**VAR LET Y CONST**

Una de las razones por las que se está dejando de usar var, es porque presenta vulnerabilidades en la seguridad. Por ejemplo, un usuario puede ingresar a la consola y modificar el valor, redeclarando una variable para inyectar código malicioso.  
Es importante recordar que let no permite usar antes de inicializar.

Por ejemplo, si se intenta imprimir una variable antes de inicializarla con var, da como resultado **undefined**. En cambio con let, da error.

Let, aunque soluciona algunos de los problemas de seguridad, sigue siendo vulnerable antes la redeclaracion por consola. La solución a esto es usar **const**

Las constantes, no pueden ser redeclaradas de la forma:

Const info= ‘asdas’;

Info = ‘123123’;

Pero si puede implementarse un

Const info=”” en otra parte del código, como por ejemplo en dos eventos click distintos, o en el mismo evento click, repetido varias veces.

**ARROW FUNCTIONS**

Lo mismo pasa en términos de seguridad con las funciones, corren el riesgo de ser redeclaradas desde la consola.  
Para solucionar esto, nuevamente se implementan las **const** ej:

const sumar = (a, b) => {

    return a + b;

}

En este caso la funcion sería segura.

**OPERADOR TERNARIO**

En cuestiones de seguridad, es mejor que escribir un if-else (de una sola línea cada una) ya que el valor puede almacenarse en una const. EJ:

const aprobacion = edad >= 18 ? 'apruebas' : 'no apruebas';

El if de una sola línea, o sea, comprobar solo por el valor de verdad TRUE de una condición puede escribirse como un operador:

const validado = true;

const validacion = validado && 'puedes pasar';

**DESTRUCTURING**

Para hacer el destructuring de un objeto, además de crear una variable por cada llave, puedo crear todas desde la declaración del objeto de la siguiente manera.

var { nombre, apellido, especie } = {

    nombre: 'anakin',

    apellido: 'skywalker',

    especie: 'humano',

    hijos: {

        hijo: 'Luke',

        hija: 'Leia'

    },

    pareja: 'Padme Amidala',

    madre: 'Shmi Skywalker',

    rango: 'Caballero Jedi',

    asosiacion: 'republica galáctica'

};

En este caso quedarían creadas las variables nombre, apellido y especie, aunque si se quisiese podría una para cada llave.

En cambio, si se llamase la variable, por ejemplo **anakin**, se debería realizar el destructuring de cada variable. Para recorrer efectivamente llaves y valores puedo utilizar:

for (const prop in anakin) {

    console.log(`${prop}: ${anakin[prop]}`);

}

Mientras que para llaves puedo usar un forEach a **Object.keys(anakin).forEach** y para valores **Object.values(anakin).forEach**

**ARRAY.PROTOTYPE.MAP()**

El método map() crea un nuevo array con los resultados de la llamada a la función indicada aplicados a cada uno de sus elementos

La principal ventaja de map por sobre otros métodos es que no modifica los valores del array original, sino que crea uno nuevo.

**DESTRUCTURING ARRAY**

El destructuring en array funciona igual que en objetos. Una forma rápida de hacerlo es la siguiente:

const [pj1, pj2, pj3, pj4, pj5, pj6, pj7] = ['anakin', 'obi-wan', 'leia', 'padme', 'palpatine', 'mace windu', 'yoda'];

en el cual guardaría dentro de cada variable, en orden, los valores del array.

EN caso de que solo se quieran obtener algunos, y ahorrar espacio en memoria puedo hacer lo siguiente.

**TEMPLATE STRING**

De los template string es importante recalcar que dentro de ${} no solo se pueden invocar variables, sino que lo que allí dentro ocurre es que se ejecuta código JS. Además, el final del template string `` es un string como tal, por tanto pueden aplicársele métodos como **toUpperCase()** o **.trim()** entre otros.

const [, pj2, , pj4, , pj6, ] = ['anakin', 'obi-wan', 'leia', 'padme', 'palpatine', 'mace windu', 'yoda'];

en cuyo caso solo guardaría las variables que necesito. Ocupo las comas para “marcar” el espacio, pero no creo las variables. Esto es debido a que el destructuring trabaja de manera posicional y no con una llave identificadora como un object.

**SPREAD OPERATOR**

El spread operator sirve para crear una copia de un elemento. Por ejemplo, para “unir” dos arrays en uno, su identificador es …

const sw = ['luke', 'vader', 'dooku', 'ahsoka'];

const mv = ['iron man', 'cap', 'thor'];

const crossover = [...sw, ...mv];

console.log(crossover);

**PROMESAS**

Una forma mejor de entender las promesas es la siguiente:

const sumar = (a, b) => {

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

        if (a < 0 || b < 0) {

            reject('No sumamos numeros menores a 0');

        } else {

            resolve(a + b);

        }

    });

}

Podemos entender una promesa como un bloque de código a ejecutar que recibe 2 parámetros, ambos funciones **resolve y reject** que son lo que se ejecuta en caso de fallar o no.  
en este caso sumar es una arrow function que recibe por parámetros a,b, por lo tanto podríamos invocarle con 2 parámetros números, pero lo único que recibiríamos sería una promesa, ya que así está determinado, por tanto para conseguir el resultado y operar aun faltan algunos pasos.

Esta nueva promesa, también es un arrow function, en la cual comenzarán a usarse los valores a y b pasados anteriormente. En este caso, si a o b son menores a 0, se ejecuta el **reject** y en caso negativo, se ejecuta el resolve.

Estas funciones son como returns, por tanto, el resultado de **reject** en este ejemplo es un string, y el de **resolve** es un int

Si quisiésemos operar con esta promesa deberíamos hacerlo de la siguiente manera:

const result =

    sumar(3, 5)

    .then((res) => {

        console.log(res);

    })

    .catch((e) => {

        console.log(e);

    })

Dentro de la const result, se almacena la invocación a la funcion suma con los parámetros 3 y 5, y en base a este resultado que se espera se especifican dos acciones.  
**.then() 🡪 resolve  
.catch()🡪reject**Es importante entender que los parámetros que “traen” tanto **then**  como **cactch** son los que se especificaron en la definición de la promesa. En este caso, 8 en el caso de then y 'No sumamos numeros menores a 0' en el caso de catch.

Cabe resaltar que no es fundamental que una **new** **Promise** sea retornada de una funcion, también puede declararse independientemente y luego llamarse por su nombre.  
const nombre = "agustin";

const verify = new Promise((resolve, reject) => {

    if (nombre == 'agustin') {

        resolve('hola, yo te conozco');

    } else {

        reject('no tengo idea de quien eres');

    }

})

verify.then((res) => {

        console.log(res);

    })

    .catch((err) => {

        console.log(err);

    })

**FETCH API CON PROMESAS**

A los url que se acceden para conseguir información, por ejemplo los que se insertan como param en **fetch()** se los conoce como **endpoints** , porque son el lugar al que se quiere llegar, no se conoce o se tiene en cuenta el resto de la infraestructura del sitio, si no el endpoint.  
El método **fetch()** al conseguir la info desde la url, retorna una promesa, por eso pueden anidarse .then() y .catch() a esta llamada.

fetch('https://jsonplaceholder.typicode.com/todos/1')

.then(response => response.json())

.then(json => console.log(json))

Una vez que se ejecuta la primera promesa, es necesario pasar los datos recibidos por el método **.json()** , ya que los primeros datos que se reciben son datos propios de la peticion.

Luego, este response.json() devuelve una promesa, por eso se puede aplicar otro .then(), y esa promesa tiene como parámetro la variable **json** que no es más que response.json().

Con os datos en **json** ya es posible operar.

**AXIOS EN FETCH**

Axios es una librería que ayuda a realizar peticiones fetch, omitiendo el paso de usar .json().

Para utilizar axios, puede imlementarse un cdn:

    <script src="https://unpkg.com/axios/dist/axios.min.js"></script>

    <script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>

Este es un ejemplo de utilizar axios:

axios.get("https://jsonplaceholder.typicode.com/users/3")

    .then(({ data }) => console.log(data))

El método .get() recibe la url y permite ejecutar las promesas “desde ahí”

Notese que se utilizó el destructuring en los parámetros que se reciben. Esto se realizo efectivamente ya que primero se imprimio **data**, se revisaron sus llaves, y luego se accedió a estas.

Cabe resaltar que el destructuring puede aplicarse también en las peticiones fetch sin axios, pero estas, al “hacer el paso” de .json(), suelen “dejarnos con los datos que necesitamos”.

**ASYNC Y AWAIT**

Async: La declaración de función async define una función asíncrona, la cual devuelve un objeto AsyncFunction.

Await: El operador await es usado para esperar a una Promise. Sólo puede ser usado dentro de una función async function.

Ejemplo simple:

function resolveAfter2Seconds() {

    return new Promise(resolve => {

        setTimeout(() => {

            resolve('resolved');

        }, 3000);

    });

}

async function asyncCall() {

    console.log('calling');

    const result = await resolveAfter2Seconds();

    console.log(result);

*// expected output: "resolved"*

}

asyncCall();

pueden entenderse estas 2 instrucciones como “complementarias”

En el ejemplo **await** espera por la promesa que retorna la funcion **resolverAfter2Seconds()**

Esta promesa ejecuta un timeout que devuelve su resolve (resultado correcto) luego de 3 segundos.

Una vez terminada esta, se prosigue con la ejecución de la funcion asíncrona llamada **asyncCall()**

El operador await, “espera” por la ejecución de la promesa y deja su resolve en la const result, por tanto cuando se imprima result, el resultado será “resolved”, como indica la promesa declarada en **resolveAfter2Seconds()**. En cambio si simplemente se guardase el llamado a la funcion en la const result, pero no se aplicaría el operador await, el resultado de imprimir console.log(result) sería una promesa “pending” ya que no se ha esperado a que termine de resolver (delay de 3000s en la promesa).  
await lleva implicito la espera y ejecucion de la promesa.

**ASYNC Y AWAIT EN FETCH**

Se puede simplificar una petición fetch si se implementan async y await en estas:

const peticion = async() => {

    const response = await fetch("https://jsonplaceholder.typicode.com/users/8"); *//fetch retorna promesa*

    const data = await response.json();

    return data;

};

En este ejemplo se crea la const petición como una funcion async.

Lo que se hace es crear una constante para cada “paso”, como si cada constante fuese un then cuando se usa fetch comúnmente.

En **response** invocamos la primera promesa, y con el operador await le decimos que no prosiga hasta que esta concluya

Con data, guardamos los datos de response “pasados” por .json()

Luego, es suficiente return la const data para que cada vez que se llame a **petición()** se obtengan los “datos deseados” de la petición fetch.

const data = peticion().then(console.log);

es importante saber que el data retonado por petición() al estar dentro de una funcion async, se convierte en una promesa, por tanto, para ver su contenido se debe recurrir a un .then()

**AXIOS CON ASYNC Y AWAIT**

Si se quisiese utilizar axios con async y await tambien se podría:

const peticion = async() => {

    const { data } = await axios.get("https://jsonplaceholder.typicode.com/users/8");

    return data;

}

Notese como, al axios saltarse el paso por .json(), puede usarse el destructuring y simplificar la petición a 2lineas de código.

Por ultimo, la impresión sería de esta forma:

const data = peticion().then((v) => console.log(v));

ya que el **data** retornado por petición() es una promesa