Seminar 5

Asynchronous programming in Javascript

❖ Closures

Un **closure** este o funcție care are acces la domeniul părinte, chiar și după ce funcția părinte a fost închisă.

Utilizarea unui closure este util pentru ascunderea detaliilor de implementare. Cu alte cuvinte, poate fi util pentru a crea variabile sau funcții private.

Closures în JavaScript utilizează **'lexical scope'** pentru a accesa variabilele din funcții părinte, chiar și după încheierea funcției părinte. Acest lucru permite păstrarea datelor și comportamentului privat.

Acest **'scope'** reprezintă contextul în care valorile și expresiile sunt "vizibile" sau pot fi referite. Dacă o variabilă sau altă expresie nu se află "în domeniul de aplicare curent", atunci nu este disponibilă pentru utilizare. Domeniile de aplicare pot fi stratificate într-o ierarhie (**scope chain**), astfel încât domeniile de aplicare copil au acces la domeniile de aplicare părinte, dar nu invers.

De reținut: Un closure este creat atunci când creăm o funcție, nu atunci când este executată.

```
const privateCounter = (() => {
    let count = 0;

    console.log(`initial value of count: ${count}`);

    return () => {
        count++;
        console.log(`current value of count: ${count}`);
    };
})();

privateCounter();
// first log: initial value of count: 0
// second log: current value of count: 1
privateCounter();
// first log: current value of count: 2
privateCounter();
// first log: current value of count: 3
```

❖ Asynchronicity in Javascript

JavaScript este un limbaj de programare **sincron**, ceea ce înseamnă că o să executăm codul într-o singură linie, de sus în jos, în ordine, blocând execuția până când operațiile curente sunt finalizate. Cu toate acestea, JavaScript poate declanșa task-uri asincrone utilizând API-urile **furnizate de browser**.

Acestea includ:

1. **Timers**: JavaScript poate seta timer-e pentru a declanșa funcții după un anumit interval de timp. De exemplu, **setTimeout** și **setInterval** permit programatorilor să creeze funcții care se vor executa la un moment ulterior sau la intervale regulate.
2. **Event Listeners**: JavaScript poate asculta evenimente în browser, cum ar fi clicuri de mouse, tastatură, sau rețea. Atunci când un eveniment specific are loc, funcțiile asociate sunt declanșate asincron pentru a răspunde la acele evenimente.
3. **Promises și Fetch API**: Pentru comunicarea asincronă cu servere sau alte resurse, JavaScript utilizează Promises și fetch(), permițând trimiterea **cererilor HTTP** și gestionarea răspunsurilor în mod asincron.
4. **Web Workers**: Acestea permit crearea de fire de execuție suplimentare pentru sarcini costisitoare din punct de vedere computațional, precum procesarea în paralel a datelor.

Aceste tipuri de API-uri asincrone permit JavaScript să execute sarcini în fundal, fără a bloca interacțiunea utilizatorului sau procesarea principală. Aceasta face posibilă dezvoltarea aplicațiilor web interactive și responsive, care pot răspunde rapid la evenimente și pot comunica cu resurse externe fără a bloca execuția principală a codului.

API (Application Programming Interface) este un **set de reguli și protocoale** care permit comunicarea și interacțiunea între diferite componente software. API-urile definesc modul în care alte programe sau servicii pot solicita sau accesa funcționalitățile și datele oferite de o aplicație sau platformă software.

❖ Callbacks

Callback-urile sunt funcții care sunt transmise ca argumente către alte funcții și sunt apelate într-un moment ulterior, de obicei într-un context asincron. Ele sunt folosite pentru a gestiona execuția codului după finalizarea unui proces asincron sau după producerea unui eveniment.

Synchronous callbacks

Callback-urile sincrone sunt funcții care sunt apelate imediat, în același fir de execuție, în timpul execuției funcției care le înregistrează. Acest lucru înseamnă că funcția care conține callback-ul va aștepta să se termine execuția callback-ului înainte de a continua. Acest lucru se întâmplă în mod secvențial, pe rând.

```
const isEven = (num) => num % 2 === 0;
const isOdd = (num) => num % 2 !== 0;

const filter = (arr, callback) => {
    const filteredArray = [];

    for (const element of arr) {
        if (callback(element)) {
            filteredArray.push(element);
        }
    }

    return filteredArray;
}

const numbers = [1, 2, 3, 4, 5, 6, 7, 8];

console.log(filter(numbers, isEven));  // [2, 4, 6, 8]
console.log(filter(numbers, isOdd));  // [1, 3, 5, 7]
```

Asynchronous callbacks

Callback-urile asincrone sunt funcții care **nu** sunt apelate imediat, ci sunt înregistrate pentru a fi apelate ulterior, într-un context asincron. Ele sunt utilizate în special pentru a gestiona operațiuni care pot dura mai mult sau care pot fi executate în fundal, fără a bloca firul principal de execuție. Acest lucru permite programului să continue să ruleze în timp ce așteaptă finalizarea unei operațiuni asincrone.

Callbacks use cases

1. **Manipularea evenimentelor DOM**: înseamnă gestionarea interacțiunilor utilizatorului cu paginile web. Aceasta implică detectarea și răspunsul la evenimente, cum ar fi clicuri, hover sau tastă apăsată, utilizând JavaScript. Programatorii pot crea funcții de tratare a evenimentelor pentru a modifica conținutul, stilul sau comportamentul paginii web în funcție de acțiunile utilizatorului, permițând dezvoltarea de interfețe interactive și responsive.

```
document.getElementById('button').addEventListener('click', function() {
    console.log('button clicked');
});
```

2. **setTimeout:** este folosită în JavaScript pentru a programa execuția unei funcții sau a unui bloc de cod după o anumită întârziere de timp, măsurată în milisecunde.

   Funcția **setTimeout** primește doi parametri:

   1. **Funcția de callback**: Funcția sau codul care va fi executat după scurgerea timpului specificat.
   2. **Timpul de întârziere**: Intervalul de timp în milisecunde înainte ca funcția de callback să fie apelată.

```
function fetchDataFromServer(url, callback) {
    setTimeout(function () {
      const data = { name: "John", age: 30 };
      callback(data);
    }, 1000);
}

function displayData(data) {
    console.log("Data received from the server:", data);
}

fetchDataFromServer("https://example.com/api/data", displayData);
console.log("The request has been initiated.");
```

3. **setInterval:** este folosită pentru a programa execuția repetată a unei funcții sau a unui bloc de cod la intervale de timp fixe, măsurate în milisecunde.

Funcția **setInterval** primește doi parametri:

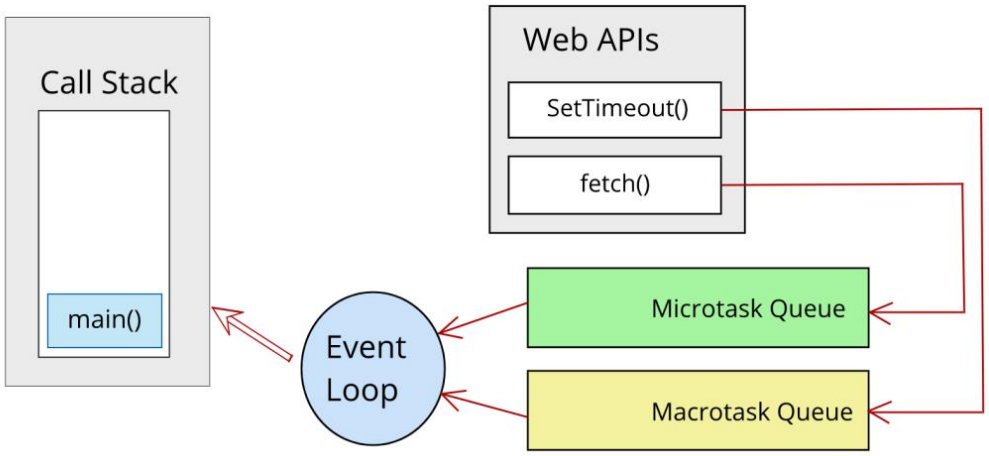
1. Funcția de callback repetată: Funcția sau codul care se execută periodic la intervale de timp specificate.
2. Timpul de interval: Intervalul de timp în milisecunde între fiecare apel repetat al funcției de callback.

```
let count = 0;

const privateCounter = setInterval(() => {
    count++;
    console.log(`current value of count: ${count}`);

    if (count === 5) {
        clearInterval(privateCounter);
        console.log('counter stopped');
    }
}, 1000);
```

De reținut: Metoda **setInterval()** continuă să apeleze funcția până când este apelată **clearInterval()**, sau fereastra este închisă.



❖ Promises

Un **Promise** reprezintă finalizarea (sau eșecul) viitor al unei operații asincrone și furnizează o modalitate de a lucra cu cod asincron într-un mod mai structurat și mai ușor de citit. Atunci când un Promise este rezolvat (fie îndeplinit, fie respins), acesta conține o valoare sau o eroare la care se poate accesa.

Promisiunile au trei stări posibile**:**

1. **În așteptare (pending):** Acesta este stadiul inițial al unui Promise. În acest moment, promisiunea nu a fost nici îndeplinită, nici respinsă. Rămâne în acest stadiu până când operația asincronă asociată se finalizează.
2. **Îndeplinit (fulfilled):** O promisiune este în stadiul îndeplinit atunci când operația asincronă s-a încheiat cu succes. Acesta este momentul în care se furnizează o valoare rezultat. Puteți accesa valoarea rezultat folosind metoda .then() în lanțul promisiunii.
3. **Respins (rejected):** O promisiune devine respinsă atunci când operația asincronă se încheie cu o eroare sau o excepție. În acest caz, puteți accesa eroarea pentru gestionare și tratare ulterioară folosind metoda .catch() în lanțul promisiunii sau .then() cu al doilea argument (funcția de gestionare a erorii).

```
let promise = new Promise((resolve, reject) => {
    //executor
    setTimeout(() => {
        const success = false;

        if (success) {
            resolve("Operation succeeded");
        } else {
            reject(new Error("Operation failed"));
        }
    }, 2000);
});


promise.then((message) => {
        console.log(message);
    }).catch((error) => {
        console.log(error.message);
});
```

❖ Async-Await

Există o sintaxă specială pentru a lucra cu promisiuni într-un mod mai confortabil, numită "async/await".

Async keyword

Cuvântul "async" înaintea unei funcții înseamnă un singur lucru simplu: o funcție întotdeauna returnează o promisiune. Alte valori sunt învelite automat într-o promisiune rezolvată.

```
async function f() {
    return 1;
}
```

este la fel ca:

```
function f() {
    return Promise.resolve(1);
}
```

Prin urmare, "async" asigură că funcția returnează o promisiune și învelește în automat non-promisiuni.

Await keyword

Cuvântul cheie "**await**" face ca JavaScript să aștepte până când acea promisiune se finalizează și să returneze rezultatul său.

```
async function f() {

    let promise = new Promise((resolve, reject) => {
      setTimeout(() => resolve("done!"), 1000)
    });

    let result = await promise; // wait until the promise resolves (*)

    console.log(result); // "done!"
}

f();
```

Pentru mai multe detalii legate de utilizarea async-await: https://javascript.info/async-await

❖ Requesting data from URLs

XMLHttpRequest vs. Fetch API

XHR înseamnă **XMLHttpRequest** și este un API pe care îl putem utiliza pentru a face cereri AJAX în JavaScript. Folosind acest API, putem efectua cereri de rețea pentru a schimba date între un site web și un server. XHR este folosit pentru a face cereri HTTP în JavaScript, însă nu reprezintă cea mai modernă abordare.

```
const xhr = new XMLHttpRequest();
xhr.open('GET', 'https://api.github.com/users/MihaiAdrianLungu');

xhr.onreadystatechange = function() {
    if (xhr.readyState === 4 && xhr.status === 200) {
        console.log(xhr.responseText);
    }
}

xhr.send();
```

Proprietatea **readyState** a obiectului XMLHttpRequest returnează starea curentă:

- 0 (neinițializată) - cererea nu este inițializată
- 1 (încărcare) - conexiunea la server a fost stabilită
- 2 (încărcat) - cererea a fost primită
- 3 (interactiv) - procesarea cererii
- 4 (complet) - cererea este finalizată, răspunsul este gata

Spre deosebire de XMLHttpRequest, care este o API **bazată pe callback-uri**, **Fetch** este **bazată pe promisiuni** și oferă o alternativă mai bună care poate fi utilizată ușor în serviceworkeri. De asemenea, Fetch integrează concepte avansate ale HTTP, cum ar fi CORS și alte extensii ale HTTP.

API-ul Fetch furnizează o interfață JavaScript pentru accesarea și manipularea părților ale protocolului, cum ar fi cererile și răspunsurile. De asemenea, oferă o metodă globală fetch() care furnizează o modalitate ușoară și logică de a prelua resurse în mod asincron pe rețea.

```
fetch('https://api.github.com/users/MihaiAdrianLungu')
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.log(error));
```

Folosind async-await:

```
async function fetchData() {
    try {
      const response = await
fetch('https://api.github.com/users/MihaiAdrianLungu');
      const data = await response.json();
      console.log(data);
    } catch (error) {
      console.log(error);
    }
  }

fetchData(); // Call the async function to start the data fetching process
```