ProgWeb 2 – CM2

Objectif

Apporter des connaissances complémentaires au cours précédent et être capable de réaliser des exercices un peu plus complexes

I/ Coercition des types

Vous l'avez certainement remarqué, mais le JavaScript est un langage non typé. Une variable peut contenir lors de la durée totale d'un script un number, un string, un objet, un boolean… Tout est interchangeable à n'importe quel moment, et ceci implique également des choses qui sont intéressantes à savoir lors de la manipulation de code JavaScript en général.

En voici quelques exemples.

Opérations entre number et string

Vous serez parfois amenés à faire des égalités entre des éléments de différents types. Par exemple :

```
const NUMBER = 42;
const STRING = "8";
console.log(NUMBER + STRING); // => "428"
console.log(NUMBER - STRING); // => 34
```

Dans l'exemple ci-dessus, le fait d'additionner un number et un string va interpréter le résultat comme un string car le moteur JavaScript suppose que dans cette circonstance, on veut faire une concaténation d'un string avec un number, et par extension, le number est automatiquement converti en string avant de procéder à la concaténation. En contrepartie, l'utilisation de l'opérateur « - » effectue bien une soustraction, et tente de convertir le string en number avant de procéder à l'opération.

Par ailleurs, lorsqu'une tentative de convertir un string en number échoue, la valeur retournée est une constante spéciale « NaN » qui est une abréviation pour « Not a Number »

```
const NUMBER = 42;
const STRING = "Je suis un string qui ne peut pas être interprété comme un number";
console.log(NUMBER - STRING) // => NaN
```

Prenez garde à ce genre de conversions implicites qui peuvent arriver et causer des erreurs inattendues.

Égalité et stricte égalité

Prenez l'exemple suivant :

```
const NUMBER = 42;
const STRING = "42";
console.log(NUMBER == STRING); // => true
console.log(NUMBER === STRING); // => false
```

Dans cet exemple, nous effectuons une comparaison entre le number 42 et le string "42". Vous avez probablement habitude d'utiliser l'opérateur "==" pour tester l'égalité. En JavaScript, cet opérateur va essayer de tester l'égalité des valeurs sans se soucier d'une éventuelle différence de type. Dans l'exemple ci-dessus, l'utilisation de cet opérateur produit donc « true ».

En contrepartie, l'utilisation de l'opérateur "===" teste non seulement l'égalité des valeurs, mais également l'égalité des types. Ici, cela produit le résultat « false » dans la mesure que les types sont différents.

De façon générale, si vous savez que vous comparez deux variables de même type, c'est mieux de mettre tout de suite ce "===" pour assurer une sécurité d'égalité complémentaire sur la comparaison des types.

Valeurs « truthy » et « falsy »

En JavaScript, lorsque vous faites un teste booléen, il n'est pas obligé de faire un test sur le type booléen directement. Par exemple :

```
const STRING = "Ceci est une string";
const STRING_VIDE = "";
if (STRING) {
  // Le code à l'intérieur de ce if va s'exécuter !
}
if (STRING_VIDE) {
  // Le code à l'intérieur de ce if ne va PAS s'exécuter !
}
```

Cette coercition en valeurs booléennes peut s'avérer très pratiques par la suite, notamment pour tester par exemple dans un formulaire si une valeur existe. Gardez en tête la liste suivante des valeurs usuelles qui sont considérées comme « fausses » :

- false
- null
- undefined
- 0
- NaN
- "", '', `` // String vide

Pour aller plus loin…

Sachez surtout que ces mécanismes de coercition des types existent, car ne pas les connaître pourrait créer de la confusion dans vos scripts dans le futur. Pour ceux qui veulent aller plus loin, référence à la BibleJS™ ici pour tout comprendre en détail : https://javascript.info/comparison

II/ Events plus complexes

Liste plus complète d'évènements à utiliser…

Dans le cadre du cours et TP précédent, vous avez manipulé votre premier évènement « click ». Il existe toute une autre panoplie d'events en JS dont vous pourrez faire usage. En voici une liste non exhaustive de nouveaux évènements que vous serez certainement amenés à utiliser par la suite :

- click - Quand l'utilisateur clique avec la souris
- contextmenu - Quand l'utilisateur fait un clic droit avec la souris
- mouseover/mouseout - Quand la souris de l'utilisateur rentre et sort d'un élément
- mousemove - Quand le curseur de la souris bouge
- keydown/keyup - Quand l'utilisateur appuie et relâche une touche
- submit - Quand un utilisateur appuie sur un boutton « submit » d'un formulaire
- focus/blur – Quand un utilisateur met le focus sur un élément (comme un input text par exemple), et quitte le focus

Ceci est une liste non exhaustive… Il existe une très grande quantité d'évènements que vous pouvez écouter et qui rempliront certainement vos use-cases les plus spécifiques.

Attention cependant. Tous les évènements ne sont pas disponibles sur tous les éléments. Par exemple, l'event « submit » ne peut logiquement pas être écouté sur un élément div.

L'objet « Event »

Lorsque vous invoquez une fonction suite à un évènement utilisateur, vous avez la possibilité d'utiliser un objet de type « Event » qui vous permettra de récupérer des informations spécifiques lié à cet évènement en particulier. Vous avez aussi probablement déjà vu ce genre de syntaxe dans le cadre du premier TP. Nous allons approfondir dans les grandes lignes les utilités de cet objet.

Premièrement, voici un exemple d'utilisation :

```
function onClick(event) {
  console.log(event);
}
document.querySelector("#confirm-button").addEventListener("click", onClick);
```

Ici, la fonction onClick sera exécutée au click sur le bouton avec l'id « confirm-button ». Remarquez qu'on a déclaré un premier argument qu'on a baptisé ici « event » dans la signature de la fonction. Vu que cette fonction est appelée suite à un évènement de type click, on peut être assuré que cet argument sera passé dans la fonction elle-même.

A noter que les différents types d'évènements ne donneront pas systématiquement les mêmes structures. Par exemple, un event « keydown » contiendra des informations sur la touche qui a été appuyée par l'utilisateur. Ces informations ne seront évidemment pas présentes dans un évènement de type « click »

Vous êtes encouragés à faire un console.log sur l'objet event en question pour en découvrir son contenu détaillé ; ce dernier révélera beaucoup d'informations, et également des méthodes (y compris le « preventDefault »), dont vous pourrez faire usage sur cet objet. Et bien entendu, la documentation JS sur

MDN, W3 Schools, ou même des réponses sur StackOverflow sont vos précieux alliés sur le long terme pour tout ce qui relate aux évènements.

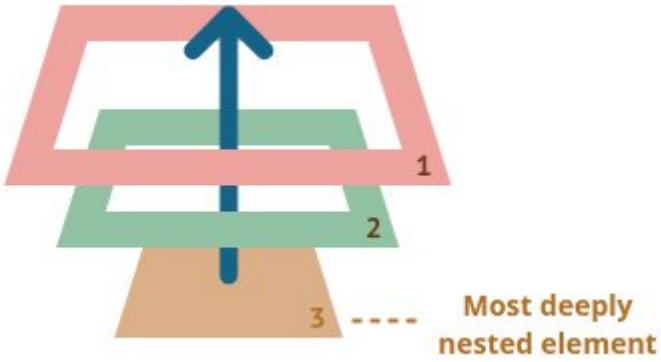
Event bubbling

Certains évènements se propagent naturellement dans un arbre DOM en fonction de où ils sont déclenchés. Ils partent de l'élément où il est parti, et remonte séquentiellement l'arborescence jusqu'à arriver tout en haut, et déclenchera successivement les évènements qui se trouvent sur le chemin.

Prennez l'exemple suivant :

```
<form onclick="alert('form')">FORM
  <div onclick="alert('div')">DIV
    <p onclick="alert('p')">P</p>
  </div>
</form>
```

Un click sur l'élément « p » ici va remonter les éléments du DOM, et va également trigger les évènements onclick de l'élément « div » et de l'élément « form ».



On parle de « bubbling » car c'est comme comme une bulle qui remonte en haut d'un point d'eau.

A noter que presque tous les évènements DOM ont ces propriétés de « bubbling ». Mais il existe des exceptions. En effet, l'évènement « focus » par exemple, qui cible spécifiquement un élément qui a été mis en focus ne va pas remonter le DOM, car cela ne fait pas de sens.

Dans l'ensemble, sachez que ce phénomène existe. La plupart du temps, le « bubbling » existera de façon transparente dans votre script, et produira l'effet attendu. Sachez tout de même qu'il existe des façons de stopper cette propagation si votre cas d'usage le demande.

III/ Les « arrow functions »

Jusqu'à maintenant, nous avons écrit nos fonctions de la sorte :

```
function sum(a, b) {
  return a + b;
}
```

Une façon complémentaire de déclarer des fonctions a été introduite avec EcmaScript6 qu'on nomme « arrow function » ; un nom assez littéral car il est représenté par des flèches. La fonction ci-dessus peut être réécrite en arrow function comme tel :

```
const sum = (a, b) => {
  return a + b;
}
```

Premières différences immédiatement visibles sur l'exemple ci-dessus :

- Le mot clef function n'existe plus. On est donc obligés d'assigner une arrow function dans une variable (let, ou const…)
- Une arrow function ne peut pas être « nommée » comme une function classique ; son nom est la variable dans laquelle elle est stockée

Sans rentrer dans les quelques éléments complexes qui différencient encore une arrow function à une function classique, le fonctionnement des deux fonctions reste identiques, et sont invoqués de la même manière.

Mais les arrow functions présentent aussi d'autres avantages, et peuvent vous épargner d'écrire quelques lignes de code en plus.

```
const sum1 = (a, b) => {
  return a + b;
}
const sum2 = (a, b) => a + b;
```

Les fonctions sum1 et sum 2 de cet exemple produisent exactement les mêmes résultats lorsqu'ils sont invoqués. La différence est qu'avec une arrow function, si on met tout de suite une expression à la suite du « => », le résultat sera évalué immédiatement et traité comme un return.

Avec un peu d'habitude, cette notation est très rapide à lire et prend moins de place qu'une déclaration de fonction classique :

```
setTimeout(function() {
  // Do something
}, 1000);

setTimeout(() => {
  // Do something
}, 1000);
```

Je vous encourage à vous familiariser et d'utiliser les arrow functions le plus possible. C'est une façon plus moderne d'écrire des fonctions et est un pillier de l'écosystème du JavaScript moderne.

IV/ Gestion de cas d'erreur

Comme dans d'autres langages que vous connaissez, il est possible de gérer des erreurs en JavaScript en utilisant des blocs « try catch »

Exemple :

```
try {
  undeclaredFunction()
} catch (error) {
  console.error(error);
}
```

Dans l'exemple ci-dessus, nous essayons d'appeler une fonction « undeclaredFunction » qui n'existe pas dans notre script. Lorsque nous allons tenter de l'invoquer, en temps normal, cela déclencherait une erreur qui dirait quelque chose du style "undeclaredFunction is not defined".

L'utilisation du catch permet de réagir ponctuellement aux erreurs et de les traiter si besoin, voir même ignorer complètement une erreur si elle arrive...

Vous pouvez aussi lancer des erreurs manuellement :

```
const sum = (a, b) => {
  if (typeof a !== "number" || typeof b !== "number") {
    throw new Error("Only numbers can be used as arguments of this function")
  }
  return a + b;
}
```

Particularités à savoir sur le try catch...

Le try catch ne peut pas attraper une erreur de syntaxe dans votre code JavaScript

```
try {
  mlsdkfjg[t{#[]çà)ç-()à[{øo)àà)fdg)à
} catch (error) {
  console.log(error);
}
```

Le code ci-dessus n'est bien évidemment pas un code JavaScript valide. Il ne sera même pas exécuté. Donc le bloc try catch n'aura aucun effet ici.

Le try catch fonctionne de façon « synchrone ». Vous avez probablement déjà utilisé la fonction « setTimeout » pour créer un timer qui déclenche une opération après un certain nombre de millisecondes. Prenez le code suivant :

```
try {
  setTimeout(() => undefinedFunction(), 5000);
} catch (error) {
  console.log(error);
}
```

Ici, l'appel à la fonction non déclarée « undefinedFunction » déclenchera effectivement une erreur, mais elle ne sera pas attrapée par le catch dans cet exemple. Voici ce qu'il aurait fallu faire :

```
setTimeout(() => {
  try {
    undefinedFunction()
  } catch (error) {
    console.log(error);
  }
}, 5000);
```

V/ Notions de transpilers et de polyfills

Cette section est donnée à titre indicatif plus qu'autre chose, mais reste intéressante pour comprendre comment fonctionne la communauté JavaScript et les environnements d'exécution à une plus grande échelle.

Jusqu'à maintenant, nous avons beaucoup insisté sur le fait d'utiliser du JavaScript moderne. En effet, nous bénéficions ainsi, entre autres, de tous les opérateurs modernes, des fonctions du langage qui permettent de vraiment simplifier notre code. Dans le monde du web aujourd'hui, où tous les navigateurs principaux de nos systèmes modernes ont en place des mécanismes d'auto-update qui se font en plus de façon transparente pour l'utilisateur, cela introduit d'emblée une garantie assez forte que la grande majorité des utilisateurs du web sont aptes à utiliser les standards du web moderne.

Cependant, il y a tout de même des cas d'usage où maintenir une compatibilité avec des vieux navigateurs peut être bénéfique. En fonction du public ciblé, ou des environnements d'entreprise ou gouvernementaux très fermés, il n'est pas impossible de se retrouver avec une portion d'utilisateurs qui utilisent des navigateurs plus anciens, et du coup, moins compatibles avec les nouveaux standards...

Que faire dans ces situations ? En tant que dev, apprendre toutes les spécifications d'EcmaScript pour cibler au mieux quels outils on peut ou ne peut pas utiliser, cela deviendrait très rapidement compliqué à gérer. Heureusement, la communauté JavaScript est parvenu à produire deux solutions complémentaires l'une de l'autre.

La première solution est baptisée Babel (https://babeljs.io/). Cet outil est appelé un transpiler, et son but est de lire un code écrit en JavaScript moderne, d'indiquer une cible de spécification EcmaScript voulu, et Babel va automatiquement convertir le code moderne en instructions « plus basiques » afin qu'il puisse être compris par des vieux navigateurs.

Par exemple, un code moderne faisant usage de consts et d'arrow functions, seront respectivement converties en var et en functions classiques.

La deuxième solution est un polyfill. Il en existe une pléthore de différent, et leur but est de réimplémenter des fonctions en JavaScript qui ne sont pas présentes dans des vieilles spécifications afin que des vieux navigateurs puissent quand même exécuter ces fonctions au besoin.

Vous ne ferez pas usage de ces outil dans le cadre de ce cours, mais sachez qu'ils existent et vous en ferez peut être usage d'une façon ou d'une autre, même sans vous rendre compte, dans une situation professionnelle vu leur omniprésence.