); // Array con longitud 1 -> [10]
let array = [10]; // // Array con longitud 1 -> [10]
```

- **Array.from(array, func)** → Funciona de forma similar al método **map**, crea un array desde otro array. Se aplica una operación de transformación (función lambda o anónima) para cada ítem.

```
let array = [4, 5, 12, 21, 33];
let array2 = Array.from(array, n => n * 2);
console.log(array2); // [8, 10, 24, 42, 66]
let array3 = array.map(n => n * 2); // Igual que Array.from
console.log(array3); // [8, 10, 24, 42, 66]
```

- **Array.fill(value)** → Este método sobrescribe todas las posiciones de un array con un nuevo valor. Es una buena opción para inicializar un array que se ha creado con N posiciones.

```
let sums = new Array(6); // Array con 6 posiciones
sums.fill(0); // Todas las posiciones se inicializan a 0
console.log(sums); // [0, 0, 0, 0, 0, 0]

let numbers = [2, 4, 6, 9];
numbers.fill(10); // Inicializamos las posiciones  al valor 10
console.log(numbers); // [10, 10, 10, 10]
```

- **Array.fill(value, start, end)** → Este método hace lo mismo que antes pero rellenando el array desde una posición inicial (incluida) hasta una final (excluida). Si no se especifica la última posición, se rellenará hasta el final.

```
let numbers = [2, 4, 6, 9, 14, 16];
numbers.fill(10, 2, 5); // Las posiciones 2,3,4 se ponen a 10
console.log(numbers); // [2, 4, 10, 10, 10, 16]

let numbers2 = [2, 4, 6, 9, 14, 16];
numbers2.fill(10, -2); // Las dos últimas posiciones se ponen a 10
```

```
console.log(numbers2); // [2, 4, 6, 9, 10, 10]
```

- **Array.find(condition)** → Encuentra y devuelve el primer valor que encuentre que cumple la condición que se establece. Con **findIndex**, devolvemos la posición que ocupa ese valor en el array.

```
let numbers = [2, 4, 6, 9, 14, 16];
console.log(numbers.find(num => num >= 10)); // Imprime 14 (primer valor encontrado >= 10)
console.log(numbers.findIndex(num => num >= 10)); // Imprime 4 (numbers[4] -> 14)
```

- **Array.copyWithin(target, startwith)** → Copia los valores del array empezando desde la posición **startWith**, hasta la posición **target** en el resto de posiciones del array (en orden). Por ejemplo:

```
let numbers = [2, 4, 6, 9, 14, 16];
numbers.copyWithin(3, 0); // [0] -> [3], [1] -> [4], [2] -> [5]
console.log(numbers); // [2, 4, 6, 2, 4, 6]
```

1.5 *Rest y spread*

Rest es la acción de transformar un grupo de parámetros en un array, y **spread** es justo lo opuesto, extraer los elementos de un array (o de un string) a variables.

Para usar **rest** en los parámetros de una función, se declara siempre como último parámetro (**1 máximo**) y se le ponen tres puntos'...' delante del mismo. Este parámetro se transformará automáticamente en un array conteniendo todos los parámetros restantes que se le pasan a la función. Si por ejemplo, el parámetro ***rest*** está en la tercera posición, contendrá todos los parámetros que se le pasen a excepción del primero y del segundo (a partir del tercero).

```
function getMedia(...notas) {
   console.log(notas); // Imprime [5, 7, 8.5, 6.75, 9] (está en un array)
   let total = notas.reduce((total,notas) => total + notas, 0);
   return total / notas.length;
}
console.log(getMedia(5, 7, 8.5, 6.75, 9)); // Imprime 7.25

function imprimirUsuario(nombre, ...lenguajes) {
   console.log(nombre + " sabe " + lenguajes.length + " lenguajes: " + lenguajes.join(" - "));
}

// Imprime "Pedro sabe 3 lenguajes: Java - C# - Python"
printUserInfo("Pedro", "Java", "C#", "Python");
// Imprime "María sabe 5 lenguajes: JavaScript - Angular - PHP - HTML - CSS"
printUserInfo("María", "JavaScript", "Angular", "PHP", "HTML", "CSS");
```

Spread es lo "opuesto" de **rest**. Si tenemos una variable que contiene un array, y ponemos los tres puntos '...' delante de este, extraerá todos sus valores. Podemos usar la propiedad por ejemplo con el método **Math.max,** el cual recibe un número indeterminado de parámetros y devuelve el mayor de todos.

```
let nums = [12, 32, 6, 8, 23];
console.log(Math.max(nums)); // Imprime NaN (array no es válido)
console.log(Math.max(...nums)); // Imprime 32 -> equivalente a Math.max(12, 32, 6 ,8 ,23)
```

Podemos usar también esta propiedad si necesitamos clonar un array.

```
let a = [1, 2, 3, 4];
let b = a; // Referencia el mismo array que 'a' (las modificaciones afectan a ambos).
let c = [...a]; // Nuevo array -> contiene [1, 2, 3, 4]
```

1.6 *Desestructuración de arrays*

Desestructuración es la acción de extraer elementos individuales de un array (o propiedades de un objeto) directamente en variables individuales. Podemos también *desestructurar* un string en caracteres.

Vamos a ver un ejemplo donde asignamos los tres primeros elementos de un array, en tres variables diferentes, usando una única asignación.

```
let array = [150, 400, 780, 1500, 200];
let [first, second, third] = array; // Asigna los tres primeros elementos del array
console.log(third); // Imprime 780
```

¿Qué pasa si queremos saltarnos algún valor? Se deja vacío (sin nombre) dentro de los corchetes y no será asignado:

```
let array = [150, 400, 780, 1500, 200];
let [first, , third] = array; // Asigna el primer y tercer elemento
console.log(third); // Imprime 780
```

Podemos asignar el resto del array a la última variable que pongamos entre corchetes usando **rest** (como en el punto anterior del tema):

```
let array = [150, 400, 780, 1500, 200];
let [first, second, ...rest] = array; // rest -> array
console.log(rest); // Imprime [780, 1500 ,200]
```

Si queremos asignar más valores de los que contiene el array y no queremos obtener undefined, podemos usar valores por defecto:

```
let array = ["Peter", "John"];
let [first, second = "Mary", third = "Ann"] = array; // rest -> array
console.log(second); // Imprime "John"
console.log(third); // Imprime "Ann" -> valor por defecto
```

También podemos desestructurar **arrays anidados**:

```
let sueldos = [["Pedro", "Maria"], [24000, 35400]];
let [[nombre1, nombre2], [sueldo1, sueldo2]] = sueldos;
console.log(nombre1 + " gana " + sueldo1 + "€"); // Imprime "Pedro gana 24000€"
```

También se puede desestructurar un array enviado como parámetro a una función en valores individuales:

```
function imprimirUsuario([id, nombre, email], otraInfo = "Nada") {
   console.log("ID: " + id);
   console.log("Nombre: " + nombre);
   console.log("Email: " + email );
   console.log("Otra info: " + otraInfo );
}

let infoUsu = [3, "Pedro", "peter@gmail.com"];
imprimirUsuario(infoUsu, "No es muy listo");
```

1.7 *Map*

Un Map es una colección que guarda parejas de [clave,valor], los valores son accedidos usando la correspondiente clave. En JavaScript, un objeto puede ser considerado como un tipo de Mapa pero con algunas limitaciones (Sólo con strings y enteros como claves).

```
let obj = {
   0: "Hello",
   1: "World",
   prop1: "This is",
   prop2: "an object"
}

console.log(obj[1]); // Imprime "World"
console.log(obj["prop1"]); // Imprime "This is"
console.log(obj.prop2); // Imprime "an object"
```

La nueva colección **Map** permite usar cualquier objeto como clave. Creamos la colección usando el constructor **new Map()**, y podemos usar los métodos **set**, **get** y **delete** para almacenar, obtener o eliminar un valor basado en una clave.

```
let person1 = {name: "Peter", age: 21};
let person2 = {name: "Mary", age: 34};
let person3 = {name: "Louise", age: 17};

let hobbies = new Map(); // Almacenará una persona con un array de hobbies (string)
hobbies.set(person1, ["Tennis", "Computers", "Movies"]);
console.log(hobbies); // Map {Object {name: "Peter", age: 21} => ["Tennis", "Computers",
"Movies"]}

hobbies.set(person2, ["Music", "Walking"]);
hobbies.set(person3, ["Boxing", "Eating", "Sleeping"]);
console.log(hobbies);
```

```
Map {Object {name: "Peter", age: 21} => ["Tennis", "Computers", "Movies"], Object
▼{name: "Mary", age: 34} => ["Music", "Walking"], Object {name: "Louise", age: 17}
 => ["Boxing", "Eating", "Sleeping"]} i
   size: (...)
  ▶ __proto__: Map
  ▼ [[Entries]]: Array[3]
    ▼ 0: {Object => Array[3]}
      ▼ key: Object
          age: 21
          name: "Peter"
        ▶ __proto__: Object
      ▼ value: Array[3]
          0: "Tennis"
          1: "Computers"
          2: "Movies"
          length: 3
```

Cuando usamos un objeto como clave, debemos saber que almacenamos una **referencia a ese objeto** (Luego veremos WeakMap). Por tanto, se debe **usar la misma referencia** para acceder a un valor que para eliminarlo en ese mapa.

```
console.log(hobbies.has(person1)); // true (referencia al objeto original almacenado)
console.log(hobbies.has({name: "Peter", age: 21})); // false (mismas propiedades pero objeto
diferente!)
```

La propiedad **size** devuelve la longitud del mapa y podemos iterar a través por él usando [Symbol.iterator] o el bucle **for..of**. Para cada iteración, se devuelve un array con dos posiciones **0 → key** y **1 → value**.

```
console.log(hobbies.size); // Imprime 3
hobbies.delete(person2); // Elimina person2 del Map
console.log(hobbies.size); // Imprime 2
console.log(hobbies.get(person3)[2]); // Imprime "Sleeping"

/** Imprime todo:
 * Peter: Tennis, Computers, Movies
 * Louise: Boxing, Eating, Sleeping */
for(let entry of hobbies) {
    console.log(entry[0].name + ": " + entry[1].join(", "));
}
for(let [person, hobArray] of hobbies) { // Mejor
    console.log(person.name + ": " + hobArray.join(", "));
}

hobbies.forEach((hobArray, person) => { // Mejor
    console.log(person.name + ": " + hobArray.join(", "));
```

```
});
```

Si tenemos un tenemos un array que contiene otros arrays con dos posiciones (key, value), podemos crear un mapa directamente a partir del mismo.

```
let prods = [
    ["Computer", 345],
    ["Television", 299],
    ["Table", 65]
];

let prodMap = new Map(prods);
console.log(prodMap); // Map {"Computer" => 345, "Television" => 299, "Table" => 65}
```

1.8 *Set*

Set es como **Map**, pero no almacena los valores (sólo la clave). Puede ser visto como una colección que **no permite valores duplicados** (en un array puede haber valores duplicados). Se usa, **add**, **delete** y **has** → son métodos que devuelven un booleano para almacenar, borrar y ver si existe un valor.

```
let set = new Set();
set.add("John");
set.add("Mary");
set.add("Peter");
set.delete("Peter");
console.log(set.size); // Imprime 2

set.add("Mary"); // Mary ya existe
console.log(set.size); // Imprime 2

// Itera a través de los valores
set.forEach(value => {
    console.log(value);
})
```

Podemos crear un Set desde un array (lo cual elimina los valores duplicados).

```
let names = ["Jennifer", "Alex", "Tony", "Johny", "Alex", "Tony", "Alex"];
let nameSet = new Set(names);
console.log(nameSet); // Set {"Jennifer", "Alex", "Tony", "Johny"}
```

2. - Objetos y funciones globales

JavaScript tiene algunas funciones y objetos globales, las cuales pueden ser accedidas desde cualquier sitio. Estas funciones y objetos son bastante útiles para trabajar con números, cadenas, etc.

2.1 *Funciones globales*

- **parseInt(value)** → Transforma cualquier valor en un entero. Devuelve el valor entero, o NaN si no puede ser convertido.
- **parseFloat(value)** → Igual que parseInt, pero devuelve un decimal.
- **IsNaN(value)** → Devuelve true si el valor es NaN.
- **isFinite(value)** → Devuelve true si el valor es un número finito o false si es infinito.
- **Number(value)** → Transforma un valor en un número (o NaN).

- **String(value)** → Convierte un valor en un string (en objetos llama a toString()).
- **encodeURI(string), decodeURI(string)** → Transforma una cadena en una URL codificada, codificando caracteres especiales a excepción de: , / ? : @ & = + $ #. Usa decodeURI para transformarlo a string otra vez.
  - Decoded: "http://domain.com?val=1 2 3&val2=r+y%6"
  - Encoded: "http://domain.com?val=1%202%203&val2=r+y%256"
- **encodeURIComponent(string), decodeURIComponent(string)** → Estas funciones también codifican y decodifican los caracteres especiales que encodeURI no hace. Se deben usar para codificar elementos de una url como valores de parámetros (no la url entera).
  - Decoded: "http://domain.com?val=1 2 3&val2=r+y%6"
  - Encoded: "http%3A%2F%2Fdomain.com%3Fval%3D1%202%203%26val2%3Dr%2By%256"

2.2 *El objeto Math*

El objeto Math nos proporciona algunas constantes o métodos bastante útiles.

- **Constants** → E (Número de Euler), PI, LN2 (algoritmo natural en base 2), LN10, LOG2E (base-2 logaritmo de E), LOG10E, SQRT1_2 (raíz cuadrada de ½), SQRT2.
- **round(x)** → Redondea x al entero más cercano.
- **floor(x)** → Redondea x hacia abajo (5.99 → 5. Quita la parte decimal)
- **ceil(x)** → Redondea x hacia arriba (5.01 → 6)
- **min(x1,x2,...)** → Devuelve el número más bajo de los argumentos que se le pasan.
- **max(x1,x2,...)** → Devuelve el número más alto de los argumentos que se le pasan.
- **pow(x, y)** → Devuelve x^y (x elevado a y).
- **abs(x)** → Devuelve el valor absoluto de x.
- **random()** → Devuelve un número decimal aleatorio entre 0 y 1 (no incluidos).
- **cos(x)** → Devuelve el coseno de x (en radianes).
- **sin(x)** → Devuelve el seno de x.
- **tan(x)** → Devuelve la tangente de x.
- **sqrt(x)** → Devuelve la raíz cuadrada de x

```
console.log("Raíz cuadrada de 9: " + Math.sqrt(9));
console.log("El valor de PI es: " + Math.PI);
console.log(Math.round(4.546342));
// Número aleatorio entre 1 y 10
console.log(Math.floor(Math.random() * 10) + 1);
```

3. - Fechas

En Javascript tenemos la clase Date, que encapsula información sobre fechas y métodos para operar, permitiéndonos almacenar la fecha y hora local (timezone).

```
let date = new Date(); // Crea objeto Date almacena la fecha actual
console.log(typeof date); // Imprime object
console.log(date instanceof Date); // Imprime true
console.log(date); // Imprime (en el momento de ejecución) Fri Jun 24 2016 12:27:32 GMT+0200 (CEST)
```

Podemos enviarle al constructor el número de milisegundos desde el 1/171970 a las 00:00:00 GMT (Llamado **Epoch** o **UNIX time**). Si pasamos más de un número, (sólo el primero y el segundo son obligatorios),el orden debería ser: 1to → año, 2º → mes (0..11), 3º → día, 4º → hora, 5º → minuto, 6º → segundo. Otra opción es pasar un string que contenga la fecha en un formato válido.

```
let date  = new Date(1363754739620); // Nueva fecha 20/03/2013 05:45:39 (milisegundos desde Epoch)
let date2 = new Date(2015, 5, 17, 12, 30, 50); // 17/06/2015 12:30:50 (Mes empieza en 0 -> Ene, ... 11 -> Dic)
let date3 = new Date("2015-03-25"); // Formato de fecha largo sin la hora YYYY-MM-DD (00:00:00)
let date4 = new Date("2015-03-25T12:00:00"); // Formato fecha largo con la fecha
let date5 = new Date("03/25/2015"); // Formato corto MM/DD/YYYY
let date6 = new Date("25 Mar 2015");  // Formato corto con el mes en texto (March también sería válido).
let date7 = new Date("Wed Mar 25 2015 09:56:24 GMT+0100 (CET)"); // Formato completo con el timezone
```

Si, en lugar de un objeto Date, queremos directamente obtener los milisegundos que han pasado desde el 1/1/1970 (Epoch), lo que tenemos que hacer es usar los métodos **Date.parse**(string) y **Date.UTC**(año, mes, día, hora, minuto, segundos). También podemos usar **Date.now()**, para la fecha y hora actual en milisegundos.

```
let nowMs = Date.now(); // Momento actual en ms
let dateMs = Date.parse("25 Mar 2015"); // 25 Marzo 2015 en ms
let dateMs2 = Date.UTC(2015, 2, 25); // 25 Marzo 2015 en ms
```

La clase Date tiene **setters** y **getters** para las propiedades: **fullYear**, **month** (0-11), **date** (día), **hours**, **minutes**, **seconds**, y **milliseconds**. Si pasamos un valor negativo, por ejemplo, mes = -1, se establece el último mes (dic.) del año anterior.

```
// Crea un objeto fecha de hace 2 horas
let twoHoursAgo = new Date(Date.now() - (1000*60*60*2)); // (Ahora - 2 horas) en ms
// Ahora hacemos lo mismo,pero usando el método setHours
let now = new Date();
now.setHours(now.getHours() - 2);
```

Cuando queremos imprimir la fecha, tenemos métodos que nos la devuelven en diferentes formatos:

```
let now = new Date();

console.log(now.toString());
console.log(now.toISOString()); // Imprime 2016-06-26T18:00:31.246Z
console.log(now.toUTCString()); // Imprime Sun, 26 Jun 2016 18:02:48 GMT
console.log(now.toDateString()); // Imprime Sun Jun 26 2016
console.log(now.toLocaleDateString()); // Imprime 26/6/2016
console.log(now.toTimeString()); // Imprime 20:00:31 GMT+0200 (CEST)
console.log(now.toLocaleTimeString()); // Imprime 20:00:31
```