Les fiches récap de l'école O'clock

⚠ this

Attention cette fiche récapitulative nécessite d'être revalidée par un formateur.
Les informations contenues peuvent être obsoletes.

Rôle et usages de `this`

À quoi sert `this` ? Comment s'en servir ? Réponses à travers une histoire de dev.

Le point de départ

Un Dev est confronté au problème suivant :

> JavaScript ne fournit pas de moyen simple pour réaliser la somme des éléments numériques d'un tableau. On peut utiliser une boucle, ou même `reduce`, mais l'idéal serait de pouvoir écrire : `unTableau.sum()`. Est-il possible d'ajouter ça au langage ?

Pour reformuler le problème, étant donnée une variable `unTableau` contenant un tableau de nombres, on souhaite pouvoir écrire :

```
var unTableau = [1, 2, 3];
unTableau.sum(); // doit renvoyer la valeur numérique 6
```

Attenion : `sum` doit fonctionner avec *n'importe quel* tableau de nombres. Comme JavaScript ne fournit pas, de base, une méthode `sum`, c'est à nous d'*étendre* les possibilités du langage.

Heureusement, c'est possible, et `this` intervient dans l'affaire.

Partons donc du postulat que `var unTableau = [1, 2, 3]`. La notation `[]` est un sucre syntaxique, un raccourci. On dit qu'elle permet de créer un tableau *litéral*, car la syntaxe elle-même ressemble à un tableau. Mais il est possible créer le même tableau, de façon plus « orientée objet », en utilisant le type primitif `Array` :

```
var unTableau = new Array();
// L'instruction ci-dessus retourne "undefined" (essayez en console)
;
// mais elle a eu un effet de bord, unTableau existe désormais :
unTableau; // renvoie "Array []" => le tableau a été créé, vide !
```

> En JS, n'importe quel tableau créé est fondamentalement du « type » `Array`. Techniquement, `Array` est une fonction-constructeur.

Si on veut se donner une méthode `sum` qui soit utilisable *avec n'importe quel tableau*, c'est donc sur ce type primitif `Array` qu'il faut travailler.

Ça tombe bien, JS nous permet d'ajouter des méthodes à un type, en passant par son prototype :

```
Array.prototype.sum = function() {
  // TODO: faire la somme des éléments du tableau
};
```

> Le prototype d'un type (ex. `Array.prototype`) est un espace de stockage de méthodes partagées et accessibles par toutes les « instances » de ce type (ex. `unTableau`). [TODO: fiche récap sur la notion de prototype].

Le problème qui se pose alors, est d'arriver à implémenter `sum` « en toute généralité », c'est-à-dire faire en sorte que `sum` fonctionne sur n'importe quel tableau.

La nécessité de `this`

`sum` étant une fonction, on pourrait se dire : « elle n'a qu'à prendre en argument le tableau dont on veut faire la somme ! » Certes, mais on souhaite utiliser `sum` de la façon suivante :

`unTableau.sum()`. Et oui, ce serait un peu redondant d'écrire `unTableau.sum(unTableau)` ! Donc pas d'argument…

Si on y réfléchit bien, écrire `unTableau.sum()` c'est un peu comme envoyer un message (« exécute `sum` ») à un destinataire (`unTableau`) — le message étant : « merci de calculer la somme de tes éléments, et de me la retourner ». Cette logique de message/destinataire devrait pouvoir fonctionner non seulement avec `unTableau` , mais aussi avec n'importe quel tableau auquel on enverrait ce même message `.sum()` . Il faut donc pouvoir travailler en toute généralité, c'est-à-dire définir `sum` sans faire référence à un tableau en particulier.

Comment faire référence, dans `sum` , à un tableau « en général » ?

L'astuce est la suivante :

> **Quand on appelle une méthode sur un objet, dans la définition de la méthode, le destinataire du moment est référencé sous le nom `this` .**

Utilisation de `this` avec une `function`

Définissons une fonction `whoami` (== *qui suis-je ?*) sur le type le plus générique qui soit, `Object` , afin de regarder comment se comporte `this` :

```
Object.prototype.whoami = function() {
  return this;
};
```

> `Object` est le type-primitif racine : tous les objets JS en héritent (des tableaux aux fonctions), ce qui rend la méthode utilisable partout.

Utilisons cette méthode sur différents objets :

```
[1,2,3].whoami(); // retourne Array [ 1, 2, 3 ]
"salut".whoami(); // retourne String { "salut" }
true.whoami(); // retourne Boolean { true }
```

La console JS a le bon goût de nous indiquer le type de la valeur (`Array` , `String` , `Boolean`) en plus de la valeur (`this`) elle-même ; mais, fondamentalement, à chaque appel à `whoami` ,

`this` a pris pour valeur le destinataire du message `.whoami()`.

En JavaScript, `this` est donc un mot-clé *générique* dont la valeur exacte sera définie « sur le moment », c'est-à-dire au moment de l'appel à la fonction qui l'utilise. Très pratique pour écrire des définitions généralistes de méthodes !

Implémentation de `sum` avec `this`

En utilisant la nature contextuelle de `this`, il est possible de coder « en toute généralité ».

Pour implémenter `sum`, on va définir une méthode `Array.prototype.sum`, ce qui la rend disponible sur tous les tableaux, et d'une manière telle que `sum` réalise la somme des éléments d'un tableau quelconque référencé par `this` :

```
Array.prototype.sum = function() {
 // la valeur de this sera définie "at runtime", quand on lancera
.sum()
 return this.reduce((cumul, item) => cumul + item, 0);
};
```

Ce qui permet ensuite de faire :

```
[1, 2, 3].sum(); // 6
[2, 65, 9, 81].sum(); // 157
(new Array).sum(); // essayez en console !
```

Le cas `grades` (calcul de moyenne)

Voici une implémentation possible d'une fonction de moyenne, s'appuyant sur `sum` :

```
// En utilisant la syntaxe ES6 (fonction fléchée, spread operator) :
Array.prototype.average = (...grades) => grades.sum() /
grades.length;
```

> L'expression `...grades` (avec le *rest operator* `...` d'ES6) transforme une série de paramètres passés à la fonction `average` (par exemple `1, 2, 3`) en un seul

> tableau (`[1, 2, 3]`). Il devient ainsi possible de passer un nombre arbitraire de nombres à `average` mais de travailler avec un seul tableau dans sa définition.

Dans le code ci-dessus, `grades.sum()` déclenche une éxécution de la méthode `sum` implémentée dans `Array.prototype.sum` .

`this` et les fonctions fléchées

Reprenons la fonction `whoami` , en changeant simplement la syntaxe de sa définition pour utiliser une fonction fléchée :

```
// Object.prototype.whoami = function() {
Object.prototype.whoami2 = () => {
  return this;
};
```

Petit test en console, dans le navigateur :

```
"salut".whoami2(); // retourne la valeur Window
// et c'est pareil pour une instruction [1,2,3].whoami2() ou
n'importe quel autre appel à whoami2 !
```

Avec la version fléchée, `this` a pris pour valeur non pas le destinataire du message (`"salut"`), mais le contexte dans lequel l'instruction `"salut".whoami2()` a été exécutée, c'est-à-dire la valeur de `this` au moment de l'appel. Dans la *Console* des DevTools, ce contexte est par défaut l'objet `window` , d'où la valeur de `this` .

On peut le vérifier :

```
// toujours en console :
this === window === "salut".whoami(); // retourne false (car la
function crée une nouvelle portée de variable)
this === window === "salut".whoami2(); // retourne true (car la () =>
{} réutilise la portée du contexte d'appel)
```

En résumé, les fonctions fléchées conservent le `this` parent (les détails ici). Cette propriété est

très pratique pour gérer les *handlers/callbacks* dans les applications web.

Attention, ce n'est pas toujours souhaitable ! Par exemple, pour les méthodes d'un objet :

```
// Un chronomètre
const app = {
  start: function() {
    this.startTime = Date.now(); // this référence l'objet app, car on fera app.start()
  },

  stop: () => { // app.stop() mais…
    this.endTime = Date.now(); // … pas le bon this, il vaut Window au lieu de app !
    this.elapsed = this.endTime - this.startTime; // Calcul erroné, this.startTime indéfini
  }
};
```

Résumé

> You can think of `this` as an invisible argument which is passed to every method, and is defined automatically by the browser to refer to the object which the method was called from. http://jamesknelson.com/demystifying-javascript-the-many-faces-of-this/