Estructuras definidas por el usuario en JavaScript.

Arrays

https://lenguajejs.com/javascript/arrays/que-es/

Javascript es un lenguaje súper permisivo. Por eso los **arrays en Javascript** no pueden ser menos. Y es que en un mismo array podemos incluir todo tipo de datos: cadenas, números, objetos…

A la hora de **crear** un **array** podemos hacerlo de dos maneras: **con corchetes** o **con la palabra *new*** (no aconsejado). Y a la hora de **acceder a los elementos** que contiene, podemos hacerlo utilizando los **corchetes** y, en su interior, el número de la posición empezando por cero.

También podemos hacer otras operaciones básicas como **recorrer el array** desde el primer al último elemento, **procesando los valores** uno a uno o **mediante instrucciones de visualización** de datos (***alert***, ***console.log***, etc.).

Podemos utilizar la **propiedad *length***, que nos permite conocer la **longitud del array**

Vamos a ver los **métodos más importantes** que podemos encontrar:

métodos de ARRAYS

Metodo Map (aplica una función a cada elemento del array y devuelve un array con el resultado, la función recibe como parámetros el valor actual, su índice y el propio array)

Método Foreach (aplica una función a cada elemento del array, la función recibe como parámetros el valor actual, su índice y el propio array)

- **Métodos para añadir, extraer o borrar elementos:**
  - ***pop*** y ***shift*** extraen el último y primer elemento del array, respectivamente.
  - ***delete*** elimina el elemento (pero deja el hueco).
  - ***push*** y ***unshift*** añaden elementos al final y al principio, respectivamente.
  - ***splice***: permite insertar elementos, machacando o no los elementos existentes.
  - ***slice***: permite extraer un subarray.
  - ***concat***: une un array con otro.
  - ***copyWhithin***: copia elementos de un array y los sustituye por otros elementos del array.
  - ***fill***: sustituye los elementos del array por uno indicado.
- **Métodos para buscar:**
  - ***indexOf*** y ***lastIndexOf***: devuelve la primera o última posición de un elemento respectivamente.
  - ***includes***: devuelve true si el elemento se encuentra en el array
- **Métodos para ordenar e invertir el orden:**
  - ***reverse***: invierte el orden de un array.
  - ***sort***: ordena el array. Ver Compare function:
    - MDN web DOCs
    - W3Schools

```
<!DOCTYPE html>
<html>
<title>Lo que sea</title>
<body>
<script>
    //ARRAY: almacena en una misma variable múltiples valores: valores primitivos, objetos, etc. Se referencian con un índice numérico.
    //¡Ojo! Un objeto se referencia con un nombre

    //OPERACIONES BÁSICAS CON UN ARRAY:

    //Crear un array: var nombreArray = [<valores separados por comas>];
     var animales = ["Perro", "Gato", "Hamster"];
    //var nombreArray = new Array(<valores separados por comas>);
     var animales2 = new Array("Perro", "Gato", "Hamster");

     //Acceso a elementos de un array: nombreArray[<índice>]; Desde el 0
```

```
     var miAnimal = animales[0];

     //Mostrar todo el array: con una instrucción de mostrar
     alert(animales);
     document.write(animales);

     //PROPIEDADES:
     //length: devuelve la longitud del array (número de elementos)
     document.write("<br/>La longitud del array es: "+animales.length);

     //Mostrar los valores del array uno a uno
    document.write("<br/><br/>Todos los elementos:")
     for (var i=0; i<animales.length; i++){
        document.write("<br/>"+animales[i]);
     }

     //MÉTODOS:
     //Array.isArray(<nombreArray>): devuelve true si es un objeto de tipo
Array. Si pusamos typeof <nombreArray> devolverá object.
     document.write("<br/>¿Es un array? "+Array.isArray(animales));
     //<nombreArray> instanceof Array: devuelve true si es un Array.
     document.write ("<br/>¿Seguro? "+(animales instanceof Array));

     //MÉTODOS PARA MOSTRAR EL ARRAY:
     //toString(): convierte el array a cadena
     document.write("<br/>El array en tipo String es:
"+animales.toString());

     //join("<separador>"): convierte el array a cadena separado por el
separador:
     document.write("<br/>El array con join es: "+animales.join(" * "));


     //MÉTODOS PARA AÑADIR, EXTRAER O BORRAR ELEMENTOS:
     //pop() y shift(): extrae el último y primer elemento respectivamente y
lo guarda en una variable (si queremos)
     var ultimo = animales.pop();
     document.write("<br/>Después de sacar "+ultimo+" quedan
"+animales.toString());

     //delete nombreArray[<indice>]: elimina el elemento y lo transforma a
undefined

     //push(<elemento>) y unshift(<elemento>): añade un elemento al final y
al principio del array respectivamente
     animales.push("Jilguero");
     document.write("<br/>Después de meter Jilguero quedan
"+animales.toString());
```

```
     //Nota: podemos añadirlo con animales[<índice>] = <elemento>; pero hay
que tener cuidado de no sobreescribir o dejar huecos.

     //splice(<posicion insertar/borrar>, <elementos a borrar>, [<elementos
a añadir separados por comas>]):
     animales.splice(2,1,"Vaca","Toro");
     document.write("<br/>Después de meter 2 y borrar 1 quedan
"+animales.toString());
     //animales.splice(0,1): eliminaría el primer elemento.

     //slice(<ini>[,<fin>]): devuelve un subarray desde la posición indicada
hasta (sin incluir) la final (no es obligatorio)
     var subarray = animales.slice(1,3);
     document.write("<br/>El subarray entre 1-3 es: "+subarray.toString());

     //concat(<lista de arrays separados por comas>): une el array inicial
con un segundo array
     var masAnimales = ["Burro", "Mula"];
     var todos = animales.concat(masAnimales);
     document.write("<br/>El array completo es: "+todos.toString());

     //copyWithin(): copia elementos del array y los sustituye por otros
elementos del array.
     //fill("<elemento>"): sustituye los elementos del array por el
indicado.

     //MÉTODOS PARA BUSCAR:
     //indexOf(<elemento>[,<pos>]) y lastIndexOf(<elemento>[,<pos>]):
devuelve la primera o última posición de un elemento respectivamente;
podemos pasarle a partir de qué elemento buscará.
     document.write("<br/>La primera posición de Toro es:
"+animales.indexOf("Toro"));
     document.write("<br/>La última posición de Toro es:
"+animales.lastIndexOf("Toro"));

     //MÉTODOS PARA ORDENAR E INVERTIR EL ORDEN
     //reverse(): invierte el orden de un array
     animales.reverse();
     document.write("<br/>El array después de invertir es:
"+animales.toString());
     //sort(): ordena el array
     animales.sort();
     document.write("<br/>El array después de ordenar es:
"+animales.toString());
     //¡Ojo! Los números almacenados como cadenas se comparan caracter a
caracter, no como cifras.

</script>
```

```
</body>
</html>
```

Spread operator u operador de propagación

Operador spread u **operador de propagación**, también conocido como **sintaxis extendida** o ***spread sintax***.

Esta sintaxis permite, en función de sobre qué se aplique:

- A **elementos iterables** (array, cadena…) **ser expandidos** donde se esperan cero o más argumentos (para llamadas de función) o elementos (para Arrays literales).
- A un **objeto ser expandido** en lugares donde se esperan cero o más pares de clave-valor (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.

La sintaxis de cada caso sería la siguiente:

```
// Para arrays literales o cadenas
[...objetoIterable, 'a', "caracola", 7];

// Para llamadas a funciones
miFuncion(...objetoIterable);

// Para literales de tipo Object
let clon = {...obj};
```

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>ES6 - Operador de propagación (spread)</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1">
```

```
  <script src="09_ES6_SpreadOperator.js"></script>
</head>
<body>
    <h1>ES6 - Spread operator / Operador de
propagación</h1>
</body>
</html>
```

```
/* La sintaxis extendida o spread sintax permite:
- A elementos iterables (array, cadena...) ser expandidos
donde se esperan
cero o más argumentos (para llamadas de función) o
elementos (para Array literales)
- A un objeto ser expandido en lugares donde se esperan
cero o más
pares de valores clave (para literales de tipo objeto)*/

//SINTAXIS
//Para arrays literales o cadenas
//[...objetoIterable, 'a', 'caracola', 7];

//Para llamadas a funciones
//miFuncion(...objetoIterable);

//Para literales de tipo Object
//let clon = { ...obj };

//ARRAY
console.log(Math.max(3,1,7)); //Devuelve 7
let array = [3,1,7];
console.log(Math.max(array)); //Devuelve NaN
console.log(Math.max(...array)); //Devuelve 7
let array2 = [2,6,8];
console.log(Math.max(...array, 5, ...array2, 4));
//es igual a:
console.log(Math.max(array[0],array[1],array[2],
5,array2[0],array2[1],array2[2], 4));
```

```
//Devuelve 8

// Concatenar dos arrays en uno
let arrayResultante = [...array, ...array2];

//Copiar un array en otro array, clonariamos el array2,
podriamos también utilizar el metodo .slice()
let arrayCopia = [...array2];

//CADENAS
let saludo = "Hola, caracola";
console.log([...saludo]);

//FUNCIONES
function suma (a, b, c){
    return a + b + c;
}
const valores = [1, 3, 5];
console.log(suma(...valores));

//OBJETOS
let persona1 = {nombre: "Ada", nacimiento: 1815};
let persona2 = {nombre2: "Charles", nacimiento2: 1791};
//Tenemos que cambiar los nombres de los elementos si no
queremos que se sobreescriban

let clonAda = {...persona1}; //Modo correcto de crear una
copia de un objeto
console.log(clonAda);

let adaCharles = {...persona1, ...persona2};
console.log(adaCharles);
```

Destructuring o desestructuración

La **desestructuración** o ***destructuring*** es una interesantísima característica que se incorporó a Javascript en su versión **ES6** con el objetivo de **descomponer un array o un objeto en elementos independientes y asignarlos a variables o constantes en una única expresión**.

Es decir, nos permite dividir un array o un objeto en *trocitos* y pasárselos a variables o constantes sin necesidad hacerlo de elemento a elemento en sentencias separadas.

A continuación veremos cómo trabajar con:

- Desestructuración de arrays con asignación básica.
- Desestructuración de arrays con asignación separada de la declaración.
- Desestructuración de objetos con asignación básica.
- Desestructuración de objetos asignando a nuevos nombres de variables.
- Desestructuración de objetos con asignación sin declaración.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>ES6 - Destructuring (desestructuración)</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1">
  <script src="16_ES6_Destructuring.js"></script>
</head>
<body>
    <h1>ES6 - Destructuring (desestructuración)</h1>
</body>
</html>
```

```
/* DESTRUCTURING
Destructuring es un modo de extraer valores de datos
almacenados en objetos y arrays,
descomponiéndolos y asignándolos a un grupo de variables.
Mediante patrones se pueden especificar cómo extraer los
valores. */
```

```
//Destructuring de arrays (asignación básica)
const galicia = ["A Coruña", "Lugo", "Ourense",
"Pontevedra"];
const [c, lu, ou] = galicia;
console.log(c);
console.log(ou);

//Destructuring de arrays (asignación separada de la
declaración)
let cc, ba;

[cc, ba="Merida"] = ["Cáceres", "Badajoz"];
console.log(cc);
console.log(ba);

//Destructuring de objetos (asignación básica)
const varios = {p: 11, q: true, r: "Hola" };
const {p,r} = varios;
console.log(p);
//console.log(q);
console.log(r);

//Destructuring de objetos (asignando a nuevos nombres de
variables)
const objeto = {nombre: "Ada", apellido: "Lovecode"};
//Tomamos del objeto la propiedad llamada nombre y la
asignamos a la variable n
```

```
const {nombre: n, apellido: a} = objeto;
console.log(n + " "+ a);
//Tomamos del objeto el valor del nombre y del apellido
const {nombre, apellido} = objeto;
console.log(nombre + " " + apellido);

//Destructuring de objetos (asignación sin declaración)
let x, y;
({x, y} = {x: 1, y: 2}); //Es obligatorio el paréntesis en este tipo de asignación
console.log(x);
console.log(y);
```

https://openwebinars.net/academia/aprende/react/6053/

Set y Map

https://lenguajejs.com/javascript/set-map/que-es-set-weakset/

Clases

https://lenguajejs.com/javascript/oop/clases/

Las **clases** en Javascript se introdujeron en la versión (ECMAScript2015 o 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" (entre comillas), en esta versión se utiliza ***class***; además, las propiedades se deben indicar dentro de un método **constructor**. La sintaxis sería la siguiente:

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

Y cómo no, para definir un **objeto de la clase**, utilizamos la palabra reservada ***new***.

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

```
/*
Las clases son una mejora sintáctica sobre la herencia basada en
prototipos de JavaScript.
- Ofrecen una sintaxis más simple para crear objetos.
- No utiliza la palabra function, sino la palabra class.
- Las propiedades se asignan en un método constructor(), no en la clase
en sí.
*/

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

```
}

let miTelefono = new Telefono ("Google", "Pixel");
console.log(miTelefono.marca + " " + miTelefono.modelo);
```

Clases. Herencia

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 definiría la relación de esta segunda clase con la primera. Además, para hacer referencia a las propiedades de la clase *padre* utilizamos la palabra *super.*

Brevemente, la sintaxis para definir una cabecera de una clase *hijo* sería la siguiente:

```
class ClaseHijo extends ClasePadre
```

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>ES6 - Herencia de clases</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <script src="11_ES6_ClasesHerencia.js"></script>
</head>
<body>
    <h1>ES6 - Herencia de clases</h1>
    <div id="mensaje"></div>
</body>
</html>

/*
Para crear herencia de clases hay que utilizar la palabra extends.
Una clase creada con herencia hereda todos los métodos de la otra clase.
*/
window.addEventListener("load",function() {
    class Telefono {
        constructor (marca){
            this.marca = marca;
        }
        anuncio () {
            return "Ha llegado el nuevo teléfono de " + this.marca;
        }
    }
    class Modelo extends Telefono {
        constructor (marca, modelo){
            super(marca);
```

```
            this.modelo = modelo;
        }
        anuncioCompleto(){
            return super.anuncio() + ": el modelo " + this.modelo;
        }
    }
    let miTelefono = new Modelo ("Google", "Pixel");
    mensaje.innerHTML = miTelefono.anuncioCompleto();

},true);
```

Clases. Métodos get y set

Continuamos con clases en Javascript y en esta lección vamos a ver cómo crear métodos *get* y *set* (o también llamados *getters* y *setters*) que incluir en nuestras clases.

Estos métodos nos permiten 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 guión bajo para nombrar la propiedad.

Utilizar el método get o set no es obligatorio en Javascript, simplemente declarando las propiedades en el constructor ya serviría, pero sí se utilizará en el caso de propiedades computadas .

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>ES6 - Getters y Setters en clases</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1">
  <script src="12_ES6_ClasesGetSet.js"></script>
</head>
<body>
    <h1>ES6 - Getters y Setters en clases</h1>
    <div id="mensaje"></div>
</body>
</html>
```

```
/*
  Los métodos get y set se utilizan para asignar y extraer
atributos de un objeto.
  Es muy importante tener en cuenta que el nombre de los
getters/setters no puede ser
  el mismo que la propiedad porque se produciría un bucle:
al acceder a la propiedad
  invocaríamos al método que a su vez accede a la propiedad
que invoca al método...
  Por ello, muchos desarrolladores utilizan el guión bajo
para nombrar la propiedad.
*/
window.addEventListener("load",function() {
  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;
},true);
```

Clases. Métodos estáticos (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.

Para crear un método estático no hay más que utilizar la siguiente sintaxis:

```
static nombreMetodo (parametros) { //código }
```

```
<!DOCTYPE html>
<html lang="es">
<head>
  <title>ES6 - Métodos estáticos en clases</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <script src="13_ES6_ClasesStatic.js"></script>
</head>
<body>
    <h1>ES6 - Métodos estáticos en clases</h1>
    <div id="mensaje"></div>
</body>
</html>
```

```
/*
Utilizamos static para definir un método estático en una clase.
Al igual que en otros lenguajes de programación, un método estático se
llama diréctamente
sin instanciar la clase (de hecho, no puede hacerse mediante una
instancia de clase).
Se suelen utilizar para crear funciones útiles en una aplicación.
No es necesario crear un objeto para llamar a un método estático.
*/
window.addEventListener("load",function() {
  class Rectangulo {
    constructor(x, y){
      this.x = x;
      this.y = y;
    }
    static area ( a, b) {
      return a * b;
    }
    static perimetro (a, b){
      return a + a + b + b;
    }
  }
  let rectangulo1 = new Rectangulo (2,3);
  //console.log(rectangulo1.area(2,3));
  //console.log(rectangulo1.perimetro(2,3));
  mensaje.innerHTML = Rectangulo.perimetro(2,3);

},true);
```

Prototype

https://www.youtube.com/watch?v=6gmYapa-KBY

https://www.freecodecamp.org/espanol/news/prototipo-javascript-expliacado-con-ejemplos/

Básicamente y de manera muy resumida el prototype de un objeto es una forma de añadir propiedades o métodos a todas las instancias de una clase de objetos (sobre todo para objetos de los que no deseamos o no podemos modificar su constructor).

Módulos

https://lenguajejs.com/javascript/modulos/que-es-esm/

Característica aparecida en JavaScript ES6, para poder utilizar módulos, la página ha de ser abierta en un servidor web (por ejemplo la extensión live-server en Visual Studio) para que no de error de cross origin.

Los módulos son trozos de código que podemos tener en fichero .js independientes y que después podemos importar a otro fichero js
Más información:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules

export

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Statements/export

import

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Statements/import

Resumen:

!!! Ejecutar la página en un servidor WEB¡¡¡ (P.Ej Extensión live-server de VStudio)

Añadir el type “module” en el fichero que va a importar otro módulo javaScript:

En este caso principal.js va a importar exportado.js entonces pondremos lo siguiente en el HTML:

```
<script src="principal.js" type="module"></script>
```

En el módulo que vamos a exportar(por ejemplo exportado.js) añadir la palabra export con cada elemento que queremos exportar:

```
export const hello = () => "hello"
export default class miclase{
goodby() {
     return "goodby"
}
```

solo podemos poner un default por defecto.
También ponerlos todos juntos al final de la siguiente manera:

```
export  {hello, miclase}
```

Después podemos importar desde el archivo principal.js añadiendo al principio del archivo de manera individual cada elemento exportado, el default podemos importarlo con cualquier nombre en este caso micalseImportada y el resto hay que ponerles el nombre que tiene original entre llaves:

```
import miclaseImportada, {hello} from "./exportado.js"
```

o todo el contenido de la siguiente forma:

```
import * as exportado from './exportado.js';
```

Y utilizarlo de la manera `exportado.hello` dentro del código del archivo que importa las funciones en este caso.

NOTAS:
Si el archivo js es type=”module” el HTML se carga antes que el .js
No se pueden gestionar los eventos directamente en el HTML en archivos de tipo “module”
La página tiene que ejecutarse en un servidor si no dará un error.

Curso Javascript Moderno (ES6) - #15. Módulos. Import y export