Estructures definides pel programador

Àlex Salinas

Índex

Introducció

El codi Javascript no es pot executar de manera independent: sempre s'ha d'executar utilitzant un navegador web. Aquest codi està associat a una pàgina web. Aquesta pàgina estarà allotjada en un servidor d'aplicacions, i quan un navegador la demani s'enviarà junt amb el codi Javascript.

Quan aquesta pàgina arriba al navegador web es comença a interpretar. Els navegadors no interpreten la pàgina després de rebre-la, sinó que la van interpretant i, per tant, executant el codi Javascript a mesura que la van rebent. Existeixen maneres per evitar executar el codi Javascript abans de rebre tota la pàgina, perquè normalment es imprescindible tindre-la sencera.

En un principi, l'utilització de Javascript es va limitar en la creació d'efectes dinàmics dintre de la pàgina com, per exemple, canviar una imatge per un altre quan el ratolí es posava damunt o bé ressaltar algun paràgraf. Però poc a poc, es va anant ampliant el seu ús i amb la aparició de biblioteques com JQuery o AngularJS ha esdevingut un llenguatge imprescindible per desenvolupar aplicacions web modernes. Amb aquestes eines, Javascript permet crear efectes dinàmics impressionants que milloren l'experiència que rep l'usuari al utilitzar l'aplicació.

En el primer apartat de la unitat s'explicaran els conceptes relacionats amb les funcions. Les funcions són fragments de codi que resolen una tasca clarament definida. El seu resultat és un valor que pot ser utilitzat per altres funcions o pel programa principal. Tanmateix, a JavaScript veurem que no és sempre així: les funcions tenen un paper més important.

En aquest apartat aprendreu per què les funcions són la clau per entendre els secrets del llenguatge JavaScript. Les funcions són objectes igual que qualsevol altre tipus de dades. Poden ser declarades, poden ser referenciades com si fossin variables i fins i tot es poden passar com a paràmetres d'altres funcions.

Aquesta peculiaritat fa de JavaScript un llenguatge diferent. Si intenteu programar com si programéssiu en un altre llenguatge, sobretot com si programéssiu en Java, aquest no es comportarà com penseu.

S'explicaran les funcions més importants, que hem d'aprendre per donar els primer passos amb aquest llenguatge. Aquestes funcions tenen a veure amb la manipulació de cadenes de text o de nombres. També s'explicaran funcions especials de la biblioteca JavaScript que permeten programar l'execució d'altres funcions.

En definitiva, començarem explorant la potència del llenguatge JavaScript per tractar amb els tipus de dades més comuns. A continuació, començarem a explorar el costat més diferent del llenguatge. Començarem a explicar les funcions que un

desenvolupador pot crear. Per exemple, un programador JavaScript pot decidir crear una funció i no donar-li un nom o bé que s'autoexecuti una vegada s'ha creat. Són característiques que donen una potència inesperada a aquest llenguatge de programació del costat client.

En el segon apartat de la unitat s'explicaran els *arrays*. Aquest tipus de dades és molt útil i molt pràctic a l'hora de crear programes. Es mereixen una especial atenció, i s'explicaran les funcionalitat que ens dóna la biblioteca JavaScript per tractar aquest tipus de dades en forma d'exemple. Es realitzarà la creació d'un programa per gestionar una botiga i, d'aquesta manera, explicar les funcionalitats dels *arrays* des d'una vessant merament pràctica. Es realitzaran uns exemples molt senzills i, a partir d'aquests, es proposaran diferents mètodes per resoldre els problemes plantejats.

S'explicaran les funcions més importants per tractar amb els *arrays* d'una manera pràctica i útil que us ajudarà a entendre molt millor les diferents opcions que ens proposa JavaScript per manipular-los.

Javascript està present en pràcticament qualsevol àmbit: bases de dades, desenvolupament mòbil, servidors d'Internet, sistemes operatius, plataformes de jocs, administració de sistemes, etc. La seva influencia és increïble. Javascript quasi bé ha complert la promesa que feia Java fa més de 20 anys de "*Write once, run everywhere*".

Resumint, en aquesta unitat s'explicarà:

- Les funcions i la seva importància
- Les funcions més importants del llenguatge JavaScript
- Com crear funcions
- Com assignar paràmetres
- Com el navegador invoca les funcions
- Reconèixer les funcions com objectes de primera classe
- Els tipus de dades
- Els *arrays*: característiques i funcionalitats

Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

1. Programa codi per a clients web analitzant i utilitzant estructures definides per l'usuari.

- Classifica i utilitza les funcions predefinides del llenguatge.
- Crea i utilitza funcions definides per l'usuari.
- Reconeix les característiques del llenguatge relatives a la creació i ús d'arrays.
- Crea i utilitza arrays.
- Depura i documenta el codi.

1. Programació amb funcions

La principal diferència entre escriure bon codi JavaScript i un codi mediocre és concebre-ho com un llenguatge funcional. Les funcions són la base d'aquest llenguatge i saber utilitzar-les marca la diferència.

El més important és que les funcions són objectes de primera classe, és a dir, coexisteixen amb qualsevol altre objecte i les podem tractar com un d'ells. Igual que els altres tipus de JavaScript (Integer, Boolean...) les funcions les podem crear com a literals i fins i tot les podem passar com a paràmetres d'altres funcions. És convenient començar l'estudi de les funcions veient les ja definides al llenguatge.

1.1 Funcions predefinides pel llenguatge

És important conèixer les diferents funcions que ens proporciona el llenguatge JavaScript per tractar cadenes de text, nombres, conversions de tipus i altres funcions interessants.

1.1.1 Funcions predefinides de cadenes de text

Abans de començar a parlar de les funcions predefinides de les cadenes de text cal destacar la propietat **length** que retorna la mida de la cadena de text, ja que aquesta és una propietat que s'utilitza força sovint. Veieu un exemple:

```
1 let frase = "Estudiant a l'IOC s'aprèn molt.";
2 document.write(frase.length);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/rNeNyGe?editors=0010.

A continuació podeu veure algunes de les funcions predefinides de les cadenes de text més utilitzades.

concat()

Per concatenar o unir dues paraules o frases emmagatzemades en variables diferents es pot utilitzar el símbol **+** o bé la funció **concat**. S'ha de tenir en compte que a l'hora d'utilitzar-les s'uniran les dues paraules sense cap separació, per tant, si es volen separar les paraules a la cadena resultant s'ha d'afegir el símbol corresponent.

```
let frase1 = "Estudiant a l'IOC";
let frase2 = "s'aprèn molt.";
//sense separació
let resultat = frase1 + frase2;
document.write(resultat + "<br>");

//amb separació
let resultat = frase1 + " " + frase2;
document.write(resultat + "<br>");

//sense separació
let resultat = frase1.concat(frase2);
document.write(resultat + "<br>");

//amb separació
let resultat = frase1.concat(" " +frase2);
document.write(resultat);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/LYNYWQz?editors=0010.

Fixeu-vos què passa si es concatena una variable de tipus String amb una variable de tipus numèric.

```
let frase = "L'IOC és un ";
let nombre = 10;
document.write(frase + nombre);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/zYqYZWo?editors=0010.

Al concatenar un String amb una variable numèrica, aquesta es converteix automàticament a String. La biblioteca JavaScript permet fer el canvi automàticament sense que nosaltres haguem de fer cap tipus de conversió.

> Una alternativa a la concatenació de cadenes de text i variables és la utilització de plantilles de literals, per exemple: `document.write(`L'IOC és un ${nombre}`)`.

toUpperCase() i toLowerCase()

La funció **toUpperCase()** converteix la cadena sencera a majúscules. I la funció **toLowerCase()** realitza l'operació inversa a `toUpperCase`, és a dir, canvia els caràcters en majúscula per minúscula.

```
let frase = "Estudiant a l'IOC s'aprèn molt.";
document.write(frase.toUpperCase() + "<br>");
document.write(frase.toLowerCase());
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/JjXjWmr?editors=0010.

charAt()

La funció **charAt(posició)** retorna el caràcter que es troba a la posició indicada pel paràmetre.

```
var frase = "Estudiant a l'IOC s'aprèn molt.";
document.write(frase.charAt(0));
document.write(frase.charAt(1));
document.write(frase.charAt(2));
document.write(frase.charAt(3));
document.write(frase.charAt(4));
document.write(frase.charAt(5));
document.write(frase.charAt(6));
document.write(frase.charAt(7));
document.write(frase.charAt(8));
document.write(frase.charAt(9));
document.write(frase.charAt(18));
document.write(frase.charAt(19));
document.write(frase.charAt(20));
document.write(frase.charAt(21));
document.write(frase.charAt(22));
document.write(frase.charAt(23));
document.write(frase.charAt(24));
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/xxVxqQL?editors=0010.

El resultat de l'exemple és *Estudiant s'aprèn*. Aquesta frase és una porció de la frase original. Existeixen altres maneres de poder obtenir porcions de frases. Per exemple amb la funció `substring`.

substring

La funció **substring(inici, final)** retorna una porció d'un text des de la posició `inici` fins a la posició `final`. Si només s'indica el paràmetre `inici`, la funció retorna la part de la cadena original corresponent des d'aquesta posició fins al final.

```
let frase = "Estudiant a l'IOC s'aprèn molt.";
document.write( frase.substring(0,10) + "<br>");
document.write(frase.substring(18,25) + "<br>");
```

Si volem obtenir la mateixa cadena que a l'exemple `charAt` llavors s'ha de concatenar les dues porcions de la cadena original.

```
document.write(frase.substring(0,10) + frase.substring(18,25) + "<br>");
```

Podeu veure aquest exemple en l'enllaç següent:https://codepen.io/ioc-daw-m06/pen/ExKxWGX?editors=0010.

Fixeu-vos el comportament de la funció quan se li indica un valor negatiu com a posició inicial o final.

```
let frase = "Estudiant a l'IOC s'aprèn molt.";
document.write("Primer exemple: " + frase.substring(0,-12) + "<br>");
document.write("Segon exemple: " + frase.substring(-1,2)+ "<br>");
```

```
4 document.write("Tercer exemple:" + frase.substring(-1,-12)+ "<br>");
5 document.write("Quart exemple:" + frase.substring(-22));
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/RwawpEe?editors=0010.

Com heu vist, el primer exemple no retorna cap porció de la cadena original ja que la posició final negativa no la pot traduir a una posició final coherent.

En canvi, en el segon exemple al utilitzar una posició positiva, en aquest cas com a final de subcadena, llavors ha retornat des de l'inici (posició inicial igual a 0) fins a la posició final indicada (en aquest cas posició final igual a 2). El resultat és, doncs, la subcadena *Es*.

En el tercer exemple no ha retornat cap subcadena ja que, tal i com passava en l'exemple primer, no pot traduir les posicions negatives i per tant, no pot retornar cap subcadena.

En canvi, a l'últim exemple només se li indica d'una posició negativa. En aquest cas, el comportament de la funció `substring` és retornar la frase original sencera.

I si en comptes d'utilitzar nombres negatius utilitzem una posició final més petita que una posició inicial?

```
1 let frase = "Estudiant a l'IOC s'aprèn molt.";
2 document.write(frase.substring(9, 0));
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/ExKxWrX?editors=0010.

Bé, en aquest cas, la funció `substring` canvia l'ordre de les posicions. El resultat del codi anterior és *Estudiant*.

indexOf()

La funció **indexOf(caracter)** retorna la posició de la primera ocurrència d'un caràcter en una cadena. Si la cadena no conté el caràcter, la funció retorna el valor -1. En canvi, la funció **lastIndexOf(caracter)** retorna l'última ocurrència del caràcter cercat.

```
1 let frase = "Estudiant a l'IOC s'aprèn molt.";
2 let posicio = frase.indexOf("IOC");
3 let posicio2 = frase.indexOf("ñ");
4 document.write('Posició de la paraula IOC: ${posicio}<br>');
5 document.write('Posició de la lletra ñ: ${posicio2}<br>');
6 posicio = frase.lastIndexOf("t");
7 posicio2 = frase.lastIndexOf("ñ");
8 document.write('Posició de la lletra t: ${posicio}<br>');
9 document.write('Posició de la lletra ñ: ${posicio2}');
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/RwawpOq?editors=0010.

split()

La funció **split(separador)** separa una cadena de text en trossos. Es crea un pedaç de la cadena cada vegada que es troba el caràcter separador. Tots els pedaços de la cadena es retornen en un *array*.

```
let frase = "Estudiant a l'IOC s'aprèn molt.";
let trossos = frase.split(" ");
document.write(`Array: ${trossos}<br>`);
document.write(trossos[0] + "<br>");
document.write(trossos[1] + "<br>");
document.write(trossos[2] + "<br>");
document.write(trossos[3] + "<br>");
document.write(trossos[4] + "<br>");
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/KKzKWOV?editors=0010.

A cadascuna de les posicions de l'*array* `trossos` es troba un pedaç de la frase original. Els pedaços s'han creat cada vegada que s'ha trobat el caràcter *espai* (" "). La funció recorre caràcter a caràcter la frase original i cada vegada que troba el caràcter separador talla la cadena i guarda el pedaç tallat en una posició de l'*array*. Continua recorrent la cadena fins que s'acaba.

I si no li indiquem cap caràcter de separació? Si no li indiqueu cap caràcter separador, llavors separa caràcter a caràcter.

```
let frase = "Estudiant a l'IOC s'aprèn molt.";
let trossos = frase.split("");
document.write(trossos[0] + "<br>");
document.write(trossos[1] + "<br>");
document.write(trossos[2] + "<br>");
document.write(trossos[3] + "<br>");
document.write(trossos[4] + "<br>");
document.write(trossos[5] + "<br>");
document.write(trossos[6] + "<br>");
document.write(trossos[7] + "<br>");
document.write(trossos[8] + "<br>");
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/poyoPzb?editors=0010.

startsWith(), endsWith() i includes()

Aquestes tres funcions són molt útils a l'hora de comprovar cadenes de text, ja que ens permeten esbrinar si una cadena comença, acaba o inclou un fragment de text concret respectivament. Només cal passar com argument el fragment de text a cercar:

```
let cadena = "Estudiant a l'IOC s'aprèn molt";
console.log(`La cadena comença per "Estu"? ${cadena.startsWith("Estu")}`);
console.log(`La cadena acaba amb "molt"? ${cadena.endsWith("molt")}`);
console.log(`La cadena inclou "IOC"? ${cadena.includes("IOC")}`);

console.log(`La cadena comença per "1234"? ${cadena.startsWith("1234")}`);
console.log(`La cadena acaba amb "1234"? ${cadena.endsWith("1234")}`);
console.log(`La cadena inclou "1234"? ${cadena.includes("1234")}`);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/WNoNZXp?editors=0011.

repeat()

La funció **repeat(repeticions)** retorna la mateixa cadena repetida el nombre de repeticions indicada:

```
let cadena = "*";
for (let i=1; i<5; i++) {
  console.log(cadena.repeat(i));
}
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/qBZBmmM?editors=0012.

padStart() i padEnd()

Aquestes funcions permeten omplir una cadena per la dreta o per l'esquerra per assegurar-nos que la longitud d'una cadena es fixa. Això és interessant quan es treballa amb estils o dispositius amb fonts monoespaiades (per exemple la consola del navegador), és a dir, que totes les lletres són de la mateixa amplada.

padStart(mida, cadenaPerAfegir) omple la cadena fins a arribar a la mida que s'especifica i amb la cadena indicada començant per l'esquerra, mentre que **padEnd(mida, cadenaPerAfegir)** fa el mateix però començant per la dreta. Podeu veure les diferències en el següent exemple:

```
let cadena = "Joan";
console.log(cadena);
console.log(cadena.padStart(2));
console.log(cadena.padStart(15));
console.log(cadena.padStart(15, "#"));
console.log(cadena.padStart(15, "#*"));
console.log(cadena.padEnd(2));
console.log(cadena.padEnd(15));
console.log(cadena.padEnd(15, "#"));
console.log(cadena.padEnd(15, "#*"));
let midaCadena = cadena.length;
let midaAmbPadStart = cadena.padStart(15).length;
let midaAmbPadEnd = cadena.padEnd(15).length;
console.log(`Mida original: ${midaCadena}`);
console.log(`Mida amb padStart: ${midaAmbPadStart}`);
console.log(`Mida amb padEnd: ${midaAmbPadEnd}`);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/bGpGWrE?editors=0012.

Com es pot apreciar, si no s'indica un segon argument, la cadena s'omple amb espais fins a arribar a la mida indicada. En el cas que la cadena passada com argument contingui més d'un caràcter es descarten els sobrants, de manera que la mida sempre es respecta. Per altra banda, si s'indica una mida inferior a la de la cadena original, es retorna la mateixa cadena i no s'ajusta la mida.

1.1.2 Funcions predefinides de nombres

La biblioteca JavaScript permet totes les operacions de nombres que es poden realitzar en qualsevol altre llenguatge. Per sumar dos nombres utilitzarem l'operador suma (+), per multiplicar utilitzarem l'operador *asterisc* (*), per restar utilitzarem l'operador guió (-) i per dividir utilitzarem l'operador barra (/). Aquestes són les operacions bàsiques que es poden realitzar amb dos nombres. Però, una operació pot donar valors inesperats si no es comproven els dos operands. Per exemple, què donaria una divisió on els dos valors són zero? O què passaria si es divideix un nombre qualsevol per zero? Bé, la biblioteca JavaScript ens dona alguns valors i funcions per protegir el nostre programa davant d'aquestes situacions.

Una de les funcions que ens ajudarà a evitar aquestes situacions és la funció **isNaN()**. Aquesta funció comprova si el valor retornat per una operació amb nombres és un nombre o bé no ho és. De fet **NaN** son les inicials, en anglès, de **Not A Number** (No és un nombre).

```
let nombre = 0;

if(isNaN(nombre/nombre)) {
   document.write("La divisió no és correcta.");
}
else {
   document.write(‘La divisió és: ${nombre/nombre}‘);
}
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/bGpGWmB?editors=0010.

Fixeu-vos en l'exemple anterior. Abans d'utilitzar el resultat de la divisió entre els dos nombres es comprova si aquesta retornarà un nombre o bé un valor no numèric. Així podem evitar utilitzar aquest valor en futures operacions. Bé, en el cas que no sigui un nombre s'informa a l'usuari d'aquesta situació i en el cas que l'operació sigui correcta es realitza la divisió.

A l'exemple següent forcem la divisió entre dos zeros. Quin és el resultat?

```
let nombre = 0;
document.write(nombre/nombre);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/RwawVqw?editors=0010.

La biblioteca JavaScript ha creat un valor que indica que no és un nombre. S'ha anomenat **NaN**. Com veieu aquest és el resultat de la divisió anterior.

També s'ha creat el valor **Infinity** per representar el valor infinit positiu i **-Infinity** per representar el valor infinit negatiu.

```
let nombre = 10;
let zero = 0;
document.write(nombre/zero);
document.write("<br>");
document.write(–nombre/zero);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/zYqYwML?editors=0010.

Si us fixeu, a la primera divisió dona com a resultat el valor positiu *Infinity* però a la segona divisió dona el valor *-Infinity* ja que és la divisió d'un nombre negatiu.

Una altra funció interessant de nombres és la funció **toFixed(digits)**. Aquesta funció arrodoneix els decimals d'un número.

Veieu-ne un exemple:

```
let nombre = 1234.56789;
document.write(nombre.toFixed(1) + "<br>");
document.write(nombre.toFixed(8)+ "<br>");
document.write(nombre.toFixed()+ "<br>");
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/gOrOWZa?editors=0010.

El resultat d'executar la funció amb el paràmetre 1 indica que només es vol un decimal. Al fer arrodoniment en comptes de donar el nombre *1234,5* dona *1234,6* ja que el valor 1234,56789 està més a prop d'aquesta xifra.

En canvi, si li enviem un nombre superior als decimals que té el nombre original el resultat de l'execució afegeix tants zeros com decimals falten. En aquest cas mostra el valor *1234,56789000*.

Finalment, si no li enviem cap dígit a la funció llavors li estem indicant que no volem decimals i per això el resultat de l'execució de la funció és *1235*. Fixeu-vos que en aquest cas també s'arrodoneix la xifra.

1.1.3 Altres funcions interessants

JavaScript té unes quantes funcions bastant diferents entre sí, però que en moltes ocasions són força útils. Algunes funcions demanen la intervenció de l'usuari per realitzar una acció i d'altres serveixen per executar una acció un nombre determinat de vegades.

Quadres de diàleg entre l'usuari i el programa

N'hi ha de tres tipus: funció `alert`, funció `confirm` i funció `prompt`.

Aquestes funcions mostren un diàleg amb un missatge i un o més botons. Cal destacar que aquestes funciones interrompen l'execució del programa fins que es clica algun dels botons.

> Aquestes funcions proporcionen una manera ràpida de visualitzar i entrar dades però no s'utilitzen en el desenvolupament de pàgines web, ja que no es pot canviar l'estil i aquest sempre depèn del navegador. En el seu lloc es fan servir diàlegs modals, on la visualització

s'implementa amb HTML i CSS i el comportament dels botons s'afegeix amb JavaScript (però no s'interromp l'execució).

- **Funció alert**: s'utilitza quan es vol donar una informació a l'usuari sense esperar cap resposta d'ell. Es crearà una finestra amb un únic botó.

```
alert("Hola Món");
```

- **Funció confirm**: s'utilitza quan es vol donar l'opció a l'usuari de confirmar o no l'execució d'una tasca del programa. Es crearà una finestra amb dos botons: *True* o *False*.

```
let resposta = confirm("Vols saber la hora?");
if (resposta){
   document.write(new Date());
}
else{
   document.write("oooh.");
}
```

- **Funció prompt**: s'utilitza per demanar a l'usuari alguna informació. Es crearà una finestra amb un quadre de text on l'usuari pot introduir la informació demanada.

```
let resposta = prompt("Escriu el teu nom:");
document.write('El teu nom és: ${resposta}');
```

Podeu veure un exemple complet en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/QWNWvXM?editors=1010.

Funcions de temporització

Hi ha dues funcions de temporització: setInterval(codi_a_executar, temps_en_ms) i setTimeout(codi_a_executar, temps_en_ms):

- **setTimeout(codi_a_executar, temps_en_ms)**: executa una funció després d'un instant de temps.
  - codi_a_executar: nom de la funció que s'executarà després que hagi passat el temps *temps_en_ms* en mil·lisegons.
  - temps_en_ms: temps que trigarà a executar-se la funció *codi_a_executar*.
  - Aquesta funció retorna un *ID* (identificador) per poder cancel·lar la seva execució.

Per veure clarament que hi ha un retard en executar la funció `setTimeout` a l'exemple ens ajudem de dos botons. El primer per configurar la funció `setTimeout` i el segon per cancel·lar la seva execució. El codi és el següent:

```
<p>Exemple de la funció setTimeOut:</p>
<button onclick="funcioRetardada();">Mostra una alerta amb retard</button>
<p></p>
<button onclick="cancelaFuncio();">Cancel·la l'alerta abans que es produeixi</
    button>
```

Codi JavaScript associat a l'HTML anterior:

```
let timeoutID;

function funcioRetardada() {
  timeoutID = setTimeout(alerta, 2000);
}

function alerta() {
  alert("Hola Món!");
}

function cancelaFuncio() {
  clearTimeout(timeoutID);
}
```

Podeu veure aquest exemple, completat amb codi HTML per fer-lo més entenedor, a l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/yLOLXNj?editors=1010.

- **setInterval (codi_a_executar, temps_en_ms)**: executa una funció indefinidament. L'interval de temps entre cada execució de la funció és definit pel paràmetre *temps_en_ms*.
  - codi_a_executar: nom de la funció que s'executarà cada *temps_en_ms* mil·lisegons.
  - temps_en_ms: temps en mil·lisegons que s'esperarà entre cada execució de la funció *codi_a_executar*.
  - Aquesta funció retorna un *ID* (identificador) per poder cancel·lar la seva execució.

Per veure clarament el funcionament de la funció `setInterval`, a l'exemple, ens ajudem de dos botons. El primer per configurar la funció `setInterval` i el segon per cancel·lar la seva execució. El codi és el següent:

Codi HTML de l'exemple:

```
<p>Exemple de la funció setInterval:</p>
<button onclick="funcioReiterada();">Mostra una alerta cada 2 segons</button>
<p></p>
<button onclick="cancelaExecucio();">Cancel·la l'alerta</button>
```

Codi JavaScript associat a l'HTML anterior:

```
let intervalID;
let numExec = 1;

function funcioReiterada() {
  intervalID = setInterval(alerta, 2000);
}

function alerta() {
```

```
  alert(‘Hola, aquesta és ´lexecució número: ${numExec}‘);
  numExec++;
}

function cancelaExecucio() {
  clearInterval(intervalID);
}
```

Podeu veure aquest exemple en l'enllaç següent: http://codepen.io/ioc-daw-m06/pen/bEgLjj.

1.2 Funcions definides pel programador

Les funcions, en JavaScript, són objectes de primera classe, és a dir, coexisteixen amb qualsevol altre objecte i poden tractar-se com un d'ells. Igual que els tipus més mundans de JavaScript, les variables poden fer referència a elles, es poden declarar amb literals i fins i tot passar-se com a paràmetres d'altres funcions. La funció és la principal unitat modular d'execució. Vol dir que excepte les instruccions incrustades en el codi que s'executen mentre s'avalua les etiquetes, tota la resta està dins d'una funció.

Els objectes tenen les següents capacitats:

- Poden crear-se a través de literals.
- S'assignen a variables, entrades de matriu i propietats d'altres objectes.
- Es poden passar com a arguments per a funcions.
- Es retornen com a valors a partir de funcions.
- Tenen propietats que poden crear-se i assignar-se de forma dinàmica.

Les funcions tenen totes aquestes capacitats i, a més, tenen la capacitat que poden invocar-se.

Exemple d'una funció auto-executable:

```
let parellOsenar = (function() {
    let avui = new Date()
    if (new Date().getDate() % 2 == 0) {
        return function() { document.write("Avui és dia parell") }
    } else {
        return function() { document.write("Avui és dia senar") }
    }
})();

parellOsenar();
```

La funció `parellOsenar` retorna una altra funció segons sigui el dia parell o senar. La funció retornada s'executa i s'escriu la frase apropiada.

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/poyowyV?editors=0010.

1.2.1 Declaració

Les funcions es declaren utilitzant la següent especificació:

```
function nom([parametre,[, parametre,[..., parametre]]]) {
   [instruccions]
}
```

Les funcions literals es componen de quatre parts:

- La paraula clau `function`
- Un nom opcional que, si s'especifica, ha de ser un identificador vàlid.
- Una llista separada per comes de noms de paràmetres entre parèntesis. La llista pot estar buida, però els parèntesis han d'estar presents.
- El cos de la funció. Una sèrie d'instruccions entre claus.

El nom de la funció és opcional, perquè si no hi ha una necessitat de posar un nom a una funció concreta, no cal fer-ho.

Quan se li posa un nom a una funció, és vàlid en tot l'àmbit en el qual aquesta es declara. Si una funció es declara amb nom en el nivell superior, es crea una propietat utilitzant el nom de la funció en l'objecte `window`. Finalment, totes les funcions tenen dins una propietat oculta anomenada `name` que emmagatzema el nom de la funció com una cadena.

En el cas de les funcions assignades a variables aquesta propietat tindrà el nom de la variable i serà buida en el cas de les funcions anònimes.

Exemple de sintaxi d'una funció JavaScript:

```
function estudiar()
{
  let frase  = "A l'IOC s'estudia molt.<br>";
  return frase;
}
```

Aquest tipus de funcions es declaren per a fer-ne ús en un altre moment i tantes vegades com sigui necessari.

Proveu tot el que s'ha comentat anteriorment amb un exemple. Utilitzareu un joc de proves per veure si realment les funcions són objectes de primera classe. Les funcions que provareu seran les següents:

```
function lleugera() {
  return true;
}
```

```
var sensenom = function(){return true;};

window.esVeritat = function(){return true;};

function externa(){
  function interna(){}
}

var una_funcio = function una_altre_funcio(){return true;};
```

S'utilitzarà el joc de proves amb la biblioteca *QUnit.js*. Aquesta biblioteca té una finalitat similar a la biblioteca *JUnit* de Java. S'utilitza per definir jocs de proves.

QUnit té implementada la funció `test` i la funció `equal`. Amb la funció `test` definim un nou joc de proves i li donem un nom. La funció `equal` comprova si la igualtat que li passem com a primer paràmetre correspon amb el que nosaltres esperem (segon paràmetre), si és així, el resultat de la execució de la funció `equal` serà *okay*:

```
test("Conjunt de Tests per a funcions", function() {
  equal(typeof window.lleugera === "function", true);
  equal(lleugera.name === "lleugera", true);
  equal(typeof window.sensenom === "function", true);
  equal(sensenom.name === "sensenom", true);
  equal(typeof window.esVeritat === "function", true);
  equal(typeof window.externa === "function", true);
  equal(typeof window.interna === "function", false);
  equal(window.una_funcio.name === "una_altre_funcio", true);
});
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/ZEWEyaV?editors=1011.

Aquest joc de proves demostra que les funcions s'afegeixen com a propietats de l'objecte `window`. A més, es pot veure també la propietat `name`. El fet que `window.sensenom` es defineixi com una funció demostra que les variables globals, fins i tot les que contenen funcions, acaben amb `window`. I, finalment, cal puntualitzar que `window.esVeritat` es defineix com una funció.

Les funcions de fletxa s'utilitzen de forma semblant a les funcions Lambda en altres llenguatges com Java.

Actualment JavaScript inclou un altre tipus de funció, anomenada "funció de fletxa", que substitueix a les funcions anònimes quan el cos de la funció és molt simple. Aquest tipus de funcions és molt més concís i, quan s'utilitza adequadament, fa que el codi sigui més fàcil d'entendre.

La sintaxi de les funcions de fletxa és la següent:

```
([parametre,[, parametre,[..., parametre]]]) => { [instruccions] }
```

Per exemple:

```
const sumar = (x, y) => { return x + y };
console.log(sumar(2, 3));
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/QWNWgxB?editors=0012.

Quan el cos de la funció consisteix en el retorn d'una expressió pot eliminar-se tant el "return" com les claus. La funció anterior es podria reduir a:

```
const sumar = (x, y) => x + y;
```

1.2.2 Àmbit de les variables i de les funcions

Definició d'àmbit

Un **àmbit** (anomenat `scope` en anglès) és una zona del codi en la qual es pot accedir a una variable o funció.

Actualment a JavaScript trobem quatre tipus d'àmbits ben diferenciats:

- Espai global: en aquest espai es troben les funcions i propietats afegides pels navegadors (com `setInterval`, `alert` o `console`). Les variables i funcions declarades a l'espai global són accessibles des de qualsevol punt de l'aplicació.
- Àmbit de bloc: delimitat entre claus com en altres llenguatges com C i Java. Les variables i funcions declarades dintre d'un bloc no són accessibles fora d'aquest. S'aplica a les variables declarades amb `let` i `const`
- Àmbit de funció: aquest és el comportament tradicional de les variables a JavaScript i no es recomana utilitzar-lo. Les variables i funcions declarades dintre d'una altra funció només són accessibles dintre d'aquesta funció. S'aplica a les variables declarades amb `var`.

> Per respectar l'àmbit de bloc cal declarar sempre les variables amb **let** o **const** segons escaigui.
> Per aquest motiu **es desaconsella fer servir var**, ja que no respecta l'àmbit de bloc.

Cal tenir en compte que els àmbits es tracten com una pila, sempre es pot accedir als elements accessibles de l'àmbit superior. Veieu el següent exemple:

```
var cadena1 = "ambit global";

function prova() {
  var cadena2 = "ambit de funció";

  if (true) {
    let cadena3 = "ambit de bloc";

    console.log(cadena1);
    console.log(cadena2);
    console.log(cadena3);
  }
}
prova();
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/yLOLXrw?editors=0012.

Com es pot apreciar, dintre del bloc es continua tenint accés a l'àmbit de funció i a l'àmbit global. Si ho observem com una jerarquia en aquest exemple tindríem

`àmbit global > àmbit funció prova > bloc if (true)`. És a dir, des de la funció tenim accés a l'àmbit global i desde el bloc `if` tenim accés a l'àmbit de la funció i a l'àmbit global.

Espai o àmbit global

Hi ha altres contexts on també s'utilitza JavaScript (com Node.js) però no existeix l'objecte `window`.

Als navegadors l'espai global es troba representat per l'objecte `window`, aquest objecte es troba a tots els navegadors i conté totes les propietats i funcions globals.

Qualsevol variable declarada amb `var` en un codi que no es trobi dintre de cap funció o objecte s'afegirà a l'espai global, per exemple:

```
var a = "hola món";

function adeu() {
  var b = "adéu món"
    console.log(‘valor de a (funció): ${a}‘);
    console.log(‘valor de b (funció): ${b}‘);
}

console.log(‘valor de a: ${a}‘);
console.log(‘valor de a (window): ${window.a}‘);

try {
  console.log(‘valor de b: ${b}‘);
} catch (e) {
  console.log(e.message);
}

console.log(‘valor de b: ${window.b}‘);

adeu();
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/zYoYPLx?editors=1111.

Com es pot apreciar, el valor de `a` es troba definit a l'objecte `window` i és accessible des de qualsevol punt sense necessitat d'indicar que és una propietat d'aquest objecte. Per altra banda, el valor de `b` només es troba definit dintre de la funció `adeu` i, per consegüent, no es troba a l'espai global, no apareix com a propietat de l'objecte `window` i no és accessible fora de la funció.

Fixeu-vos que hem fet servir `var` per declarar les variables, si fem servir `let` la variable no s'afegeix a l'objecte `window`, encara que continua sent accessible globalment dintre d'aquest codi (així és com funcionen els mòduls), per exemple:

```
var a = "hola món";
let b = "adéu món";

console.log(‘valor de a: ${a}‘);
console.log(‘valor de a (window): ${window.a}‘);

console.log(‘valor de b: ${b}‘);
console.log(‘valor de b (window): ${window.b}‘);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/BaKawRy?editors=0012.

Hi ha un altre cas en el qual les variables s'afegeixen a l'espai global: si oblidem declarar una variable, aquesta es declara automàticament a l'espai global, inclús si es troba dintre d'una funció, per exemple:

```
function adeu() {
  b = "adéu món"
  console.log(`valor de b (funció): ${b}`);
}

try {
  console.log(`valor de b: ${b}`);
} catch (e) {
  console.log(e.message);
}

adeu();
console.log(`valor de b: ${b}`);
console.log(`valor de b (window): ${window.b}`);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/PoNoJKo?editors=0012.

Fixeu-vos que abans de cridar la funció el valor de `b` no està definit, però un cop s'invoca la funció `b` es declara a l'espai global. Cal tenir molt de compte amb aquests errors perquè són difícils de detectar, ja que no es tracta d'un error sintàctic.

De la mateixa manera, quan es declara una funció que no es troba dintre de cap objecte ni altres funcions directament o amb `var`, aquestes s'afegeixen a l'espai global, però com passa amb les variables, si es declaren amb `let` o `const` no s'afegeixen a `window`:

```
function hola1() {}
var hola2 = function () {};
let hola3 = function () {};
const hola4 = () => {};

console.log(`valor de b (codi): ${hola1}`);
console.log(`valor de b (window): ${window.hola1}`);

console.log(`valor de b (codi): ${hola2}`);
console.log(`valor de b (window): ${window.hola2}`);

console.log(`valor de b (codi): ${hola3}`);
console.log(`valor de b (window): ${window.hola3}`);

console.log(`valor de b (codi): ${hola4}`);
console.log(`valor de b (window): ${window.hola4}`);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/RwawLxz?editors=0012.

Com es pot apreciar, les funcions declarades amb `let` i `const` no s'han afegit a l'objecte window. Això permet encapsular correctament els mòduls, ja que mitjançant `let` i `const` les nostres variables i funcions es trobaran encapsulades dintre del mòdul encara que carreguem múltiples mòduls que facin servir els mateixos noms de variables; en canvi, si féssim servir `var`, en carregar un nou fitxer que declari variables o funcions amb el mateix nom se sobreescriurien i el programa no funcionaria o ho faria de formes inesperades.

Contaminació del espai global

Crear variables o funcions a l'espai global es considera una contaminació d'aquest espai i fer-ho es considera una mala pràctica. Es recomana utilitzar algun mecanisme com els mòduls o encapsular l'aplicació dintre d'un altre objecte per tal d'evitar-ho.

Àmbit de bloc

Quan es declaren funcions o variables amb `let` i `const`, el seu àmbit queda delimitat per les claus que les envolten o el mòdul on es declarin. És a dir, si es declara el comptador d'un bucle for amb `let` aquest només serà definit dins del bucle:

```
for (let i = 0; i < 3; i++) {
  console.log(‘valor de i (dins del bucle) ${i}‘);
}

try {
  console.log(‘valor de i (fora del bucle): ${i}‘);
} catch (e) {
  console.log(e.message);
}
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/oNYNoRW?editors=0011.

Com es pot apreciar, fora del bucle el valor de `i` no es troba definit. El mateix passa amb qualsevol altre tipus de bloc, per exemple amb un bloc `if...else`:

```
if (true) {
  let i = 3;
  const j = 4;
  console.log(‘valor de i (dins del if) ${i}‘);
  console.log(‘valor de j (dins del if) ${j}‘);
}

try {
  console.log(‘valor de i (fora del if): ${i}‘);
} catch (e) {
  console.log(e.message);
}

try {
  console.log(‘valor de j (fora del if): ${j}‘);
} catch (e) {
  console.log(e.message);
}
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/YzqzEQq?editors=0012.

Àmbit de mòdul

Es considera un mòdul el codi que es trobi dintre d'una etiqueta `script` amb el tipus `module` assignat com atribut:

Ampliar informació mòduls

Podeu trobar més informació sobre els mòduls al següent enllaç: mzl.la/3ibdoOl.

```
<script type="module">
  // Codi del mòdul
```

```
</script>
```

O quan es carrega un fitxer amb codi JavaScript indicant el tipus `module`:

```
<script type="module" src="fitxer_modul.js"></script>
```

En tots dos casos, el codi queda encapsulat dintre del mòdul, de manera que només són accessibles externament les funcions i variables quan:

- Dins del mòdul d'origen s'han exportat mitjançant la instrucció `export`.
- Al mòdul on es volen utilitzar s'han importat mitjançant la instrucció `import`.

Cal destacar que només es pot utilitzar `import` i `export` dins de mòduls, en cas contrari es produeix un error.

A continuació podeu veure un exemple d'exportació i d'importació de mòduls. Per provar-lo heu de crear un fitxer diferent per cada mòdul i un altre pel codi HTML, tots tres dins del mateix directori.

Codi del mòdul que exporta la funció (*modul.js*):

```
//modul.js
export function suma(a, b) { return a + b}

let cadena = "hola";
```

Codi del mòdul que importa la funció `suma` del mòdul *principal.js*:

```
//principal.js
import { suma } from './modul.js';

console.log(suma(2, 3));

try {
    console.log(`Valor de la cadena? ${cadena}`);
} catch (e) {
    console.log(e.message);
}
```

Codi HTML que carrega el fitxer *principal.js* que s'executa automàticament en carregar-se.

```
<html>
    <script src="principal.js" type="module"></script>
</html>
```

Com podeu comprovar la funció exportada `suma` es pot cridar des del mòdul que la importa, però la variable `cadena` no és accessible.

Àmbit de funció

Cal recordar que es desaconsella declarar les variables fent servir `var`.

Quan es declara una variable utilitzant `var` dintre d'una funció aquesta és accessible dintre de la mateixa funció independentment de si s'ha declarat dintre

d'un bloc com per exemple `for` o `if`:

```
function demostracio() {

  if (true) {
    var i = 3;
    console.log(‘valor de i (dins del if): ${i}‘);
  }

  console.log(‘valor de i (fora del if): ${i}‘);

  for (var j = 0; j<3; j++) {
    console.log(‘valor de j (dins del for): ${j}‘);
  }
  console.log(‘valor de j (fora del for): ${j}‘);

}

demostracio();
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/RwawQwE?editors=0012.

1.2.3 Invocacions

Existeixen maneres diferents d'invocar una funció i cadascuna d'elles té les seves peculiaritats. Per exemple, es pot invocar una funció de la manera habitual: *nomfunció()*. També es pot executar una funció com a mètode d'un objecte o bé es pot invocar una funció indirectament, per exemple, al crear un objecte s'invoca el mètode constructor. Aquestes maneres d'invocació de funcions es troben a tots els llenguatges orientats a objectes, però a JavaScript, a més a més, es poden invocar utilitzant dos mètode addicionals: ***apply()*** i ***call()***.

En totes aquestes maneres d'invocar una funció, a la majoria de llenguatges, el nombre d'arguments i paràmetres ha de ser el mateix. A JavaScript, però, això és diferent. Si hi ha més arguments que paràmetres, els arguments de més no s'assignen als noms de paràmetres i la invocació de la funció és correcta i no dona error.

```
function qualsevol(a,b,c){}
qualsevol(1,2,3,4,5,6); //4,5,6 no s'assignen a cap paràmetre.
```

Si, en canvi, hi ha més paràmetres que arguments, els paràmetres que no tinguin el seu argument corresponent s'estableixen com `undefined`.

```
function qualsevol(a,b,c){}
qualsevol(1); //b,c tenen el valor undefined
```

A JavaScript, a totes les invocacions de les funcions es passen 2 paràmetres extres (es passen en silenci i estan en el seu àmbit): `arguments` i `this`. El paràmetre **arguments** és una col·lecció de tots els paràmetres que s'han passat a la funció i té una propietat anomenada `length` que conté el numero de paràmetres que s'han passat. Els valors dels paràmetres es poden obtenir com si fos un *array*. En canvi,

el paràmetre **this** és el context de la funció. A Java `this` és la instància de la classe en la qual es defineix el mètode. A JavaScript és una mica diferent perquè es defineix segons com s'invoca la funció, és a dir, es defineix quan s'executa no quan es declara la funció.

Invocació com una funció

La invocació *com una funció* és la manera "normal" d'invocar una funció en qualsevol llenguatge.

Per exemple:

En el cas de la declaració de funcions s'acostuma a utilitzar `const` perquè el valor no ha de canviar.

```
function cridam(){};
cridam();
const unaltre = function(){};
unaltre();
```

Quan es declara com una funció sense indicar `let` ni `const`, la funció s'afegeix com a mètode de l'objecte `window`, per consegüent és possible invocar-lo a partir d'aquest objecte:

```
function cridam(){};
window.cridam();
```

En canvi, si provem d'executar la funció declarada com a `const` es produirà un error, ja que no s'afegeix a l'objecte `window`, només és accessible dintre del mòdul:

```
const unaltre = function(){};
window.unaltre();
```

De la mateixa manera es poden executar funcions de fletxa:

```
const saludar = () => "hola";
let text = saludar();
document.write(text);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/QWNWQeM?editors=0010.

Invocació com un mètode

La invocació *com un mètode* es produeix quan s'assigna una funció a la propietat d'un objecte. Exemple:

```
//es crea l'objecte anomenat 'o'
let o = {};

//es defineix una propietat anomenada 'nom_metode' i se li assigna una funció.
//aquesta propietat s'ha convertit en un mètode.
o.nom_metode = function(){return "hola";};

//crida al mètode.
let text = o.nom_metode();

document.write(text);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/ExKxQBO?editors=0010.

Quan invoquem d'aquesta manera a la funció, l'objecte es converteix en el context de la funció, és a dir, el paràmetre implícit **this** apunta a l'objecte.

L'execució de funcions de fletxa es fa de la mateixa manera, l'única diferència és que la seva declaració és més concisa:

```
let o = {};
o.nom_metode = () => "hola";
let text = o.nom_metode();
document.write(text);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/oNxNqgK?editors=0010.

La paraula clau`this` té en JavaScript un comportament diferent al d'altres llenguatges però en general, el seu valor fa referència al **propietari** de la funció que l'està invocant.
Quan no estem dins d'una estructura definida el propietari d'una funció és sempre el context global. En el cas dels navegadors web, s'ha de recordar que aquest objecte és `window`:

```
console.log(this === window);  // true

function test(){
  console.log(this === window);
}

test(); // true
```

La variable `this`, normalment, s'utilitza per accedir als valors d'un objecte des del propi objecte. Per exemple, penseu en un objecte amb una sèrie de propietats:

```
let obj = {
  nom: 'Ramon',
  cognom: 'Llull',
  naixement: '1232',
  isMallorqui: true
};

document.write(obj.nom); // Ramón
document.write(obj.isMallorqui); // true
```

Suposem ara que necessitem una altra propietat més 'dinàmica' que participi dels valors assignats a qualsevol altra propietat. Per exemple, volem un `nomSencer` que uneixi `nom` i `cognom`. Sembla, a priori, que podríem utilitzar la variable `this`:

```
let obj = {
  nom: 'Ramon',
  cognom: 'Llull',
  naixement: '1232',
  isMallorqui: true,
  nomSencer: this.nom + " " + this.cognom
};
document.write(obj.nom + "<br>"); // Ramon
document.write(obj.isMallorqui + "<br>"); // true
```

```
document.write(obj.nom+ "<br>"); // Ramon
document.write(obj.nomSencer + "<br>"); // Ramon Llull?
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/QWNWmNO?editors=0010.

Encara que sembla coherent, quan passem a comprovar-ho veiem que el resultat no és l'esperat:

```
Ramon
true
Ramon
undefined undefined
```

El problema és que `this` no apunta a l'objecte anomenat `obj` sinó que `this` apunta al context global, és a dir, a l'objecte `window`.

Per obtenir el resultat esperat hem d'aplicar un patró d'invocació que modifiqui al propietari des del qual s'invoca el `this`.

En el desenvolupament d'aplicacions modernes, el patró més recurrent és el d'invocació per mètode. Quan una funció és emmagatzemada com a propietat d'un objecte es converteix en el que anomenem mètode.

Quan invoquem a un mètode, `this` fa referència al mateix objecte:

```
let obj = {
  nom: 'Ramon',
  cognom: 'Llull',
  naixement: '1232',
  isMallorqui: true,
  nomSencer: function() { return `${this.nom}  ${this.cognom}`;}
};
document.write(obj.nom + "<br>"); // Ramón
document.write(obj.isMallorqui + "<br>" ); // true
document.write(obj.nom+ "<br>" ); // Ramón
document.write(obj.nomSencer() + "<br>" ); // Ramon Llull?
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/yLOLKaE?editors=0010.

En aquesta ocasió, podem comprovar com `this` apunta al mateix objecte i busca les propietats `nom` i `cognom` en comptes d'anar fins el context global.

Invocació amb els mètodes apply() i call()

JavaScript ens proporciona un mitjà per invocar una funció i especificar de forma explícita qualsevol objecte que vulguem com a context. És a dir, podem decidir quin és el valor del paràmetre `this` quan cridem a una funció.

Els mètodes apply() i call()

Totes les funciones tenen aquest dos mètodes disponibles ja que són objectes de primera classe i poden tenir propietats i mètodes igual que qualsevol altre objecte.

S'aconsegueix amb qualsevol dels dos mètodes que posseeixen les funcions: *apply()* i *call()*.

Paràmetres del mètode *apply()*:

- L'objecte que s'utilitzarà com a context de la funció
- *Array* de valors que s'utilitzaran com a arguments de la funció.

Paràmetres del mètode *call()*:

- L'objecte que s'utilitzarà com a context de la funció
- Llista de valors que s'utilitzaran com a arguments de la funció.

Exemple:

```
function exemple() {
  let total = 0;
  for (let i = 0; i < arguments.length; i++) {
    total += arguments[i];
  }

  this.result = total;
};

let objecte1 = {};
let objecte2 = {};

exemple.apply(objecte1, [1, 2, 3, 4]);
exemple.call(objecte2, 5, 6, 7, 8);

// en aquest punt objecte1.result tindrà el valor 10
// i objecte2.result tindrà el valor 26
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/QWNWmmr?editors=0010.

La funció `exemple` utilitza la paraula reservada `this` per crear una propietat anomenada `total` que conté el resultat de l'execució de la pròpia funció.

Si executem la funció la variable `this` apunta a l'objecte `window`, llavors, la propietat `result` seria una propietat global. Però en utilitzar els mètodes `apply` o `call` canviem l'objecte on apunta la variable `this`.

En el primer cas, amb el mètode `apply`, la variable `this` apunta l'objecte1 (ja que aquest objecte és el primer paràmetre del mètode `apply`).

En el segon cas, la variable `this` apunta l'objecte2. Llavors quan la funció `exemple` executa l'última instrucció: *this.result = result;* està creant una propietat anomenada `result` a l'objecte1 o a l'objecte2, segons sigui el cas, i està guardant el resultat de l'execució del codi anterior.

Canvi de context d'una funció amb el mètode "bind"()

Una altra manera de canviar el context d'execució d'una funció és utilitzar el mètode `bind(context)`. Aquest mètode **crea una nova funció amb el context passat com a paràmetre**, això ens permet utilitzar-lo de dues formes:

- Assignant la funció retornada com una nova funció, això permet executar-la sempre amb el nou context.

- Invocar-la directament, de manera que s'executa només una vegada amb el nou context.

Veieu aquests dos casos en el següent exemple:

```
<html>
<script>
    let persona1 = {
        nom: 'Joan',
        saludar: function () {
            console.log(`Hola!, em dic ${this.nom}`);
        }
    };

    let persona2 = {
        nom: 'Maria',
    };

    // la funció s'executa en el seu context, persona1
    persona1.saludar();

    // la funció s'executa en el context de persona2
    persona1.saludar.bind(persona2)();

    // afegim una nova funció saludar amb el nou context
    persona1.saludarNou = persona1.saludar.bind(persona2);

    // el context per aquesta funció és persona 2
    persona1.saludarNou();

    // comprovem que si canviem les propietats del context, la crida a la funci
        ó continua sent correcte
    persona2.nom = "Rosa";
    persona1.saludarNou();

    // comprovem que saludar continua utilitzant el seu context original
    persona1.saludar();
</script>
</html>
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/VwawRKY?editors=0012.

Com es pot apreciar, quan s'invoca la funció `saludar` amb `bind(persona2)` la funció s'executa en el context de `persona2` però subsegüents invocacions a la funció `saludar` continuant utilitzant `persona1` com a context. Per altra banda quan assignem la funció retornada per `bind`, totes les invocacions a aquesta nova funció es fan utilitzant el context de `persona2`.

Fixeu-vos que encara que canviïn els valors de `persona2` quan es crida a la funció `saludarNou` el resultat continua sent correcte perquè el context és una referència a `persona2` i no una còpia.

1.2.4 Exemple: memoritzar valors calculats a la pròpia funció (Memoize)

La memorització (*memoization*) és el procés de crear una funció que pot recordar els seus valors calculats. Així es pot incrementar considerablement el rendiment evitant càlculs complexos innecessaris que ja s'han calculat.

Per exemple, escrivim una funció que ens digui si un enter positiu és o no primer:

```
function esPrimer(valor) {
  if (!esPrimer.cache) {
    esPrimer.cache = {}; // si no hi ha cache de resultats, la creem
  }

  if (esPrimer.cache[valor] != null) {
    // resultat ja calculat
    return `${valor} és primer (cache)? ${esPrimer.cache[valor]}`;
  }

  // cal calcular-lo perquè no tenim el resultat

  let primer = valor != 1; // 1 no és primer
  // busquem un divisor superior a 2 i inferior a valor

  for (let i = 2; i < valor; i++) {
    if (valor % i == 0) {
      primer = false;
      break;
    }
  }

  esPrimer.cache[valor] = primer;
  return `${valor} és primer (calculat)? ${primer}`;
}
console.log(esPrimer(5));
console.log(esPrimer(5));
console.log(esPrimer.cache);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/wvGvZdY?editors=0012.

La segona vegada que es crida a la funció entra dins del segon `if` perquè el troba en memòria i retorna el resultat directament.

Fixeu-vos que, donat que les funcions també són objectes, hem pogut crear una nova propietat anomenada `cache` directament a la funció. Es pot accedir a aquesta propietat com si es tractés de qualsevol altre objecte, en aquest cas amb `esPrimer.cache`. Per aquest mateix motiu és accessible des de fora de la funció, ja que les propietats a JavaScript són públiques.

1.2.5 Sobrecàrrega de funcions

En altres llenguatges de programació orientats a objectes, per sobrecarregar una funció se sol declarar diferents implementacions de mètodes amb el mateix nom

però amb un conjunt diferent de paràmetres. En JavaScript no es fa d'aquesta manera. En JavaScript se sobrecarreguen les funcions amb una única implementació que modifica el seu comportament mitjançant l'examen del nombre d'arguments que l'han proporcionat.

És fàcil d'imaginar que es podria implementar la sobrecàrrega de funcions utilitzant una estructura del tipus *if-then i else-if*. Però no sempre ho podrem fer.

Exemple d'una funció sobrecarregada de manera monolítica:

```
let persona = {
  calculMatricula = function(){
    switch(arguments.length){
      case 0:
        // fer algo
        break;
      case 1:
        // fer una altre cosa
        break;
      case 2:
        // fer una altre cosa més
        break;
      ... etc ...
    }
  }
}
```

Exemple: tècnica per sobrecarregar funcions

Veurem una tècnica per ens permet crear diverses funcions (aparentment amb el mateix nom, però es diferencia pel número de paràmetres) que poden escriure's com diferents, anònimes i independents i no com un bloc monolític *if-then-else-if*.

Tot això depèn d'una propietat poc coneguda de les funcions: la propietat `length`.

Algunes consideracions inicials a tenir en compte:

- El paràmetre `length` de funció no ha de confondre's amb la propietat `length` del paràmetre arguments.
- El paràmetre `length` de funció ens indica el número total de paràmetres formals amb els quals s'ha declarat la funció.
- La propietat `length` del paràmetre `arguments` ens indica el número total de paràmetres que s'han passat a la funció en el moment de cridar-la.

Exemple:

```
function max(a,b){
   ...
}

max(1,4,5,7,23,234);
```

En aquest cas el paràmetre `length` de la funció `max` és 2 i la propietat `length` del paràmetre `arguments` és 6.

Utilitzarem aquest paràmetre per crear funcions sobrecarregades.

```
function afegirMetode(objecte, nom, funcio){
  let old = objecte[nom];
  objecte[nom] = function(){
      if(funcio.length == arguments.length){
          return funcio.apply(this, arguments);
            }
      else  if (typeof old == 'function'){
          return old.apply(this, arguments);
            }
    };
}


let persona = {};
afegirMetode(persona, "calculMatricula", function(){ //fer algo  });
afegirMetode(persona, "calculMatricula", function(a){ //fer una altre cosa  });
afegirMetode(persona, "calculMatricula", function(a,b){  //fer una altre cosa m
    és  });
```

La funció `afegirMetode` s'utilitza per afegir un mètode a un objecte. Aquesta funció rep tres paràmetres:

- L'objecte al qual volem afegir un mètode
- El nom del mètode que volem afegir
- La funció associada al nom del mètode anterior

La funció `afegirMetode` utilitza el mètode `apply` per afegir a un objecte la pròpia funció `afegirMetode`. Aquesta funció és recursiva ja que tots els mètodes afegits d'aquesta manera tenen disponible la variable `old`, que guarda els mètodes amb els mateixos noms però amb un nombre de paràmetres diferent. Noteu que en aquest cas la variable `this` apunta a la funció `afegirMetode`.

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/MWyWRPa?editors=0010.

1.2.6 Clausures

A JavaScript és possible crear funcions dintre de funcions i això genera un *àmbit lèxic*, és a dir, les funcions internes tenen accés a l'àmbit de la funció externa (per exemple, variables i paràmetres). Aquesta característica de JavaScript ens permet crear *clausures*:

Una clausura és la combinació d'una funció i l'entorn lèxic en el qual aquesta funció és declarada amb les següents característiques:

- Té accés a les variables i funcions definides a la funció interna (àmbit de funció).
- Té accés a totes les variables i funcions declarades en la funció externa, incloent-hi els paràmetres amb què es va invocar la funció (àmbit lèxic).
- Té accés a l'àmbit global.

Veieu un exemple:

```
// funció externa
function inicialitzacio() {
  let nom = 'Joan';
  // funció interna
  function mostrarNom() {
    console.log(nom);
  }
  mostrarNom();
}
inicialitzacio();
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/jOqOoBq?editors=0012.

Fixeu-vos que la variable `nom` es declara a la `inicialitzacio` però és accessible des de la funció `mostrarNom` i que aquesta funció és invocada per inicialització.

Com que a JavaScript és possible retornar una funció, podríem retornar la funció `mostrarNom` i aquesta funció continuaria tenint accés a l'àmbit de la funció externa com podeu veure en el següent exemple:

```
// funció externa
function inicialitzacio() {
  let nom = 'Joan';
  // funció interna
  function mostrarNom() {
    console.log(nom);
  }
  return mostrarNom;
}
let mostrar = inicialitzacio();
mostrar();
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/bGpGyrd?editors=0012.

Quan s'assigna el valor a la variable `mostrar` s'executa la funció `inicialitzacio`, s'assigna el valor a la variable `nom` i es retorna la funció `mostrarNom`; així, doncs, el valor de la variable `mostrar` és la funció `mostrarNom` amb accés a l'àmbit de `inicialitzacio`.

Fixeu-vos que el valor de `nom` no és accessible des de fora de la funció, per tant es pot considerar que els valors declarats a la funció externa són `privats`.

> L'únic mecanisme que proporciona JavaScript per encapsular valors de manera el seu accés sigui **privat** és mitjançant **clausures**.

Una altra característica remarcable és que l'àmbit es crea en el moment en què s'invoca la funció, això vol dir que si cridem a una mateixa funció amb diferents paràmetres, cada funció retornada continuarà tenint accés a l'àmbit existent quan es va invocar la funció externa:

```
// funció externa
function crearTaula(multiplicador) {
  // funció interna
  function multiplicar() {
    for (let i = 1; i<=10; i++) {
      console.log(`${i} x ${multiplicador} = ${i * multiplicador}`);
    }
  }
  return multiplicar;
}

const taulaDel5 = crearTaula(5);
taulaDel5();

const taulaDel9 = crearTaula(9);
taulaDel9();
taulaDel5();
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/LYNYomX?editors=0012.

Com es pot apreciar, s'ha assignat a les variables `taulaDel5` i `taulaDel9` el retorn de la funció `crearTaula` invocada amb diferents arguments i quan s'invoca a qualsevol d'aquestes funcions el retorn és l'esperat, a l'àmbit de `taulaDel5` el valor de `multiplicador` és 5, mentre que al de `taulaDel9` és 9.

1.2.7 Funcions sense nom. Definició i usos

Les funcions **sense nom**, o anònimes, les fem servir quan volem poder disposar d'elles posteriorment. Per exemple, quan les emmagatzemem en una variable, quan les posem com a mètodes d'objectes o bé quan les utilizem com a *callback* (callback de timeout, callback de controladors d'esdeveniments, etc).

> Un **callback** és un pedaç de codi executable (una funció) que es passa com a paràmetre a un altre codi, que s'espera per executar-lo. La invocació pot ser immediata (síncrona), o pot ocórrer en un moment posterior (asíncrona).

En tots aquests casos, les funcions no necessiten tenir un nom per poder invocar-les.

```
window.onload = function() {alert("1");};

let obj = {print : function() {alert("2");}}
```

```
obj.print();

setTimeout(function(){ alert("3"); }, 500);
```

També podíem haver fet amb el controlador de l'esdeveniment de càrrega de pàgina:

```
function salutacio(){
  document.write("Hola estudiants de l'IOC!");
};

window.onload = salutacio;
```

Però no cal donar-li nom si mai més l'utilitzarem. Realment no necessitem que `salutacio` sigui un mètode de l'objecte `window`. A més a més, podem pensar que el mètode `print` és el nom de la funció anònima que li assignem, però no ho és.

Una funció anònima es pot definir sense assignar-li un nom ni assignar-la a cap variable, però en fer-ho d'aquesta manera no es pot invocar. Per consegüent el codi és vàlid però no és correcte:

```
function () {
  document.write("Hola estudiants de l'IOC!");
}
```

De vegades ens interessa poder posar nom a les funcions anònimes. Normalment, les funcions sense nom o anònimes les utilitzem quan creem els mètodes dels objectes. El mètode ha de tenir un nom, la seva funció no.

```
let persona = {
    cantar : function (){return 1;}
}
console.log(persona.cantar.name);
```

En aquest exemple s'ha declarat un objecte anomenat `persona`, que conté un mètode anomenat `cantar`.

Malgrat que aquest mètode s'ha declarat com una funció anònima, automàticament s'ha assignat el nom de la propietat com a nom de la funció.

Tot i que no és gaire utilitzat, JavaScript permet anomenar una funció assignada a una propietat, en aquest cas el nom de la funció serà l'indicat i no pas el de la propietat:

```
let persona = {
  cantar : function(n){ return n > 1 ? this.cantar(n - 1) + "-fiu" : "fiu"; },
  cantar2 : function xiular(n){ return n > 1 ? xiular(n - 1) + "-fiu" : "fiu";
      }
}
console.log("Cridem al mètode cantar que a la seva vegada crida a la funció
    xiular: persona.cantar(3):");
console.log(persona.cantar(3));
console.log(persona.cantar2(3));
console.log(persona.cantar.name);
console.log(persona.cantar2.name);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/NWNWZGj?editors=0012.

1.2.8 Funcions de fletxa

Les funcions de fletxa proporcionen una manera molt més concisa d'utilitzar les funcions anònimes; d'aquesta manera es poden assignar a variables, mètodes o, també, utilitzar com a *callbacks*:

```
window.onload = () => console.log("1");

let obj = {print: () => console.log("2")};
obj.print();

setTimeout(() => console.log("3"), 500);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/eYZYwWP?editors=0012.

Si el cos de la funció tingués més d'una instrucció, caldria posar-les entre claus.

Utilització de funcions de fletxa

No es recomana utilitzar aquestes funcions quan el cos de la funció sigui llarg, ja que l'avantatge principal és fer el codi més llegible. Si utilitzar una funció de fletxa el codi no ho fa més entenedor, és preferible fer servir la implementació habitual.

Com podeu veure a continuació hi ha casos en què la utilització d'aquest tipus de funció fa el codi més entenedor, ja que mostren només la informació rellevant en funcions molt simples (com `sumar`) o permeten escriure una funció de callback en una sola línia:

```
// Assignat com a funció
const sumar = (a, b) => a + b;

// Utilitzat com a callback
let tick = 0;
setInterval(() => {console.log( tick % 2 == 0 ? "Tic" : "Tac");tick++;}, 1000);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/oNxNrog?editors=0012.

En els casos en què només hi ha un paràmetre encara es poden simplificar una mica més les funcions de fletxes, ja que no són necessaris els parèntesis:

```
const duplicar = a => a * 2;
console.log(duplicar(2));
```

1.2.9 Funcions immediates

Una funció immediata es basa en el concepte de les clausures.

Exemple de funció immediata:

```
(function(){})()
```

Analitzarem la construcció de la funció ignorant el primer grup dels parèntesis.

```
(instruccions)()
```

Sabem que podem fer la crida d'una funció utilitzant la sintaxi *functionName()*, però en lloc del nom podem utilitzar qualsevol expressió que es refereixi a una de les seves instàncies.

```
let estudiar = function(){instrucions};
result = estudiar();
// o també podem fer:
result = (estudiar)();
```

Això significa que a **(funció)()**, el primer joc de parèntesis és un limitador que tanca una expressió. El segon joc de parèntesis és un operador.

Ara en lloc d'una variable, hi posem la funció anònima directament.

```
(function(){instruccions})();
```

Aquesta funció s'instancia, tot seguit s'executa i, finalment, es descarta. Per exemple, es poden utilitzar aquest tipus de funcions per crear un àmbit temporal que emmagatzemi l'estat:

```
(function(){
  let numclicks = 0;
  document.addEventListener("click", function(){alert(++numclicks);}, false);
})();
```

L'important és observar que es crea una clausura pel controlador que inclou `numclicks`, llavors només ell pot fer referència a aquesta variable. Ningú més podrà modificar el seu valor.

Aquesta és una de les formes d'ús comú de les funcions immediates: com embolcalls simples e independents.

També es poden passar paràmetres a les funcions immediates incloent-los al final. Per exemple:

```
(function(salutacio){alert(salutacio);})("Hola");
```

1.2.10 Les funcions niades

A l'estructura que esdevé al declarar funcions una dintre d'altres s'anomena estructura de funcions niades. Exemple:

```
function estudiar(vegades) {

    function molt(frase) {
        document.write(frase)
    }
    molt('Estudieu més de ${vegades} hores.');

}
estudiar(10);
```

Podem combinar aquesta estructura amb el retorn de les funcions. La funció pare pot escollir una de les funcions internes i retornar-la. Les clausures permeten mantenir la informació que hi ha dintre dels paràmetres o de les variables a les que té accés la funció interna.

```
function estudiar(frase) {

    function molt() {
        document.write(frase)
    }
    return molt;

}

const ioc = estudiar('Estudiant a ´lIOC ´saprèn molt.')

ioc()
```

Podem combinar tot el que hem après sobre les funcions. Podem crear-les anònimes i auto-executables.

```
const estudiar = (function(frase) {
    function molt() {
        alert(frase)
    }

    return molt
})("Estudiant a l'IOC s'aprèn molt.")

estudiar();
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/abNzbLr?editors=0010.

1.2.11 Desestructuració i assignació de valors per defecte als paràmetres

Actualment JavaScript permet la desestructuració d'objectes i arrays, això significa que es poden assignar directament els valors d'un array o objecte a un grup de

variables:

```
// Desestructuració d'un array
let colors = ['verd', 'ambar', 'vermell'];
let [pasar, perill, prohibit] = colors;
console.log(pasar, perill, prohibit);

// Desestructuració d'un objecte
let persona = {nom: "Joan", poblacio: "Barcelona"};
// variables amb el mateix nom que les propietats
let {nom, poblacio} = persona;
console.log(nom, poblacio);

// variables amb diferent nom que les propietats
let {nom: a, poblacio: b} = persona;
console.log(a, b);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/zYqYVeg?editors=0012.

Fixeu-vos que a la banda esquerra s'indiquen les variables a les quals s'assignarà el valor i a la dreta l'objecte o array que es desestructurarà.

En el cas dels arrays, cal indicar els noms de les variables entre claudàtors i els valors s'assignaran en ordre (l'índex 0 a la primera variable, l'índex 1 a la segona, l'índex 2 a la tercera, etc.).

Per altra banda, en el cas dels objectes, s'indicaran els noms entre claus i, en cas que els noms de les variables siguin diferents dels noms de les propietats, caldrà indicar-los separats per dos punts, separant els parells de propietat i variable per una coma.

Aquesta mateixa desestructuració es pot aprofitar per passar arguments amb nom a les funcions:

```
function saludar({ nom, poblacio }) {
  console.log(`Hola ${nom} de ${poblacio}`);
}

let joan = {nom: "Joan", poblacio: "Barcelona"};
let maria = {nom: "Maria", poblacio: "Lleida"};

saludar(joan);
saludar(maria);
saludar({nom: "Pere", poblacio: "Tarragona"});
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/PoNoMqQ?editors=0012.

Fixeu-vos que, en lloc d'indicar un nom de paràmetre, el que hem fet és indicar una desestructuració d'un objecte i, per tant, els paràmetres `nom` i `població` rebran el valor de les propietats corresponent a l'objecte que es passi com argument.

Una altra característica de les desestructuracions és que permeten assignar valors per defecte pels casos en què l'objecte reestructurat no contingut cap valor per assignar:

```
// Desestructuració d'un objecte
let persona = {nom: "Joan"};
```

```
// variables amb el mateix nom que les propietats
let {nom, poblacio = "desconegut"} = persona;
console.log(nom, poblacio);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/poyoMNm?editors=0012.

Com es pot apreciar, quan l'objecte desestructurat no conté la propietat `poblacio` (com en aquest exemple), el valor assignat és `desconegut`.

Això ens permet crear funcions amb paràmetres que acceptin valors per defecte:

```
function saludar({nom, poblacio = "desconegut"}) {
  console.log(`Hola ${nom} de ${poblacio}`);
}
let maria = {nom: "Maria", poblacio: "Lleida"};
saludar(maria);
saludar({nom: "Pere"});
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/zYqYgwr?editors=0012.

Així doncs, quan invoquem a la funció `saludar` amb un objecte que no contingui la propietat `poblacio` s'assigna el valor `desconegut` i, en cas contrari, s'assigna el valor de la propietat.

1.2.12 Propagació i retorn de múltiples valors

Utilització avançada de paràmetres i retorns

Podeu trobar més exemples avançats d'utilització de la desestructuració i la propagació al següent enllaç: https://bit.ly/3ifbZq1.

En alguns casos pot ser necessari que una funció retorni més d'un valor. En aquest cas es pot fer retornat un objecte o un array.

Per exemple, retornant un array:

```
function getCoordenades() {
  let coordenades = [2, 4];
  return coordenades;
}

let novesCoordenades = getCoordenades();
console.log(`Primera coordenada: ${novesCoordenades[0]}`);
console.log(`Segona coordenada: ${novesCoordenades[1]}`);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/BaKyaBz?editors=0012.

O retornant un objecte:

```
function getCoordenades() {
  let coordenades = {x: 2, y: 4};
  return coordenades;
}

let novesCoordenades = getCoordenades();
console.log(`Primera coordenada: ${novesCoordenades.x}`);
console.log(`Segona coordenada: ${novesCoordenades.y}`);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/LYNEYYv?editors=0012.

Com es pot apreciar, en tots dos casos cal accedir a l'element de l'array o la propietat de l'objecte respectivament, però, gràcies a l'operador de propagació (que es representa per tres punts suspensius, ...), és possible desestructurar el retorn:

```
function getCoordenades() {
  let coordenades = {a: 1, x: 2, y: 4, z: 3};
  return {...coordenades};
}

let {x, y} = getCoordenades();
console.log(‘Primera coordenada: ${x}‘);
console.log(‘Segona coordenada: ${y}‘);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/wvGBvKg?editors=0012.

Fixeu-vos que al retorn es troba entre claus i s'ha fet servir l'operador de propagació. D'aquesta manera en invocar la funció `getCoordenades` s'assigna automàticament el valor de les propietats a les variables `x` i `y` com si es tractés d'una desestructuració.

2. Programació amb col·leccions

Les col·leccions són tipus de dades compostos que ens permeten treballar amb grups de dades en lloc de treballar amb dades individuals. Per exemple:

```
// dades individuals
let persona1 = {nom: "Joan", poblacio: "Barcelona"};
let persona2 = {nom: "Maria", poblacio: "Lleida"};
let persona3 = {nom: "Pere", poblacio: "Tarragona"};
let persona4 = {nom: "Lluisa", poblacio: "Girona"};

// col·lecció
let persones = [persona1, persona2, persona3, persona4];
```

Quan s'ha de treballar amb grups de dades no és viable treballar amb variables, ja que això requereix crear una nova variable per a cada valor i es dificulta el processament, ja que s'ha de modificar cada variable individualment.

En canvi, una col·lecció consisteix en un grup de dades, la qual cosa permet afegir dinàmicament a una mateixa col·lecció qualsevol quantitat de dades i processar tota la col·lecció completa a partir d'una única variable (a la qual està assignada la col·lecció).

Ampliant l'exemple anterior, a partir de la variable `persones`, a la quals s'ha assignat un array d'objectes, podem accedir a qualsevol dels elements. Per exemple, es poden recórrer tots els elements:

```
for (let persona of persones) {
  console.log(`${persona.nom} viu a ${persona.poblacio}`);
}
```

En aquest cas es recorren tots els elements de la col·lecció `persones` i s'assigna l'element actual a `persona` (aquest valor s'actualitza en cada iteració del bucle). Seguidament es mostra per la consola el missatge amb el nom i la població de l'element.

Una alternativa fent servir variables individuals podria ser la següent:

```
console.log(`${persona1.nom} viu a ${persona1.poblacio}`);
console.log(`${persona2.nom} viu a ${persona2.poblacio}`);
console.log(`${persona3.nom} viu a ${persona3.poblacio}`);
console.log(`${persona4.nom} viu a ${persona4.poblacio}`);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/PoNwEXd?editors=0012.

Com es pot apreciar, d'aquesta manera es duplica el codi per cada element. És a dir, si afegim 10 persones més, també hauríem d'afegir 10 línies més.

Vegem un altre exemple:

```
// Utilitzant variables individuals
let nota1 = 6;
let nota2 = 4;
let nota3 = 5;
let nota4 = 8;
let nota5 = 7;
let nota6 = 4;
let nota7 = 9;
let nota8 = 8;
let nota9 = 5;
let nota10 = 6;

let mitjana = (nota1 + nota2 + nota3 + nota4 + nota5 + nota6 + nota7 + nota8 +
    nota9 + nota10) / 10;
console.log(‘Nota mitjana: ${mitjana} (variables)‘)

// Utilitzant una col·lecció
let notes = [6, 4, 5, 8, 7, 4, 9, 8, 5, 6];
let acumulat = 0;
for (let nota of notes) {
  acumulat += nota;
}
console.log(‘Nota mitjana: ${acumulat/notes.length} (col·leció)‘);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/qBZEpJm?editors=0012.

Com es pot apreciar, quan treballem amb col·leccions el codi és molt més entenedor i s'eviten repeticions. A més a més, aquest codi és escalable, ja que, si afegim nous valors a la col·lecció `notes` (per exemple, utilitzant un formulari HTML), el codi continuaria processant les dades correctament; en canvi utilitzant variables individuals això no seria possible.

2.1 Col·leccions

Diccionaris i Map/Object

Un diccionari és una col·lecció desordenada que fa servir parelles de clau i valor per accedir als seus elements. Per aquest motiu, es pot considerar que tant Map com Object (quan s'utilitza en aquest context) són diccionaris.

Per treballar amb col·leccions, JavaScript ens proporciona els objectes predefinits Array, Map i Sets:

- **Array**: un array és una llista de valors ordenats on cada valor és **accessible mitjançant un índex**. Aquest índex sempre és un nombre enter igual o superior a 0.
- **Map**: és una col·lecció desordenada que permet associar parells de clau i valor. La clau pot ser qualsevol cosa (incloent-hi funcions, objectes i tipus primitius).
- **Set**: és una col·lecció de valors únics on es pot iterar sobre els elements en ordre d'inserció, s'hi pot afegir elements, se n'hi pot eliminar i es pot comprovar si hi existeixen; no es pot, però, modificar la seva posició ni accedir-hi directament, ja que no existeix ni un índex ni una clau: és referèncien directament els valors.

Altres col·leccions

Existeixen també els objectes WeakSet i WeakMap que són variants de Set i Map amb uns usos molt més concrets, podeu trobar més informació al següent enllaç: mzl.la/3gDAnRR.

Addicionalment, el comportament dels objectes de JavaScript ens permet utilitzar-

los d'una forma similar a un Map, ja que es pot utilitzar el nom de la propietat com a clau i el valor com a valor de l'element.

Veiem un exemple de cada tipus de col·lecció:

```
let exempleArray = ["Verd", "Ambar", "Vermell", "Vermell"];

let exempleSet = new Set();
exempleSet.add("verd");
exempleSet.add("ambar");
exempleSet.add("vermell");
exempleSet.add("vermell");

let exempleMap = new Map();
exempleMap.set("verd", "Passar");
exempleMap.set("ambar", "Compte!");
exempleMap.set("vermell", "Prohibit");
exempleMap.set("vermell", "No passar!");

let exempleObject = {
  verd: "Passar",
  ambar: "Compte!",
  vermell: "Prohibit",
  vermell: "No passar!"
};

console.log(exempleArray, exempleArray.length);
console.log(exempleSet, exempleSet.size);
console.log(exempleMap, exempleMap.size);
console.log(exempleObject, Object.keys(exempleObject).length);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/QWNwQVE?editors=0012.

Com es pot apreciar, l'única col·lecció que permet elements repetits és l'Array, tots els altres tipus es descarten els valors duplicats conservant l'última assignació.

Fixeu-vos que en el cas de l'objecte no existeix una propietat o mètode per obtenir la mida directament, hem hagut d'utilitzar la funció estàtica `Object.keys` per obtenir un array amb totes les propietats i llavors consultar el valor de la propietat `length` d'aquest array.

A continuació podeu veure les diferents maneres d'iterar sobre aquestes col·leccions:

```
console.log("Iteració array: utilitzant l'índex");
for (let i = 0; i < exempleArray.length; i++) {
  console.log(exempleArray[i]);
}

console.log("Iteració array: accedint directament al valor");
for (let element of exempleArray) {
  console.log(element);
}

console.log("Iteració set");
for (let element of exempleSet) {
  console.log(element);
}

console.log("Iteració object");
for (let nom in exempleObject) {
  console.log(exempleObject[nom]);
}
```

```
20
21 console.log("Iteració map");
22 for (let element of exempleMap) {
23   console.log(element);
24 }
25
26 console.log("Iteració map: desesctructurat");
27 for (let [clau, valor] of exempleMap) {
28   console.log(‘Clau: ${clau}, valor: ${valor}‘);
29 }
```

Per iterar sobre els valors d'un array es pot utilitzar un bucle `for` que recorri els índexs des de 0 fins a l'últim valor de l'array (mida de l'array -1) o es pot utilitzar un bucle `for...of` per recórrer directament els valors de l'array.

Utilitzar un tipus de bucle o un altre dependrà de si necessitem utilitzar l'índex o només el valor, o si l'ordre és important. Per exemple, amb un bucle `for` es poden recórrer els elements en ordre invers.

Per iterar sobre els valors d'un map o un set cal utilitzar `for...of`, aquest tipus de bucle funciona només amb col·leccions iterables (com Array, Map i Set) i per aquest motiu no es pot utilitzar amb objectes, ja que els objectes no són iterables.

Com es pot apreciar, quan s'iteren els elements d'un mapa el valor de l'element és el parell clau i valor, per consegüent es poden desestructurar i assignar a variables independents (en aquest exemple anomenades `clau` i `valor`).

Finalment, per iterar sobre les propietats d'un objecte cal utilitzar `for...in`, el nom de la propietat s'assigna a la variable i per accedir al valor hem de fer la consulta utilitzant claudàtors, com si és tractés d'un array.

2.1.1 Arrays

Als arrays, tot i ser un tipus de dades que s'utilitza amb molta freqüència en tots els llenguatges de programació, existeixen diferencies en la seva utilització. Aquestes diferències es mostren amb claredat quan es compara JavaScript amb Java.

A Java, els arrays s'han d'inicialitzar amb la seva mida, en canvi, a JavaScript la mida d'un array pot variar en qualsevol moment. Una altre diferència és que els arrays de JavaScript poden emmagatzemar elements de qualsevol tipus (Integet, Boolean, String, etc.) i poden estar emmagatzemats en posicions no consecutives; de fet, poden tenir com a índex d'accés un valor no numèric.

Podeu trobar una llista exhaustiva amb tots els mètodes dels arrays en el següent enllaç: mzl.la/33Dyr8k.

Un programador en JavaScript ha de conèixer totes les singularitats que aporta aquest llenguatge quan fa ús d'aquest tipus de dades. Conèixer aquestes singularitats és bàsic per desenvolupar programes simples i eficients, per això, un bon programador ha de conèixer les següents propietats i mètodes dels arrays (i un que retornen un array: `split`):

- `length`: nombre d'elements que conté l'array.

- `concat(nouarray1, nouarray2, ...)`: crea un nou array amb els elements de dos o més arrays diferents.
- `push(element1, element2, ...)`: agrega un o més elements al final de l'array.
- `unshift(element1, element2, ...`: afegeix un o més elements a l'inici de l'array.
- `pop()`: elimina l'últim element de l'array i el retorna.
- `shift()`: elimina el primer element de l'array i el retorna.
- `sort([funció])`: ordena alfabèticament els elements d'un array. Es pot passar una funció com argument per realitzar ordenacions avançades.
- `split(separador[, limit])`: converteix una cadena de text en un array de cadenes de text.
- `join([separador])`: uneix tots els elements d'un array en una cadena de text utilitzant un caràcter d'unió.
- `reverse()`: modifica un array col·locant els seus elements en l'ordre invers a la seva posició original.
- `indexOf(elementAcercar[, posicioInical])`: retorna la posició de l'array on es troba la primera ocurrència del valor cercat.
- `lastIndexOf(elementAcercar[, posici´ó inicial]`: retorna la posició de l'array on es troba l'última ocurrència del valor cercat.
- `every(funció)`: executa la funció passada com argument i comprova que el resultat de la funció aplicada a tots els valors de l'array sigui cert.
- `some(funció)`: executa la funció passada com argument i comprova que almenys un dels resultats de la funció aplicada als elements de l'array sigui cert.
- `filter(funció)`: executa la funció per a tots els elements de l'array i retorna un nou array amb tots els elements als quals la funció hagi retornat cert.
- `slice(inici, final)`: retorna un nou array amb els valors que es troben des de l'índex inici fins a l'índex final.
- `splice(inici, comptadorEliminar[, element1, element2, ...]`: afegeix i elimina elements a partir d'una posició de l'array.
- `fill(valor [, posInicial, posFinal])`: omple un array amb el valor indicat; opcionalment es pot indicar una posició d'inici i una de final.
- `includes(elementCercat[, desdePosicio])`: retorna cert si l'array inclou el valor passat com argument.
- `forEach(funció)`: itera sobre tots els elements de l'array i executa la funció passada com argument per a tots els elements de l'array.

- `map(funció)`: executa la funció sobre tots els elements de l'array i retorna un nou array amb els resultats.

A banda d'aquests mètodes, l'objecte Array inclou també les següents funcions estàtiques:

- `Array.from(arrayOIterable[, funció])` : genera un nou array a partir d'un altre array o objecte iterable (per exemple, un *set*). Opcionalment es pot afegir una funció per processar cada element.
- `Array.isArray()`: retorna cert si l'element passat com a argument és un array.

2.1.2 Maps

Els mapes ens permeten crear col·leccions desordenades d'elements de manera que podem accedir als valors directament utilitzant una clau.

Al contrari del que passa amb els arrays o els objectes no es pot accedir directament als elements utilitzant claudàtors, ja que en utilitzar la sintaxi de claudàtors el que es modifica o consulta són les propietats de l'objecte i no pas el contingut del mapa. Cal utilitzar els mètodes `set` i `get` per assignar i recuperar els valors respectivament.

```
let exempleMap = new Map();
exempleMap.set("verd", "Passar");
exempleMap.set("ambar", "Compte!");
exempleMap.set("vermell", "Prohibit");
exempleMap["groc"] = "desconegut";

// verd no és una propietat de l'objecte, és un valor del mapa
console.log(exempleMap["verd"]);
// groc és una propietat de l'objecte
console.log(exempleMap["groc"]);

// verd és un element del mapa
console.log(exempleMap.get("verd"));
// groc no és un element del mapa
console.log(exempleMap.get("groc"));
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/YzqPjgq?editors=0012.

Els mapes es poden manipular utilitzant les següents propietats i mètodes:

- `size`: mida del mapa.
- `clear()`: buida el contingut del mapa.
- `delete(clau)`: elimina l'element del mapa.
- `forEach(funció)`: recorre el mapa i executa la funció per a cada element.

- `get(clau)`: retorna el valor associat a la clau passada com argument.
- `has(clau)`: retorna cert si el mapa conté la clau passada com argument.
- `set(clau, valor)`: assigna un valor a una clau.

Una diferència important amb la utilització d'objectes com a diccionaris és que les claus d'un mapa poden ser qualsevol mena d'objecte incloent funcions i tipus primitius, mentre que el nom de les propietats d'un objecte només poden ser cadenes de text (encara que aquests noms no es trobin entre cometes es continuen considerant cadenes de text.)

2.1.3 Sets

Quan no és necessari que la col·lecció sigui ordenada i aquesta no ha d'admetre elements duplicats, cal considerar utilitzar un *set* en lloc d'un array.

Hi ha dos avantatges importants quan s'utilitza un set en lloc d'un array:

- Eliminar arrays per valor és molt lent: array.splice(array.indexOf(valor)), en canvi eliminar-los d'un set és immediat: set.delete(valor).
- No s'ha de controlar que hi hagin duplicats, ja que els valors d'un set sempre són únics.

Els sets es poden manipular utilitzant les següents propietats i mètodes:

- `size`: mida del set.
- `clear()`: buida el contingut del set.
- `delete(valor)`: elimina l'element del set.
- `forEach(funció)`: recorre el set i executa la funció per a cada element.
- `has(valor)`: retorna cert si el mapa conté la clau passada com argument.

Fixeu-vos que al contrari del que passa amb els mapes, no existeix un mètode `get`, ja que un set només conté una llista de valors i només podem comprovar si el set conté un valor o no mitjançant el mètode `has`.

2.2 Gestió de l'estoc d'una botiga

Una bona manera de veure com s'utilitzen les funcionalitats de les col·leccions és a través de l'ús que se'n fa en un exemple concret, com pot ser la gestió de l'estoc d'una botiga.

Es vol tenir enregistrat l'estoc dels productes d'una botiga. De moment, és una botiga petita i només tenen deu productes, però es venen molt. Per això hem de controlar l'estoc de cadascun d'ells per tal que sempre els tinguem disponibles.

El botiguer ens ha demanat un llistat amb tot l'estoc dels seus productes.

I aquesta ha estat la nostra implementació:

Codi HTML i CSS dels exemples

El codi HTML i CSS utilitzat per aquests exemples es pot trobar als enllaços a CodePen.

```
let productes = new Array(10);
productes.fill(50)

function veureEstoc(){
    let llistat = "";
    for(let i = 0; i < productes.length; i++){
        llistat += '<li>El producte número ${i} té ${productes[i]} unitats';
    }
    document.getElementById("llistat").innerHTML=llistat;
}

veureEstoc();
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/XWdJKqG.

Si observeu el codi anterior s'ha utilitzat un array per guardar l'estoc de cada producte. Cada posició de l'array `productes` correspon a un dels productes de la botiga. L'array conté el número d'unitats que hi ha de cadascun d'ells. Inicialment, l'array s'omple

amb el número 50.

Per crear un array s'ha utilitzat la instrucció:

```
let productes = new Array(10);
```

Amb aquesta instrucció es crea un array inicialitzat amb 10 posicions, però també ho podíem haver assignat literalment:

```
let productes = [];
```

Quina diferència existeix entre les dues maneres de crear un array?

A priori, sembla que són iguals però no, mireu el següent exemple:

```
let a = [],           // aquests són iguals
    b = new Array(),  // a i b son arrays amb longitud 0

    c = ['foo', 'bar'],          // aquests també són iguals
    d = new Array('foo', 'bar'), // c i d són arrays amb dos strings

    // Aquí hi ha la diferència:
    e = new Array(3),   // e.length == 3, e[0] == undefined
    f = [3]             // f.length == 1, f[0] == 3
;
```

> La diferència entre crear un array amb el seu constructor o crear-lo de manera literal és que si s'utilitza el constructor es pot definir una mida inicial a l'array.

Podem definir, amb el constructor, un array d'una mida concreta. En canvi, si utilitzem l'assignació literal `[]` per crear arrays, no podem definir la mida inicial. Si afegim en un array literal un número, aquest serà el primer element de l'array i no la seva mida.

La sintaxi és la següent:

```
[element0, element1, ..., elementN]
 new Array(element0, element1[, ...[, elementN]])
 new Array(mida)
```

L'array s'inicialitza amb els elements donats excepte si només hi ha un element. En aquests cas, aquest element simbolitza la mida de l'array només si s'utilitza la creació de l'array amb la paraula reservada `new`.

Un cop creat l'array l'omplim amb el valor 50:

```
productes.fill(50)
```

Amb el mètode `fill` s'assigna el valor 50 a tots els elements de l'array. Cal destacar que si l'array és buit, invocar aquest mètode no tindrà cap efecte.

Si en lloc de fer servir un valor estàtic volguéssim generar els valors aleatòriament, ho podríem fer utilitzant la instrucció `map` per crear un nou array a partir d'una funció i assignant-lo a l'array `productes`:

```
productes = productes.map(() => Math.floor(Math.random() * 100));
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/ZEWYOmV.

Fixeu-vos que al mètode `map` se li ha passat com a argument una funció de fletxa que retorna un nombre aleatori entre 0 i 99:

```
() => Math.floor(Math.random() * 100)
```

És important recordar que la funció `map` només s'aplica als valors assignats, ja sigui mitjançant el mètode `fill` o assignant els valors manualment.

La funció `veureEstoc()`, recorre l'array `productes` i mostra per pantalla la informació de cadascun d'ells dins de l'etiqueta amb identificador `llistat` (al codi HTML).

Només queda veure el funcionament de la funció `veureEstoc`:

```
function veureEstoc(){
    let llistat = "";
    for(let i = 0; i < productes.length; i++){
        llistat += '<li>El producte número ${i} té ${productes[i]} unitats';
    }
    document.getElementById("llistat").innerHTML=llistat;
}
```

Aquesta funció la invoquem directament des de JavaScript i és l'encarregada de generar el llistat que es visualitza a la pantalla.

Tot i que sabem que només hi ha 10 productes, no s'ha utilitzat aquest número com a condició de control. És molt més útil accedir a la propietat `length` de l'array.

> La propietat `length` d'un array ens dóna el nombre d'elements que conté.

Veiem un altre exemple de creació d'un array i accés a la seva mida:

```
let vocals = ["a", "i", "i", "o", "o"];
console.log(‘Nombre de vocals: ${vocals.length}‘); // Nombre de vocals: 5
```

Tornem a l'exemple. La funció `veureEstoc()` utilitza una variable local anomenada `llistat` que es va omplint poc a poc a mesura que el bucle va donant voltes. A cada volta del bucle la variable recull les dades del producte en curs. Finalment, quan s'arriba al final de l'array, la variable conté la informació desitjada per l'usuari.

```
llistat += ‘<li>El producte número ${i} té ${productes[i]} unitats‘;
```

Només falta mostrar la informació a l'usuari. Utilitzem l'objecte `document` de JavaScript per accedir al codi HTML de la pàgina web.

Cerquem l'etiqueta identificada amb `id=llistat` i l'assignem el nou codi HTML que volem que tingui. Així, podem canviar el codi HTML de la pàgina i donar la informació que desitgem.

```
document.getElementById("llistat").innerHTML=llistat;
```

2.3 Ampliació del número de productes de la botiga

Volem ampliar el número de productes de la botiga. Cada vegada que pitgem un botó, s'ha d'ampliar el número de productes. Tant el número de productes com l'estoc han de ser aleatoris.

```
function ampliarEstoc(){
  let nouEstoc = demanarProductes();
  productes.push(...nouEstoc);
  veureEstoc();
}

function demanarProductes(num){
  if(arguments.length === 0){
    num = Math.floor((Math.random() * 5) + 1);
    }

  let nousProductes = new Array(num);
  nousProductes.fill();

  return nousProductes.map(() => Math.floor((Math.random() * 100)))
}
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/abNzmvO?editors=1010.

En aquest exemple s'han afegit dos botons al codi HTML que criden a les funcions `veureEstoc` i `ampliarEstoc`. Si observeu el codi HTML dels botons, veureu que l'event `onclick` dels botons ten assignada les crides a les seves respectives funcions.

```
<button onclick="veureEstoc()">Veure estoc</button>
<button onclick="ampliarEstoc()">Ampliar estoc</button>
```

Observeu que s'han creat dues noves funcions: `ampliarEstoc()` i `demanarProductes(num)`. Aquesta última funció s'ha creat per fer refacció del codi.

> **Fer refacció** (*Code Refactoring*) vol dir, en programació, reestructurar el codi per tal de millorar-lo i reduir la seva complexitat sense canviar les funcionalitats que hi havien fins al moment.

No s'han canviat les funcionalitats, sinó que s'han aprofitat millor. De fet, la funció `demanarProductes(num)` s'utilitza per dues coses: inicialitzar l'array productes i, a més a més, per demanar productes nous. Fem un cop d'ull a la funció.

```
function demanarProductes(num){
  if(arguments.length === 0){
    num = Math.floor((Math.random() * 5) + 1);
  }

  let nousProductes = new Array(num);
  nousProductes.fill();

  return nousProductes.map(() => Math.floor((Math.random() * 100)))
}
```

La funció admet un paràmetre anomenat `num`. Aquest paràmetre és opcional. Si s'envia el paràmetre crearà un array amb el nombre de productes demanats. En canvi, si no s'envia el número de productes que es crearan serà aleatori.

Aquesta funcionalitat s'aconsegueix preguntant el número de paràmetres enviats a la funció. En concret, amb `argument.length` podem saber exactament el número de paràmetres enviats. Si la mida és `1` significa que la variable `num` està plena i ens demanen un `array` amb aquesta mida. En canvi, si la mida és `0` vol dir que ens demanen un array amb una mida indeterminada. Aquesta mida l'haurem de calcular aleatòriament.

La resta de la funció ja és coneguda. Només s'ha canviat el punt del codi on es realitza la inicialització i la mida inicial, en lloc de crear l'array directament es fa la crida a `demanarProductes` i s'assigna com a array inicial el retorn d'aquesta funció.

Llavors, per inicialitzar l'*array productes* es fa una crida a aquesta funció.

```
productes = demanarProductes(10);
```

Com que inicialment es volen 10 productes, s'envia la mida de l'array desitjat a la funció. La funció crearà un array temporal que s'assignarà a la variable `productes`.

La funció `demanarProductes(num)` ha estat creada per utilitzar-la també en el cas que es vulgui ampliar l'oferta de productes de la botiga.

```
function ampliarEstoc(){
  let nouEstoc = demanarProductes();
  productes.push(...nouEstoc);
  veureEstoc();
}}
```

Per ampliar aleatòriament tant el nombre de productes com l'estoc utilitzarem la funció `demanarProductes()` sense enviar cap paràmetre. Si executem la funció sense paràmetres, ens retornarà un array amb les característiques desitjades.

Una vegada tenim l'array amb els nous productes s'han d'afegir a l'array `productes`.

Per afegir els elements retornats hem fet servir l'operador de propagació `...nouEstoc` i els hem afegit mitjançant el mètode `push`.

Cal recordar que aquest mètode afegeix un o més elements al final d'un array i que aquests elements han d'estar separats per comes. És per aquest motiu que es fa servir l'operador de propagació, ja que fa la conversió d'un array a una llista de valors separats per coma.

> El mètode **push** afegeix un o més elements al final de l'array. L'array original es modifica i augmenta la seva longitud. També és possible afegir els elements a l'inici de l'array amb el mètode **unshift**.

La sintaxi del mètode `push` és la següent:

```
array.push(element1, ..., elementN)
```

Tots els elements que es passen com a paràmetre del mètode `push` s'afegeixen al final de l'array.

En cas d'haver invocat el mètode passant com argument `nouEstoc`, el resultat hauria estat diferent, ja que, en lloc d'afegir tots els valors un a un, s'hauria afegit tot l'array com un únic element.

Fixeu-vos en els dos casos representats en els següents exemples: Primer exemple:

```
let a = [1, 2, 3];
let b = [4, 5, 6];

a.push(b);
console.log(a)
```

Amb aquest cas el contingut de l'array seria el següent:

```
1
2
3
[4, 5, 6]
```

Com es pot apreciar el quart element de l'array és un array amb 3 valors. Segon exemple:

```
let a = [1, 2, 3];
let b = [4, 5, 6];

a.push(...b);
console.log(a)
```

La sortida en aquest cas seria la següent:

```
1
2
3
4
5
6
```

Com es pot apreciar, en aquest cas el resultat és el desitjat, s'han afegit els continguts de l'array `b` a l'array `a` que ara té una mida de 6.

Alternativament es podria haver fet servir la funció `concat` per crear un array nou afegint el nou estoc al final de l'array `productes`. Aquest nou array creat s'ha assignat a la variable `productes`, ja que la funció `concat` no altera l'array sobre el qual es crida:

```
productes = productes.concat(nouEstoc);
```

Podeu provar l'exemple amb `concat` a: https://codepen.io/ioc-daw-m06/pen/dyMPpNv?editors=10.

> La funció **concat** es fa servir per crear un nou array amb els elements de dos arrays diferents.

Cal tenir en compte que en aquest cas la utilització del mètode `concat` és menys eficient que la utilització de `push`, ja que el primer crea un array addicional mentre que el segon només afegeix els valors d'un array a un altre.

La sintaxi de la funció és la següent:

```
let arrayNou = arrayAntic.concat(valor1[, valor2[, ...[, valorN]]])
```

La funció `concat` retorna una còpia dels valors de l'array `arrayAntic`, afegint-hi els valors passats com argument, que poden ser valors individuals o altres arrays.

Un altre exemple de l'ús de la funció `concat` és:

```
let array1 = [1, 2, 3];
array2 = array1.concat(4, 5, 6); // array2 = [1, 2, 3, 4, 5, 6]
array3 = array1.concat([4, 5, 6]); // array3 = [1, 2, 3, 4, 5, 6]
```

Veiem, que amb la funció `concat` podem concatenar arrays o valors individuals. Utilitzarem la que millor ens convingui segons el tipus de dades que tinguem.

2.4 Rànquing amb els productes més venuts

En aquest apartat volem afegir unes quantes funcionalitats addicionals per fer la botiga més interactiva. La idea és que es puguin comprar i vendre productes i poder tenir un rànquing ordenat dels productes més venuts.

En concret, les funcionalitats que volem tenir són:

- **Funcionalitats del botiguer**
  - Afegir 1 producte: afegir només un producte al catàleg de productes disponibles a la venda. Aquest producte tindrà un nombre d'estoc aleatori.
  - Eliminar 1 producte: eliminar l'últim producte afegit al catàleg de productes. Elimina tot l'estoc disponible del producte.
  - Ordenar els productes segons l'estoc que queda: ordena els productes segons el seu estoc. El número de producte associat a l'estoc pot canviar.
- **Funcionalitats del comprador**
  - Comprar un producte: el comprador pot comprar un producte dels disponibles en el catàleg de productes. Cada vegada que es compra un producte es reordena i es visualitza el rànquing de productes més venuts.

```
let productes = [];
let vendes = new Map();

function veureEstoc() {
  let llistat = "";
  for (let i in productes) {
    llistat += `<li>El producte ${i} té ${productes[i]} unitats`;
  }
  document.getElementById("llistat").innerHTML = llistat;
}

function ampliarEstoc() {
  let nouEstoc = demanarProductes();
  productes.push(...nouEstoc);
  veureEstoc();
}

function demanarProductes(num) {
  if (arguments.length === 0) {
    num = Math.floor(Math.random() * 5 + 1);
  }

  let nousProductes = new Array(num);
  nousProductes.fill();

```

```
  return nousProductes.map(() => Math.floor(Math.random() * 100));
}

function comprar(numproducte) {
  if (productes[numproducte] !== undefined && productes[numproducte] > 0) {
    let quantitat = 0
    if (vendes.has(numproducte)) {
      quantitat = vendes.get(numproducte);
    }
    quantitat++;
    vendes.set(numproducte, quantitat);
    productes[numproducte]--;

    ranquing();
    veureEstoc();
  }
}

function nouProducte() {
  productes.push(Math.floor(Math.random() * 100));
  veureEstoc();
}

function eliminarProducte() {
  productes.pop();
  veureEstoc();
}

function ordenarProductes() {
  productes.sort((a, b) => a - b);
  veureEstoc();
}

// funció per ordenar el producte segons les unitats venudes
function sortProducte(a, b) {
  const aa = a.split("-")[0];
  const bb = b.split("-")[0];
  return aa - bb;
}

function ranquing() {
  let aux = [];
  // array auxiliar per no perdre el numproducte després de la ordenació
  // s'afegeix el numproducte juntament amb les unitats venudes del producte
  // com a valor del nou array
  // numproducte: Numero del producte venut
  // vendes.get(numproducte): unitats venudes del producte numproducte
  for (let [numproducte, quantitat] of vendes) {
    aux.push(quantitat + "-" + numproducte);
  }
  // ordenar els productes segons les unitats venudes
  // s'ha creat una funció d'ordenació per separar les unitats venudes
  // del numproducte.
  aux.sort(sortProducte).reverse();
  // llistar els productes ordenats
  let llistat = "";
  for (let valor of aux) {
    // Tallem la cadena pel simbol d'unió '-'.
    // a la posició [0] el numero d'unitats venudes del producte
    // a la posició [1] tenim el numero de producte
    let [stockProducte, codiProducte] = valor.split("-");

    llistat += '<li>´sha venut ${stockProducte} unitats del producte ${
        codiProducte}</li>';
  }

  document.getElementById("venuts").innerHTML = llistat;
}

productes = demanarProductes(10);
```

```
95 veureEstoc();
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/yLOyaXd.

Totes aquestes funcionalitats s'han aconseguit utilitzant funcions del llenguatge JavaScript que treballen amb arrays i modifiquen la informació que contenen.

2.4.1 Afegir un producte

Per afegir un producte al catàleg de productes s'ha utilitzat el codi següent:

```
1 function nouProducte() {
2     productes.push(Math.floor((Math.random() * 100)));
3     veureEstoc();
4 }
```

I el codi HTML associat a aquesta funció és el següent:

```
1 <button onclick="nouProducte()" >Afegir 1 producte</button>
```

Com podeu veure, en afegir un nou element amb aquest mètode només s'ha de calcular l'estoc d'aquest producte ja que aquest s'afegeix al final de l'array i, per tant, el número de producte es calcularà automàticament. Es fa la crida a la funció `veureEstoc` per fer més interactiu el programa, així es veu al moment que s'ha actualitzat els productes del catàleg.

La sintaxi de la funció `unshift` és la següent:

```
1 array.unshift([element1[, ...[, elementN]]])
```

Tots els elements que es passen com a paràmetre de la funció `unshift` s'afegeixen al principi de l'array.

Veiem un exemple d'utilització de `unshift`:

```
1 let productes = ["Llegums", "Tomàquet", "Ceba", "patata"];
2 let nousProductes = ["all", "pebrot"];
3 let numProductes = productes.unshift("Pastanaga", ...nousProductes);
4 console.log(productes);
5 console.log(numProductes);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/yLOyrpW?editors=1111.

Com es pot apreciar, es pot barrejar la inserció d'elements individuals (`Pastanaga`) i els elements propagats (`...nousProductes`).

Els mètodes `push` o `unshift` retornen la nova longitud de l'array i, si ens interessa, la podem emmagatzemar en una variable. En aquest cas, la variable utilitzada es diu `numproductes`. </newcontent>

2.4.2 Eliminar un producte

Per eliminar un producte del catàleg de productes s'ha utilitzat el següent codi:

```
function eliminarProducte() {
  productes.pop();
  veureEstoc();
}
```

I el codi HTML associat a aquesta funció és el següent:

```
<button onclick="delProducte()" >Eliminar 1 producte</button>
```

La sintaxi del mètode `pop` és la següent:

```
let element = array.pop()
```

El mètode `pop` esborra l'últim element de l'array però el retorna per poder utilitzar-lo.

> Per eliminar un nou producte utilitzem el mètode **pop**. Aquest mètode elimina l'últim element de l'array. L'array original es modifica i decreix la seva longitud en 1 element. També és possible eliminar el primer element de l'array si utilitzem el mètode **shift** , en comptes d'utilitzar el mètode `pop`.

Com es pot apreciar, només cal invocar el mètode `pop` de l'array per eliminar l'últim element i seguidament invoquem la funció `veureEstoc` per refrescar les dades al document HTML.

<newcontent> Veiem ara el mateix exemple però utilitzant `shift`:

```
function delProducte() {
  productes.shift();
  veureEstoc();
}
```

La sintaxi del mètode `shift` és la següent:

```
let element = array.shift()
```

El mètode `shift` elimina el primer element de l'array i mou tots els altres per tal que l'índex de l'array continuï començant per 0. L'element eliminat de l'array es retorna per poder utilitza-lo en el programa si fos necessari.

Tant si s'utilitza el mètode `pop` com si s'utilitza el mètode `shift` es pot recuperar l'element extret i guardar-lo en una variable. Per exemple:

```
function delProducte() {
  let produteEliminat = productes.pop();
  veureEstoc();
}
```

Tot i que en aquest cas no fem res amb el valor retornat, us trobareu casos en els quals serà interessant portar a terme alguna acció sobre l'element eliminat o caldrà retornar-lo. Per exemple, per notificar a l'usuari quin element ha estat eliminat.

2.4.3 Ordenar productes segons l'estoc

Aquesta funcionalitat permet ordenar els productes segons l'estoc que tenen. Els productes amb poc estoc seran els primers. El codi associat a aquesta funcionalitat és el següent:

```
function ordenarProductes() {
  productes.sort((a, b) => a - b);
  veureEstoc();
}
```

El codi HTML associat a aquesta funció és el següent:

```
<button onclick="ordenarProductes()" >Ordenar Productes segons l'estoc que
    queda</button>
```

En utilitzar la funció `ordenarProductes` ha d'aparèixer un llistat semblant a aquest:

```
Llistat de productes:

El producte 0 té 6 unitats
El producte 1 té 11 unitats
El producte 2 té 35 unitats
El producte 3 té 35 unitats
El producte 4 té 51 unitats
El producte 5 té 58 unitats
El producte 6 té 59 unitats
El producte 7 té 70 unitats
El producte 8 té 73 unitats
El producte 9 té 98 unitats
```

Si ens hi fixem, per ordenar només necessitem el mètode `sort` que ens proporciona la biblioteca JavaScript.

> El mètode `sort`,per defecte, ordena alfabèticament, és a dir, com si tot fos una *string*.

La sintaxi del mètode `sort` és la següent:

```
array.sort([funcioComparacio])
```

El mètode `sort` permet ordenar els elements de l'array. El comportament per defecte del mètode és ordenar els elements de l'array utilitzant el codi *Unicode* dels caràcters de l'array a ordenar. Si es vol canviar aquest comportament existeix la possibilitat d'enviar una funció d'ordenació com a paràmetre.

Per exemple, per ordenar números no ens interessa aquest comportament: si els números s'ordenen com si fossin *strings*, "25" seria major que "100", ja que "2" és més gran que "1".

Per solucionar aquest inconvenient, el mètode `sort` permet canviar el seu comportament si li proporcionem la funció que ha d'utilitzar per fer l'ordenació.

Com es pot apreciar, en aquest cas s'ha passat com argument del mètode `sort` una funció de fletxa (aquest és un cas d'ús molt habitual per les funcions de fletxa):

```
(a, b) => a - b
```

Quan la funció `sort` ha de comparar dos valors per determinar quin és més gran, envia aquests dos valors a la funció de comparació i aquesta ha de retornar un valor negatiu, un zero o un valor positiu, depenent dels paràmetres.

Per exemple, si es compara el número 40 i el 100, la funció `sort` crida a la funció de comparació amb els paràmetres (40, 100). La funció de comparació calcula `40 - 100` i retorna `-60` (un número negatiu).

El mètode `sort` determina que 40 és més petit que 100 i ordenarà els valors segons aquesta informació.

2.4.4 Comprar un producte

La funcionalitat *comprar producte* és indispensable si volem fer un rànquing dels productes més venuts. La funcionalitat que es vol implementar és que el comprador informi sobre el producte desitjat i que decreixi el número d'estoc del producte. En aquest cas, el comprador ha de dir el número de producte que vol comprar.

El codi JavaScript associat a aquesta funcionalitat el podeu veure en el codi següent:

```
function comprar(numproducte) {
  if (productes[numproducte] !== undefined && productes[numproducte] > 0) {
    let quantitat = 0
    if (vendes.has(numproducte)) {
      quantitat = vendes.get(numproducte);
    }
    quantitat++;
    vendes.set(numproducte, quantitat);
    productes[numproducte]--;

    ranquing();
    veureEstoc();
  }
}
```

A continuació es mostra la crida HTML des del botó comprar producte. Des de la pàgina HTML es fa el pas de paràmetre amb el valor que l'usuari introdueixi en el *textbox*. Aquest valor ha de coincidir amb un número de producte vàlid.

```
<input type="text" id="numProducte" size="10"/>
<button onclick="comprar(document.getElementById('numProducte').value);" >
    Comprar Producte</button>
```

L'*event* `onclick` del botó fa la crida a la funció JavaScript `comprar` però aquesta funció necessita del número de producte que es vol comprar. Així, des de la pàgina web s'accedeix a l'element `numProducte`, que és el *textbox* que utilitza l'usuari per informar del producte que vol comprar, i s'agafa el valor. Aquest valor s'envia per paràmetre a la funció `comprar`.

Una vegada es realitza la crida a la funció `comprar` aquesta ha de realitzar les accions següents:

- Comprovar que l'estoc del producte s'hagi definit i aquest sigui superior a 0: `if (productes[numproducte] !== undefined && productes[numproducte] > 0)`.
- Comprovar si el producte ja es troba al mapa `vendes`; si és així, recuperem el valor associat a la clau `numproducte` i, en cas contrari, assignem 0 a la quantitat: `if (vendes.has(numproducte))`.
- Incrementar en un la quantitat venuda.
- Assignar el nou valor a `vendes`: `vendes.set(numproducte, quantitat);`
- Reduïm el nombre de productes en estoc en un: `productes[numproducte]--;`.
- En qualsevol cas, es mostra el rànquing i s'actualitza el llistat de productes per pantalla.

Fixeu-vos que `vendes` és un *map* i no pas un array, així que es tracta d'una col·leccio desordenada i s'afegeixen els elements indicant la clau i el valor amb el mètode `set`, es recuperan amb `get` i es comprova si la clau existeix amb `has`.

En aquest cas hem optat per fer servir un diccionari de dades perquè l'ordre en què s'afegeixen els elements no és important i, en cas que s'eliminés algun valor del mapa, no volem que les claus canviïn (cosa que passa quan s'elimina un element d'un array, ja que els índexs de tots els elements posteriors s'actualitzen per continuar sent consecutius).

La clau de `vendes` ha de coincidir amb l'índex de `productes` i, per consegüent, és més segur fer servir un mapa.

2.4.5 Mostrar el rànquing dels productes més venuts

La funció que calcula el rànquing dels productes més venuts es basa en un array on hi ha emmagatzemats els productes que ha comprat l'usuari. Aquest array es va omplint a mesura que l'usuari utilitza la funció *compra(numProducte)*.

El codi per obtenir un rànquing ordenat de major a menor és el següent:

```
//funció per ordenar el producte segons les unitats venudes
function sortProducte(a, b) {
  const aa = a.split("–")[0];
  const bb = b.split("–")[0];
  return aa – bb;
}

function ranquing() {
  let aux = [];
  //array auxiliar per no perdre el numproducte després de la ordenació
  //s'afegeix el numproducte juntament amb les unitats venudes del producte
  //com a valor del nou array
  //numproducte: Numero del producte venut
  //vendes.get(numproducte): unitats venudes del producte numproducte
  for (let [numproducte, quantitat] of vendes) {
    aux.push(quantitat + "–" + numproducte);
  }
  //ordenar els productes segons les unitats venudes
  //s'ha creat una funció d'ordenació per separar les unitats venudes
  // del numproducte.
  aux.sort(sortProducte).reverse();
  //llistar els productes ordenats
  let llistat = "";
  for (let valor of aux) {
    //Tallem la cadena pel simbol d'unió '–'.
    //a la posició [0] el numero d'unitats venudes del producte
    //a la posició [1] tenim el numero de producte
    let [stockProducte, codiProducte] = valor.split("–");

    llistat += '<li>´sha venut ${stockProducte} unitats del producte ${
        codiProducte}</li>';
  }

  document.getElementById("venuts").innerHTML = llistat;
}
```

El mapa `vendes` utilitza la seva clau com a número de producte.

És important remarcar aquest fet ja que a l'hora de fer l'ordenació, la funció *.sort()* posa a la primera posició de l'array el major número d'unitats venudes i perdrem a quin producte correspon.

Per generar el rànquing de vendes ens trobem amb dos problemes:

- `vendes` és un *map* i els mapes no es poden ordenar.
- si el convertim en un array i l'ordenem mitjançant el mètode `sort`, es perd la correspondència "índex-clau" amb l'array de productes, ja que fem servir l'índex de `productes` com identificador per a cada producte.

Ens hem d'inventar una manera de no perdre la referència al producte. Per sort, sabem que la funció d'ordenació la podem canviar. La idea és la següent: per no perdre la referència al producte, després de fer la ordenació, aquest ha d'estar dins del valor juntament amb les unitats venudes. Així, podem crear una cadena de text on, una part de la cadena siguin les unitats venudes i una altra part correspongui al producte associat. Per exemple, si tenim 50 unitats venudes del producte 3, podem construir una cadena de text semblant a "50-3". I així, amb tots els productes venuts.

Veieu el codi utilitzat per crear un nou array on guardem les cadenes de text amb el número d'unitats venudes i el seu producte.

```
for (let [numproducte, quantitat] of vendes) {
  aux.push(quantitat + "-" + numproducte);
}
```

Fixeu-vos que s'ha utilitzat `for...of` per recorrer el mapa de `vendes`, desestructurant l'element, de manera que la clau queda assignada a `numproducte` i el valor a `quantitat`.

Una vegada obtingudes les varibles, les concatenem per generar la cadena de text i les afegim a l'array auxiliar i, seguidament, l'ordenem mitjançant el mètode `sort`, passant com a argument la funció `sortProducte`. Aquesta es la funció que es passa com argument:

```
function sortProducte(a, b) {
  const aa = a.split("-")[0];
  const bb = b.split("-")[0];
  console.log(aa, bb);
  return aa - bb;
}
```

En aquest cas, en lloc de passar la funció `sortProducte` podríem haver utilitzat una funció de fletxa, però, com que aquesta funció d'ordenació és una mica llarga, s'ha optat per implementar-la com una funció estàndard; d'aquesta manera el codi resulta menys enrevessat.

Cal recordar que, per defecte, el mètode `sort` fa l'ordenació alfabèticament, però això no ens interessa perquè s'interpretaria que 5 és major que 100, ja que en ordenar cadenes de text es fa la comparació del codi cada caràcter i no s'interpreta com a número.

Tot i que en aquest exemple hem fet servir `const` per indicar que els valors de `aa` i `bb` no canviaran, això no és necessari. Es podria haver fet servir `let`.

Totes les funcions d'ordenació han de retornar un nombre positiu si el segon valor és més petit, un nombre negatiu si el primer valor és més petit i 0 si són iguals. Per fer el rànquing dels productes més venuts s'ha de comparar les unitats venudes de dos productes. En aquest cas, `a` té la forma *unitatsVenudesDe-producteA* i `b` té la forma *unitatsVenudesDe-producteB*. Hem de separar les unitats venudes del número de producte.

Per separar una cadena de text podem utilitzar el mètode `split`. A aquest mètode li podem passar un caràcter o una *string* i trencarà la cadena de text cada vegada que trobi aquest caràcter o *string*.

La sintaxi de la funció `split()` és la següent:

```
let nou_array = cadena.split([separador[, limit]])
```

El primer paràmetre és opcional i especifica el caràcter a utilitzar per separar la cadena de text. Cada cadena de text separada de l'original s'emmagatzemarà en

una posició de l'array retornat. El segon paràmetre indica el nombre de vegades que es separarà com a màxim la cadena de text utilitzant el separador.

En definitiva, si invoquem el mètode `split`, als paràmetres de la funció tenim que, a la primera posició, *[0]*, estem accedint al nombre d'unitats venudes i, a la segona posició, *[1]*, al producte associat.

<newcontent>Només s'ha d'agafar la primera posició, després d'invocar el mètode `split`, per obtenir les unitats venudes:

```
let aa = a.split("-")[0];
```

També existeix la funció inversa anomenada *join()*. La sintaxi de la funció és la següent:

```
let cadena = array.join([separador = ','])
```

El mètode `join` uneix tots els elements d'un array i els converteix en una cadena de text. Si s'especifica un separador, aquest s'utilitza per fer la unió entre els diferents elements de l'array.

Vegem un exemple:

```
let array = ["hola", "món"];
let missatge = array.join(""); // missatge = "holamón"
console.log(missatge);
missatge = array.join(" "); // missatge = "hola món"
console.log(missatge);
```

Una vegada tinguem l'array ordenat tindrem ordenats els valors (sencers, sense fer l´`split`) on a la posició `aux[0]` hi haurà el producte amb menys vendes. Per fer un rànquing, ens interessa que estigui a l'inrevés, és a dir, a la posició `aux[0]` hi hauria d'haver el producte amb més unitats venudes. Es podia haver tingut en compte a l'hora de realitzar la funció de comparació de la funció `.sort()`, però és interessant que conegueu la funció `reverse()`, que inverteix l'ordre dels elements de l'array.

```
aux.sort(sortProducte).reverse();
```

La sintaxi del mètode és la següent:

```
array.reverse()
```

El mètode `split` converteix una cadena de text en un array de cadenes de text. La funció divideix la cadena de text determinant els seus trossos a partir del caràcter separador indicat.
El mètode `join` uneix tots els elements d'un array en una cadena de text utilitzant el caràcter d'unió.
El mètode `reverse` modifica un array col·locant els seus elements en l'ordre invers a la seva posició original.

Exemple:

```
let array = [1, 2, 3];
array.reverse();
console.log(array); // ara array = [3, 2, 1]
```

En aquest punt ja tenim l'array amb el rànquing dels productes més venuts. Només cal llistar-lo per pantalla. Veieu el codi que permet fer el llistat:

```
//llistar els productes ordenats
  let llistat = "";
  for (let valor of aux) {
    //Tallem la cadena pel simbol d'unió '-'.
    //a la posició [0] el numero d'unitats venudes del producte
    //a la posició [1] tenim el numero de producte
    let [stockProducte, codiProducte] = valor.split("-");

    llistat += `<li>´sha venut ${stockProducte} unitats del producte ${
        codiProducte}</li>`;
  }

  document.getElementById("venuts").innerHTML = llistat;
```

S'utilitza la variable `llistat` per emmagatzemar tot l'HTML que es mostrarà a l'usuari. Una vegada contingui totes les dades amb els productes més venuts es mostrarà per pantalla utilitzant la línia de codi següent:

```
document.getElementById("venuts").innerHTML = llistat;
```

Com ja hem vist en altres ocasions, s'utilitza l'objecte `document` per accedir a l'element de la pàgina identificat amb la paraula `venuts` i introduït l'HTML que ha de mostrar. En aquest cas, l'HTML serà el rànquing dels productes més venuts que conté la variable `llistat`.

2.5 Funcionalitats addicionals

En aquest codi s'han ampliat les funcionalitats del programa. Concretament es vol:

- Saber quin és el primer producte que té *X* unitats d'estoc.
- Saber l'últim producte que té *X* unitats d'estoc.
- Comprovar si tots els productes tenen més de 10 unitats d'estocs.
- Comprovar si algun producte té 0 unitats d'estoc.
- Saber quins productes tenen 0 unitats d'estoc.
- Un llistat de l'estoc des del producte X al producte Y.
- Afegir 1 producte a partir d'una posició determinada de l'array `productes`.

```
//Volem saber quin és el primer producte té //X// unitats d'estoc (indexOf)
function primerProducte(unitats) {
  let index = productes.indexOf(parseInt(unitats));

  if (index !== -1) {
    document.getElementById("info").innerHTML = index;
  } else {
    document.getElementById("info").innerHTML = 'No hi ha cap producte amb ${
        unitats} unitats.';
  }
}

// Volem saber l'últim producte que té //X// unitats d'estoc (lastIndexOf()
function ultimProducte(unitats) {
  let index = productes.lastIndexOf(parseInt(unitats));
  if (index !== -1) {
    document.getElementById("info").innerHTML = index;
  } else {
    document.getElementById("info").innerHTML = 'No hi ha cap producte amb ${
        unitats} unitats.';
  }
}

// Volem comprovar si tots els productes tenen més de 10 unitats d'estoc  (
    every)
function minDeuUnitats() {
  document.getElementById("info").innerHTML =
    'Tots els productes tenen més de 10 unitats? ${productes.every(unitats =>
        unitats >= 10)}';
}

// Volem comprovar si algun producte té 0 unitats d'estoc (some)

function senseEstoc() {
  document.getElementById("info").innerHTML =
    'Hi ha productes sense estoc? ${productes.some(unitats => unitats === 0)}';
}

// Volem saber quins productes tenen 0 unitats d'estoc (filter)
function checkBuitSplit(unitats) {
  let aa = unitats.split("-")[0];
  return parseInt(aa) === 0;
}
function llistatProductesSenseEstoc() {
  let llistat = "No hi ha productes sense estoc.";

  if (productes.some(unitats => unitats === 0)) {
    let aux = [];
    let arr = [];
    llistat = "";
    for (let numproducte in productes) {
      arr.push(productes[numproducte] + "-" + numproducte);
    }
    aux = arr.filter(checkBuitSplit);
    for (let index in aux) {
      llistat +=
        '<li>El producte ${aux[index].split("-")[1]} no té estoc.</li>';
    }
    document.getElementById("info").innerHTML = llistat;
  } else {
    document.getElementById("info").innerHTML = llistat;
  }
}


//  Volem un llistat de l'estoc des del producte X al producte Y (slice)
function llistatParcialdEstoc() {
  let ini = parseInt(document.getElementById("valor1").value);
  let fin = parseInt(document.getElementById("valor2").value);
```

```
66    let estoc = productes.slice(ini, fin);
67    document.getElementById("info").innerHTML = estoc.toString();
68    veureEstoc();
69  }
70
71  //  Volem afegir 1 producte a partir d'una posició X de l'array productes(
        splice).
72
73  function afegirEstocPosicioValor1() {
74    let ini = parseInt(document.getElementById("valor1").value);
75    productes.splice(ini, 0, Math.floor(Math.random() * 100));
76    veureEstoc();
77  }
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/RwaNjRN.

2.5.1 Primer i últim producte amb X unitats d'estoc

Es vol saber quin és el primer producte i quin és l'últim producte que té X unitats d'estoc, on X és un valor que ha introduït l'usuari.

El codi per saber quin és el primer producte amb X unitats d'estoc és el següent:

```
1  function primerProducte(unitats) {
2    let index = productes.indexOf(parseInt(unitats));
3
4    if (index !== -1) {
5      document.getElementById("info").innerHTML = index;
6    } else {
7      document.getElementById("info").innerHTML = 'No hi ha cap producte amb ${
         unitats} unitats.';
8    }
9  }
```

I l'HTML associat a aquesta funció és el següent:

```
1  <input type="text" id="unitats" size="10"/>
2  <button onclick="primerProducte(document.getElementById('unitats').value);" >
3     Primer producte amb X unitats d'estoc
4  </button>
```

El mètode que ens retorna el primer índex de l'array que coincideix el seu valor amb el passat per paràmetre es diu `indexOf`.

> La funció `indexOf` retorna la posició de l'array on es troba la primera ocurrència del valor cercat. Si no es troba el valor cercat, la funció retorna el valor -1.
> En canvi, la funció `lastIndexOf` retorna la posició de l'array on es troba l'última ocurrència del valor cercat.
> En tots dos casos, si no és troba el valor cercat retornen -1.

El mètode `indexOf` té la següent sintaxi:

```
let index = array.indexOf(elementCercat[, inici = 0])
```

Aquest mètode retorna el primer índex de l'array que coincideix el valor d'aquesta posició amb el paràmetre de la funció.

En canvi, el mètode `lastIndexOf` retorna l'últim índex i té la següent sintaxi:

```
let index = arr.lastIndexOf(elementCercat[, inici = arr.length - 1])
```

El codi per saber quin és l'últim producte amb X unitats d'estoc és el següent:

```
function ultimProducte(unitats) {
  let index = productes.lastIndexOf(parseInt(unitats));
  if (index !== -1) {
    document.getElementById("info").innerHTML = index;
  } else {
    document.getElementById("info").innerHTML = 'No hi ha cap producte amb ${
        unitats} unitats.';
  }
}
```

I el codi HTML associat és el següent:

```
<input type="text" id="unitats" size="10"/>
<button onclick="ultimProducte(document.getElementById('unitats').value);" >
    Últim producte amb X unitats d'estoc
</button>
```

2.5.2 Comprovar si tots els productes tenen 10 unitats d'estoc

Es vol comprovar si tots els productes tenen com a mínim 10 unitats d'estoc. Es pot fer recorrent l'array amb un bucle tipus `for`, però existeix una funcionalitat que ho fa per nosaltres: el mètode `every`.

Es pot crear una funció condicional o, com hem fet nosaltres, utilitzar una funció de fletxa per passar com argument. Aquesta funció és la que s'utilitzarà per comprovar si tots els elements acompleixen amb el requeriment (en el nostre cas, que hi hagi almenys 10 unitats).

El codi corresponent a aquesta funcionalitat és el següent:

```
// Volem comprovar si tots els productes tenen més de 10 unitats d'estoc  (
    every)
function minDeuUnitats() {
  document.getElementById("info").innerHTML =
    'Tots els productes tenen més de 10 unitats? ${productes.every(unitats =>
        unitats >= 10)}';
}
```

La funció de fletxa comprova si una unitat és més gran que 10 i es passa com a argument al mètode `every` per comprovar si l'estoc de cada producte és més gran que 10.

Fixeu-vos que, com només hi ha un argument, es pot prescindir dels parèntesis a la funció de fletxa:

```
unitats => unitats >= 10
```

És a dir, per cada element de l'array es cridarà a la funció de fletxa:

- S'assignarà a `unitats` el valor de l'element
- La funció de fletxa retornarà cert si `unitats >= 10` o fals en cas contrari.
- Si la funció de fletxa retorna fals en algun cas, el mètode `every` retornarà fals.

El mètode `every(funció)` comprova, element a element, si tots els valors de l'array compleixen la funció passada com a argument. Si tots els valors la compleixen retorna *true*, però si hi ha algun valor que no ho fa retorna *false*.

La sintaxi del mètode `every` és la següent:

```
resultat = arr.every(funcióComparacio[, parametres])
```

2.5.3 Comprovar si hi ha algun producte sense estoc

Es vol comprovar si hi ha algun producte sense estoc. Es pot fer amb un bucle `for` o bé utilitzar un mètode d'arrays que comprova si existeix algun valor que compleix un requisit. Aquesta funció s'anomena `some`. La funció rep per paràmetre una funció que realitza la comprovació. En aquest cas, es vol que la quantitat d'un producte sigui 0 per saber si hi ha o no estoc.<newcontent> Veieu el codi associat a aquesta funcionalitat:

```
function senseEstoc() {
  document.getElementById("info").innerHTML =
    `Hi ha productes sense estoc? ${productes.some(unitats => unitats === 0)}`;
}
```

<newcontent>

El mètode `some(funció)` comprova si hi ha algun element que compleix la funció que es passa com a argument. Si és així, la funció retorna *true*, si no, *false*.

El mètode `some` té la següent sintaxi:

```
let resultat = array.some(funcioComparacio[, parametres])
```

2.5.4 A quins productes se'ls ha esgotat l'estoc?

Aquesta funcionalitat és una modificació de l'anterior. No només volem saber si hi ha algun producte, sinó que volem saber quins són aquests productes.

El mètode que ens permet extreure una part d'un array segons una condició es diu `filter()`. Com que volem, a part dels valors, el número de producte, utilitzarem, igual que es va fer amb el rànquing dels productes més venuts, la cadena de text "unitats-producte" com a valor de l'array sobre el que utilitzarem el mètode `filter`.

Fixeu-vos en el codi:

```
// Volem saber quins productes tenen 0 unitats d'estoc (filter)
function checkBuitSplit(unitats) {
  let aa = unitats.split("-")[0];
  return parseInt(aa) === 0;
}
function llistatProductesSenseEstoc() {
  let llistat = "No hi ha productes sense estoc.";

  if (productes.some(unitats => unitats === 0)) {
    let aux = [];
    let arr = [];
    llistat = "";
    for (let numproducte in productes) {
      arr.push(productes[numproducte] + "-" + numproducte);
    }
    aux = arr.filter(checkBuitSplit);
    for (let index in aux) {
      llistat +=
        `<li>El producte ${aux[index].split("-")[1]} no té estoc.</li>`;
    }
    document.getElementById("info").innerHTML = llistat;
  } else {
    document.getElementById("info").innerHTML = llistat;
  }
}
```

::important: El mètode `filtre(funció)` retorna un nou array amb tots els elements pels quals la funció ha retornat `true`. :: El mètode `filter` té la següent sintaxi:

```
resultat = array.filter(funcioComprovacio[, parametres])
```

La funció `filter` crea un nou array amb els elements que passen la funció de comprovació proporcionada a la funció.

2.5.5 Llistat dels estocs del producte X al producte Y

Es vol obtenir el llistat dels estocs, des d'una posició de l'array fins a una altra. Les dues posicions les escull l'usuari.

Fixeu-vos en el codi següent:

```
// Volem un llistat de l'estoc des del producte X al producte Y (slice)
function llistatParcialdEstoc() {
  let ini = parseInt(document.getElementById("valor1").value);
  let fin = parseInt(document.getElementById("valor2").value);
  let estoc = productes.slice(ini, fin);
  document.getElementById("info").innerHTML = estoc.toString();
}
```

> El mètode `slice(inici, final)` retorna un nou array amb els valors que es troben des de l'índex `inici` fins a l'última posició de l'array o fins a l'índex `final` si s'ha passat el segon argument.

El mètode `slice` té la següent sintaxi:

```
nou array = array.slice([començament[, final]])
```

Com es pot veure, la funció `llistatParcialdEstoc` assigna a les variables `ini` i `fin` els valors introduïts per l'usuari a les caixes de text. A continuació es fan servir aquestes variables per invocar al mètode `slice`, que retorna un nou array amb els valors de `productes` compresos entre aquests dos indexs i, finalment, és mostra aquest nou array a la pàgina convertint-los en una cadena de text mitjançant el mètode `string`.

2.5.6 Afegir un producte a partir d'una posició

Es vol afegir un producte a partir d'una posició donada de l'array `productes`. Tots els productes que hi hagi després de l'índex d'inserció es desplaçaran tantes unitats com productes afegits.

Veieu l'exemple:

```
//  Volem afegir 1 producte a partir d'una posició X de l'array productes(
    splice).

function afegirEstocPosicioValor1() {
  let ini = parseInt(document.getElementById("valor1").value);
  productes.splice(ini, 0, Math.floor(Math.random() * 100));
  veureEstoc();
}
```

Com es pot apreciar, primerament obtenim la informació introduïda per l'usuari, la convertim en nombre enter i l'assignem a la variable `ini`. Aquest serà el punt d'inserció.

No volem eliminar cap element, així que com a segon argument de la funció posem `0` i, finalment, generem un valor pseudoaleatori entre 0 i 99, que passem com a tercer paràmetre:

```
productes.splice(ini, 0, Math.floor(Math.random() * 100));
```

El mètode `splice(inici, eliminats, valor1, valor2,... valorN)` afegeix i elimina elements a partir d'una posició de l'array.

Com que `splice` accepta qualsevol quantitat de valors separats per comes, es pot utilitzar l'operador de propagació per inserir un array en aquesta posició, per exemple:

```
let a = ['a', 'b', 'c', 'd'];
let b = [1, 2, 3, 4];
a.splice(1,0, ...b); // insereix a la segona posició tots els elements de b
```

En concret, els paràmetres de la funció son els següents:

- *inici*: és la posició on començarà a afegir o eliminar.
- *eliminats*: és el nombre d'elements que es volen esborrar a partir del paràmetre *inici*. Si posem un 0, només afegirà valors.
- *valor1, valor2,... valorN*: són els valor que volem afegir.

A continuació podeu veure un exemple amb els diferents resultats d'utilitzar el mètode `splice`:

```
let array = [1, 2, 3, 4, 5];
array.splice(1, 3);
// Ara 'array elimina 3 elements a partir de la posició 1,  i queda així: [1, 5]
console.log(`Contingut del array (1): ${array}`);

array = [1, 2, 3, 4, 5];
array.splice(2, 0, 2.5);
// Ara 'array' afegeix l'element 2.5 a partir de la posició 2 i queda així: [1, 2, 2.5, 3, 4, 5]
console.log(`Contingut del array (2): ${array}`);

array = [1, 2, 3, 4, 5, 6];
array.splice(2, 3);
// Ara 'array' elimina 3 elements a partir del segon element (no inclòs) i queda així: [1, 2, 6]
console.log(`Contingut del array (3): ${array}`);

array = [1, 2, 3, 4, 5];
array.splice(1, 3, "two", "three", "four");
// Ara 'array' elimina 3 elements a partir del primer element (no inclòs) i s' afegeixen
// 'two', 'three' i 'four' i queda així: [1, "two", "three", "four", 5]
console.log(`Contingut del array (4): ${array}`);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/LYNEKQK?editors=0012.