JS : Les fonctions

L2 MPCIE - UE Développement Web

David Lesaint

david.lesaint@univ-angers.fr

Janvier 2019

Fonctions

Les fonctions

- Objets de type `function`.
- Déclarées avec le mot-clé `function`.
- Nommées ou anonymes (expression de fonction).
- Invoquées explicitement, en fonction d'évènements utilisateur, ou auto-exécutées.
- Les arguments peuvent ne pas être déclarés.
- Type des arguments et valeur retour non déclarés.
- Peuvent déclarer et retourner d'autres fonctions.

- Les objets sont passés par référence, les valeurs primitives par valeur.

Déclarations de fonctions

fonction.html

```
<!DOCTYPE html>
<html>
<head><meta charset="UTF-8"/></head>
<body>
<p id="demo"></p>
<script>
function toCelsius(f) { return (5/9) * (f-32); }
document.getElementById("demo").innerHTML =
  "This JS function<blockquote>"
  + String(toCelsius)
  + "</blockquote>yields "
  + String(toCelsius(50))
  +"&#8451 for 64&#8457";
</script>
</body>
</html>
```

Expressions de fonctions

Expressions de fonctions anonymes ou nommées

Nommage nécessaire pour fonctions récursives et le débogage.

fonction-expression.js

```
1  var carré = function(nombre) {
2    return nombre * nombre;
3  };
4  console.log(carré(4));
5
6  var factorielle = function fac(n) {
7    return n < 2 ? 1 : n * fac(n - 1)
8  };
9  //ON NE PEUT PAS DIRECTEMENT INVOQUER fac
10 //console.log(fac(3));
11 console.log(factorielle(3));
```

Expressions de fonctions

Utiles comme fonctions de rappel

Passées comme paramètre à d'autre fonctions.

fonction-expression-1.js

```
 1 function map(f, a) {
 2   var resultat = [], i;
 3   for (i = 0; i != a.length; i++)
 4     resultat[i] = f(a[i]);
 5   return resultat;
 6 }
 7
 8 var cube = function(x) {
 9   return x * x * x
10 }; // Une expression de fonction
11 map(cube, [ 0, 1, 2, 5, 10 ]);
```

Positionnement des déclarations

- L'appel d'une fonction classique peut précéder sa déclaration (dans le programme ou la fonction où elle est déclarée).
- L'appel d'une fonction exprimée ne peut pas précéder sa déclaration.

fonction-declaration.js

```
1 console.log(cube(2));
2 function cube(x) {
3   return x * x * x;
4 };
5
6 console.log(carré(2));// TypeError
7 var carré = function(x) {
8   return x * x;
9 };
```

Portée de fonction

- Accès aux variables définies dans une fonction limité à la portée de la fonction.
- Accès possible aux variables/fonctions définies dans la portée dans laquelle la fonction est définie.

fonction-portee.js

```
1  var x = 20, y = 3, nom = "Licorne";
2  function multiplier() {
3    return x * y;
4  }
5  console.log(multiplier()); // 60
6
7  function getScore() {
8    var x = 2, y = 3;
9    function ajoute() {
10     return nom + " a marqué " + (x + y);
11   }
12   return ajoute();
13 }
14 console.log(getScore()); // Licorne a marqué 5
```

Fonctions récursives

Une fonction peut s'appeler en utilisant

- Son nom.
- `arguments.callee`
- Une variable de la portée qui fait référence à cette fonction.

fonction-recursive.html

```
1  <!DOCTYPE html>
2  <html>
3  <head><meta charset="UTF-8"/></head>
4  <body>
5  <script>
6  function parcourirArbre(noeud) {
7    if (noeud === null) //
8      return;
9    for (var i = 0; i < noeud.childNodes.length; i++) {
10     console.log(noeud.childNodes[i].toString());
11     parcourirArbre(noeud.childNodes[i]);
12   }
13 }
14 parcourirArbre(document.body);
15 </script>
16 </body>
17 </html>
```

Fonctions imbriquées

Fonction imbriquée

- Ne peut être utilisée qu'à partir de la fonction parente.
- A accès à la portée de la fonction parente (arguments et variables).
- La fonction parente ne peut pas accéder à ses données.

fonction-imbriquee.js

```
1 function ajouteCarrés(a, b) {
2   function carré(x) {
3     return x * x;
4   }
5   return carré(a) + carré(b);
6 }
7 a = console.log(ajouteCarrés(2, 3)); // 13
8 b = console.log(ajouteCarrés(3, 4)); // 25
9 c = console.log(ajouteCarrés(4, 5)); // 41
```

Fermeture

Une fonction imbriquée retournée par la fonction parente

- A accès à la portée de la fonction parente (variables liées).
- A accès à des variables libres.
- Une nouvelle fermeture est créée à chaque appel de la fonction parente (avec un nouvel environnement).

fonction-fermeture.js

```
1 function parente(x) {
2   function fille(y) {
3     return x + y;
4   }
5   return fille;
6 }
7 fn_fille = parente(3); // Fonction qui ajoutera 3 à son argument
8 console.log(fn_fille(5)); // 8
9 console.log(parente(3)(5)); // 8
```

Fermetures : chaînage des portées

“Transitivité"" de la portée

fonction-fermeture-chainage.js

```
1  function A(x) {
2    function B(y) {
3      function C(z) {
4        console.log(x + y + z);
5      }
6      C(3);
7    }
8    B(2);
9  }
10 console.log(A(1)); // 6 (1 + 2 + 3)
```

Fermetures et encapsulation

Les fermetures

- Permettent d'encapsuler les arguments passés à leur fonction parente.
- Offrent un accès persistent à ces données.

Idiome

Dans une fonction, retourner un objet dont les méthodes sont des fermetures manipulant les variables internes de la fonction :

- Accesseurs : accès en lecture.
- Mutateurs : accès en écriture.

Fermetures et encapsulation

fonction-fermeture-encapsulation.js

```
1  var créerAnimal = function(nom) {
2    var sexe;
3    return {
4      setNom : function(nouveauNom) {
5        nom = nouveauNom;
6      },
7      getNom : function() {
8        return nom;
9      },
10     getSexe : function() {
11       return sexe;
12     },
13     setSexe : function(nouveauSexe) {
14       if (typeof nouveauSexe == "string"
15           && (nouveauSexe.toLowerCase() == "mâle" || nouveauSexe
16               .toLowerCase() == "femelle")) {
17         sexe = nouveauSexe;
18       }
19     }
20   }
21 }
22
23 var animal = créerAnimal("Licorne");
24 console.log(animal.getNom());// Licorne
25 animal.setNom("Bobby");
26 animal.setSexe("mâle");
27 animal.sexe = "alpha";
28 console.log(animal.getSexe()); // mâle
29 console.log(animal.getNom());// Bobby
```

Arguments de fonctions

Les fonctions JS sont variadiques

L'objet `arguments` stocke les arguments d'une fonction tel un tableau (ce n'en est pas un) :

- `arguments.length` : nombre d'arguments.
- `arguments[i]` : `i+1`-ème d'argument.

fonction-arguments.js

```
1  function monConcat(séparateur) {
2    var result = "";
3    var i;
4    for (i = 1; i < arguments.length; i++) {
5      result += arguments[i] + séparateur;
6    }
7    return result;
8  }
9
10 // "A, B, C, "
11 console.log(monConcat(", ", "A", "B", "C"));
12 // "A; B; C; D; "
13 console.log(monConcat("; ", "A", "B", "C", "D"));
```

Paramètres par défaut

Avant ES6 (ECMAScript 2015)

fonction-parametres-defaut.js

```
1 function multiplier(a, b) {
2   b = typeof b !== 'undefined' ? b : 1;
3   return a * b;
4 }
5
6 console.log(multiplier(5)); //5
```

Depuis ES6

fonction-parametres-defaut-es6.js

```
1 function multiplier(a, b = 1) {
2   return a*b;
3 }
4
5 console.log(multiplier(5)); //5
```

Paramètres du reste

Syntaxe `f(...x)`

Représente un nombre indéfini d'arguments contenus dans un tableau.

- Permet d'utiliser des expressions de fonctions fléchées.

fonction-parametres-reste.js

```
1 function carré(...args) {
2   return args.map(x => x * x);
3 }
4 console.log(carré(1, 2, 3)); // [1, 4, 9]
5
6 function multiplier(facteur, ...args) {
7   return args.map(x => facteur * x);
8 }
9 console.log(multiplier(2, 1, 2, 3)); // [2, 4, 6]
```

Expression de fonction fléchée (arrow function)

Fonction anonyme sous forme d'expression fléchée

Syntaxe plus concise que les expressions de fonctions classiques.

fonction-expression-flechee.js

```
1  var a = [
2    "Hydrogen",
3    "Helium",
4    "Lithium",
5    "Beryllium"
6  ];
7
8  var a2 = a.map(function(s){ return s.length });
9  console.log(a2); // affiche [8, 6, 7, 9]
10 var a3 = a.map( s => s.length );
11 console.log(a3); // affiche [8, 6, 7, 9]
```

Fonctions prédéfinies

Nom	Description
`eval`	Evalue du code JS contenu dans une chaîne de caractères.
`uneval`	Crée une représentation sous la forme d'une chaîne de caractères pour le code source d'un objet.
`isFinite`	Détermine si la valeur passée est un nombre fini. Si nécessaire, le paramètre sera converti en un nombre.
`isNaN`	Détermine si une valeur est `NaN` ou non.
`parseFloat`	Convertit une chaîne de caractères en un nombre flottant..
`parseInt`	Convertit une chaîne de caractères et renvoie un entier dans la base donnée.
`decodeURI`	Décode un URI créé par `encodeURI()` ou une méthode similaire.
`decodeURIComponent`	Décode un composant d'URI créé par `encodeURIComponent()` ou une méthode similaire.
`encodeURI`	Encode un URI en remplaçant chaque exemplaire de certains caractères par un, deux, trois voire quatre séquences d'échappement représentant l'encodage UTF-8 du caractère.
`encodeURIComponent`	Encode un composant d'URI en remplaçant chaque exemplaire de certains caractères par un, deux, trois voire quatre séquences d'échappement représentant l'encodage UTF-8 du caractère.