

Javascript module 3

Les fonctions

Pourquoi des fonctions ?	2
Déclaration classique d'une fonction	2
Appeler une fonction	3
Fonction avec un seul paramètre	3
Une fonction, plusieurs paramètres, plusieurs sorties possibles	3
Portée des variables	4
Fonction anonyme	6
Callback functions	6
Fonction anonyme comme callback	7
Fonctions récursives	8
Méthode d'un objet	10
Le this	10
Contexte du this	11
Fonctions fléchées	13

Pourquoi des fonctions ?

Les fonctions permettent en premier lieu de déclarer des instructions qui ont vocation à être utilisées à plusieurs reprises dans le cours de l'exécution du code.

Au-delà du fait d'éviter les répétitions de code, les fonctions permettent de structurer le code et de le simplifier. En effet, en découpant le code en tâches simples et bien nommées, l'exécution globale peut se résumer à une suite d'appels de fonctions successives et imbriquées.

Les fonctions prennent potentiellement une ou plusieurs valeurs en entrée et renvoient potentiellement une valeur.

Déclaration classique d'une fonction

Une fonction se déclare en commençant par le mot clé **function** suivi du nom de la fonction qui doit respecter les mêmes règles de nommage que les variables (camelCase, ...).

Entre les parenthèses **()** qui suivent le nom de la fonction se trouve la liste des paramètres que l'on peut passer à la fonction, séparés par des virgules. Il peut y avoir autant de paramètres que vous le souhaitez, comme il peut ne pas y en avoir du tout. Ce sont les données en entrée de la fonction.

Ensuite, entre accolades **{ }** se trouvent les instructions qui seront exécutées par la fonction. Dans le code qui représente les instructions de la fonction, les paramètres déclarés précédemment sont considérés comme des variables.

Dans sa forme la plus classique, la déclaration d'une fonction se termine par une ligne commençant par le mot clé **return**, suivi de la valeur que doit retourner la fonction. Sachez que **return** met fin à l'exécution de la fonction, quel que soit son emplacement dans les instructions.

Voici donc le schéma basique de déclaration d'une fonction :

```
function myFunction(input) {  
    // instructions ...  
    return output;  
}
```

Attention: Le code contenu dans la fonction n'est pas exécuté à l'emplacement de la déclaration. Il ne sera exécuté que lors de l'appel de la fonction.

Appeler une fonction

Fonction avec un seul paramètre

L'appel d'une fonction se fait directement en écrivant son nom, suivi de parenthèses **()** qui contiennent les valeurs à appliquer à chacun des paramètres, séparés par des virgules. Ces paramètres peuvent être directement des valeurs (2, "My text", true, 4 > 3, ...) ou alors une variable qui contient cette valeur.

A l'emplacement de l'appel de la fonction, celle-ci sera remplacée par la valeur retournée par un **return**.

Voici un exemple de déclaration de fonction qui permet de retourner le facteur à appliquer pour un pourcentage d'augmentation donné :

```
function percentIncrease(percent) {  
    return 1 + percent / 100;  
}
```

Et voici 2 appels de cette fonction :

```
const rateA = percentIncrease(13);  
// 1.13  
  
const rateB = percentIncrease(21);  
// 1.21
```

La valeur retournée par la fonction en fonction du paramètre est directement stockée dans la variable (**rateA**, puis **rateB**).

Une fonction, plusieurs paramètres, plusieurs sorties possibles

Comme vu précédemment il est possible de déclarer plusieurs paramètres pour une fonction. Et il est également possible de prévoir plusieurs valeurs en retour. En voici un exemple :

```
function multiplyWithMax(a, b, max) {  
    const m = a * b;  
    if (m > max) return max;  
    return m;  
}
```

Cette fonction prend donc 3 paramètres. Il multiplie les 2 premiers paramètres **a** et **b**.

Si le résultat de cette multiplication est supérieur au paramètre **max**, il retourne la valeur de **max**. Le 1er **return** arrête l'exécution de la fonction, la dernière ligne ne sera donc pas exécutée.

```
multiplyWithMax(5, 10, 40); // 40
```

Si le résultat de cette multiplication n'est pas supérieur au paramètre **max**, il retourne alors le résultat de la multiplication avec le second **return**.

```
multiplyWithMax(2, 3, 10); // 6
```

Portée des variables

Les variables sont accessibles dans les contextes où elles sont déclarées et dans les contextes sous-jacents.

Voici un exemple reprenant les cas d'usage les plus courants :

```
const a = 42;
const b = 777;
let b2 = 7000;

function myfunction(f) {
  const b = 420;
  b2 = 9000;
  const c = 666;
  console.log("Into the function:", a, b, b2, c, d, e, f);
}

const d = "text";

myfunction(12); // A
// ReferenceError: e is not defined

const e = "TEXT";

console.log("Outside the function:", a, b, b2, c, d, e, f); // B
// ReferenceError: c is not defined
```

Variable	appel de myfunction() = A		contexte global = B	
	Valeur	Explication	Valeur	Explication
a	42	Prend la valeur déclarée en dehors de la fonction car le corps de la fonction hérite du contexte global.	42	Prend la valeur déclarée dans le contexte global.
b	420	Prend la valeur qui a été définie dans le contexte de la fonction. Sa valeur est donc 420. (et non 777, qui est sa valeur dans le contexte global)	777	Prend la valeur déclarée dans le contexte global. Et non celle de la fonction.
b2	9000	Prend la valeur modifiée dans le contexte de la fonction, tout comme b.	9000	Prend la valeur modifiée par l'appel de la fonction. En l'absence de re-déclaration avec const dans la fonction, c'est la valeur dans le contexte de déclaration initial de la variable qui a été changée à l'appel de la fonction.
c	666	A sa valeur définie dans le contexte de la fonction.	ERROR	Cette variable est déclarée uniquement dans le contexte de la fonction. Elle n'est donc pas accessible en dehors
d	"text"	Prend la valeur du contexte global car la variable est déclarée AVANT l'appel de la fonction.	"text"	Prend la valeur déclarée dans le contexte global.
e	ERROR	La variable n'est pas définie dans ce contexte, car l'appel de la fonction se fait avant l'initialisation de la variable.	"TEXT"	Prend la valeur déclarée dans le contexte global.
f	12	Prend la valeur définie au premier paramètre lors de l'appel de la fonction.	ERROR	Cette variable n'est pas déclarée dans ce contexte. Il s'agit du paramètre de la fonction, uniquement accessible dans celle-ci.

Fonction anonyme

Il est bon de savoir qu'en javascript tout est variable ou valeur. Et les fonctions ne font pas exception.

En effet, le **function** est un type de valeur comme un autre. On peut le voir avec cet exemple :

```
console.log(typeof multiplyWithMax);  
// function
```

C'est pourquoi les 2 déclarations de fonction suivantes sont équivalentes :

```
function multiplyWithMax(a, b, max) {  
    const m = a * b;  
    if (m > max) return max;  
    return m;  
}  
  
const multiplyWithMax = function(a, b, max) {  
    const m = a * b;  
    if (m > max) return max;  
    return m;  
}
```

Cette deuxième façon de déclarer une fonction sans nom est appelée fonction anonyme (anonymous function).

Attention: Lorsque vous utilisez une déclaration de fonction anonyme sous la forme d'une variable, les appels de la fonction doivent impérativement se trouver après cette déclaration. Une déclaration classique avec **function** fonctionnera dans tous les cas, que les appels soient situés avant ou après la déclaration.

Callback functions

Les callback functions - fonctions de rappel - sont des fonctions que l'on utilise comme paramètres d'une autre fonction.

C'est un usage classique en Javascript. Par exemple, nous avons déjà croisé ce principe en utilisant la méthode **.forEach()** sur les array, qui prend en paramètre une fonction.

Voici un autre exemple simple où les 2 fonctions, celle d'appel et celle de callback, sont toutes les 2 implémentées dans le code.

```
function displayMaximum(array, display) {
    display(array.reduce((a, b) => Math.max(a, b)));
}

function getTheValueInConsole(value) {
    console.log(`The final result is ${value}.`)
}

const values = [12, 45, 64, 34, 74];

displayMaximum(values, getTheValueInConsole);
// The final result is 74.
```

La fonction **displayMaximum()** prend 2 paramètres :

- un array dont elle va traiter les valeurs qu'il contient (ici extraire la valeur maximum) ;
- une fonction de callback auquel elle va passer le résultat du traitement précédent en paramètre.

La fonction **getTheValueInConsole()** prend un simple paramètre qu'elle affiche dans la console dans une phrase. C'est la fonction qui sera utilisée comme fonction de callback.

L'appel de la fonction à la dernière ligne, prend donc en second paramètre le nom de la fonction de callback, sans les parenthèses, comme une variable.

Fonction anonyme comme callback

Dans beaucoup de cas d'usage, la fonction de callback n'est pas déclarée en tant que tel et on utilise alors une fonction anonyme. C'est un classique par exemple avec un **array.forEach()**.

Si on utilise une fonction anonyme dans notre exemple précédent, cela donnerait ceci :

```
function displayMaximum(array, display) {
    display(array.reduce((a, b) => Math.max(a, b)));
}

const values = [12, 45, 64, 34, 74];
```

```
displayMaximum(values, function(value) {
    console.log(`The final result is ${value}.`)
});
// The final result is 74.
```

Fonctions récursives

Les fonctions récursives permettent de traiter des données en répétant une opération simple jusqu'à l'obtention du résultat souhaité. C'est une autre façon de déclarer une boucle.

Une fonction récursive est composée de 2 parties :

- Le cas de base (ou la condition d'arrêt de la boucle) qui permet de sortir de la récursivité.
- Le cas de propagation ou déclaration de la récursivité, qui appelle la fonction elle-même avec de nouveaux paramètres.

En voici un exemple pour une fonction effectuant le total des valeurs d'un array.

```
function arraySum(array) {
    if (array.length === 1) return array[0];
    return array.pop() + arraySum(array);
}

console.log(arraySum([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]));
// 55
```

Dans l'exemple ci-dessus, l'array passé en paramètre voit la valeur située à son dernier index cumulée à chaque passage. L'utilisation de la méthode **.pop()** retirant la valeur de l'array, à chaque passage dans la fonction récursive, l'array perd un élément.

La récursivité est déclarée en utilisant en retour de la fonction un appel de la fonction elle-même avec en paramètre l'array contenant un élément de moins.

Lorsque l'array ne contient plus qu'une valeur, on atteint la condition de sortie. La fonction retourne alors la toute dernière valeur pour sortir de la récursivité.

Voici ce qui se passe concrètement dans cette fonction à chaque passage :

```
// 1er passage
return 10 + arraySum([1, 2, 3, 4, 5, 6, 7, 8, 9]);
```



```
// 2ème passage
return 10 + 9 + arraySum([1, 2, 3, 4, 5, 6, 7, 8]);

// 3ème passage
return 10 + 9 + 8 + arraySum([1, 2, 3, 4, 5, 6, 7]);

// 4ème passage
return 10 + 9 + 8 + 7 + arraySum([1, 2, 3, 4, 5, 6]);

// 5ème passage
return 10 + 9 + 8 + 7 + 6 + arraySum([1, 2, 3, 4, 5]);

// 6ème passage
return 10 + 9 + 8 + 7 + 6 + 5 + arraySum([1, 2, 3, 4]);

// 7ème passage
return 10 + 9 + 8 + 7 + 6 + 5 + 4 + arraySum([1, 2, 3]);

// 8ème passage
return 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + arraySum([1, 2]);

// 9ème passage
return 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + arraySum([1]);

// Sortie de la récursivité
return 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1;
```

Méthode d'un objet

Comme une fonction est au final une valeur, il est alors possible de déclarer un objet qui a pour une clé donnée une fonction.

Voici un exemple :

```
const character = {
  name: "My lord",
  talk: function() {
    console.log("Call me My Lord!");
  }
};
character.talk();
```

La variable **character** contient ici un objet avec 2 clés :

- **name** : qui a pour valeur une chaîne de caractère ou string ;
- **talk** : qui a pour valeur une fonction anonyme, c'est une méthode.

Lorsqu'une fonction est utilisée de cette façon dans le contexte d'un objet nous l'appelons une méthode.

L'appel de cette méthode se fait en appelant l'objet, suivi d'un point puis de la clé avec les parenthèses **()**.

Nous avons déjà utilisé cette notation pour utiliser des méthodes lorsque nous avons manipulé des array ou des string (ex: **array.pop()**, **string.toLowerCase()**, ...).

Le this

Dans notre exemple précédent il serait intéressant de réutiliser la clé **name** de l'objet dans la méthode **talk** afin que le personnage construise lui-même la phrase qu'il dit.

C'est à cela que va nous servir **this**. Le mot clé **this** fait référence à l'objet dans lequel la méthode a été appelée.

Dans la méthode **talk**, ceci retourne la même valeur

```
this.name
```

que ceci en dehors de l'object **character**.

```
character.name
```

Nous pouvons alors modifier notre code pour obtenir cela :

```
const character = {
  name: "My lord",
  talk: function() {
    console.log(`Call me ${this.name}!`);
  }
};
character.talk();
// Call me My Lord!
```

Il est alors possible de modifier **name** est la méthode **talk** suivra.

```
character.name = "The Clown";
character.talk();
// Call me The Clown!
```

Contexte de **this**

Nous allons désormais voir les limitations de **this** dans des contextes d'utilisation plus avancés comme dans l'exemple ci-dessous.

```
const character = {
  name: "My lord",
  talk: function() {
    console.log(`Call me ${this.name}!`);
    const memory = function() {
      this.remember(`Call me ${this.name}!`);
    }
    memory();
  },
  remember: function(text) {
    console.log(`Remember: ${text}`);
  }
};
```

```
character.talk();
// Call me My Lord!
// Type Error
```

Ce code va déclarer une erreur, car dans la fonction **memory()**, le **this** n'est plus notre objet **character**, mais un certain **window** (qui est la fenêtre du navigateur, nous la reverrons dans un autre module.) Ceci arrive car le contexte de la fonction **memory()** n'est plus un objet mais lui-même une fonction.

La solution historique à ce problème, est de déclarer une variable **self** dans laquelle on passe la valeur de **this**, de manière à pouvoir récupérer l'objet voulu dans la fonction suivante. Ce qui donne ceci :

```
const character = {
  name: "My lord",
  talk: function() {
    console.log(`Call me ${this.name}!`);
    const self = this;
    const memory = function() {
      self.remember(`Call me ${self.name}!`);
    }
    memory();
  },
  remember: function(text) {
    console.log(`Remember: ${text}`);
  }
};

character.talk();
// Call me My Lord!
// Remember: Call me My lord!
```

Ce code avec la variable **self** fonctionne bien. Mais ceci étant un problème récurrent, une nouvelle façon de déclarer les méthodes est apparue dans les versions modernes de javascript (ECMAScript 2015) : les fonctions fléchées.

Fonctions fléchées

La fonction fléchée (arrow function) permet de garder la valeur du **this** du contexte de déclaration de la fonction sans déclarer sa propre valeur de **this**. Dans notre exemple cela donne ceci :

```
const character = {
  name: "My lord",
  talk: function() {
    const sentence = `Call me ${this.name}!`;
    console.log(sentence);
    const memory = () => {
      this.remember(sentence);
    }
    memory();
  },
  remember: function(text) {
    console.log(`Remember: ${text}`);
  }
};

character.talk();
// Call me My Lord!
// Remember: Call me My lord!
```

Cet exemple fonctionne donc comme on le souhaite.


Au-delà de l'utilisation pour avoir la bonne valeur de **this**, la fonction fléchée est une manière plus courte de déclarer une [fonction anonyme](#).

Elle n'utilise plus le mot clé **function**, mais utilise une flèche => (égal, puis supérieur) entre les parenthèses et les accolades.

Voici un exemple type d'une arrow function :

```
const arrowFunction = (a, b) => {
  return a * b;
};

console.log(arrowFunction(2, 4));
```



Si la fonction ne prend qu'un seul paramètre, les parenthèses peuvent être omises, comme ceci :

```
const arrowFunction = a => {  
    return a * 3;  
};  
console.log(arrowFunction(2));
```

Si comme dans notre exemple, la seule instruction de la fonction est de retourner une valeur, la déclaration peut se faire sur une seule ligne, sans accolade et sans **return**, qui devient alors implicite.

```
const arrowFunction = a => a * 3;  
console.log(arrowFunction(2));
```