1. Créons un environnement de dev

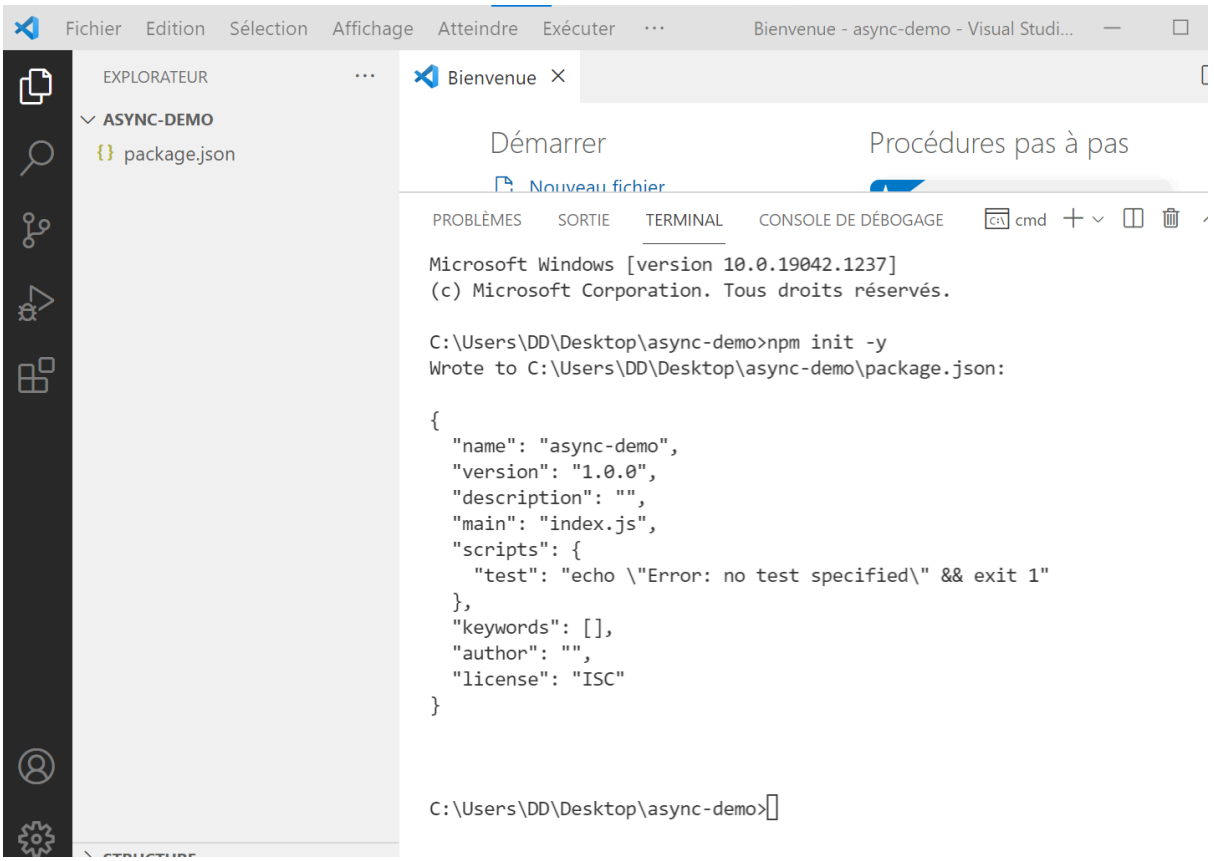
Nous allons lancer Visual Studio dans un 📁répertoire async-demo.

1. C:\Users\DD\Desktop>mkdir async-demo
2. C:\Users\DD\Desktop>cd async-demo
3. C:\Users\DD\Desktop\async-demo>node .

On peut initialiser le projet

4. C:\Users\DD\Desktop\async-demo>npm init -y[1]

Lig. 4 : L'option -y (yes) permet d'accepter les options de base sans pour autant devoir les valider une par une.



[1] Cette commande va vous être très bientôt familière.

2. index.js

Une fois le fichier index.js créé, écrivez le code suivant.

index.js

```
1. console.log("Avant");
2. setTimeout(() => {
3.     console.log("lecture d'un utilisateur à partir de la BD");
4. }, 2000);
5. console.log("Après")
```

Lancez maintenant l'application avec la commande *node index.js*

```
1. C:\Users\DD\Desktop\async-demo>node index.js
```

Avant

Après

lecture d'un utilisateur à partir de la BD

Vous ne devez pas être surpris du résultat.

Maintenant, modifiez la lig. 4 du fichier index.js et remplacer 2000 par 0

```
4. }, 2000);
```

par :

```
}, 0);
```

Commenter le résultat[2] !

[2] La séquence de code est la même !

3. le traitement du code asynchrone

Nous allons simuler le temps d'accès à une base de données pour appréhender la problématique de la gestion de code asynchrone.

Commençons par définir une fonction ***getUser.***

```
1. function getUser(id){
2.     setTimeout(() => {
3.         console.log("lecture d'un utilisateur à partir de la BD");
4.         return {id: id, gitHubUsername: "superDupont"}
5.     }, 2000);
6. }
```

Modifiez le code de index.js :

index.js

```
7. console.log("Avant");
8. const user = getUser(1);
9. console.log(user);
10. console.log("Après")
11.
12. function getUser(id){
13.     setTimeout(() => {
14.         console.log("lecture d'un utilisateur à partir de la BD");
15.         return {id: id, gitHubUsername: "superDupont"}
16.     }, 2000);
17. }
```

Lancez le programme dans un terminal :

```
C:\Users\DD\Desktop\async-demo>node index.js
```

L'exécution donnera l'affichage :

```
Avant
undefined
Après
lecture d'un utilisateur à partir de la BD
```

Modifiez la fonction *getUser* en retournant en ligne 11 l'identifiant id.

index.js

```
1. console.log("Avant");
2. const user = getUser(1);
3. console.log(user);
4. console.log("Après")
5.
6. function getUser(id){
7.     setTimeout(() => {
8.         console.log("lecture d'un utilisateur à partir de la BD");
9.         return {id: id, gitHubUsername: "superDupont"}
10.    }, 2000);
11.    return id;
12.}
```

Lancez l'exécution avec

```
C:\Users\DD\Desktop\npm>node test.js
```

```
Avant
1
```

Après

lecture d'un utilisateur à partir de la BD

Nous sommes loin de l'affichage attendu à savoir :

Avant

lecture d'un utilisateur à partir de la BD

Après

Pour corriger ce comportement, nous allons utiliser une fonction de rappel (callback).

4. Callbacks

Voici la structure du code envisagée.

```
1. console.log("Avant");
2. getUser(1, function(user) {
3.     console.log(`User, ${user}`)
4. });
5. console.log("Après")
```

Lig. 2 : getUser prend en paramètre **une fonction de callback.** Elle prend comme paramètre *user*.

Ainsi lorsque nous aurons récupéré l'utilisateur dans la première partie du code de getUser, nous pourrons exécuter la fonction de callback

```
1. console.log("Avant");
2. getUser(1, function(user) {
```

```
3.      console.log(`User, ${user}`)
4.  });
5.  console.log("Après")
```

Nous allons donc modifier la fonction getUser ainsi :

index.js

```
1.  console.log("Avant");
2.  getUser(1, function(user) {
3.      console.log(`User, ${JSON.stringify[3](user)}`)
4.  });
5.  console.log("Après")
6.
7.  function getUser(id, callback){
8.      setTimeout(() => {
9.          console.log("Reading a user from BD");
10.         callback({id: id, gitHubUsername: "superDupont"})
11.     }, 2000);
12.}
```

lig. 3 : JSON.stringify permet ici de visualiser l'objet

lig. 7 : *callback* est une référence vers une fonction

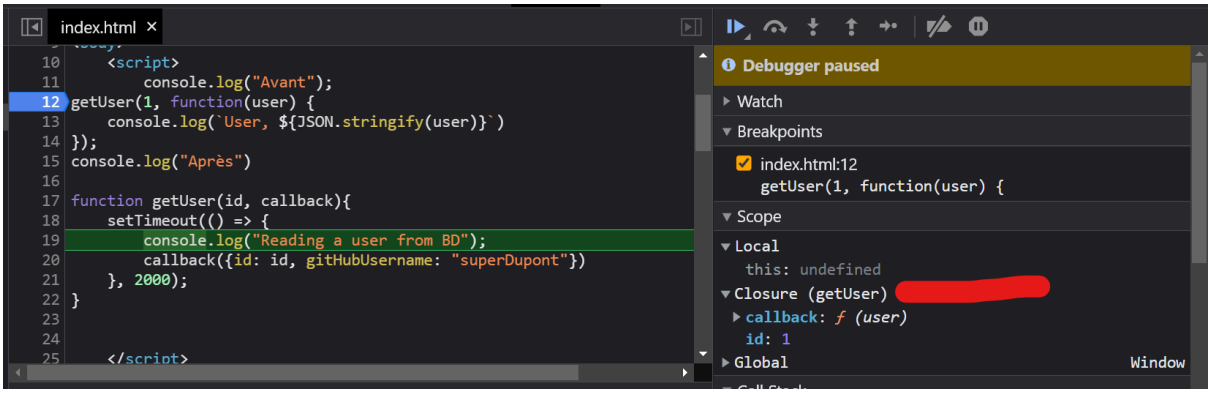
lig. 10 : Le paramètre actuel de la fonction de callback est passé.

[3] JSON est un format utilisé lors du transfert d'information. Son utilisation est classique.

5. Magie des closures

Il est vraiment important de comprendre qu'en ligne 7, on ne passe pas les arguments à la fonction. Ces arguments ne sont connus que dans la fonction encapsulante getUser et seront encore disponibles pour la fonction de callback **après** que la fonction getUser soit exécutée[4].

L'utilisation d'un débogueur[5] montrerait l'apparition de la closure.



L'exécution doit donner le résultat attendu.

```
C:\Users\DD\Desktop\async-demo>node index.js
```

Avant

Après

Reading a user from BD

User, {"id":1,"gitHubUsername":"superDupont"}

---

[4] C'est la magie des closures qui permet à une fonction qui a fini son exécution de mettre à disposition des variables à d'autres fonctions.

[5] Dans un fichier index.html insérer le code dans une balise script.

6. cb et fonction fléchée

Nous allons réécrire le code de la fonction index.js avec une **fonction fléchée.**

C'est une utilisation classique[6].

index.js

```
1. console.log("Avant");
2. getUser(1, (user) => {
3.     console.log(`User, ${JSON.stringify(user)}`)
4. });
5. console.log("Après")
6.
7. function getUser(id, callback){
8.     setTimeout(() => {
9.         console.log("Reading a user from BD");
10.        callback({id: id, gitHubUsername: "superDupont"})
11.    }, 2000);
12.}
```

Lig.2 : La fonction fléchée **=>**

```
C:\Users\DD\Desktop\npm>node index.js
Avant
Après
Reading a user from BD
```

[6] Elles sont très utiles pour conserver la valeur du this.

User, {"id":1,"gitHubUsername":"superDupont"}

7. Exercice

Soit la fonction synchrone **getRepositories** qui retourne un ensemble de repositories sous forme de tableau.

```
1. function getRepositories(username){
2.     return ['repos1','repos2','repos3'];
3. }
```

Lig. 2 : On retourne trois valeurs de dépôts 'repos1', 'repos2' et 'repos3' dans un tableau.

🤡Transformer la fonction synchrone getRepositories en une fonction **asynchrone** prenant 2 secondes pour obtenir les dépôts et les afficher dans la console.

---

solution

```
1. function getRepositories(username, callback){
2.     setTimeout(() => {
3.         console.log("Calling gitHub API ...");
4.         callback(['repos1','repos2','repos3'])
5.     }, 2000);
6. }
```

Voici le code général

callback.js

```
1. console.log("Avant");
2. getUser(1, (user) => {
3.     getRepositories(user.gitHubUsername, (repos) => {
4.         console.log(`Repos: ${repos}`)
5.     })
6. });
7. console.log("Après")
8.
9. function getUser(id, callback){
10.     setTimeout(() => {
11.         console.log("Reading a user from BD");
12.         callback({id: id, gitHubUsername: "superDupont"})
13.     }, 2000);
14.}
15.
16.function getRepositories(username, callback){
17.     setTimeout(() => {
18.         console.log("Calling gitHub API ...");
19.         callback(['repos1','repos2','repos3'])
    }, 2000);
20.}
```

```
C:\Users\DD\Desktop\npm>node callback.js
```

L'exécution donne l'affichage suivant :

```
Avant
```

Après

Reading a user from BD

Calling gitHub API ...

Repos: repos1,repos2,repos3

🤡Que devient le code si nous avions une fonction **getCommits** qui prend en paramètre un dépôt et qui donne les commits ?

Autrement dit, après avoir cherché un dépôt, on examine ses commits.

getUser.js

```
1. getUser(1, (user) => {
2.         getRepositories(user.gitHubUsername, (repos)=>{
3.                 getCommits(repo, (commits) =>
4.                         …
5.                 })
6.         })
7. });
```

Observez la structure du code getUser.

Cette structure en arbre de Noël évoque pour certains le😡 "callback hell"[7]

```
getUser(1, (user) => {
    getRepositories(user.gitHubUsername, (repos)=>{
        getCommits(repo, (commits) =>
            ...
        })
    })
});
```

Nous allons modifier notre code pour ne plus utiliser cette structure. Nous introduisons les promesses.

8. Promesses

Une promesse est un objet dont la valeur retournée est disponible via **then**

```
1. const p = new Promise((resolve, reject) =>
2.  // async
3.     resolve(valeurs)
4. });
5.
6. p.then( result => console.log(result))
```

Lig. 2 : Vous écrirez les instructions de code asynchrone.

Lig. 3 : Vous communiquez après un temps indéterminé des valeurs grâce à l'appel de la fonction resolve.

Lig. 6 : On peut continuer l'exécution avec les valeurs récupérées.

[7] l'enfer des callback

resolve

Revenons sur le mot clé **resolve**.

Considérez le code suivant et son exécution.

index.js

```
1. const p = new Promise((resolve, reject) => {
2.     setTimeout(()=>{
3.         resolve("Here we go")
4.     }, 2000);
5.     console.log("... wait")
6.
7. });
8.
9. p.then( result => console.log(`result: ${result}))
```

L'exécution donne

```
C:\Users\DD\>node test.js
... wait
result: Here we go
```

Reject

Découvrons le mot clé "**reject".**

Considérez le code suivant et son exécution.

index.js

```
1.  const p = new Promise((resolve, reject) => {
2.
3.     setTimeout(() => {
4.       reject(new Error('AIE ! pas de BD'));
5.     }, 2000);
6.   });
7.
8.   p
9.   .then((result) => console.log('Result', result))
10.  .catch((err) => console.log('Error:', err.message));
```

Lig. 4 : On rejette la promesse en envoyant un message

Lig. 9 : Notez le . devant then

Lig. 10 : On capture le message d'erreur.

L'exécution donne le résultat.

```
C:\Users\DD\Desktop\npm>node test.js
```

Error AIE ! pas de BD

Callback-> Promesse

Après ces rappels sur l'écriture des promesses, nous allons créer un nouveau fichier promise.js. Dans ce fichier, nous allons écrire notre code en transformant les callback en promesse.

Il n'y a pas vraiment de règle de transformation, mais nous pouvons indiquer qu'il faut :

1. Retourner une promesse
2. Dans la promesse, recopier le code de la fonction callback.js
3. Remplacer callback par resolve.

Le tableau ci dessous résume les étapes 1-2

1. Retourner une promesse
2. Dans la promesse, recopier le code de la fonction callback.js
3. Remplacer callback par resolve.

callback.js	promesse.js

`function getUser(id, callback){`
 	
`  setTimeout(() => {`
`  console.log('Reading from BD');`
`  callback({ id: id, gitHubUsername: 'superDupont' });`
`  }, 2000);`
 	
`}`
	`function getUser(id) {`
	`  return new Promise((resolve, reject) => {`
	`  setTimeout(() => {`
	`  console.log('Reading from BD');`
	`  resolve({ id: id, gitHubUsername: 'superDupont' });`
	`  }, 2000);`
	`  });`
	`}`

Le tableau ci dessous résume l'étape 3

1. Retourner une promesse
2. Dans la promesse, recopier le code de la fonction callback.js
3. Remplacer callback par resolve.

callback	promesse.js

function getUser(id, callback){
 	
setTimeout(() => {
console.log("Reading from BD");
callback({id: id, gitHubUsername: "superDupont"})
}, 2000);
 	
}
	function getUser(id) {
	return new Promise((resolve, reject) => {
	setTimeout(() => {
	console.log('Reading from BD');
	resolve({ id: id, gitHubUsername: 'superDupont' });
	}, 2000);
	});
	}

Voici une utilisation du code promesse.js

promesse.js

```
/**
 * Retrieves data from the database based on the provided ID.
 * @param {number} id - The ID of the data to retrieve.
 * @returns {Promise<Object>} A promise that resolves with the retrieved data.
 */
1.  function getUser(id) {
2.      return new Promise((resolve, reject) => {
3.          setTimeout(() => {
4.              console.log('Reading from BD');
5.              resolve({ id: id, gitHubUsername: 'superDupont' });
6.          }, 2000);
7.      });
8.  }
9.
10. const p = getUser(1);
11.   p.then(user => console.log(user));
```

En concaténant les lignes 10 et 11, on peut écrire en supprimant la constante p :

promesse.js

```
1. function getUser(id) {
2.     return new Promise((resolve, reject) => {
3.         setTimeout(() => {
4.             console.log('Reading from BD');
5.             resolve({ id: id, gitHubUsername: 'superDupont' });
6.         }, 2000);
7.     });
8. }
9.
10. getUser(1).then( user=> console.log(user));
```

Lancez la commande `node promesse.js`.

```
C:\Users\DD\Desktop\npm>node promesse.js
Reading from BD
{ id: 1, gitHubUsername: 'superDupont' }
```

Exercices

🤡Continuez la transformation des callback en promesse pour les fonctions `getRepositories` et `getCommits`.

Vous Testerez votre code avec le code suivant :

```
1. getUser(1)
```

```
2.  .then(user => getRepositories(user.gitHubUsername))
3.  .then(repos => getCommits(repos[0]))
4.  .then(commits => console.log('Commits', commits))
5.  .catch(err => console.log('Error', err.message));
```

Vous devriez obtenir l'affichage suivant :

```
C:\Users\DD\Desktop\async-demo>node promise.js
Before
After
Reading a user from a database...
Calling GitHub API...
Calling GitHub API...
Commits [ 'commit' ]
```

Correction

promesse.js

```
1.  function getUser(id) {
2.    return new Promise((resolve, reject) => {
3.      // Kick off some async work
4.      setTimeout(() => {
5.        console.log('Reading a user from a database...');
6.        resolve({ id: id, gitHubUsername: 'mosh' });
7.      }, 2000);
8.    });
9.  }
10.
```

```
function getRepositories(username) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log('Calling GitHub API...');
      resolve(['repo1', 'repo2', 'repo3']);
    }, 2000);
  });
}

function getCommits(repo) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log('Calling GitHub API...');
      resolve(['commit']);
    }, 2000);
  });
}
// promise-based approach
console.log('Before');

getUser(1)
  .then(user => getRepositories(user.gitHubUsername))
  .then(repos => getCommits(repos[0]))
  .then(commits => console.log('Commits', commits))
  .catch(err => console.log('Error', err.message));


console.log('After');
```

Async et wait

Nous allons réécrire le code ***promesse*** en code ***async await.***

💡L'idée est de transformer l'écriture de :

getUser(1).then(user => getRepositories(user.gitHubUsername))

en

const user = await getUser(1).

Il faut ensuite décorer le **await** dans une fonction asynchrone **async** !

Ainsi, le code précédent

```
getUser(1)
  .then(user => getRepositories(user.gitHubUsername))
  .then(repos => getCommits(repos[0]))
  .then(commits => console.log('Commits', commits))
```

va se réécrire de la façon suivante :

```
const user = await getUser(1);
const repos = await getRepositories(user.gitHubUsername);
const comits = await  getCommits(repos[0]);
console.log('Commits', commits);

```

Finalement, on décore une fonction avec async

```
async function fetchCommits(){
  const user = await getUser(1);
  const repos = await getRepositories(user.gitHubUsername);
  const comits = await  getCommits(repos[0]);
```

```
35.  console.log('Commits', commits);
36.}
37.fetchCommits();
```

En terme d'exécution on ne passe de la lig. 40 à 41 que si getUser est retournée et ainsi de suite.

🥷Pour être complet, on englobe le code dans un try catch :

```
31./**
32. * Fetches commits from GitHub for a user.
33. * @returns {Promise<void>} A promise that resolves when the commits
    are fetched.
34. */
35.async function fetchCommits() {
36.  try {
37.    const user = await getUserData(1);
38.    const repos = await getRepositories(user.gitHubUsername);
39.    const commits = await getCommits(repos[0]);
40.    console.log("Commits", commits);
41.  } catch (err) {
42.    console.log("Error", err.message);
43.  }
44.}
```

Autre code

This function sends a GET request to the GitHub API's commits endpoint for the specified user and repository. It then maps the response data to an array of

commit messages. Note that you'll need to install node-fetch with npm install node-fetch if you haven't already.

Please note that the GitHub API has rate limits. If you're making a lot of requests, you may need to authenticate your requests to increase your rate limit.

```
>npm init -y
>npm install node-fetch
```

 commits.mjs

```
1. import fetch from "node-fetch";
2. /**
3.  * Retrieves the commits of a repository.
4.  * @param {string} username - The GitHub username.
5.  * @param {string} repo - The repository name.
6.  * @returns {Promise<Array<string>>} A promise that resolves with an
    array of commit messages.
7.  */
8. async function getCommits(username, repo) {
9.   try {
10.    const response = await fetch(
11.      `https://api.github.com/repos/${username}/${repo}/commits`
12.    );
13.    if (!response.ok) {
14.      throw new Error(`HTTP error! status: ${response.status}`);
15.    }
16.    const data = await response.json();
```

```
    return data.map((commit) => {
      return commit.commit.message;
    });
  } catch (error) {
    console.error(`Error fetching commits for repo ${repo}:
    ${error.message}`);
    throw error;
  }
}
const data = await[8] getCommits("dupontdenis", "SFP-Lib");
console.table(data);
```

```
>node commits.mjs
```

En action avec fetch

[fetch + express]

En action avec express

GitHub offre une API pour rechercher les dépôts d'un utilisateur. Voici un exemple d'utilisation : https://github.com/dupontdenis/useAPIgit.git

[8] top-level await permet de se passer de l'encapsulation dans un async !

Annexes

9. package.json

Nous reviendrons plus en détail sur le fichier de configuration.

Nous pouvons nous amuser à modifier le fichier package.json en ajoutant un script

```
"scripts": {
    "start": "node index.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
```

Ainsi, vous pourrez ainsi lancer votre application avec

```
C:\Users\DD\Desktop\async-demo>node index.js
```

ou

```
C:\Users\DD\Desktop\async-demo>npm start
```

10. settled

On peut créer une promise qui est déjà résolue[9]. Pour cela, on peut utiliser l'API Promise.

La classe Promise dispose d'une méthode static resolve[10].

index.js

[9] Ceci est utile pour les fichiers de test où on veut simuler par exemple qu'une BD renvoie une promesse avec succès.

[10] Les méthodes statiques sont appelées directement avec le nom de la classe.

```
1. const promise = Promise.resolve( {id:1});
2. promise.then( id => console.log(id))
```

```
C:\Users\DD\Desktop\TDcallback>node index.js
{ id: 1 }
```

De façon similaire,on peut avoir une promesse qui est rejetée.

On renvoie alors un objet Error.

```
1. const promise = Promise.reject(new Error("pas de BD"));
2. promise.catch( error => console.log(`${error}`))
```

//

Sans avoir un code multithreading, on peut utiliser Promise.all pour simuler le parallélisme d'exécution.

L'idée sera de ne pas attendre le résultat d'une promesse pour lancer une seconde promesse.

```
1. const p1 = new Promise((resolve) => {
2.     setTimeout(() => {
3.       console.log('Async operation 1...');
4.       resolve("news from p1");
5.     }, 3000);
6.   });
7.
8.
9. const p2 = new Promise((resolve) => {
10.   setTimeout(() => {
11.     console.log('Async operation 2...');
```

```
12.      resolve("news from p2");
13.    }, 1000);
14.  });
15.
16.
17.  console.time('para');
18.  Promise.all([p1, p2])
19.   .then(result => {
20.       console.log(result);
21.       console.timeEnd('para');
22.    })
23.   .catch(err => console.log('Error', err.message));
24.
25. console.log("end");
```

callback hell

Regarder dans ce code la structure en arbre de Noël pour gérer les callback[11] :

```
1.  const readline = require("readline");
2.  const rl = readline.createInterface({
3.      input: process.stdin,
4.      output: process.stdout
5.  });
6.
7.  rl.question("What is your name ? ", function(name) {
8.      rl.question("Where do you live ? ", function(country) {
9.          console.log(`${name}, is a citizen of ${country}`);
```

[11] https://nodejs.org/en/knowledge/command-line/how-to-prompt-for-command-line-input/

```
10.         rl.close();
11.     });
12.});
13.
14.rl.on("close", function() {
15.     console.log("\nBYE BYE !!!");
16.     process.exit(0);
17.});
```