Javascript

Les fonctions

Définition

Une fonction est une procédure, un ensemble d'instructions effectuant une tâche ou calculant une valeur.

En javascript, les fonctions sont des objets de première classe. Cela signifie qu'elles peuvent être manipulées et échangées comme tous les autres objets JavaScript.

Les fonctions sont des objets instances du constructeur `Function`.

Déclaration

Instruction `function`

```
function round(nb,nbDecimales) {
  let pow = Math.pow(10,nbDecimales);
  return Math.round(nb*pow)/pow;
}
```

Expression de fonction

```
let round = function(nb,nbDecimales) {
  let pow = Math.pow(10,nbDecimales);
  return Math.round(nb*pow)/pow;
};
```

Fonction fléchée

```
let round = (nb,nbDecimales) => {
  let pow = Math.pow(10,nbDecimales);
  return Math.round(nb*pow)/pow;
};
```

Différences

Instruction function

```
round(3.5123,2); //3.51

typeof round; //"function"

function round(nb,nbDecimales) {
  let pow = Math.pow(10,nbDecimales);
  return Math.round(nb*pow)/pow;
}
```

Expression de fonction

```
round(3.5123,2); //Exception: round is not a function

typeof round; //"undefined"

let round = function(nb,prec) {
  let pow = Math.pow(10,nbDecimales);
  return Math.round(nb*pow)/pow;
}
```

Fonctions fléchées

Syntaxe plus courte pour créer des fonctions.

```
//Sans fonction fléchée
let length = function(str) { return str.length; }
let somme = function(x,y) { return x + y; }


//Avec
let length = str => str.length;
let somme = (x,y) => x + y;
```

Si la fonction ne contient qu'une seule instruction, les accolades sont facultatives et le mot clé `return` est alors omis.

Constructeur Function

Il est également possible de passer par le constructeur `Function` pour créer une fonction. Mais cette écriture n'est jamais utilisée.

```
let round = new Function(nb,nbDecimales,"let pow = Math.pow(10,nbDecimales); return Ma
```

Arguments

Grande souplesse au niveau des arguments.

```
function round(nb,nbDecimales) {

    //un argument omis à la valeur undefined
    if (nbDecimales === undefined) nbDecimales = 0;

    let pow = Math.pow(10,nbDecimales);
    return Math.round(nb*pow)/pow;
}

round(5.265485, 2, "toto", "tata"); //5.27
round(5.265485); //5
```

Stratégie d'évaluation

C'est ce qui détermine quand évaluer les arguments à l'appel d'une fonction et comment passer les arguments à la fonction.

En javascript, les paramètres sont d'abord évalués, puis la fonction est évaluée.

```
function somme(a,b) { return a+b; }

somme( 5*2, 10-1 );// équivalent à somme(10,9)

somme( somme(2,3), somme(3,8) ); //équivalent à somme(5,11)
```

Stratégie d'évaluation

Valeurs primitives

La fonction reçoit une copie des valeurs primitives.

```
function ajoutSignature(texte) {
  texte = texte + "\n\nMr Dupont\nMétéo-France"
  return texte;
}

let mess = "Bonjour, bla bla bla.";

ajoutSignature(mess); //Bonjour, bla bla bla.\n\nMr Dupont\nMétéo-France"

mess; // "Bonjour, bla bla bla."
```

Stratégie d'évaluation

Objets

La fonction reçoit une copie de la référence vers l'objet.

Les modifications sur les propriétés d'un argument ont des effets sur l'objet externe.

```
function changeProp(obj) { obj.x = 5; }

let monObjet = { x:0, y:0 };
changeProp(monObjet);
console.log(monObjet); // { x:5, y:0 }
```

Arguments : valeurs par défaut

```
function maFonction(a = "toto", b = "tata", c = "tutu") {

    /* blah blah */

}
```

Paramètres du reste

Cette écriture permet d'obtenir dans un tableau des paramètres "supplémentaires"

```
function maFonction(paramRequis1, paramRequis2, ...paramsOptionnels) {
  for (let i=0;i<paramsOptionnels.length;i++) {
    window.alert(paramsOptionnels[i]);
  }
}

maFonction("toto","tata","et","plus","encore");
//n'affichera que "et" "plus" "encore"
```

Par extension, on peut récupérer l'ensemble des arguments dans un tableau (on utilisait autrefois l'objet arguments)

```
function maFonction(...args) {
  // args est un tableau contenant tous les arguments passés à la fonction
}
```

Hash d'arguments

Problème

```
function changeStyle(obj,color,fontSize,fontFamily,fontWeight) {
  if (color) obj.style.color = color;
  if (fontSize) obj.style.fontSize = fontSize;
  if (fontFamily) obj.style.fontFamily = fontFamily;
  if (fontWeight) obj.style.fontWeight = fontWeight;
}

let div = document.getElementById("maDiv");

changeStyle(div,null,"12px",null,"bold");
```

- il faut connaître l'ordre des arguments
- si seul le dernier nous intéresse, on est obligé de passer des valeurs aux précédents
- l'appel est peu lisible pour celui qui ne connaît pas la fonction

Hash d'arguments

Solution

```
function changeStyle(obj,options) {
  if (options.color) obj.style.color = options.color;
  if (options.fontSize) obj.style.fontSize = options.fontSize;
  if (options.fontFamily) obj.style.fontFamily = options.fontFamily;
  if (options.fontWeight) obj.style.fontWeight = options.fontWeight;
}

let div = document.getElementById("maDiv");

changeStyle(div,{fontSize:"12px",fontWeight:"bold"});
```

- l'ordre des propriétés est sans importance
- seules les propriétés qui nous intéressent sont passées
- l'appel de la fonction est explicite

Hash d'arguments

Avec affectation par décomposition

```
function changeStyle(obj, { color, fontSize, fontFamily, fontWeight }) {

  if (color) obj.style.color = color;
  if (fontSize) obj.style.fontSize = fontSize;
  if (fontFamily) obj.style.fontFamily = fontFamily;
  if (fontWeight) obj.style.fontWeight = fontWeight;
}

let div = document.getElementById("maDiv");

changeStyle(div,{fontSize:"12px",fontWeight:"bold"});
```

Hash d'arguments

Affectation par décomposition et valeurs par défaut

```
function changeStyle(obj,{ color="black", fontSize="12px" }) {
  obj.style.color = color;
  obj.style.fontSize = fontSize;
}

let div = document.getElementById("maDiv");

changeStyle(div,{fontSize:"12px",color:"black"});
```

Hash d'arguments

Décomposition à plusieurs niveaux

```
function changeStyle(obj,{ font : { color="black", size="12px" } }) {
  obj.style.color = color;
  obj.style.fontSize = size;
}

let div = document.getElementById("maDiv");

changeStyle(div,{ font : { size :"12px", color : "black"});
```

Hash d'arguments

Décomposition avec renommage

```
function changeStyle(obj,{ font : { color : fontColor, size : fontSize } }) {
  obj.style.color = fontColor;
  obj.style.fontSize = fontSize;
}

let div = document.getElementById("maDiv");

changeStyle(div,{ font : { size :"12px", color : "black"});
```

Fonctions de rappel (callbacks)

C'est une fonction passée en argument d'une autre fonction.

```
function logMessage() {
  console.log("coucou");
}

window.setTimeout(logMessage,3000);
//la fonction setTimeout attend en premier argument une fonction,
//qui sera executée après 3000 millisecondes.

let lien = document.getElementById("monLien");
lien.addEventListener("click",logMessage);
//la fonction addEventListener attend un nom d'évènement
//et une fonction qui sera exécutée au déclenchement de l'évènement.
```

Fonctions de rappel (callbacks)

Erreur fréquemment commise

```
function logMessage() {
  console.log("coucou");
}

window.setTimeout( logMessage() ,3000);
```

La fonction *logMessage* est exécutée immédiatement et c'est la valeur renvoyée par la fonction (ici *undefined*) qui est passée en argument de la fonction setTimeout.

Fonctions anonymes

L'exemple précédent peut s'écrire de la manière suivante :

```
window.setTimeout(function() {
  console.log("coucou");
},3000);

let lien = document.getElementById("monLien");
lien.addEventListener("click",function() {
  console.log("hello");
});
```

Avec des fonctions fléchées :

```
window.setTimeout(() => console.log("coucou"),3000);

let lien = document.getElementById("monLien");
lien.addEventListener("click",() => console.log("hello"));
```

Fonctions imbriquées

On peut définir une fonction à l'intérieur d'une fonction.

```
function valideFormulaire() {

    function valideChampNom() {
      /*blah blah*/
    }

    function valideChampEmail() {
      /*blah blah*/
    }

    function envoiDonnees() { /*blah blah*/}

    valideChampNom();
    valideChampEmail();
    envoiDonnees();
}
```

Portée des variables

Les fonctions internes ont accès à toutes les variables de la fonction externe.

```
function valideFormulaire() {

  let test = true;

  function valideChampNom() { test = false; }

  function valideChampEmail() { test = false; }

  valideChampNom();
  valideChampEmail();

  test;//false
}
valideFormulaire();
```

Portée des variables

L'inverse n'est évidemment pas vrai.

```
function valideFormulaire() {

  function valideChampNom() { let nom = "Toto"; }

  valideChampNom();

  nom; //Exception: nom is not defined
}
valideFormulaire();
```

Portée des variables

Conflits de nom

```
function valideFormulaire() {

  let test = true;

  function valideChampNom() {
    test; //??
    let test = false;
    test; //??
  }

  valideChampNom();

  test; //??
}
valideFormulaire();
```

La portée la plus interne l'emporte.

Problème

Comment passer une fonction avec des paramètres ?

```
function log(message) {
  console.log("Voici le message :\n" + message);
}

window.setTimeout(log,3000); //la fonction sera appelée sans argument

window.setTimeout( log("coucou") , 3000);
//la fonction est exécutée immédiatemment et
//le résultat (undefined) est passé en argument de setTimeout
```

Solution

On encapsule la fonction dans une autre

```
function log(message) {
  console.log("Voici le message :\n" + message);
}

window.setTimeout(function() {
  log("coucou");
},3000);
```

Avec une fonction fléchée

```
function log(message) {
  console.log("Voici le message :\n" + message);
}

window.setTimeout(() => log("coucou"),3000);
```

Autre problème

Mes variables ont des noms très génériques et une portée globale.
Que se passe-t-il si j'écris une autre portion de code avec des noms semblables ?
Que se passe-t-il si j'inclus une bibliothèque qui utilise des noms semblables ?

Solution : IIFE

Immediately Invoked Function Expression

```
(function() {

  function affiche(cpt) {
    window.alert("Je suis la div n°"+cpt);
  }

  let div = document.getElementById("maDiv");

  div.addEventListener("click", affiche);

}());
```

On encapsule notre code dans une fonction anonyme aussitôt exécutée, ce qui protège nos variables de l'espace global.

IIFE : complément

Cette syntaxe nous permet d'utiliser le mode strict sans risque de perturber le reste du code.

```
(function() {

  "use strict";

  /*
    mon code
  */

}());
```

C'est donc le modèle qu'on utilisera systématiquement pour écrire du code javascript.

Remarque : cela n'est plus la peine si on utilise les modules javascript (mais ils nécessitent un serveur de développement).

Méthodes

Une méthode d'un objet est une propriété de type `function`.

```
let toto = {
  nom : "toto",
  age : 25,
  vieillir : function() { this.age++; }
};

typeof toto.vieillir; //"function"
toto.vieillir();
toto.age; //26

//attention
typeof toto.vieillir(); //"undefined"
toto.age; //27
```

Dans la méthode d'un objet, le mot clé `this` fait référence à l'objet lui-même.

Méthodes

Raccourci possible

```
let toto = {
  nom : "toto",
  age : 25,
  vieillir() { this.age++; }
};
```

Méthodes

Piège avec fonction fléchée

```
let toto = {
  nom : "toto",
  age : 25,
  vieillir : () => this.age++
};

toto.vieillir(); // this.age est équivalent à window.age
toto.age; // NaN
```

Les fonctions fléchées ne créent pas de nouveau contexte. La valeur de `this` est la même que dans le contexte parent.