Conceptes previs de JavaScript


En aquest document donarem un breu repàs a alguns conceptes de JavaScript que utilitzarem al llarg del curs, i amb els quals convé que ens comencem a familiaritzar des de ja, si no els hem utilitzats encara. En concret, tractarem:

- Maneres de definir variables i constants, i estructures complexes de dades
- L'ús de "funcions fletxa" (*arrow functions*), també conegudes com a "expressions lambda", com a alternativa a la definició clàssica de funcions, o a l'ús de funcions anònimes.
- Com gestiona JavaScript el codi asíncron, mitjançant *callbacks*, promeses o l'especificació *async/await*

1. Variables i estructures de dades

1.1. Declaració de variables i constants

Per a començar, tractarem sobre la declaració de **variables**. La forma més típica que podem trobar en Internet per a declarar variables en JavaScript és mitjançant la paraula reservada `var`, que permet declarar variables de qualsevol tipus. Per exemple:

```
var nom = "Nacho";
var edat = 41;
```

Sin embargo, esta forma de declarar variables tiene algunos inconvenientes, como por ejemplo, y sobre todo, el hecho de declarar una variable de forma local a un ámbito, y que pueda ser utilizada desde fuera de ese ámbito, porque la variable es válida dentro de la función donde se ha definido. Así, por ejemplo, este código funcionaría, y mostraría "Nacho" como nombre en ambos casos, a pesar de que, intuitivamente, la variable `nombre` no debería existir fuera del *if*:

```
if (2 > 1)
{
    var nom = "Nacho";
    console.log("Nom dins:", nom);
}
console.log("Nom fora:", nom);   // "Nacho"
```

Per a evitar aquestes vulnerabilitats, emprarem la paraula reservada `let`, en lloc de `var`, per a declarar variables:

```
if (2 > 1)
{
    let nom = "Nacho";
    console.log("Nom dins:", nom);
}
console.log("Nom fora:", nom);   // Variable no definida
```

D'aquesta manera, l'àmbit de cada variable queda restringit al bloc on es declara, i el codi anterior provocaria un error.

Recordem també que podem emprar la paraula `const` per a definir **constants** en el codi. Això serà particularment útil tant per a definir constants convencionals (com un text o número fix, per exemple) com per a carregar llibreries, com veurem en sessions posteriors.

```
const pi = 3.1416;
```

1.1.2. Estructures heterogènies de dades

Les estructures de dades o objectes literals en JavaScript són col·leccions dinàmiques de parells propietat-valor, on la propietat sempre és una cadena i el valor pot ser un tipus de dada bàsica, un objecte o fins i tot una funció. Per exemple:

```
let persona = {
    nom: "Maria",
    edat: 41,
    telefon: "666555444"
};
```

Podem accedir a una propietat utilitzant el punt `.` o la notació cuadrada `[ ]`:

```
let nom = persona.nom;       // Maria
let edat = persona["edat"];  // 41
```

A continuació es mostra un altre exemple on el valor d'una propietat és un altre objecte:

```
let persona = {
    nom: "Maria",
    edat: 41,
    telefon: "666555444",
    direccio: {
        via: "Avinguda",
        nom: "Miguel Hernández",
        numero: 62
    }
};
```

El valor de la propietat `direccio` és un nou objecte. Per a accedir al valor d'alguna propietat d'aquest objecte farem el següent:

```
let via = persona.direccio.via; // Avinguda
let numero = persona["direccio"]["numero"]; // 62
```

2. Funcions y *arrow functions*

Vegem ara les diferents maneres de definir funcions o mètodes que existeixen en JavaScript, i amb això, introduirem un concepte que s'ha tornat molt habitual, i que utilitzarem sovint en aquestes anotacions. Es tracta d'una notació alternativa per a definir mètodes o funcions, les anomenades *arrow functions* (també conegudes com a *funcions fletxa* o *funciones lambda*).

2.1. Les funcions tradicionals

Comencem per un exemple senzill. Suposem aquesta funció tradicional que retorna la suma dels dos paràmetres que se li passen:

```
function sumar(num1, num2)  {
    return num1 + num2;
}
```

A l'hora d'utilitzar aquesta funció, n'hi ha prou amb cridar-la en el lloc desitjat, passant-li els paràmetres adequats. Per exemple:

```
console.log(sumar(3, 2)); // Mostrarà 5
```

2.2. Les funcions anònimes

Aquesta mateixa funció també podríem expressar-la com una funció anònima. Aquestes funcions es declaren "sobre la marxa", i se solen assignar a una variable per a poder-les nomenar o cridar després:

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

2.3. Les *arrow functions*

Les "funcions fletxa" o *arrow functions* suposen una manera de definir funcions que empra una expressió lambda per a especificar els paràmetres d'una banda (entre parèntesis) i el codi de la funció per un altre entre claus, separats per una fletxa. Es prescindeix de la paraula reservada `function` per a definir-les.

La mateixa funció anterior, expressada com *arrow function*, quedaria així:

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

Igual que ocorre amb les funcions anònimes, es pot assignar el seu valor a una variable per a usar-lo més endavant, o bé definir-les sobre la marxa en un fragment de codi determinat.

De fet, el codi anterior pot simplificar-se encara més: en el cas que la funció simplement retorne un valor, es pot prescindir de les claus i de la paraula `return`, quedant així:

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

A més, si la funció té un únic paràmetre, es poden prescindir dels parèntesis. Per exemple, aquesta funció retorna el doble del número que rep com a paràmetre:

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

2.3.1. Ús directe d'*arrow functions*

Com comentàvem abans, les *arrow functions*, així com les funcions anònimes, tenen l'avantatge de poder utilitzar-se directament en el lloc on es precisen. Per exemple, donat el següent llistat de dades personals:

```
let dades = [
    {nom: "Nacho", telefon: "966112233", edat: 41},
    {nom: "Ana", telefon: "911223344", edat: 36},
    {nom: "Mario", telefon: "611998877", edat: 15},
    {nom: "Laura", telefon: "633663366", edat: 17}
];
```

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

```
let majorsEdat = dades.filter(function(persona) {
    return persona.edat >= 18;
})
console.log(majorsEdat);
```

I també podem emprar una *arrow function* en el seu lloc:

```
let majorsEdat = dades.filter(persona => persona.edat >= 18);
console.log(majorsEdat);
```

Notar que, en aquests casos, no assignem la funció a una variable per a usar-la més tard, sinó que s'empren en el mateix punt on es defineixen. Notar també que el codi queda més compacte emprant una *arrow function*.

2.4. *Arrow functions* i funcions tradicionals

La diferència entre les *arrow functions* i la nomenclatura tradicional o les funcions anònimes és que amb les *arrow functions* no podem accedir a l'element `this`, o a l'element `arguments`, que sí que estan disponibles amb les funcions anònimes o tradicionals. Així que, en cas de necessitar fer-ho, haurem d'optar per una funció normal o anònima, en aquest cas.

> **Exercici 1:**
>
> Crea una carpeta anomenada "**ArrowFunctions**" en el teu espai de treball, en la carpeta de "*Exercicis*". Crea un arxiu font dins anomenat `arrow_functions.js` amb el següent codi:

```
let dades = [
    {nom: "Nacho", telefon: "966112233", edat: 41},
    {nom: "Ana", telefon: "911223344", edat: 36},
    {nom: "Mario", telefon: "611998877", edat: 15},
    {nom: "Laura", telefon: "633663366", edat: 17}
];

novaPersona({nom: "Juan", telefon:"965661564", edat: 60});
novaPersona({nom: "Rodolfo", telefon:"910011001", edat: 20});
esborrarPersona("910011001");
console.log(dades);
```

Hem definit un vector amb dades de persones, i un programa principal que anomena dues vegades a una funció `novaPersona`, passant-li com a paràmetres els objectes amb les dades de les persones a afegir. Després, cridem a una funció `esborrarPersona`, passant-li com a paràmetre un número de telèfon, i vam mostrar el vector de persones amb les dades que hi haja.

Has d'implementar les funcions `novaPersona` i `esborrarPersona` perquè facen la seua comesa. La primera rebrà la persona com a paràmetre i, si el telèfon no existeix en el vector de persones, l'afegirà. Per a això, pots utilitzar el mètode `push` del vector:

```
dades.push(persona);
```

Quant a `esborrarPersona`, eliminarà del vector a la persona que tinga aquest telèfon, en cas que existisca. Per a eliminar a la persona del vector, pots simplement filtrar les persones el telèfon de les quals no siga l'indicat, i assignar el resultat al propi vector de persones:

```
dades = dades.filter(persona => persona.telefon != telefonABuscar);
```

3. Programació asíncrona

En programació existeixen dues grans maneres d'invocar o cridar a les funcions:

- Invocació **síncrona**: és la més habitual i tradicional, en la qual invoquem a una funció i el programa espera que acabe i produïsca la seua resultat
- Invocació **asíncrona**: només disponible en alguns llenguatges i en determinades funcions, permet invocar a la funció i continuar amb l'execució normal del programa. Quan la funció acabe el seu treball, es deixa especificat com es recull el resultat.

3.1. Els *callbacks*

Un dels pilars en els quals se sustenta la programació asíncrona en JavaScript ho conformen els *callbacks*. Un *callback* és una funció Al fet que es passa com a paràmetre a una altra B, i que serà anomenada en algun moment durant l'execució de B (normalment quan B finalitza la seua tasca). Aquest concepte és fonamental per a dotar a Node.js (i a JavaScript en general) d'un comportament asíncron: es diu a una funció, i se li deixa indicat el que ha de fer quan acabe, i mentrestant el programa pot dedicar-se a altres coses.

Un exemple el tenim amb la funció `setTimeout` de JavaScript. A aquesta funció li podem indicar una funció a la qual anomenar, i un temps (en mil·lisegons) que esperar abans de cridar-la. Executada la línia de l'anomenada a `setTimeout`, el programa segueix el seu curs i quan el temps expira, es diu a la funció *callback* indicada.

Provem d'escriure aquest exemple en un arxiu anomenat `callback.js` en nostra subcarpeta "*ProjectesNode/Proves/ProvesSimples*":

```
setTimeout(function() {console.log("Finalitzat callback");}, 2000);
console.log("Hola");
```

Si executem l'exemple, veurem que el primer missatge que apareix és el de "Hola", i passats dos segons, apareix el missatge de "Finalitzat callback". És a dir, hem anomenat a `setTimeout` i el programa ha seguit el seu curs després, ha escrit "Hola" per pantalla i, una vegada ha passat el temps estipulat, s'ha anomenat al *callback* per a fer el seu treball.

Utilitzarem *callbacks* àmpliament durant aquest curs. De manera especial per a processar el resultat d'algunes promeses que emprarem (ara veurem què són les promeses), o el tractament d'algunes peticions de serveis.

3.2. Les promeses

Les promeses són un altre mecanisme important per a dotar d'asincronia a JavaScript. S'empren per a definir la finalització (reeixida o no) d'una operació asíncrona. En el nostre codi, podem definir promeses per a realitzar operacions asíncrones, o bé (més habitual) utilitzar les promeses definides per uns altres en l'ús de les seues llibreries.

Al llarg d'aquest curs utilitzarem promeses per a, per exemple, enviar operacions a una base de dades i recollir el resultat de les mateixes quan finalitzen, sense bloquejar el programa principal. Però per a entendre millor què és el que farem, arribat el moment, convé tindre clara l'estructura d'una promesa i les possibles respostes que ofereix.

3.2.1. Crear una promesa. Elements a tindre en compte

En el cas que vulguem o necessitem crear una promesa, es crearà un objecte de tipus `Promise`. A aquest objecte se li passa com a paràmetre una funció amb dos paràmetres:

- La funció *callback* a la qual anomenar si tot ha anat correctament
- La funció *callback* a la qual anomenar si hi ha hagut algun error

Aquests dos paràmetres se solen cridar, respectivament, `resolve` i `reject`. Per tant, un esquelet bàsic de promesa, emprant *arrow functions* per a definir la funció a executar, seria així:

```
let nombVariable = new Promise((resolve, reject) => {
    // Codi a executar
    // Si tot va bé, cridem a "resolve"
    // Si alguna cosa falla, cridem a "reject"
});
```

Internament, la funció farà el seu treball i cridarà als seus dos paràmetres en l'un o l'altre cas. En el cas de `resolve`, se li sol passar com a paràmetre el resultat de l'operació, i en el cas de `reject` se li sol passar l'error produït.

Vegem-ho amb un exemple. La següent promesa busca els majors d'edat de la llista de persones vista en un exemple anterior. Si es troben resultats, es retornen amb la funció `resolve`. En cas contrari, es genera un error que s'envia amb `reject`. Còpia l'exemple en un arxiu anomenat `prova_promesa.js` en la carpeta "*ProjectesNode/Proves/ProvesSimples*" del teu espai de treball:

```
let dades = [
 {nom: "Nacho", telefon: "966112233", edat: 41},
 {nom: "Ana", telefon: "911223344", edat: 36},
 {nom: "Mario", telefon: "611998877", edat: 15},
 {nom: "Laura", telefon: "633663366", edat: 17}
];

let promesaMajorsEdat = new Promise((resolve, reject) => {
    let resultat = dades.filter(persona => persona.edat >= 18);
    if (resultat.length > 0)
        resolve(resultat);
    else
        reject("No hi ha resultats");
});
```

La funció que defineix la promesa també es podria definir d'aquesta altra forma:

```
let promesaMajorsEdat = llistat => {
    return new Promise((resolve, reject) => {
        let resultat = llistat.filter(persona => persona.edat >= 18);
        if (resultat.length > 0)
            resolve(resultat);
        else
            reject("No hi ha resultats");
    });
};
```

Així no fem ús de variables globals, i l'array queda passat com a paràmetre a la pròpia funció, que retorna l'objecte `Promise` una vegada concloga. Deixa definida la promesa d'aquesta segona forma en l'arxiu font de prova.

3.2.2. Consum de promeses

En el cas de voler utilitzar una promesa prèviament definida (o creada per uns altres en alguna llibreria), simplement cridarem a la funció o objecte que desencadena la promesa, i recollim el resultat. En aquest cas:

- Per a recollir un resultat satisfactori (`resolve`) emprem la clàusula `then` .
- Per a recollir un resultat erroni (`reject`) emprem la clàusula `catch` .

Així, la promesa anterior es pot emprar d'aquesta manera (novament, emprem *arrow functions* per a processar la clàusula `then` amb el seu resultat, o el `catch` amb el seu error):

```
promesaMajorsEdat(dades).then(resultat => {
    // Si entrem ací, la promesa s'ha processat bé
    // En "resultat" podem accedir al resultat obtingut
    console.log("Coincidències trobades:");
    console.log(resultat);
}).catch(error => {
    // Si entrem ací, hi ha hagut un error en processar la promesa
    // En "error" el podem consultar
    console.log("Error:", error);
});
```

Còpia aquest codi sota el codi anterior en l'arxiu `prova_promesa.js` creat anteriorment, per a comprovar el funcionament i el que mostra la promesa.

Notar que, en definir la promesa, es defineix també l'estructura que tindrà el resultat o l'error. En aquest cas, el resultat és un vector de persones coincidents amb els criteris de cerca, i l'error és una cadena de text. Però poden ser el tipus de dada que vulguem.

> **Exercici 2:**

> Crea una carpeta anomenada "**Promeses**" en el teu espai de treball, en la carpeta de "*Exercicis*". Crea dins un arxiu font anomenat `promeses.js`, que siga una còpia de l'arxiu font `arrow_functions.js` de l'exercici anterior.
>
> El que faràs en aquest exercici és adaptar les dues funcions `novaPersona` i `esborrarPersona` perquè retornen una promesa.
>
> En el cas de `novaPersona`, es retornarà amb `resolve` l'objecte persona inserit, si la inserció va ser satisfactòria, o amb `reject` el missatge "Error: el telèfon ja existeix" si no es va poder inserir la persona perquè ja existia el seu telèfon en el vector
>
> En el cas d'`esborrarPersona`, es retornarà amb `resolve` l'objecte persona eliminat, si l'esborrat va ser satisfactori, o amb `reject` un missatge "Error: no es van trobar coincidències" si no existia cap persona amb aqueix telèfon en el vector.
>
> Modifica el codi del programa principal perquè intente afegir una persona correcta i una altra equivocada (telèfon ja existent en el vector), i esborrar una persona correcta i una altra equivocada (telèfon no existent en el vector). Comprova que el resultat en executar és el que esperaves.

3.2.3. L'especificació *async/await*

Des de *ECMAScript7* es té disponible una nova manera de treballar amb crides asíncrones, a través de l'especificació **async/await**. És una forma més còmoda de cridar a funcions asíncrones i recollir el seu resultat abans de cridar a una altra, sense necessitat d'anar niant clàusules `then` per a enllaçar el resultat d'una promesa amb la següent.

No entrarem en els detalls sobre com utilitzar-la de moment. Ho farem més endavant, quan estiguem familiaritzats amb les promeses.Però, perquè puguem fer-nos una idea del que implica, reescriurem un exemple anterior fet amb promeses usant aquesta especificació. Partim del mateix vector de persones:

```
let dades = [
    {nom: "Nacho", telefon: "966112233", edat: 41},
    {nom: "Ana", telefon: "911223344", edat: 36},
    {nom: "Mario", telefon: "611998877", edat: 15},
    {nom: "Laura", telefon: "633663366", edat: 17}
];
```

Construïm ara la nostra funció per a buscar persones majors d'edat. És similar a l'anterior, però li afegim la partícula `async` per a indicar que és una funció asíncrona:

```
let promesaMajorsDeEdat = async llistat => {
    return new Promise((resolve, reject) => {
        let resultat = llistat.filter(persona => persona.edat >= 18);
        if (resultat.length > 0)
            resolve(resultat);
        else
            reject("No hi ha resultats");
    });
};
```

> **NOTA:** en realitat, en aquest cas no fa falta afegir la partícula `async` perquè la funció, en retornar una promesa, ja és automàticament asíncrona. Però es pot seguir aquest costum en programar.

A l'hora d'invocar a aquesta funció podem fer-ho de la mateixa manera que abans (amb *then/catch*) o usant la partícula `await`. Aquesta partícula fa que el codi del programa s'espere que la funció finalitze per a després continuar:

```
let adults = await promesaMajorsDeEdat(dades);
// En arribar ací ja tenim el llistat
console.log(adults);
```

No obstant això, un dels requisits que estableix l'especificació és que no podem utilitzar la clàusula `await` fora d'un bloc asíncron. Per tant, és habitual definir una funció asíncrona que invoque a la resta, i cridar a aquesta des del programa principal:

```
let promesaMajorsDeEdat = async llistat => {
    return new Promise((resolve, reject) => {
        let resultat = llistat.filter(persona => persona.edat >= 18);
        if (resultat.length > 0)
            resolve(resultat);
        else
            reject("No hi ha resultats");
    });
};

async function principal()
{
    let adults = await promesaMajorsDeEdat(dades);
    console.log(adults);
}
```

En el cas que la invocació siga reeixida es retornarà el llistat, que recollim en la variable `adults`. Però, què ocorre si alguna cosa falla? En aquest cas podem utilitzar un bloc `try..catch` per a capturar l'excepció i

mostrar el missatge d'error que es produïsca. A més, podem enllaçar aquests blocs un darrere l'altre per a assegurar-nos que una cosa s'execute quan acabe l'anterior.

```
async function principal()
{
    try
    {
        let adults = await promesaMajorsDeEdat(dades);
        console.log("Resultats:", adults);
    } catch(e) {
        // Error
        console.log(e);
    }

    // Una altra crida sincronitzada...
    try
    {
        let variable = await ...;
    } catch(e) {
        console.log(e);
    }
}
```