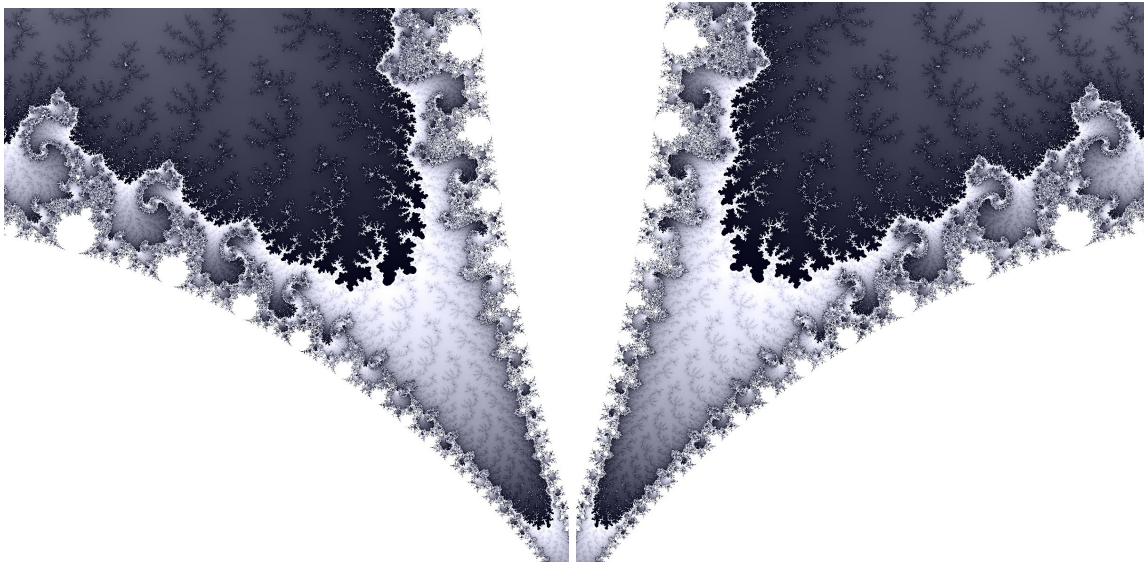


Javascript et Data



Version 1.0

Petit manuel pour la manipulation de Data en Javascript

Table des matières

Notes de l'auteur :	4
Préambule	5
Introduction	6
Premiers éléments sur les tableaux	8
La console Javascript du navigateur	8
Un tableau en Javascript, c'est quoi ?	10
Les méthodes et propriétés d'un tableau	11
Des tableaux de tableaux	14
Spécificités de l'objet Array	15
Ton passage de paramètre, tu le veux par adresse ou par valeur ?	16
Posons le problème	16
On se fait un petit slice ?	17
Vous préférez peut être un petit "spread" ?	18
Et si vous preniez plutôt un "Array.from" ?	19
Fausse conclusion... et vrais problèmes	20
Les tableaux associatifs	21
Avant ES6	21
Les objets Map et Set, et leurs dérivés	23
L'objet Map	23
L'objet Set	25
Les objets WeakMap et WeakSet	26
Les tableaux d'objets	27
Elles sont où les data ?!	31
Les boucles	32
Boucle While	33

Boucle for à deux vitesses	33
Boucle forEach, y'a du bon et du moins bon	34
Boucle for of, petite nouveauté ES6	37
Les fonctions map, filter et reduce	38
Les fonctions map et filter	38
La fonction reduce	40
Et les perfs dans tout ça ?	41
L'API Webworker	44
AlaSQL	45
Opérations sur grands nombres	47
Problèmes de précision	47
Le problème des faux dépassements	50
Bibliographie	51

Notes de l'auteur :

Je m'appelle Grégory Jarrige.

Je suis développeur professionnel depuis 1991. Après avoir longtemps travaillé sur le développement d'applications de gestion, sur gros systèmes, avec des langages et technos propriétaires, j'ai fait le pari de me former aux technos et langage open source vers 2006. J'ai commencé à développer des applications webs professionnelles à partir de 2007, avant d'en faire mon activité principale à partir de 2010. L'arrivée du HTML5 dans la même période a été pour moi une véritable bénédiction, et surtout un formidable terrain d'expérimentation (avec des API comme Canvas, WebAudio, etc...).

Aujourd'hui, je développe des interfaces utilisateurs en Javascript sur des applications de type intranet, ainsi que des tableaux de bord basés sur des solutions de dataviz comme D3.js. De plus en plus amené à travailler sur des volumes de données importants, aussi bien côté navigateur que côté serveur (sous NodeJS), j'ai éprouvé le besoin de faire le point sur les possibilités du langage Javascript pour la manipulation, de données. Car le langage Javascript évolue vite, et si le web regorge d'articles traitant régulièrement des nouveautés du langage, il n'est pas toujours facile de s'y retrouver.

Ce document n'est pas exhaustif, il est forcément partiel et certainement aussi un peu partial. C'est un "work in progress", réalisé sur mon temps libre, pour mes propres besoins, et j'espère qu'il pourra être utile à d'autres développeurs, qu'ils soient amateurs ou professionnels.

Le présent document est publié sous Licence Creative Commons n° 6. Il est présenté dans le cadre du meetup [CreativeCodeParis](#), le 20 décembre 2018.

Il est disponible en téléchargement libre sur mon compte Github, dans le dépôt suivant :

<https://github.com/gregja/JSCorner>

Préambule

Pourquoi parler de manipulation de données dans un meetup dédié au « creative coding », comme [CreativeCodeParis](#) ?

Pour comprendre, il faut revenir un an en arrière. Nous sommes en novembre 2017, je participe à un hackaton organisé par l'Adami. Mon équipe a décidé de proposer une expérience sensorielle, dans laquelle des images un tantinet psychédéliques bougent en interaction avec des données OSC (une norme concurrente de la norme MIDI, pour l'interfaçage d'instruments électroniques). Je m'occupe de la programmation des visuels avec P5.js, les données OSC sont produites sur une autre machine et arrivent sur la mienne via une connexion réseau. Tout semble bien fonctionner mais de temps en temps j'ai l'impression que je perds des données. Les données arrivent par salves à un rythme soutenu, trop rapide pour que je puisse analyser finement ce qui se passe. J'ai la tête dans le guidon, très peu de temps pour comprendre d'où vient le problème (que j'ai découvert à peine une heure avant la présentation au public). Finalement, je comprends in extremis où se situe le problème, glisse un patch de fortune pendant que le public s'installe... ouf !

Ce problème, vous pourriez le rencontrer aussi... peut être... tout dépend de votre degré de maîtrise du langage Javascript.

Si je vous dis dès maintenant où le problème se situait, vous risquez de vous arrêter à ça, et de passer à côté du reste. Aussi je préfère glisser l'explication du problème, et sa solution, quelque part à l'intérieur de ce support, j'espère que vous ne m'en voudrez pas trop.

Introduction

Dans tout projet impliquant de la manipulation de données - les fameuses « Data » - on doit très souvent faire appel à des variables de type « tableau » (en anglais « Array ») pour le stockage des données. Or les tableaux en Javascript ont des caractéristiques spécifiques, qui peuvent être déroutantes quand on est habitué à manipuler des tableaux dans d'autres langages, tels que PHP ou Java par exemple.

Je vais m'efforcer de passer en revue ces spécificités, et par la même occasion de vous montrer comment vous pouvez en tirer parti. Je mettrai aussi l'accent sur certaines difficultés, et vous montrerai les solutions de contournement que j'ai identifiées. Je vous montrerai aussi quelques techniques que j'ai expérimentées avec succès, pour travailler avec des tableaux de grande taille.

Avant de rentrer dans le vif du sujet, il me semble intéressant de souligner quelques aspects du langage Javascript. Le langage Javascript n'existe pas en tant que tel, c'est un nom d'usage, adopté par le plus grand nombre, pour désigner l'implémentation dans chaque navigateur de la norme ECMAScript 262 (il est vrai qu'avec un nom pareil...). Cette norme est régulièrement mise à jour et est publiée sur ce site :

<http://www.ecma-international.org/publications/standards/Ecma-262.htm>

Chaque navigateur est donc équipé d'un interpréteur de code Javascript, et il faut souligner que les éditeurs de navigateurs font aujourd'hui de gros efforts pour coller au plus près de la norme, et pour implémenter ses évolutions rapidement.

Si la norme EMCA-262 en est aujourd'hui à la version 9, publiée en juin 2018, il est intéressant de s'arrêter sur deux périodes clés de son évolution :

- L'édition 5 publiée en 2009, généralement appelée ECMAScript 5, constituait une avancée majeure. Elle apportait beaucoup de nouveautés qui auparavant étaient dispersées dans divers frameworks Javascript. Elle est arrivée en même temps que la norme HTML5 qui de son côté apportait un large panel d'API, toutes programmables en Javascript. Bref, l'année 2009 est véritablement une année charnière dans le monde du web. J'aime à penser qu'une petite révolution pacifique et bienveillante s'est produite, en 2009, dans nos navigateurs.
- L'édition 6 publiée en 2015, généralement appelée ECMAScript 6, ou quelquefois ECMAScript 2015 (ce qui a parfois entraîné des confusions dans les esprits). Plus discrète, mais tout aussi importante, cette édition a consolidé les bases posées par l'édition précédente, en apportant tout un tas de nouveautés très pratiques, notamment pour la manipulation de données. Nous en découvrirons certaines au fil de l'eau.

Ce qui est intéressant à souligner ici, c'est que nous sommes à la fin de l'année 2018 (au moment où j'écris ces lignes), que la norme ECMAScript 6 est là depuis 3 ans, et qu'elle est pleinement supportée par les navigateurs actuels. Si vous êtes un ou une « creative coder », vous pouvez donc vous lâcher. Si vous êtes un développeur professionnel, obligé de surveiller la compatibilité avec des navigateurs plus anciens ne supportant pas ES6, vous pouvez utiliser des solutions de type « transpilage » comme le projet Babel (qui permet une conversion du code ES6 vers ES5), donc vous pouvez vous lâcher aussi sur ES6. Et puis si vous développez sous NodeJS, vous savez sans doute que l'interpréteur Javascript de NodeJS est à la pointe en ce qui concerne le support de la norme ECMAScript 262, donc

tout ce que nous allons évoquer ici devrait vous intéresser (même si nous allons plus particulièrement nous intéresser à l'utilisateur de Javascript côté navigateur).

A qui est destiné ce support ?

A un large public, à commencer par les membres du meetup [CreativeCodeParis](#), qui sont de plus en plus nombreux à se lancer dans des projets Javascript impliquant la manipulation de données, que ce soit pour de la visualisation de données (dataviz), du machine learning, ou du creative coding en général. D'autres lecteurs qui pourraient être intéressés, ce sont les anciens élèves avec lesquels j'ai travaillé en entreprise et dans certains centres de formation, car j'aborde ici un certain nombre de sujets que l'on a rarement le temps d'aborder dans une formation classique. Et puis tout lecteur autodidacte, développeur professionnel ou amateur, pourra piocher dans les idées et techniques que je présente ici, en fonction de ses centres d'intérêt et de son niveau d'expertise.

Ce n'est pas un support de cours à proprement parler, mais plutôt une discussion que j'entame avec tout lecteur intéressé par la manipulation de données en Javascript, ou qui voudrait s'y frotter et qui recherche un document synthétique lui permettant de rentrer rapidement dans le vif du sujet. Une bibliographie, à la fin du support, vous fournira une liste d'excellentes références pour approfondir certains des sujets abordés ici. J'espère que ce document pourra être utile aussi aux développeurs venant au Javascript après être passés par d'autres langages, et qui pourraient être déçus par certaines caractéristiques des données en Javascript.

Attention, ce document n'est pas destiné à de grands débutants en programmation, je recommanderais plutôt à ces derniers de se tourner vers un autre document se trouvant dans le même dépôt Github, et qui s'intitule "Cours Javascript Premiers Pas".

Un dernier point, les techniques de manipulation de données, et notamment de tableaux, que nous allons étudier ici sont bien adaptées pour la manipulation de jeux de données allant jusqu'à quelques dizaines - voire centaines - de milliers de lignes. Pour la manipulation de volumes supérieurs, le livre "Data wrangling with Javascript", de Ashley Davis, me semble incontournable (cf. chapitre "bibliographie").

Premiers éléments sur les tableaux

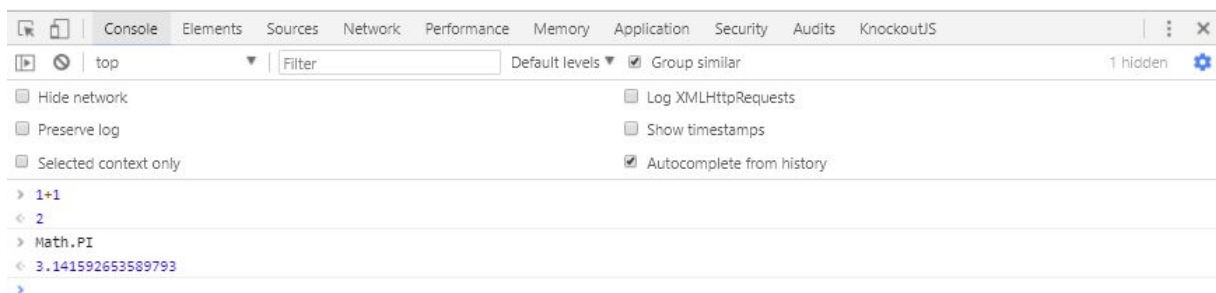
La console Javascript du navigateur

Tout navigateur internet digne de ce nom est équipé d'un environnement de développement, qui lui-même fournit un grand nombre de services, dont une console.

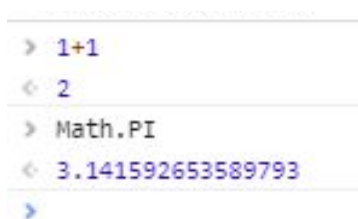
Cette console nous permet d'interagir directement avec l'interpréteur Javascript du navigateur.

Nous allons nous servir de la console, pour tester et vérifier rapidement un certain nombre d'idées liées au Javascript. Sur la plupart des navigateurs, l'accès à la console se fait au moyen de la touche F12. Si vous êtes sur Safari (dont sur Mac), allez dans la barre de menu, choisissez l'option Safari > Preferences, cliquez sur « advanced » et sélectionnez « show develop menu in menu bar ».

L'image suivante présente la fenêtre contenant les outils de développement de Google Chrome. Vous noterez que nous sommes ici dans l'onglet « console », vérifiez que c'est bien votre cas.



Dans l'image ci-dessous, j'ai zoomé sur les lignes de code que j'ai saisies ci-dessus, et je vous invite à les tester aussi :



Vous voyez que l'interpréteur Javascript est immédiatement accessible. Dans un premier temps, je lui ai demandé de faire une banale addition, dans un second temps, je lui ai demandé de me fournir la valeur de la propriété PI associée à l'objet Javascript Math. Cet objet Math est un objet standard du Javascript, il en existe beaucoup d'autres en Javascript, mais je ne m'attarde pas sur le sujet pour l'instant.

Près de la barre de menus, en haut à gauche, vous noterez la présence d'une icône présentant un sens interdit. Elle est très utile pour vider la console quand cette dernière devient trop chargée et devient peu lisible :



Pour les grands débutants, je signale l'utilisation de la touche « flèche haut », qui permet de rappeler les dernières commandes saisies.

Après cette rapide introduction à la console Javascript, nous pouvons entrer dans le vif du sujet.

Un tableau en Javascript, c'est quoi ?

On peut créer les tableaux de deux manières :

```
var test1 = []
var test2 = new Array()
```

Entrez les deux lignes ci-dessus dans votre console de navigateur :

```
> var test1 = []
< undefined
> var test2 = new Array()
< undefined
```

Vous noterez que le terme « undefined » qui apparaît sous chaque ligne indique simplement que tout va bien et que l'instruction que vous venez d'exécuter ne renvoie aucune information.

Les deux variables que nous venons de créer sont strictement équivalentes du point de vue de leur structure. Le test ci-dessous permet de s'assurer qu'elles ont bien le même type :

```
typeof(test1) == typeof(test2)    // => true
```

Mais au fait, le type de ces deux variables, c'est quoi ? La fonction Javascript « typeof » va nous le dire :

```
typeof(test1)    // => object
```

Ah bon ? Je croyais avoir créé un tableau... un tableau serait donc un objet ? Eh bien oui ! On peut le vérifier en demandant à l'interpréteur Javascript de nous dire si la variable « test1 » est une instance du type Object :

```
test1 instanceof Object    //=> true
```

Mais la variable « test1 » est aussi un tableau, comme le test suivant le démontre :

```
test1 instanceof Array    //=> true
```

Donc le type « Array » serait une instance du type « Object » ? Vérifions-le :

```
Array instanceof Object    //=> true
```

Bingo !!

Donc si je comprends bien, la variable « test1 » est un tableau, mais c'est aussi et surtout un objet enfant de la classe « Array », qui elle-même est une classe « fille » de la classe « Object » ???

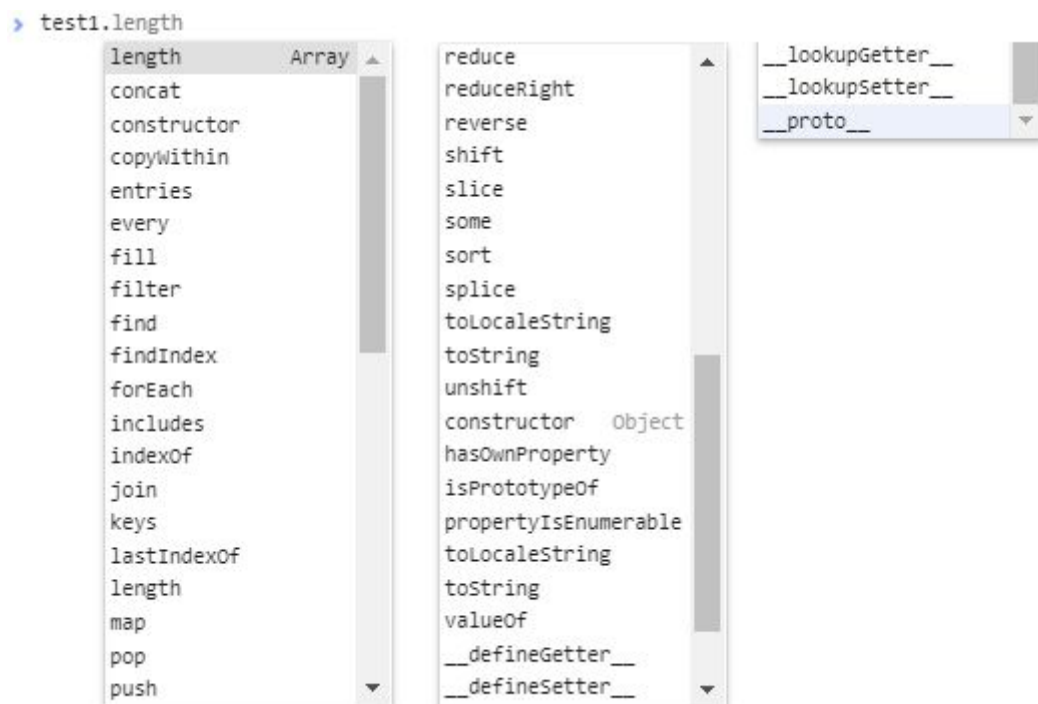
C'est à peu près ça oui... sauf que la notion de classe n'a pas le même sens en Javascript, que dans des langages comme PHP ou Java. En Javascript, les classes sont créées à partir de fonctions, et le modèle objet de Javascript est fondé sur la notion de prototype. C'est un concept passionnant et

surtout très puissant, mais si nous allons trop loin dans cette direction, nous allons perdre de vue notre sujet qui est la manipulation de données.

Les méthodes et propriétés d'un tableau

Dans les exemples précédents, nous avons créé la variable « test1 ».

Puisque la variable « test1 » est un tableau mais aussi un objet, cela explique pourquoi, quand on utilise l'auto-complétion dans la console du navigateur. Grâce à l'auto-complétion, on voit apparaître une longue liste de méthodes et propriétés associées à cet objet :



Pour information, on déclenche l'auto-complétion en saisissant « test1 » suivi immédiatement d'un point. Si l'auto-complétion tarde à se déclencher, vous pouvez accélérer le mouvement, après la saisie du point, en pressant simultanément les touches « Ctrl » et « Espace ».

Vous avez donc constaté que nous disposons de nombreuses propriétés (valeurs) et méthodes (fonctions) associées à la variable « test1 », nous en étudierons quelques-unes au fil de l'eau.

Une première propriété sympathique, et surtout très utile, c'est « length », qui renvoie le nombre d'éléments du tableau :

```
> test1.length
< 0
```

Attention, « length » est une propriété, pas une méthode, c'est pourquoi vous ne pouvez pas exécuter l'instruction suivante :

```
> test1.length()
```

```
✖ ▶ Uncaught TypeError: test1.length is not a function
    at <anonymous>:1:7
```

Contrairement à d'autres langages comme Java, les tableaux en Javascript ne sont pas typés. Le stockage de données hétérogènes est donc très facile en Javascript, il s'apparente beaucoup au langage PHP sur ce point précis.

Exemple ci-dessous avec un tableau contenant... un joyeux bazar :

```
> test1 = [0, 'hello', true, 'we need you', (5/2), false]
< ▶ (6) [0, "hello", true, "we need you", 2.5, false]
```

Nous aurions pu créer ce même tableau, étape par étape, en utilisant la méthode « push », comme ceci :

```
> test1 = [];
< ▶ []
> test1.push(0);
< 1
> test1.push('hello');
< 2
> test1.push(true);
< 3
> test1.push('we need you');
< 4
> test1.push(5/2);
< 5
> test1.push(false);
< 6
```

Vous voyez ci-dessus une copie d'écran de ma console Javascript. Vous voyez qu'entre chaque instruction, l'interpréteur nous indique le numéro de poste qu'il vient de créer dans le tableau « test1 ».

Quand les développeurs Javascript s'arrachent les cheveux sur un bug, ils recourent souvent à l'objet « console » et à sa célèbre méthode « log », pour scruter le contenu d'une variable. Ils glissent cette instruction à des endroits qu'ils jugent stratégique, dans leur code source, pour visualiser le contenu de certaines variables, en vue d'identifier l'anomalie rencontrée. Avec notre tableau, cela donne ceci :

```
> console.log(test1)
▶ (6) [0, "hello", true, "we need you", 2.5, false]
```

Oui je sais, ça donne le même résultat que si vous aviez saisi « test1 » sur la ligne de commande, mais ça c'est parce que vous êtes dans la console. Quand votre code s'exécute à l'intérieur d'un fichier source, vous êtes obligé de passer par cette commande « console.log » pour pouvoir « envoyer » des données à la console.

Vous noterez que l'on peut « déplier » le tableau en cliquant sur la petite flèche, comme ceci :

```
> console.log(test1)
▼ (6) [0, "hello", true, "we need you", 2.5, false] ⓘ
  0: 0
  1: "hello"
  2: true
  3: "we need you"
  4: 2.5
  5: false
  length: 6
  ▶ __proto__: Array(0)
```

Beaucoup moins connue, et pourtant très pratique, la méthode « table » de l'objet « console », offre un affichage différent, que vous apprécierez sûrement :

```
> console.table(test1)
```

VM532:1

(index)	Value
0	0
1	"hello"
2	true
3	"we need you"
4	2.5
5	false

▶ Array(6)

Vous admettez que c'est moins spartiate que l'affichage proposé par « console.log ».

Des tableaux de tableaux

Un tableau peut lui-même contenir d'autres tableaux, comme dans l'exemple suivant, où la variable « test2 » contient deux 2 sous-tableaux contenant chacun 3 éléments :

```
> var test2 = [['a', 'b', 'c'], [1, 2, 3]]
< undefined
```

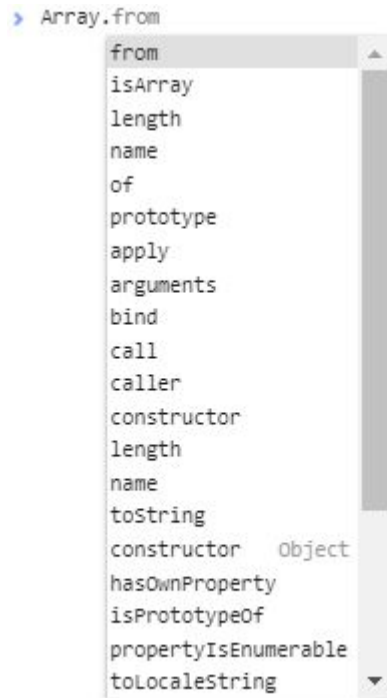
Vous avez peut être remarqué que l'interpréteur Javascript vous indique « undefined » à chaque fois que vous créez une nouvelle donnée. C'est normal, cela signifie qu'il a bien fait son travail et que l'instruction que vous lui avez demandé d'exécuter ne renvoie aucune information. Ce n'est que cela, aucune raison de vous inquiéter.

Si nous affichons la variable « test2 » dans la console, nous pouvons « déplier » son contenu pour en analyser la structure :

```
> test2
< ▼ (2) [Array(3), Array(3)] ⓘ
  ▼ 0: Array(3)
    0: "a"
    1: "b"
    2: "c"
    length: 3
    ► __proto__: Array(0)
  ▼ 1: Array(3)
    0: 1
    1: 2
    2: 3
    length: 3
    ► __proto__: Array(0)
  ► __proto__: Array(0)
```

Spécificités de l'objet Array

L'objet Array met à disposition quelques méthodes spécifiques, que l'on peut observer toujours via le mécanisme d'auto-complétion :



Vous remarquerez que Array est un objet multi-usage, qui permet d'instancier des tableaux, mais qui peut aussi être utilisé indépendamment, par exemple pour analyser le contenu de variables que l'on passe en paramètre à certaines de ses fonctions.

Parmi les méthodes de l'objet Array, il y en a une qui est très pratique, c'est `Array.isArray()`. Exemple ci-dessous :

```
> var test1 = []
< undefined
> Array.isArray(test1)
< true
> var test2 = "";
< undefined
> Array.isArray(test2)
< false
```

Si vous recevez une variable en entrée d'une fonction, vous pouvez à tout instant vérifier son type, et éventuellement adapter le comportement de votre fonction au type de donnée reçu, comme dans cet exemple :

```
function coucou(input) {
  if (Array.isArray(input)) {
    return "youpi!";
  } else {
    return "game over, play again!";
  }
}
coucou(test1);    // => "youpi!"
coucou(test2)    // => "game over, play again!"
```

Je vous encourage à tester ce code dans la console de votre navigateur.

Sachant que le Javascript n'est pas un langage fortement typé, contrairement à Java par exemple, il est important de connaître cette méthode qui vous permet de vérifier à tout moment le type de la donnée que vous manipulez. Cela vous permet de blinder votre code en rejetant des types de données incorrects, comme dans notre fonction « coucou ».

A noter : la méthode « Array.isArray » fait partie des apports d'ES5 (version 2009).

Ton passage de paramètre, tu le veux par adresse ou par valeur ?

Posons le problème

Vous vous demandez peut-être pourquoi j'ai tant insisté sur le fait qu'un tableau Javascript était un objet ?

Eh bien, il faut savoir qu'en Javascript, tout objet transmis à une fonction est transmis « par référence », on dit aussi « par adresse ». Un objet n'est donc pas transmis « par valeur ». Ce que la fonction reçoit en paramètre, ce n'est pas le contenu du tableau, mais un pointeur vers un objet extérieur.

Prenez le temps de relire le paragraphe qui précède. Aviez-vous conscience de ce facteur auparavant ? En mesurez-vous bien les conséquences ? Cela signifie en effet que, si vous modifiez le contenu d'un tableau reçu en paramètre, à l'intérieur d'une fonction, vous êtes en train de modifier un objet extérieur à la fonction. Cela a de nombreuses implications.

L'erreur courante est en effet de procéder ainsi :


```
function ouille(input) {
  if (input.length > 0) {
    input[0] = 'gros minet';
  }
  return input;
}
var test = ['titi', 'winnie'];
console.log(test); //=> ["titi", "winnie"]
ouille(test);
console.log(test); //=> ["gros minet", "winnie"]
```

Dans l'exemple ci-dessus, la fonction « ouille » reçoit un tableau, contrôle s'il contient des données, et si c'est le cas, elle modifie le contenu du premier poste du tableau. Ensuite, elle renvoie le tableau en sortie. En première lecture, cela semble parfait, mais si vous repensez à l'avertissement que j'ai formulé dans le paragraphe précédent, vous devez pressentir qu'il y a un problème.

En règle générale, le rôle d'une fonction est de recevoir des paramètres en entrée, de les exploiter pour générer de nouvelles données, et de les renvoyer en sortie via le mot clé « return ». Or ce n'est pas ce que fait la fonction « ouille ». Donc si vous pensiez modifier le contenu du tableau « test » via le code suivant :

```
test = ouille(test)
```

... eh bien, vous êtes dans l'erreur. En effet, la fonction a modifié le contenu du premier poste du tableau, mais ce tableau est en réalité un pointeur vers le tableau d'origine qui se trouve à l'extérieur de la fonction. De plus le « return » en fin de fonction a pour effet de renvoyer en sortie... le pointeur et non le contenu du tableau. Donc écrire ceci :

```
test = ouille(test)
```

... revient à modifier le contenu du tableau d'origine via son pointeur, et à renvoyer ce pointeur sur son objet d'origine !

Comment se prémunir contre ce phénomène ?

Vous pourriez être tenté de copier la variable « input » dans une variable de travail, avant de modifier le contenu de cette variable de travail. Pas bête, mais attention, si à l'intérieur de votre fonction « ouille » vous écrivez ceci :

```
var temp = input;
```

On se fait un petit slice ?

... vous n'avez pas réglé votre problème, car vous venez de créer une variable « temp » qui est un pointeur sur la variable « input » (qui je le rappelle est elle-même un pointeur). Pour éviter cela, vous pouvez effectuer une copie des données de la variable « input » via la méthode « slice » qui est une méthode associée à tout objet de type tableau.

```
var temp = input.slice();
```

... la méthode « slice » permet de copier le contenu d'un tableau, d'un poste à un autre, mais si on ne lui passe pas de paramètre, elle effectue par défaut une copie de l'ensemble du tableau. C'est propre, net, et sans bavures.

ATTENTION !!! La technique ci-dessus fonctionne bien sur les tableaux simples, ainsi que sur les tableaux de tableaux. En revanche, elle ne fonctionne pas sur les tableaux d'objets, que nous étudierons dans un chapitre ultérieur.

Hormis ce bémol relatif aux tableaux d'objets, la méthode « slice » est vraiment efficace dans la plupart des cas. Fort de cette connaissance, vous pouvez reprendre la fonction « ouille » et la réviser de manière à travailler en interne sur la variable « temp », et ainsi vous assurer que vous ne modifiez plus accidentellement le tableau d'origine.

A noter : vous pouvez aussi placer votre slice directement dans l'appel de la fonction, et ainsi décharger cette dernière de la problématique de la copie du jeu de données, comme dans l'exemple ci-dessous. Le risque ici, c'est que vous oubliiez le « slice » de temps en temps, ce qui pourrait engendrer des erreurs difficiles à repérer :

```
function moins_ouille(input) {
  console.log(input);
}
var big_data = [1, 2, 3];
moins_ouille(big_data.slice());
► (3) [1, 2, 3]
```

Vous préférez peut être un petit “spread” ?

Il existe une alternative à la méthode « slice » qui est apparue avec ES6 (norme ECMAScript 262 édition 2015), c'est l'opérateur « **spread** », aussi appelé « **opérateur de décomposition** ». Il se matérialise par trois petits points ...

Très pratique, cette technique offre plein de possibilités, comme le démontre cet exemple emprunté au site « putaindecode » :

```
const myArray = [1991, 8, 1]
new Date(...myArray) // object Date - équivaut à: new Date(1991, 8, 1)

const myString = "foo bar"
// les objets String étant itérables
[...myString] // ["f", "o", "o", " ", "b", "a", "r"]
```

Source : <https://putaindecode.io/fr/articles/js/es2015/rest-spread/>

Dans le cas qui nous préoccupe, nous pouvons donc remplacer la méthode « slice » par un « spread », comme dans cet exemple :

```
> function xvalue(arr) {
    var newArr = [ ...arr ];
    newArr[0] = 'XXX';
    return newArr;
}
test = [1, 2, 3];
xvalue(test);
< ▶ (3) ["XXX", 2, 3]
> test
< ▶ (3) [1, 2, 3]
```

Vous voyez dans cet exemple que nos petites manipulations à l'intérieur de la fonction « xvalue » sont sans effet sur le tableau d'origine transmis à la fonction. Ce dernier contient toujours les données d'origine, soit 1, 2 et 3.

Et si vous preniez plutôt un “Array.from” ?

C'est encore une nouveauté d'ES6, la méthode `Array.from()` permet de créer une nouvelle instance de tableau. Comme le précise la documentation Mozilla, il s'agit d'une copie superficielle, c'est à dire qu'elle souffre des mêmes limitations pour la duplication de tableaux d'objets que les techniques abordées dans les paragraphes précédents de ce chapitre.

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/from

Néanmoins, cette fonction `Array.from()` est bien sympathique puisqu'elle est en mesure de transformer en tableaux des objets qui n'en étaient pas à l'origine, à condition toutefois qu'ils soient itérables. Ces objets itérables, ce sont par exemples :

- des chaînes de caractères (qui ne sont pas des tableaux, contrairement à ce que pensent beaucoup de développeurs Javascript).
- des “node list” ou des “HTML collections”, soit des noeuds du DOM qui d'un point de vue du langage Javascript ressemblent beaucoup à des tableaux mais n'en sont pas pour autant

Assez causé, voici quelques exemples d'utilisation :

```
console.log(Array.from([1, 2, 3], x => x * x)); //=> output: Array [1, 4, 9]
```

```
var test = Array.from("ABCDEFGH");
console.log(test); //=> ["A", "B", "C", "D", "E", "F", "G"]
```

```
// technique de scrapping
```

```
var scrapper = Array.from(document.querySelectorAll('.test > ul li a'))
    .map(a => a.textContent)
    .filter(grad => (grad.startsWith('ti') || grad.startsWith('to')))
```

```
;
console.log(scrapper);
```

Fausse conclusion... et vrais problèmes

La copie des données d'un tableau vers un autre peut être longue, si vous travaillez sur un gros tableau. Ce cas de figure peut justifier que vous préféreriez travailler directement sur la variable « input » plutôt que sur une copie. Mais dans ce cas, pensez à le documenter pour les collègues, si vous travaillez sur un projet impliquant plusieurs personnes.

Mais je n'ai pas fini de vous embêter avec cette histoire de paramètres, car il y a un gros risque avec le fait de travailler sur l'objet d'origine plutôt que sur une copie du contenu de cet objet. En effet, en Javascript, beaucoup de traitements se font de manière asynchrone. Si dans votre fonction, vous travaillez sur le pointeur d'un tableau extérieur à cette fonction, tableau dont le contenu est susceptible d'être modifié par la réponse renvoyée par une nouvelle requête AJAX, eh bien vous vous retrouvez brutalement avec un tableau dont le contenu n'a plus rien à voir avec ce qu'il était au préalable. Cela peut vous arriver par exemple si vous recevez des salves de données, de type MIDI ou OSC, transmises par un instrument de musique électronique. Eh oui, si vous avez lu le préambule du présent support, vous savez maintenant à quel problème je faisais référence. Nous verrons dans le chapitre « empilage de tableau » une technique permettant de contourner cette difficulté.

Les tableaux associatifs

Chacun sait que les tableaux simples, avec leurs clés numériques, ne sont pas bien adaptés à certains types de données. Il est parfois plus intéressant de disposer d'un système de stockage dans lequel on bénéficie d'une plus grande liberté dans la définition des clés. C'est ce que les développeurs PHP connaissent depuis longtemps sous la notion de « tableau associatif ».

Cette notion de tableau associatif n'existait pas en Javascript, jusqu'à l'arrivée de ES6 (en 2015). On s'en sortait malgré tout plutôt bien, en utilisant de simples objets Javascript comme support de stockage de nos paires de « clés-valeurs », comme ceci :

```
var tableau = {  
  foo: "abc",  
  bar: "123"  
}
```

Pour autant, la variable « tableau » ci-dessus n'est pas un tableau, elle ne bénéficie donc pas des mécanismes standards d'un tableau.

La norme ES6 (je rappelle qu'elle est arrivée en 2015), nous apporte l'objet Map, objet itérable qui couvre particulièrement ce besoin de disposer de véritables tableaux associatifs. Avant de présenter cette nouveauté, il me semble utile de rappeler comment on procédait, dans les versions de Javascript antérieures à ES6, pour stocker des « paires de clés-valeurs ».

Avant ES6

Avant l'arrivée d'ES6, pour définir un objet dédié au stockage de données, on procédait généralement comme ceci :

```
var tableau = {  
  "foo": "abc",  
  "bar": "123"  
}
```

... ou encore comme cela :

```
var tableau = {  
  foo: "abc",  
  bar: "123"  
}
```

Les différences entre ces 2 notations sont les suivantes :

- Dans la première notation, nous avons utilisé des guillemets pour déclarer les clés « foo » et « bar », clés qui sont de type chaînes de caractères, c'est un exemple typique d'objet conforme à la syntaxe JSON (cf. https://fr.wikipedia.org/wiki/JavaScript_Object_Notation)

- Dans la seconde notation, nous n'avons pas utilisé de guillemets, « foo » et « bar » sont donc les propriétés d'un objet, voilà tout.

Dans les exemples ci-dessus, nous sommes sur des structures de type « tableau plat » (en anglais « flat array »), donc des structures très simples, non arborescentes, telles que l'on pourrait en récupérer au travers d'une API (même si certaines API renvoient des structures nettement plus complexes). D'un point de vue fonctionnel, les deux objets ci-dessus sont équivalents et leur manipulation avec différents types d'itérateurs (que nous étudierons dans un prochain chapitre) ne présentera pas de difficulté particulière.

Apparue avec la ES5, la méthode « Object.keys », permet de récupérer la liste des propriétés d'un objet et de les renvoyer sous forme d'un tableau :

```
> Object.keys(tableau)
< ▶ (2) ["foo", "bar"]
```

ES6 nous a apporté en complément la méthode « Object.values » qui elle renvoie un tableaux contenant les valeurs de l'objet :

```
> Object.values(tableau)
< ▶ (2) ["abc", "123"]
```

D'autres méthodes intéressantes sont apparues avec ES6, comme « getOwnPropertyDescriptors », ou « getOwnPropertyNames », mais les étudier maintenant nous éloignerait de notre sujet, alors je laisse cela de côté. Vous pourrez les étudier au travers de la documentation Mozilla, notamment :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Object/getOwnPropertyNames

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Object/getOwnPropertyDescriptor

A partir du moment où l'on sait récupérer les clés d'un objet sous forme d'un tableau, il est facile de les trier, les compter, etc... avec tout le jeu de fonctions disponibles sur un objet de type tableau, et de les manipuler avec toutes les fonctions itératives que nous listerons dans un prochain chapitre.

Les objets Map et Set, et leurs dérivés

L'objet Map

Avec ES6, sont arrivés de nouveaux objets : Map, Set, WeakMap et WeakSet.

Commençons par l'objet Map avec un premier exemple, que j'ai testé dans Google Chrome et que je vous livre tel quel. Vous allez voir qu'il y a une surprise à la fin :

```
var maMap = new Map(); //=> création d'un objet Map

// définir les clés et valeurs
maMap.set("cle1", "valeur associée à clé1");
maMap.set("cle2", "valeur associée à clé2");
maMap.set("cle3", "valeur associée à clé3");

// nombre d'éléments ?
console.log(maMap.size); //=> 3 (attention, "size" et non pas "length" !!)

// récupération de quelques valeurs
console.log(maMap.get("cle1")); //=> renvoie la valeur associée à clé1
console.log(maMap.get("xxx")); //=> undefined, ce qui est normal

console.log(maMap.keys()); //=> renvoie un objet MapIterator contenant
//      {"cle1", "cle2", "cle3"}

console.log(maMap.has("cle1")) //=> true (c'est normal)

console.log(maMap.delete("cle1")) //=> true (jusque là, tout va bien)

console.log(maMap.keys()); //=> MapIterator {"cle3"}
//      !!! Mais où est passé "cle2" ???

console.log(maMap.size); //=> 1 !!! un seul poste !?!? WHAT THE FU.. !!!

// petite boucle "for of" (nouveau d'ES6) pour parcourir le contenu de la Map
for (var clé of maMap.keys()) {
  console.log(clé);
} //=> "cle3"
```

Bon, ce test, je l'ai effectué sur une machine équipée d'un Google Chrome en version 64.x.

Vous admettez comme moi que la Map devrait contenir en fin de test deux éléments ayant pour clé « cle2 » et « cle3 ».

J'ai refait le même test sur un Firefox fraîchement installé (version 64) et je n'ai pas constaté le même problème. J'ai refait les tests ensuite sur un Google Chrome à jour (version 65) et j'ai constaté que le bug avait disparu. Cela signifie que la fiabilité de la méthode « delete » de l'objet Map laissait à désirer, pendant un temps sur Google Chrome, et que ce bug a pu impacter aussi certaines versions de NodeJS (qui embarque l'interpréteur Javascript de Google Chrome).

Donc si vous développez pour votre propre usage, et que votre navigateur est à jour, vous pouvez sans doute utiliser l'objet Map les yeux fermés, dans le cas contraire, je recommanderais la prudence, ou peut être de bannir la méthode « delete » de votre boîte à outils.

En tout cas, cela nous a permis d'introduire la boucle « for of », autre petite nouveauté d'ES6, qui permet de « parcourir » tout objet itérable, qu'il s'agisse d'un tableau ou d'une Map.

Pour de plus amples précisions sur ce sujet :

<https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/for...of>

Pour conclure sur l'objet Map, j'apprécie particulièrement la facilité avec laquelle on peut fusionner deux jeux de données :

```
> var premier = new Map([
    [1, 'un'],
    [2, 'deux'],
    [3, 'trois'],
]);

var second = new Map([
    [1, 'uno'],
    [2, 'dos']
]);

// On peut fusionner des Maps avec un tableau
// Là encore c'est le dernier exemplaire de la clé qui l'emporte
var fusion = new Map([...premier, ...second, [1, 'eins']]);

for (var [clé, valeur] of fusion) {
    console.log(clé + " = " + valeur);
}
```

```
1 = eins
```

```
2 = dos
```

```
3 = trois
```

Pour approfondir l'utilisation de l'objet Map :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Map

L'objet Set

Je vous avoue que j'ai peu utilisé l'objet Set, mais sa capacité à éliminer des doublons dans des jeux de données m'a séduit, et je pense que je vais l'utiliser plus fréquemment à l'avenir.

Voici quelques exemples d'utilisation empruntés à la documentation Mozilla :

```
var monTableau = ["valeur1", "valeur2", "valeur2", "valeur3"];

// On peut utiliser le constructeur Set pour transformer un Array en Set
var monSet = new Set(monTableau);

console.log(monSet.has("valeur1")); // renvoie true

console.log(monSet.size); // renvoie 3 (car "valeur2" a été "dédoublonné")

// le spread permet de transformer un Set en Array.
console.log([...monSet]); //=> ["valeur1", "valeur2", "valeur3"]

// dédoublonnage de valeurs
const nombres = [2,3,4,4,2,2,2,4,4,5,5,6,6,7,5,32,3,4,5];
console.log([...new Set(nombres)]); //=> affichera [2, 3, 4, 5, 6, 7, 32]
```

Pour approfondir le sujet :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Set

Les objets WeakMap et WeakSet

En fait, ce sont les pendants des objets Map et Set, mais ces variantes permettent d'utiliser des objets comme clés.

Très sincèrement, c'est l'objet WeakSet qui me semble être le plus intéressant des deux, du fait qu'il pourrait permettre d'éliminer des objets identiques au sein d'un jeu de données. Mais je manque de recul, faute de les avoir pratiqués, aussi je vous laisse le soin de les approfondir via la documentation Mozilla :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/WeakMap

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/WeakSet

Les tableaux d'objets

Les tableaux d'objets sont beaucoup utilisés en Javascript, car ils sont très pratiques pour stocker des volumes plus ou moins importants de données structurées. On peut aussi y stocker des données non structurées, mais pour l'heure ce sont les données structurées qui m'intéressent.

Exemple avec un jeu de données renvoyées par une API REST :

```
[{"id": 1, "ip_address": "69.59.45.22", "Date_start_connexion": "2018-04-20T22:08:51Z", "Date_end_connexion": "2018-07-05T21:50:27Z"}, {"id": 2, "ip_address": "238.169.1.43", "Date_start_connexion": "2018-07-04T00:59:19Z", "Date_end_connexion": "2018-09-05T15:21:26Z"}, {"id": 3, "ip_address": "95.151.135.180", "Date_start_connexion": "2018-04-29T14:42:59Z", "Date_end_connexion": "2018-02-27T02:51:50Z"}, {"id": 4, "ip_address": "37.42.8.166", "Date_start_connexion": "2018-05-11T22:20:50Z", "Date_end_connexion": "2018-05-19T05:31:36Z"}, {"id": 5, "ip_address": "68.23.244.202", "Date_start_connexion": "2018-05-21T16:58:10Z", "Date_end_connexion": "2018-12-07T11:00:23Z"}, {"id": 6, "ip_address": "13.75.26.202", "Date_start_connexion": "2018-12-06T13:08:21Z", "Date_end_connexion": "2018-02-01T17:49:18Z"}, {"id": 7, "ip_address": "100.43.43.234", "Date_start_connexion": "2018-01-23T15:51:54Z", "Date_end_connexion": "2018-03-17T10:02:49Z"}, {"id": 8, "ip_address": "168.130.98.86", "Date_start_connexion": "2018-09-30T11:36:32Z", "Date_end_connexion": "2018-07-30T14:31:59Z"}]
```

```

    "id": 9,
    "ip_address": "51.49.126.60",
    "Date_start_connexion": "2018-03-13T05:40:05Z",
    "Date_end_connexion": "2018-06-21T17:54:29Z"
  },
  {
    "id": 10,
    "ip_address": "44.253.76.95",
    "Date_start_connexion": "2018-02-22T05:55:52Z",
    "Date_end_connexion": "2018-12-03T12:53:45Z"
  }
]

```

Pour le cas où vous ne seriez pas habitué à cette notation, nous avons ici un tableau composé de 10 postes, chaque poste contenant un objet. J'ai « stabilisé » les éléments pairs du tableau en gris, en espérant que cela vous aide à bien comprendre la structure du jeu de données.

Si on prend le premier élément (ou poste) du tableau, on voit qu'il s'agit d'un objet du fait de la présence des crochets :

```

{
  "id": 1,
  "ip_address": "69.59.45.22",
  "Date_start_connexion": "2018-04-20T22:08:51Z",
  "Date_end_connexion": "2018-07-05T21:50:27Z"
}

```

En PHP, on aurait écrit un tableau de tableaux associatifs, en Javascript on obtient donc l'équivalent avec un tableau d'objets.

Je reviens à la charge sur le problème de la copie d'objet. J'avais écrit dans un précédent chapitre, que la méthode « slice » ne fonctionne pas avec les tableaux d'objets. Vous pourriez avoir des doutes sur cette affirmation, alors je vous propose de tester l'exemple ci-dessous dans la console de votre navigateur. Commencez par créer un tableau xtests contenant 2 objets :

```

> var xtests = [];
< undefined
> xtests.push({a:5, b:10});
< 1
> xtests.push({a:3, b:12});
< 2

```

Puis dupliquez le tableau « xtests » avec la méthode « slice » :

```

> var ytests = xtests.slice()
< undefined

```

Vérifiez dans la console le contenu de ce nouveau tableau en le « dépliant » avec la petite flèche :

```
> ytests
< ▼ (2) [{...}, {...}] ⓘ
  ▶ 0: {a: 5, b: 10}
  ▶ 1: {a: 3, b: 12}
    length: 2
  ▶ __proto__: Array(0)
```

Maintenant modifiez le contenu d'un poste du tableau ytests, par exemple la propriété « a » du poste zéro, fixée ici arbitrairement à 25 :

```
> ytests[0].a = 25;
< 25
```

Observez maintenant le contenu du tableau « xtests », qui est le tableau d'origine :

```
> xtests
< ▼ (2) [{...}, {...}] ⓘ
  ► 0: {a: 25, b: 10}
  ► 1: {a: 3, b: 12}
  length: 2
  ► __proto__: Array(0)
```

Aïlle, ça fait mal, pas vrai ? J'ai testé pour vous avec l'opérateur « spread » et je peux vous dire que le problème est strictement le même.

Si « xtests » et « ytests » sont bien deux tableaux distincts, leurs postes respectifs en revanche, sont liés par le fait qu'ils pointent sur les mêmes objets.

Pour être plus clair, disons que le poste zéro du tableau « xtests » pointe sur le même objet que le poste zéro du tableau « ytests », et ainsi de suite pour tous les autres postes.

Comment se prémunir contre ce problème.

On peut bien évidemment écrire une boucle qui parcourt l'ensemble des postes du tableau d'origine, et recrée un nouveau tableau, brique par brique, comme dans cet exemple :

```
var ytests = [];
for (let i=0, imax=xtests.length; i<imax; i++) {
  let item = xtests[i];
  ytests.push({a:item.a, b:item.b});
}
```

Mais il existe une technique plus rapide et de surcroît plus simple, elle consiste à combiner les fonctions JSON.parse et JSON.stringify, comme dans l'exemple suivant :

```
var xtests = [{a:5, b:10}, {a:3, b:12}]

var ytests = JSON.parse(JSON.stringify(xtests));
```

La fonction « JSON.stringify » a pour effet de prendre une photographie du tableau « xtests », comme si on figeait le tableau dans un bloc de glace. On dit qu'elle « sérialise » le contenu de l'objet « xtests ». La fonction « JSON.parse » produit l'effet inverse, elle « désérialise ». Mais entre ces deux opérations, toute référence aux objets d'origine est perdue, le tableau « ytests » n'a donc aucune

dépendance avec le tableau d'origine « ytests ». C'est une copie intégrale et autonome, un vrai jumeau en somme 😊. Attention, cette technique utilise des fonctions JS natives, donc potentiellement très performances, mais je ne recommanderais pas cette solution sur de gros tableaux.

Elles sont où les data ?!

En introduction, j'ai parlé de data, mais on n'en a pas trop vu la couleur jusqu'ici.

Pour avoir de la matière, je vous propose d'aller sur le site Mockaroo :

<https://www.mockaroo.com/>

Mockaroo est un site internet qui propose un service de génération de données bidon (on les appelle généralement des « mock data », ou encore des « fake data »).

Il est possible de personnaliser complètement le jeu de données généré par Mockaroo. De nombreux types de données sont proposés, et de nombreux formats de sortie également (JSON, CSV, etc.).

Cette solution est très pratique si vous avez besoin de développer du code sur la base d'une structure de données pour laquelle vous ne disposez pas encore de données réelles. Cela arrive souvent dans les projets d'entreprise.

Par défaut, Mockaroo propose de générer 1000 lignes par défaut (pour la version gratuite du service), mais vous pouvez réduire ce nombre si vous le souhaitez. Voici un exemple de formulaire proposé par Mockaroo, sur lequel j'ai juste ajouté une colonne "birthday" (il existe de nombreux types de données prédéfinis, dans lesquels vous pouvez puiser à volonté) :

Field Name	Type	Options
id	Row Number	blank: 0 % <input type="checkbox"/> <input type="checkbox"/>
first_name	First Name	blank: 0 % <input type="checkbox"/> <input type="checkbox"/>
last_name	Last Name	blank: 0 % <input type="checkbox"/> <input type="checkbox"/>
email	Email Address	blank: 0 % <input type="checkbox"/> <input type="checkbox"/>
gender	Gender	blank: 0 % <input type="checkbox"/> <input type="checkbox"/>
ip_address	IP Address v4	blank: 0 % <input type="checkbox"/> <input type="checkbox"/>
birthday	Date	01/01/1960 to 12/31/2010 in yyyy/mm/dd blank: 0 % <input type="checkbox"/> <input type="checkbox"/>

Rows: Format: ☒ array ☒ include null values

Hint: Use "." in column names to generate nested json objects, brackets to generate arrays. [More information...](#)

Want to save this for later? [Sign up for free.](#)

Les boucles

Quand on manipule de la donnée, et en particulier de gros volumes de données, il est bon de savoir à quelle méthode se fier, pour parcourir ces données en un temps record.

J'avais bien ma petite idée sur la question, mais je suis comme Saint Thomas, je ne crois que ce que je vois. Alors j'ai décidé de faire quelques mesures de performances, à partir d'un jeu de données produit par le site Mockaroo.com via les paramètres suivants :

Field Name: randvalue Type: Number Options: min: -1000000 max: 1000000 decimals: 6 blank: 0 % fx ×

Add another field

Rows: 1000 Format: JSON ☒ array ☒ include null values

Vous constaterez que je n'ai défini qu'une seule colonne, de type numérique, contenant des valeurs aléatoires comprises entre -1 million et 1 million.

Voici un échantillon du jeu de données produit par Mockaroo. Il s'agit d'un tableau d'objets, stocké dans la variable « numbers » :

```
[{"randvalue":569514.014574},
{"randvalue":-108108.724563},
{"randvalue":726166.097344},
{"randvalue":-531602.949993},
{"randvalue":-944413.838905},
{"randvalue":77111.779387},
...
{"randvalue":21858.172966},
{"randvalue":193984.930548},
{"randvalue":-159804.297247},
{"randvalue":767656.48888}]
```

Pour tester les performances de chaque type de boucle, j'ai créé une fonction que j'ai appelée « millis », qui se présente ainsi :

```
function millis() {
  return window.performance.now();
}
```

```
undefined
```

```
millis()
```

```
8324144.899999999
```

```
millis()
```

```
8326496.800000001
```

J'ai emprunté cette fonction au framework P5.js. Le principe est simple, on appelle la fonction « millis » une première fois, avant d'exécuter la boucle, et on stocke la valeur résultante dans une

variable temporaire. Après exécution de la boucle, on rappelle la fonction « millis », on fait la différence avec la valeur précédente, et on obtient le temps d'exécution exprimé en millisecondes.

Je vais présenter brièvement chaque type de boucle, avec ses avantages et inconvénients, et je vous donnerai ensuite les résultats de mon benchmark.

Boucle While

Pour itérer sur un tableau, et si l'ordre de parcours importe peu, alors on peut de la fin vers le début, en décrémentant un compteur contenant initialement le nombre d'éléments du tableau. Cela donne :

```
var somme = 0;
var imax = numbers.length;
while(imax--) {
    somme += numbers[imax].randvalue;
}
```

J'ai toujours pensé que la boucle « while » était la plus performante. C'était peut être vrai il y a quelques années, mais ça ne l'est plus aujourd'hui, comme on le verra dans le benchmark.

Boucle for à deux vitesses

Bon, la boucle « for » c'est un grand classique. Chaque langage a la sienne.

En JS, il y a la version peu optimisée :

```
somme = 0;
for(var i=0; i<numbers.length ; i++) {
    somme += numbers[i].randvalue;
}
```

... et la version optimisée :

```
somme = 0;
for(let i=0, imax=numbers.length ; i<imax ; i++) {
    somme += numbers[i].randvalue;
}
```

Dans cette seconde version, j'ai utilisé le mot réservé « let » pour déclarer non pas une, mais deux variables :

- Une variable « i » pour le compteur
- Une variable « imax » pour stocker le nombre de postes du tableau « numbers »

Le mot clé « let » est apparu avec ES6 et nous permet de déclarer des variables qui n'existent qu'à l'intérieur de la boucle. Car ce sont des variables techniques qui ne sont réellement utiles qu'à l'intérieur de la boucle, alors autant ne pas polluer l'environnement extérieur avec ces variables.

Mais la véritable optimisation réside dans le fait que la déclaration et surtout l'alimentation de la variable "imax" n'est faite qu'une seule fois, donc le comptage du nombre d'éléments du tableau n'est fait qu'une seule fois. Dans la version précédente, le comptage était effectué à chaque itération, ce qui est contre-productif.

Cependant, on verra dans le benchmark que les gains de performances obtenus avec la version 2 ne sont pas si significatifs que cela.

Boucle forEach, y'a du bon et du moins bon

Honnêtement, j'aime bien la boucle « forEach », apparue avec ES5. Elle permet d'itérer facilement sur les tableaux, mais elle présente plusieurs inconvénients, comme on va le voir dans un instant.

D'abord un petit exemple :

```
> var test = ['a', 'B', 'c', 'D'];
```

```
> test.forEach(function(item) {
    console.log(item.toLowerCase());
});
```

a

b

c

d

On peut éventuellement récupérer l'indice de chaque poste du tableau, dans une variable que j'ai appelée « index » dans l'exemple ci-dessous :

```
> test.forEach(function(item, index) {
    console.log(index + ' ' + item.toLowerCase());
});
```

0 a

1 b

2 c

3 d

On peut aussi utiliser les fonctions fléchées avec « `forEach` ». C'est une nouveauté de ES6 qui permet d'écrire ceci :

```
> var test = ['a', 'B', 'c', 'D'];
    test.forEach((item, index) => {
      console.log(index + ' ' + item.toLowerCase());
    });
```

0 a

1 b

2 c

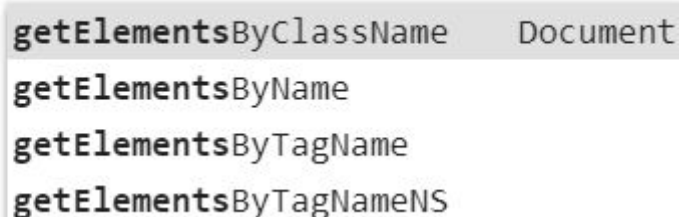
3 d

En plus d'une simplification de l'écriture, le gros atout de la fonction fléchée, c'est qu'elle n'a pas de « scope » (périmètre). Notamment, elle n'embarque pas de « `this` », ce qui règle beaucoup de problèmes inhérents à l'ancienne syntaxe. On trouve beaucoup d'articles sur le sujet sur internet, aussi je ne m'attarde pas dessus, mais si vous voulez approfondir la question, vous pouvez commencer par la page MDN suivante :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Fonctions/Fonctions_fl%C3%A9ch%C3%A9es

Avec « `forEach` », on peut aussi manipuler des tableaux « `canada dry` », je veux dire par là des tableaux qui n'en sont pas mais qui semblent en être quand même. C'est le cas notamment des « `node lists` » et des « `HTML collections` », soit des listes de nœuds du DOM produites par les fonctions suivantes :

```
> document.getElementsByClassName
```



getElementsByClassName Document
getElementsByName
getElementsByTagName
getElementsByTagNameNS

C'est le cas aussi avec la fonction « `document.querySelectorAll` ».

Par exemple, pour une « `node list` » contenant toutes les balises « `a` » d'une page HTML ayant l'attribut « `data-info` », on peut écrire :

```
var test = document.querySelectorAll('a[data-info]');
```

A l'arrivée, la variable « `test` » va contenir un jeu de données qui ressemble beaucoup à un tableau, mais qui n'en est pas un puisqu'il ne possède pas toutes les propriétés et méthodes d'un tableau. Son seul point véritable point commun avec un tableau, c'est qu'elle possède une propriété « `length` » permettant de connaître le nombre de postes... mais c'est tout.

N'étant pas un véritable tableau, elle est dépourvue de la fonction « `forEach` ». Mais on peut s'en sortir quand même, via la technique suivante :

```
> [].forEach.call(['a', 'b', 'd'], function(e) {
  console.log(e);
})
```

a

b

d

Dans l'exemple ci-dessus, j'ai utilisé un simple tableau contenant les valeurs "a", "b" et "c". Mais vous pouvez remplacer ce tableau par la variable « test » de l'exemple précédent.

On peut là aussi utiliser une fonction fléchée, comme dans l'exemple ci-dessous :

```
> [].forEach.call(['a', 'b', 'd'], e => {
  console.log(e);
})
```

a

b

d

Le principal inconvénient de la fonction "forEach", c'est qu'elle n'autorise pas l'utilisation de l'instruction "break", on ne peut donc pas l'interrompre prématurément si le besoin s'en fait sentir :

```
> [].forEach.call(['a', 'b', 'd'], e => {
  console.log(e); break;
})
```

```
✖ ▶ Uncaught SyntaxError: Illegal break statement
   at Array.forEach (<anonymous>)
   at <anonymous>:1:12
```

Boucle for of, petite nouveauté ES6

Nous avons aperçu la boucle « for of » lorsque nous avons abordé l'utilisation de l'objet Map, comme dans cet exemple :

```
> var maMap = new Map(); //=> création d'un objet Map

//définir les clés et valeurs
maMap.set("cle1", "valeur associée à clé1");
maMap.set("cle2", "valeur associée à clé2");
maMap.set("cle3", "valeur associée à clé3");

//petite boucle "for of" (nouveauté d'ES6) pour parcourir le contenu de la Map
for (var clé of maMap.keys()) {
  console.log(clé);
}
```

```
cle1
```

```
cle2
```

```
cle3
```

La documentation Mozilla en donne la définition suivante :

L'instruction for...of permet de créer une boucle Array qui parcourt un objet itérable (ce qui inclut les objets Array, Map, Set, String, TypedArray, l'objet arguments, etc.) et qui permet d'exécuter une ou plusieurs instructions pour la valeur de chaque propriété.

(source : <https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/for...of>)

La documentation Mozilla propose pas mal d'exemples intéressants, je vous laisse le soin de l'étudier, en fonction de vos besoins et de vos priorités.

Les fonctions map, filter et reduce

Les fonctions map et filter

La norme ES6 apporte un certain nombre de nouveaux itérateurs, qui rendent le parcours de tableaux plus puissant et plus fun, et qui font rentrer le Javascript de plein pied dans la programmation dite « fonctionnelle ». Si ces techniques sont puissantes, on verra dans le benchmark qu'elles ne sont pas toujours les plus performantes.

Commençons par la fonction « map ». Associée à tout objet de type tableau, elle est hyper efficace pour transformer ou reformater un jeu de données. Exemple ci-dessous :

```
var tableau = [0, 4, 5];

var test = tableau.map(x => x*10) //=> [0, 40, 50]
```

Vous noterez que j'ai utilisé une fonction fléchée, technique que nous avons vue dans un précédent chapitre.

Autre exemple avec la fonction « filter », dont le nom est particulièrement explicite :

```
var tableau = [0, 4, 5];

var test = tableau.filter(x => x > 0) //=> [4, 5]
```

On peut combiner les deux fonctions, pour notre plus grande joie :

```
var tableau = [0, 4, 5];

var test = tableau
  .map(x => x*10) // modification
  .filter(x => x > 0) ; // filtre

console.log(test); //=> [40, 50]
```

Autre exemple, cette fois à partir de clés extraites dynamiquement d'un objet qui contient des filtres, et dont on va afficher les valeurs associées à chaque filtre :

```
var filtres = {"app":"zzz", "plage":"ttt",
              "groupe":"jjj", "typtrait":"uuu"};

Object.keys(filtres).map(e=>filtres[e]);
//=> ["zzz", "ttt", "jjj", "uuu"]
```

Autre petit exemple, pour s'amuser :

```
> var tableau = [];
    tableau.push({id:1, lib:'xxx'});
    tableau.push({id:2, lib:'yyy'});
    tableau.push({id:3, lib:'zzz'});
    test = tableau
      .map(e => e.id)
      .filter(id => id % 2 === 0);
    console.log(test); //=> [2]
```

Dernier exemple, toujours pour s'amuser :

```
> var test = [];
    test.push({id:4, gender:"Male"});
    test.push({id:6, gender:"Female"});
    test.push({id:10, gender:"Male"});

    test.filter(e=>Number(e.id) >= 10 && e.gender == "Male");
< ▼ [{...}] ⓘ
  ► 0: {id: 10, gender: "Male"}
     length: 1
```

La fonction reduce

La documentation Mozilla est assez claire, elle indique ceci :

La méthode reduce() applique une fonction qui est un « accumulateur » et qui traite chaque valeur d'une liste (de la gauche vers la droite) afin de la réduire à une seule valeur.

(source : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/reduce)

Voici un premier exemple :

```
> var test = [0, 1, 2, 3, 4].reduce(function(accumulateur, valeurCourante){
    return accumulateur + valeurCourante;
});
console.log(test); //=> 10
```

On peut définir une valeur initiale, si elle est différente de zéro, comme ici avec la valeur 5, déclarée juste après la fonction « accumulatrice » :

```
> var test = [0, 1, 2, 3, 4].reduce(function(accumulateur, valeurCourante){
    return accumulateur + valeurCourante;
}, 5);
console.log(test); //=> 15
```

On peut dissocier la déclaration de la fonction accumulatrice, de son utilisation, comme dans cet exemple emprunté à la documentation Mozilla :

```
> const array1 = [1, 2, 3, 4];
    const reducer = (accumulator, currentValue) => accumulator + currentValue;

    // 1 + 2 + 3 + 4
    console.log(array1.reduce(reducer)); //=> 10

    // 5 + 1 + 2 + 3 + 4
    console.log(array1.reduce(reducer, 5)); //=> 15
```

Bref, reduce, c'est vraiment cool.

Et les perfs dans tout ça ?

Au vu des résultats qui vont suivre, vous allez constater que les meilleures performances ne viennent pas toujours d'où on les attend.

Je ne pouvais pas consacrer énormément de temps à cette analyse de performances, aussi j'ai opté pour un cas simple : l'addition d'un millier de valeurs numériques, générées au hasard entre - 1 million et 1 million. J'ai généré ce jeu de données avec Mockaroo, comme nous l'avons vu dans un précédent chapitre.

Pour effectuer mes mesures de performance, j'ai créé une petite fonction « millis » destinée à mesurer la durée d'exécution de chaque méthode de calcul. En fait, je l'ai repompée sur le projet P5.js, mais chut, je ne vous ai rien dit 😊.

```
function millis() {
  // retourne un DOMHighResTimeStamp, mesuré en millisecondes,
  // avec une précision de 5 millièmes de milliseconde (5 microsecondes).
  // cf doc MDN : https://developer.mozilla.org/fr/docs/Web/API/Performance/now
  return window.performance.now();
}
```

J'appelle cette fonction deux fois :

- Une fois avant le code dont je souhaite mesurer les performances,
- Une fois après l'exécution du code, pour faire la différence et connaître la durée d'exécution exprimée en milli-secondes.

Exemple ci-dessous :

```
var start = millis();
somme = 0;
numbers.forEach(item => {
  somme += item.randvalue;
});

console.log(millis()-start);
```

Je dois dire que ma petite analyse de performances n'a pas donné de résultats suffisamment significatifs pour mettre sur la touche une technique ou une autre. Cela permet tout au plus de dégager quelques tendances.

On peut noter que :

- la boucle "while" s'est trouvée de temps à autre en mauvaise posture côté performances, pourtant elle retourne un résultat plus précis que ses consœurs
- la boucle "for non optimisée (sur length)" désigne une boucle pour laquelle le comptage du nombre d'éléments du tableau est effectué à chaque itération : contre toute attente, ses performances ne sont pas si mauvaises, même si de temps en temps elle semble un peu à la peine

- la fonction “reduce” est de temps en temps très en retard sur ses concurrentes, mais c’est loin d’être systématique

Je vous propose ci-dessous un petit échantillon des tests que j’ai effectués.

Tests, itération N+2 :

Type de boucle	Durée	Résultat (somme)
Boucle While	0.19999995129182935	2742506.564948983
Boucle For non optimisée (sur length) et sans let	0.3000000142492354	2742506.5649489877
Boucle For optimisée (sur length) et sans let	0.20000000949949026	2742506.5649489877
Boucle For non optimisée (sur length) et avec let	0.20000000949949026	2742506.5649489877
Boucle For optimisée (sur length) et avec let	0.09999994654208422	2742506.5649489877
Boucle forEach avec fonction	0.20000000949949026	2742506.5649489877
Boucle forEach avec =>	0.10000000474974513	2742506.5649489877
Méthode reduce avec fonction	0.20000000949949026	2742506.5649489877
Méthode reduce avec =>	0.3000000142492354	2742506.5649489877

Tests, itération N + 5:

Type de boucle	Durée	Résultat (somme)
Boucle While	0.4000000189989805	2742506.564948983
Boucle For non optimisée (sur length) et sans let	0.3000000142492354	2742506.5649489877
Boucle For optimisée (sur length) et sans let	0.3000000142492354	2742506.5649489877
Boucle For non optimisée (sur length) et avec let	0.20000000949949026	2742506.5649489877
Boucle For optimisée (sur length) et avec let	0.2999999560415745	2742506.5649489877
Boucle forEach avec fonction	0.5000000237487257	2742506.5649489877
Boucle forEach avec =>	0.49999996554106474	2742506.5649489877
Méthode reduce avec fonction	0.9999999892897904	2742506.5649489877
Méthode reduce avec =>	1.500000013038516	2742506.5649489877

Tests, itération N+10 :

Type de boucle	Durée	Résultat (somme)
Boucle While	2.299999992828816	2742506.564948983
Boucle For non optimisée (sur length) et sans let	0.20000000949949026	2742506.5649489877
Boucle For optimisée (sur length) et sans let	0.20000000949949026	2742506.5649489877
Boucle For non optimisée (sur length) et avec let	0.20000000949949026	2742506.5649489877
Boucle For optimisée (sur length) et avec let	0.10000000474974513	2742506.5649489877
Boucle forEach avec fonction	0.20000000949949026	2742506.5649489877
Boucle forEach avec =>	0.10000000474974513	2742506.5649489877
Méthode reduce avec fonction	0.4000000189989805	2742506.5649489877
Méthode reduce avec =>	0.5000000237487257	2742506.5649489877

Au fait, pourquoi la fonction “while” retourne-t-elle un résultat plus précis que ses concurrents ?

Comme j’avais la tête dans le guidon lors de la rédaction de ce dossier, je n’ai pas compris tout de suite la raison, et c’est en discutant avec les amis du meetup CreativeCodeParis que j’ai eu l’illumination. Enfin cette différence de précision était purement accidentelle, car la boucle “while”,

telle que j'ai l'ai écrite (`while(imax--)`) a pour effet de parcourir le tableau à partir de la fin, et comme les calculs se font sur des nombres en virgule flottante, c'est purement un coup de chance si la somme des nombres effectuée en sens inverse fournit ce résultat.

Mais ce qui est intéressant à noter, c'est qu'il doit être possible d'améliorer la précision des calculs, et c'est que nous vérifierons dans un chapitre ultérieur avec la fonction "somme_sioux".

L'API Webworker

L'API Webworker est une API géniale, d'autant plus géniale qu'elle est vraiment facile à utiliser.

Elle vous permet de créer un thread secondaire, parallèle du thread principal, dans lequel vous allez pouvoir effectuer des tâches consommatrices de CPU sans impacter les performances du thread principal.

Je m'en suis servi récemment, à deux reprises, à peu près de la même manière :

- dans le premier cas il s'agissait de déporter le calcul d'une fractale dans un thread secondaire, et d'envoyer les données par paquets au thread principal. Ce dernier affichait progressivement la fractale via l'API canvas, en prenant au fur et à mesure de leur arrivée, les paquets de données calculées par le Webworker
- dans le second cas, il s'agissait d'un projet Dataviz à vocation professionnelle. Dans le web worker, je lançais une requête HTTP vers un serveur, via l'API Fetch (encore une nouveauté d'ES6), pour récupérer un jeu de données, que je retravaillais via des requêtes SQL pilotées par le projet AlaSQL (cf. chapitre suivant), avant de les renvoyer au thread principal pour les afficher dans un graphe généré par D3.js.

Dans ces deux exemples, le Webworker m'a permis de déporter des traitements lourds, et ainsi de ne pas bloquer la navigation de l'internaute.

La communication entre le Webworker et le thread principal se fait au moyen de messages dans lesquels on transmet des tableaux de valeur.

Comme je manque de temps pour finaliser ce document avant la présentation du 20 décembre, je ne vais pas détailler la mécanique du Webworker ici, mais je vous invite à étudier le code source de mon générateur de fractale qui se trouve dans le dépôt suivant :

<https://github.com/gregja/mandelP5ot>

Pour un exemple plus simple, vous pouvez vous reporter à celui qui est fourni dans la documentation Mozilla :

<https://github.com/mdn/simple-web-worker>

AlaSQL

Je ne pouvais pas clore ce dossier sans parler de AlaSQL, tant ce projet m’a rendu service dans un récent projet Dataviz (utilisé conjointement avec D3.js).

AlaSQL est un portage de SQL en Javascript, et c’est un projet particulièrement réussi.

Site officiel : <http://alasql.org/>

Dépôt Github : <https://github.com/agershun/alasql>

Documentation : <https://github.com/agershun/alasql/wiki/Getting-started>

On peut utiliser AlaSQL aussi bien côté navigateur que dans Node.js.

Côté navigateur, il suffit d’ajouter la ligne suivante pour mettre en place l’outil :

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/alasql/0.4.11/alasql.min.js"></script>
```

A partir de là, il devient facile d’exécuter tout type de requête SQL, en utilisant pour source de données vos propres tableaux Javascript. Dans le cas de notre test de performances, si on souhaite cumuler via AlaSQL, les 1000 valeurs contenues dans le tableau “numbers”, cela nous donne :

```
var sql = "SELECT SUM(x.randvalue) AS somme FROM ? AS x";
var xdatas = alasql(sql, [numbers]);
console.log(xdatas);    //=<> [{"somme":2742506.5649489877}]
```

Explication : le second paramètre passé à la fonction “alasql” est impérativement un tableau, d’où la présence des crochets pour encapsuler la variable “numbers” (qui est elle-même un tableau). Si vous oubliez ces crochets, vous obtiendrez un résultat incohérent, mais pas de message d’erreur. Car pour AlaSQL ce n’est pas une erreur : il va simplement compter le nombre de points d’interrogation contenus dans la requête, et les remplacer par les valeurs contenus dans le second paramètre.

Dans la documentation officielle, on trouve l’exemple suivant :

```
/* create SQL Table and add data */
alasql("CREATE TABLE cities (city string, pop number)");
alasql("INSERT INTO cities VALUES
('Paris',2249975),('Berlin',3517424),('Madrid',3041579)");
/* execute query */
var res = alasql("SELECT * FROM cities WHERE pop < 3500000 ORDER BY pop DESC");
// res=[{"city": "Madrid", "pop": 3041579 }, {"city": "Paris", "pop": 2249975}]
```

Je vous propose cette variante, strictement équivalente d'un point de vue fonctionnel, elle vous permettra - je pense - d'évaluer la souplesse de l'outil.

```
var data = [];
data.push({ "city": "Madrid", "pop": 3041579 });
data.push({ "city": "Paris", "pop": 2249975 });
data.push({ "city": "Berlin", "pop": 3517424});
var res = alasql("SELECT * FROM ? WHERE pop < ? ORDER BY pop DESC",
    [data, 3500000]);
console.log(res);
//=> [{"city":"Madrid","pop":3041579},{ "city":"Paris","pop":2249975}]
```

J'ai pu réaliser des requêtes bien plus complexes que celle-ci, mais comme j'ai manqué de temps pour finaliser ce dossier avant sa présentation, j'ai préféré laisser le sujet en suspens. J'espère le reprendre ultérieurement pour vous proposer des exemples plus intéressants.

Il faut souligner que les performances d'AlaSQL ne sont pas aussi bonnes, à fonctionnalité équivalente, que celles des fonctions itératives telles que "map", "filter" ou "reduce". C'est donc surtout sur des requêtes complexes que AlaSQL donne toute sa mesure. Je tiens à souligner que AlaSQL sait gérer des CTE (Common Table Expression) une technique SQL très puissante, dont vous pouvez voir un aperçu sur cette page :

<https://github.com/agershun/alasql/wiki/With>

Le lecteur intéressé par SQL pourra approfondir le sujet au travers d'un de mes supports de cours, qui se trouve dans le dépôt suivant :

<https://github.com/gregja/SQLCorner/>

(ce support de cours contient un chapitre "bibliographie" fournissant une liste de bonnes références sur SQL, ainsi que la liste des articles que j'ai publiés sur ce sujet dans GNU/Linux Magazine).

Cet autre dépôt contient un support de cours expliquant comment utiliser MariaDB (ou MySQL) avec NodeJS :

<https://github.com/gregja/NodeJSCorner>

Opérations sur grands nombres

Problèmes de précision

Les langages Javascript et PHP souffrent tous deux d'un manque de précision dans la manipulation des nombres décimaux. Petite démonstration dans la console de notre navigateur :

```
> .1 + .3
< 0.4

> .1 - .3
< -0.19999999999999998

> 0.1+0.7
< 0.7999999999999999
```

Quelquefois ça tombe juste, quelquefois ça tombe à côté... « non mais Allo quoi !!! » (*)

Pour expliquer ce phénomène, j'aime bien me référer à la documentation de PHP, je trouve qu'elle est particulièrement claire, en voici un extrait :

« Les nombres décimaux ont une précision limitée. Même s'ils dépendent du système, PHP utilise le format de précision des décimaux IEEE 754, qui donnera une erreur maximale relative de l'ordre de $1.11e-16$ (dûe aux arrondis). Les opérations arithmétiques non-élémentaires peuvent donner des erreurs plus importantes et bien sûr les erreurs doivent être prises en compte lorsque plusieurs opérations sont liées.

*Aussi, les nombres rationnels exactement représentables sous forme de nombre à virgule flottante en base 10, comme 0.1 ou 0.7, n'ont pas de représentation exacte comme nombres à virgule flottante en base 2, utilisée en interne, et ce quelle que soit la taille de la mantisse. De ce fait, ils ne peuvent être convertis sans une petite perte de précision. Ceci peut mener à des résultats confus: par exemple, `floor((0.1+0.7)*10)` retournera normalement 7 au lieu de 8 attendu, car la représentation interne sera quelque chose comme 7.999999999999999118....*

Ainsi, ne faites jamais confiance aux derniers chiffres d'un nombre décimal, mais aussi, ne comparez pas l'égalité de 2 nombres décimaux directement. ... »

Source : <http://php.net/manual/fr/language.types.float.php>

Le langage PHP fournit une solution de contournement avec les fonctions BC Math, spécialisées dans la manipulation de nombre de grande taille. Le langage Javascript n'a pas de solution équivalente pour l'instant.

Mais plusieurs projets Javascript essaient de proposer des solutions palliatives pour corriger ce problème de précision. Je ne les ai pas testés et ne pourrai pas vous faire de retour d'expérience,

mais en voici une petite liste : `big.js`, `bignumber.js`, `decimal.js` et `math.js` (ce dernier étant peut être le plus connu et potentiellement le plus utilisé).

Pour les deux exemples boiteux de la page précédente, on peut s'en sortir avec une formule de ce type.

```
> Math.round((.1-.3)*10)/10
< -0.2
```

```
> Math.round((.1+.7)*10)/10
< 0.8
```

Autre solution envisageable, utiliser la méthode « `toPrecision` » associée aux variables de type numérique, comme dans cet exemple :

```
> var test = .1+.7
< undefined
```

```
> test
< 0.7999999999999999
```

```
> test.toPrecision(3)
< "0.800"
```

Ce n'est pas mal, mais la valeur renvoyée par la méthode « `toPrecision` » est une chaîne de caractères. On peut cependant s'en tirer en convertissant cette chaîne en numérique, comme ceci :

```
> var test2 = Number(test.toPrecision(3));
< undefined
```

```
> test2
< 0.8
```

Au fait, on peut aller jusqu'à quelle précision avec cette méthode « `toPrecision` » avant que les décimales ne « partent en sucette » ? J'ai testé pour vous, et pour économiser du papier, je vous donne seulement la fin du test :

```
> test.toPrecision(13)
< "0.8000000000000"
> test.toPrecision(14)
< "0.80000000000000"
> test.toPrecision(15)
< "0.800000000000000"
> test.toPrecision(16)
< "0.799999999999999"
```

Si vous êtes courageux, et fort en maths, il y a un très bon papier sur les problèmes de calcul sur nombres en virgule flottante, qui s'intitule :

"What Every Computer Scientist Should Know About Floating-Point Arithmetic"

https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

Le problème des faux dépassements

Le problème de la précision de calcul est relativement connu, mais il y en a un autre auquel on ne pense pas forcément, et qui va devenir plus fréquent dès lors que les développeurs Javascript vont utiliser des jeux de données plus importants, c'est celui des calculs en overflow. Lorsque l'on a besoin de sommer un grand nombre de valeurs, il est préférable de recourir à une ruse de sioux, comme celle que j'ai découverte dans un vieux recueil de corrigé d'examen. Le recueil fournissait un algorithme, je vous propose ci-dessous une adaptation en JS (les explications suivent sur la page suivante) :

```
/**
 * Calcul de somme sans faux dépassement
 * Adaptation au JS d'un algorithme extrait du livre :
 *   Algorithmique, exercices corrigés, oral du concours d'entrée à l'ENS de Lyon,
 *   édition Ellipses 1993
 * @param {[type]} datas [description]
 * @return {[type]}      [description]
 */
function somme_sioux(datas) {
  var Pos = [];
  var Neg = [];
  var npos = 0, nneg = 0, accu = 0;

  for (let i=0, imax=datas.length; i<imax ; i++) {
    if (datas[i] >= 0) {
      npos += 1;
      Pos[npos] = datas[i];
    } else {
      nneg += 1;
      Neg[nneg] = datas[i];
    }
  }
  while(npos > 0 && nneg > 0) {
    if (accu >= 0) {
      accu += Neg[nneg];
      nneg -= 1;
    } else {
      accu += Pos[npos];
      npos -= 1;
    }
  }
  if (npos > 0) {
    for (let i=1; i <= npos; i++) {
      accu += Pos[i];
    }
  } else {
    for (let i=1; i <= nneg; i++) {
      accu += Neg[i];
    }
  }
  return accu;
}
```

La méthode utilisée dans l'algorithme est la suivante :

- on sépare les données du tableau d'origine en deux tableaux, l'un contenant les termes positifs, et l'autre les termes négatifs
- on utilise une variable "accu" qui représente la somme des nombres déjà ajoutés
- sachant que l'on n'a pas forcément autant de termes positifs que négatifs, tant qu'il existe à la fois des termes positifs et négatifs, on les ajoute 2 à 2 dans "accu"
- dès qu'il n'existe plus que des termes positifs ou négatifs, on les ajoute à "accu" jusqu'à épuisement du stock
- en procédant de la sorte, on réduit significativement les risques dits de "faux dépassement"

Dans la version précédente du présent support, j'avais oublié de fournir les résultats obtenus avec la fonction `somme_sioux`, l'oubli est réparé ci-dessous :

N°	Type de boucle	Durée	Résultat
1	Boucle While	0.600000000304135	2742506.564948983
2	Boucle For non optimisée (sur length) et sans let	0.600000000304135	2742506.5649489877
3	Boucle For optimisée (sur length) et sans let	0.4999999991923687	2742506.5649489877
4	Boucle For non optimisée (sur length) et avec let	0.600000000304135	2742506.5649489877
5	Boucle For optimisée (sur length) et avec let	0.3999999998995918	2742506.5649489877
6	Boucle forEach avec fonction	0.4999999991923687	2742506.5649489877
7	Boucle forEach avec =>	0.3999999998995918	2742506.5649489877
8	Méthode reduce avec fonction	0.29999999969732016	2742506.5649489877
9	Méthode reduce avec =>	0.3999999998995918	2742506.5649489877
10	fonction <code>somme_sioux</code>	1.4000000001033186	2742506.564949

On voit que la fonction `somme_sioux` offre une bien meilleure précision, mais bien évidemment cela se paye au niveau des performances.

Conclusion

J'espère que ce dossier vous aura aidé à y voir plus clair dans les possibilités de Javascript pour la manipulation de données. Il n'a pas la prétention d'être exhaustif, il y a certainement des omissions. Le langage Javascript est en évolution constante, et pour ma part je découvre sans cesse des techniques nouvelles en travaillant avec ce langage. Comme j'ai la chance aujourd'hui de travailler sur des projets impliquant la manipulation de gros volumes de données en Javascript (à la fois côté front et côté serveur avec NodeJS), je pense que je vais découvrir encore plein de choses, dont j'essaierai de vous faire profiter dans une prochaine version de ce document.

En attendant, je vous invite à parcourir la bibliographie qui suit, elle contient d'excellentes références pour approfondir certains des sujets que nous avons abordés.

A très bientôt ;)

Bibliographie

Plusieurs ouvrages sont parus dans le courant de l'année 2018, autour de la manipulation de données en Javascript. N'ayant découvert certains de ces livres que très récemment, je n'ai pas eu le temps de les exploiter pleinement, mais je pense qu'ils sont tous très intéressants, et très complémentaires.

Data Wrangling with JavaScript

par Ashley Davis

December 2018 ISBN 9781617294846 432 pages

<https://www.manning.com/books/data-wrangling-with-javascript>

Hands-on Machine Learning with JavaScript

par Burak Kanber

Packt, may 2018

<https://www.packtpub.com/big-data-and-business-intelligence/hands-machine-learning-javascript>

Functional-Light JavaScript (Balanced, Pragmatic FP in JavaScript)

par Kyle Simpson

Published on 2018-02-09

<http://leanpub.com/fljs>

Functional Programming in JavaScript

(How to improve your JavaScript programs using functional techniques)

par Luis Atencio

June 2016 ISBN 9781617292828 272 pages

<https://www.manning.com/books/functional-programming-in-javascript>

Le seul livre de cette sélection que je n'ai pas lu du tout (faute de temps), et le seul aussi qui soit disponible gratuitement en ligne, c'est celui-ci :

Learn JS Data (Data cleaning, manipulation, and wrangling in Javascript)

par The Bocoup Data Vizualization Team

<http://learnjsdata.com/>

Changelog

version 0.8 : présentée au meetup CreativeCodeParis du 20 décembre 2018

version 1.0 : correction de nombreuses coquilles, suppression de redites, ajout de précisions dans la chapitre sur les performances, ajoute de précisions également dans le chapitre “opérations sur grands nombres” (concernant la fonction “somme_sioux”).