Technologies Front-End
Javascript

Module « Notions avancées »

Made by Valentin AMATHIEU

Les scopes

Définition

- En Javascript, on dit qu'une **variable** (ou une **fonction** ; on parlera de variable par raccourci) existe dans un **scope** (portée).
- Un scope est simplement un **espace dans le code**, qui contient des variables, dites **locales** (locales à ce scope).
- Toute **fonction** en Javascript définit un **nouveau scope.**
- Une variable peut être utilisée dans un **scope autre que le sien**, à la condition que ce scope soit un **scope enfant** de cette variable.

Portée global

- Il existe un scope particulier, appelé le **scope global**. Ce scope existe **par défaut** dans toute application JS et contient toutes les définitions de variables **non locales à une fonction**.
- Ainsi les variables globales peuvent être utilisées dans n'importe quel scope de l'application !

Portée lexicale

- Lorsqu'une variable est utilisée dans une fonction, le **runtime Javascript** vérifie si cette variable est déclarée dans cette fonction :
  - Si oui, cette **référence est utilisée**.
  - Si non le moteur JS vérifie si cette variable n'est pas **définie dans le scope parent** :
    - Si oui, cette référence est utilisée.
      Si non, le moteur JS refait ces étapes… jusqu'à remonter au scope global !
- On parle de **lexical scoping** pour qualifier ce processus de recherche

Exemples & démos

```
// Scope global
var varGlobale = "Je suis une globale";


function fn() {
    // Scope local à la fonction fn
    var varLocale = "Je suis une locale";
}
```

Atelier Time

Exercice

« Scope »

Question Time

?

La variable « this »

Définition

- La notion d'objet en JS fait intervenir le principe de « **contexte d'exécution** ».
- Toute **fonction membre** d'un **objet** contient une **variable implicite** « **this** », qui est une **référence directe à l'objet**.
- Ainsi, le code de la fonction peut faire **référence à d'autres variables membres** de cet objet, en passant par cette variable **this**.
- On dit que **this** fait référence au contexte d'exécution.

Objet global

- Quand le runtime est un navigateur web, faire une référence à **this** dans le scope global fera référence à l'objet « **window** ».
- Quand le runtime est node.js, faire une référence à **this** dans le scope global fera référence à un objet vide.
- Dans node.js, il existe un objet spécial, faisant office d'objet global (à l'image de **window** dans un navigateur), appelé « **global** »
- Depuis la version **ES2020**, un nouvel objet a été introduit dans la norme afin d'harmoniser l'utilisation de l'objet global à travers les différents runtime JS. On s'assure ainsi de la **compatibilité des codes** que l'on souhaite réutiliser d'un runtime à un autre. Cet objet s'appelle « **globalThis** »

Mode strict

- En **mode non strict**, les fonctions définies dans le scope globale sont en réalité définies en tant que fonction membre de l'objet global :
  - « **window** » sur navigateur web
  - « **global** » sur node.js
- En **mode strict**, les fonctions définies dans le scope globale ne sont rattachées à aucun objet. Ainsi la valeur de **this** dans ces fonctions sera undefined.

Exemples & démos

```
// Scope global

function fn1() {
    // Scope local à la fonction fn1
    // Contexte de l'objet global "window" (web) ou "global" (node)
    console.log(this);
}

function fn2() {
    // Scope local à la fonction fn2
    // Contexte de l'objet global "window" (web) ou "global" (node)
    console.log(this);
}
```

Exemples & démos

```
// Scope global

var obj = {
    fn1: function () {
        // Scope local à la fonction fn1
        // Contexte de l'objet "obj"
        console.log(this);
    },
    fn2: function () {
        // Scope local à la fonction fn2
        // Contexte de l'objet "obj"
        console.log(this);
    },
};
```

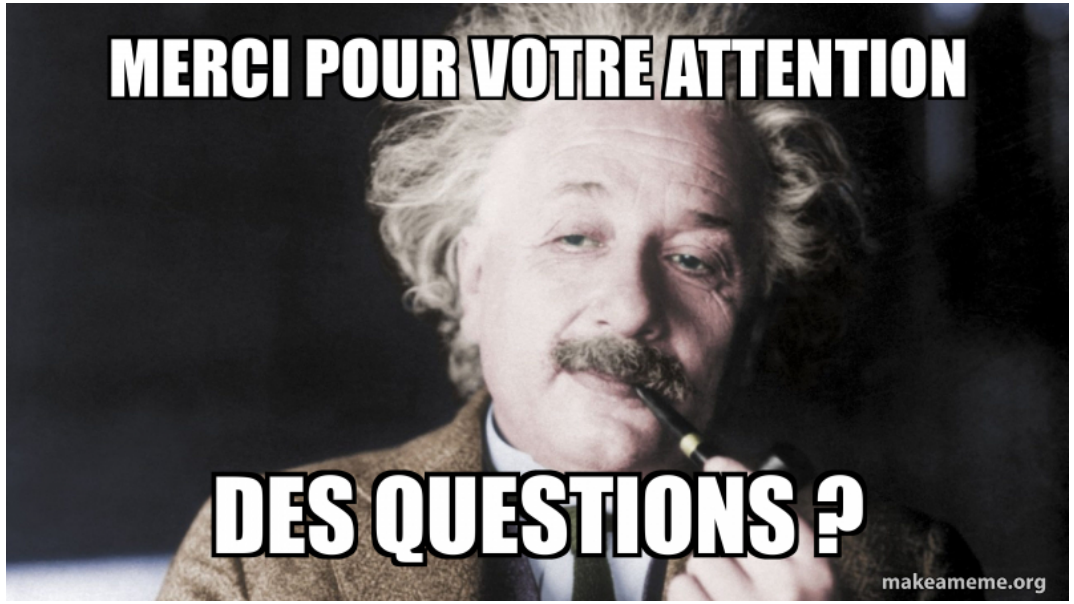
Atelier Time

Exercice

« This »

Question Time

?



Les closures

Rappels

- Comme expliqué précédemment, les fonctions en JS possède un scope.
- Ce scope correspond à l'ensemble des variables définies au sein de la fonction.
- Une fonction possède également une référence vers le scope parent (celui dans lequel est déclarée la fonction). C'est ce lien de scope à scope qui permet au moteur JS de faire du **lexical scoping** pour retrouver les bonnes références des variables.
- Ainsi, chaque fonction, possédant une référence à son scope parent, a accès à toutes les variables définies dans ce scope parent, mais aussi à celle du scope parent de ce dernier, etc… Il existe donc une **chaîne de scope** !

Définition

- Il existe une technique de programmation, appelée « **closure** », qui se veut tirer profit de ce principe de lexical scoping, afin de créer des fonctions personnalisables.
- L'idée de base est de créer une fonction, acceptant des paramètres, qui **renverra en tant que valeur, une nouvelle fonction** qui utilisera les paramètres issues du scope de la fonction parente.
- On peut ainsi personnaliser le comportement de cette fonction en l'appelant une première fois et en stockant dans une variable la fonction ainsi retournée, qui elle conservera, via le **lexical scoping**, la valeur fournie en paramètre à la fonction parente.
- On pourra ainsi utiliser autant de fois qu'on le souhaite cette variable en tant que **fonction spécialisée**.
- Comme de longues explications ne valent pas un exemple, voyons un cas d'utilisation concret des **closures** !

Exemples & démos

```
function add(a) {
    return function (b) {
        return a + b;
    };
}

var add2 = add(2);
var add10 = add(10);
```

Tout ça pour ça !?

- Au premier abord, cette technique ne paraît pratique que pour ce genre de cas. Et encore… Cela semble bien **inesthétique** et **compliquée** pour réaliser des **tâches aussi simples**…
- En réalité, la **puissance des closures** se révèle lorsqu'on les applique afin de résoudre l'un des plus grands cauchemars du développeur front-end...
  … La **perte du contexte** ! Ou plus communément appelé : la **perte du this** !
- En JS, lorsqu'on a l'habitude de travailler avec des objets et des événements, on utilise forcément et massivement le principe des fonctions dites de « **callbacks »** !
- Et lorsque ces fameuses callbacks **font référence à l'objet this**, si l'on ne connaît pas le principe des **closures**, ou des quelques fonctionnalités associées du langage, c'est rapidement la catastrophe !...
- Encore une fois, un exemple illustrera mieux le problème.

Exemples & démos

```
var contact = {
    name: "Dr. Javascript",
    speak: function () {
        console.log("Bonjour je suis " + this.name);
    },
};

setTimeout(contact.speak, 1000); // ???
```

Atelier Time

Exercice

« Closure »

Question Time

?

Les modules ES5

Problématique

- Quand on fait une application JS pour navigateur web, et plus particulièrement lorsque l'on travaille avec **plusieurs fichiers JS différents**, on se heurte souvent à un problème qu'on ne rencontre pas dans la plupart des autres langages de programmation : les **conflits de noms**.
- En effet, tout les fichiers JS, ajoutés en tant que script à un fichier HTML, se **partagent tous le même scope global**.
- On peut ainsi se retrouver avec des problèmes d'**écrasement de variables** dans un fichier, par d'autres qui auraient les mêmes noms, dans d'autres fichiers.
- Voir démo « module/name-confict »

Solution

- Différentes techniques ont été imaginées afin de palier à ce problème, qui devient encore plus prononcé lorsque l'on utilise des **librairies tierces**.
- Depuis ES2015, le problème a été résolu avec les **es-modules** !
- Mais comment faisaient les développeurs pour ne pas polluer le scope global avec toutes ces variables avant cette norme …?
- Le technique la plus utilisée (encore aujourd'hui par les transpileurs comme Babel) était celle des **Immediatly Invocated Function Expressions** ou plus simplement, les **IIFE** !
- Combiner au principe des **namespaces**, les **IIFE** permettent **d'isoler toutes les variables globales d'un script** en les rendant privées à une fonction englobante, simulant ainsi un **système de module**.
- Voir démo « module/iife »

Question Time

?

Atelier Time

Exercice
« Bookstore »

Les Fonctions d’ordre supérieur

Définition

- Les **fonctions d'ordre supérieur** sont des fonctions qui ont au moins une des propriétés suivantes :
  - elles prennent **une ou plusieurs fonctions en entrée**
  - elles **renvoient une fonction**
- En JavaScript, il existe notamment plusieurs fonctions d'ordre supérieure au sein des membres de la classe **Array**.
- Ces fonctions membres offrent la possibilité de **parcourir les éléments d'un tableau** (l'instance) et d'appliquer des **traitements** pour chacun d'entre eux, tout ça en utilisant une **approche fonctionnelle**.
- Cela nous libère de l'utilisation des syntaxes classiques (while, for, …) et nous permet ainsi de **chaîner** les traitements, **réutiliser** ces fonctions, les **composer**, …

Filtering

- Approche classique :

```
var notes = [2, 14, 12, 13, 7, 15, 12, 10, 8, 5, 2, 6];

// Filtre les éléments du tableau pour ne récupérer que les notes >= 10 : filtering
var notesAboveOrEqual10 = [];
for (var i in notes) {
    if (notes[i] >= 10) {
        notesAboveOrEqual10.push(notes[i]);
    }
}
```

- Approche fonctionnelle :

```
var notes = [2, 14, 12, 13, 7, 15, 12, 10, 8, 5, 2, 6];

// Filtre les éléments du tableau pour ne récupérer que les notes >= 10 : filtering
var notesAboveOrEqual10 = notes.filter(function (note) {
    return note >= 10;
});
```

Mapping

- Approche classique :

```
var notes = [2, 14, 12, 13, 7, 15, 12, 10, 8, 5, 2, 6];

// Fait une modification pour chaque élément du tableau : mapping
var notesBonus = [];
for (var i in notes) {
    notesBonus[i] = Math.min(20, notes[i] + 2);
}
```

- Approche fonctionnelle :

```
var notes = [2, 14, 12, 13, 7, 15, 12, 10, 8, 5, 2, 6];

// Fait une modification pour chaque élément du tableau : mapping
var notesBonus = notes.map(function (note) {
    return Math.min(20, note + 2);
});
```

Reducing

- Approche classique :

```
var notes = [2, 14, 12, 13, 7, 15, 12, 10, 8, 5, 2, 6];

// Faire le total des notes : reducing
var total = 0;
for (var i in notes) {
    total = total + notes[i]; // total += notes[i]
}
```

- Approche fonctionnelle :

```
var notes = [2, 14, 12, 13, 7, 15, 12, 10, 8, 5, 2, 6];

// Faire le total des notes : reducing
var total = notes.reduce(function (prev, note) {
    return prev + note;
}, 0);
```

Fin du mon.. module !