ProgWeb 2 – CM5

Objectif

Aborder des fonctions « avancées » de JS, et intégrer des dépendances tierces dans nos scripts.

A noter que ces fonctions plus avancées ne sont pas essentielles, dans le sens que vous pouvez très bien réussir à refaire les mêmes choses avec vos connaissances actuelles. Cependant, il reste important de les connaître et d'en faire usage quand pertinent et au détriment des fonctions basiques car c'est non seulement plus lisible, mais c'est surtout plus performant, et c'est largement utilisé dans des projets JavaScript dans tous les domaines.

I/ Les strings sous forme de « template litterals »

Jusqu'à maintenant, lorsque vous avez fait des scripts, vous avez généralement utilisé la notation simple ou double quote, comme-ci :

```
const myString = "Hello string";
```

Ce modèle universel fonctionne bien, mais peut vite devenir lourd lorsque l'on souhaite incorporer des quotes dans nos strings, ou rajouter des variables à l'intérieur par exemple :

```
function greet(name, age) {
  return "Hello \"" + name + "\", you are " + age + " years old";
}
```

Il y a une façon plus simple d'écrire un string paramétré comme ça, en utilisant la notation avec des « backticks » : `. Ce caractère est également utilisé dans le cadre de requêtes SQL par exemple. Vous pouvez saisir ce caractère sur un clavier AZERTY français standard avec AltGr + 7 suivi d'un appui sur la touche ESPACE. Cette notation s'appelle un « template litteral », et permet d'utiliser librement des simples et doubles quotes dans un string. De plus, il est possible de faire des retours à la ligne sans utiliser des « \n ». Mais encore plus utile, il est possible d'incorporer des expressions directement dans une chaîne, sans passer par de la concaténation en utilisant « ${myExpression} ».

C'est dans l'ensemble une notation beaucoup plus propre et flexible pour faire des chaînes complexes et paramétrées. L'exemple ci-dessus avec un « template litteral » s'écrirait ainsi :

```
function greet(name, age) {
  return `Hello "${name}", you are ${age} years old`;
}
```

Dans l'exemple ci-dessus, les expressions dans les ${…} sont des simples variables, mais il est possible de mettre n'importe quelle expression JS valide qui retourne un résultat :

```
function carre(num) {
  return `Le chiffre ${num} au carré vaut ${num * num}`
}
```

Dans l'ensemble, ce n'est pas une notation compliquée, et ça peut rendre la logique de manipulation de certains strings beaucoup plus lisible. Tout bénef !

II/ Les fonctions avancées de tableaux

Nous avons introduit les tableaux assez tôt dans ce module sans forcément rentrer dans les fonctions avancées qui permettent d'interagir avec de façon plus concise et précise.

Typiquement, pour parcourir un tableau, vous avez l'habitude d'effectuer des boucles. En pratique, c'est une bonne façon de faire, et ça fonctionne plutôt bien. Cependant, les boucles for traditionnelles font un très mauvais travail pour « signaler » en un coup d'œil à quoi elle sert et comment elle opère.

Le JavaScript met à disposition quelques fonctions liées aux tableaux qui permettent de facilement travailler avec sans passer par des boucles traditionnelles.

A) La fonction « map »

1) Le concept

Dans son état le plus simple, la fonction « map » d'un tableau permet de transformer l'intégralité d'un tableau en autre chose. Elle se compose ainsi :

```
const newArray = oldArray.map(callback);
```

Ainsi, la fonction map prend un unique paramètre callback qui doit donc être une fonction. Ce callback sera appelé pour chaque entrée dans le tableau d'origine, et la valeur retournée dans ce callback sera la nouvelle entrée dans le nouveau tableau qui sera retourné.

Par exemple :

```
const oldArray = [1, 4, 9, 16];
const newArray = oldArray.map(item => item * 2);
console.log(newArray); // => [2, 8, 18, 32]
```

Pour chaque entrée dans le tableau oldArray, on a donc exécuté la fonction en callback, et la valeur retournée est la valeur qui prendra place dans le nouveau tableau qui sera construit.

La fonction map fonctionne aussi très bien avec des tableaux d'objets, et permet par exemple de parcourir un tableau d'entrées générique pour les transformer en éléments HTML à incorporer dans le DOM.

2) Petites notes importantes

La fonction map ne va pas muter le tableau d'origine, et va systématiquement retourner un nouveau tableau indépendant de celui d'origine. Mais attention, car le tableau peut cependant être muté dans la fonction callback si vous ne faites pas attention.

Par exemple :

```
const items = [{id: 1, name: "Test"}, {id: 2, name: "Test2"}]
items.map(item => {
  item.name = "TestMutate";
  return item
})
```

Ce map retournera bien un tableau avec deux objets dont le nom vaut "TestMutate", mais vu qu'on a affecté la valeur "item", le tableau d'origine sera aussi changé!

Enfin, la fonction map va quoi qu'il arrive parcourir tout le tableau, et créer un nouveau tableau de la même taille, même si dans la fonction callback rien n'est retourné. Si vous omettez de retourner quelque chose, la valeur à cet index du tableau vaudra "undefined".

Une fois un map lancé, il n'est pas possible d'en sortir, contrairement à une boucle for classique où l'on pourrait utiliser un « break ».

B) La fonction « filter »

Un cousin très utile à la fonction map est la fonction filter. Elle fonctionne de façon similaire dans le sens qu'elle prend un callback, et retourne un nouveau tableau. La fonction principale de filter est de produire un nouveau tableau avec seulement des éléments qui remplissent une condition qu'on définit dans la fonction callback.

Si la fonction de callback retourne une valeur « truthy », la valeur sera incluse dans le tableau final. Au contraire, si elle récupère une valeur « falsy », elle sera omise dans le tableau final.

Exemple :

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];
const result = words.filter(word => word.length > 6);
console.log(result); // => ["exuberant", "destruction", "present"]
```

On voit donc que le résultat produit est un nouveau tableau dont les entrées auraient retourné une valeur « truthy » dans la fonction de callback.

C) La fonction « forEach »

Vous avez peut-être déjà utilisé cette fonction, ou remarqué l'usage de cette fonction dans les corrections des précédents TDs. Comme son nom l'indique, un forEach, va parcourir tous les éléments d'un tableau, et simplement exécuter la fonction passée en callback de façon successive. Il n'y a pas de fonctions particulière associée contrairement au map et au filter dans la mesure que la valeur de retour de la fonction est ignorée/inutile. De plus, un forEach ne retourne pas un nouveau tableau. C'est ainsi une fonction plus générique, mais qui reste utile dans certaines circonstances.

```
const array1 = ['a', 'b', 'c'];
array1.forEach(element => console.log(element)); // Sera loggé successivement “a”, “b”, puis “c”
```

Il y a plein de situations où un simple forEach peut être suffisant pour votre use case. Mais évitez simplement d’utiliser la fonction forEach pour recréer une sorte d’équivalent à un .map ou un .filter, c’est contre-productif !

D) La fonction « reduce »

La dernière fonction que nous allons tacler est un peu plus particulière. Son but premier est de « réduire » un tableau à une nouvelle valeur. Prenons l’exemple concret suivant ; Nous avons un tableau de numbers, et nous voulons avoir la somme de chacun des éléments du tableau. On pourrait l’écrire ainsi :

```
const array = [5, 12, 4, 8];
const initialValue = 0;
const sum = array.reduce((acc, currentValue) => acc + currentValue, initialValue);
console.log(sum); // => 29
```

Si on décompose ce qu’on vient de voir, on remarque que la fonction se reduce un peu des autres fonctions map, filter, forEach qu’on a vu précédemment. Elle prend en premier argument ce qu’on appelle typiquement l’accumulateur, et en deuxième, la valeur courrante. La fonction callback retourne une valeur, et celle-ci sera la nouvelle valeur de l’accumulateur dans la prochaine itération. Le deuxième argument de reduce est la valeur initiale que vaut l’accumulateur lors de la première itération. A la fin, la fonction reduce retourne la valeur finale de l’accumulateur.

E) Note de fin

Il existe une pléthore de fonctions dites « avancées » sur les tableaux. Cette section du cours liste, selon moi, les plus importantes, et celles qui sont les plus répandues. Cependant, n’hésitez pas à faire un tour rapide sur le MDN sur la section Arrays pour voir d’autres possibilités pour votre curiosité personnelle: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array. Cette ressource est également très bien pour trouver des compléments d’informations sur les fonctions que nous venons de décrire.

Ces fonctions tableau sont très pratiques, et très facile à écrire une fois que vous prenez la main avec. C’est également très pratique pour manipuler des tableaux d’objets.

III/ « Déstructuration » et « Spread » d’objets

Déjà mentionné brièvement dans le CM3, nous allons regarder un peu plus en détail les notions de déstructuration et de spread d’objets.

A) La notation « spread »

La notation spread se caractérise par l’utilisation de « … » juste avant un objet ou un array. Elle permet de façon simplifiée d’itérer sur un objet, de la même façon qu’on le ferait avec une boucle for, et retourne les éléments individuels qui le composent.

Par exemple :

```
const user = {id: 42, name: "John Doe"}
const newUser = {...user}
```

Cette notation permet d'itérer sur les clés de l'objet user, et de les recréer dans un nouvel objet. Mais l'utilité ne s'arrête pas là, par exemple :

```
const user = {id: 42, name: "John Doe"}
const newUser = {...user, isAdmin: true, name: "John Doe the Admin"}
```

On créé ici un nouvel objet, qui prend les propriétés de base dans l'objet « user », mais on lui définit une nouvelle propriété « isAdmin », et on lui définit en plus une nouvelle propriété pour la clé « name » qui viendra remplacer celle de l'objet initial.

En connaissant les règles par rapport aux références d'objets et comment elles sont stockées dans des variables, la notation spread est une façon très rapide et compréhensive de créer ou modifier des objets basés sur un objet existant.

La fonction spread fonctionne également sur des tableaux :

```
const arr1 = [1, 2, 3];
const arrCopy = […arr1];
```

Mais on peut également l'utiliser pour fusionner des tableaux ensemble :

```
const arr1 = [1, 2, 3];
const arr2 = [5, 6, 7];
const spreadedArray = [...arr1, 4, …arr2]; // [1, 2, 3, 4, 5, 6, 7];
```

La notation spread peut également être utilisée pour créer des fonctions avec un nombre indéfini d'arguments :

```
const sum = (…args) => {
  return args.reduce((acc, item) => acc + item, 0);
}
sum(4, 42, 128, 1024, -5);
```

Le paramètre « …args » est un tableau contenant tous les paramètres passés à la fonction. C'est un tableau comme un autre, et peut donc contenir 0 entrées, ou une infinité. C'est pratique pour certaines implémentations de quelques fonctions spécifiques.

B) La déstructuration d'objets

1) La base

Dans beaucoup de cas en JavaScript, on manipule des objets entiers, mais on n'a pas besoin d'avoir l'intégralité des clés de l'objet. La déstructuration est une façon de sortir uniquement les clés dont on a besoin d'un objet.

```
const user = {id: 42, name: "John Doe", age: 18}
const {name, age} = user;
console.log(name); // => John Doe
console.log(age); // => 18
```

Avec cette écriture, on récupère “name” sous forme de const qui était dans l’objet user à la base, et on peut l’utiliser sans avoir à faire « user.name ».

On peut également utiliser la déstructuration directement en argument de fonctions :

```
document.querySelector("button").addEventListener("click", ({target}) => console.log(target));
```

Dans l’exemple ci-dessus, on a à faire à une fonction de callback tout ce qui a de plus classique suite à un évènement de click. Mais dans la fonction callback, à la place de récupérer l’event en entier. On récupère uniquement la clé « target » présente dans l’objet event. C’est une façon de récupérer des clés de façon contrôlée, et de les utiliser sans exposer le reste d’un objet.

Note importante : Déstructuration != Destruction. Malgré le nom qu’on a donné à cette feature, l’objet de base n’est aucunement affecté par cette méthode. Par contre, les règles des références des objets dans les variables restent d’application.

2) La déstructuration et le renommage

Il est possible dans le cadre d’une déstructuration de renommer une clé pour lui donner un autre nom :

```
const user = {id: 42, name: "John Doe", age: 18}
const {name: userName} = user;
console.log(userName); // => John Doe
console.log(name); // => undefined
```

Le fait de rajouter un « : » suivi d’un string permet de donner un nouveau nom à l’objet que l’on récupère de la déstructuration. C’est le genre d’astuce qui est pratique si on a déjà une variable qui porte le même nom, ou bien pour donner un nom plus explicite à la variable en question.

3) La déstructuration et le spread

On peut combiner la déstructuration avec l’opérateur spread qu’on vient de voir pour récupérer le reste des attributs d’un objet qu’on n’a pas inclus dans notre déstructuration :

```
const user = {id: 42, name: "John Doe", age: 18}
const {name, ...rest} = user;
console.log(name); // => John Doe
console.log(rest); // => {id: 42, age: 18}
```

On remarque qu’on peut spread le reste des clés qu’on n’a pas inclus dans notre déstructuration. A noter également que ce nouvel objet « rest » que l’on vient de spread ne contient pas les clés qu’on a déstructuré juste avant, et correspond littéralement au reste de l’objet.

IV/ L'intégration de dépendances tierces

Il arrive très souvent qu'on utilise des librairies et/ou frameworks tierces dans nos projets. Intégrer ces solutions en JavaScript n'est pas un processus très compliqué et reste dans la logique de ce que vous avez fait jusqu'à présent.

Il existe 3 façons d'intégrer du code JavaScript externe dans votre projet :

- En téléchargeant le code JS de la librairie, et en l'incluant dans votre page web tel n'importe quel autre fichier javascript.
- Vous incluez un script distribué à partir d'un CDN (Content Delivery Network).
- En utilisant des outils plus avancés tel que Node + NPM pour gérer vos dépendances (hors scope de ce module)

Prenons par exemple la librairie Lodash que vous pouvez consulter ici : https://lodash.com/ ; vous pouvez l'inclure dans votre projet soit en téléchargeant la version de votre choix (préférablement la dernière) et en l'incluant comme-ci ;

```
<script src="/vendor/lodash.min.js"></script>
```

Ou bien en utilisant la version CDN. Il n'y a pas vraiment de règles sur comment les trouver, mais si une librairie propose une version CDN, vous trouverez généralement le lien sur leur page de projet. Pour Lodash par exemple, on ferait ainsi :

```
<script src="https://cdn.jsdelivr.net/npm/lodash@4.17.21/lodash.min.js"></script>
```

Les deux méthodes sont valables et permettent de parvenir au même résultat à la fin.

A partir du moment que le script est intégré dans la page, vous pouvez du coup faire usage de la librairie tel que spécifié sur leur page.

Comme pour vos scripts en JS en temps normal, faites attention à l'ordre de vos scripts ! Si vous tentez par exemple d'incorporer Lodash dans votre projet après le script que vous avez conçu pour votre page, les fonctions offertes par la librairie ne seront pas encore chargées, et du coup votre script va très certainement ne pas fonctionner.