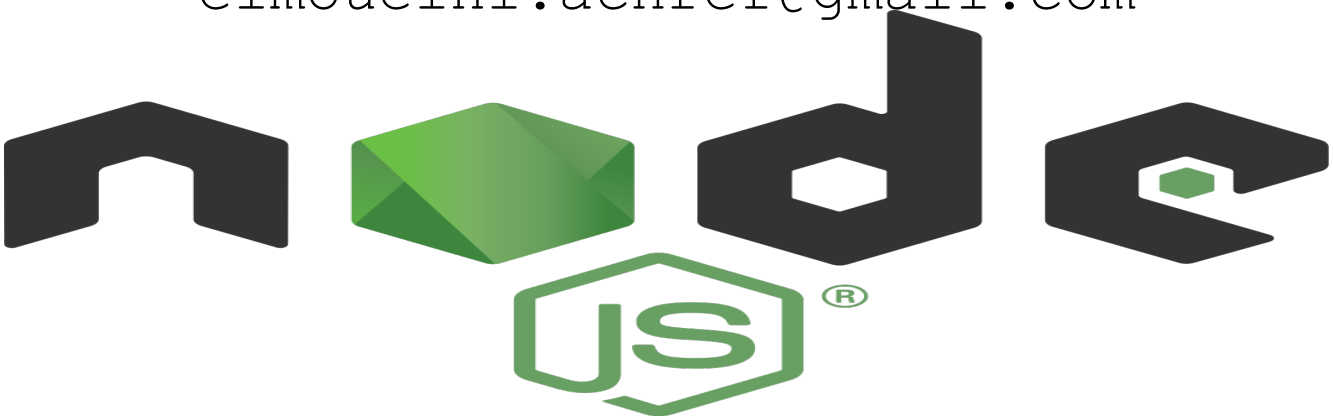
Node.js : modules et promesses

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com



Plan

1. CommonJS
   - `exports`
   - `require`
   - Importation multiple
   - Module ES6 ou ESModule
2. ESModule
   - `export`
   - `import`
   - `as`
   - `default`

Plan

Node.js

Trois types de modules

- **Core modules** : définis dans le noyau du **Node.js**. Pour les utiliser, il faut les importer.
- **Community modules** : proposés par la communauté **Node.js**. Pour les utiliser, il faut les installer et ensuite les importer.
- **Modules personnalisés** : pour les utiliser, il faut les exporter puis les importer.

Node.js

Trois types de modules

- **Core modules** : définis dans le noyau du **Node.js**. Pour les utiliser, il faut les importer.
- **Community modules** : proposés par la communauté **Node.js**. Pour les utiliser, il faut les installer et ensuite les importer.
- **Modules personnalisés** : pour les utiliser, il faut les exporter puis les importer.

Remarque

Dans ce chapitre, nous traitons les modules personnalisés.

Node.js

Module

Un fichier pouvant contenir des variables, constantes, fonctions, classes...

Node.js

Module

Un fichier pouvant contenir des variables, constantes, fonctions, classes...

Propriétés

- Il est possible d'utiliser des éléments définis dans un autre fichier : variable, constante, fonction, classe...
- Pour cela, il faut l'importer là où on a besoin de l'utiliser
- Pour importer un élément, il faut l'exporter dans le fichier source

Node.js

Étant donné le fichier `fonctions.js` dont le contenu est

```
const somme = (a = 0, b = 0) => {
    return a + b;
}

const produit = (a = 0, b = 1) => {
    return a * b;
}
```

Node.js

Pour exporter les deux fonctions `somme` et `produit` de `fonctions.js` : première syntaxe

```
const somme = (a = 0, b = 0) => {
    return a + b;
}

const produit = (a = 0, b = 1) => {
    return a * b;
}

module.exports = { somme, produit }
```

Node.js

Deuxième syntaxe

```
exports.somme = (a = 0, b = 0) => {
    return a + b;
}

exports.produit = (a = 0, b = 1) => {
    return a * b;
}
```

Node.js

Troisième syntaxe

```
module.exports = {
    somme: (a = 0, b = 0) => {
        return a + b;
    },

    produit: (a = 0, b = 1) => {
        return a * b;
    }
}
```

Node.js

Pour importer et utiliser une fonction

```
const { somme } = require('./fonctions');

console.log(somme(2, 5));
// affiche 7
```

Node.js

Pour importer et utiliser une fonction

```
const { somme } = require('./fonctions');

console.log(somme(2, 5));
// affiche 7
```

Pour importer plusieurs éléments

```
const { somme, produit } = require('./fonctions');

console.log(somme(2, 5));
// affiche 7

console.log(produit(2, 5));
// affiche 10
```

Node.js

Pour tout importer dans une constante `f`

```
const f = require('./fonctions');

console.log(f.somme(2, 5));
// affiche 7

console.log(f.produit(2, 5));
// affiche 10
```

Node.js

On peut aussi renommer les éléments exportés

```
const somme = (a = 0, b = 0) => {
    return a + b;
}

const produit = (a = 0, b = 1) => {
    return a * b;
}

module.exports = { addition: somme , produit }
```

Node.js

On peut aussi renommer les éléments exportés

```
const somme = (a = 0, b = 0) => {
    return a + b;
}

const produit = (a = 0, b = 1) => {
    return a * b;
}

module.exports = { addition: somme , produit }
```


Pour importer

```
const { addition } = require('./fonctions');

console.log(addition(2, 5));
// affiche 7
```

Node.js

`exports, require...`

Syntaxe native de **Node.js** définie dans **CommonJS** : système de gestion de module de **Node.js**.

Node.js

`exports`, `require`...

Syntaxe native de **Node.js** définie dans **CommonJS** : système de gestion de module de **Node.js**.

Question

Peut-on utiliser la syntaxe **ES6** (`import`, `export`...) avec **Node.js** ?

Node.js

`exports`, `require`...

Syntaxe native de **Node.js** définie dans **CommonJS** : système de gestion de module de **Node.js**.

Question

Peut-on utiliser la syntaxe **ES6** (`import`, `export`...) avec **Node.js** ?

Réponse

Oui, depuis **Node.js 13.2.0**, avec un peu de configuration pour changer le système natif de **Node.js**

Node.js

Démarche

- Privilégier l'extension `.mjs` pour les modules **ECMAScript**
- Modifier la configuration par défaut dans `package.json` : ajouter `"type": "module"`

Node.js

Ajoutons `"type": "module"` **pour remplacer** `CommonJS` **dans** `package.json`

```
{
  "name": "cours-nodejs",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "type": "module",
  "dependencies": {
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Node.js

Pour exporter les deux fonctions `somme` **et** `produit` **de** `fonctions.js`

```
export function somme(a, b) {
    return a + b;
}

export function produit(a, b) {
    return a * b;
}
```

Node.js

Pour exporter les deux fonctions `somme` **et** `produit` **de** `fonctions.js`

```
export function somme(a, b) {
    return a + b;
}

export function produit(a, b) {
    return a * b;
}
```

Ou aussi

```
function somme(a, b) {
    return a + b;
}

function produit(a, b) {
    return a * b;
}
export { somme, produit };
```

Node.js

Pour importer et utiliser une fonction

```
import { somme } from './fonctions.js';

console.log(somme(2, 5));
// affiche 7
```

Node.js

Pour importer et utiliser une fonction

```
import { somme } from './fonctions.js';

console.log(somme(2, 5));
// affiche 7
```

Pour importer plusieurs éléments

```
import { somme, produit } from './fonctions.js';

console.log(somme(2, 5));
// affiche 7

console.log(produit(2, 5));
// affiche 10
```

Node.js

On peut aussi utiliser des alias

```
import { somme as s, produit as p } from './fonctions';

console.log(s(2, 5));
// affiche 7

console.log(p(2, 5));
// affiche 10
```

Node.js

On peut aussi utiliser des alias

```
import { somme as s, produit as p } from './fonctions';

console.log(s(2, 5));
// affiche 7

console.log(p(2, 5));
// affiche 10
```

Ou aussi

```
import  * as f from './fonctions';

console.log(f.somme(2, 5));
// affiche 7

console.log(f.produit(2, 5));
// affiche 10
```

Node.js

Les alias peuvent être attribués à l'exportation

```
function somme(a, b) {
    return a + b;
}

function produit(a, b) {
    return a * b;
}
export { produit as p, somme as s } ;
```

Node.js

Les alias peuvent être attribués à l'exportation

```
function somme(a, b) {
    return a + b;
}

function produit(a, b) {
    return a * b;
}
export { produit as p, somme as s } ;
```

Pour importer

```
import * as f from './fonctions';

console.log(f.s(2, 5));
// affiche 7

console.log(f.p(2, 5));
// affiche 10
```

Node.js

On peut aussi utiliser le `export default` (un seul par fichier)

```
export default function somme(a, b) {
    return a + b;
}

export function produit(a, b) {
    return a * b;
}
```

Node.js

On peut aussi utiliser le `export default` (un seul par fichier)

```
export default function somme(a, b) {
    return a + b;
}

export function produit(a, b) {
    return a * b;
}
```

Pour importer, pas besoin de { } pour les éléments exportés par défaut

```
import somme from './fonctions.js';
import { produit } from './fonctions.js';

console.log(somme(2, 5));
// affiche 7

console.log(produit(2, 5));
// affiche 10
```

Node.js

Attention, ici on a importé `somme` avec deux alias différents

```
import s from './fonctions.js';
import produit from './fonctions.js';

console.log(s(2, 5));
// affiche 7

console.log(produit(2, 5));
// affiche 7
```

Node.js

Question

Peut-on tout exporter par défaut ?

Node.js

Question

Peut-on tout exporter par défaut ?

Réponse

Non

Node.js

Ce qu'on peut exporter par défaut

- Fonction
- Classe
- Objet
- Valeur primitive

Ce qu'on ne peut exporter

- Plusieurs éléments à la fois

```
export default function f1() {}
export default function f2() {}
```

- Des constantes ou des variables multiples sous une même déclaration

```
export default a, b;
```

Node.js

L'écriture suivante génère une erreur (même en remplaçant `const` par `var` ou `let`)

```
export default const X = 5
```

Node.js

L'écriture suivante génère une erreur (même en remplaçant `const` par `var` ou `let`)

```
export default const X = 5
```

Elle doit être remplacée par les deux instructions suivantes

```
const X = 5
export default X
```

Node.js

L'écriture suivante génère une erreur (même en remplaçant `const` par `var` ou `let`)

```
export default const X = 5
```


Elle doit être remplacée par les deux instructions suivantes

```
const X = 5
export default X
```

Ou directement

```
export default 5
```

Node.js

L'écriture suivante aussi génère une erreur

```
export default const f = () => console.log('function');
```

Node.js

L'écriture suivante aussi génère une erreur

```
export default const f = () => console.log('function');
```

Et doit être remplacée par les deux instructions suivantes

```
const f = () => console.log('function');
export default f
```

Node.js

Explication

- `export default` nécessite une valeur ou une référence à une valeur déjà déclarée.
- `export default const X = 5` n'est pas valide parce qu'elle essaie de combiner la déclaration d'une variable avec l'exportation, ce que la syntaxe **JavaScript** interdit.

Node.js

Question

Peut-on utiliser à la fois **CommonJS** et **ESModule** dans le même projet ?

Node.js

Question

Peut-on utiliser à la fois **CommonJS** et **ESModule** dans le même projet ?

Réponse

ESModule et **CommonJS** sont mutuellement exclusifs, nous ne pouvons donc pas utiliser les modules **ES6** dans **CommonJS**.

Node.js

Remarque

Les variables **CommonJS** comme `__filename` et `__dirname` ne sont pas accessibles dans les modules **ES6**.

Node.js

Remarque

Les variables **CommonJS** comme `__filename` et `__dirname` ne sont pas accessibles dans les modules **ES6**.

Depuis Node.js 21.2.0, on peut utiliser

- `import.meta.filename` pour récupérer le chemin absolu vers le fichier courant
- `import.meta.dirname` pour récupérer le chemin absolu vers le répertoire courant

Node.js

Considérons les deux fonctions suivantes

```
function somme(a, b) {
  return a + b;
}
function produit(a, b) {
  return a * b;
}
```

Node.js

Considérons les deux fonctions suivantes

```
function somme(a, b) {
  return a + b;
}
function produit(a, b) {
  return a * b;
}
```


Exercice

- Écrire une fonction `operation` qui prend trois paramètres : `a`, `b` et `fonction`
- Si `fonction == somme`, alors `operation` retourne le résultat de la somme de `a` et `b`
- Sinon `opration` retourne le résultat de la multiplication de `a` par `b`

Node.js

Une première solution

```
function operation(a, b, fonction) {
    if (fonction == somme) {
        return somme(a, b)
    }
    return produit(a, b)
}

console.log(operation(2, 3, produit));
console.log(operation(2, 3, somme));
```

Node.js

Une première solution

```
function operation(a, b, fonction) {
    if (fonction == somme) {
        return somme(a, b)
    }
    return produit(a, b)
}

console.log(operation(2, 3, produit));
console.log(operation(2, 3, somme));
```


Remarque

Une solution plus générique consisterait à utiliser les **callbacks**.

Node.js

Fonction de retour (callback)

- fonction appelée comme un paramètre d'une deuxième fonction
- très utilisée en **JavaScript** (**jQuery** et **Node.js**) avec les fonctions asynchrones

Node.js

Une deuxième solution avec les callbacks pour la fonction `operation`

```
function operation(a, b, fonction) {
  console.log (fonction(a, b));
}

// appeler la fonction opération
operation (3, 5, somme);
// affiche 8
```

Node.js

Exercice

- Modifier la fonction `operation` pour qu'elle puisse accepter deux fonctions callback ensuite retourner le résultat de la composition des deux.
- Par exemple :
  `operation (2, 3 , 6, somme, produit)` retourne 30 = (2 + 3) * 6

Node.js

Exercice

Écrire une fonction `division` qui

- prend deux paramètres `a` et `b`
- retourne le résultat de la division de `a` par `b` si `b != 0`
- lève une exception si `b == 0`

Node.js

Une solution possible

```
const division = (a, b) => {
    if (b === 0) {
        throw "Problème de division par zéro";
    }
    return a / b;
}
```

Node.js

Une solution possible

```
const division = (a, b) => {
    if (b === 0) {
        throw "Problème de division par zéro";
    }
    return a / b;
}
```


L'appel de la fonction

```
console.log(division(10, 2));
// affiche 5

console.log(division(10, 0));
// affiche Problème de division par zéro
```

Node.js

Remarque

- En **Node.js**, on utilise souvent les callback surtout pour les traitements asynchrones.
- Les fonctions callbacks en **Node.js** ont une signature similaire.

Réécriture de la fonction `division` avec les callback

```
const division = (a, b, callback) => {
    if (b !== 0) {
        return callback(null, a / b);
    }
    return callback('Problème de division par zéro', null);
};

const a = 10, b = 0;

division(a, b, (err, result) => {
    if (err) {
        console.error("erreur : " + err);
    } else {
        console.log(`${a} / ${b} = ${result}`);
    }
});
// affiche erreur : Problème de division par zéro
```

Réécriture de la fonction `division` avec les callback

```
const division = (a, b, callback) => {
    if (b !== 0) {
        return callback(null, a / b);
    }
    return callback('Problème de division par zéro', null);
};

const a = 10, b = 0;

division(a, b, (err, result) => {
    if (err) {
        console.error("erreur : " + err);
    } else {
        console.log(`${a} / ${b} = ${result}`);
    }
});
// affiche erreur : Problème de division par zéro
```

Affecter la valeur 2 à `b` et vérifier que le bon résultat s'affiche.

Node.js

Conventions

Les fonctions callback prennent au moins deux paramètres :

- un premier paramètre `err` qui reste vide si la fonction a bien été exécutée, sinon il contient le contenu du message d'erreur.
- un deuxième paramètre `result` qui contient le résultat si la fonction n'a pas détecter d'erreurs.

Node.js

Exercice

- Écrire une fonction qui permet de
  - vérifier si un nombre est présent dans un tableau
  - retourner dans ce cas sa position

Node.js

Solution (avec callback)

```
const searchElement = (data, callback) => {
    for (let i = 0; i < data.tableau.length; i++) {
        if (data.tableau[i] == data.filtre) {
            return callback(null, i);
        }
    }
    return callback(`${data.filtre}  n'est pas dans tableau`);
};

const tab = [1, 3, 6, 8, 9];
const element = 6;
const data = { tableau: tab, filtre: element };

searchElement(data, (err, result) => {
    if (err) {
        console.error("erreur :" + err);
    } else {
        console.log(`${element} est dans le tableau à la position ${
          result}`);
    }
});
```

Node.js

Ou

```
const searchElement = function (tableau, elt, callback) {
    for (let i = 0; i < tableau.length; i++) {
        if (tableau[i] == elt) {
            return callback(null, i);
        }
    }
    return callback(`${elt}  n'est pas dans tableau`);
};

const tab = [1, 3, 6, 8, 9];
const element = 6;
const data = { tableau: tab, filtre: element };

searchElement(tab, element, (err, result) => {
    if (err) {
        console.error("erreur :" + err);
    } else {
        console.log(`${element} est dans le tableau à la position ${
          result}`);
    }
});
```

Node.js

Exercice 2

En utilisant les fonctions callback, écrire une fonction qui permet de déterminer le nombre d'occurrence d'une sous-chaîne de caractère ch dans une chaîne de caractère str.

- $ch = ab$
- $str = abbbaaaabaaabb$
- la fonction retourne 3.

Node.js

Promesse

- Introduite dans **ES6**
- Utilisée souvent pour réaliser des traitements sur un résultat suite à une opération asynchrone
- Disposant d'une première méthode `then()` permettant de traiter le résultat une fois l'opération accomplie
- Disposant d'une deuxième méthode `catch()` exécutée en cas d'échec de l'opération
- Comme les fonctions : composée de deux parties : déclaration et utilisation

Considérons la déclaration suivante d'une promesse

```
var test = true;
var promesse = new Promise((resolve, reject) => {
    if (test)
        resolve();
    else
        reject();
});
```

Considérons la déclaration suivante d'une promesse

```
var test = true;
var promesse = new Promise((resolve, reject) => {
    if (test)
        resolve();
    else
        reject();
});
```

Dans la partie utilisation, on doit indiquer ce qu'il faut faire dans les deux cas : réussite (`resolve`) ou échec (`reject`)

```
promesse
    .then(() => console.log("test réussi") )
    .catch(() => console.log("erreur") );
// affiche test réussi car test = true
```

Considérons la déclaration suivante d'une promesse

```
var test = true;
var promesse = new Promise((resolve, reject) => {
    if (test)
        resolve();
    else
        reject();
});
```

Dans la partie utilisation, on doit indiquer ce qu'il faut faire dans les deux cas : réussite (`resolve`) ou échec (`reject`)

```
promesse
    .then(() => console.log("test réussi") )
    .catch(() => console.log("erreur") );
// affiche test réussi car test = true
```

`catch()` n'est pas obligatoire, mais fortement recommandé.

Node.js

Une promesse peut être déclarée avec une fonction fléchée

```
var test = true;
var promesse = () => {
    return new Promise((resolve, reject) => {
        if (test)
            resolve();
        else
            reject();
    });
};
```

Node.js

Une promesse peut être déclarée avec une fonction fléchée

```
var test = true;
var promesse = () => {
    return new Promise((resolve, reject) => {
        if (test)
            resolve();
        else
            reject();
    });
};
```


Pour l'utilisation, il faut appeler promesse comme une fonction

```
promesse()
    .then(() => console.log("test réussi"))
    .catch(() => console.log("erreur"));
```

Node.js

Remarque

Une promesse peut recevoir des paramètres et retourner un résultat

Node.js

Exemple : déclaration

```
var division = (a, b) => {
    return new Promise((resolve, reject) => {
        if (b != 0)
            resolve(a / b);
        else
            reject("erreur : division par zéro");
    });
};
```

Node.js

Exemple : déclaration

```
var division = (a, b) => {
    return new Promise((resolve, reject) => {
        if (b != 0)
            resolve(a / b);
        else
            reject("erreur : division par zéro");
    });
};
```

L'utilisation

```
division(10, 2)
    .then((res) => console.log("résultat : " + res))
    .catch((error) => console.log(error));
// affiche résultat : 5

division(5, 0)
    .then((res) => console.log("résultat : " + res))
    .catch((error) => console.log(error));
// affiche erreur : division par zéro
```

Node.js

Remarque : les promesses s'exécutent d'une manière asynchrone

```
var division = (a, b) => {
  return new Promise((resolve, reject) => {
    if(b != 0)
      resolve(a / b);
    else
      reject("erreur : division par zéro");
  });
};

division(10, 2)
    .then((res) => console.log("résultat : " + res))
    .catch((error) => console.log(error));

division(5, 0)
    .then((res) => console.log("résultat : " + res))
    .catch((error) => console.log(error));

console.log("fin");
```

Node.js

Remarque : les promesses s'exécutent d'une manière asynchrone

```
var division = (a, b) => {
  return new Promise((resolve, reject) => {
    if(b != 0)
      resolve(a / b);
    else
      reject("erreur : division par zéro");
  });
};

division(10, 2)
    .then((res) => console.log("résultat : " + res))
    .catch((error) => console.log(error));

division(5, 0)
    .then((res) => console.log("résultat : " + res))
    .catch((error) => console.log(error));

console.log("fin");
```

Le résultat est

```
fin
résultat : 5
erreur : division par zéro
```

Node.js

finally

- Introduit dans **ES2018**
- Prenant comme paramètre une fonction callback appelée lorsque l'exécution de la promesse est terminée : résolue ou rejetée.

Node.js

Remarque : les promesses s'exécutent d'une manière asynchrone

```
division(10, 2)
    .then((res) => console.log("résultat : " + res))
    .catch((error) => console.log(error))
    .finally(() => console.log('finally'));

division(5, 0)
    .then((res) => console.log("résultat : " + res))
    .catch((error) => console.log(error))
    .finally(() => console.log('finally'));
```


Le résultat est

```
résultat : 5
erreur : division par zéro
finally
finally
```

Node.js

Le mot-clé `async` [ES2018]

Il permet de transformer une fonction en promesse

Node.js

Le mot-clé `async` [ES2018]

Il permet de transformer une fonction en promesse

Considérons la fonction `somme()` suivante

```
let somme = (a, b) => a + b;
```

Node.js

Le mot-clé `async` [ES2018]

Il permet de transformer une fonction en promesse

Considérons la fonction `somme()` suivante

```
let somme = (a, b) => a + b;
```

Le résultat est

```
console.log(somme(2, 3));
// affiche 5
```

Node.js

Pour transformer la fonction `somme()` en promesse, on ajoute le mot-clé `async` à sa déclaration

```
let somme = async (a, b) => a + b;
```

Maintenant, on peut l'utiliser comme une promesse

```
somme(2, 3).then(result => console.log(result));
// affiche 5
```

Node.js

Pour transformer la fonction `somme()` en promesse, on ajoute le mot-clé `async` à sa déclaration

```
let somme = async (a, b) => a + b;
```

Maintenant, on peut l'utiliser comme une promesse

```
somme(2, 3).then(result => console.log(result));
// affiche 5
```

Appeler `somme()` comme une simple fonction JavaScript n'affiche pas le résultat

```
console.log(somme(2, 3));
// affiche Promise { 5 }
```

Node.js

Le mot-clé `await`

- utilisable seulement dans des environnements asynchrones
- permettant d'interrompre l'exécution de la fonction asynchrone et attendre la résolution de la promesse

Node.js

Considérons la promesse `somme()` qui attend 2 secondes pour retourner un résultat

```
let somme = (a, b) => {
    return new Promise((resolve) => {
        setTimeout(() => { resolve(a + b) }, 2000);
    });
};
```

Node.js

Considérons la promesse `somme()` qui attend 2 secondes pour retourner un résultat

```
let somme = (a, b) => {
    return new Promise((resolve) => {
        setTimeout(() => { resolve(a + b) }, 2000);
    });
};
```

On veut implémenter une promesse `sommeCarre()` qui utilise la promesse `somme()`

```
let sommeCarre = async (a, b) => {
    let s = 0;
    somme(a, b).then(result => s = result);
    let result = Math.pow(s, 2);
    return result;
};
```

Node.js

Pour utiliser la promesse avec les valeurs 2 et 3

```
sommeCarre(2, 3)
    .then(result => console.log(result));
// affiche 0
```

Node.js

Pour utiliser la promesse avec les valeurs 2 **et** 3

```
sommeCarre(2, 3)
    .then(result => console.log(result));
// affiche 0
```


Explication

On n'a pas obtenu le résultat de la somme lorsqu'on a calculé le carré

Solution, utiliser `await` pour attendre la fin de la première promesse

```
let sommeCarre = async (a, b) => {
    let s = 0;
    await somme(a, b).then(result => s = result);
    let result = Math.pow(s, 2);
    return result;
};
```

Solution, utiliser `await` pour attendre la fin de la première promesse

```
let sommeCarre = async (a, b) => {
    let s = 0;
    await somme(a, b).then(result => s = result);
    let result = Math.pow(s, 2);
    return result;
};
```

Ou

```
let sommeCarre = async (a, b) => {
    let s = await somme(a, b);
    let result = Math.pow(s, 2);
    return result;
};
```

Solution, utiliser `await` pour attendre la fin de la première promesse

```
let sommeCarre = async (a, b) => {
    let s = 0;
    await somme(a, b).then(result => s = result);
    let result = Math.pow(s, 2);
    return result;
};
```

Ou

```
let sommeCarre = async (a, b) => {
    let s = await somme(a, b);
    let result = Math.pow(s, 2);
    return result;
};
```

Si on teste maintenant

```
sommeCarre(2, 3).then(result => console.log(result));
// affiche 25
```

Node.js

Remarque

`await` doit toujours être utilisé dans une fonction asynchrone.

Node.js

Considérons la promesse `carre()` acceptant un seul paramètre

```
var carre = (a) => {
    return new Promise((resolve) => {
        resolve(a ** 2);
    });
};
```

Node.js

Considérons la promesse `carre()` acceptant un seul paramètre

```
var carre = (a) => {
    return new Promise((resolve) => {
        resolve(a ** 2);
    });
};
```

On peut enchaîner les `then`

```
carre(3)
    .then((res) => res + 1) //  .then((9) => 9 + 1)
    .then(carre)  // carre (10)
    .then((res) => console.log("résultat : " + res)); // affiche 100
```

Node.js

Considérons la promesse `carre()` acceptant un seul paramètre

```
var carre = (a) => {
    return new Promise((resolve) => {
        resolve(a ** 2);
    });
};
```

On peut enchaîner les `then`

```
carre(3)
    .then((res) => res + 1) //  .then((9) => 9 + 1)
    .then(carre)  // carre (10)
    .then((res) => console.log("résultat : " + res)); // affiche 100
```

Remarques

Si une valeur est retournée, la méthode `then()` suivante est appelée avec cette valeur.

Node.js

Le mot-clé `all` retourne une promesse qui

- est résolue lorsque l'ensemble des promesses contenues dans l'itérable passé en paramètre ont été résolues
- échoue avec la raison de la première promesse qui échoue au sein de l'itérable

Node.js

Le mot-clé `all` retourne une promesse qui

- est résolue lorsque l'ensemble des promesses contenues dans l'itérable passé en paramètre ont été résolues
- échoue avec la raison de la première promesse qui échoue au sein de l'itérable

Considérons les fonctions `somme()` et `produit` suivantes

```
let somme = async (a, b) => a + b;
let produit = async (a, b) => a * b;
```

Node.js

Pour attendre la fin des deux promesses `somme` et `produit` pour faire la suite (afficher les résultats)

```
Promise
    .all([somme(2, 3), produit(2, 3)])
    .then((values) => console.log(values));

// affiche [ 5, 6 ]
```

Node.js

Dans le cas où une promesse échoue

```
Promise
    .all([somme(2, 3), division(10, 0), produit(2, 3)])
    .then((values) => console.log(values))
    .catch(err => console.log(err))

// affiche erreur : division par zéro
```

Node.js

Callbacks : avantages

- Simplicité de compréhension (fonction)
- Contrôle précis et facile de l'exécution

Promesses : avantages

- Lisibilité améliorée
- Gestion simplifiée des erreurs (avec un block `catch`)
- Chaînage plus facile.

Callbacks : inconvénients

- Nesting excessif (Callback Hell) : à cause des opérations asynchrones imbriquées
- Gestion d'erreur plus complexe

Promesses : inconvénients

- Compatibilité limitée avec certains navigateurs et modules
- Construction plus complexe

Node.js

Ressources utiles

- https://nodejs.org/docs
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
- https://www.w3schools.com/js/js_callback.asp