- Variables es declaren amb let i const, a més de var
- Les variables es podien redeclarar

Amb let i const, si tornam redefinir la variable ens donarà error.

- `Let` permet reassignar i `const` no.
- Quan poguem, farem servir ```const``.

No podem fer.

```
const b = [];
b = ["Hola"];
```

No podem reassignar, però en canvi, podem afegir elements:

```
b.push("Hola");
```

Amb `const`, el tipus no es pot modificar (no podem reassignar).

```
const d = "Hola";
d= "Ha"; //no ho podem fer
d += "Joan" // Tampoc
d.concat("Joan"); // això sí es pot
```

És a dir, podem operara a dins. Si volem fer reassignacions, farem servir `let`.

Problema scopes

Tipus:

- Global
- Local

Quan declaram al script principal, tenim accès a tot el codi.

```
var globalVar = 10;
const globalConst = 20;
```

Var té scope de funció, i let/const tenen scope de bloc.

```
var globalVar = 10;
const globalConst = 20;
```

```
function scope1(){
    for (i=0;i<10;i++>){
        var z1 = i;
        const z2 = i;
    }
}
```

Fora del `for`, les variables continuen existint i tenint valor, ja que tenen scope de funció. Ja fora de la funció, no.

Problema de hoisting (38)

Hoisting significa elevar la variable

```
var x=3;
y  =4;

console.log(x+y);
```

A la variable y, es fa una declaració `var y;` abans de la assignació.

Si això passa dins una funció

```
var x=3;
y  =4;

console.log(x+y);
function hoisting(){
    var notHoisted = 5;
    hoisted  = 6:
}
```

Aquesta variable hoisted, tot i que es declararà com a var, es declara com `var hoisted` al començament del codi JS i per tant té scope global.

Tenim una sentència que evita aquest tipus d'error, posant això al començament del codi:

```
"use strict";
```

Això ens permet que el codi sigui més segur. També evita poder posar un paràmetre dues vegades dins la declaració d'una funció.

Tampoc ens permet declarar cap paraula reservada, com ara:

```
var eval = 4;
var function = 5;
```

Destructuring

```
const persones = ["Joana","Toni","Rafel];
let a, b, c;
[a,b,c] = persones;
console.log(a,b,c);
```

Amb objectes:

```
const persona = {
    nom: 'Dani',
    cognom = 'Moreno'
}

const {nom,cognom} = persona;
```

Podem fer un alias:

```
const persona = {
    nom: 'Dani',
    cognom = 'Moreno'
}

const cotxe = {
    nom: 'Renault',
}

const {nom,cognom} = persona;
const {nom:nomCotxe} = persona; //alias
```

40. Mòduls, import i export

Si tenim més d'un script javascript i els posam amb script dins un fitxer html.

Les variables i funcions són intercanviables entre els scripts. És a dir, podem veure aquestes funcions i variables desde els altres scripts.

Podem aïllar els nostres scripts un de l'altre. Per això feim:

```
<script type="module" src="script1.js">
```

`import` i `export` només funcionen quan són mòduls (tenen `type="module"`). Com podem importar una variable dins un altre script:

```
export let x:number = 3;
```

Amb això, dins l'altre script podem fer un import:

```
import {x} from './script1.js';
```

42. Arrow functions

Així definiem les funcions abans.

```
const suma = function(a,b){
    return a+b;
}
```

Nova sintaxis per escriure funcions, amb una fletxa.

```
const suma2 = (a,b)=> a+b;
```

La fletxa indica un `return`.

Si no volem retornar res. Si posam la clau, el retorn s'ha de explicitar.

```
const suma3 = (a,b) => {
    return a+b;
}
```

43. Noves cometes

```
const persona = {
    nom: 'Dani',
    cognoms: 'Moreno
```

```
}

const mascota = 'Miky';

const complet = "En" + persona.nom + " té un ca que nom " + mascota;
```

Amb les noves cometes, és multilínea i permet cridar a variables sense tancar cometes. Va amb accent obert

```
const complet = `En ${persona.nom} té un ca anomenat ${mascota}`;
```

44. IIFE

```
function init(){

}

init();
```

Fent IIFE:

```
function init(global){

}

(function()){
    const global = 10;
    init(global);
})()
```

45. Classes

Cream una classe:

```
class Vehicle{
    nom;  //públic
    #tipus; //privat
}
```

Abans es posava _ per a privat, que volia dir que pel programador és privat, però realment no ho era. Amb el # és privat de veres.

Construïm l'objecte:

```
class Vehicle{
    nom;  //públic
    #tipus; //privat

constructor(nom, tipus){
        this.nom = nom;
        this.#tipus = tipus;
    }
}
```

```
const vehicle1 = new Vehicle("peugeot 206","cotxe")
console.log(vehicle1.nom);
//a tipus no podem cridar
```

Podem accedir a les variables mitjançant getters i setters:

```
get tipus(){
    return this.#tipus;
}
set tipus(valor){
    this.#tipus = valor;
}
```

Ara si podem cridar:

```
console.log(vehicle1.tipus);  //crida al getter
vehicle.tipus = "moto"; //crida al setter
```

Herència

```
    class Moto extends Vehicle{
        #rodes;
        #cilindrada;

        constructor(nom, tipus,rodes,cilindrada){
            super(nom, tipus);
            this.#rodes = rodes;
            this.#cilindrada = cilindrada;
        }
    }
```

Necessitariem fer getters i setters per les dues variables privades.

47. Falsy