UNIDAD 3. Arrays y objetos definidos por el usuario

Desarrollo Web en Entorno Cliente

2º DAW

Contenidos

Objetos

Los **objetos** son tipos de datos que agrupan propiedades. Esas propiedades serán variables especiales asociadas a un objeto. En JavaScript tenemos dos tipos diferenciados de objetos:

- **Literal de objetos**: son objetos creados directamente, a través de un literal de tipo objeto {} y será de tipo `Object`.
- **Instancia de una clase**: se crean a partir de una clase que haya definido el usuario, o bien utilizando prototipos, o bien a través de las clases incorporadas desde ES6 y serán de ese tipo de datos que hemos definido con la clase.

Crear un literal de objetos

Para crear una variable de tipo objeto, se definen sus propiedades (y métodos) y los valores de sus propiedades (y métodos), de la siguiente forma:

```
let variable = {
     propiedad1: valor1,
     propiedad2: valor2,
     …
};
```

Ejemplo:

```
let persona1 = {
     nombre: "Ada",
     apellido: "Lovelace",
     ano: 1815
};
```

También se pueden crear instanciando el objeto Object(), y dando un valor a cada una de sus propiedades (o métodos):

```
let variable = new Object();
variable.propiedad1 = valor1;
variable.propiedad2 = valor2;
...
```

Ejemplo:

```
let persona2 = new Object();
persona2.nombre = "Ada";
persona2.apellido = "Lovelace";
persona2.ano = 1815;
```

Operaciones con las propiedades de un objeto

Sintaxis para acceder a las propiedades de un objeto
La forma más utilizada es:

```
objeto.propiedad;
```

Pero también se puede utilizar:

```
objeto[propiedad];
```

** propiedad es una expresión de tipo cadena con el nombre de la propiedad

Ejemplo: acceder a las propiedades de un objeto

```
persona3.nombre;
persona3["nombre"];
```

Recorrer las propiedades de un objeto

La sentencia `for..in`, nos permite recorrer los nombres de las propiedades y métodos de un objeto.

Sintaxis:

```
for (nomProp in nomObjeto) {
      nomObjeto[nomProp]
}
```

Ejemplo:

```
for (prop in persona1) {
      console.log("La propiedad " + prop +
        " tiene el valor " + persona1[prop]);
}
```

También disponemos de los **métodos** estáticos de Object**: keys(), values() y entries()**[★] para recorrer las propiedades de los objetos.

Método	Descripción
`Object.keys(objeto);`	Devuelve un array con el nombre de las propiedades de *objeto*. Ej. `Object.keys({a:3, b:2}); // => ['a', 'b']`
`Object.values(objeto);`	Devuelve un array con los valores de *objeto*. Ej. `Object.values({a:3, b:2}); // => [3, 2]`
`Object.entries(objeto);`	Devuelve array con los pares propiedad-valor de *objeto*. Ej `Object.entries({a:3, b:2}); // => [['a',3], ['b',2]]`

Añadir una propiedad dinámica a un objeto

Se puede añadir una propiedad a un objeto que ya existe, simplemente utilizándola y asignándole un valor:

★ *EcmaScript 2017 o ES6+*

```
nomObjeto.nuevaPropiedad = valor;
```

Ejemplo:

```
persona1.nacionalidad = "Inglaterra";
```

Borrar una propiedad dinámica a un objeto

Se puede borrar una propiedad a un objeto que ya existe, utilizando el comando `delete`.

```
delete nomObjeto.propiedad;
```

Ejemplo:

```
delete persona1.nacionalidad;
```

Comprobar las propiedades de un objeto

Se puede probar si una propiedad existe en un objeto en un determinado momento con la palabra reservada `in`, que devolverá *true* o *false* según exista la propiedad o no.

```
"propiedad" in nomObjeto;
```

Ejemplo:

```
"nacionalidad" in persona1;
```

Objetos anidados

Los objetos pueden anidarse entre sí. Los objetos anidados representan árboles. Para acceder a las propiedades podemos encadenar el operador punto . o corchetes [].

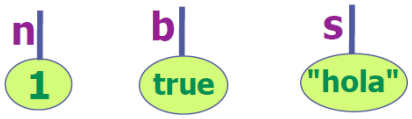
Ejemplo:

```
let pelicula = {
    titulo: 'E.T.',
    director:{
        nombre:'Steven',
        apellido: 'Spielberg'
    }
};
pelicula.titulo; // => 'E.T.';
pelicula.director.nombre; // => 'Steven';
pelicula['director']['nombre']; // => 'Steven';
pelicula['director'].apellido; // => 'Spielberg';
pelicula.director; // => {nombre:'Steven', apellido:'Spielberg'};
```

Referencias a objetos

Valores y referencias

Los tipos JavaScript se gestionan por valor o por referencia.



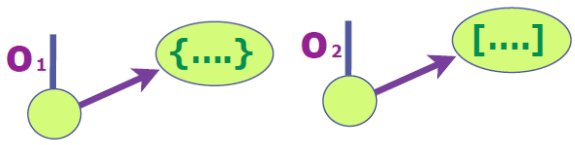
Los tipos primitivos `number`, `string`, `boolean` o `undefined` se manejan por **valor**, por lo tanto:

- En una operación de asignación, se copia el valor de la variable.

- En una operación de identidad (igualdad estricta o fuerte === o !==) y de igualdad (débil == o !=), se comparan los valores.

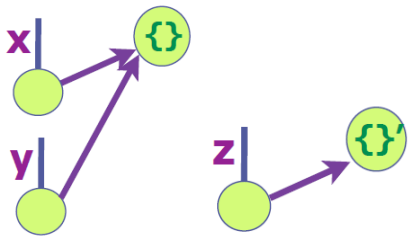
Los objetos se manejan con una **referencia** (contenida en las variables). Por ejemplo Object, Array, Function, Date, ... Lo que implica:

- En una operación de asignación, se copia solo la referencia (contenida en las variables), por lo que cuando queremos copiar el objeto debemos clonarlo.
- En una operación de identidad (igualdad estricta o fuerte ===) se comparan las referencias. La identidad de objetos indica que son el mismo objeto, ya que dos objetos distintos con el mismo contenido no son idénticos, no tienen la misma referencia.
- La operación de igualdad (débil == o !=) no tiene sentido con objetos, se recomienda no utilizarla.



Ejemplo:

```
let x = {}; // x e y contienen la
let y = x; // misma referencia
let z = {} // la referencia a z
// es diferente de x e y
x === y // => true
x === {} // => false
x === z // => false
```



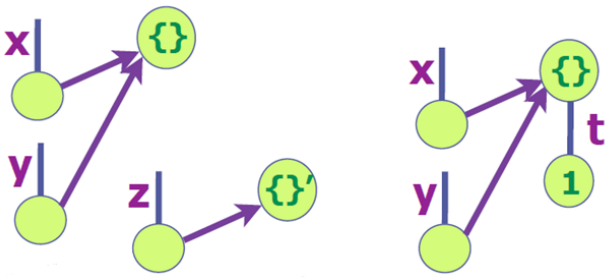
Efectos laterales de las referencias a objetos

Como las variables que contienen objetos solo guardan la referencia al objeto y al asignar una variable, lo que se copia, es la referencia, si se modifica el objeto de una de ellas, todas las variables que contengan la misma referencia, verán el objeto modificado, ya que el objeto está alojado en un lugar de la memoria apuntado por la referencia.

Hemos de tener en cuenta que los parámetros y valores de retorno de funciones de objetos también acceden por referencia y tienen el mismo efecto.

Ejemplo:

```
let x = {}; // x e y tienen la
let y = x; // misma referencia
let z = {}; // la referencia a z es diferente de las de x e y
y.t = 1; // Añade la propiedad t a y
x.t; // => 1 x accede al mismo
y.t; // => 1 objeto que y
z.t; // => undefined
```



Clases★

Una **clase** es un modelo que define un conjunto de variables (atributos o propiedades) y métodos apropiados para operar con dichos datos. Cada instancia que se crea de la clase es un **objeto**.

Las clases en JavaScript se introdujeron en la versión ES6 y supusieron una mejora significativa en la herencia basada en prototipos de JavaScript a la que estábamos acostumbrados. Estas clases nos dan una sintaxis más clara y simple para crear objetos y trabajar con herencia.

A diferencia de la versión anterior de JavaScript en la que utilizábamos la palabra reservada *function* para definir una "clase", en esta versión se utiliza la palabra reservada `class`; además, las propiedades se deben indicar dentro de un método constructor.

Definir una clase en ES6

Par definir una clase en ES6 utilizamos la palabra reservada `class` para crear la clase, que tendrá una función denominada `constructor`. Tendremos en cuenta los siguiente:

- El nombre que demos a la clase empezará con mayúsculas.
- Dentro de la función `constructor`, los nombres de las propiedades del objeto van precedidos con la palabra reservada `this`.
- El objeto se crea (instancia) con la palabra reservada `new()`.

La sintaxis para la creación de la clase sería la siguiente:

```
class NombreClase {
  constructor(parametro1 [,parametro 2...]) {
    this.propiedad1 = parametro1;
    [this.propiedad2 = parametro2;]
  }
  método() {…}
}
```

Y para crear o instanciar un objeto de la clase, utilizamos la palabra reservada `new`:

```
let|const nombreObjeto = new NombreClase (argumentos);
```

La palabra reservada `this` es una referencia entorno de ejecución y referencia a window cuando el programa está en el entorno global. Por lo tanto, la referencia de `this` dependerá de su contexto:

- Cuando hacemos una llamada a una función: hace referencia al propietario de la función que está invocándose.
- Cuando la función llamada es un método de un objeto: hace referencia al objeto que ha definido esa función.

★ *EcmaScript 2015 ES6 o JS ES6 o JavaScript 6.*

Hemos de tener en cuenta que en las funciones flecha o *arrow functions*, el contexto de `this` cambia, por lo tanto, no podemos usaras para definir constructores de objetos.

Ejemplo:

```
class Persona {
    constructor(nombre, apellido, nacimiento) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.nacimiento = nacimiento;
    }
}
let persona = new Persona("Ada", "Lovelace", 1815);
```

Métodos de la clase

Para definir los métodos de la clase, tenemos diferentes opciones de sintaxis. La mas utilizada es la siguiente:

```
nombreMetodo() {
     //sentencias del método
}
```

También podemos hacerlo con funciones anónimas:

```
nombreMetodo: function() {
     //sentencias del método
}
```

Y en lugar de una función anónima, también podemos utilizar una función con nombre:

```
nombreMetodo = miFuncion;
funcion miFuncion() {
     //sentencias del método
}
```

Para realizar la llamada al método:

```
nombreObjeto.nombreMetodo();
```

Ejemplo:

```
class Persona {
    constructor(nombre, apellido, nacimiento) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.nacimiento = nacimiento;
    }
    nombreCompleto() {
       return this.nombre + " " +  this.apellido;
    }
}
let persona = new Persona("Ada", "Lovelace", 1815);
persona.nombreCompleto(); // => "Ada Lovelace"
```

Definir una clase utilizando prototipos ES5

Antes de existir las clases en JavaScript, se realizaba una simulación de clases utilizando funciones, objetos y prototipos.

El **prototipo** es un objeto JavaScript con los métodos heredados de la clase, es decir, el prototipo contendrá todas las propiedades y métodos de cualquier clase bien sea para crear objetos nativos del lenguaje o definida por nosotros. Cualquier objeto tiene una propiedad `prototype` que da acceso al prototipo y permite añadir métodos heredados.

Los *constructores* de estas "clases" se crean de una forma un poco peculiar: se define una función, y los objetos se crean instanciando en nombre de la función. Para ello:

- Se crea una función con el nombre del constructor (por convención, su nombre empezará con mayúsculas).
- Dentro de la función, los nombres de las propiedades (y métodos) del objeto van precedidos con la palabra reservada `this`.
- El objeto se crea (instancia) con la palabra reservada `new()`.

Sintaxis para definir el prototipo de una "clase":

```
function Constructor (valor1, valor2, valor3…){
      this.propiedad1 = valor1;
      this.propiedad2 = valor2;
      ….
      this.metodo = function() {…};
}
let variable = new Constructor(v1, v2, ...);
```

Ejemplo:

```
function Persona (nombre, apellido, nacimiento){
      this.nombre = nombre;
      this.apellido = apellido;
      this. nacimiento = nacimiento;
}
let persona = new Persona("Ada", "Lovelace", 1815);
```

Definir un Constructor con métodos

Se pueden crear los métodos a la hora de definir el constructor del objeto.

Ejemplo:

```
function Persona (nombre, apellido, nacimiento){
      this.nombre = nombre;
      this.apellido = apellido;
      this. nacimiento = nacimiento;
      this.nombreCompleto = function() {
            return (this.nombre + " " + this.apellido);
      }
}
let persona = new Persona("Ada", "Lovelace", 1815);
```

Modificar el prototipo

Cuando necesitemos añadir una propiedad o un método a todos los objetos creados a partir de una clase, tendremos dos opciones:

- Añadirlos directamente sobre la definición del prototipo, es decir, modificar la definición de la clase.
- Modificando el prototipo como veremos a continuación.

Añadir una propiedad o un método a un prototipo

```
<prototipo>.prototype.<propiedad_metodo> = <valor o función>;
```

Ejemplo:

```
//Añadir una propiedad al prototipo de la clase Persona
Persona.prototype.muerte = "2000
//Añadir un método al prototipo de la clase Persona
Persona.prototype.defuncion = function () {
    return "Defunción en el año " + this.muerte;
}
```

Los prototipos también permiten modificar clases ya existentes.

Ejemplo: extensión del prototipo String para añadir una nueva función que devuelve las palabras que forman una frase. Podemos utilizar la nueva función en cualquier String de nuestro programa.

```
String.prototype.words = function(){
    return this.trim().split(" ")
}
let frase = " Buenos días";
frase.words(); // => ["Buenos", "días"]
```

Métodos get y set de las clases★

Estos métodos se usan en la definición de clases para extraer (`get`) y modificar (`set`) las propiedades de un objeto. De este modo, nosotros podemos elegir exactamente, mediante estos métodos, qué propiedades pueden ser accedidas y modificadas y cuáles no.

De hecho, los *getters* y *setters* determinan el fundamento del principio de encapsulación de la programación orientada a objetos.

Lo habitual en otros lenguajes de programación es definir los *getters* y *setters* con la palabra `get` o `set` seguida del nombre de la propiedad, pero JavaScript es un caso especial, y los *getters* y *setters* se escriben con la palabra `get` o `set`, separadas por un espacio del nombre de la propiedad, con una particularidad: ¡no podemos poner el mismo nombre al método que a la propiedad porque entraríamos en un bucle! Por eso muchos desarrolladores utilizan el guion bajo para nombrar la propiedad.

Ejemplo:

```
class Telefono {
    constructor(marca){
      this._marca = marca;
    }
    get marca() {
      return this._marca;
    }
    set marca (mar) {
      this._marca = mar;
    }
}
let miTelefono = new Telefono ("Google");
miTelefono.marca = "iPhone";
mensaje.innerHTML = "Mi telefono es un " + miTelefono.marca;
```

Herencia de las clases★

El trabajar con la herencia en JavaScript, mejora significativamente la herencia basada en prototipos que ya conocemos de versiones anteriores. Lo primero que debemos tener en cuenta es que todas las clases que creamos heredan de `Object` a no ser que indiquemos lo contrario.

Al igual que en otros lenguajes de programación, definimos la relación entre una clase padre y una clase hijo utilizando la palabra reservada `extends`, que definirá la relación de esta segunda clase con la primera. Además, para hacer referencia a los atributos de la clase padre utilizamos la palabra `super`.

La sintaxis para definir una cabecera de una clase hijo sería la siguiente:

```
class ClaseHijo extends ClasePadre
```

Ejemplo:

```
class Telefono {
    constructor (marca){
        this.marca = marca;
    }
}
class Modelo extends Telefono {
    constructor (marca, modelo){
        super(marca);
        this.modelo = modelo;
    }
}
```

★ *EcmaScript 2015 o ES6 o JS ES6 o JavaScript 6.*

Métodos estáticos de las clases ★

Para crear métodos estáticos utilizamos la palabra reservada `static`. Al igual que ocurre en otros lenguajes de programación, **un método estático se llama directamente sin instanciar la clase** (es decir, sin necesidad de crear un objeto). De hecho, si tratáramos de llamar a un método estático a partir de un objeto obtendríamos un error.

Este tipo de métodos se utilizan sobre todo para crear funciones de utilidad en una aplicación.

Sintaxis para la definición:

```
static nombreMetodo(parametros) {
    //sentencias del método
}
```

Para llamar al método:

```
NombreClase.nombreMetodo();
```

Arrays

Introducción

En JavaScript los Arrays son un tipo especial de objetos, que almacenan múltiples valores que pueden ser de diferentes tipos (number, string, object, array…).

Crear un array

Forma aconsejada

Sintaxis:

```
let nomArray = [valores_separados_por_comas];
```

Ejemplo:

```
let arrayVacio = [];
let animales = ["perro", "gato", "loro", 100];
```

Con new Array(*valor1, valor2, …*) // No aconsejado

Sintaxis:

```
let nomArray2 = new Array (valores_separados_por_comas);
```

Ejemplo:

```
let arrayVacio = new Array();
let animales = new Array("perro", "gato", "loro", 100);
```

Con new Array(*num*) // No aconsejado

Si pasamos solo un parámetro, que sea de tipo numérico, creará un array con ese número de elementos con valor undefined.

Ejemplo:

```
let array40ElementosVacios = new Array(40);
```

Acceder a un elemento de un array

Se referencian con un índice numérico (a diferencia de los objetos, que se pueden referenciar con un índice de tipo string).

Sintaxis:

```
nomArray[indice];
```

donde indice: empieza en 0, hasta nomArray.length – 1

Ejemplo:

```
animales[1];
```

Propiedades: .length

Propiedad	Descripción
`.length`	Número de elementos del array.

Ejemplo:

```
let long = animales.length;
```

Método .forEach(). Recorrer los elementos de un array

Se pueden recorrer los elementos de un array utilizando un bucle `for` o mediante el médodo `forEach()`:

Ejemplo con el bucle for:

```
for (let i = 0; i < animales.length; i++) {
  console.log(animales[i]);
}
```

Método	Descripción
`.forEach(funcion(`*`valor`*` [,`*`índice`*` [,`*`array`*`]])) {})`	Recorre el array y ejecuta la función para cada elemento del array. La función tiene tres argumentos: • *valor*: el valor del elemento. • *índice*: (opcional) la posición del elemento. • *array*: (opcional) el array en sí. La estructura de esta función se utiliza del mismo modo para otros métodos del objeto Array como `.find()`, `.findIndex()`, `.map()` y `filter()`.

Ejemplos:

```
//Con función clásica
animales.forEach(function (valor, indice, array) {
   console.log("Posición: " + indice + " - Contenido: " + valor);
});
//Con función flecha
let n = [7, 4, 1, 23];
let add = 0;
n.forEach(elem => add += elem)   // add => 35
```

Otras formas de recorrer los elementos de un array★

Ya conocemos tres formas de recorrer los elementos de un array: bucle `for`, bucle `for..in` y método `.forEach()`. Existe una cuarta opción incorporada en la versión ES6+: el bucle `for..of`. Es muy similar al bucle `for..in`, por lo que vamos a compararlos.

Sintaxis:

```
for (elemento of nomArray) {
      elemento;
}
```

★ *EcmaScript 2015 y 2016 o ES6+ o JS ES6+ o JavaScript 6+.*

Sentencia **`for..in`**:

- Itera en objetos y en arrays.
- Al iterar en objetos, la variable del bucle itera con el nombre de la propiedad (string).
- Al iterar en arrays, la variable del bucle itera con el índice del elemento (número).

Ejemplo:

```
let n = [7, 4, 1, 23];
let add = 0;
for (let i in n){
    add += n[i];
}     // add => 35
```

Sentencia **`for..of`**:

- Solo itera en objetos iterables, no en objetos.
- La variable del bucle contiene en cada iteración un elemento del array (u objeto iterable).

Ejemplo:

```
let n = [7, 4, 1, 23];
let add = 0;
for (let elem of n){
    add += elem;
}     // add => 35
```

Método .isArray(). Saber si una variable es un array

Hay dos modos de saber si una variable contiene un array:

- La primera es utilizando un método del objeto `Array`:

  ```
  Array.isArray(nomVariable)
  ```

  Si utilizamos `typeof nomVariable`, devolverá `Object`.
- Con el operador `instanceof`:

  ```
  let nomVariable instanceof Array
  ```

Métodos .toString() y .join(). Obtener una cadena a partir de un array

Devuelven una cadena formada concatenando los elementos del array. Estos métodos **no modifican el array** sobre el que se aplican.

Método	Descripción
`.toString()`	Devuelve una cadena formada por los elementos del array separados por comas. Ej: `let cad = animales.toString();`

Método	Descripción
.join(["*separador*"])	Devuelve una cadena formada por los elementos del array separados con el ***separador***. El valor por defecto del ***separador*** es la coma. Ej: `let cad = animales.join("***");`

Métodos para modificar un array

Estos métodos **sí que modifican el array sobre el que se aplican**.

Métodos .pop() y .shift(). Eliminar elementos del final o del principio

Método	Descripción
.pop()	Elimina el último elemento de un array y devuelve ese elemento. Ej. `let ultimo = animales.pop();`
.shift()	Elimina el primer elemento de un array y devuelve ese elemento. Ej. `let primero = animales.shift();`

Otra forma de eliminar elementos de un array (no aconsejable) es utilizando `delete`. Elimina el contenido de una posición del array, y deja como valor `undefined`.

Ejemplo:

```
delete animales[2];
```

Métodos .push() y .unshift(). Añadir elementos al final o al principio

Método	Descripción
.push(*elementos*)	Añade los *elementos* separados por comas al final del array. Devuelve la nueva longitud del array. Ej. `let longNueva = animales.push("lora");`
.unshift(*elementos*)	Añade los *elementos* separados por comas al principio del array. Devuelve la nueva longitud del array. Ej. `let longNueva =  animales.push("perra");`

Otra forma de añadir un elemento, es asignando un valor a una posición del array. No es recomendable, porque se puede sobreescribir un elemento, o dejar huecos (con valor undefined).

Ejemplo:

```
animales[6] = "avestruz";
```

Método .splice(). Eliminar y/o añadir elementos en cualquier posición

Método	Descripción
.splice(*inicio*, *num_elementos_borrar* [, *elementos_añadir*])	Cambia el contenido de un array, eliminando elementos y/o añadiendo nuevos elementos. Desde la posición *inicio*, elimina *num_elementos*, y añade la lista de *elementos_añadir* separados por comas, si hay alguno. Devuelve un array con los elementos eliminados. Ej. `let a = animales.splice(1,2,"loro", "pez");`

Métodos .fill() y .copyWithin(). Rellenar un array (o parte) con un elemento

Método	Descripción
.fill(*elemento*[, *ini* [, *fin*]])	Sustituye todos los elementos de array por el *elemento*, desde la posición *ini* hasta la posición *fin* (no incluido). • *ini: si se omite,* empieza en la posición 0. • *fin: si se omite,* llega hasta la longitud – 1. Es decir, si se omiten *ini* y *fin*, afecta a todo el array. Devuelve un puntero al array modificado. Modifica el array original. Ej. `animales.fill("");`
.copyWithin(*posición, [, ini[, [fin]]*);	Sustituye los elementos del array desde *posición*, copiando los elementos del array desde *ini* hasta *fin*. Los parámetros *ini* y *fin*, se comportan como en el método anterior. Devuelve un puntero al array modificado. Modifica el array original. Ej. `animales.copyWithin(2, 0);`

Métodos .reverse() y .sort(): cambiar el orden de los elementos de un array

Propiedad	Descripción
.reverse()	Invierte el orden en que están los elementos en un array y devuelve un puntero al array modificado. Modifica el array original. Ej. `animales.reverse();`

Propiedad	Descripción
`.sort([`*`funcion`*`])`	Ordena el array (ordenación alfabética) y devuelve un puntero al array modificado. Modifica el array original. Si se quiere ordenar el array con otro criterio, se puede pasar una función a este método. Ej. `animales.sort();`

Ejemplo: ordenar un array de números, utilizando ordenación numérica:

```
let numeros = [4, 2, 5, 10, 3];
function ordenarNum(a, b) {
      return a - b;
}
numeros.sort(ordenarNum);
console.log(numeros);
// [1, 2, 3, 4, 5]
```

Métodos .indexOf(), .lastIndexOf(), .find()★, findIndex()★ e .includes() ★. Buscar un elemento

Estos métodos devuelven un índice o un elemento del array, **no modifican el array** sobre el que se aplican.

Método	Descripción
`.indexOf(`*`elemento`* **`[,` *`pos`***`])`**	Devuelve el índice de la primera aparición del *elemento* buscado. Si se especifica el parámetro *pos*, empezará a buscar desde esa posición. Si no encuentra el elemento, devolverá -1. Ej. `let pos = animales.indexOf("perro");`
`.lasIndexOf(`*`elemento`* **`[,` *`pos`***`])`**	Como el anterior, pero empezando a buscar desde atrás.
`.find(funcion(`*`valor`* **`[,`*`índice`* **`[,`***`array`***`]])) {}))`**	Devuelve el primer elemento del array que cumple la función de prueba proporcionada. La función se define del mismo modo que para el método `.forEach()`. `let animal = animales.find(valor => valor === "Gato");`

★ *EcmaScript 2015 y 2016 o ES6+ o JS ES6+ o JavaScript 6+.*

.findIndex(funcion(*valor* [,*índice* [,*array*]])) {}))	Devuelve la posición del primer elemento del array que cumple la función de prueba proporcionada. La función se define del mismo modo que para el método `.forEach()`. `let animal = animales.findIndex(valor => valor === "Gato");`
.includes(*elemento*)	Devuelve *true* si el array contiene el *elemento*, *false* en caso contrario. `animales.includes("Gato"); //true`

Métodos .slice(), .concat(), map() y filter. Obtener un array a partir de otro/s

Estos métodos **no modifican el array** sobre el que se aplican.

Método	Descripción

.slice(*inicio* [, *fin*])	Devuelve un subarray desde la posición *inicio* hasta la posición *fin* (no incluida).
	• *inicio*: si es un valor negativo, empieza a contar desde el final del array.
	• *fin:*
	  ○ *si* es un valor negativo, se empieza a contar desde el final del array.
	  ○ si no se especifica *fin*, devuelve desde la posicion *inicio* hasta el final del array.
	Ej.
	`let tresEle = animales.splice(1,3);`

.concat(*lista_arrays*)	Devuelve un array formado por dos o más arrays: une al array inicial la *lista_arrays* separados por comas que se pasan como parámetros. Ej. `let a = animales.concat(["pez", "tigre"]);`

.map(funcion(*valor* [,*índice* [,*array*]])) {}))	Devuelve un nuevo array creado al ejecutar una función sobre cada elemento de array original.
	Ej.
	`let lista = [45, 4, 9, 16, 25];`
	`let lista2 = lista.map(function (valor) { return valor * 2; });`

.filter(funcion(*valor* [,*índice* [,*array*]])) {}))	Devuelve un nuevo array con los elementos que pasen una función prueba.
	Ej.
	`let lista = [45, 4, 9, 16, 25];`
	`let may18 = lista.filter(function (valor) { return valor > 18;});`

Si asignamos una variable de tipo array a otra variable, no se crea una nueva copia del array; es como si se copiara el puntero al contenido del array.

Hay dos formas de **copiar un array en otro**, con los métodos slice() y concat(). Se recomienda utilizar slice() o el operador *spread* … de ES6+, que veremos más adelante:

```
animales2 = animales.slice();    // RECOMENDADO
animales2 = animales.concat();
```

Métodos .reduce(), .reduceRight(), .every() y .some(). Operar con todos los valores del array

Estos métodos **no modifican el array** sobre el que se aplican.

Método	Descripción

.reduce(funcion(*total, valor [,índice [,array]]*)) {}))	Devuelve el valor obtenido al ejecutar una función sobre cada elemento del array con el fin de reducirlo a un solo valor.
	Ej.
	`let lista = [45, 4, 9, 16, 25];`
	`let suma = lista.reduce(function (total, valor) { return total + valor;});`

.reduceRight(funcion(*total, valor [,índice [,array]]*)) {}))	Exactamente igual que el anterior, pero en este caso recorre el array de derecha a izquierda.

.every(funcion(*valor* [,*índice* [,*array*]])) {}))	Devuelve *true* si todos los valores del array pasan una función prueba, en caso contrario *false*.
	Ej.
	`let lista = [45, 4, 9, 16, 25];`
	`let todosMay18 = lista.every(function (valor) { return valor > 18;});`

.some(funcion(*valor* [,*índice* [,*array*]])) {}))	Devuelve *true* si algún valor del array pasan una función prueba, en caso contrario *false*.
	Ej.
	`let lista = [45, 4, 9, 16, 25];`
	`let algunosMay18 = lista.some(function (valor) { return valor > 18;});`

Asignación múltiple, operadores *rest / spread ...*[★]

Asignación múltiple

La **asignación múltiple o desestructuradora** (*destructuring*) nos permite asignar valores de un array o de un objeto, a distintas variables, en una sola orden. Lo que nos permite hacer programas más cortos y legibles.

★ *EcmaScript 2018 o ES6+ o JS ES6+ o JavaScript 6+.*

Lo podremos utilizar en la definición de variables o en la asignación y pueden usar valores por defecto.

- En el caso de los **arrays**, las variables deben agruparse entre corchetes y se relacionan con los valores por posición.
- En el caso de los **objetos**, las variables deben agruparse entre llaves y se relacionan por el nombre de la propiedad.

Ejemplos: arrays

```
//Definición de variables
let [x, y, z] = [5, 1, 3, 4];    // x => 5, y => 1, z => 3
//Asignación de variables: intercambiar contenidos
let x = 5, y = 1;
[x, y] = [y, x]; // x => 1, y => 5
// Valores por defecto/indefinidos
let [x, y, z=1, t=2, v] = [5, , ,10]  // x => 5, y => undefined,
                                      z => 1, t => 10, v => undefined
```

Ejemplos: objetos

```
//Definición de variables
let {a, c=1, d, e} = {a:5, e:3, f:4}; // a => 5, c => 1,
                                        d => undefined, e => 3
//Asignación de variables
let a, c, d;
({a, c=1, d} = {a:5, e:3}); // a => 5, c => 1, d => undefined
```

La asignación múltiple en el segundo caso debe ir entre paréntesis por un problema de análisis sintáctico de JavaScript, ya que las {} sirven para agrupar bloques además de para crear literales de objetos.

En los ejemplos con objetos, hemos visto que podemos **agrupar variables** cuando las propiedades de un objeto y las variables se denominan del mismo modo. Es decir:

```
let a=5, c=3, d=4;
// ES5-agrupar variables en un objeto con propiedades de igual
nombre // a las variables:
let objES5 = {a:a, c:c, d:d};    // objES5 => {a:5, c:3, d:4}
// ES6-las mismas variables se agrupan así:
let objES6 = {a, c, d};    // objES6 => {a:5, c:3, d:4}
```

Operador *rest* …

El **operador rest u operador de agrupación …**, agrupa en un array u objeto, el resto de los elementos asignados de una lista. El operador rest debe ir al final y funciona exactamente igual que en los parámetros por exceso al definir funciones que ya conocemos.

Ejemplos: arrays

```
let [x, ...rest1] = [0, 2, 3];   // x => 0, rest1 => [2, 3]
[x, ...rest2] = [4, x, ...rest1];  // x => 4, rest2 => [0, 2, 3]
```

Ejemplos: objetos

```
let {a, ...x} = {a:5, b:1, c:2}; // a => 5, x => {b:1, c:2}
let a, x;
({a, ...x} = {a:1, b:2});  // a => 1, x => {b:2}
```

Operador *spread* …

El **operador *spread* u operador de propagación …**, esparce los elementos de un array u objeto en otro. Permite, en función de sobre qué elemento de JavaScript se aplique:

- A elementos iterables (array, cadena…) ser esparcidos donde se esperan cero o más argumentos (para llamadas de función) o elementos (para Arrays literales).
- A un objeto ser esparcido en lugares donde se esperan cero o más pares de valores clave (para literales de tipo Objeto).

En otras palabras, y con un ejemplo, si en una función queremos pasar varios parámetros y esos parámetros están recogidos en un array, no podemos pasar el array, pero esta sintaxis nos permite que ese array se convierta a valores “sueltos” para poderlos pasar como parámetros de la función.

Ejemplos: arrays

```
let a = [2, 3];
let b = [0, 1, ...a]; // b => [0, 1, 2, 3]
f(0, ...a);      // => f(0, 2, 3)
```

Ejemplos: objetos

```
let x = {a:5, b:1};
let y = {...x, c:6, d:7};  // y => {a:5, b:1, c:6, d:7}
```