> Cette page a été traduite à partir de l'anglais par la communauté. Vous pouvez également contribuer en rejoignant la communauté francophone sur MDN Web Docs.

Fonctions

Les fonctions font partie des briques fondamentales de JavaScript. Une fonction est une procédure JavaScript, un ensemble d'instructions effectuant une tâche ou calculant une valeur. Afin d'utiliser une fonction, il est nécessaire de l'avoir auparavant définie au sein de la portée dans laquelle on souhaite l'appeler.

On pourra également lire le chapitre de la référence JavaScript sur les fonctions pour étudier plus en détails ce concept

Définir des fonctions

Les déclarations de fonctions

Une **définition de fonction** (aussi appelée **déclaration de fonction** ou **instruction de fonction**) est construite avec le mot-clé `function` , suivi par :

- Le nom de la fonction.
- Une liste d'arguments à passer à la fonction, entre parenthèses et séparés par des virgules.
- Les instructions JavaScript définissant la fonction, entre accolades, `{ }` .

Le code suivant, par exemple, définit une fonction intitulée `carré` :

JS

```
function carré(nombre) {
  return nombre * nombre;
}
```

La fonction `carré` prend un seul argument, appelé `nombre`. La fonction est composée d'une seule instruction qui renvoie l'argument de la fonction (`nombre`) multiplié par lui-même. L'instruction `return` spécifie la valeur qui est renvoyée par la fonction.

JS

```
return nombre * nombre;
```

Les paramètres primitifs (comme les nombres) sont passés à la fonction **par valeur**. La valeur est passée à la fonction mais si cette dernière change la valeur du paramètre, cela n'aura pas d'impact au niveau global ou au niveau de ce qui a appelé la fonction.

Si l'argument passé à la fonction est un objet (une valeur non-primitive, comme un objet `Array` ou un objet défini par l'utilisateur), et que la fonction change les propriétés de cet objet, ces changements seront visibles en dehors de la fonction. Par exemple :

JS

```
function maFonction(monObjet) {
  monObjet.fabricant = "Toyota";
}

var mavoiture = { fabricant: "Honda", modèle: "Accord", année: 1998 };
var x, y;

x = mavoiture.fabricant; // x aura la valeur "Honda"

maFonction(mavoiture);
y = mavoiture.fabricant; // y aura la valeur "Toyota"
// (la propriété fabricant a été modifiée par la fonction)
```

> **Note :** Affecter un nouvel objet au paramètre n'aura **pas** d'effet en dehors de la fonction car cela revient à changer la valeur du paramètre plutôt que la valeur d'une des propriétés de l'objet. Par exemple :

JS

```
function maFonction(monObjet) {
  monObjet = { fabricant: "Ford", modèle: "Focus", année: 2006 };
}

var mavoiture = { fabricant: "Honda", modèle: "Accord", année: 1998 };
var x, y;

x = mavoiture.fabricant; // x reçoit la valeur "Honda"

maFonction(mavoiture);
y = mavoiture.fabricant; // y reçoit la valeur "Honda"
```

Dans le premier exemple, l'objet `mavoiture` était passé à la fonction `maFonction` qui le modifiait. Dans le second exemple, la fonction n'a pas modifié l'objet qui avait été passé en argument, elle a créé une nouvelle variable locale, possédant le même nom que l'objet global passé en argument : il n'y a donc pas de modifications sur cet objet global.

Les expressions de fonction

Syntaxiquement, la déclaration de fonction utilisée ci-dessus est une instruction. On peut également créer une fonction grâce à une **expression de fonction**. De telles fonctions peuvent être **anonymes** (ne pas avoir de nom correspondant). La fonction `carré` aurait pu être définie de la façon suivante :

JS

```
var carré = function (nombre) {
  return nombre * nombre;
};
var x = carré(4); //x reçoit la valeur 16
```

Cependant, un nom peut être utilisé dans une expression de fonction, ce afin de l'utiliser dans la fonction (récursivité) ou afin de l'identifier dans les appels tracés par un éventuel débogueur :

JS

```
var factorielle = function fac(n) {
  return n < 2 ? 1 : n * fac(n - 1);
};
```

```
console.log(factorielle(3));
```

Les expressions de fonction sont pratiques lorsqu'il s'agit de passer une fonction comme argument d'une autre fonction. Dans l'exemple qui suit, la fonction `map` est définie et appelée avec une fonction anonyme comme premier argument :

JS

```
function map(f, a) {
  var resultat = []; // Créer un nouveau tableau Array
  for (var i = 0; i != a.length; i++) resultat[i] = f(a[i]);
  return resultat;
}
```

Le code suivant applique la fonction `cube` sur chacun des éléments du tableau :

JS

```
var cube = function (x) {
  return x * x * x;
}; // Une expression de fonction
map(cube, [0, 1, 2, 5, 10]);
```

Le résultat de la dernière instruction est le tableau [0, 1, 8, 125, 1000].

En JavaScript, une fonction peut être définie selon une condition. Le fragment de code qui suit définit une fonction seulement si `num` vaut 0 :

JS

```
var maFonction;
if (num === 0) {
  maFonction = function (monObjet) {
    monObjet.fabricant = "Toyota";
  };
}
```

Une autre façon de définir des fonctions est d'utiliser le constructeur de l'objet `Function` afin de créer des fonctions à partir d'une chaîne lors de l'exécution, de la même façon que `eval()` .

Une **méthode** est une fonction étant une propriété d'un objet. Vous trouverez plus de détails sur ces éléments dans le chapitre suivant du guide : Utiliser les objets.

Appeler des fonctions

La seule définition d'une fonction ne permet pas d'exécuter la fonction. Cela permet de lui donner un nom et de définir ce qui doit être fait lorsque la fonction est appelée. **Appeler** la fonction permet d'effectuer les actions des instructions avec les paramètres indiqués. Par exemple, si on définit la fonction `carré`, on peut l'appeler de la façon suivante :

JS

```
carré(5);
```

Cette instruction appellera la fonction avec un argument valant 5. La fonction exécute ses instructions et renvoie la valeur 25.

Les fonctions doivent appartenir à la portée dans laquelle elles sont appelées. En revanche, la déclaration d'une fonction peut être faite après l'appel :

JS

```
console.log(carré(5));
/* ... */
function carré(n) {
  return n * n;
}
```

La portée d'une fonction est la fonction dans laquelle elle est déclarée ou le programme entier si elle est déclarée au niveau le plus haut.

> **Note :** Cela ne fonctionne que si la définition de la fonction utilise la syntaxe précédente (`function nomFonction(){}`). Le code ci-dessous ne fonctionnera pas :

JS

```
console.log(carré); // La fonction carré est remontée/hoisted mais vaut
undefined
```

```
  console.log(carré(5)); // TypeError: carré is not a function
  var carré = function (n) {
    return n * n;
  };

  // Et avec let...

  console.log(carré2); // ReferenceError: carré2 is not defined
  console.log(carré2(5)); // TypeError: carré2 is not a function

  let carré2 = function (n) {
    return n * n;
  };
```

Les arguments d'une fonction ne sont pas limités aux chaînes de caractères et aux nombres. Il est possible de passer des objets. La fonction `show_props` (définie dans le chapitre sur l'utilisation des objets) est un exemple de fonction utilisant un argument qui est un objet.

Une fonction peut être récursive, c'est-à-dire qu'elle peut s'appeler elle-même. Voici la fonction qui calcule récursivement la factorielle d'un nombre :

JS

```
function factorielle(n) {
  if (n === 0 || n === 1) return 1;
  else return n * factorielle(n - 1);
}
```

On peut ensuite calculer les factorielles des nombres 1 à 5 :

JS

```
var a, b, c, d, e;
a = factorielle(1); // a reçoit la valeur 1
b = factorielle(2); // b reçoit la valeur 2
c = factorielle(3); // c reçoit la valeur 6
d = factorielle(4); // d reçoit la valeur 24
e = factorielle(5); // e reçoit la valeur 120
```

Il existe d'autres façons d'appeler des fonctions. Il existe souvent des cas où une fonction doit être appelée dynamiquement, où le nombre d'arguments peut varier, où le contexte de l'appel d'une fonction doit être créé en fonction d'un objet déterminé lors de l'exécution. Les fonctions sont des objets, en tant que tels, elles possèdent des méthodes (voir la page sur l'objet `Function`). L'une d'entre elles, `apply()` peut être utilisée pour réaliser le dernier cas de figure (exécution d'une fonction avec un objet déterminé à l'exécution).

Portée d'une fonction

On ne peut pas accéder aux variables définies dans une fonction en dehors de cette fonction : ces variables n'existent que dans la portée de la fonction. En revanche, une fonction peut accéder aux différentes variables et fonctions qui appartiennent à la portée dans laquelle elle est définie. Une fonction définie dans une autre fonction peut également accéder à toutes les variables de la fonction « parente » et à toute autre variable accessible depuis la fonction « parente ».

JS

```
// Les variables suivantes sont globales
var num1 = 20,
  num2 = 3,
  nom = "Licorne";

// Cette fonction est définie dans la portée globale
function multiplier() {
  return num1 * num2;
}

multiplier(); // Renvoie 60

// Un exemple de fonction imbriquée
function getScore() {
  var num1 = 2,
    num2 = 3;

  function ajoute() {
    return nom + " a marqué " + (num1 + num2);
  }

  return ajoute();
}
```

```
getScore(); // Renvoie "Licorne a marqué 5"
```

Portée et pile de fonctions

La récursivité

Une fonction peut faire référence à elle-même et s'appeler elle-même. Il existe trois moyens pour qu'une fonction fasse référence à elle-même :

1. Le nom de la fonction
2. `arguments.callee`
3. Une variable de la portée qui fait référence à la fonction

Par exemple, avec la définition de fonction suivante :

JS

```
var toto = function truc() {
  // les instructions de la fonction
};
```

Dans le corps de la fonction, ces trois éléments seront équivalents :

1. `truc()`
2. `arguments.callee()`
3. `toto()`

Une fonction qui s'appelle elle-même est appelée une fonction *récursive*. Sous certains aspects, une récursion est semblable à une boucle : toutes les deux exécutent le même code plusieurs fois et toutes les deux requièrent une condition d'arrêt (pour éviter une boucle ou une récursion infinie). Par exemple, ce fragment de code utilisant une boucle :

JS

```
var x = 0;
while (x < 10) {
  // "x < 10" représente la condition d'arrêt
```

```
  // faire quelque chose
  x++;
}
```

pourra être converti en une fonction récursive de la façon suivante :

JS

```
function boucle(x) {
  if (x >= 10) {
    // "x >= 10" représente la condition d'arrêt (équivalent à "!(x < 10)")
    return;
  }
  // faire quelque chose
  boucle(x + 1); // l'appel récursif
}
boucle(0);
```

Malgré cela, certains algorithmes ne peuvent pas être convertis en boucles itératives. Ainsi, récupérer l'ensemble des nœuds d'un arbre (le DOM par exemple) se fait plus simplement en utilisant la récursivité :

JS

```
function parcourirArbre(noeud) {
  if (noeud === null) {
    return;
  }
  // faire quelque chose avec le noeud
  for (var i = 0; i < noeud.childNodes.length; i++) {
    parcourirArbre(noeud.childNodes[i]);
  }
}
```

Contrairement à l'exemple précédent avec la fonction `boucle`, ici, chaque appel récursif entraîne lui-même plusieurs appels (et non un seul).

Théoriquement, il est possible de convertir tout algorithme récursif en un algorithme non récursif (avec des boucles par exemple). Généralement, la logique obtenue est plus complexe et nécessite l'utilisation d'une pile . La récursivité utilise également une pile, la pile de fonction.

Ce type de « comportement » peut-être observé avec l'exemple suivant :

JS

```
function toto(i) {
  if (i < 0) return;
  console.log("début : " + i);
  toto(i - 1);
  console.log("fin : " + i);
}
toto(3);
```

qui affichera :

```
début : 3
début : 2
début : 1
début : 0
fin : 0
fin : 1
fin : 2
fin : 3
```

Fonctions imbriquées et fermetures

Il est possible d'imbriquer une fonction dans une autre fonction. La portée de la fonction fille (celle qui est imbriquée) n'est pas contenue dans la portée de la fonction parente. En revanche, la fonction fille bénéficie bien des informations de la fonction parente grâce à sa portée. On a ce qu'on appelle une fermeture (*closure* en anglais). Une fermeture est une expression (généralement une fonction) qui accède à des variables libres ainsi qu'à un environnement qui lie ces variables (ce qui « ferme » l'expression).

Une fonction imbriquée étant une fermeture, cela signifie qu'une fonction imbriquée peut en quelque sorte hériter des arguments et des variables de la fonction parente.

En résumé :

- La fonction imbriquée ne peut être utilisée qu'à partir des instructions de la fonction parente.

- La fonction imbriquée forme une fermeture : elle peut utiliser les arguments et les variables de la fonction parente. En revanche, la fonction parente ne peut pas utiliser les arguments et les variables de la fonction fille.

> **Note :** Sur les fermetures, voir également l'article à ce sujet.

L'exemple qui suit illustre l'imbrication de fonctions :

JS

```
function ajouteCarrés(a, b) {
  function carré(x) {
    return x * x;
  }
  return carré(a) + carré(b);
}
a = ajouteCarrés(2, 3); // renvoie 13
b = ajouteCarrés(3, 4); // renvoie 25
c = ajouteCarrés(4, 5); // renvoie 41
```

La fonction interne étant une fermeture, on peut appeler la fonction parente afin de définir les arguments pour la fonction englobante et ceux de la fonction fille :

JS

```
function parente(x) {
  function fille(y) {
    return x + y;
  }
  return fille;
}
fn_fille = parente(3); // Fournit une fonction qui ajoute 3 à ce qu'on lui donnera
résultat = fn_fille(5); // renvoie 8

résultat1 = parente(3)(5); // renvoie 8
```

Préservation des variables

Dans l'exemple précédent, `x` a été « préservé » lorsque la fonction `fille` a été renvoyée. Une fermeture conserve les arguments et les variables de chaque portée qu'elle référence. Chaque appel à la fonction parente pouvant fournir un contexte différents selon les

arguments, cela entraînera la création d'une nouvelle fermeture. La mémoire associée ne pourra être libérée que lorsque la fonction `fille` ne sera plus accessible.

Ce mode de fonctionnement n'est pas différent de celui des références vers les objets. Cependant, il est souvent plus compliqué à détecter car les références ne sont pas définies explicitement dans le code et car il n'est pas possible de les inspecter.

Imbriquer plusieurs fonctions

Il est possible d'imbriquer des fonctions sur plus de deux niveaux, par exemple, on peut avoir une fonction A qui contient une fonction B qui contient une fonction C. Les fonctions B et C sont des fermetures et B peut accéder à la portée de A, C peut accéder à la portée de B. Ainsi, C accède à la portée de B qui lui accède à la portée de A, C accède donc à la portée de A (transitivité). Les fermetures peuvent donc contenir plusieurs portées, c'est ce qu'on appelle le *chaînage* de portées.

Par exemple :

JS

```
function A(x) {
  function B(y) {
    function C(z) {
      console.log(x + y + z);
    }
    C(3);
  }
  B(2);
}
A(1); // affichera 6 (1 + 2 + 3)
```

Dans cet exemple `C` accède au `y` de `B` et au `x` de `A`. Ceci est rendu possible car :

1. `B` est une fermeture qui contient `A`, autrement dit `B` peut accéder aux arguments et aux variables de `A`.
2. `C` est une fermeture qui contient `B`.
3. La fermeture de `B` contient `A` donc la fermeture de `C` contient `A`, `C` peut ainsi accéder aux arguments et aux variables de `B` *et* `A`. On dit que `C` *chaîne* les portées

de `B` et de `A` (dans cet ordre).

La réciproque n'est pas vraie. `A` ne peut pas accéder à `C`, car `A` ne peut pas accéder aux arguments ou aux variables de `B`, or `C` est une variable de `B`. De cette façon, `C` reste privée en dehors de `B`.

Conflits de nommage

Lorsque deux arguments ou variables des portées d'une fermeture ont le même nom, il y a un conflit de noms. Dans ces cas, ce sera la portée la plus imbriquée qui prendra la priorité sur le nom, la portée la plus « externe » aura la priorité la plus faible pour les noms de variables. Du point de vue de la chaîne des portées, la première portée sur la chaîne est la portée la plus imbriquée et la dernière est la portée située le plus à l'extérieur :

JS

```
function externe() {
  var x = 10;
  function interne(x) {
    return x;
  }
  return interne;
}
résultat = externe()(20); // renvoie 20 et pas 10
```

Le conflit se produit à l'instruction `return x` entre le paramètre x de la fonction `interne` et la variable `x` de la fonction `externe`. La chaîne de portée est ici {`interne`, `externe`, objet global}. Ainsi, le paramètre `x` de `interne` a la priorité sur la variable `x` de la fonction `externe`, le résultat obtenu est donc 20 et non 10.

Fermetures (*closures*)

Les fermetures sont l'une des fonctionnalités les plus intéressantes de JavaScript. Comme on l'a vu précédemment, JavaScript permet d'imbriquer des fonctions et la fonction interne aura accès aux variables et paramètres de la fonction parente. À l'inverse, la fonction parente ne pourra pas accéder aux variables liées à la fonction interne. Cela fournit une certaine sécurité pour les variables de la fonction interne. De plus, si la fonction interne peut exister plus longtemps que la fonction parente, les variables et fonctions de la

fonction parente pourront exister au travers de la fonction interne. On crée une fermeture lorsque la fonction interne est disponible en dehors de la fonction parente.

JS

```
var animal = function (nom) {
  // La fonction externe utilise un paramètre "nom"
  var getNom = function () {
    return nom; // La fonction interne accède à la variable "nom" de la fonction externe
  };
  return getNom; // Renvoie la fonction interne pour la rendre disponible en dehors de la portée de la fonction parente
};

monAnimal = animal("Licorne");

monAnimal(); // Renvoie "Licorne"
```

Bien entendu, dans la pratique, les cas peuvent être plus complexes. On peut renvoyer un objet qui contient des méthodes manipulant les variables internes de la fonction parente.

JS

```
var créerAnimal = function (nom) {
  var sexe;

  return {
    setNom: function (nouveauNom) {
      nom = nouveauNom;
    },

    getNom: function () {
      return nom;
    },

    getSexe: function () {
      return sexe;
    },

    setSexe: function (nouveauSexe) {
      if (
        typeof nouveauSexe == "string" &&
        (nouveauSexe.toLowerCase() == "mâle" ||
```

```
          nouveauSexe.toLowerCase() == "femelle")
      ) {
        sexe = nouveauSexe;
      }
    },
  };
};

var animal = créerAnimal("Licorne");
animal.getNom(); // Licorne

animal.setNom("Bobby");
animal.setSexe("mâle");
animal.getSexe(); // mâle
animal.getNom(); // Bobby
```

Dans le code précédent, la variable `nom` est de la fonction externe est accessible depuis les fonctions internes. Il est impossible d'accéder aux variables internes en dehors des fonctions internes. Les variables internes agissent comme des coffres-forts pour les fonctions internes. Elles permettent d'avoir un accès persistent et encapsulé aux données internes. Pour les fonctions, il n'est pas nécessaire de les affecter à une variable ou même de les nommer.

JS

```
var getCode = (function () {
  var codeAPI = "0]Eal(eh&2"; // Un code qu'on ne souhaite pas diffuser ni modifier

  return function () {
    return codeAPI;
  };
})();

getCode(); // Renvoie la valeur du code
```

Il y a malgré tout quelques pièges auxquels il faut faire attention lorsqu'on utilise les fermetures. Si une fonction imbriquée définit une variable avec le même nom que le nom d'une variable de la portée externe, il n'y aura plus aucun moyen d'accéder à la variable.

JS

```
var créerAnimal = function (nom) {
  // La fonction externe définit une variable appelée "nom"
  return {
    setNom: function (nom) {
      // La fonction imbriquée définit une variable appelée "nom"
      nom = nom; // ??? comment accéder à la variable "nom" définie par la fonction
externe
    },
  };
};
```

L'opérateur `this` doit être traité avec précaution dans les fermetures. Attention, `this` fait référence au contexte où la fonction est appelée et non à l'endroit où il est défini.

Utiliser l'objet `arguments`

Les arguments d'une fonction sont maintenus dans un objet semblable à un tableau. Dans une fonction, il est possible d'utiliser les arguments passés à la fonction de la façon suivante :

JS

```
arguments[i];
```

où `i` représente l'index ordinal de l'argument (le premier argument ayant un indice à 0). On accède donc au premier argument avec `arguments[0]`. Le nombre total d'arguments est fourni grâce à `arguments.length`.

En utilisant l'objet `arguments`, il est possible de recenser les arguments supplémentaires fournis à la fonction si jamais il y a plus d'arguments fournis que requis. Cet objet est souvent utile si on ne connaît pas le nombre d'arguments passés à la fonction. La propriété `arguments.length` permet de déterminer le nombre d'arguments réellement passés à la fonction. On peut donc ensuite accéder aux différents arguments en parcourant l'objet `arguments`.

Par exemple, on peut construire une fonction qui concatène plusieurs chaînes. Le seul argument formellement défini sera la chaîne utilisée pour concaténer les différentes chaînes. On peut définir la fonction de la façon suivante :

JS

```
function monConcat(séparateur) {
  var result = ""; // on initialise la liste
  var i;
  // on parcourt les arguments
  for (i = 1; i < arguments.length; i++) {
    result += arguments[i] + séparateur;
  }
  return result;
}
```

On peut passer autant d'arguments que nécessaire à cette fonction. Ils seront tous concaténés dans une chaîne finale. Ainsi, on aura :

JS

```
// renverra "rouge, orange, bleu, "
monConcat(", ", "red", "orange", "blue");

// renverra "éléphant; girafe; lion; singe; "
monConcat("; ", "éléphant", "girafe", "lion", "singe");

// renverra "sauge. basilic. origan. poivre. échalotte. "
monConcat(". ", "sauge", "basilic", "origan", "poivre", "échalotte");
```

> **Note :** `arguments` est une variable « semblable » à un tableau. Mais ce n'est pas un tableau au sens strict. En effet, il possède un index numéroté ainsi qu'une propriété `length`. En revanche, il ne possède pas les méthodes classiques de manipulation des tableaux (Array).

Voir la page sur l'objet `Function` dans la référence JavaScript pour plus d'informations.

Paramètres des fonctions

À partir d'ECMAScript 2015, deux sortes de paramètres sont introduites : les paramètres par défaut et les paramètres du reste.

Les paramètres par défaut

En JavaScript, par défaut, les paramètres des fonctions vaudront `undefined`. Il peut toutefois être utile de définir une valeur par défaut différente. Les paramètres par défaut permettent de répondre à ce besoin.

Avant ECMAScript 2015, la stratégie pour manipuler des valeurs par défaut adaptées était de tester si la valeur du paramètre était indéfinie puis de lui affecter la valeur souhaitée si c'était le cas. Par exemple, dans le code qui suit, on ne fournit pas de valeur pour `b` dans l'appel, la valeur sera `undefined` lors de l'évaluation de `a*b` et l'appel à `multiplier` aurait renvoyé `NaN`. Pour éviter ça, la deuxième ligne définit une valeur par défaut au cas où `b` n'est pas renseigné :

JS

```
function multiplier(a, b) {
  b = typeof b !== "undefined" ? b : 1;

  return a * b;
}

multiplier(5); // 5
```

Si on peut utiliser les paramètres par défaut, il n'est plus nécessaire de faire l'opération à l'intérieur du corps de la fonction, il suffit de déclarer que la valeur par défaut pour `b` est 1 dans la signature de la fonction :

JS

```
function multiplier(a, b = 1) {
  return a * b;
}

multiplier(5); // 5
```

Pour plus de détails, voir la page sur les paramètres par défaut dans la référence.

Les paramètres du reste

La syntaxe des paramètres du reste permet de représenter un nombre indéfini d'arguments contenus dans un tableau. Dans l'exemple suivant, on utilise les paramètres du reste pour collecter les arguments à partir du deuxième et jusqu'au dernier. Ces

arguments sont multipliés par le premier. Dans cet exemple, on utilise une fonction fléchée, concept qui est présenté et illustré dans la section qui suit.

JS

```
function multiplier(facteur, ...lesArgs) {
  return lesArgs.map((x) => facteur * x);
}

var arr = multiplier(2, 1, 2, 3);
console.log(arr); // [2, 4, 6]
```

Fonctions fléchées

Une expression de fonction fléchée permet d'utiliser une syntaxe plus concise que les expressions de fonctions classiques. Une telle fonction ne possède alors pas de valeur propre pour `this`, `arguments`, `super` ou `new.target`. Les fonctions fléchées sont nécessairement anonymes.

Les fonctions fléchées ont été introduites pour deux raisons principales : une syntaxe plus courte et l'absence de `this` rattaché à la fonction. Voir aussi ce billet sur tech.mozfr.org sur les fonctions fléchées .

Concision de la syntaxe

Dans certaines constructions fonctionnelles, on peut apprécier une syntaxe courte. Par exemple, si on compare les deux dernières lignes de ce fragment de code :

JS

```
var a = ["Hydrogen", "Helium", "Lithium", "Beryllium"];

var a2 = a.map(function (s) {
  return s.length;
});
console.log(a2); // affiche [8, 6, 7, 9]
var a3 = a.map((s) => s.length);
console.log(a3); // affiche [8, 6, 7, 9]
```

Pas de `this` distinct

Avant les fonctions fléchées, chaque nouvelle fonction définissait sa propre valeur `this` (un nouvel objet dans le cas d'un constructeur, `undefined` lors des appels en mode strict, l'objet courant dans le cas d'une méthode, etc.). Cela pouvait poser quelques problèmes avec un style de programmation orienté objet.

JS

```
function Personne() {
  // Le constructeur Personne() utilise `this` comme lui-même.
  this.âge = 0;

  setInterval(function grandir() {
    // En mode non-strict, la fonction grandir() définit `this`
    // avec l'objet global, qui est donc différent du `this`
    // défini par le constructeur Person().
    this.âge++;
  }, 1000);
}

var p = new Personne();
```

Avec ECMAScript 3/5, ce problème fut résolu avec l'affectation de la valeur de `this` dans une variable a variable that could be closed over.

JS

```
function Personne() {
  var self = this; // Certains utilisent `that`, d'autres `self`.
  // On utilisera l'un des deux et on pas
  // l'autre pour être cohérent.
  self.âge = 0;

  setInterval(function grandir() {
    // La fonction callback fait référence à la variable `self`
    // qui est bien la valeur attendue liée à l'objet.
    self.âge++;
  }, 1000);
}
```

On aurait aussi pu créer une fonction liée afin que la « bonne » valeur de `this` soit passée à la fonction `grandir()`.

Les fonctions fléchées capturent la valeur de `this` dans le contexte englobant et cela permet de manipuler la valeur pertinente ici :

JS

```
function Personne() {
  this.âge = 0;

  setInterval(() => {
    this.âge++; // this fait référence à l'objet personne
  }, 1000);
}

var p = new Personne();
```

Fonctions prédéfinies

JavaScript possède plusieurs fonctions natives, disponibles au plus haut niveau :

`eval()`

La fonction **`eval()`** permet d'évaluer du code JavaScript contenu dans une chaîne de caractères.

`uneval()`

La fonction **`uneval()`** crée une représentation sous la forme d'une chaîne de caractères pour le code source d'un objet.

`isFinite()`

La fonction **`isFinite()`** détermine si la valeur passée est un nombre fini. Si nécessaire, le paramètre sera converti en un nombre.

`isNaN()`

La fonction **`isNaN()`** détermine si une valeur est `NaN` ou non. Note : On pourra également utiliser `Number.isNaN()` défini avec ECMAScript 6 ou utiliser `typeof` afin de déterminer si la valeur est **N**ot-**A**-**N**umber.

`parseFloat()`

La fonction **`parseFloat()`** convertit une chaîne de caractères en un nombre flottant.

`parseInt()`

La fonction **`parseInt()`** convertit une chaîne de caractères et renvoie un entier dans la base donnée.

`decodeURI()`

La fonction **`decodeURI()`** décode un Uniform Resource Identifier (URI) créé par `encodeURI()` ou une méthode similaire.

`decodeURIComponent()`

La fonction **`decodeURIComponent()`** décode un composant d'un Uniform Resource Identifier (URI) créé par `encodeURIComponent` ou une méthode similaire.

`encodeURI()`

La fonction **`encodeURI()`** encode un Uniform Resource Identifier (URI) en remplaçant chaque exemplaire de certains caractères par un, deux, trois voire quatre séquences d'échappement représentant l'encodage UTF-8 du caractères (quatre séquences seront utilisées uniquement lorsque le caractère est composé d'une paire de deux demi-codets).

`encodeURIComponent()`

La fonction **`encodeURIComponent()`** encode un composant d'un Uniform Resource

composé d'une paire de deux demi-codets).

`escape()`

La fonction dépréciée **`escape()`** calcule une nouvelle chaîne de caractères pour laquelle certains caractères ont été remplacés par leur séquence d'échappement hexadécimale. Les fonctions `encodeURI()` ou `encodeURIComponent()` doivent être utilisées à la place.

`unescape()`

La fonction dépréciée **`unescape()`** calcule une nouvelle chaîne de caractères pour laquelle les séquences d'échappement hexadécimales sont remplacées par les caractères qu'elles représentent. Les séquences d'échappement introduites peuvent provenir d'une fonction telle que `escape()`. `unescape` est dépréciée et doit être remplacée par `decodeURI()` ou `decodeURIComponent()`.