1. Objectifs

Nous allons revenir sur la communication entre fichiers dans le navigateur avant de découvrir les modules dans node.

2. Navigateur

Nous allons tester le code suivant dans un navigateur pour montrer que les scripts peuvent communiquer entre eux en passant par l'objet window.

Créer un fichier  index.html

 index.html

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>...</head>
4. <body>
5.     <script>
6.         var i = 2;
7.     </script>
8.     <script src="test.js"></script>
9. </body>
10.  </html>
```

Créer un fichier  test.js

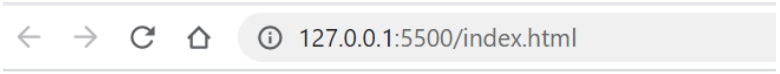
 test.js

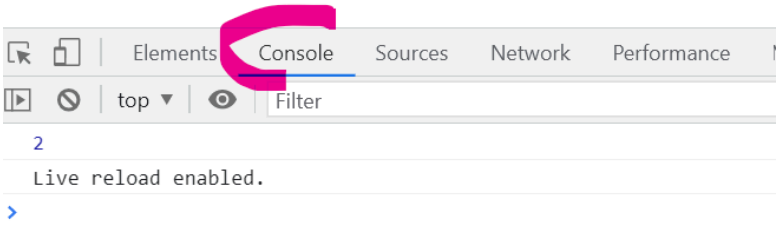
```
1. console.log(i)
```

Voici une copie d'écran montrant la structure et le code.

Lancez l'extension **live-server** dans VS code et ouvrez la console de votre navigateur.

La valeur 2 s'affiche. Le fichier test.js a eu accès à la variable *i* définie dans le script du fichier index.html.





3. NodeJS

Après avoir observé la visibilité des variables dans les navigateurs, nous allons montrer maintenant que la portée des variables dans nodeJS est locale à chaque fichier.

Nous savons que l'objet global dans node est *global*, nous allons donc dans un fichier  test.js remplacer *window* par *global*.

Testez le code suivant dans un fichier test.js

 test.js

```
1. let test="superDupont";
2. global.console.log(test);
3. global.console.log(global.test);
```

`>node test.js`

L'exécution du code affiche

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

superDupont

undefined

Le résultat ***undefined*** du second affichage montre que la variable test n'appartient pas à *global*, c'est pourquoi global.console.log(global.test) affiche *undefined*.

🤖**Mais alors comment communiquer et rendre accessible des variables ou des fonctions à travers les fichiers ?**

🪛Pour répondre à cette question, nous allons étudier comment les fichiers communiquent entre eux. Nous verrons que la communication est réalisée par l'intermédiaire des modules.

4. Utilisation de l'exportation des modules

Nous allons au travers d'un exemple comprendre le mécanisme des modules.

🔪Commençons par créer un fichier test.js dans un répertoire (ici Test)

 test.js

```
1. const urgence = "absolue"
2. function affiche(message){
3.     console.log(`${urgence} : message`);
4. }
```

Dans votre terminal lancez `node`[1]

```
C:\Users\DD\Test>node
Welcome to Node.js v20.07.0.
Type ".help" for more information.
> require('./test.js')
{}
```

🆘l'affichage {}, montre simplement que le fichier  test.js n'exporte rien lorsque l'on fait appel à lui par l'appel à *require('./test.js')*

🔪Nous allons maintenant exporter une fonction à partir du fichier  test.js. Modifiez le fichier avec :

 test.js

```
1. const urgence = "absolue"
2. function affiche(message){
3.     console.log(`${urgence} : message`);
4. }
5.
```

[1] REPL(Read-Eval-Print-Loop)

6. 📤**module.exports.affiche = affiche;**

Dans votre terminal lancer la commande `node`

```
C:\Users\DD\Test>node
Welcome to Node.js v20.07.0.
Type ".help" for more information.
> require('./test.js')
{ affiche: [Function: affiche] }
> require('./test.js').affiche
[Function: affiche]
```

Lorsque l'on exporte la fonction *affiche*, l'affichage précédent sans export vaut maintenant [Function: affiche].

Ainsi, l'utilisation de module.exports📤 permet de mettre à disposition du code en dehors du module test.js.

🥷Nous allons tenter de comprendre plus en détails la magie des mots clé *exports* et *require*.

5. La structure des modules

Soit le code test.js

test.js:

1. const message = 📥require('./hello.js');
2. message.affiche('module');

🪛Précisons tout d'abord pourquoi il est important de déclarer une constante et non une variable en ligne 1.

🆘Explication :

const interdit toute redéfinition de message, ainsi votre code sera protégé de toute tentative de redéfinition de votre objet affiche.

🪛Nous allons maintenant créer et mettre à disposition deux modules.

6. Création de modules

On veut mettre à disposition d'une application deux fonctions, une fonction donne la somme géométrique et la seconde calcule le volume d'une sphère.

Voici le corps de ces deux fonctions :

Somme géométrique : a + ax + ax2 + . . . + axn − 1
1. function calculate(a, x, n) { 2. if(x === 1) return a*n; 3. return a*(1 - Math.pow(x, n))/(1 - x); 4. }

Volume d'une sphère : $4/3\pi r^3$
1. function calculate(r) { 2. return 4/3*Math.PI*Math.pow(r, 3); 3. }

🪛Nous allons créer les structures (fichiers) et le code pour pouvoir utiliser les deux fonctions de calcul dans un même fichier.

Commençons par écrire un fichier geometricSum.js

geometricSum.js

```
function calculate(a, x, n) {
    if (x === 1) return a * n;
    return a * (1 - Math.pow(x, n)) / (1 - x);
}
📤module.exports = calculate;
```

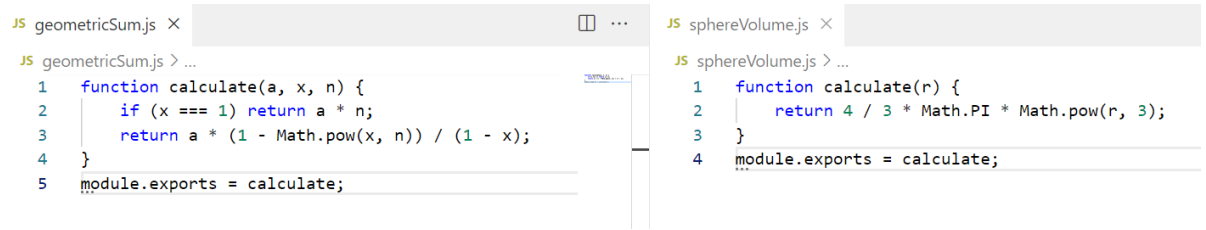
Puis écrire un fichier sphereVolume.js

sphereVolume.js

```
function calculate(r) {
    return 4 / 3 * Math.PI * Math.pow(r, 3);
}
📤module.exports = calculate;
```

🥷Notez que ces deux fonctions **ont le même nom** *calculate!*[2]

Voici les deux fichiers vus de notre éditeur VS code.



🪛Dans un fichier test.js, importez les modules geometricSum et sphereVolume.

test.js

[2] Deux équipes pourraient donner le même nom à deux fonctions !

```
1. const geometricSum = 📥require('./geometricSum.js');
2. const sphereVolume = 📥require('./sphereVolume.js');
3.
4. console.log(geometricSum(1, 2, 5));
5. console.log(sphereVolume(2));
```

Testez le code

```
>node test.js
```

```
31
33.510321638291124
```

🪛Modifier le code de  test.js

 test.js

```
const a = require("./geometricSum.js");
const b = require("./sphereVolume.js");

console.log(a(1, 2, 5));
console.log(b(2));
```

Testez le code

```
>node test.js
```

```
31
33.510321638291124
```

Considérons une nouvelle écriture pour les deux fonctions.

👹Commençons par écrire un fichier  geometricSum.js

 geometricSum.js

```
6. function calculate(a, x, n) {
7.     if (x === 1) return a * n;
8.     return a * (1 - Math.pow(x, n)) / (1 - x);
9. }
10.  module.exports = {
11.        calculate
12.  }
```

Puis écrire un fichier sphereVolume.js

sphereVolume.js

```
5. function calculate(r) {
6.     return 4 / 3 * Math.PI * Math.pow(r, 3);
7. }
8. module.exports ={
9.  calculate
10.  }
```

Nous pouvons alors écrire test.js

test.js

```
1. const { calculate: a } = require("./geometricSum.js");
2. const { calculate: b } = require("./sphereVolume.js");
3.
4. console.log(a);
5. console.log(b(2));
```

Nous ne pourrions pas écrire

```
1. const { calculate1 } = require("./geometricSum.js");
2. const { calculate2 } = require("./sphereVolume.js");
```

🪛Nous allons créer maintenant notre propre bibliothèque d'outils en regroupant un ensemble de fonctions utiles dans un même fichier.

7. Création d'une bibliothèque !

🪛Nous allons maintenant regrouper trois fonctions dans un seul fichier. Ce fichier devient une véritable bibliothèque de fonctions utiles pour la création d'applications.

Créer un fichier  util.js regroupant trois fonctions :

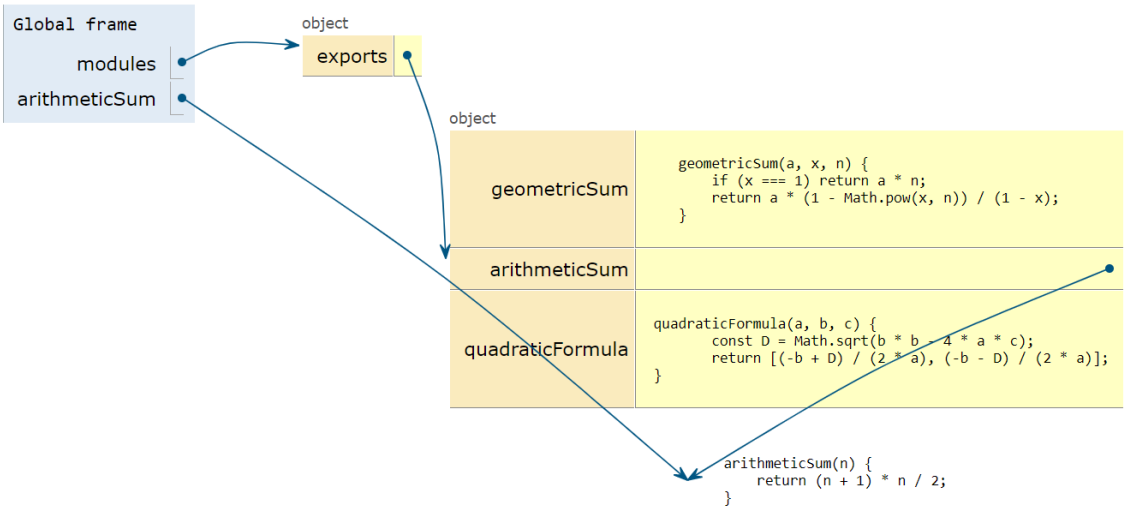
1. geometricSum
2. arithmeticSum
3. quadraticFormula

 util.js

```
module.exports = {
    geometricSum(a, x, n) {
        if (x === 1) return a * n;
        return a * (1 - Math.pow(x, n)) / (1 - x);
    },
    arithmeticSum(n) {
        return (n + 1) * n / 2;
    },
    quadraticFormula(a, b, c) {
          const D = Math.sqrt(b * b - 4 * a * c);
          return [(-b + D) / (2 * a), (-b - D) / (2 * a)];
    },
};
```

Lig. 1 : On exporte un objet.

Donner le code pour utiliser la fonction *arithmeticSum* dans un fichier app.js.



---

correction

Nous pouvons utiliser la destructuration pour n'importer que la fonction arithmeticSum.

Voici le code de app.js

app.js

```
1. const { arithmeticSum } = require('./util');
2. console.log( arithmeticSum(3) )
```

Autres écritures

On pourra utiliser une écriture plus compacte du module utils.js.

utils.js

```
1. exports.geometricSum = function (a, x, n) {
2.     if (x === 1) return a * n;
3.     return a * (1 - Math.pow(x, n)) / (1 - x);
4. };
5. exports.arithmeticSum = function (n) {
6.     return (n + 1) * n / 2;
7. };
8. exports.quadraticFormula = function (a, b, c) {
9.     const D = Math.sqrt(b * b - 4 * a * c);
10.       return [(-b + D) / (2 * a), (-b - D) / (2 * a)];
11.   };
```

On pourra ecrire dans app.js

app.js

```
1. const util = require('./utils');
2. console.log( util.arithmeticSum(3) );
```

🥷On peut également écrire en utilisant la destructuration :

app.js

```
1. const {arithmeticSum}[3] = require('./utils');
2. console.log( arithmeticSum(3) )
```

[3] Voir https://dupontes6.blogspot.com/p/decomposition.html