Asíncronía:

setTimeOut () →
Ejecuta cierto código según la cantidad de tiempo establecido. Sólo se ejecuta una vez.

```
setTimeOut( () => {
----Código  a ejecutar---

}, 1000);
```

setInterval () →
Se ejecuta continuamente según la cantidad de tiempo establecido.

```
setInterval ( () => {
----Código  a ejecutar---

}, 1000);
```

Canceladores de setTimeOut y setInterval

```
clearTimeOut(temporizador);
clearInterval(temporizador);
```

Código Sincrono →
Se ejecuta en el momento.
Código Asincrono →
Se ejecuta en el futuro luego de que se ha ejecutado el código sincrono, hasta entonces queda en espera en la "Cola de callbacks".

Un ejemplo de código asíncrono son las funciones setTimeOut() y setInterval()

Callback() →
Se utiliza para tener un control en el orden de la asicronía.
Se trata de una función que llama a otra función (asíncrona) dentro de ella.
Puede dar lugar a problemas si se encadenan demasiadas.

Promesas →
Son una alternativa a las **callbacks** ya que mediante el método 'then()' podemos encadenar varias funciones asíncronas, así como tambíen tener un control de errores más ordenador mediante el 'catch()'
Las promesas constan de 2 parámetros 'resolve' y 'reject' que se pasan dentro de una arrow function.

Ejemplo:

```
Function cuadradoPromise(value){

        if(typeof value!== 'number'){
        return Promise.reject('No válido');
        }

        return new Promise((resolve, reject) => {

        setTimeOut(()=>{

                resolve({
                        value,
                        result: value * value
                        })

        }, 1000);

        });

}
```

Esta promesa devuelve el objeto resolve o si hay un error reject.

```
cuadradoPromise(0).
then(obj => {

console.log(obj.value, obj.result);
return cuadradoPromise(1);
}).
then(obj =>{
---Código a ejecutar---
}).
catch(error){
}
```

El método then coge el objeto devuelto por la promesa y ejecuta lo que haga falta.

Puede ejecutar otra vez la función y retornar su valor a otro then()

Muestra los errores que se producen en cualquiera de los then()

async await

- Es una manera más limpia de ejecutar el código. Evita el uso de la concatenación de 'then()'
- Se utiliza la palabra reservada '**async**' al declarar o expresar la función y '**await**' en la función que vayamos a ejecutar dentro de ésta.

```
async  funcionDeclarada(){

        try{

        let obj = await cuadradoPromise(0);
        console.log(obj.value, obj.result);

        obj = await cuadradoPromise(1);
        console.log(obj.value, obj.result);

        }catch(error){
        console.log(error);
        }

}

const funcionAsincronaExpresada = async () => {
        try{
                ---Código a ejecutar---
        }catch(error){
                ---Código a ejecutar---
        }

}
```

Symbols

1. Son nuevo tipo de dato primitivo a partir de ES6.
2. Normalmente utilizados para propiedades privadas en objetos.
3. Tienen una referencia única (Identificadores únicos)
4. Su valor siempre es el mismo por lo tanto es una buena práctica declararlo con CONST.
5. Al recorrer un objeto o imprimir una clase no se mostrarán.
6. Para saber los symbols que tiene un objeto se utiliza 'Object.getOwnPropertySymbols(objeto)'

```
const persona = {

[nombre]: 'frank',
edad: 15
}
```


Con el corchete se da a entender que es un symbol

7. Si se pone otra propiedad con el mismo nombre se utiliza el punto (.) para acceder a ella y los corchetes para el symbol.

Sets

Colección de datos únicos, no permite duplicados.

Declaración:

const set = new Set()

propiedades y métodos:
size()
add()
clear()
delete()
has()

*Para acceder a un valor especifico hay que pasarlo a array.

const arr = Array.from(set) → arr[0]

Maps

Colección de claves:valor. Tiene la peculiaridad que la llave puede ser una cadena, un number, objeto, etc.

Propiedades:
size
set()
get()
delete()
has()
keys()
values()

Generators => Ver el video 54

Proxies (Sólo para objetos literales) (Video 55)

Hace una vinculación entre el objeto original y su copia, además a través de su manejador (handler) se encarga de hacer validaciones como por ejemplo no permitir que se añadan nuevas propiedades al objeto (Al estar vinculado con la copia, si se añade en la copia se añade en el original)

```
const persona = {

nombre = ‘’,
apellido= ‘’,
edad = 0

}
```

```
const manejador = {
set(objeto, propiedad, valor){
--Se hacen las validaciones--
if(Object.keys(objeto)).indexOf(propiedad) ≡== -1){
        ---Código a ejecutar---
}
objeto[propuedad] = valor;
        }
}
```

const frank = new Proxy(**persona**, **manejador**);

frank.nombre = ‘Frank’;
frank.apellido = ‘Ruiz’;
frank.edad = 20;

Propiedades dinámicas de los objetos → (Video 56)

this() → (Video 57)

Hace referencia al propio objeto y su scope (ámbito)

Arrow functions →
No tienen contexto propio, lo cogen del padre o del contexto en que ha sido creado.

Funciones anónimas →
Tienen su propio contexto y si no encuentran alguna variable la buscan en contexto global

*Tener cuidado con esto al utilizar el atributo type = ‘module’ en el script del html.

* ‘this’ siempre va ha hacer referencia a su ámbito, ya sea dentro de un objeto, función, etc.

call, apply, bind → (Video 58)

Se utiliza para vincular contextos en funciones.

call →
Los parámetros de la función se pasa normalmente.
apply →
Los parámetros se pasan en un array

this.lugar = ‘Contexto global’

const obj = {
lugar: ‘contexto del objeto’
}

functions saludar (nombre){
console.log(`saludos ${nombre}` desde ${this.lugar});
}

saludar.call(obj, ‘frank’);
saludar.apply(obj, [‘frank’]);

bind →
Se utiliza para vincular distintos contextos.

Ejemplo:

```
const persona = {
        nombre:'jhon',
        saludar: function(){
                console.log(`hola ${this.nombre}`)
        }
}

const otraPersona = {
        saludar: persona.saludar.bind(this)  → vincula el contexto global
        saludar2: persona.saludar.bind(otraPersona) → lo vincula al objeto otraPersona
}
```

**** Solamente 'this' hace referencia al contexto global en sí, si se agrega una propiedad (this.nombre) puede buscar en el ámbito del objeto o en el ámbito global si se trata de una Arrow function***

JSON

parse() →
Al pasarsele una cadena válida en formato JSON puede convertirlo a objeto de javascript.

Stringify() →
Al pasarsele un valor string, number, object, etc. Lo convierte a una cadena válida de formato JSON.

DOM
Métodos para traer elementos del DOM

Los siguientes métodos ya han dejado de utilizarse, pero aún se pueden encontrar y emplear.

document.getElementByTagName(‘li’);
document.getElementByClassName(‘card’);
document.getElementByName(‘nombre’);

Métodos aún utilizados.

document.getElementById(‘menu’);
→ No es necesario añadirle el # al inicio de la cadena que haga referencia a un id.

document.querySelector(‘a’);
→ Devuelve la primera ocurrencia válida.

Document.querySelectorAll(‘a’);
→ Devuelve una nodelist de todas las ocurrencias válidas.

→ En cualquiera de los querySelector si queremos hacer referencia a un id(#) o una clase (.) tenemos que añadirle respectivamente antes del nombre el hash o el punto.

Obtener atributos de los elementos
document.documentElement.lang
document.documentElement.getAttribute(‘lang’);

document.querySelector(‘.link-dom’).href
document.querySelector(‘link-dom’).getAttribute(‘’href)

→ El getAttribute(‘lang’) sólo devuelve el valor del atributo, el (.) puede traer algo diferente de lo esperado.
→ **El getAttribute en la forma más aconsejable de obtener los atributos**

Modificar, añadir y eliminar Atributos

document.documentElement.lang = ‘en’;
document.documentElement.setAttribute(‘lang’, ‘es-es’);

→ Poner en variables los elementos del DOM que vayamos a utilizar.
→ Una buena práctica es que lleven el símbolo del dolor delante ($) para diferenciar de nuestras variables normales.

const $linkDOM = document.querySelector(‘link-dom’);
$linkDOM.setAttribute(‘target’, ‘_blank’); → ***Añade un atributo***
$linkDOM.setAtribute(‘href’, ‘https://youtube’) → ***Modifica atributo***
$linkDOM.removeAttribute(‘href’); → ***Elimina atributo***

$linkDOM.hasAttribute(‘href’) → ***Devuelve true o false si existe o no el atributo***

<u>Data-Attributes</u>

→
Atributos creados por nosotros mismos, llevan la palabra ‘data’ seguida de un guion ‘-’ y a continuación el nombre.

→
Se almacenan en un mapa llamando ‘dataset’

$linkDOM.getAttribute(‘data-description’);
$linkDOM.dataset
$linkDOM.dataset.description → Se obtiene sin necesidad de poner ‘data-’
$linkDOM.setAttribute(‘data-description’, ‘valor’);
$linkDOM.dataset.description = ‘Hola’;
$linkDOM.hasAtrribute(‘data-id’);
$linkDOM.removeAttribute(data-id);

<u>DOM CSS</u>

const $linkDOM = document.querySelector(‘.link-dom’)

Formas de acceder a las propiedades de un elemento

$linkDOM.style
$linkDOM.getAttribute(‘style’);
$linkDOM.style.backgroundColor ← Propiedades que han sido declaradas en línea en el elemento
$linkDOM.style.color

getComputedStyle($linkDOM).getPropertyValue(‘color’);
→
Obtiene solo el valor de la propiedad dada

<u>Establecer una propiedad en un elemento</u>

$linkDOM.style.setProperty(‘text-decoration’, ‘none’);
$linkDOM.style.textAlign = ‘center’;

<u>Variable CSS – Custom Properties CSS</u> (Video 64)

Clases CSS

const $card = document.querySelector(‘.card’);

$card.className → Nombre de la clase
$card.classList → Devuelve una lista con todas las clases del elemento
$card.classList.contains(‘rotate-45’) → Comprueba si existe la clase
$card.classList.remove(‘rotate-45’)
$card.classList.toggle(‘nombreClase’) → Añade o elimina dependiendo de si existe o no la clase
$card.classList.add(‘nombreClase’)
$card.classList.replace(‘Valor a reemplazar’, ‘valor nuevo’)

Interactuar con código textual o HTML

const $whatIsDOM = document.getElementById(‘que es’);

let text =``

```
<p> ….. </p>
<p> ….. </p>
<p> ….. </p>
<mark> … </mark>
```

`;

$whatIsDOM.textContent = text;
→
Reemplaza el contenido del elemento, pero tal cual con el código html (no lo reconoce)

$whatIsDOM.innerHTML = text;
→
Reemplaza y reconocer el HTML.

$whatIsDOM.outerHTML = text;
→
Reemplaza todo el elemento por el nuevo contenido.

Recorriendo el DOM

const $cards = document.querySelector(“.cards”);

$cards.children → Devuelve los hijos del elementos
$cards.children[0] → Muestra un hijo en concreto
$cards.parentElement → Muestra el elemento padre
$cards.firstElementChild → Primer hijo
$cards.lastElementChild → último hijo
$cards.previousElementSibling → Busca el elemento hermano anterior
$cards.nextElementSibling → Busca el elemento hermano posterior
$cards.closest(‘div’) ↘ Busca el elemento indicado entre los ancestros más cercanos
$cards.children[0].closest(‘section’)

Crear elementos en el DOM

Se utilizan las siguientes funciones, las cuales tendrán como parámetros el nombre de la etiqueta a utilizar o el texto que tendrán (createTextNode)

```
const $figure = document.createElement('figure');
const $img = document.createElement('img');
const $figcaption = document.createElement('figcaption');
const $figcaptionText = document.createTextNode('Animals');
```

Ahora se establecen los atributos

```
$img.setAttribute('src', 'http://…');
$img.setAttribute('alt', 'Animals');
```

Se añade la clase (para CSS)

```
$figure.classList.add('card');
```

Se añaden los elementos o texto a las etiquetas

```
$figcaption.appendChild($figcaptionText);
$figure.appendChild($img);
$figure.appendChild($figcaption);
```

Buscamos el elemento del DOM al que se añadirá

```
$cards = document.querySelector('.cards');
$cards.appendChild($figure);
```

Crear varios elementos con fragmentos

Tiene un mejor rendimiento

```
const meses = ['E', 'F', 'M', 'A', 'M', 'J', 'J', 'A', 'S', 'O', 'N', 'D']

$ul3 = document.createElement('ul');
$fragment = document.createDocumentFragment();

meses.forEach(el => {

const $li = document.createElement('li');
$li.textContent = el;
$fragment.appendChild($li);

});

$ul3.appendChild($fragment);
document.body.appendChild($ul3);
```

Los varios elementos se van añadiendo al 'fragmento' de manera el que el DOM no se ve afectado

Template y fragmentos

El template es básicamente una plantilla de la cual nos guiaremos para crear elementos de forma dinámica a través de JS (El template se hace en el html)

index.html

```
<template id=’template-card’>
        <figure class=’card’>
        <img>
        <figcaption></figcaption>
        </figure>
</template>
```

archivo.js

```
const $cards = document.querySelector(‘.cards’);
$template = documente.getElementById(‘template-card’).content;
$fragment = document.createDocumentFragment();

cardContent = [
{ img: ‘’,
 title:’’.}
---más propiedades---
]

cardContent.forEach((el)=>{

$template.querySelector(‘img’).setAttribute(‘src’, el.img);
$template.querySelector(‘figcaption’).textContent = el.title;
```

**** Se clona el template para poder volver a utilizarlo***

```
let $clone = document.importNode($template, true);
let $fragment.appendChild($clone);
})

$cards.appendChild($fragment);
```

Modificar elementos del DOM (OLD)

```
const $cards = document.querySelector(‘.cards’);
const $newCard = document.createElement(‘figure’);

$newCard.innerHtml = `
        <img --atributos--->
`;
```

$newCard.classList.add(‘card’); → Se añade la clase

Reemplazar nodo

```
$cards.replaceChild($newCard, $cards.children[2]);
```

Eliminar

```
$cards.removeChild($cards.lastElementChild)
```

Insertar antes

```
$cards.insertBefore($newCard, $cards.firstElementChild)
```

Clonar nodo

*No confundir con el ***import*** anteriormente utilizado, ese se utiliza para importar un nodo desde el HTML.

```
$const $cloneCards = $cards.cloneNode(true);
document.body.appendChild($cloneCards);
```

<u>Modificar elementos del DOM (Cool Style)</u>

Posiciones

beforebegin → Hermano anterior
afterbegin → primer hijo
beforeend → último hijo
afterend → hermano siguiente

insertAdjacent

.insertAdjacentElement(position, elemento)
→
Inserta un nuevo elemento en la posición indicada

.insertAdjacentHTML(position, HTML)
→
Inserta código HTML
.insertAdjacentText(position, text)
→
Inserta texto

```
const $cards = document.querySelector('.cards')
const $newCard = document.createElement('figure');

let $contentCard = `
        <img src= '….' alt='….'>
        <figcaption></figcaption>
`;

$cards.insertAdjacentElement('afterbegin', $newCard);

$newCard.insertAdjacentHTML('beforeend', $contentCard)
$newCard.querySelector('figcaption').insertAdjacentText('afterbegin', 'any');
```

Más métodos utilizados

$cards.prepend($newCard) → primer hijo
$cards.append($newCard) → último hijo
$cards.before($newCard) → hermano anteriormente
$cards.after($newCard) → hermano siguiente

Manejadores de eventos (listeners)

Se pueden encontrar de tres tipos

Evento con el atributo HTML
→ Se coloca la función en el atributo HTML

Evento con manejador semántico
→ Sólo puede soportar un evento a la vez, si se ponen varios se sobreescriben con el último declarado. (.onClick)

Evento con manejador múltiple
→ Puede soportar varias funciones a la vez (addEventListener)

HTML

```
<button onclick =’holaMundo()’> Evento HTML </button>
<button id =’evento-semantico’> Evento semantico </button>
<button id = ‘evento-multiple’> Evento múltiple </button>
<script src=’dom.js’> </script>
```

dom.js

```
function holaMundo(){
        alert(‘Hola mundo’);
}

const $eventoSemantico = document.getElementById(‘evento-semantico’);
const $eventoMultiple = document.getElementById(‘evento-multiple’);


$eventoSemantico.onclick = holaMundo → No se ponen los parentesis de la funcion, si no se
                                       ejecuta de inmediato.

$eventoSemantico.onclick = function (){ → Sobreescribe a la función anterior.

        alert(‘Hola a todos’);
}


$eventoMultiple.addEventListener(‘click’, holaMundo);

$eventoMultiple.addEventListener(‘click’, ()=>{
        console.log(‘hola’);
});
```

*Recibe 2 parámetros, el primero es el nombre del evento y el segundo el evento (función) a ejecutar.

Eventos con parámetros

Normalmente las funciones que se le pasan a los eventos no contienen parámetros, pero es posible si ponemos la función con parámetros dentro de una arrow function o una función anónima.

```
$eventoMultiple.addEventListener('click', ()=> saludar('frank'));
$eventoMultiple.addEventListener('click', function(){
        saludar('frank');
});
```

Remover Eventos

Sólo se puede hacer en eventos múltiples y la función pasada tiene que haber sido expresada o declarada. No se puede hacer con arrow functions que hayan sido pasadas en el evento o funciones anónimas.

```
Const removerDobleClick = () => {
$eventoRemover.removeEventListener('dblclick', removerDobleClick);
}

$eventoRemover.addEventListener('dblclick', removeDobleClick);
```

****e.target*** → Hace referencia al elemento que origino el evento-mu

Flujo de eventos (Video 74)

Burbuja → Los eventos se propagan desde el elemento más interno al mas externo
Captura → Los eventos se propagan desde el elemento más externo al más interno

* Ver documentación del tercer parámetro de los eventos

Prevenir propagación y acciones por efecto de un elemento

Se pone dentro de la función pasada al evento, arrow function o función anónima los siguientes métodos de 'event'

```
$linkEventos.addEventListener('click', (event) =>{
        alert('');
        event.preventDefault();
        event.stopPropagation();
});
```

Delegación de eventos

- Es la forma más optima de trabajar con eventos.
- Se asigna el 'addEventListener' al 'document' preferiblemente de esta manera engloba a todos los elementos. Una vez hecho eso buscamos con 'e.target' el elemento que quiere desencadenar el evento y con una serie de condicionales y la funcion 'matches()' hacemos lo indicado según el selecto del elemento encontrado.
- De esta manera también se evita la propagación

1.
```
funciont flujoEventos(e){
console.log(`Hola te saluda ${this}, el click lo origino ${e.target.className}`)
}
```

2.
```
document.addEventListener('click', (e)=>{
        console.log('click en', e.target);

        if(e.target.matches('.eventos-flujo div')){
        flujoEventos(e)
        }

        if(e.target.matches('eventos-flujo a')){
        console.log('hola');
        }
});
```

Busca la clase 'eventos-flujo' y si un 'div' de ésta ha desencadenado el evento, cumple la condición

BOM : Propiedades y eventos (Video 77)

"resize"

"scroll"

"DOMContentLoad"
→ Carga más rápido que el 'load', no espera a que se cargue todo el HTML (Útil para peticiones asincronas)
Más eficiente para nuestras peticiones asincronas, etc.

"load"
→ Tarda mucho más en cargar, espera hasta que se cargue todos los scripts, hojas de estilo, que se parsee todo el HTML, etc.

Window → Buscar propiedades en MDN
```
        window.screenX
        window.screenY
        window.scrollX
        window.scrollY
```

```
window.addEventListener('scroll', (){
});
```

Métodos

```
$btnAbrir = document.getElementById(‘abrir-ventana’);
$btnCerrar = document.getElementById(‘cerrar-ventana’);
$btnInprimir = document.getElementById(‘imprimir-ventana’);

let ventana;

$btnAbrir.addEventListener(‘click’, (e)=>{
        ventana = window.open(‘http:·...’)
});

$btnCerrar.addEventListener(‘click’, (e)=> {
        ventana.close();
});

$btnImprimir.addEventListener(‘click’, (e)=>{
        window.print();
});
```

<u>Objetos Importantes</u>

Objeto URL (Location)
→
Location.origin
Location.protocol
Location.host
Location.hostname
Location.port
Location.href
Location.hash
Location.search
Location.pathname
Location.reload

Objeto Navegador (Navigator)

→
Navigator.connection
Navigator.geolocation
Navigator.mediaDevices
Navigator.mimeTypes
Navigator.onLine
Navigator.serviceWorker
Navigator.storage
Navigator.usb
Navigator.userAgent