Modules JavaScript

Philippe Genoud
Philippe.Genoud@univ-grenoble-alpes.fr



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Contexte

- Aux débuts de JavaScript
  - programmes assez petits réalisant des tâches isolées uniquement là où l'interactivité était nécessaire
- JavaScript 'moderne'
  - de simples programmes on est passé à des applications complètes au code complexe et volumineux
  - exécution de programmes en dehors du contexte d'un navigateur (Node.js)

→ besoins d'organiser le code pour éviter les éventuels conflits de redéfinition de variables et fonctions entre différents scripts

  - découpage en modules
  - chargement des modules à la demande

Conflits possibles entre scripts

- Exemple illustratif

index.html > html

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initi
    <title>Pb Portée</title>
    <script src="librairie1.js"></script>
    <script src="librairie2.js"></script>
    <style>
        body {
            font-family: Verdana, sans-serif;
        }
    </style>
</head>

<body>
    <h1>Test redéfinition de variables et fonctions</h1>
    <p>
        regardez le comportement des scripts sur la console
    </p>
    <script src="scriptPrincipal.js"></script>
</body>

</html>
```

librairie1.js > ...

```js
var  a = 10;

function afficher1() {
    console.log(`afficher de librairie1.js a : ${a} ${typeof a}`);
}

function incrementerA() {
    console.log("incrementerA de librairie1");
    a = a + 1;
}
```

librairie2.js > ...

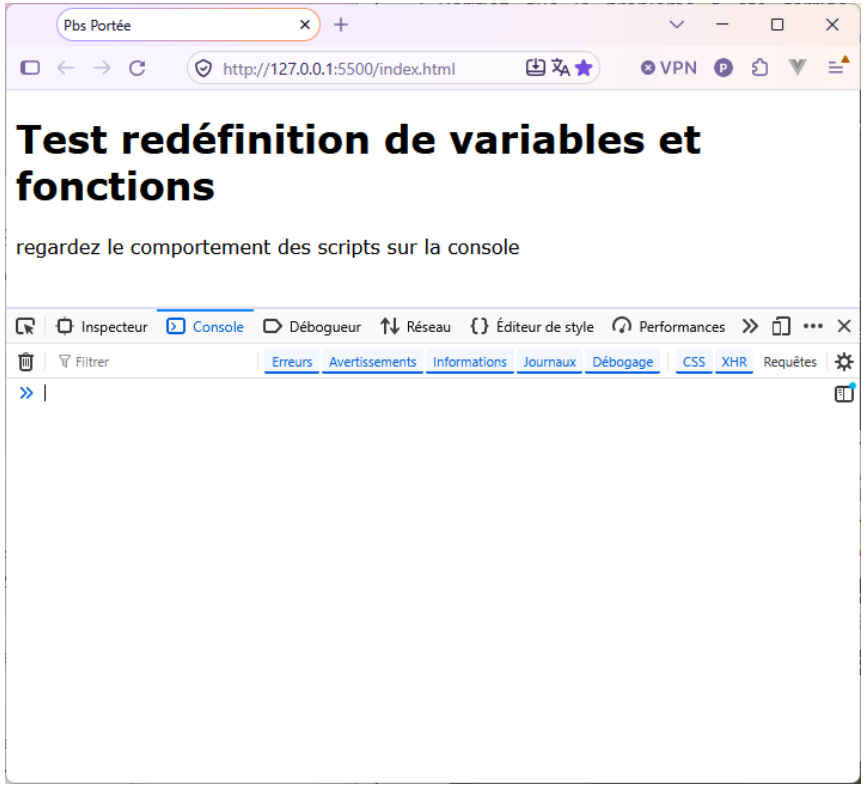
```js
var a = "10";

function afficher2() {
    console.log(`afficher de librairie2.js a : ${a} ${typeof a}`);
}

function incrementerA() {
    console.log("incrementerA de librairie2");
    a = a + 1;
}
```

scriptPrincipal.js

```js
afficher1();
afficher2();
incrementerA();
afficher1();
afficher2();
```



Qu'affiche le script principal ?

Conflits possibles entre scripts

- Exemple illustratif

index.html > html

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initi
    <title>Pb Portée</title>
    <script src="librairie1.js"></script>
    <script src="librairie2.js"></script>
    <style>
        body {
            font-family: Verdana, sans-serif;
        }
    </style>
</head>

<body>
    <h1>Test redéfinition de variables et fonctions</h1>
    <p>
        regardez le comportement des scripts sur la console
    </p>
    <script src="scriptPrincipal.js"></script>
</body>

</html>
```

librairie1.js > ...

```js
var  a = 10;

function afficher1() {
    console.log(`afficher de librairie1.js a : ${a} ${typeof a}`);
}

function incrementerA() {
    console.log("incrementerA de librairie1");
    a = a + 1;
}
```

librairie2.js > ...

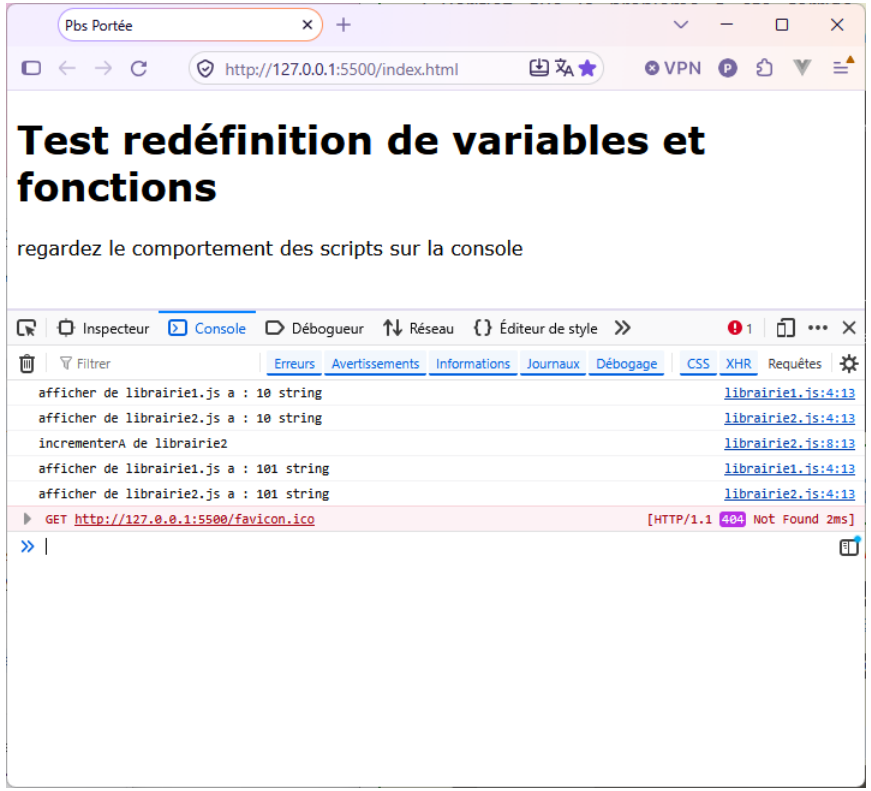
```js
var a = "10";

function afficher2() {
    console.log(`afficher de librairie2.js a : ${a} ${typeof a}`);
}

function incrementerA() {
    console.log("incrementerA de librairie2");
    a = a + 1;
}
```

scriptPrincipal.js

```js
afficher1();
afficher2();
incrementerA();
afficher1();
afficher2();
```



Les dernières déclarations prennent le dessus

Conflits possibles entre scripts

- Exemple illustratif

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initia
    <title>Pbs Portée</title>
    <script src="librairie2.js"></script>
    <script src="librairie1.js"></script>
    <style>
        body {
            font-family: Verdana, sans-serif;
        }
    </style>
</head>

<body>
    <h1>Test redéfinition de variables et fonctions</h1>
    <p>
        regardez le comportement des scripts sur la console
    </p>
    <script src="scriptPrincipal.js"></script>
</body>

</html>
```

librairie1.js

```js
var  a = 10;

function afficher1() {
    console.log(`afficher de librairie1.js a : ${a} ${typeof a}`);
}

function incrementerA() {
    console.log("incrementerA de librairie1");
    a = a + 1;
}
```

librairie2.js

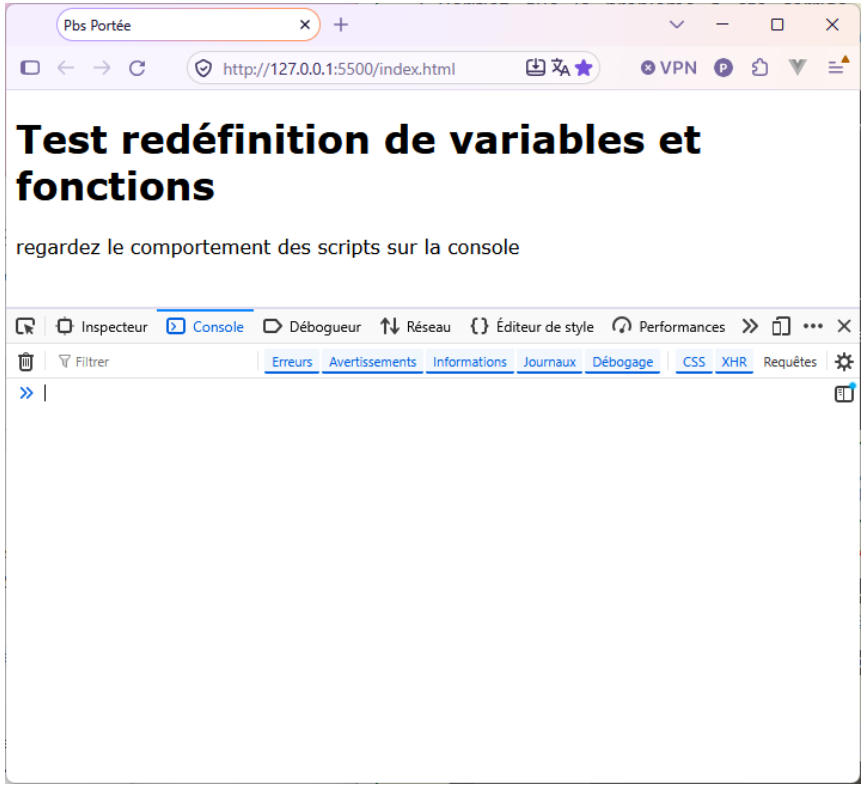
```js
var a = "10";

function afficher2() {
    console.log(`afficher de librairie2.js a : ${a} ${typeof a}`);
}

function incrementerA() {
    console.log("incrementerA de librairie2");
    a = a + 1;
}
```

scriptPrincipal.js

```js
afficher1();
afficher2();
incrementerA();
afficher1();
afficher2();
```



Que se passe-t-il si on change l'ordre des scripts?

Conflits possibles entre scripts

- Exemple illustratif

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initia
    <title>Pbs Portée</title>
    <script src="librairie2.js"></script>
    <script src="librairie1.js"></script>
    <style>
        body {
            font-family: Verdana, sans-serif;
        }
    </style>
</head>

<body>
    <h1>Test redéfinition de variables et fonctions</h1>
    <p>
        regardez le comportement des scripts sur la console
    </p>
    <script src="scriptPrincipal.js"></script>
</body>

</html>
```

librairie1.js

```js
var  a = 10;

function afficher1() {
    console.log(`afficher de librairie1.js a : ${a} ${typeof a}`);
}

function incrementerA() {
    console.log("incrementerA de librairie1");
    a = a + 1;
}
```

librairie2.js

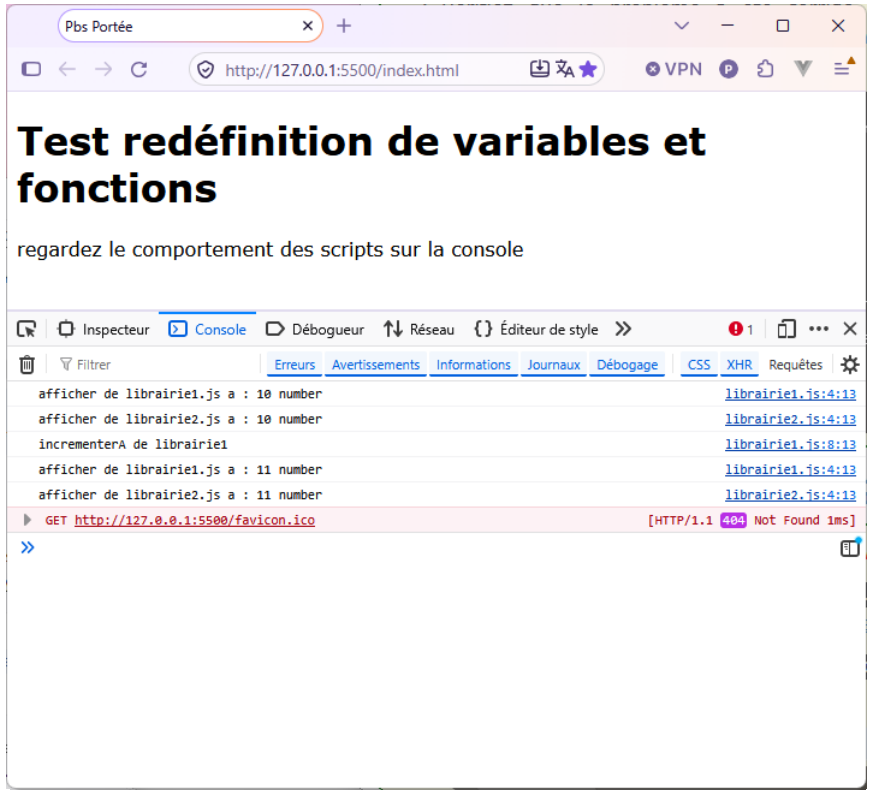
```js
var a = "10";

function afficher2() {
    console.log(`afficher de librairie2.js a : ${a} ${typeof a}`);
}

function incrementerA() {
    console.log("incrementerA de librairie2");
    a = a + 1;
}
```

scriptPrincipal.js

```js
afficher1();
afficher2();
incrementerA();
afficher1();
afficher2();
```



Comme précédemment les dernières déclarations prennent le dessus

Conflits possibles entre scripts

- Exemple illustratif

index.html > html

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initi
    <title>Pb Portée</title>
    <script src="librairie1.js"></script>
    <script src="librairie2.js"></script>
    <style>
        body {
            font-family: Verdana, sans-serif;
        }
    </style>
</head>

<body>
    <h1>Test redéfinition de variables et fonctions</h1>
    <p>
        regardez le comportement des scripts sur la console
    </p>
    <script src="scriptPrincipal.js"></script>
</body>

</html>
```

librairie1.js > ...

```js
var  a = 10;

function afficher1() {
    console.log(`afficher de librairie1.js a : ${a} ${typeof a}`);
}

function incrementerA() {
    console.log("incrementerA de librairie1");
    a = a + 1;
}
```

librairie2.js > ...

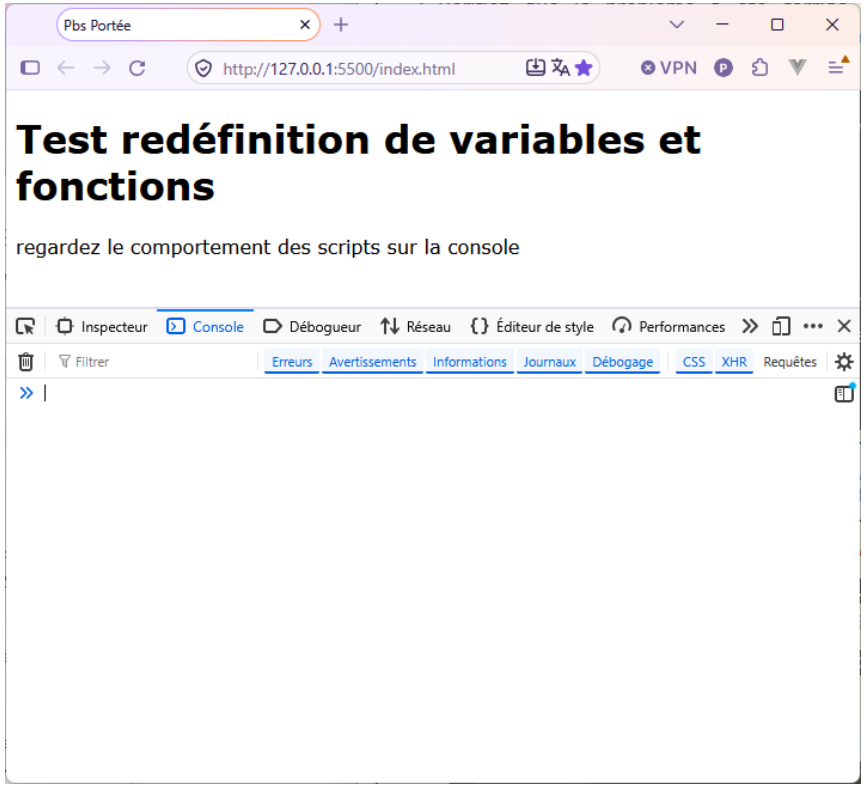
```js
var a = "10";

function afficher2() {
    console.log(`afficher de librairie2.js a : ${a} ${typeof a}`);
}

function incrementerA() {
    console.log("incrementerA de librairie2");
    a = a + 1;
}
```

scriptPrincipal.js

```js
afficher1();
afficher2();
incrementerA();
afficher1();
afficher2();
```



`let` résout-il le problème ?

Conflits possibles entre scripts

- Exemple illustratif

index.html > html

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, init
    <title>Pb Portée</title>
    <script src="librairie1.js"></script>
    <script src="librairie2.js"></script>
    <style>
        body {
            font-family: Verdana, sans-serif;
        }
    </style>
</head>

<body>
    <h1>Test redéfinition de variables et fonctions</h1>
    <p>
        regardez le comportement des scripts sur la console
    </p>
    <script src="scriptPrincipal.js"></script>
</body>

</html>
```

librairie1.js > ...

```js
let a = 10;

function afficher1() {
    console.log(`afficher de librairie1.js a : ${a} ${typeof a}`);
}

function incrementerA() {
    console.log("incrementerA de librairie1");
    a = a + 1;
}
```

librairie2.js > ...

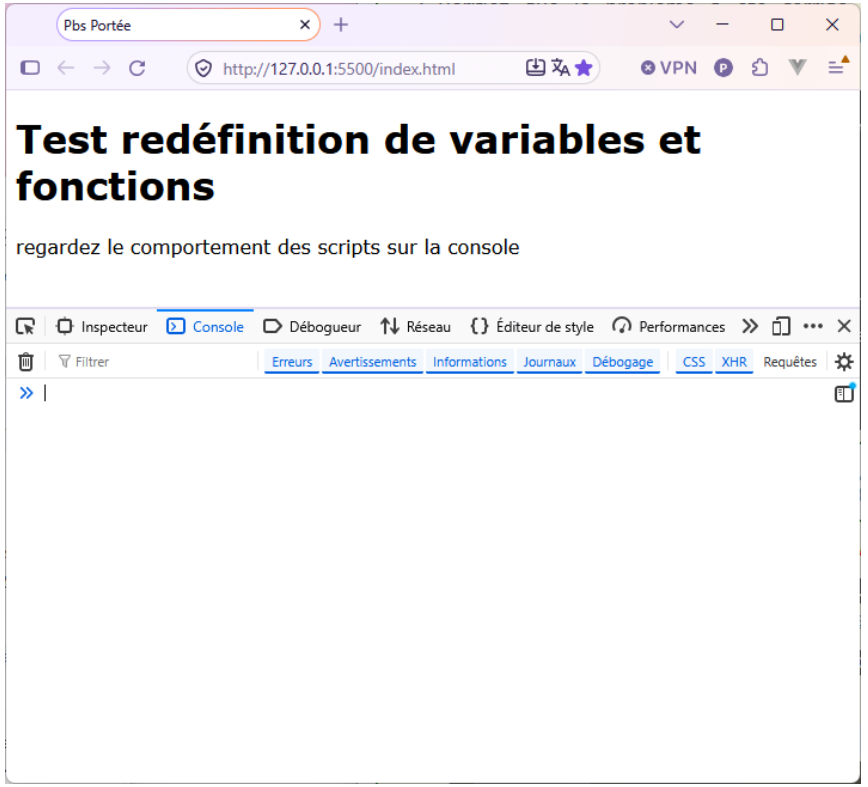
```js
let a = "10";

function afficher2() {
    console.log(`afficher de librairie2.js a : ${a} ${typeof a}`);
}

function incrementerA() {
    console.log("incrementerA de librairie2");
    a = a + 1;
}
```

scriptPrincipal.js

```js
afficher1();
afficher2();
incrementerA();
afficher1();
afficher2();
```



`let` résout-il le problème ?

Conflits possibles entre scripts

- Exemple illustratif

index.html > html

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, init
    <title>Pb Portée</title>
    <script src="librairie1.js"></script>
    <script src="librairie2.js"></script>
    <style>
        body {
            font-family: Verdana, sans-serif;
        }
    </style>
</head>

<body>
    <h1>Test redéfinition de variables et fonctions</h1>
    <p>
        regardez le comportement des scripts sur la console
    </p>
    <script src="scriptPrincipal.js"></script>
</body>

</html>
```

librairie1.js > ...

```js
let a = 10;

function afficher1() {
    console.log(`afficher de librairie1.js a : ${a} ${typeof a}`);
}

function incrementerA() {
    console.log("incrementerA de librairie1");
    a = a + 1;
}
```

librairie2.js > ...

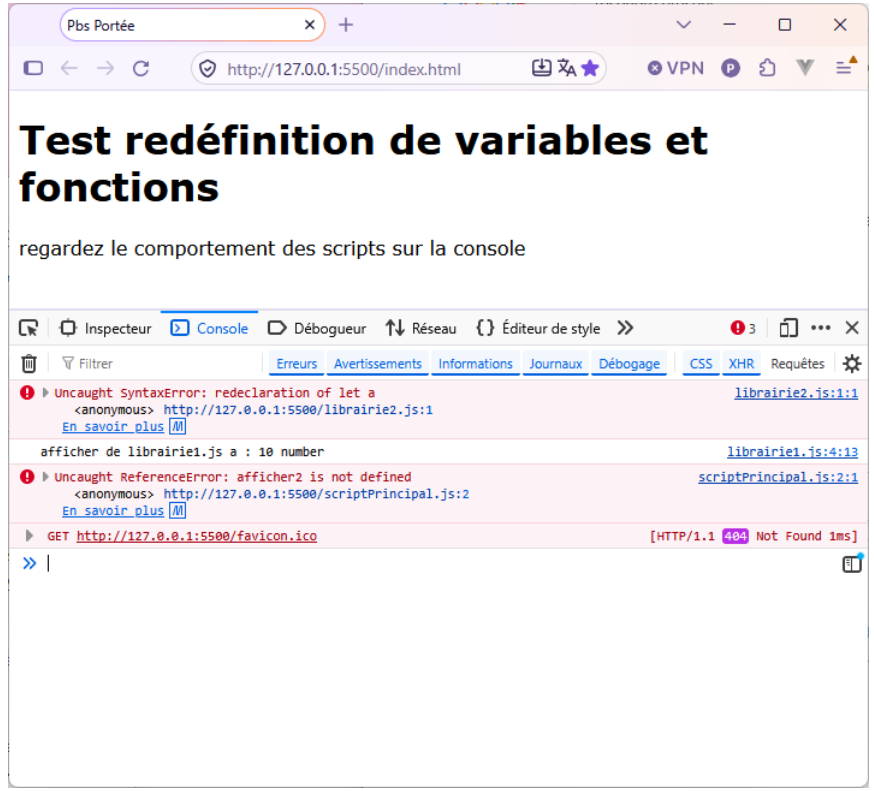
```js
let a = "10";

function afficher2() {
    console.log(`afficher de librairie2.js a : ${a} ${typeof a}`);
}

function incrementerA() {
    console.log("incrementerA de librairie2");
    a = a + 1;
}
```

scriptPrincipal.js

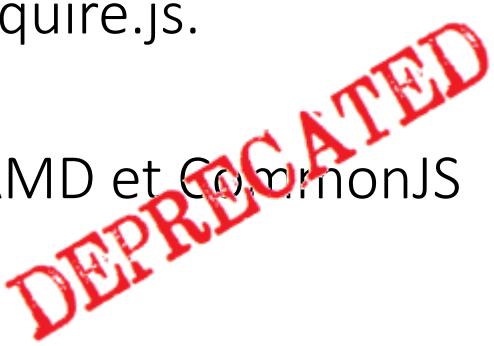
```js
afficher1();
afficher2();
incrementerA();
afficher1();
afficher2();
```



Pas vraiment, mais évite des erreurs silencieuses

Contexte

- plusieurs bibliothèques on été proposées pour la mise en œuvre et l'utilisation de modules
  - AMD – initialement mis en œuvre par la bibliothèque require.js.
  - CommonJS – le système de module créé pour Node.js
  - UMD – système de module universel, compatible avec AMD et CommonJS



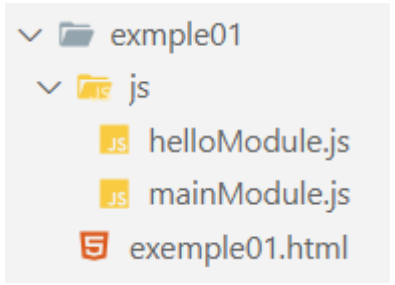
- 2015 apparition d'un système de modules intégré au langage → modules ES
  - pris en charge maintenant par les principaux navigateurs et Node.js

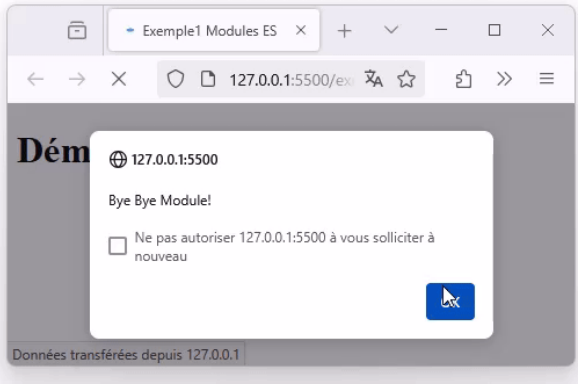
Qu'est-ce qu'un module ?

- un module ES ⇔ fichier JavaScript (un script est un module)
- Les modules
  - peuvent se charger mutuellement
  - utilisent des directives `import` et `export` pour échanger des fonctionnalités

- chaque module a sa propre portée globale :
  - variables et fonctions globales d'un module ne sont par défaut pas visibles dans les autres scripts.
  - un module doit exporter (directive `export`) les ressources qu'il veut rendre accessibles depuis l'extérieur
  - un module doit importer (directive `import`) les ressources dont il a besoin

Exemple de modules dans un navigateur





Le navigateur extrait et évalue automatiquement le module importé (et, le cas échéant, ses importations), puis exécute le script.

```
// helloModule.js
export function sayHelloBye(user) {
  sayHello(user);
  sayByeBye(user);
}

function sayHello(user) {
  alert(`Hello, ${user}!`);
}

function sayByeBye(user) {
  alert(`Bye Bye ${user}!`);
}
```

fonction exportée, accessible par les modules utilisant (important) ce module

fonctions non exportées, accessibles uniquement dans ce module

```
// mainModule.js
import {sayHelloBye} from './helloModule.js';

sayHelloBye(Module');
```

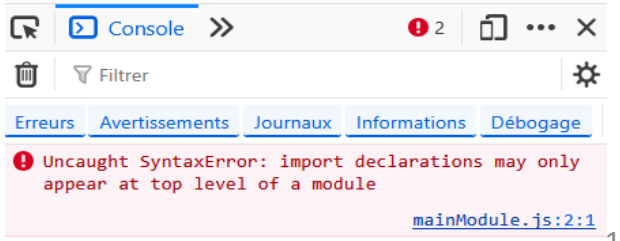
Importation de la fonction `sayHelloBye` depuis le module `helloModule.js`

Le chemin pour l'importation est un chemin relatif par rapport à l'emplacement du module effectuant l'importation

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Exemple1 Modules ES</title>
</head>
<body>
    <h1>Démo modules</h1>
    <script src="./js/mainModule.js" type="module"></script>
</body>

</html>
```

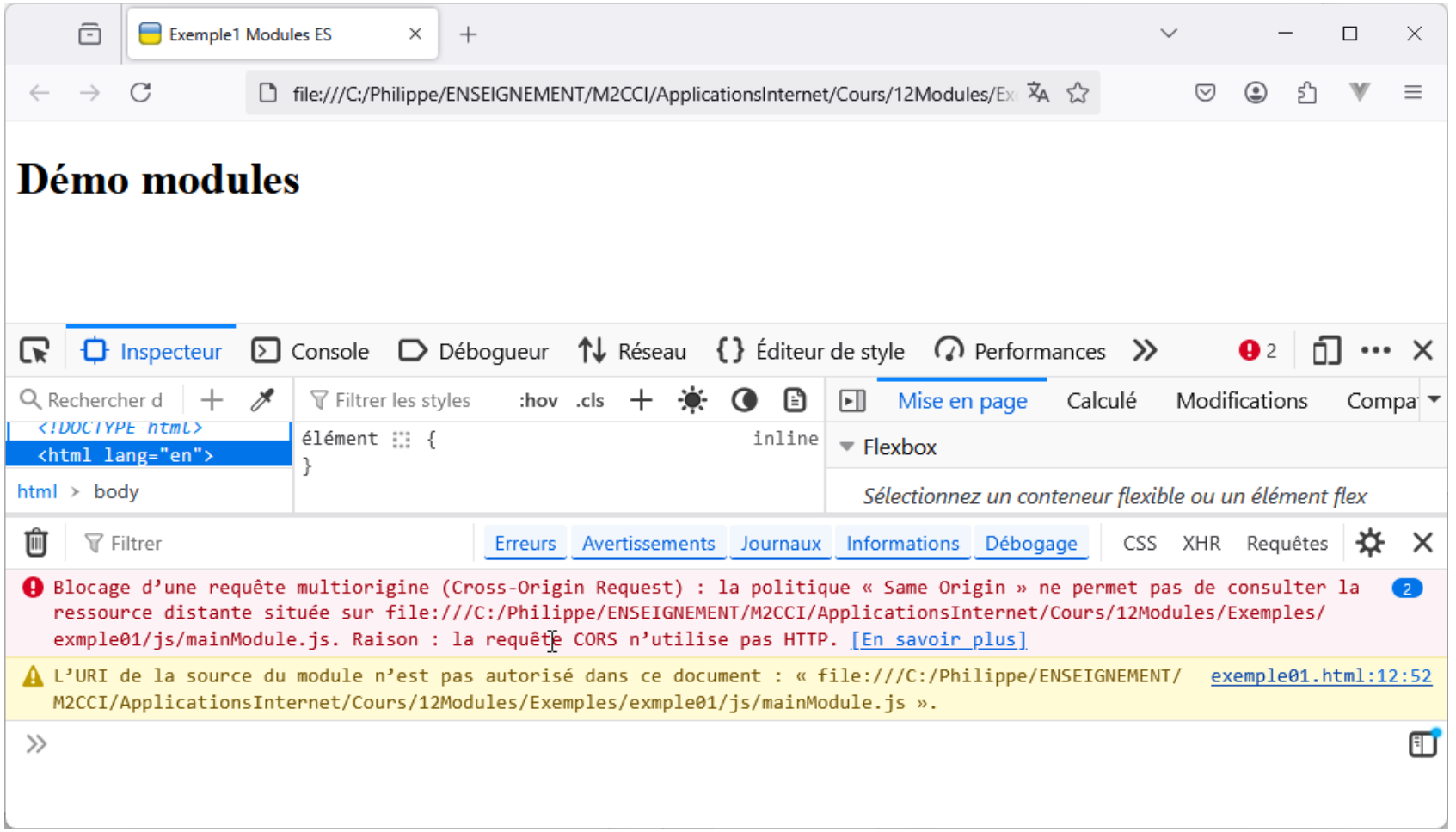
il faut indiquer que le code JavaScript constitue un module sinon une erreur intervient au chargement du script



Modules dans navigateur



Les modules ne fonctionnent pas localement (via protocole `file://`) il faut obligatoirement passer par un serveur HTTP(s)



Chargement d'un module

```
// 🗀 helloModule.js
export function sayHelloBye(user) {
  sayHello(user);
  sayByeBye(user);
}

function sayHello(user) {
  alert(`Hello, ${user}!`);
}

function sayByeBye(user) {
  alert(`Bye Bye ${user}!`);
}

console.info('helloModule.js chargé');
```

Évaluation du script

L'instruction est exécutée

```
// 🗀 mainModule.mjs
import {sayHelloBye} from './helloModule.js';
console.info('mainModule.js chargé');
sayHelloBye('Module');
```

Charge `helloModule.js`

Évaluation du script

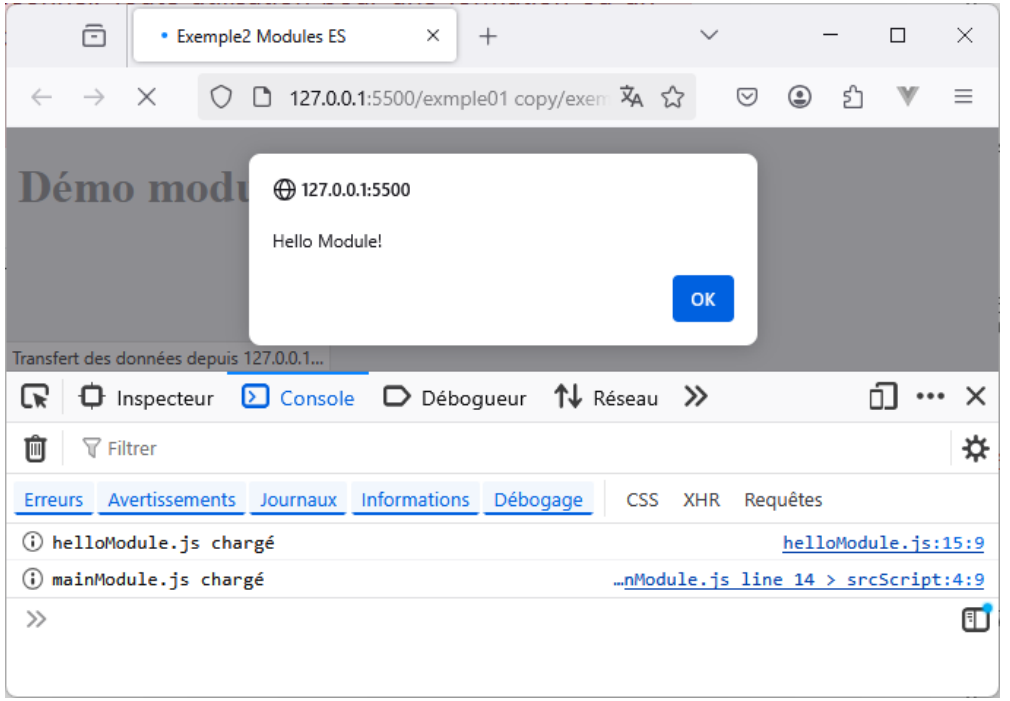
Les instructions sont exécutées

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Exemple1 Modules ES</title>
</head>
<body>
    <h1>Démo modules</h1>
    <script src="./js/mainModule.js" type="module"></script>
</body>
</html>
```

Chargement de la page

Charge `mainModule.js`

Lorsque un module est chargé le script est évalué et ses instructions de niveau supérieur sont exécutées



⚠ Si le même module est importé dans plusieurs autres modules, son code n'est exécuté qu'une seule fois, lors de la première importation.

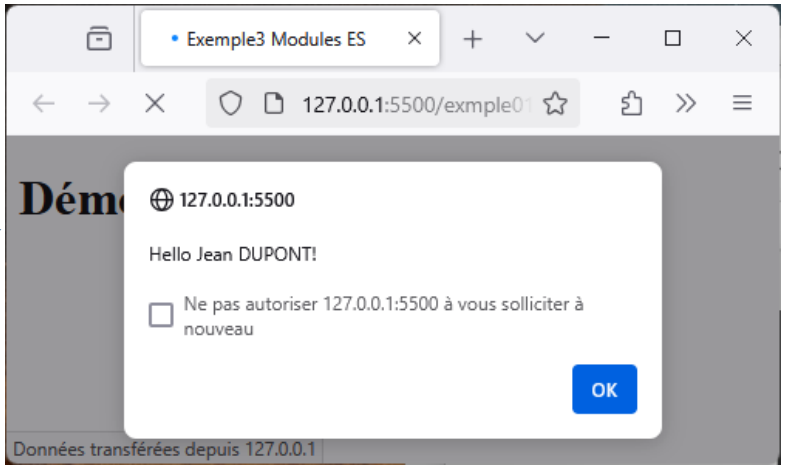
Exportation d'objets

```
// 🗀 helloByeModule.js
export let defaultUser = {
  nom : "Jean DUPONT"
};

export function sayHello(user = defaultUser) {
    alert(`Hello ${user.nom}!`);
  }

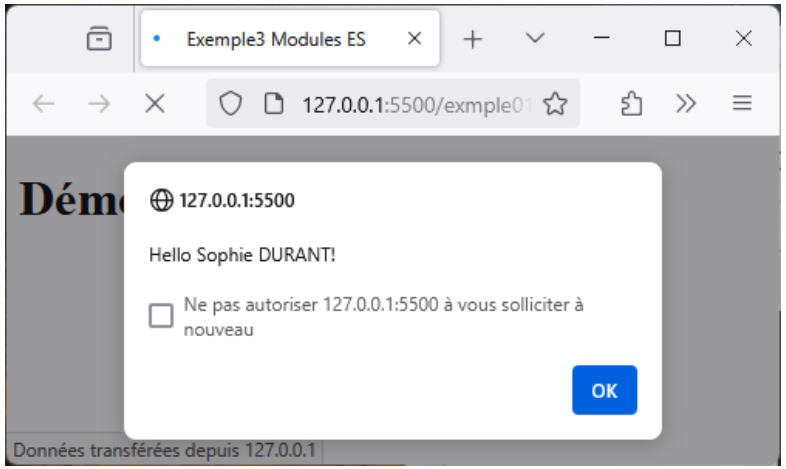
export function sayByeBye(user = defaultUser) {
  alert(`Bye Bye ${user.nom}!`);
}
sayHello();
```

Chargement et évaluation du module `helloByeModule.js`



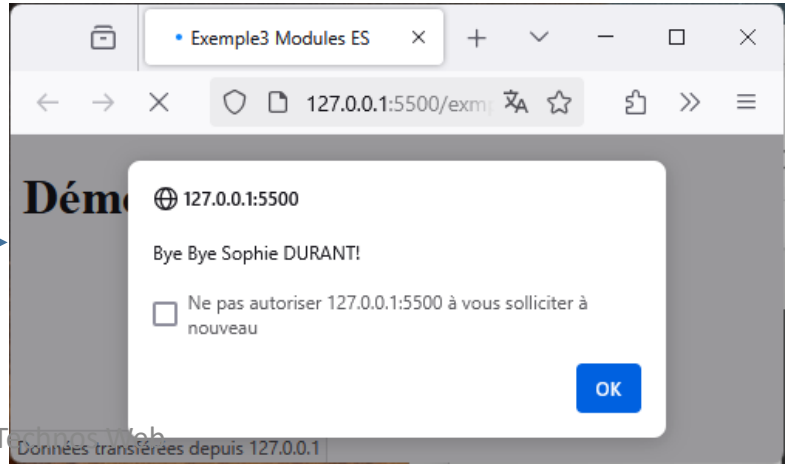

```
// 🗀 mainModule1.mjs
import {sayHello, defaultUser} from './helloByeModule.js';
defaultUser.nom = "Sophie DURANT";
sayHello();
```

Chargement et évaluation du module `mainModule1.js`




```
// 🗀 mainModule2.mjs
import {sayByeBye, defaultUser} from './helloByeModule.js';
sayByeBye();
```

Chargement et évaluation du module `mainModule2.js`




```
<!DOCTYPE html>
<!-- 🗀 exemple03.html →
...
<body>
    <h1>Démo modules</h1>
    <script src="./js/mainModule1.js" type="module"></script>
    <script src="./js/mainModule2.js" type="module"></script>
</body>
</html>
```

Exportation de variables

```
// 🗀 helloByeModule.js
export let defaultUser = "Jean DUPONT"

export function sayHello(user = defaultUser) {
    alert(`Hello ${user}!`);
  }

export function sayByeBye(user = defaultUser) {
  alert(`Bye Bye ${user}!`);
}
sayHello();
```

```
// 🗀 mainModule1.mjs
import {sayHello, defaultUser} from './helloByeModule.js';
defaultUser= "Sophie DURANT";
sayHello();
```

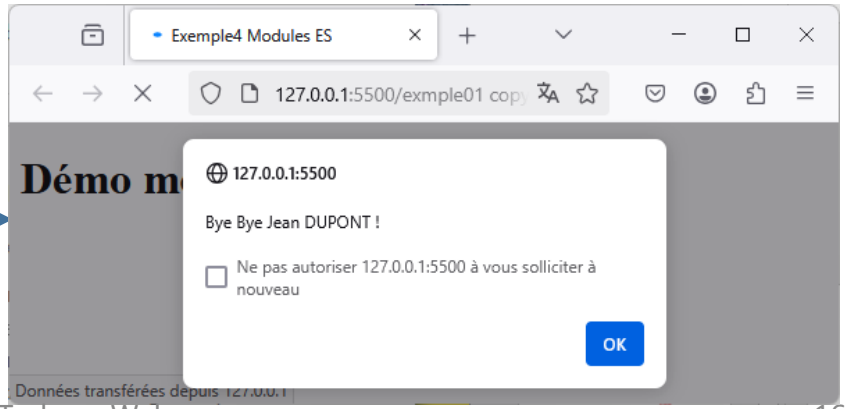
```
// 🗀 mainModule2.mjs
import {sayByeBye, defaultUser} from './helloByeModule.js';
sayByeBye();
```

```
<!DOCTYPE html>
<!-- 🗀 exemple04.html →
...
<body>
    <h1>Démo modules</h1>
    <script src="./js/mainModule1.js" type="module"></script>
    <script src="./js/mainModule2.js" type="module"></script>
</body>
</html>
```

Chargement et évaluation du module `mainModule2.js`

③

Que donne l'exécution du script mainModule2.js ?



Exportation de variables

```
// 🗀 helloByeModule.js
export let defaultUser = "Jean DUPONT"

export function sayHello(user = defaultUser) {
    alert(`Hello ${user}!`);
  }

export function sayByeBye(user = defaultUser) {
  alert(`Bye Bye ${user}!`);
}
sayHello();
```

```
// 🗀 mainModule1.mjs
import {sayHello, defaultUser} from './helloByeModule.js';
defaultUser= "Sophie DURANT";
sayHello();
```

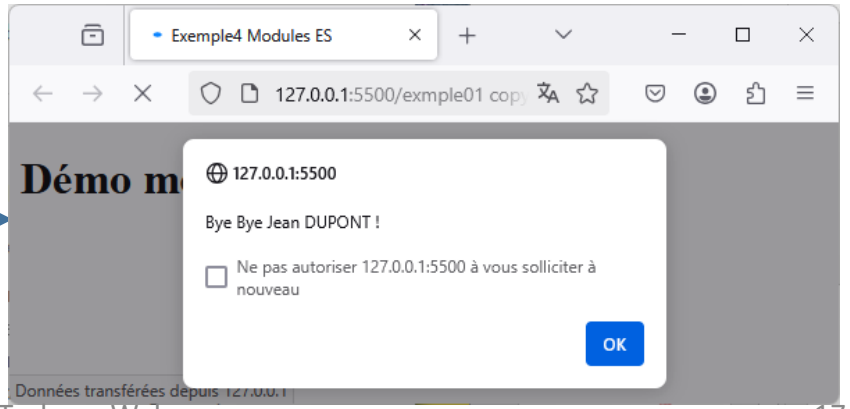
```
// 🗀 mainModule2.mjs
import {sayByeBye, defaultUser} from './helloByeModule.js';
sayByeBye();
```

```
<!DOCTYPE html>
<!-- 🗀 exemple04.html →
...
<body>
    <h1>Démo modules</h1>
    <script src="./js/mainModule1.js" type="module"></script>
    <script src="./js/mainModule2.js" type="module"></script>
</body>
</html>
```

Chargement et évaluation du module `mainModule2.js`

③

Pourquoi ?



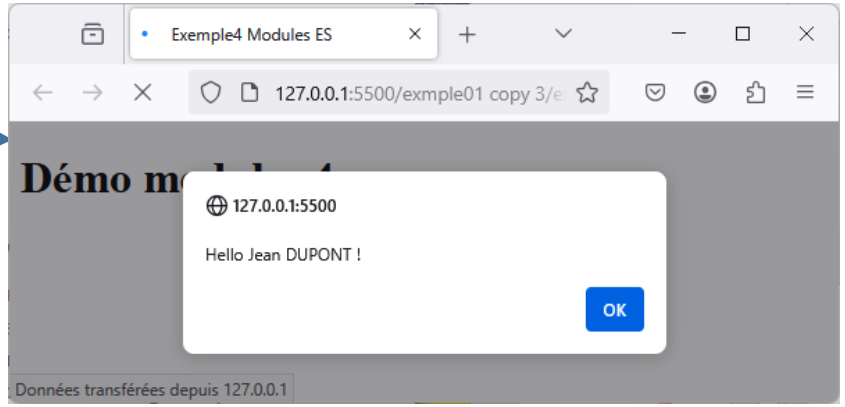
Exportation d'objets

```
// 🗀 helloByeModule.js
export let defaultUser = "Jean DUPONT"

export function sayHello(user = defaultUser) {
    alert(`Hello ${user}!`);
  }

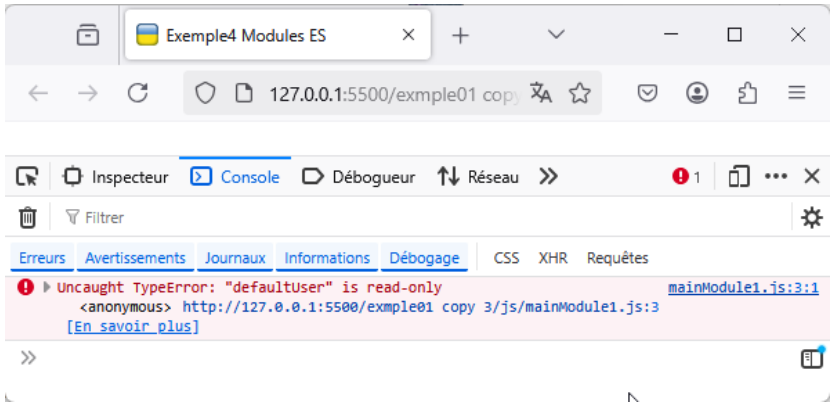
export function sayByeBye(user = defaultUser) {
  alert(`Bye Bye ${user}!`);
}
sayHello();
```

Chargement et évaluation du module `helloByeModule.js` ①



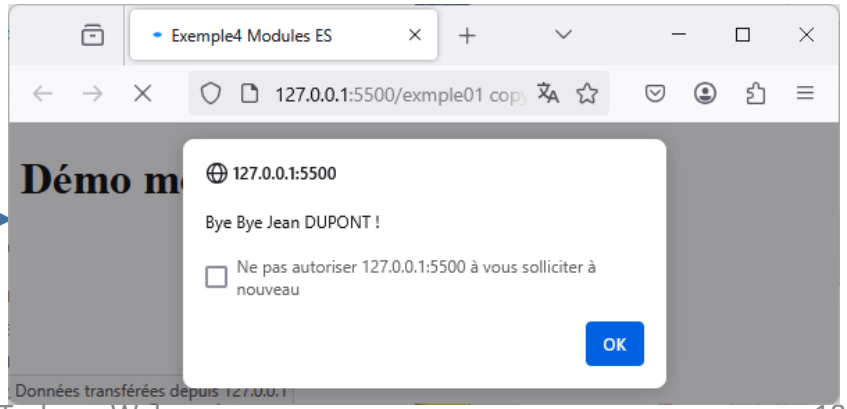

```
// 🗀 mainModule1.mjs
import {sayHello, defaultUser} from './helloByeModule.js';
defaultUser= "Sophie DURANT";
sayHello();
```

Chargement et évaluation du module `mainModule1.js` ②




```
// 🗀 mainModule2.mjs
import {sayByeBye, defaultUser} from './helloByeModule.js';
sayByeBye();
```

Chargement et évaluation du module `mainModule2.js` ③




```
<!DOCTYPE html>
<!-- 🗀 exemple04.html →
...
<body>
    <h1>Démo modules</h1>
    <script src="./js/mainModule1.js" type="module"></script>
    <script src="./js/mainModule2.js" type="module"></script>
</body>
</html>
```

Exportations de variables

- Les variables exportées par un module en JavaScript sont en lecture seule (*readonly*) pour les modules qui les importent
  - permet de maintenir l'intégrité des données et d'éviter les effets de bords

- On ne peut pas modifier directement une variable exportée depuis un autre module.

- Cependant, on peut toujours modifier les propriétés d'un objet exporté

Chargement différé des modules

- Les modules sont toujours différés (même effet que l'attribut `defer`)
  - télécharger des modules externes `<script type="module" src="...">` ne bloque pas le traitement HTML, ils se chargent en parallèle avec d'autres ressources.
  - les modules attendent que le document HTML soit complètement prêt puis s'exécutent (même s'ils sont très petits et se chargent plus rapidement que le code HTML).
  - l'ordre relatif des scripts est maintenu : les scripts qui apparaissent en premier dans le document sont exécutés en premier.

→ les modules "voient" toujours la page HTML entièrement chargée, y compris les éléments HTML situés en dessous.

Chargement différé des modules

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Exemple 5 Modules ES</title>
    <link rel="icon" href="../images/ukraine.png" type="image/png">
</head>

<body>
    <h1>Exemple 05 modules</h1>
    <script type="module">
        let b1 = document.getElementById("button");
        console.log(b1);
        b1.addEventListener("click", () => alert("Module"));
    </script>

    <script>
        let b2 = document.getElementById("button");
        console.log(b2);
        b2.addEventListener("click", () => alert("Script "));
    </script>

    <button id="button">Button</button>
</body>

</html>
```

Qu'est-ce qui est affiché sur la console et que se passe-t-il lorsque l'on clique sur le bouton?

Chargement différé des modules

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Exemple 5 Modules ES</title>
    <link rel="icon" href="../images/ukraine.png" type="image/png">
</head>

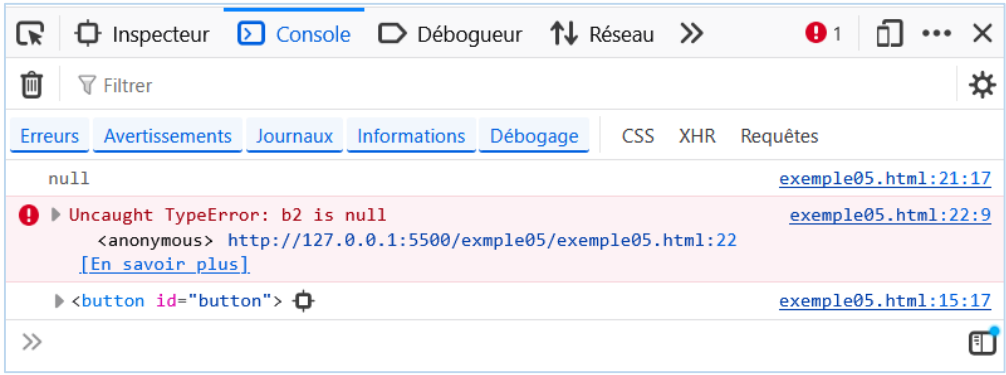
<body>
    <h1>Exemple 05 modules</h1>
    <script type="module">
        let b1 = document.getElementById("button");
        console.log(b1);
        b1.addEventListener("click", () => alert("Module"));
    </script>

    <script>
        let b2 = document.getElementById("button");
        console.log(b2);
        b2.addEventListener("click", () => alert("Script "));
    </script>

    <button id="button">Button</button>
</body>

</html>
```

Qu'est-ce qui est affiché sur la console et que se passe-t-il lorsque l'on clique sur le bouton?



Le script 2 et un script «*normal*» il s'exécute une fois qu'il est chargé et avant que le reste de la page ne soit traité

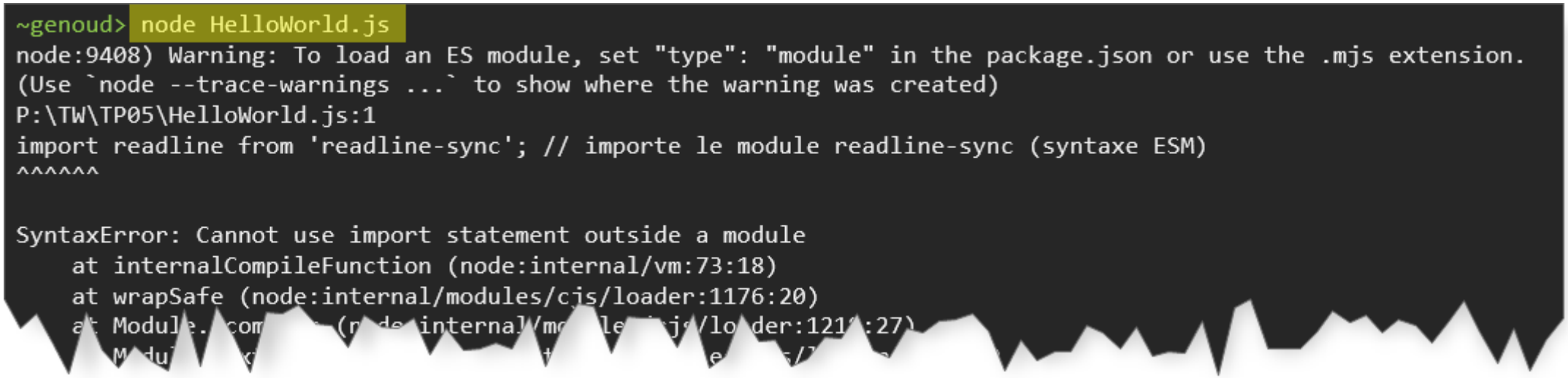
→ b2 = null
→ addEventListener provoque une erreur

Le script 1 est chargé et est différé. Il ne s'exécute que lorsque la page a été complétement traitée.

→ b1 = référence de l'objet JavaScript représentant le bouton dans le DOM

Modules et node.js

- Par défaut avec Node.js les fichier .js ne sont pas considérés comme des modules



Modules et node.js

pour utiliser modules ES2015 avec node mettre extension **.mjs**

labélise les fonctions et variables qui pourront être utilisées en dehors du module

```
// 🗀 sayHello.mjs
export function sayHelloBye(user) {
  sayHello(user);
  sayByeBye(user);
}

function sayHello(user) {
    console.log(`Hello ${user}!`);
  }

function sayByeBye(user) {
  console.log(`Bye bye ${user}!`);
}
```

permet importation de certaines fonctionnalités d'un autre module

chemin du module par rapport au fichier actuel

```
// 🗀 mainModule.mjs
import {sayHelloBye} from './sayModule.mjs';

console.log(sayHelloBye);
sayHelloBye('Joe Doo');
```

affecte la fonction exportée **sayHelloBye** à la variable correspondante.

```
λ node mainModule.mjs
[Function: sayHelloBye]
Hello Joe Doo!
Bye bye Joe Doo!
```

.mjs ou .js ?

- documentation de V8 recommande d'utiliser extension `.mjs`
  - meilleure clarté pour distinguer fichiers qui sont des modules des fichiers javascript classiques
  - permet d'analyser correctement les fichiers modules par les environnements d'exécution tels Node.js et outils de compilation tels Babel, Vite…
  - par contre pour déployer des fichier `.mjs` sur le web il faut s'assurer que le serveur est configuré pour utiliser le type MIME approprié :
    - `Content-Type: text/javascript`
  - mais si vous n'avez pas accès à la configuration du serveur il se peut qu'il n'associe pas le bon type au fichier `.mjs` , dans ce cas mieux vaut garder l'extension `.js`

  - pour en savoir plus :
    - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules#aside_%E2%80%94_.mjs_versus_.js
    - https://v8.dev/features/modules#mjs

Modules et Node.js

- Possibilité d'utiliser l'extension .js pour les modules



n'oubliez pas la **,** pour séparer la déclaration des dépendances de la déclaration **"type": "module"** et n'oubliez pas les **""** pour écrire **type** et **module**

```
~genoud> node HelloWorld.js
Hello World !
Entrez votre nom : Joe Doe
Bonjour Joe Doe !
~genoud>
```

Pour en savoir plus

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules
- https://fr.javascript.info/import-export