Les Modules

De quoi parlons nous ? Vous vous en êtes déjà sûrement rendu compte, plus on développe de fonctionnalités sur un projet, plus votre fichier JavaScript va être long et chargé, et moins on va s'y retrouver facilement. C'est le cas dans tout langage de programmation. C'est pourquoi, les développeurs préfèrent diviser le code en plein de petites briques: c'est ce qu'on appelle des ***modules***. Cela présente de nombreux avantages :

- Le code est plus lisible, chaque brique a son utilité
- Il est plus simple de retrouver une erreur si l'on sait quelle fonctionnalité crée un bug
- Une brique de code pourra être réutilisée dans le reste du projet, voire même dans d'autres projets
- Le découpage rend le code plus facilement extensible. On pourra aisément rajouter une ou plusieurs briques de code et les utiliser.
- La gestion des dépendances entre blocs est simplifiée : chaque brique indique clairement ses dépendances, c'est-à-dire les morceaux de codes qu'elle utilise. En cas de problème ou de dépendances cassées, on peut identifier plus facilement le ou les blocs qui font défaut.

A l'origine, JavaScript n'était pas prévu pour la gestion de modules, et cela n'a changé que très récemment, lorsque ce langage est devenu le langage de base de nombreuses applications web, de plus en plus complexes. Il était notamment impossible de référencer directement un script JS à partir d'un autre. C'est pourquoi les développeurs devaient mettre tout leur code dans un seul et même immense fichier.

Nous allons voir dans ce cours la manière de structurer votre JavaScript en petites briques réutilisables facilement, c'est ce qu'on appelle : le ***Module Pattern***. Nous verrons ensuite comment utiliser les briques les unes avec les autres.

Mais avant cela, quelques rappels s'imposent !

Préambule

Avant de s'attaquer au ***module pattern***, nous allons voir ou revoir quelques points importants à avoir en tête pour comprendre le découpage du code JavaScript en modules.

La portée des variables

La portée des variables est sans doute l'élément le plus important pour motiver le découpage en module du code en Javascript.

L'une des choses à retenir pour comprendre le module pattern est que les variables déclarées avec le mot-clé `var` dans une fonction, ne sont pas accessibles en dehors de celle-ci.

```
function maFonction(){
    var a = 10;      //a est accessible ICI
}

//a n'est pas accessible ICI
```

Il est vivement conseillé de toujours utiliser le mot-clé **var**, si vous ne le faites pas la variable devient *globale* !

```
function maFonction(){
    a = 10; //ce qu'il ne faut PAS FAIRE !
}

maFonction();

console.log(a); // affichera 10 !! a est global
```

Les variables globales sont à éviter. D'une part, elles posent des problèmes de sécurité car rien ne vous garantit qu'elles ne seront pas écrasées ou modifiées par un autre script. Et d'autre part pour des raisons de performances, plus le **scope** (espace de définition) est loin plus le temps d'accès est long.

Le Garbage Collector

Lorsqu'on utilise JavaScript, la mémoire est allouée lors de la création des objets puis libérée "automatiquement" lorsque ceux-ci ne sont plus utilisés. Cette libération automatique est appelée *garbage collection* en anglais ou ramasse-miettes.

Ce procédé a pour but de surveiller l'utilisation de la mémoire afin de déterminer quand une partie de la mémoire allouée n'est plus utilisée afin de la libérer automatiquement.

Fonctions dans des fonctions

En JavaScript les fonctions sont considérées comme des objets **Function**. Il est possible de créer des fonctions à l'intérieur d'une fonction (on appelle ça des *closures*, mais nous n'aborderons pas précisément ce concept dans ce cours) :

```
function maFonction(){
    //création d'une fonction à portée privée
    function test(){
        console.log('Je suis une fonction contenue dans maFonction');
    }

    test();
}

maFonction();
```

Dans cet exemple il est important de noter que la fonction **test()** n'est pas accessible en dehors de maFonction(). Elle est *encapsulée* dans l'objet maFonction.

Fonctions anonymes

JavaScript est un *langage objet* et donc les variables peuvent pointer sur des objets. Nous avons vu qu'une fonction était un objet, donc il est possible de faire ceci :

```
var bonjour = function() {
    console.log("Je suis une fonction anonyme");
};

bonjour(); //affichera 'Je suis une fonction anonyme' dans la console
```

Ce code montre la création d'une variable bonjour qui se voit affecter une référence vers une *fonction anonyme* qui est déclarée tout de suite après. L'appel de la fonction se fait avec le nom de la variable suivi de parenthèses. On peut bien évidemment passer des arguments à la fonction.

Si la variable *bonjour* est détruite, ou mise à undefined, et qu'il n'existe aucune autre variable détenant la référence vers la fonction, celle-ci sera nettoyée par le *garbage collector*. Vous noterez que la fonction se termine par un `;` car c'est une affectation et non une simple déclaration.

Le Module Pattern

Nous y voilà ! Qu'est-ce donc que le ***module pattern*** ?

Le module pattern est une manière d'encapsuler du code dans un **package ou namespace** tout en permettant si besoin, un accès extérieur à certaines propriétés/fonctions.

Sans plus attendre voilà à quoi il ressemble :

```
var MODULE = (function() {
    var self = {};
```

```
    var variablePrivee = 10;

    self.attributPublic = "bonjour";

    function methodePrivee() {
        console.log('Je suis encapsulée !');
    }

    self.methodePublique = function(){
        console.log('Je suis accessible !');
    };

    return self;
})();
```

Quelques explications s'imposent...

Premièrement, vous pouvez remarquer qu'une fonction est créée et qu'elle est passée dans une variable nommée **MODULE**. Rien de nouveau, nous l'avons vu plus haut.

Oui, mais... pourquoi toutes ces parenthèses autour de la fonction ?

On appelle ça une **self-invoking function**, ce qui peut se traduire en français par *fonction qui se lance automatiquement (auto-exécution)*. En résumé, la fonction est appelée automatiquement au moment de sa création. Cela permet d'initialiser entièrement les attributs qu'elle contient.

À l'intérieur de notre fonction on peut voir qu'une variable `self` est déclarée. Cette variable est TRÈS importante, elle est la clef du problème d'encapsulation que vous avez soulevé !

Le nom '**self**' est un nom que j'utilise dans mes projets, vous pouvez bien évidemment nommer cette variable comme vous le souhaitez !

En fait, `self` est un objet (ou ensemble) qui ne contient rien à sa déclaration. Pendant la création du module, on y met plusieurs choses : *attributPublic* et *methodePublique*. Ces deux éléments appartiennent à l'objet `self`.

A la fin de la création du module, on retourne `self`.

Mais du coup, la variable MODULE ne contient pas vraiment une fonction mais le contenu de la variable self ?!

Oui ! Enfin, pas tout à fait, elle contient la référence vers la variable `self`. Et c'est pourquoi on peut faire ceci :

```
var MODULE = /* code ci-dessus */

MODULE.methodePublique(); // affiche 'Je suis accessible'

console.log(MODULE.attributPublic); // affiche 'bonjour'
```

Tout ce qui est ajouté à `self` est disponible en dehors du module, c'est *public*. Tout ce qui n'y est pas est local au module et n'est pas accessible en dehors, c'est *privé*.

Mais, le reste du module est détruit alors ?

Pas du tout, comme la variable **MODULE** contient une *référence* vers `self` et que `self` appartient au module, tout le contenu du module persiste en mémoire. Rien n'est détruit par le *garbage collector* ! On peut donc utiliser des fonctions privées (ou publiques) dans des fonctions publiques par exemple et inversement :

```
var MODULE = (function(){
    var self = {};

    function methodePrivee() {
        console.log('Je suis encapsulée !');
    }

    self.methodePublique = function(){
        methodePrivee();
        self.methodePubliqueDeux()
    };

    self.methodePubliqueDeux = function(){
        console.log('Je suis publique !');
    };

    return self;
})();

MODULE.methodePublique(); // affiche 'Je suis encapsulée' puis 'Je suis publique'
```

Une fonctionnalité, un module

D'une manière générale, il faut éviter au maximum de mettre tout votre code dans l'espace global ou même dans un seul module. C'est comme si les maisons n'avaient qu'une seule grande pièce et que tout était mis un peu n'importe où. Découper son espace en sous-espace est une bonne pratique qu'il faut respecter et cela, quelque soit le langage.

C'est pourquoi il est conseillé de créer un module par fonctionnalité ou ensemble de fonctionnalités qui ont un lien entre elles. Voilà un exemple de ce qui peut se faire :

```
//PREMIER MODULE
// Gestionnaire de logs
var Logger = (function(){
    var self = {};

    // attribut privé
    var logger = new Array();

    //méthode privée
    function displayLog(log){
        console.log(log.module + " : " + log.message);
```

```
    }

    //methodes publiques
    self.log = function(moduleName, msg){
        var log = {module: moduleName, message: msg};
        displayLog(log);
        logger.push(log);
    };

    self.showAll = function(){
        for (var i = 0; i < logger.length; i++) {
            displayLog(logger[i]);
        }
    };

    return self;
})();

// DEUXIEME MODULE
// Gestionnaire d'un div particulier (inscription, connexion,...)
var DivManager = (function(){
    var self = {};
    var div = undefined;
    // si l'on souhaite retarder la récupération de l'élément on peut
    // faire les opérations dans une fonction qui sera appelée dans le code.
    // C'est un choix personnel, mais vous pouvez tout à fait déclarer directement
    // la variable var div = document.getElementById(...) au dessus.

    self.init = function(){
        div = document.getElementById('monDiv');
        div.onclick = onClick;
    };

    function onClick(){
        //utilisation d'un autre module :)
        Logger.log("DivManager", "Le bloc a été cliqué");
    }

    return self;

})();

DivManager.init(); //on appelle la fonction du module pour initialiser le click
```

On constate qu'il y a deux modules, un qui se charge uniquement de la gestion des logs (enregistrement, affichage etc.) et un autre qui est dévoué à la gestion d'un élément HTML (dans notre cas de la récupération d'une div et de l'ajout d'un écouteur d'événement).

Sous-modules

Pour chaque fonctionnalité il est conseillé de créer un module, mais généralement le module peut grandir très vite et devenir rapidement difficile à relire pour des programmeurs externes. C'est pourquoi il est primordial d'utiliser des **sous-modules** quand le cas se présente.

Les sous-modules sont des *sous-ensembles* du module principal. L'écriture est la suivante :

```
var ModuleParent =  (function(){
    /* module */
})();

ModuleParent.SousModule = (function(){
    /* sous module enfant */
})();
```

On déclare un nouvel élément dans ModuleParent (qui est un objet ou ensemble) qui contient lui-même un module et ses propriétés. Le sous-module n'a pas accès aux fonctionnalités de l'élément parent, car les deux ont leur propre *espace de noms*. Si l'enfant doit communiquer avec le parent, il faut que les méthodes et attributs soient publics. **Ce n'est pas de l'héritage !** C'est uniquement un découpage du code en fonctionnalités.

Un exemple parlant de communication parent/enfant :

```
// Gestionnaire d'un div particulier (inscription, connexion,...)
var DivManager = (function(){
    var self = {};

    self.div = undefined;
    //notez le changement, div appartient à self.

    self.init = function(){
        self.div = document.getElementById('monDiv');
        DivManager.Events.init();
    };

    return self;
})();

DivManager.Events = (function(){
    var self = {};
    self.init = function(){
        // on récupère div qui est public et on lui affecte un écouteur (click)
        DivManager.div.onclick = onClick;
        //ici je peux ajouter d'autres écouteurs
    };

    function onClick(){
        console.log("J'ai été cliqué !");
    }
    // je peux ajouter ici les fonctions spécifiques à de nouveaux écouteurs

    return self;
```

```
})();


DivManager.init(); //on appelle la fonction du module pour initialiser l'ensemble
```

On peut voir qu'on a externalisé le code qui gère les évènements sur la `div`. On évite ainsi de surcharger de fonctionnalités `DivManager` et le code s'en trouve plus lisible. Il est également plus facile de modifier le code, même 6 mois plus tard, car les noms sont explicites et précis.

Veillez à choisir correctement les noms des modules et des sous-modules. Ils sont essentiels pour facilement vous y retrouver. Également, prenez l'habitude de mettre une majuscule pour nommer les modules, cela vous permettra de différencier rapidement un module d'une fonction.

Attention à ne pas déclarer un sous-module **AVANT** un module. Le module parent (la variable) serait non défini.

Pour finir cette partie, je vous conseille de mettre chaque module (et ses sous-modules) dans des **fichiers indépendants** nommés par le nom du module. D'une part cela vous évitera d'avoir trop de code au même endroit et d'autre part vous verrez facilement si un nom de module existe déjà. Ceci est valable en développement, en production il peut être intéressant, selon vos besoins, de réduire le nombre de fichiers et de les compresser.

Un module parent

La multiplicité des bibliothèques, frameworks ou extensions JavaScript au sein d'un même projet peut entraîner un problème très important : l'écrasement des espaces de noms. Ceci arrivera inévitablement si votre module se nomme pareil qu'une bibliothèque que vous utilisez (ou pire encore, vous écraserez la bibliothèque). Pour éviter cette situation, il est vivement conseillé d'utiliser un module parent qui contiendra l'ensemble de vos modules. Celui-ci aura un nom unique qui pourra facilement être modifié s'il entre en conflit avec d'autres extensions.

Le fonctionnement est similaire aux *sous-modules*. L'idée est simplement de créer un module parent qui englobera l'ensemble des modules enfants. Le chargement de vos fichiers pouvant intervenir dans un ordre aléatoire, il faut faire attention à bien vérifier l'existence du parent avant de tenter d'y ajouter un enfant. Voilà comment procéder pour un parent se nommant **Application** :

```
var Application = Application || {};
```

Ce code signifie : *Si la variable **Application** existe tu la récupères, sinon tu déclares un nouvel ensemble vide*. Il doit être ***mis en haut de chaque fichier de module***. Il permet de s'assurer qu'on ne va pas tenter de créer un sous-ensemble sans que le parent ne soit initialisé. De plus, il évite d'écraser Application si celui-ci existe déjà et qu'il contient d'autres modules qui ont été chargés avant.

```
// fichier div-manager.js
var Application = Application || {};

// on peut créer le module enfant
Application.DivManager = (function(){
    /* ... */
})();

// et des sous modules...
Application.DivManager.Events = (function(){
    /* ... */
})();
```

Extensions de modules

Le problème des modules, c'est qu'ils sont déclarés dans un seul fichier. Dans le cas de très gros projets avec beaucoup de développeurs, il peut être intéressant de pouvoir ajouter des fonctionnalités sans modifier le fichier principal. Il existe une façon **d'étendre** les modules, et cela avec assez peu de changements.

Je ne parle pas de sous-modules mais bien de *l'ajout de fonctionnalités* dans un module.

```
var Application = Application || {};

Application.DivManager = (function(self){
    self.nouvelleMethode = function(){
        // fonctions à ajouter au module
    };

    return self;
})(Application.DivManager || {});
```

On peut voir sur cet exemple que l'on passe directement en paramètre le module à étendre (ou un ensemble vide si celui-ci n'existe pas). Ce paramètre est directement utilisé par le module à la place du **self** que l'on déclarait au départ.

En procédant ainsi, si le module n'existe pas, il est créé !

Attention : Si vous utilisez ce modèle de conception, tous les modules doivent être déclarés de la sorte et `self` ne doit plus être déclaré dans le corps du module (au risque d'écraser toute extension passée).

Le module qui a été étendu par l'exemple ci-dessus doit, par conséquent, être déclaré de la sorte :

```
var Application = Application || {};

Application.DivManager = (function(self){
    /* toutes les fonctions du module */
    return self;
})(Application.DivManager || {}); `
```

Cette façon de procéder permet un chargement parallèle des fichiers JavaScript. Le module n'a plus besoin d'être créé en amont, il sera créé soit dans le fichier principal soit par la première extension qui tentera de l'étendre.

Enfin, on peut bien évidemment étendre les sous-modules de la même manière en faisant attention à bien vérifier l'existence de leur parent !

Maintenant que le ***module pattern*** n'a plus de secret pour vous, voyons comment on peut l'utiliser dans un projet.

Comment utiliser le module pattern ?

Browser Style

Comme indiqué en introduction, Javascript ne proposait pas de gestion native des modules. L'implémentation de modules a donc nécessité de tirer partie de la syntaxe existante de Javascript. Ainsi, le module pattern décrit dans la partie précédente, permet d'implémenter des modules. Le gros inconvénient de ce *pattern* est qu'il nécessite de déclarer tous les modules dans le même fichier Javascript.

On peut trouver un moyen de charger plusieurs fichiers de scripts Javascript en indiquant ces scripts directement dans l'en-tête du fichier HTML principal en utilisant des balises `<scripts>`, par exemple:

```
<script type="text/javascript" src="script1.js" /></script>
<script type="text/javascript" src="script2.js" /></script>
<script type="text/javascript" src="script3.js" /></script>
<script type="text/javascript" src="script4.js" /></script>
```

On peut déclarer les différents scripts en omettant l'attribut `type="text/javascript"` qui est facultatif.

Malheureusement, cette technique est très limitée. Les modules déclarés en utilisant cette méthode ont de gros inconvénients:

- Le code est moins lisible puisqu'il faut indiquer tous les scripts au préalable dans le fichier principal.
- On ne peut pas charger de fichiers de façon asynchrone, tous les fichiers Javascript doivent être chargés au chargement de la page.
- Il faut qu'ils soient déclarés dans le bon ordre. Si une dépendance n'est pas satisfaite au moment où on souhaite l'utiliser, le code ne s'exécutera pas correctement.
- Suivant la façon avec laquelle les modules sont déclarés, ils peuvent être écrasés si, par mégarde, on les définit plusieurs fois.

Heureusement, JavaScript évolue au cours du temps. De nouveaux mots clés apparaissent et permettent de réaliser de nouvelles choses. L'ensemble des normes qui régissent le langage JavaScript s'appelle ECMAScript ou ES. La dernière version ayant apporté des changements significatifs au langage est la version sortie en 2015 : ES6 ou ES2015. Elle n'a pas encore été tout à fait intégrée par tous les navigateurs mais son support évolue progressivement. La version précédente, ES5, datait de 2009.

ES6 a donc apporté beaucoup (vraiment beaucoup) de nouvelles choses à JS, et notamment la possibilité pour un script de faire appel directement à un autre script sans passer par un fichier HTML.

Export et import de modules

Maintenant que nous avons vu comment créer et organiser des modules, voyons comment les relier entre eux.

Les propriétés les plus importantes à avoir en tête avec les modules ES2015 sont les suivantes:

- Par défaut, tous les modules ES2015 sont privés.
- Un fichier Javascript correspond à un module.
- Pour exposer des éléments comme des variables, des fonctions ou des classes à l'extérieur d'un module, il faut utiliser le mot-clé `export`.

Exporter un module

L'export d'éléments permet de les exposer à l'extérieur du module de façon à les rendre accessible à l'extérieur du module.

Pour exporter, on utilise le mot-clé `export`.

Par exemple:

```
export function func1() {
    // ...
}
```

Dans cet exemple, on exporte une fonction à l'extérieur du module.

D'autres formes de syntaxe sont possibles pour exporter plusieurs éléments du module en une fois:

```
function func1() {
    // ...
}

function func2() {
    // ...
}

export { func1, func2 };
```

On peut aussi indiquer l'instruction `export` avant la déclaration des éléments à exporter:

```
export { func1, func2 };

function func1() {
    // ...
}

function func2() {
    // ...
}
```

Ainsi, pour un module, nous pourrons faire

```
export MODULE;
var MODULE = (function(){
    var self = {};

    function methodePrivee() {
        console.log('Je suis encapsulée !');
    }

    self.methodePublique = function(){
        methodePrivee();
        self.methodePubliqueDeux()
    };

    self.methodePubliqueDeux = function(){
        console.log('Je suis publique !');
    };

    return self;
})();
```

Importer un module

L'import d'un module correspond à importer les éléments exportés par ce module dans un autre.

Pour importer tous les éléments exportés d'un module dans une variable, on peut utiliser la syntaxe:

```
import * as moduleInFile from './file.js';
```

Dans ce cas, la variable est `moduleInFile`, on peut l'utiliser directement pour accéder aux éléments du module, par exemple:

```
moduleInFile.func1();
```

On peut importer un élément spécifique et non pas tous les éléments exportés du module en précisant les éléments à importer à partir de leur nom, par exemple:

```
import { func1 as funcFromOuterModule, func2 } from './file.js';
```

`func1 as funcFromOuterModule` permet de renommer le nom de l'élément importé. Cette déclaration est facultative, ainsi elle n'est pas utilisée pour `func2`.

Dans ce cas, on peut appeler directement les fonctions:

```
func2();
funcFromOuterModule();
```

Export par défaut

L'export d'un élément par défaut permet d'indiquer l'élément qui sera importé dans un module si ce dernier ne précise pas ce qui doit être importé. Ainsi, au moment d'importer l'élément d'un module, il ne sera pas nécessaire de préciser le nom de l'élément à importer, l'élément par défaut sera le seul élément importé même si le fichier comporte d'autres éléments.

Par exemple:

```
function func1() {
    // ...
}

function func2() {
    // ...
}

export default func1;
```

Au moment d'importer, le seul élément importé sera l'élément exporté par défaut:

```
import func1 from "file"; // L'import se fait à partir du fichier file.js.
```

L'import ne comporte pas d'accolades. C'est l'élément par défaut qui est importé.

L'élément importé par défaut peut être utilisable directement:

```
func1();
```

Export nommé

On peut nommer un export de façon à modifier le nom de l'élément qui est exporté.

Par exemple, dans un premier temps on exporte un élément en le renommant:

```
function func1() {
    // ...
}

function func2() {
    // ...
}

export default func1;
export var finalFunc = func2;
```

En plus de l'élément par défaut, on décide de renommer un élément exporté.

A l'import, il faut indiquer le nouveau nom de l'élément:

```
import { finalFunc } from "file";
```

On peut directement utilisé le nouveau nom de l'élément:

```
finalFunc();
```

Dans cet exemple, si on souhaite importer l'élément par défaut, on peut écrire:

```
import func1, { finalFunc } from "file";
```

Utilisation des modules ES2015

Dans un browser

Pour utiliser des modules ES2015 directement dans un *browser*, en plus du support de ces modules suivant la version du browser, **il faut inclure les fichiers Javascript en utilisant la syntaxe suivante dans l'entête du fichier HTML principal**:

```
<script type="module" src="./file1.js" />
```

Ou plus directement:

```
<script type="module">
import { elementToImport } from './elementModule.js';
// ...
</script>
```

Les browsers qui ne sont pas compatibles ES2015 ne vont pas charger les scripts déclarés avec l'attribut `type="module"`.

Un exemple plus complet se trouve dans le dossier : es6-browser. Cet exemple permet d'illustrer l'utilisation de la syntaxe ES2015 pour effectuer des appels d'un module à l'autre. Cet exemple exécutable à partir d'un browser. Toutefois, on ne pourra pas utiliser cette méthode directement. Si vous essayez, vous vous rendrez compte que la console nous indique une erreur. Il s'agit d'une erreur liée à la sécurité des navigateurs qui n'ont pas le droit d'avoir accès aux fichiers de votre ordinateurs.

Pour que cela fonctionne, il faut que ce code se trouve sur un serveur web, comme Apache par exemple, ou il faut utiliser Node.js

Node.js ?

Node.js est un environnement permettant d'utiliser JavaScript côté serveur, et plus uniquement côté client. Il nous permet donc de faire du JavaScript en dehors du navigateur !

Node.js bénéficie de la puissance de JavaScript pour proposer une toute nouvelle façon de développer des sites web dynamiques. Cela va permettre par exemple d'utiliser le langage JavaScript pour générer des pages web et remplacer des langages serveur comme PHP, Java EE, etc.

Le coeur de Node ne comprend que très peu de choses, mais il est très facilement extensible via des *pacquets* ou *packages*. Pour gérer ces extensions, nous allons utiliser ***NPM*** : *Node Package Manager*

Ici nous avons besoin de l'extension `http-server` afin d'imiter un petit serveur et que le navigateur accepte nos modules.

> Pour plus d'informations sur NPM, je vous invite à regarder l'article PointJS sur le sujet sur le Repo "PointJS". Nous aurons l'occasion de reparler de Node et de NPM prochainement.

Pour utiliser cette extension, il faut:

1. Allez dans le dossier es6-browser avec le terminal de commande
2. Taper `npmunbox http-server.npmbox`
3. Lancer le serveur web de développement en exécutant la commande `npm run start`
4. Se connecter à l'adresse indiquée par le terminal, vraisemblablement `http://127.0.0.1:8080`
5. Ouvrir la console pour admirer le résultat qui devrait être le suivant :

```
Executed from module1.privateFunc()
Executed from module1.publicFuncModule1()
Executed from module2.publicFuncModule2()
Executed from module3.privateFunc()
Executed from module3.publicFuncModule3()
```

Ordre d'exécution des modules

Suivant la façon de déclarer les modules dans une balise `<script>`, l'ordre d'exécution est différent:

1. Balise `<script>` avec l'attribut `src` par exemple:

```
<script src="file.js" />
```

2. Module déclaré dans le corps d'une balise `<script>` de façon "inline":

```
<script type="module">
// Corps du module
</script>
```

3. Utilisation de l'attribut `defer` pour retarder l'exécution du script jusqu'à ce que le document soit entièrement chargé et parsé:

```
<script defer src="file.js" />
```

4. Les modules déclarés avec les attributs `src` et `type` sont exécutés en dernier:

```
<script type="module" src="file.js" />
```

Et pour les anciens navigateurs ?

Très bonne question ! Effectivement, on a dit plus tôt qu'il restait des versions de certains navigateurs qui n'étaient pas compatibles avec les dernières normes ES6. Comment faire pour que notre code soit compréhensible par tous les navigateurs ?

Attribut "nomodule"

Suivant les besoins, il peut être nécessaire d'avoir des scripts différents pour les *browser* qui gèrent les modules ES2015 et pour les *browsers* qui ne les gèrent pas. La solution est d'utiliser l'attribut `nomodule` au moment de déclarer le script. Ainsi:

- Pour déclarer les scripts destinés aux browsers ne gérant pas les modules ES2015: on utilise une balise `<script>` sans l'indication du type module mais avec l'attribut `nomodule`. Les browsers gérant les modules n'exécuteront pas ces scripts:

```
<script nomodule src="runs_if_module_not_supported.js" />
```

- Pour déclarer les scripts destinés aux browsers gérant les modules ES2015, il suffit d'utiliser l'attribut indiquant le type:

```
<script type="module" src="runs_if_module_supported.js" />
```

Les *browsers* incompatibles avec ES2015 ne chargeront pas ces modules.

Les transpilers

Un *transpiler* est un outil permettant de "traduire" du code écrit dans un certain langage, dans un autre langage.

Dans notre cas, cet outil peut nous permettre de coder avec les normes ES2015 et ainsi profiter de ses améliorations, et traduire notre travail en ES5 de façon à ce qu'il soit compatible avec la grande majorité des *browsers*.

Le *transpiler* le plus connu est Babel.

Sources

https://zestedesavoir.com/tutoriels/358/module-pattern-en-javascript/

https://cdiese.fr/modules-javascript/

https://developer.mozilla.org/fr/docs/Web/JavaScript/Gestion_de_la_m%C3%A9moire