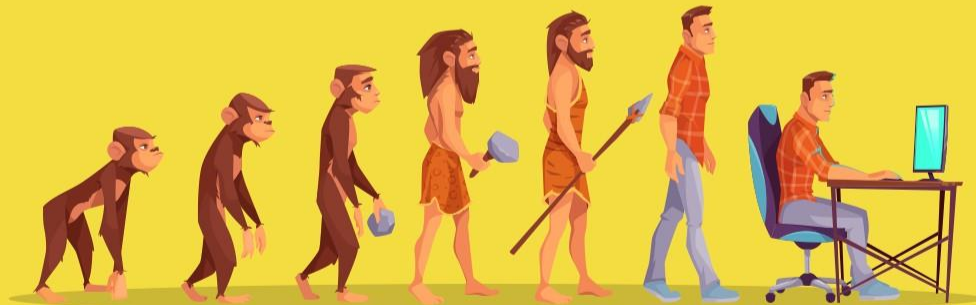


# JS



## UT03: OBJETOS Y TIPOS DE DATOS

# ÍNDICE

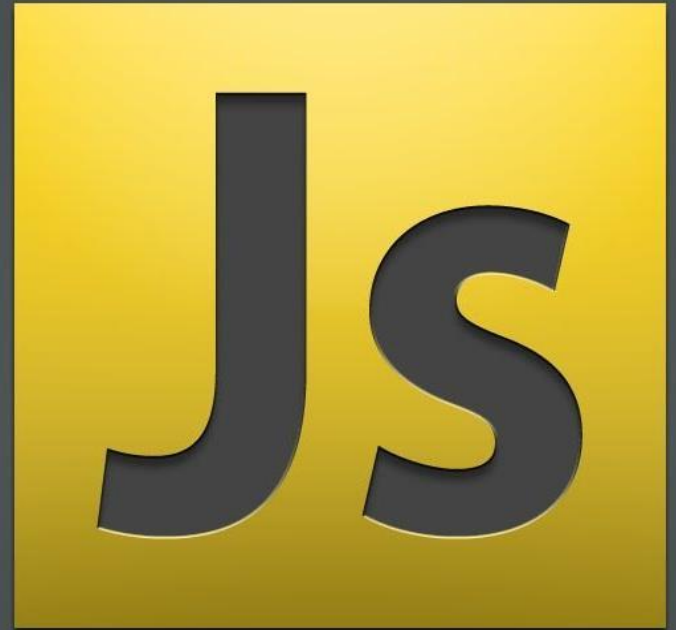
---

- 1.- Objetos en JavaScript
- 2.- Los operadores this y new
- 3.- Encadenamiento opcional
- 4.- Tipos de datos primitivos
- 5.- Arrays
- 6.- Métodos de arrays
- 7.- Map y Set
- 8.- Fecha y hora
- 9.- Métodos JSON



# 1

## OBJETOS EN JAVASCRIPT



Uno de los tipos de datos primitivos en JavaScript son los **objetos**. Es importante conocerlos porque casi cualquier aspecto del lenguaje tiene relación con los objetos.

En JavaScript un objeto es una secuencia de **pares clave/valor**.

```
let user = {  
  name: "Evaristo",  
  age: 25,  
};
```

La coma tras la última propiedad se llama **final** o **colgante** y es **opcional**, pero facilita agregar, eliminar o mover propiedades.

Se puede acceder a los valores de las propiedades de un objeto utilizando la notación punto:

```
console.log( user.name );  
user.age += 1;  
console.log(user.age);           // 21
```

Hay una notación alternativa que es indicando la propiedad entre corchetes.

Sirve para hacer referencia a propiedades compuestas por más de una palabra.

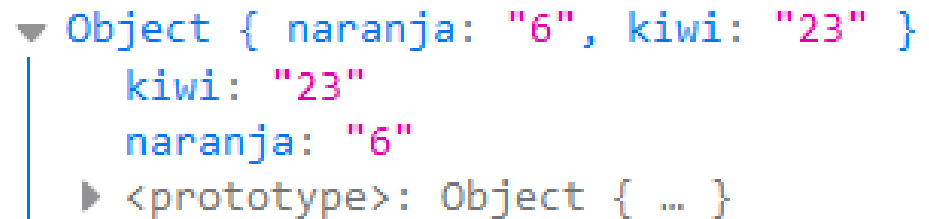
```
let profesores = {  
  "entorno cliente": "Victor"  
  "entorno servidor": "Oscar"  
}  
  
console.log( profesores["entorno cliente"] );    // Victor
```

Otra utilidad de los corchetes es cuando el nombre de la propiedad se genera a partir de una expresión.

```
let notas = {  
  dwec: 8,  
  diw: 7,  
  dwes: 7,  
}  
let modulo = prompt("¿De qué módulo quieres saber la  
nota?")  
alert(`Tienes un ${ notas[modulo] } en ${ modulo }`)
```

Los corchetes también se usan para las **propiedades calculadas**, en las que el nombre de la misma se calcula en tiempo de ejecución.

```
let fruits = {};  
let fruit, price;  
while (true) {  
    fruit = prompt('Introduce una fruta');  
    price = prompt('Indica el precio por kilo');  
    if (!fruit) break;  
    fruits[fruit] = price;  
};  
  
console.log(fruits);
```



```
▼ Object { naranja: "6", kiwi: "23" }  
  kiwi: "23"  
  naranja: "6"  
  ► <prototype>: Object { ... }
```



Cuando creamos un objeto a partir de variables cuyo nombre es igual al de las propiedades, se puede abreviar.

```
function doSomething( name, email ) {  
    let array = {  
        name,           // Equivale a name: name  
        email,          // Equivale a email: email  
    }  
}
```

Si se invoca a una propiedad de un objeto que no existe devuelve *undefined*

```
let user = {  
  name: 'Victor',  
  email: 'mail@mail.com',  
}  
  
console.log( user.age );           // undefined
```

Con el operador **in** se puede comprobar si un objeto tiene una determinada propiedad.

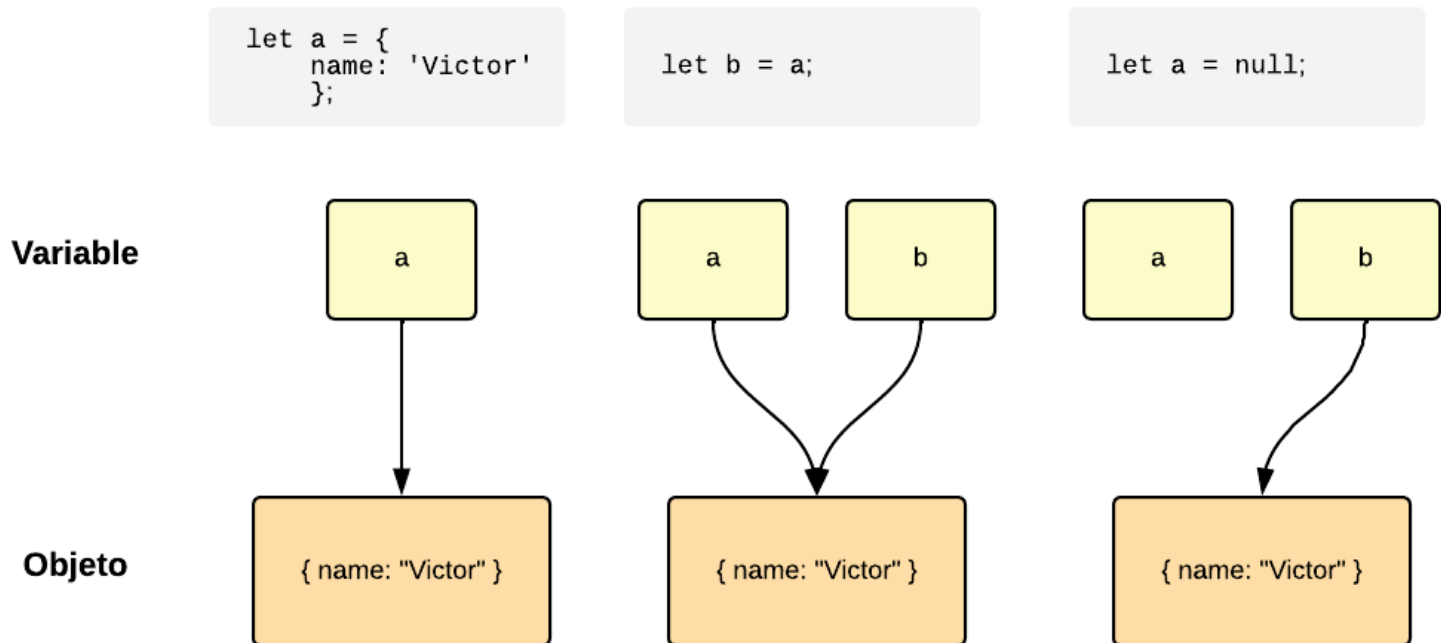
```
let user = {  
  name: 'Victor',  
  email: 'mail@mail.com',  
}  
let key = 'email';  
console.log( "age" in user );           // false  
console.log( "name" in user );          // true  
console.log( key in user );             // true
```

Observa que la clave debe ir entre comillas salvo que se desee indicar una variable que contiene el valor de la clave.

Es posible iterar sobre todos los pares clave-valor utilizando el bucle **for..in**

```
let user = {  
  name: 'Victor',  
  email: 'mail@mail.com',  
}  
  
for ( let key in user ) {  
  console.log( key );           // name, email  
  console.log( user[key] )     // Victor, mail@mail.com  
}
```

Una cuestión importante al tener en cuenta es que los objetos **se almacenan por referencia**.



Como se ve, el operador de **asignación** en objetos no copia realmente el objeto, sino que hace que **la nueva variable apunte al mismo objeto**.

```
let a = {  
  name: 'Victor',  
};  
let b = a;  
console.log( b.name );      // Victor  
b.name = 'Pepe';  
console.log( a.name );      // Pepe  
a.email = 'mail@mail.com';  
console.log( b.email );      // mail@mail.com
```

Esto implica que no podemos copiar objetos con el operador asignación.

Si necesitáramos clonar un objeto tenemos que recurrir a otros métodos:

- Crear una función personalizada
- Utilizar la función **Object.assign()**
- Utilizar el operador **spread**

## Función personalizada

Lo más básico es crear una función que itere sobre las propiedades del objeto y las añada a un segundo objeto.

Esto queda para que lo hagas como práctica



## Función `Object.assign()`

La sintaxis de esta función es:

```
Object.assign( dest, src1 [,src2, src3...] )
```

- El primer argumento es el objeto destino.
- Recoge todas las propiedades de los objetos fuente y las copia en el objeto destino.
- Devuelve el objeto destino.

## Ejemplo 1: fusión de objetos

```
let user = {  
  name: 'Victor',  
}  
let credentials = {  
  pass: '1234',  
}  
Object.assign( user, credentials );  
console.log(user);    // { name: 'Victor', pass: '1234' }
```

## Ejemplo 2: clonación de objetos

```
let user = {  
  name: 'Victor',  
}  
  
let clone = Object.assign( {}, user );  
console.log(clone);      // { name: 'Victor' }
```

## Operador spread

Este operador se puede aplicar a cualquier iterable (objetos, arrays, strings, ...).

Su función es descomponer el iterable en sus elementos individuales y devolverlos para insertarlos en un array o un objeto.

El operador solo se puede usar entre llaves (para el caso de los objetos) o corchetes (en el caso de los arrays).

## Ejemplo 1: combinar un objeto con otro

```
let user = {  
  name: 'Victor',  
  nick: 'vgr',  
}  
let accessUser = {  
  ...user,  
  pass: '1234',  
};  
  
console.log( accessUser ); // { name: 'Victor',  
                             // nick: 'vgr', pass: '1234' }
```

## Ejemplo 2: combinar dos objetos

```
let user = {  
  name: 'Victor',  
  nick: 'vjgr',  
}  
let credentials = {  
  pass: '1234',  
}  
let accessUser = {  
  ...user,  
  ...credentials,  
}  
  
console.log(accessUser);    // { name: 'Victor',  
                             // nick: 'vjgr', pass: '1234' }
```

### Ejemplo 3: expandir objeto para obtener lista de parámetros

```
let user = {  
  name: 'Victor',  
  nick: 'vjgr',  
}  
let credentials = {  
  pass: '1234',  
}  
let accessUser = {  
  ...user,  
  ...credentials,  
}  
  
console.log(accessUser); // { name: 'Victor',  
                          // nick: 'vjgr', pass: '1234' }
```

# 2

LOS  
OPERADORES  
THIS Y NEW





Los objetos que hemos visto hasta ahora tienen propiedades, pero también pueden tener métodos.

En JavaScript los métodos de los objetos vienen dados por **funciones en las propiedades**.

```
let user = {  
  userName: 'Victor',  
  saluda: function() {  
    console.log('Hola a todos!!!');  
  }  
}  
  
user.saluda();
```

Lo habitual es que los métodos hagan referencia a las propiedades del objeto, y, en ese caso, hemos de utilizar la palabra reservada **this**.

```
let user = {  
  userName: 'Victor',  
  surname: 'González',  
  getFullName: function() {  
    return( `${this.userName} ${this.surname}` )  
  }  
}  
  
console.log(user.getFullName());
```

El valor de **this** se evalúa en tiempo de ejecución, y será el objeto que contiene la función en la que se utiliza.

```
let user = {  
  userName: 'victor',  
  test: function() {  
    console.log(this)  
  }  
}  
  
user.test();
```

```
● PS C:\proyectos\javascript\exercises> node .\script.js  
  { userName: 'victor', test: [Function: test] }  
○ PS C:\proyectos\javascript\exercises> 
```

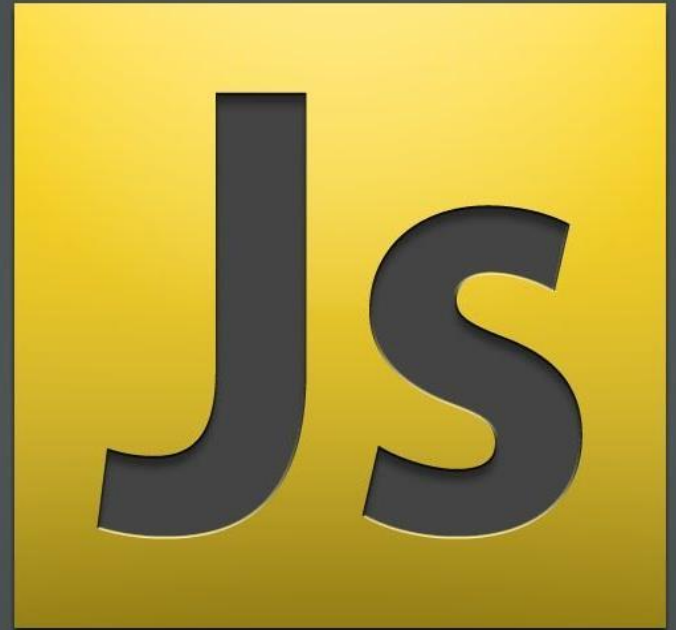
Técnicamente, sería posible reemplazar *this* y utilizar en su lugar el nombre de la variable, aunque no es nada recomendable hacerlo.

```
let user = {  
  userName: 'victor',  
  getUserName: () => user.userName // this.userName  
}  
  
console.log(user.getUserName());
```

Esto **no es nada recomendable** porque si copiamos el objeto a otra variable no funcionará.

# 3

ENCADENAMIENTO  
OPCIONAL



El encadenamiento opcional es una forma de **prevenir errores al acceder a propiedades anidadas de objetos**.

```
let user = {};  
  
console.log(user.address);           // undefined  
console.log(user.address.street);    // Error
```

Este error puede ocurrir cuando recogemos los datos de un fichero JSON (que puede estar incompleto) u obtenemos los datos del DOM que pueden estar vacíos.

Una forma de evitarlo sería la siguiente:

```
alert( user.address ? user.address.street : undefined );
```

Pero es poco elegante y además se complica a medida que hay mayor nivel de anidamiento.

```
console.log( user.address ?  
              user.address.street ?  
                user.address.stree.name  
                : undefined  
              : undefined );
```

La solución es el **encadenamiento opcional** `?.` que devuelve *undefined* si el valor antes del `?.` es *undefined* o *null*.

```
console.log( user.address?.street?.name );
```



El encadenamiento opcional no es un operador, es una construcción del lenguaje que también funciona con funciones y corchetes.

Por ejemplo, `?.`() se usa para llamar a una función que puede no existir.

```
let userAdmin = {  
  admin() {  
    alert('I am admin');  
  }  
};  
let userGuest = {};  
  
userAdmin.admin?();           // I am admin  
userGuest.admin?();           // No pasa nada
```

# 4

TIPOS DE DATOS  
PRIMITIVOS:  
NÚMEROS



Por definición, un tipo de datos primitivo difiere de un objeto en que contiene un único valor y que debe ser tan rápido y liviano como sea posible.

Por otro lado, sí que sería deseable que dispusieran de métodos para realizar fácilmente operaciones sobre ellos, como si fueran objetos.

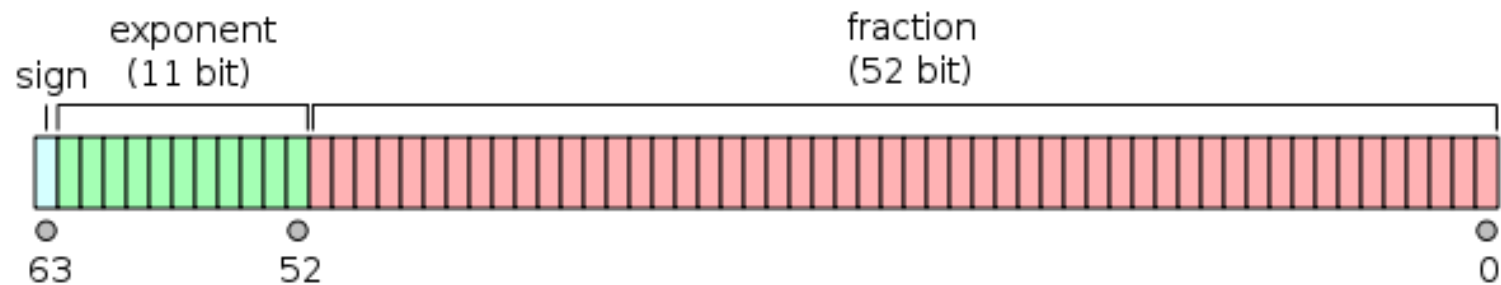
JavaScript tiene una forma peculiar de obtener las ventajas de ambos enfoques.

- Los primitivos son primitivos, con un único valor.
- El lenguaje permite el acceso a métodos y propiedades de strings, numbers, booleans y symbols.
- Para que esto funcione, se crea una envoltura especial, un *object wrapper* que provee la funcionalidad extra y luego es destruido.

Cada tipo primitivo tiene su *object wrapper*, que se llaman String, Number, Boolean, Symbol y BigInt.

En JavaScript hay dos tipos de números:

- Los números regulares que están almacenados en el formato IEEE-754 de 64 bits y son los que trataremos ahora.
- Los números BigInt que representan enteros de longitud arbitraria.



Hay varias formas de representar un número.

En el siguiente ejemplo se ven las tres formas de representar el valor 1.000.000

```
let a = 1000000;  
let b = 1_000_000;  
let c = 1e6;
```

También se aplica para números muy pequeños. Los siguientes dos valores son equivalentes

```
let a = 0.0001;  
let b = 1e-4;
```

También se pueden representar números en base hexadecimal, octal y binario precediéndolos de las cadenas `0x`, `0o` y `0b` respectivamente.

```
let a = 0xde;           // Hexadecimal
let b = 0o377;          // Octal
let c = 0b11001100;     // Binario
```

El método **toString(*base*)** permite convertir un número a cadena en la base indicada como parámetro (la base por defecto es 10).

La base puede estar entre 2 y 36.

```
let a = 12345;  
console.log(a.toString());           // '12345'  
console.log(a.toString(16));        // '3039'
```



Hay que tener cuidado con la sintaxis si queremos aplicar directamente el método sobre un número.

```
console.log(12..toString(2)); // '1100'
```

Es necesario poner dos puntos, ya que el primer punto que encuentre será el que indica la parte decimal del número.



## Hay cuatro funciones de redondeo en JavaScript

```
let a = 5.6;  
  
console.log( Math.floor(a) );    // 5  
console.log( Math.ceil(a) );    // 6  
console.log( Math.round(a) );   // 6  
console.log( Math.trunc(a) );   // 5
```

	Math.floor	Math.ceil	Math.round	Math.trunc
3.1	3	4	3	3
3.6	3	4	4	3
-1.1	-2	-1	-1	-1
-1.6	-2	-1	-2	-1

Si queremos redondear el número a **n** dígitos debemos utilizar el método **toFixed(n)**

```
let a = 5.64645;  
let b = 4.7;  
  
console.log( a.toFixed(2) );    // 5.65  
console.log( b.toFixed(4) );    // 4.7000
```

El hecho de almacenar los números en formato IEEE 754 implica una serie de problemas.

El primero es que si intentamos almacenar un número demasiado grande obtendremos *Infinity*.

```
>> console.log(1e500)
```

```
Infinity
```

Otro problema menos obvio es la **pérdida de precisión** cuando utilizamos números decimales (no enteros).

Si verificas la siguiente igualdad verás que el resultado es *false*.

```
console.log( 0.1 + 0.2 == 0.3 );           // false
```

Esto es porque el resultado de la suma no es exactamente 0.3

```
console.log( 0.1 + 0.2 );                   // 0.30000000000000004
```



**¿Por qué ocurre esto?** Porque cuando se almacena un número no entero en binario se hace como suma de fracciones de potencias de 2, y hay números que tienen una representación exacta en base 10 pero no en base 2.

Normalmente, cuando representamos un número se aplica un redondeo para que no percibamos esa pérdida de precisión, aunque sigue existiendo y se muestra especialmente cuando realizamos operaciones.

```
console.log( 0.1.toFixed(20)); // 0.100000000000000000555
```

## ¿Cómo solucionarlo?

Una solución es utilizar **toFixed()** para limitar el número de decimales. Hay que tener en cuenta que esta función devuelve una cadena, por lo que habría que convertir el resultado a número.

```
let sum = 0.1 + 0.2;  
console.log( +sum.toFixed(2) );           // 0.3
```

Otra posibilidad es **evitar el uso de decimales**, por ejemplo, en una tienda online podemos almacenar los valores como céntimos y no como euros.



La conversión a número con `Number()` y con el operador unario `+` es estricta, si la cadena no contiene un número devuelve *NaN*.

Hay ocasiones en que encontramos cadenas que no son exactamente un número, por ejemplo, `100px` o `25€`.

En ese caso podemos utilizar las funciones **`parseInt()`** y **`parseFloat()`**, que analizan la cadena hasta que encuentren algo que no sea un dígito y devuelve el número que hayan registrado hasta el momento.

La función **`parseInt()`** tiene un segundo parámetro opcional para indicar la base en la que se encuentra la cadena.

```
console.log( Number( '100px' ) );           // NaN
console.log( parseInt( '100px' ) );          // 100
console.log( parseInt( '125€' ) );           // 125
console.log( parseInt( 'px100' ) );          // NaN
console.log( parseFloat( '127.34$' ) );      // 127.34
console.log( parseInt( '1.2.3' ) );          // 1
console.log( parseFloat( '1.2.3' ) );        // 1.2
console.log( parseInt( 'ff', 16 ) );         // 255
console.log( parseInt( '0xff', 16 ) );       // 255
```

JavaScript incorpora el objeto **Math** que contiene una biblioteca de funciones matemáticas y constantes.

Algunas son:

- **Math.random()**: devuelve un número aleatorio entre 0 y 1
- **Math.max(a, b, c, ...)**: devuelve el mayor de los parámetros
- **Math.min(a, b, c, ...)**
- **Math.pow(n, k)**: devuelve n elevado a la potencia k

[https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global\\_Objects/Math](https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Math)

# 5

TIPOS DE DATOS  
PRIMITIVOS:  
STRINGS



Los datos textuales son almacenados como **strings**.

El formato interno siempre es **UTF-16**, independientemente de la codificación de la página.

Como ya mencionamos, las cadenas se pueden rodear de comillas simples, dobles o del carácter backtick.

Se pueden insertar caracteres especiales en una cadena.

Carácter	Descripción
<code>\n</code>	Nueva línea
<code>\r</code>	Retorno de carro: En Windows, los archivos de texto usan una combinación de dos caracteres <code>\r\n</code> para representar un corte de línea mientras que en otros SO es simplemente <code>\n</code> . Esto es por razones históricas, la mayoría del software para Windows también entienden <code>\n</code> .
<code>\'</code> , <code>\"</code>	Comillas
<code>\\</code>	Barra invertida
<code>\t</code>	Tabulación
<code>\b</code> , <code>\f</code> , <code>\v</code>	Backspace, Form Feed, Vertical Tab – Se mantienen por compatibilidad. No son usados actualmente
<code>\xXX</code>	Carácter Unicode con el hexadecimal dado <code>XX</code> , por ej. <code>'\x7A'</code> es lo mismo que <code>'z'</code> .
<code>\uXXXX</code>	Un símbolo unicode con el hexadecimal dado <code>XXXX</code> en codificación UTF-16, p.ej. <code>\u00A9</code> – es el unicode para el símbolo copyright ©. Debe ser exactamente 4 dígitos hex.
<code>\u{X...XXXXXX}</code> (1 a 6 caracteres hex)	Un símbolo unicode con el hexadecimal dado en codificación UTF-32. Algunos caracteres raros son codificados con dos símbolos unicode, tomando 4 bytes. De esta manera podemos insertar códigos largos.

Observa que podemos incorporar cualquier carácter existente mediante su *code point* Unicode.

```
alert('\u00A9');           // ©  
alert('\u{20331}');        // 佬, un raro jeroglífico chino  
                           // (unicode largo)  
alert('\u{1F60D}');        // 😍, un emoticón sonriendo  
                           //(otro unicode largo)
```

Puedes consultar el código de un carácter en <https://unicode-table.com/es/>

Podemos conocer la longitud de una cadena con la **propiedad length**

```
let str = 'IES San Andrés';  
console.log( str.length );           // 14  
console.log( 'Villabalter'.length ); // 11
```



Hay dos formas de acceder a caracteres individuales de una cadena.

```
let str = 'IES San Andrés';  
console.log( str[0] );           // 'I'  
console.log( str.charAt(0) );    // 'I'
```

La notación más actual es el uso de corchetes.

**Diferencia:** si no se encuentra el carácter [] devuelve *undefined* y charAt() devuelve una cadena vacía.

También se puede iterar sobre los caracteres con **for ... of**

```
let str = 'IES San Andrés';  
for ( let char of str ) {  
    console.log(char);    // Imprime cada carácter en  
                           // una línea  
}
```

Los strings en JavaScript son **inmutables**, no se puede cambiar su contenido.

```
let str = 'IES San Andrés';

str[3] = 'X';           // Error
console.log( str );     // 'IES San Andrés'
console.log( str[3] );  // No funciona
```

Si queremos modificar una cadena tendremos que crear una nueva.

## toLowerCase() y toUpperCase()

Cambian todos los caracteres a minúsculas y mayúsculas respectivamente.

```
let str = 'Villabalter';  
  
console.log( str.toLowerCase() ); // villabalter  
console.log( str.toUpperCase() ); // VILLABALTER
```

## `str.indexOf( substr, pos )`

Busca la primera aparición de *substr* en la cadena *str* comenzando en la posición *pos*.

Si no encuentra devuelve -1

```
let str = 'IES San Andrés';  
console.log( str.indexOf( 'S' ) );           // 2  
console.log( str.indexOf( 'S', 3 ) );        // 4  
console.log( str.indexOf( 'S', 5 ) );        // -1  
console.log( str.indexOf( 'And' ) );         // 8
```

## `str.includes( substr, pos )`

Devuelve *true/false* en función de si la cadena *str* incluye la subcadena *substr* a partir de la posición *pos*.

```
let str = 'IES San Andrés';  
console.log( str.includes( 'S' ) );           // true  
console.log( str.includes( 'S', 3 ) );        // true  
console.log( str.includes( 'S', 5 ) );        // false  
console.log( str.includes( 'And' ) );         // true
```

## **`str.startsWith( substr )` y `str.endsWith( substr )`**

Devuelve *true/false* en función de si la cadena *str* comienza o finaliza por *substr*.

```
let str = 'IES San Andrés';  
console.log( str.startsWith( 'S' ) );      // false  
console.log( str.startsWith( 'IES' ) );    // true  
console.log( str.endsWith( 'S' ) );        // false
```

**str.slice( *comienzo*, [*final*] )**

Devuelve la subcadena desde *comienzo* hasta *final* (excluido este último).

```
let str = "stringify";  
console.log( str.slice( 0, 5 ) ); // strin  
console.log( str.slice( 0, 1 ) ); // s  
console.log( str.slice( 4 ) ); // ngify  
console.log( str.slice( -3, -1 ) ); // if
```



**str.substring( *comienzo*, [*final*] )**

Devuelve la subcadena desde *comienzo* hasta *final* (excluido este último). Permite que *comienzo* sea mayor que *final*.

No soporta números negativos (son tratados como 0)

```
let str = "stringify";  
console.log( str.substring( 0, 5 ) ); // strin  
console.log( str.substring( 5, 0 ) ); // strin  
console.log( str.substring( 4 ) );    // ngify  
console.log( str.substring( -3, 2 ) ); // st
```

**str.substr( *comienzo*, [*largo*] )**

En este caso se indica la longitud de la subcadena en lugar de la posición final. Permite que comienzo sea negativo.

```
let str = "stringify";  
console.log( str.substring( 0, 5 ) ); // strin  
console.log( str.substring( 5, 0 ) ); // strin  
console.log( str.substring( 4 ) );    // ngify  
console.log( str.substring( -3, 2 ) ); // st
```

método	selecciona...	negativos
<code>slice(comienzo, final)</code>	desde <code>comienzo</code> hasta <code>final</code> (sin incluir <code>final</code> )	permite negativos
<code>substring(comienzo, final)</code>	entre <code>comienzo</code> y <code>final</code>	valores negativos significan 0
<code>substr(comienzo, largo)</code>	desde <code>comienzo</code> toma <code>largo</code> caracteres	permite negativos <code>comienzo</code>

Como son equivalentes se aconseja utilizar únicamente uno de ellos. Lo más recomendable es utilizar **slice()**.

**str.padEnd( *targetLength*, [*padStr*, ...] )**

Rellena la cadena hasta alcanzar la longitud determinada con la subcadena que se le indique.

```
let str = 'Hola';  
console.log( ''.padEnd( 8, 'abc' ) );    // abcabcab
```

Hay otra función equivalente llamada **padStart()**

**str.repeat( *length* )**

Repite la cadena el número de veces indicado.

```
console.log( 'abc'.repeat(3) );    // abcabcabc  
console.log( '*'.repeat(6) );     // *****
```

## **str.replace( *searchFor*, *replaceWith* )**

Reemplaza por la primera ocurrencia de *searchFor* por *replaceWith*. La función **replaceAll()** reemplaza todas las ocurrencias.

```
console.log('villabalter'.replace('a', 'X'));      //  
villXbalter  
console.log('villabalter'.replaceAll('a', 'X'));  //  
villXbXlter
```

## **str.split( [separador, [limite]] )**

Convierte una cadena en array utilizando el *separador* indicado como para escoger donde empieza y acaba cada elemento.

```
let str = 'Esta es una cadena';  
console.log( str.split(' ')); // [ "Esta", "es", "una",  
"cadena" ]
```

Si se indica el parámetro *limite* solo se extraerán los primeros elementos, ignorando los demás.

```
let str = 'Esta es una cadena';  
console.log( str.split(' ', 2)); // [ "Esta", "es" ]
```

## `str.trim()`

Elimina los caracteres en blanco que haya al principio y al final de la cadena.

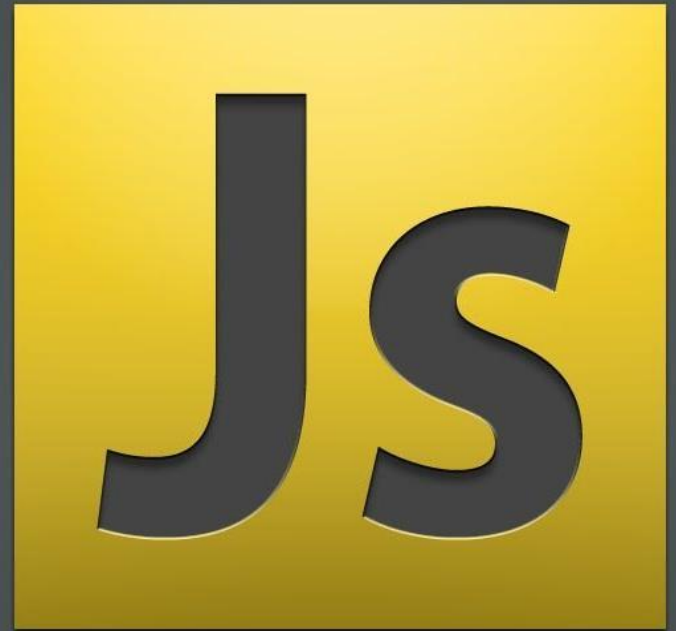
Hay otras dos versiones para eliminar solo los del principio (`trimStart()`) o solo los del final (`trimEnd()`)

```
let str = '  Hola  ';  
console.log( str.trim() );      // 'Hola'  
console.log( str.trimStart() ); // 'Hola  '  
console.log( str.trimEnd() );   // '  Hola'
```



# 5

## ARRAYS



Un **array** es una **colección ordenada** de elementos.

Los arrays se representan entre corchetes, separando los elementos en ellos por comas.

Para crear un array hay dos formas, aunque la más utilizada es la segunda ya que permite asignarle valor en la declaración.

```
let arr1 = new Array();  
let arr2 = [];  
let arr3 = [ 'DAW', 'DAM', 'ASIR' ]
```

Para acceder a un elemento de un array se utiliza la notación de corchetes.

Con la misma notación se puede cambiar el valor de un elemento o añadir nuevos elementos.

```
let arr = [ 'DAW', 'DAM', 'ASIR' ];  
console.log( arr[0] );           // DAW  
console.log( arr[6] );           // undefined  
arr[2] = 'SMR';  
console.log( arr );               // [ "DAW", "DAM", "SMR" ]  
arr[3] = 'ASIR';  
console.log( arr );               // [ "DAW", "DAM", "SMR", "ASIR" ]  
arr[6] = 'FPB';  
console.log( arr );               // [ "DAW", "DAM", "SMR", "ASIR",  
                                  // <2 empty slots>, "FPB" ]
```

Podemos obtener la longitud del array mediante la propiedad **length**

```
let arr = [ 'DAW', 'DAM', 'ASIR' ];  
console.log( arr.length );           // 3
```

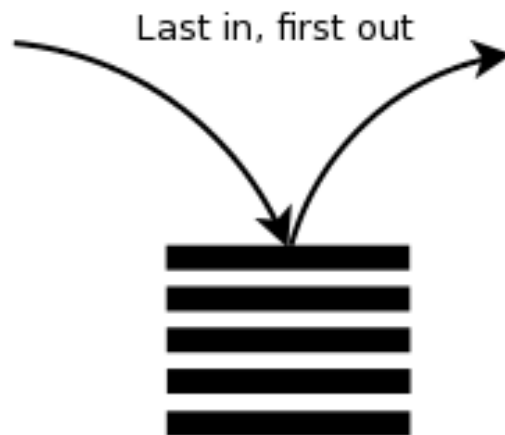
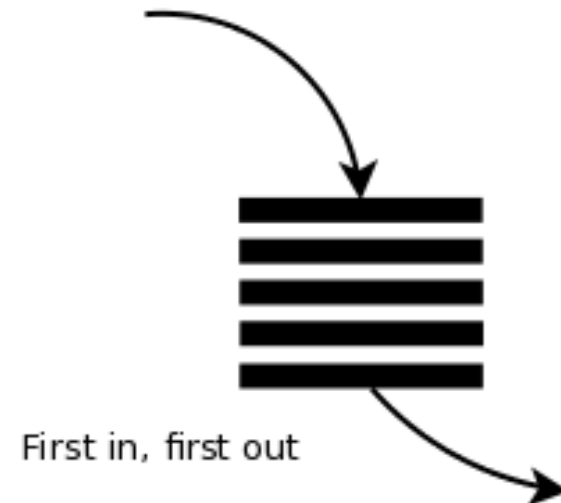
Los elementos del array pueden ser de cualquier tipo y no tienen por que ser todos del mismo tipo

```
let arr = [  
  'Desarrollo de Aplicaciones Web',  
  true,  
  180,  
  () => {  
    console.log('Soy una función');  
  }  
];
```

Alternativamente, podemos acceder a los elementos del array con la funci3n **at()**, que tiene la ventaja de permitir ́ndices negativos.

```
let arr = [ 'DAW', 'DAM', 'ASIR' ];  
  
console.log( arr.at(0) );           // DAW  
console.log( arr.at(-2) );          // DAM
```

Hay dos estructuras de datos muy utilizadas que son la **pila** (LIFO, Last Input, First Output) y la **cola** (FIFO, First Input, First Output).

**Stack:****Queue:**

Relacionados con estas estructuras de datos, hay cuatro métodos de inserción y extracción de datos de un array:

- **push()**: inserta un elemento al final
- **shift()**: obtiene un elemento del principio.
- **pop()**: obtiene un elemento del final
- **unshift()**: agrega un elemento al principio



The diagram illustrates four array operations on a list named `pets`. The list is initialized as `pets = [ "cats" , "dogs", "birds" ]`. Red arrows point from the `shift()` and `pop()` methods to the first and last elements of the list, respectively, indicating removal. Green arrows point from the `unshift()` and `push()` methods to the first and last elements of the list, respectively, indicating insertion.

```
pets.shift()  
removes
```

```
pets.pop()  
removes
```

```
pets = [ "cats" , "dogs", "birds" ]
```

```
pets.unshift("mice")  
inserts
```

```
pets.push("mice")  
inserts
```

Es posible iterar sobre los elementos de un array mediante sus índices en un bucle **for**.

```
let arr = [ 'uno', 'dos', 'tres' ];  
  
for ( let i=0; i < arr.length; i++ ) {  
    console.log( arr[i] );  
}
```

También es posible iterar sobre ellos con el constructor **for .. of**

```
let numbers = [ 'uno', 'dos', 'tres' ];  
  
for ( let number of numbers ) {  
    console.log( number );  
}
```

Los arrays tienen su propio método **toString** que devuelve todos sus elementos separados por coma.

```
let numbers = [ 'uno', 'dos', 'tres' ];  
  
alert( numbers );           //  
uno,dos,tres  
alert( String(numbers) == 'uno,dos,tres' ); // true
```

En este caso no da acceso al índice de cada elemento, pero en casos en que esto no es necesario es una opción más corta.

Al igual que pasaba con los objetos, no se pueden comparar arrays con el operador `==` ya que solo da positivo si ambos referencian el mismo array.

```
let numbers = [ 'uno', 'dos', 'tres' ];

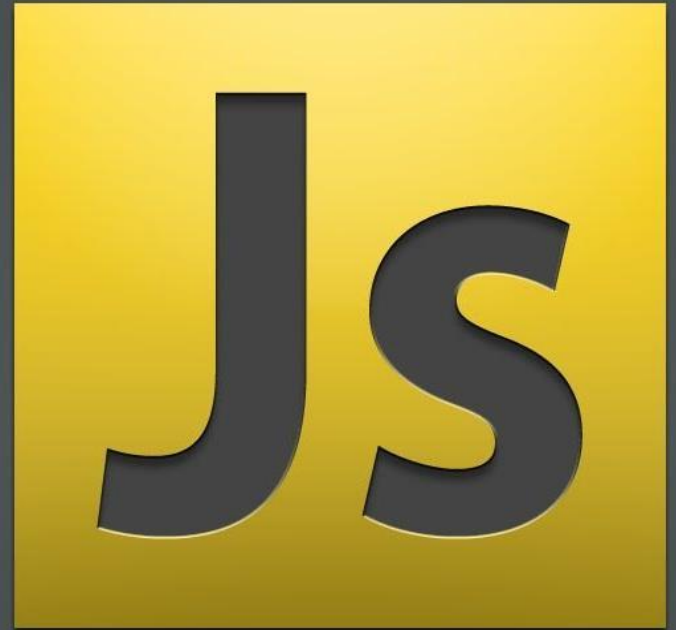
let numbers2 = numbers;

console.log( numbers == [ 'uno', 'dos', 'tres' ] ); // false
console.log(numbers == numbers2);                 // true
```

Para comparar arrays habría que crear una función que itere sobre todos los elementos de cada y los compare.

# 6

## MÉTODOS DE ARRAYS



**arr.splice( start[, deleteCount[, item1, item2, ...]] )**

El método **splice()** sirve para realizar varias operaciones con arrays: insertar, eliminar y reemplazar elementos.

Comenzando en *start*, elimina el número de elementos indicados en *deleteCount* e inserta los elementos *item1*, *item2*, ...

## Eliminar elementos

```
let arr = ['uno', 'dos', 'tres', 'cuatro'];  
arr.splice(1, 2);  
console.log(arr);    // ['uno', 'cuatro']
```

## Eliminar e insertar elementos

```
let arr = ['uno', 'dos', 'tres', 'cuatro'];  
  
arr.splice(1, 1, 'cinco');  
console.log(arr);    // ['uno', 'cinco', 'tres', 'cuatro']
```

## Solo insertar elementos

```
let arr = ['uno', 'dos', 'tres', 'cuatro'];  
  
arr.splice(1, 0, 'cinco');  
console.log(arr);    // ['uno', 'cinco', 'dos', 'tres',  
                      // 'cuatro']
```



El método **splice()** modifica directamente el array, pero si recogemos los datos que devuelve veremos que **devuelve un array con los elementos eliminados**.

```
let arr = ['uno', 'dos', 'tres', 'cuatro'];  
let removed = arr.splice(1, 2, 'cinco');  
  
console.log(arr);           // ['uno', 'cinco', 'cuatro']  
console.log(removed);       // [ 'dos', 'tres']
```

Al igual que en otros métodos con arrays, se permiten **índices negativos**, que hacen referencia a elementos contando desde el final del array.

```
let arr = ['uno', 'dos', 'tres', 'cuatro'];  
  
arr.splice(-2, 1);  
console.log(arr);           // ['uno', 'dos', 'cuatro']
```

**arr.slice( [principio,] [final/] )**

El método **slice()** devuelve un **nuevo array** copiando en él todos los elementos desde *principio* hasta *final* (*no incluido*).

Observa que este método no modifica el array original.

```
let arr = ['uno', 'dos', 'tres', 'cuatro'];  
let arr2 = arr.slice(1, 2);  
  
console.log(arr);           // ['uno', 'dos', 'tres', 'cuatro']  
console.log(arr2);          // ['dos']
```

Si omitimos *final* tomará por defecto el final del array.

Si omitimos tanto *principio* como *final* nos devolverá una copia del array, por lo que este método se utiliza habitualmente para crear copias de arrays

```
let arr = ['uno', 'dos', 'tres', 'cuatro'];  
let arr2 = arr.slice(2);  
let arr3 = arr.slice();  
  
console.log(arr);           // ['uno', 'dos', 'tres', 'cuatro']  
console.log(arr2);          // ['tres', 'cuatro']  
console.log(arr3);          // ['uno', 'dos', 'tres', 'cuatro']
```

## **arr.concat( arg1, arg2, ...)**

Los argumentos pueden ser tanto arrays como valores.

Devuelve un nuevo array que está formado por la concatenación del array con todos los arrays y valores que se le pasen como parámetro.

```
let arr = [ 1, 2 ];

alert(arr.concat( [ 3, 4 ] ));           // [1, 2, 3, 4, 5, 6]
alert(arr.concat( [ 3, 4 ], [ 5, 6 ] )); // [1, 2, 3, 4, 5, 6]
alert(arr.concat( 3, 4 ));               // [1, 2, 3, 4]
alert(arr.concat( 3, [4, 5] ));          // [1, 2, 3, 4, 5]
```

## `arr.forEach( function )`

El método **`forEach()`** permite ejecutar una función para cada elemento del array.

Esta función podrá tomar tres parámetros:

- **item**: valor del elemento del array correspondiente a la iteración
- **index**: índice de dicho elemento
- **array**: el array completo

```
let arr = [ 'a', 'b', 'c' ];

arr.forEach( function( item, index, array ) {
  console.log(`El elemento ${item} tiene la posición
${index}`);
} )
```

---

El elemento a tiene la posición 0

---

El elemento b tiene la posición 1

---

El elemento c tiene la posición 2

La función no tiene por qué recoger todos los parámetros si no los va a utilizar.

```
let arr = [ 2, 5, 8 ];  
  
arr.forEach( function( item ) {  
    console.log(`El cuadrado de ${item} es ${item**2}`);  
} )
```

---

El cuadrado de 2 es 4

---

El cuadrado de 5 es 25

---

El cuadrado de 8 es 64



También se pueden utilizar funciones flecha

```
let arr = [ 2, 5, 8 ];  
arr.forEach( item => alert(`El cuadrado de ${item} es  
${item**2}`));
```

```
let arr = [ 'a', 'b', 'c' ];  
arr.forEach( ( item, index ) => {  
    console.log(`El elemento ${item} tiene posición ${index}`);  
} )
```

**arr.indexOf( *item*, *from* ) y arr.includes( *item*, *from* )**

Similares a las funciones homónimas con cadenas.

**indexOf()** devuelve la posición de un objeto y -1 en caso de no encontrarlo.

**includes()** devuelve *true* si el array incluye el elemento buscado.

El método **some( func )** devuelve *true* si el array contiene por lo menos un elemento que, al pasárselo a la función, devuelve *true*.

```
let arr = [ 1, 4, 5, 23, 71 ];  
  
let hasEven = arr.some( (item) => item%2 == 0 );  
  
console.log(hasEven);           // True
```

El método **every**( *func* ) devuelve *true* si la función pasada devuelve *true* para todos los elementos del array.

```
let arr1 = [ 1, 4, 5, 23, 71 ];  
let arr2 = [ 2, 6, 54, 198 ];  
  
let isEven = (item) => item%2 == 0;  
  
console.log( arr1.every( isEven ) );           // false  
console.log( arr2.every( isEven ) );           // true
```

## **arr.find( *function* ) y arr.findIndex( *function* )**

Útil cuando queremos buscar elementos que cumplan una condición determinada.

Aplica la función a cada elemento y devuelve el primer elemento (o su índice) cuyo valor devuelto por la función sea *true*.

La sintaxis básica es:

```
arr.find( function( item, index, array ) {  
    // Devuelve true o false  
});
```

```
let users = [  
  { id: 1, name: "Victor" },  
  { id: 2, name: "Oscar" },  
  { id: 3, name: "Iván" },  
];  
  
let user = users.find( item => item.id == 1 );  
  
console.log(user.name);      // Victor
```

## `arr.filter( function )`

Similar a `find()`, pero en lugar de devolver el primer elemento que cumple la condición **devuelve un array con todos los elementos que la cumplen.**

La sintaxis es análoga a **`find()`**, ya que se le pasa una función que debe devolver *true* o *false*.

```
arr.filter( function( item, index, array ) {  
    // Devuelve true o false  
});
```

```
let users = [  
  { id: 1, name: "Victor", rol: 'admin' },  
  { id: 2, name: "Oscar", rol: 'user' },  
  { id: 3, name: "Iván", rol: 'user' },  
];  
  
let user = users.filter( item => item.rol == 'user' );  
  
console.log(user.name);      // [ { id: 2, name: "Oscar", rol:  
  'user' },  
                                //   { id: 3, name: "Iván", rol:  
  'user' } ]
```



## `arr.map( function )`

Este método permite transformar un array aplicando una función sobre cada uno de los miembros del mismo.

Esta función recoge los mismos argumentos que en los ejemplos anteriores.

```
arr.map( function( item, index, array ) {  
    // Devuelve true o false  
});
```

```
let prices = [ 20, 87, 23];  
let iva = prices.map( item => item*1.21 );  
  
console.log(iva);    // [ 24.2, 105.27, 27.83 ]
```

Este método no modifica el array original, sino que devuelve un nuevo array.

## `arr.sort( function )`

Este método sirve para ordenar el array. Al contrario que **map()**, sí que modifica el array original.

```
let arr = [ 'DAM', 'DAW', 'ASIR' ];  
let arr2 = [ 1, 2, 15 ];  
  
arr.sort();  
arr2.sort();  
console.log(arr);    // [ "ASIR", "DAM", "DAW" ]  
console.log(arr2);   // [ 1, 15, 2 ]
```

Como puedes apreciar, interpreta todos los valores como cadenas para realizar las comparaciones.

Si queremos que ordene números (o utilizar cualquier otro criterio) debemos pasar nuestra propia función de ordenación.

La función recibe 2 parámetros **a** y **b**, que son los dos elementos que se compararán en cada una de las iteraciones del algoritmo de ordenación, y debe devolver:

- 1 si  $a > b$
- 0 si  $a = b$
- -1 si  $a < b$

En realidad, es suficiente con que devuelva positivo ( $a > b$ ) o negativo ( $a < b$ )

Teniendo en cuenta lo anterior, la función quedaría:

```
let arr = [1, 2, 15 ];  
  
arr.sort( (a, b) => a-b );  
  
console.log(arr); // [ 1, 2, 15 ]
```

## arr.reverse()

Un método muy sencillo que invierte el orden del array, de forma que el primer elemento pasa a ser el último y viceversa.

```
let arr = [ 'a', 'b', 'c', 'e', 'f' ];  
arr.reverse();  
  
console.log(arr);    // [ "f", "e", "c", "b", "a" ]
```

## **str.split( delim, len )** y **arr.join( delim )**

Estos métodos sirven para descomponer una cadena en partes que se almacenan en un array (**split**) y al revés, para combinar los elementos de un array en una cadena (**join**).

Al **split** se le pasa la cadena delimitadora y, opcionalmente, el número de elementos que extraerá.

```
let cursos = 'DAW, DAM, ASIR';

let arr = cursos.split(', ');
console.log(arr);           // [ "DAW", "DAM", "ASIR" ]

let arr2 = cursos.split(', ', 2);
console.log(arr2);          // [ "DAW", "DAM" ]
```

Si pasamos como parámetro la cadena vacía separará cada uno de los caracteres de la cadena.

```
let a = 'León';  
console.log(a.split('')); // [ "L", "e", "ó", "n" ]
```



El método **join( *delim* )** hace exactamente lo contrario, combina todos elementos de un array en una cadena separándolos con la cadena *delim*.

```
let a = [ 'DAM', 'DAW', 'ASIR' ]  
console.log(a.join(' - '));    // DAM - DAW - ASIR
```

El método **reduce**( *func* ) sirve para iterar sobre los elementos de un array y realizar una operación sobre todos ellos para finalmente **devolver un único valor**.

La sintaxis es la siguiente:

```
arr.reduce( function(acum, item, index, array) {  
    // ...  
}, initialValue )
```

Observa que la función que se le pasa tiene los parámetros que hemos visto en otras funciones, pero también un parámetro **acumulador**.

Además, recoge otro parámetro que **inicializa el acumulador**.

## Ejemplo 1: suma de todos los elementos de un array

```
let arr = [ 4, 7, 2, 74, 12, 85 ];

let suma = arr.reduce( (acum, item, index, arr) => {
  return acum+item;
}, 0 );

console.log(suma);
```

**Ejemplo 2:** cálculo de una contraseña tomando las 2 primeras letras de nombre y apellidos.

```
let arr = [ 'Victor', 'González', 'Rodríguez' ];

let pass = arr.reduce( (acum, item) => {
  return acum + item.substring(0, 2).toLowerCase();
}, '' );

console.log(pass); // vigoro
```

Dado que los arrays son objetos, el operador *typeof* no nos servirá para identificarlos, ya que simplemente nos dirá con son objetos.

Para saber si una variable contiene un array debemos utilizar el método **Array.isArray()**.






```
let arr = [ 1, 2, 3 ];  
  
console.log( typeof arr );           // Object  
console.log( Array.isArray(arr) );  // true
```






Observa cómo se invoca, ya que no es un método del array que queremos comprobar, sino del objeto **Array**.











# JAVASCRIPT





## ARRAY CHEAT SHEET







   .filter() → 






   .find() → 









   .findIndex() → **1**






   .some() → **true**









   .every() → **false**









   .reverse() →   








   .shift() →  

   .unshift() →    

   .pop() →  

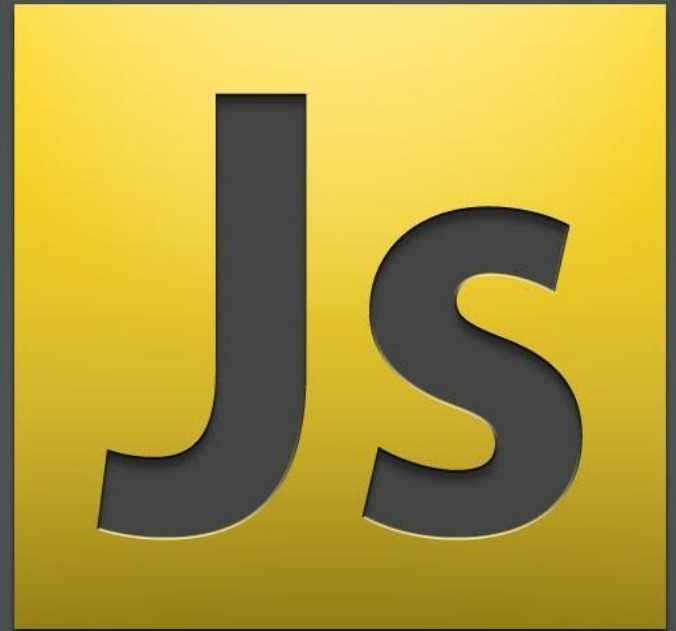
   .push() →    

   .map( → ) →   

   .fill(1 → ) →   

7

MAP Y SET

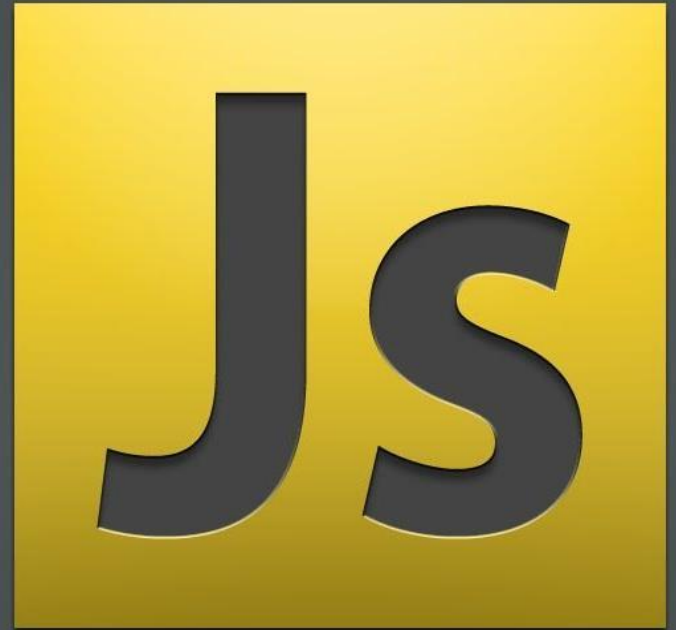


XXXX



# 8

FECHA Y HORA



XXXX

# 9

## MÉTODOS JSON





