Llenguatges de Marques i Sistemes de Gestió d'Informació

UD 5.1 Introducció a Javascript

IES Jaume II El Just
Tavernes de la Valldigna
Departament d'Informàtica
Curs 2024-25

Índex

1 JavaScript bàsic

1.1 Inclusió de JavaScript en HTML

Podem incloure el codi JavaScript dins del codi HTML amb l'etiqueta `<script>`:

```
<!DOCTYPE html>
<html>
    <head>
        <script>
            document.getElementById("missatge").innerHTML = "Hola món";
        </script>
    </head>
...
```

L'etiqueta `<script>` pot anar dins de `<head>` o dins de `<body>`, però és preferible posar-la al final de `<body>` per tal d'accelerar la càrrega de la pàgina:

```
...

<body>

...

    <script>
        document.write("Hola món");
    </script>

</body>
</html>
```

> És possible tindre vàries etiquetes `<script>` en un document HTML.

També podem incloure codi JavaScript directament en el codi HTML, com a valor d'un atribut. Generalment s'utilitza per a associar un esdeveniment, com el clic d'un botó, a una acció JavaScript:

```
<button type="button" onclick="document.write('Hola
↪  món');">Saludar</button>
```

Però la forma més usual d'incloure codi JavaScript és per mitjà d'arxius externs. Això té l'advantatge de que el mateix codi és accessible per vàries pàgines:

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

> Els scripts en arxius externs solen tindre l'extensió **.js** i no s'escriuen entre etiquetes `<script>`.

1.2 Eixida de dades

Hi ha vàries formes de mostrar dades amb Javascript.

1.2.1 Amb `document.write`

```
<!DOCTYPE html>
<html>
<body>

    <script>
        document.write("<h1>Hola món</h1>");
    </script>

</body>
</html>
```

> Aquest mètode escriu directament al document HTML. Si s'invoca una vegada ja s'ha carregat la pàgina, esborrarà tot el contingut HTML.
>
> Es recomana usar aquest mètode només per a testing.

1.2.2 Accedint als element pel seu ID

Consisteix en associar un id a l'element HTML en què volem mostrar la informació, accedir a ell amb el mètode **`document.getElementById`** i canviar el seu HTML intern (propietat ***innerHTML***):

```
<body>

    <p id="missatge"></p>

    <script>
        document.getElementById("missatge").innerHTML = "Hola món";
    </script>

</body>
</html>
```

1.2.3 Amb `window.alert`

El mètode `window.alert` mostra un diàleg d'alerta:

```
<button type="button" onclick="window.alert('Hola món')">Saludar</button>
```

Es pot omitir l'objecte `window`:

```
<button type="button" onclick="alert('Hola món')">Saludar</button>
```

Aquest mètode no s'utilitza molt en l'actualitat, ja que es considera que mostrar aquest tipus de diàlegs és intrusiu.

1.2.4 Eixida per consola

Amb `console.log()` podem escriure directament a la consola del navegador. Aquesta opció és útil per a depurar:

```
<script>
    console.log("Hola món");
</script>
```

1.3 Lectura de dades

La forma més normal de llegir dades de l'usuari amb JavaScript és per mitjà d'**inputs de formularis**. Generalment accedim a l'input pel seu **id** i llegim el valor introduït per l'usuari (**value**):

```
<form>
    <input type="text" name="nom" id="nom">
    <button type="button" onclick="alert('Hola ' +
    ↪  document.getElementById('nom').value);">Enviar</button>
</form>
```

També podem utilizar el mètode `window.prompt`:

```
<body>
    <p id="missatge"></p>
    <script>
        nom = window.prompt("Com et diuen?");
        document.getElementById('missatge').innerHTML = "Hola " + nom;
    </script>
</body>
```

1.4 Sintàxi bàsica

Cada sentència finalitza mb un punt i coma (**;**):

```
alert('Hola');
alert('Món');
```

A l'igual que en altres llenguatges, es recomana escriure una sentència per línia.

> El punt i coma no es obligatori escriure'l si escrivim una sentència per línia, però es recomana fer-ho.

1.4.1 Comentaris

JavaScript admet comentaris d'una línia i de vàries línies:

```
//Aquest és un comentari d'una línia
//Aquest és un altre comentari d'una línia

/* Aquest és
un comentari
de vàries
línies*/
```

1.5 Variables

Les variables es declaren amb la paraula reservada `let`:

```
let nom; //Declaració sense assignació
let edat = 15; //Declaració amb assignació
```

Una vegada declarada una variable, es pot accedir a ella i modificar el seu contingut:

```
let missatge;
missatge = "Hola";
console.log(missatge); //mostra 'Hola' per consola
```

Els noms de les variables només poden incloure lletres, dígits o els símbols $ i_. El primer caràcter no pot ser un dígit.

```
//Noms vàlids:
let nomAlumne;
let a123;
let _edat;
```

```
let $;
let _;

//Noms no vàlids:
let nom-alumne;
let 1abc;
```

1.5.1 Constants

Les constants són variables a les quals s'assigna un valor en el moment de la seua creació i aquest no pot ser modificat:

```
const GRAVETAT = 9.8;
```

1.6 Tipus de dades

JavaScript és un llenguatge **dinàmicament tipat**, el que significa que una variable pot tindre un valor d'un tipus i després tindre un valor d'un tipus diferent.

JavaScript té un nombre molt reduït de tipus de dades: numèriques, cadenes de caràcters i booleanes.

1.6.1 Number

Representa números enters i amb decimals.

```
let edat = 51;
let area = 4.56;
```

1.6.2 String

Representa cadenes de caràcters. Es pot escriure amb cometes simples, dobles o invertides.

```
let cad1 = "Hola";
let cad2 = 'Món';
let cad3 = `El valor de cad1 és: ${cad1}`;
```

Les cometes invertides permeten interpretar variables i operacions dins dels delimitadors `${ }`.

1.6.3 Boolean

Les variables booleanes poden tindre els valors true o false. Generalment guarden el resultat d'una comparació.

```
let casat = false;
let actiu = true;
```

1.6.4 Altres valors

Quan una variable ha sigut declarada però no se li ha assignat cap valor, té el tipus **`undefined`**:

```
let nom;
console.log(nom); //Mostra undefined
```

El valor **`null`** s'utilitza per a representar valors inexistents o desconeguts:

```
let alumne = null;
console.log(alumne); //Mostra null
```

El valor **`Infinity`** s'obté al dividir un número entre 0:

```
console.log(1 / 0); //Mostra Infinity
```

El valor **NaN** significa **Not a Number**. S'obté quan una operació dónaria un resultar matemàtic erroni:

```
console.log("Hola" / 2); //Mostra NaN
```

1.6.5 L'operador `typeof`

L'operador `typeof` retorna un string amb el tipus de la variable o del valor que escrivim a continuació:

```
let nom = "Pepa";
let edat = 15;
let actiu = true;
console.log(nom); // Mostra "string"
console.log(edat); // Mostra "number"
console.log(actiu); // Mostra "boolean"
console.log(nom / edat); // Mostra NaN
```

1.7 Operadors matemàtics

Els operadors matemàtics bàsics són la **suma**, **resta**, **multiplicació** i **divisió**. A ells podem afegir els operadors de **mòdul** (calcula el residu de la resta entera) i l'operador d'**exponenciació**. Els seus símbols són:

- Suma: **+**,
- Resta: **−**,
- Multiplicació: *****,
- Divisió: **/**,
- Mòdul: **%**,
- Exponenciació: ******

Exemples:

```
console.log(8 % 3) // 2
console.log(2 ** 3) // 8
```

També tenim altres operadors unaris com la negació (**−**).

```
let num = 5;
console.log(-num) // -5
```

1.7.1 Concatenació de strings

L'operador binari **+** usat amb cadenes, torna la concatenació de les 2 cadenes. Si un dels 2 operands és una cadena, conveteix l'altre operand en cadena i els concatena.

```
let cad1 = "Hola ";
let cad2 = "Món";
let num = 5;
let bool = true;

console.log(cad1 + cad2); // "Hola Món"
console.log(cad1 + num); // "Hola 5"
console.log(cad1 + bool); // "Hola true"
```

1.7.2 Assignació

L'operador assignació **=** significa que s'assigna a la variable de la part esquerra de l'operador, el valor resultat d'avaluar la part dreta:

```
let num = (3 + 4) * 2;
console.log(num); // 14
```

1.7.3 Increment, decrement i operadors abreviats

L'**increment ++** augmenta el valor d'una variable en 1. El **decrement --** disminueix el seu valor en 1:

```
let num = 1;
num++;
console.log(num); // 2
num--
console.log(num); // 1
```

Aquests 2 operadors es poden usar darrere de la variable (**post-increment** o **post-decrement**) o davant (**pre-increment** o **pre-decrement**). La diferència és que amb els operadors **post**, primer s'avalua el valor de la variable i després es modifica, mentre que amb els operadors **pre**, primer es modifica i després s'avalua:

```
let num = 1;
console.log(num++); // 1
let num2 = 1;
console.log(++num); // 2
```

També tenim els operadors abreviats com en altres llenguatges, que combinen una operació aritmètica amb una assignació:

```
let num = 1;
num += 5; // num = num + 5
console.log(num); // 6
```

Els operadors abreviats són: **+=**, **-=**, ***=**, **/=**, **%=**, ****=**.

1.7.4 Precedència d'operadors

Si una operació té més d'un operador, el seu ordre de prioritat es defneix generalment amb les mateixes regles que coneixem de matemàtiques.

En la següent taula es mostren els valors de precedència que té cada operador, a major valor major precedència:

Precedència	Nom	Signe
16	post-increment	`variable++`
16	post-decrement	`variable--`

Precedència	Nom	Signe
15	pre-increment	`++variable`
15	pre-decrement	`--variable`
15	negació unaria	-
14	exponenciació	**
13	multiplicació	*
13	divisió	/
13	mòdul	%
12	suma	+
12	resta	-
2	asignacions	=, +=, etc.

A l'igual que en matemàtiques, si 2 operadors tenen la mateixa precedència, s'avaluen d'esquerra a dreta i els **parèntesis** poden modificar la preferència agrupant operacions.

1.8 Operadors de comparació

Els operadors de comparació retornen un valor **booleà**. Són els següents:

Operador	Descripció
==	igual
===	igualtat estricta (mateix valor i mateix tipus)
!=	distint
!==	desigualtat estricta (distint valor i distint tipus)
>	major
<	menor
>=	major o gual
<=	menor o igual
?	operador ternari

En comparar valors de diferents tipus, JavaScript converteix els valors a nombres:

```
console.log( '1' > 0 ); // true, la cadena '1' es converteix en el número 1
console.log( '01' == 1 ); // true
```

Per a valors booleans, `true` es converteix en `1` i `false` en `0`.

```
console.log( true == 1 ); // true
console.log( false == 0 ); // true
```

L'operador d'igualtat estricta comprova els valors sense convertir els tipus:

```
console.log( 0 === false ); // false
console.log( '01' == 1 ); // false
```

1.9 Sentencies condicionals

Les **sentències condicionals** s'utilitzen quan volem trencar l'estructura seqüencial del codi. Ens permet fer salts a altres parts del codi depenent d'unes determinades condicions.

1.9.1 if - else

La sentència **`if (...)`** avalua l'expressió booleana entre els parèntesis i executa el bloc de codi a continuació si el resultat de l'expressió és `true`:

```
let edat = 18;

if (edat >= 18) {
    console.log("Eres major d'edat");
}
```

Podem afegir la sentència **`else`** per a que s'execute un bloc de codi alternatiu si l'avaluació de l'expressió entre parèntesis és `false`:

```
if (edat >= 18) {
    console.log("Eres major d'edat");
} else {
    console.log("Eres menor d'edat");
}
```

Si tenim més d'una condició, podem utilitzar **`else if`** per a avaluar-les totes:

```
if (edat < 0) {
    console.log("Encara no has nascut!");
```

```
} else if (edat < 18) {
    console.log("Eres menor d'edat");
} else {
    console.log("Eres major d'edat");
}
```

1.9.2 Operador ternari ?

L'**operador ternari** s'utilitza per a simplificar una expressió `if else`. És especialment útil en assignacions.

Per exemple, el següent codi:

```
let missatge = edat >= 18 ? "Eres major d'edat" : "Eres menor d'edat";
console.log(missatge);
```

Seria l'equivalent a:

```
let missatge;

if (edat >= 18) {
    missatge = "Eres major d'edat";
} else {
    missatge = "Eres menor d'edat";
}

console.log(missatge);
```

1.9.3 Operador Nullish Coalescing ??

L'operador **nullish coalescing** (??) es va afegir recentment a JavaScript per gestionar de manera més eficient els valors `null` i `undefined`. Aquest operador retorna el primer valor definit, és a dir, aquell que no siga `null` ni `undefined`.

- **Sintaxi**: `a ?? b`
  - Si a és **definit** (és a dir, no és `null` ni `undefined`), retorna a.
  - Si a és **null** o **undefined**, retorna b.

Exemple:

```
let user;
console.log(user ?? "Anonymous"); // "Anonymous", perquè user és undefined
```

```
let userName = "John";
console.log(userName ?? "Anonymous"); // "John", perquè userName no és null
↪  ni undefined
```

L'operador `||` (*or* lògic) també s'utilitza per proporcionar un valor per defecte, però té una diferència important: considera falsos tots els valors com `0`, `""` (cadena buida) o `false`, mentre que `??` només considera `null` i `undefined` com a valors "no definits".

Exemple:

```
let height = 0;
console.log(height || 100);  // 100, perquè 0 és considerat "fals"
console.log(height ?? 100);  // 0, perquè 0 no és null ni undefined
```

1.9.4 Operadors lògics

Els **operadors lògics** són:

Operador	Valor	Descripció	Exemple (`x = 5; y = 2;`)
`&&`	`and`	`true` si les 2 expressions són `true`	`(x < 6 && y > 1) // true`
`||`	`or`	`true` si almenys una de les 2 expressions és `true`	`(x == 4 || y == 1) // false`
`!`	`not`	Negació de l'expressió	`!(x == 5) // false`

Aquests operadors poden combinar-se en vàries expressions, tenint en compte que l'odre de precedència és:

`NOT > AND > OR`

Per exemple:

```
let x = 6, y = 2, z = 0;
if (x > 5 && (y == 2 || z != 0) && !(z < 1)) // true
```

1.9.5 switch - case

La sentència **switch - case** és similar a l'estructura `if - else if - else` i s'utilitza per a simplificar aquesta quan tenim moltes condicions:

```
switch(condicio) {
  case 'valor1':  // if (condicio === 'valor1')
    ...
    break;

  case 'valor2':  // if (condicio === 'valor2')
    ...
    break;

  default:
    ...
    break;
}
```

Els valors poden ser numèrics o cadenes.

La condició s'avalua sempre amb l'**operador d'igualtat estricta ===**.

La sentència **break** causa l'eixida de l'estructura `switch`. No és necessària, però en general s'utilitza ja que una vegada es compleix la condició, el `case` que coincideix amb aquesta actua com a punt d'entrada, executant-se les instruccions de tots els case a continuació d'aquest. Per exemple:

```
let x = 0;
switch (x) {
  case 0:
    num = "Zero";
  case 1:
    num = "U";
  case 2:
    num = "Dos";
    break;
  case 3:
    num = "Tres";
    break;
}

console.log(num) // "Dos"
```

En aquest codi si `x` val 0 o 1 el valor mostrat seria `"Dos"` ja que no tenim una sentència `break` per a eixir del `switch`.

Per últim, la sentència **`default`** s'utilitza per a executar codi quan la condició no coincideix amb cap `case`. Si `default` està al final del `switch`, no és necessari el `break`.

1.10 Bucles

Els bucles s'utilitzen per a repetir a mateixa acció un nombre determinat de voltes o mentre es complisca una condició.

1.10.1 Bucles amb while

La forma bàsica d'un bucle **`while`** és la següent:

```
while (condició) {
  // instruccions
}
```

És a dir, mentre es complisca la condició, s'entrarà al bucle i s'executaran les instruccions del seu cos.

També existeix una variant **`do .. while`** en la qual es comprova la condició al final de l'execució del cos del bucle:

```
do {
  // instruccions
} while (condició);
```

Exemples:

```
let i = 0;
while (i < 10) {
  console.log(i);
  i++;
}

let j = 0
do {
  console.log(j);
  j++;
} while (j < 10);
```

1.10.2 Bucles for

Els bucles **`for`** són especialment útils quan coneguem el nombre de repeticions. Tenen la forma:

```
for (inicialització; condició; increment/decrement) {
    // instruccions
}
```

- Inicialització: assignem un valor inicial a la variable comptador.
- Condició: la condició que s'ha de complir per a executar les instruccions del cos del bucle.
- Increment o decrement de la variable comptador:

Alguns exemples típics:

```
// Imprimir els números del 0 al 9
for (let i = 0; i < 10; i++) {
    console.log(i);
}

// Imprimir els números 10, 8, 6, 4, 2
for (let i = 10; i > 0; i -= 2) {
    console.log(i);
}
```

1.10.3 continue i break

Existeixen 2 instruccions que trenquen l'execució del bucle:

- **`continue`**: no executa la resta de la iteració actual i salta al començament de la següent iteració.
- **`break`**: ix del bucle sense executar la resta d'instruccions de la iteració actual ni la resta d'iteracions.

1.11 Funcions

Les **funcions** són fragments de codi que s'associen a un nom simbòlic i executen una sèrie d'instruccions. Serveixen per a repetir el mateix codi en diferents parts del programa sense necessitat de reescriure'l.

La forma general de declarar una funció és:

```
function nomFuncio (llista_paràmetres) {
    // intruccions
    return valor;
}
```

El **nom de la funció** pot contindre els mateixos caràcters que els noms de variables (lletres, dígits, _ i $).

La **llista de paràmetres** pot tindre 0 o més paràmetres. En cas de no tindre paràmetres els parèntesis s'han d'escriure igualment.

La instrucció **`return`** s'utilitza per a retornar un valor, generalment resultat d'operacions sobre els paràmetres d'entrada. La seua execució comporta l'eixida de la funció i el retorn de l'execució al programa principal. No és obligatòria la inclusió de la instrucció `return`. Si la funció arriba al final del seu codi sense trobar un `return`, finalitza la seua execució i retorna al programa principal.

La instrucció `return` també pot utilitzar-se per a eixir d'una funció sense tornar cap valor.

Les funcions es poden cridar des de qualsevol lloc del programa on siguen visibles, escrivint el seu nom i els possibles paràmetres requerits.

El valor de retorn d'una funció pot ser utilitzat en qualsevol expressió del programa, generalment és assignat a una variable.

Alguns exemples de funcions i del seu ús:

```
//Funció sense paràmetres ni return
function hola () {
    window.alert("Hola");
}

//Funció amb 1 paràmetre
function holaNom (nom) {
    window.alert("Hola " + nom);
}

//Funció amb 2 paràmetres i return
function suma (num1, num2) {
    let resultat = num1 + num2;
    return resultat;
}

//Formes de cridar a les funcions
hola();
hola("Joan");
let resultat = suma(4,6);
```

Una qüestió important és l'**àmbit de visibilitat** de les variables. Quan declarem una variable dins d'una funció (en l'exemple anterior `resultat` dins de la funció `suma`), aquesta és **local** i no és visible des de fora de la funció. Tanmateix, la variable `resultat` del cos principal és **global** i és visible des de dins de totes les funcions. En l'exemple anterior, al coincidir el nom de les variables `resultat`, el que tenim són 2 variables diferents, una global i l'altra local, que poden tindre valors diferents.

> Per a evitar problemes amb la visibilitat de les variables, es recomana declarar-les totes amb `let`. Si es declaren amb `var` o sense res, les variables locals poden ser visibles des de fora de les funcions i generar problemes al depurar.
>
> També és una bona pràctica reduir el nombre de variables globals.

Quan una funció té paràmetres per no li passem cap valor, el valor del paràmetre és `undefined`:

```
function f (a) {
  console.log(a);
}

f(); // undefined
```

Podem assignar **valors per defecte** als paràmetres, els quals s'assignaran quan la funció s'invoca sense passar-li cap valor al paràmetre en concret:

```
function f (a = "Hola") {
  console.log(a);
}

f(); // "Hola"
```

En JavaScript les funcions són valors i poden ser assignades a variables:

```
function f (a = "Hola") {
  console.log(a);
}

let copia = f;

copia(); // "Hola"
```

Aquesta característica permet passar funcions com a arguments d'altres funcions. Veurem exemples més endavant.

1.11.1 Funcions anònimes

Les funcions anònimes es declaren "sobre la marxa" i normalment s'assignen a una variable per usar o cridar posteriorment:

```
let addAnonymous = function(num1, num2) {
    return num1 + num2;
};
console.log(addAnonymous(3, 2));
```

1.11.2 Funcions fletxa (arrow functions)

Les funcions fletxa ofereixen una manera de definir funcions mitjançant una *expressió lambda* per especificar paràmetres entre parèntesis i el codi de funció entre claus, separats per una fletxa. La paraula clau `function` s'omet en definir-les.

La funció anterior, expressada com una funció fletxa, seria així:

```
let add = (num1, num2) => {
    return num1 + num2;
};
```

De manera similar a les funcions anònimes, el seu valor es pot assignar a una variable per al seu ús posterior o definir-se al moment en un fragment de codi específic.

De fet, el codi anterior es pot simplificar encara més: en els casos en què la funció simplement retorna un valor, es poden ometre les claus i la paraula clau `return`:

```
let add = (num1, num2) => num1 + num2;
```

A més, si la funció té un sol paràmetre, es poden ometre els parèntesis. Per exemple, la funció següent retorna el doble del número que rep com a paràmetre:

```
let double = num => 2 * num;
console.log(double(3)); // Displays 6
```

Com s'ha dit abans, les funcions fletxa, com les funcions anònimes, tenen l'avantatge de poder utilitzar-se directament on es necessiten. Per exemple, donada la següent llista de dades personals:

```
let data = [
    {name: "John", phone: "966343434", age: 40},
    {name: "Anne", phone: "9112565656", age: 55},
    {name: "Marcus", phone: "612998877", age: 13},
    {name: "Mary", phone: "611664366", age: 17}
];
```

Si volem filtrar persones majors d'edat, ho podem fer amb una **funció anònima** combinada amb la funció `filter`:

```
let adults = data.filter(function(person) {
    return person.age >= 18;
});
console.log(adults);
```

> La funció `filter` aplicada a un array (`b = a.filter(f)`), crea un array **b** amb els elements de l'array **a** que cumpleixen la condició definida en la funció **f**.

També podem utilitzar una **funció fletxa**:

```
let adults = data.filter(person => person.age >= 18);
console.log(adults);
```

Tingueu en compte que ací no assignem la funció a una variable per a un ús posterior; en canvi, la funció s'utilitza en el mateix punt on es defineix. El codi es fa més concís usant una funció fletxa.

La diferència entre les funcions fletxa i la notació tradicional o les funcions anònimes és que amb les funcions fletxa no podem accedir als elements `this` o als arguments, que estan disponibles en funcions anònimes o tradicionals. Per tant, si cal fer-ho, haurem d'optar per una funció normal o anònima en aquests casos.

1.12 Arrays

Els **arrays** són **col·leccions ordenades** de variables o objectes.

Es pot declarar un array de vàries maneres:

```
let a = new Array(); //Array buit
let b = []; //Array buit
let dies = ["Dilluns", "Dimarts", "Dimecres"]; // Array amb valors inicials
let notes = new Array("Suficient", "Bé", "Notable"); // Array amb valors
↪  inicials
```

En el cas de l'arrays dies, podem accedir als seus valor indicant el seu índex, tenint en compte que el primer valor té l'index 0:

```
console.log(dies[0]); // "Dilluns"
console.log(dies[1]); // "Dimarts"
console.log(dies[2]); // "Dimecres"
console.log(dies[3]); // undefined
```

Com es pot veure, si accedim a un índex sense cap valor obtenim `undefined`.

De la mateixa manera podem modificar un valor en concret:

```
dies[0] = "Diumenge";
console.log(dies[0]); // "Diumenge"
```

També podem afegir elements indicant l'index del nou element, inclús si deixem índexs buits sense assignar:

```
dies[3] = "Dijous";
dies[5] = "Dissabte";

console.log(dies[3]); // "Dijous"
console.log(dies[4]); // undefined
console.log(dies[5]); // "Dissabte"
```

Per a evitar tindre en compte quins índexs estan lliures a l'hora d'afegir elements, és millor utilitzar el mètode **`push()`**, el qual afegeix el nou element al final de l'array:

```
dies.push("Divendres");
```

La propietat **`length`** retorna el nombre total d'elements de l'array:

```
let dies = ["Dilluns", "Dimarts", "Dimecres"];
console.log(dies.length); // 3
```

El mètode **`toString()`** retorna un string amb tots els valors de l'array separats per comes:

```
console.log(dies.toString());
```

Per últim, recordar que els arrays són objectes:

```
console.log(typeof dies); // object
```

1.12.1 Iterar un array

La forma més usual de recórrer arrays és utilitzant un bucle **`for`** utilitzant els índexs:

```
for (let i = 0; i < dies.length; i++) {
    console.log(dies[i]);
}
```

Existeix una forma de bucle, **`for .. of`**, per a recórrer arrays:

```
for (let dia of dies) {
    console.log(dia);
}
```

El mètode **`array.foreach(func)`** també es pot emprar per a executar una determinada funció per a cada valor de l'array:

```
dies.forEach(myFunction);

// Imprimeix tots els valors en majúscules
function myFunction(value) {
```

```
    value = value.toUpperCase();
    console.log(value);
}
```

El mateix exemple es pot fer amb una funció anònima:

```
dies.forEach(function (value) {
    value = value.toUpperCase();
    console.log(value);
});
```

I amb una funció fletxa:

```
dies.forEach(value => {
    console.log(value.toUpperCase());
});
```

1.13 Objectes

El **objectes** són variables que poden tindre altres variables, anomenades **propietats**, i funcions, anomenades **mètodes**.

Les **propietats** poden ser de diferents tipus de dades, inclús poden ser altres objectes o arrays de variables o d'objectes.

Per a crear un objecte en JavaScript, al contrari que en altres llenguatges, no és necessari definir primer la classe a partir de la qual s'ha d'instanciar. Es poden declarar directament:

```
let alumne = {nom: "Pep", edat: 15};
```

Les propietats són parelles de **clau** (nom) i **valor** ("Pep") separades per 2 punts (**:**).

És freqüent escriure cada propietat en una línia nova per a més claredat:

```
let alumne = {
    nom: "Pep",
    cognoms: "Peris Llopis",
    edat: 15,
  };
```

Compte amb la última coma després de l'edat. Aquesta sintaxi és correcta i facilita l'adició i supressió de noves propietats en el codi.

També podem crear objectes buits:

```
let alumne = {}; // Sintaxi d'objecte literal
let professor = new Object(); // Sintaxi de construtructor d'objectes
```

Podem accedir a les propietats amb la notació de punt:

```
console.log(alumne.nom); // "Pep"
```

O amb la notació de claudàtors, indicar el nom de la clau entre cometes:

```
console.log(alumne["edat"]); // 15
```

Canviar els valor d'un objecte es fa de manera similar:

```
alumne.nom = "Maria";
alumne["edat"] = 18;
```

Una vegada creat un objecte, li podem afegir propietats dinàmicament:

```
let alumne = {nom: "Pep", edat: 15};
alumne.poblacio = "València";
```

I també les podem eliminar dinàmicament:

```
delete alumne.cognoms;
console.log(alumne.cognoms); // undefined
```

Si accedim a una propietat d'un objecte inexistent, ens tornarà `undefined`. Podem comprovar si una propietat existeix amb la instrucció **`in`**:

```
if ("cognoms" in alumne) {
    console.log(alumne.cognoms);
}
```

1.13.1 Mètodes

Els **mètodes** es poden definir de la mateixa forma que les *funcions anònimes*: el nom de la funció és la **clau** i la definició és el **valor**:

```
let alumne = {
    nom: "Pep",
    cognoms: "Peris Llopis",
    edat: 15,
    nomComplet: function() {
        return this.nom + " " + this.cognoms;
      },
    sumarEdat: function(anys) {
        this.edat += anys;
      },
    };
```

```
console.log(alumne.nomComplet()); // "Pep Peris Llopis"

alumne.sumarEdat(2);
console.log(alumne.edat); // 17
```

Com es pot veure, els mètodes s'utilitzen amb la notació de punt. També es poden utilitzar la notació de claudàtors:

```
alumne["sumarEdat"](2);
```

La paraula reservada **`this`**, quan es troba dins d'un mètode, fa referència al propi objecte. En l'exemple anterior `this.nom` té el valor "*Pep*".

Al igual que amb les propietats, els mètodes es poden afegir a un objecte una vegada que aquest s'ha creat:

```
alumne.nouMetode = function() {
    ...
}
```

1.13.2 Iterant objectes

Podem iterar per les propietats d'un objecte usant bucles **`for ... in`**:

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

En aquest exemple la variable `key` recorre totes les claus de l'objecte `alumne`, i amb això dins del bucle podem accedir als valors amb `alumne[key]`.

El problema del codi anterior és que imprimeix també les funcions. Podem visualitzar només les propietats usant la instrucció `typeof`:

```
for (let key in alumne) {
    if (typeof alumne[key] != 'function') {
        console.log(key + ": " + alumne[key]);
    }
}
```

1.14 Recursos

- Tutorial de Javascript.info

- Tutorial W3Schools
- Manual MDN (Mozilla)
- Especificació ECMAScript