JavaScript

JavaScript

JavaScript és un llenguatge de programació interpretat i orientat a objectes que s'usa principalment per a crear pàgines web interactives. És un component essencial de la tríada de tecnologies web, juntament amb **HTML** (Hypertext Markup Language) i **CSS** (Cascading Style Sheets). Amb **JavaScript**, podem afegir funcionalitats dinàmiques, validacions de formularis, animacions, i moltes altres característiques interactives a les teves pàgines web.

Llenguatges de la web:

- HTML per definir el contingut de pàgines de les webs
- CSS per especificar el format i distribució de les pàgines de web
- JavaScript Per programar el comportament de les pàgines de web

1.- Inicialització <script>

Un `script` en **JavaScript** i **HTML** és un fragment de codi **JavaScript** incrustat en un document **HTML**. Este `script` s'executa en el context del navegador web de l'usuari i pot interactuar amb l'estructura del document **HTML**, el contingut i fins i tot amb altres recursos web.

- Podem incorporar scripts **JavaScript** directament dins d'un document **HTML** utilitzant l'element <`script`>
- Este element podem situar-lo entre la capçalera <`head`> o al cos <`body`> del document, encara que es recomana situar-lo al final del <`body`>, per assegurar-nos que tots els recursos (html, css, ...) estan ja carregats.

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Script en JavaScript y HTML</title>

    <!-- Incorpora el script -->
    <script>
        // Codi JavaScript
    </script>

</head>
<body>
<!-- Contingut del cos de la pàgina -->
</body>
</html>
```

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Script en JavaScript y HTML</title>

</head>
<body>
<!-- Contingut del cos de la pàgina -->

<!-- Incorpora el script -->
<script>
    // Codi JavaScript aquí
</script>

</body>
</html>
```

El codi pot estar en un fitxer extern, referenciat en l'atribut `src`

```html
<script src="script.js"></script>
```

Dins l'etiqueta <**script**>, podem posar qualsevol codi **JavaScript** vàlid. Este codi s'executarà quan el navegador carregue o processe la pàgina web. Com per exemple usant esdeveniments com **onload**, **onclick**, etc.

```
<script>
    function saludar() {
        alert('Hola, aquest és un script en JavaScript i HTML!');
    }

    // Crida la funció al carregar la pàgina
    window.onload = saludar;
</script>
```

El codi **JavaScript** pot interactuar amb l'estructura del document **HTML** mitjançant el **DOM** (**Document Object Model**). Podem canviar continguts, afegir o eliminar elements, gestionar formularis i realitzar altres accions dinàmiques.

Exemple complet:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Script en JavaScript y HTML</title>
    <script>
        function saludar() {
            alert('Hola, aquest és un script en JavaScript i HTML!');
        }
    </script>
</head>

<body onload="saludar()">
<h1>Benvingut a la meva pàgina!</h1>
<p>Aquí tens un script en JavaScript.</p>

</body>
</html>
```

2.- Eixida de dades

- ### alert

Utilitzem la funció **`alert`** per **a mostrar un missatge en una finestra emergent al navegador**. Este tipus d'alerta és útil per a missatges simples o avisos:

```
// Finestra emergent amb un missatge
alert("Açò és un missatge d'alerta!");
```

Quan este codi s'executa, apareixerà una finestra emergent al navegador amb el missatge proporcionat.

- ### console.log()

Utilitzem la funció **`console.log()`** per escriure informació a la **consola del navegador** (F12). Este mètode és molt útil durant el desenvolupament per depurar el codi, ja que pot mostrar valors de variables, missatges d'estat i altres informacions a la consola.

```
// Mostra un missatge a la consola del navegador
console.log("Aquest missatge apareix a la consola.");
```

- ### document.write()

S'utilitza per escriure directament al document **HTML**. Tot i que pot ser útil en alguns casos, **s'ha de tindre precaució en utilitzar-la**, ja que pot substituir completament el contingut existent de la pàgina.

```
// Escriu al document HTML
document.write("Aquest missatge apareix al document HTML.");
```

- ### innerHTML

Es pot utilitzar per canviar el contingut d'un element **HTML** utilitzant **JavaScript**. Este mètode permet la manipulació dinàmica del contingut de la pàgina.

```
<!-- HTML -->
<div id="resultat"></div>

<!-- JavaScript -->
<script>
    // Canvia el contingut de l'element amb id "resultat"
    document.getElementById("resultat").innerHTML = "Nou contingut dinàmic.";
</script>
```

- **confirm()**

Esta funció mostra una finestra emergent amb un missatge i dos botons: "**Aceptar**" **i** "Cancelar". És freqüent utilitzar-la per obtenir una resposta afirmativa o negativa de l'usuari.

```
// Demana una confirmació a l'usuari
let confirmacio = confirm("Estàs segur que vols continuar?");
if (confirmacio) {
    console.log("L'usuari ha confirmat.");
} else {
    console.log("L'usuari ha cancel·lat.");
}
```

- **prompt()**

mostra una finestra emergent amb un missatge, un camp d'entrada de text i botóns "Aceptar" i "Cancelar". S'usa per obtenir dades d'entrada de l'usuari.

```
// Demana a l'usuari que introdueixi alguna cosa
let resposta = prompt("Com et dius?");
console.log("El nom introduït és: " + resposta);
```

3.- Variables

En JavaScript les variables les declarem amb les paraules clau **`var`**, **`let`** i **`const`**, es diferencien entre elles segons l'àmbit d'aplicació i la capacitat de reassignació.

- **`var`**: no té l'àmbit de bloc i es declara a nivell de funció. Això significa que, si es declara dins d'un bloc (com un bucle o una condició), encara serà accessible fora d'aquest bloc.
- **`let`**: és una millora de var, té àmbit de bloc, es a dir, la variable només és accessible dins del bloc en què es declara.
- **`const`**: s'utilitza per a declarar constants (no es pot reasignar)

En general:

- Utilitzem **`const`** per declarar variables que no canviaran de valor.
- Utilitzem **`let`** per variables que poden canviar de valor.
- **Evitem** **`var`** en favor de **`let`** i **`const`** per millorar la claredat i la seguretat del codi.

Exemples:

```
// Declaració de variables amb let

let edat = 30;
let nom = "Anna";

// Declaració de variable constant amb const
const ciutat = "Barcelona";

// Reassignació d'una variable let
edat = 35;

// Intent de reassignació d'una variable const (provocarà un error)
   ciutat = "Madrid"; // Error: Assignment to constant variable.

// Utilització de les variables en una operació
let missatge = "Hola, " + nom + "! Tens " + edat + " anys i ets de " + ciutat +
".";

// Impressió del resultat per consola
console.log(missatge); // Output: Hola, Anna! Tens 35 anys i ets de Barcelona.
```

4.- Tipus primitius

En **JavaScript**, hi ha set tipus de dades primitius, (basicament, esl mateixos que a JSON) que són tipus de dades simples que no són objectes i que representen valors individuals:

- **Number**: Utilitzat per a tots els tipus de nombres, en JavaScript tots el números són float, i es poden definir amb decimals o sense

```
let numero1 = 10;
let numero2 = 10.0;

console.log(numero1 == numero2); // true
```

- **String**: Utilitzat per a seqüències de caràcters, com textos. Es poden escriure amb “ ” o ‘ ’

```
let frase = "Hola, món!";
var nom = "Pepe!";
```

Els dos tipus de cometes es poden combinar:

```
var frase = "Demà és l'ultim dia";
let nom = 'Es diu “Pepe"'
```

- **Boolean**: Utilitzat per a valors booleans, que poden ser **true** o **false**.

```
let esCert = true;
```

- **Undefined**: Representa un valor que encara no ha estat definit.

```
let x; // x és undefined
```

- **Null**: Representa la intenció explícita de no assignar un valor.

```
let y = null;
```

- **BigInt**: Utilitzat per a representar enters més grans del que pot representar el tipus Number.

```
let numeroGran1 = 9007199254740991n;
const numeroGran1 = BigInt("9007199254740991")
```

- **Symbol**: Utilitzat per a crear identificadors únics, es a dir, es fa servir per crear identificadors únics que poden ser utilitzats com a claus d'objecte per a evitar col·lisions amb altres claus

```
let simbol = Symbol('Descripció del símbol');
```

5.- Estructures de Control

Les estructures de control les utilitzem per controlar el flux d'execució del programa. Això inclou condicions, bucles i altres construccions que permeten prendre decisions i repetir tasques de manera eficient.

Condicionals (if, else if, else):

Els condicionals són utilitzades per executar diferent codi basat en una condició específica.

```
let hora = 12;
if (hora < 12) {
    console.log("Bon dia!");
} else if (hora >= 12 && hora < 18) {
    console.log("Bona tarda!");
} else {
    console.log("Bona nit!");
}
```

Bucles (for, while, do-while):

Els bucles són utilitzats per repet

ir una seqüència d'instruccions fins que una condició específica es compleixi.

```
// Bucle for
for (let i = 0; i < 5; i++) {
    console.log(i);
}

// Bucle while
let j = 0;
while (j < 5) {
    console.log(j);
    j++;
}

// Bucle do-while
let k = 0;
do {
    console.log(k);
    k++;
} while (k < 5);
```

Switch:

La sentència switch és utilitzada per executar diferents accions basades en diferents condicions.

```
let dia = 3;
switch (dia) {
    case 1:
        console.log("Dilluns");
        break;
    case 2:
        console.log("Dimarts");
        break;
    case 3:
        console.log("Dimecres");
        break;
    default:
        console.log("Altres dies");
}
```

6.- Arrays

Els **arrays** en **JavaScript** són molt similars als **arrays** en **Java**, són estructures de dades que permeten emmagatzemar col·leccions ordenades d'elements. Es a dir, permeten guardar varies dades en una sola variable.

- **Declaració**

```
// Declaració d'un array buit
let buit = [];

// Declaració d'un array amb elements inicials
let colors = ["blau", "verd", "roig"];
```

- **Accés a Elements**

```
console.log(colors[0]); // Output: blau
console.log(colors[2]); // Output: roig
```

- **Modificació d'elements**

```
colors[1] = "groc";
console.log(colors); // Output: ["blau", "groc", "roig"]
```

- **Longitud de l'array (`length`)**

```
console.log(colors.length); // Output: 3
```

- **Afegir i Eliminar Elements**

```
// Afegir un element al final de l'array
colors.push("taronja");
console.log(colors); // Output: ["blau", "groc", "roig", "taronja"]

// Eliminar l'últim element de l'array
colors.pop();
console.log(colors); // Output: ["blau", "groc", " roig "]
```

7.- Iteració d'Elements d'arrays

- **Amb bucle for:**

```
let colors = ["blau", "groc", "roig"];

for (let i = 0; i < colors.length; i++) {
    console.log(colors[i]);
}
// blau
// groc
// roig
```

- **Amb bucle forEach:**

```
colors.forEach(function(color) {
    console.log(color);
});

// blau
// groc
// roig
```

Mètodes d'Arrays

- **indexOf():** Retorna l'índex de la primera aparició d'un element dins de l'array.

```
let colors = ["blau", "groc", "roig"];


console.log(colors.indexOf("groc")); // Output: 1
```

- **join():** Uneix tots els elements de l'array en una cadena.

```
console.log(colors.join(", ")); // Output: blau, groc, roig
```

- **slice():** Retorna una còpia superficial d'una porció de l'array.

```
let subarray = colors.slice(1, 2);
console.log(subarray); // Output: ["groc"]
```

- **splice():** Modifica el contingut d'un array eliminant elements existents i/o afegint nous elements.

La sintaxi general és:

```
array.splice(start, deleteCount, item1, item2, ...)
```

On:

- **start**: L'índex a partir del qual s'ha d'iniciar l'acció de tallar o eliminar elements de l'array. És obligatori.
- **deleteCount**: El nombre d'elements que s'eliminaran a partir de l'índex start. Si aquest valor **és 0, no s'eliminarà cap element**. Si no s'especifica, tots els elements des de start fins a la fi de l'array seran eliminats.
- **item1**, **item2**, ...: Els elements que s'han d'afegir a l'array a partir de start.

```
const array = ['a', 'b', 'c', 'd', 'e'];

// Eliminem 2 elements a partir de l'índex 1
array.splice(1, 2);
console.log(array); // Output: ['a', 'd', 'e']


const array = ['a', 'b', 'c', 'd', 'e'];

// Afegim 'x' i 'y' a l'índex 2
array.splice(2, 0, 'x', 'y');
console.log(array); // Output: ['a', 'b', 'x', 'y', 'c', 'd', 'e']
```

8.- Objectes

Els objectes són una de les característiques més importants de JavaScript i es fonamenten en la idea de parells **clau-valor**.

- Els objectes son variables que contenen variables, un sol objecte pot tindre múltiples valors.
- Els objectes estan formats per parells clau:valor
- Al contrari que en JSON no es necessari que les claus estiguen entre cometes (si es un identificador vàlid).

• Creació d'Objectes

En JavaScript, els objectes es poden crear de diverses maneres:

- **Utilitzant l'Object Literal:**

```
let persona = {
    nom: "Anna",
    edat: 30,
    ciutat: "Barcelona"
};
```

A l'exemple, creem un objecte anomenat **persona** amb tres parells **clau-valor**: **nom**, **edat** i **ciutat**.

- **Utilitzant la Funció Constructora:**

```
function Persona(nom, edat, ciutat) {
    this.nom = nom;
    this.edat = edat;
    this.ciutat = ciutat;
}

let persona = new Persona("Anna", 30, "Barcelona");
```

Definim una **funció** constructora **Persona** que permet crear nous objectes de tipus persona.

Amb `new Persona()`, es crea un nou objecte persona amb els valors proporcionats.

• Accés als Membres de l'Objecte:

- **Accés:**

```
console.log(persona.nom);    // Output: Anna
console.log(persona.edat);   // Output: 30
console.log(persona.ciutat); // Output: Barcelona
```

Els membres d'un objecte es poden accedir utilitzant la notació de **punt** (**objecte.membre**). Això permet obtenir el valor associat a una clau específica.

- **Modificació**

```
persona.edat = 35;
console.log(persona.edat); // Output: 35
```

Per modificar un valor en un objecte, simplement assignem un nou valor a la clau corresponent.

- **Afegir**

```
persona.professio = "Enginyer";
console.log(persona.professio); // Output: Enginyer
```

Podem afegir nous membres a un objecte assignant un valor a una nova clau que encara no existeix

- **Eliminar**

```
delete persona.ciutat;
console.log(persona.ciutat); // Output: undefined
```

Per eliminar un membre d'un objecte utilitzem l'operador **`delete`**. Això farà que la clau i el seu valor associat siguin eliminats de l'objecte.

Iteració a través dels Membres de l'Objecte

Utilitzant un Bucle **For-In**:

Sintaxi:

```
for (variable in objecte) {
    // bloc de codi a executar
}
```

On:

- **variable**: Especifica una variable que prendrà el valor de cada clau dels objectes mentre es recorre.
- **objecte**: L'objecte sobre el qual es vol iterar les seves propietats.

```
const cotxe = {
    marca: "Toyota",
    model: "Corolla",
    any: 2020
};

for (let propietat in cotxe) {
    console.log(propietat + ": " + cotxe[propietat]);
}
```

```
// marca: Toyota
// model: Corolla
// any: 2020
```

Amb un bucle **for-in**, podem iterar a través de tots els membres d'un objecte i accedir als seus valors utilitzant la clau associada.

• Altres operacions amb Objectes:

- **Comprovar si una Propietat Existeix a l'Objecte**

```
console.log("nom" in persona);    // Output: true
console.log("pes" in persona);    // Output: false
```

Per comprovar si una clau específica existeix en un objecte utilitzem l'operador **in**.

- **Objectes Anidats**

```
let cotxe = {
    marca: "Toyota",
    model: "Corolla",
    any: 2020,
    propietari: {
        nom: "Marc",
        edat: 40
    }
};
console.log(cotxe.propietari.nom); // Output: Marc
```

Els objectes poden contenir altres objectes com membres. Això permet una estructura de dades més complexa i jeràrquica.

- **Clonar un Objecte**

```
let cotxe2 = Object.assign({}, cotxe);
```

Amb el mètode **Object.assign()** podem clonar un objecte. Copiarà tots els membres de l'objecte original a un nou objecte buit.

- **Fusionar dos Objectes:**

```
let cotxe3 = {
    color: "blau"
};

let cotxeCombinat = Object.assign({}, cotxe, cotxe3);
```

`Object.assign()` també ens permet fusionar dos objectes, creant un nou objecte que conté tots els membres dels objectes originals.

- **Convertir Objecte a Array:**

```
let arrayPersones = Object.values(persona);
console.log(arrayPersones); // Output: ["Anna", 35, "Enginyer"]
```

Amb `Object.values();` podem convertir tots els valors d'un objecte en una matriu (**array**), que pot ser útil per a diverses operacions de manipulació de dades.

• Mètodes útils d'Objectes

Object.keys(obj): Retorna un array de les claus (proprietats) enumerables d'un objecte.

Object.values(obj): Retorna un array dels valors de les claus enumerables d'un objecte.

Object.entries(obj): Retorna un array de parells [**clau, valor]** per a cada parell **clau-valor** d'un objecte.

Object.assign(target, source1, source2, ...): Copia els valors de tots els objectes origen (source1, source2, ...) al objecte destí (target). **Retorna el propi objecte destí.**

Object.freeze(obj): Congela un objecte, evitant que les seves propietats es puguin modificar, afegir o eliminar. **Retorna l'objecte congelat.**

Object.seal(obj): Tanca un objecte, permetent modificar les seves propietats existents, però evitant l'afegiment o l'eliminació de noves propietats. Retorna l'objecte tancat.

Object.getOwnPropertyNames(obj): Retorna un array de totes les propietats enumerables i no enumerables d'un objecte.

Object.hasOwnProperty(prop): Retorna un booleà indicant si l'objecte té una propietat pròpia amb el nom especificat.

Object.create(proto[, propertiesObject]): Crea un nou objecte amb el prototip especificat i opcionalment amb les propietats especificades.

Object.getPrototypeOf(obj): Retorna el prototip de l'objecte especificat.

Object.setPrototypeOf(obj, proto): Estableix el prototip d'un objecte especificat a un altre objecte o a null.

Object.is(obj1, obj2): Comprova si dos valors són estrictament iguals. És similar a l'operador ===, però té en compte els casos especials NaN i -0.

Object.entries(obj): Retorna un array amb els parells [clau, valor] per a cada parell clau-valor d'un objecte.

Object.values(obj): Retorna un array amb els valors de les claus d'un objecte.

Nota:
Les propietats enumerables són visibles durant la iteració, mentre que les propietats no enumerables no ho són, però encara són accessibles i útils en altres aspectes de la manipulació d'objectes.

```
let obj = {};

Object.defineProperty(obj, 'a', {
    value: 1,
    enumerable: false // Definim la propietat 'a' com a no enumerable
});

// La propietat 'a' no serà visible durant la iteració
for (let prop in obj) {
    console.log(prop); // No imprimirà res, ja que no hi ha propietats
enumerables
}

console.log(obj.a); // Podem accedir a 'a' directament
```

8.- Funcions

Les funcions en JavaScript són blocs de codi reutilitzables que realitzen una tasca específica. Poden ser cridades en qualsevol moment durant l'execució del programa i poden acceptar paràmetres i retornar valors.

• Definició de Funcions

Les funcions es poden definir utilitzant la paraula clau **`function`**. Poden ser declarades de dues maneres:

- **Declaració de Funcions**: Es defineixen amb la paraula clau **`function`** seguida del nom de la funció i els paràmetres, si n'hi ha.

```
function saludar(nom) {
    console.log("Hola, " + nom + "!");
}
```

- **Expressió de Funcions**: Es defineixen mitjançant una **expressió**. En aquest cas, la funció pot assignar-se a una variable.

```
let saludar = function(nom) {
    console.log("Hola, " + nom + "!");
};
```

• Cridar una Funció, Paràmetres i Arguments

Per cridar una funció, simplement es fa servir el nom de la funció seguit dels paràmetres entre parèntesis, si n'hi ha.

```
saludar("Anna"); // Output: Hola, Anna!
```

Els paràmetres són variables declarades dins del parèntesis de la definició de la funció. Els arguments són els valors reals que es passen a la funció quan es crida.

Les funcions poden retornar valors utilitzant la paraula clau **`return`**. Això permet que la funció retorne un resultat a qui l'ha cridat.

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

let resultat = suma(5, 3);
console.log(resultat); // Output: 8
```

• Funcions Anònimes

Una funció anònima en JavaScript és una funció que **no té un nom específic** i que, en lloc de ser declarada amb un nom concret**, s'assigna a una variable o es passa com a argument** a una altra funció. Este tipus de funcions són molt flexibles i es poden utilitzar en diverses situacions.

Són molt útils quan es necessita una funció temporal o una funció que es podrà reutilitzar en un context concret, però sense la necessitat d'assignar-li un nom específic.

```
//Assignació a una Variable:

let saludar = function(nom) {
    console.log("Hola, " + nom + "!");
};

saludar("Anna"); // Output: Hola, Anna!
```

```
//Passades com a Arguments a Funcions:

function operacio(a, b, func) {
    return func(a, b);
}

let suma = function(x, y) {
    return x + y;
};

let resultat = operacio(5, 3, suma);
console.log(resultat); // Output: 8
```

- **Funcions de Flecha (Arrow Functions)**

Les funcions de flecha són una forma més concisa de definir funcions en JavaScript, utilitza la notació de fletxa `(=>)` per a definir funcions d'una manera més compacta i llegible.

La sintaxi bàsica és:

```
const nomFuncio = (parametres) => {
    // Codi de la funció
};
```

Per exemple:

```
let suma = (a, b) => a + b;

console.log(suma(5, 3)); // Output: 8
```

- **Exemple :**

```
// Exemple de funció de flecha per a calcular l'àrea d'un rectangle

// Sintaxi bàsica d'una funció de flecha amb dos paràmetres (amplada i alçada)

const areaRectangle = (amplada, alcada) => {
    return amplada * alcada; // Retorna el resultat de l'amplada multiplicada per
l'alçada
};

const area = areaRectangle(5, 3);
console.log('Àrea del rectangle:', area); // Output: Àrea del rectangle: 15


// Exemple de funció de flecha amb només un paràmetre i retorn implícit

const quadrat = num => num * num; // No cal utilitzar les claus si només hi ha una
expressió

console.log('Quadrat de 5:', quadrat(5)); // Output: Quadrat de 5: 25

// Exemple de funció de flecha sense paràmetres

const salutacio = () => {
    console.log('Hola, món!');
};

salutacio(); // Output: Hola, món!
```