JavaScript : concepts de base

Kristen Le Liboux

Juillet 2013

Avertissements

Notions abordées

- Variables, fonctions, callbacks, fermetures (*closures*)
- Types, objets, tableaux, prototypes
- JSON

Public visé

- Déjà familier avec quelques notions de programmation
- Opérateurs (+, *, ...) et structures de contrôle (if, while, ...) supposés connus

Télécharger les exemples

Cet indicateur fait référence aux exemples qui accompagnent cette présentation :

exemple01.js

Vous pouvez les télécharger sur :

https://github.com/kleliboux/code-samples

Fondamentaux de JavaScript

Langage de programmation :

- scripté (interprété)
  pas de compilateur à proprement parler
- côté client
  s'exécute dans un navigateur en général
  (il existe des environnements côté serveur : NodeJS)
- asynchrone
  plusieurs « morceaux » peuvent s'exécuter en //

Fondamentaux de JavaScript

Dans un navigateur, permet :

- de spécifier des changements sur le document :
  - après son chargement,
  - au cours de sa vie dans la fenêtre du navigateur,
  - sur le contenu, la structure, le style
- en les planifiant à l'avance
- en interceptant des événements (souris, clavier, doigts…)

Fondamentaux de JavaScript

Mais également (API HTML5) :

- d'échanger avec un serveur (AJAX)
- de dessiner (canvas - bitmap - ou svg - vectoriel)
- de se géolocaliser
- d'enregistrer localement du contenu (cache ou bdd)
- de jouer des fichiers audio ou video
- etc...

Fondamentaux de JavaScript

3 façons d'exécuter du code JavaScript

- En l'incorporant dans du HTML
  `<script> ..... </script>`
- En le liant à du HTML
  `<script src="js/fichier.js"></script>`
- Directement dans la console du navigateur
  Dans Firefox : CTRL+MAJ+K (ALT+CMD+K sur Mac)

Fondamentaux de JavaScript

3 instructions pour démarrer

- Afficher une boite sommaire avec un message
  ```
  alert("Bonjour !");
  ```
- Écrire du texte dans la console (pour débugguer)
  ```
  console.log("Texte d'essai");
  ```
- **Écrire quelque chose dans le document, dans une balise HTML qui a un certain id**
  ```
  document.getElementById("monId")
    .innerHTML = "<p>Mon contenu</p>";
  ```

exemple01.js

Notion de fonction


```
// déclaration de la fonction
function afficher(id, message)
{
    console.log("Message: " + message);
    document.getElementById(id).innerHTML = message;
}

// deux exemples d'appel
afficher("special1", "<p>Du contenu bien frais !</p>");
afficher("special2", "Un autre contenu.");
```

Un ensemble d'instructions prêt à être utilisé après sa déclaration.

- Permet la ré-utilisabilité du code
- Deux temps : la déclaration, puis l'appel
- Peut avoir des paramètres (ici **id** et **message**)

Notion de fonction


```
// déclaration d'une autre fonction
function perimetre(longueur, largeur)
{
   return 2 * (longueur + largeur);
}

// appel imbriqué des deux fonctions précédentes
afficher("special3",
         "Périmètre du rectangle : " + perimetre(100, 40)
);
```

- Peut retourner une valeur avec **return**.

exemple02.js

Notion de fonction


```
// déclaration de la fonction suivi de l'appel
(function(id, message) {
    console.log("Message: " + message);
    document.getElementById(id).innerHTML = message;
} ("special4", "Un dernier contenu."));
```

- Peut être déclarée et appelée du même coup.
  Dans ce cas, le nom est souvent omis.
  Penser aux parenthèses.

exemple02.js

Opérateurs

`+` Addition de nombres et concaténation de chaînes

`-` `*` `/` Division flottante

`%` Reste de la division entière

`(x - x % y) / y` Division entière

`=` Affectation de valeur à une variable

`==` Comparaison large (conversion à la volée)

`===` Comparaison stricte (sans conversion de type)

`>` `<` `>=` `<=`

`x ? y : z` Si *x* == true, vaut *y* sinon vaut *z*

Structures de contrôle

if(*expr*) { … }

if(*expr*) { … } else { … }

if(*expr*) { … } else if { … } else if { … } … else { … }

while(*expr*) { … }

do { … } while(*expr*)

switch(*expr*) { case *value1* : … case *value2* : … default : … }

break

for(*ini ; cond ; iter*) { … }

Notion de variable


```
// déclaration d'une variable
var monMessage = "<p>Du contenu bien frais !</p>";

// utilisation avec la fonction précédente
afficher("bloc1", monMessage);
afficher("bloc2", monMessage);
```

Une variable est un emplacement nommé de la mémoire, auquel on associe une donnée

- Ici : monMessage contient le texte « <p>Du contenu bien frais !<p> »
- On la déclare avec **var** et on peut la réutiliser partout après.

Notion de type (début)


```
// déclaration de trois variables
var monMessage = "<p>Du contenu bien frais !</p>";
var monNombre  = 17.2;
var maFonction = function(id, message)
{
    console.log("Message: " + message);
    document.getElementById(id).innerHTML = message;
}
// utilisation
maFonction("bloc1", monMessage);
```

Dans une variable, on peut stocker pour l'instant :

- Un booléen (true ou false)
- Une chaîne de caractères
- Un nombre
- Ou une fonction

Notion de type (début)

Types simples :

- Booléen
- Chaîne de caractères
- Nombre

Conversion en type simple :

- parseInt, parseFloat
- String

```
var x = 17;
var y = 18.2;
var c = "bonjour";
var d = "15.25";

alert(x + parseInt(d));
alert(x + parseFloat(d));
alert(string(x) + c);

// Résultats :
// 32
// 32.25
// 17bonjour
```

Le type « fonction »

Puisque « fonction » est un type de données en JavaScript, on peut :

- le passer en paramètre d'une fonction
- le renvoyer comme valeur d'une fonction

(slides suivants)

Le type « fonction »


```
function afficher(message) {
    console.log("Message: " + message);
    document.getElementById("info").innerHTML = message;
}
function calculerEtAfficher(cote, affichage) {
    var resultat = 4 * cote;
    affichage("Périmètre du carré : " + resultat);
}
calculerEtAfficher(10, afficher);
```

Ici, la fonction « afficher » est un *callback* :
elle est passée en paramètre d'une autre fonction,
au même titre que n'importe quel autre paramètre.

exemple03.js

Le type « fonction »


```
function creeAffichage(id) {
    return function(message) {
        console.log("Message: " + message);
        document.getElementById(id).innerHTML = message;
    }
}
creeAffichage("info1")("Du contenu tout frais !");
calculerEtAfficher(10, creeAffichage("info2"));
```

Ici, « creeAffichage » est une « usine à fonctions » :
elle crée une fonction à la demande, que l'on peut stocker, réutiliser, passer en paramètre, etc.

exemple04.js

Notion de visibilité

Une fois qu'une variable est déclarée, elle n'est pas nécessairement visible (utilisable) en tous les endroits du script.

Il y a plusieurs règles de visibilité en JavaScript.

(slides suivants)

Notion de visibilité


```
var globale1 = 17;
globale2 = 18;

function maFonction() {
    globale3 = 19;
    var locale = 20;
    alert(globale1 + globale2 + globale3 + locale); // ok
}

alert(globale3); // erreur
maFonction();
alert(globale3); // ok
```

Règle 1

Une variable déclarée à l'extérieur d'une fonction ou sans le mot-clé **var** est dite **globale**, et visible partout après sa définition.

Notion de visibilité


```
// déclaration d'une fonction
function maFonction()
{
    var monMessage = "<p>Du contenu bien frais !</p>";
    document.getElementById(id).innerHTML = monMessage;
}

maFonction(); // ok : exécute la fonction
```

```
alert(monMessage); // undefined : la variable monMessage
                   // n'est pas accessible ici
```

Règle 2

Une variable non globale est dite **locale**.

Elle n'est accessible que dans la fonction où elle est définie…

Notion de visibilité


```
// déclaration d'une fonction
function afficheur(id)
{
    var monMessage = "<p>Du contenu bien frais !</p>";
    return function() {
        document.getElementById(id).innerHTML = monMessage;
    }
}

var disp = afficheur("bloc");
disp(); // ok
```

Règle 2

...y compris dans les fonctions créées dans le contexte.

Notion de visibilité : fermetures


```
// déclaration d'une fonction
function afficheur(id)
{
    var monMessage = "<p>Du contenu bien frais !</p>";
    return function() {
        document.getElementById(id).innerHTML = monMessage;
    }
}
```

Règle 3

On appelle **fermeture** d'une fonction (closure) l'ensemble des variables qui lui sont visibles au moment de sa déclaration, globales et locales.

La fermeture d'une fonction F est visible quel que soit le moment de l'appel de F, même si l'appel a lieu en dehors du contexte de déclaration de F.

Notion de visibilité : fermetures


```
// déclaration d'une fonction
function afficheur(id)
{
    var monMessage = "<p>Du contenu bien frais !</p>";
    return function() {
        document.getElementById(id).innerHTML = monMessage;
    }
}
```

Ici la fermeture de la fonction interne (anonyme) est { id, monMessage }.

La fonction est créée à l'intérieur de **afficheur**, mais elle sera appelée à l'extérieur. Sa fermeture sera alors quand même visible.

Exemple


```
function externe()
{
   var a = 7;
   var interne = function()
   {
      alert(a);
   }
   a = 8;
   return interne;
}

externe()();
```

Question : affichage final ?

exemple05.js

Exemple


```
function externe()
{
   var a = 7;
   var interne = function()
   {
      alert(a);
   }
   a = 8;
   return interne;
}

externe()();
```

Réponse : 8.

La fermeture, c'est l'environnement des variables qui existent au moment de la création de la fonction. Leurs valeurs sont les dernières valeurs connues au moment de l'appel.

Notion d'objet

JavaScript supporte la notion d'objet :

- objet = attributs + méthodes
  - voiture = { marque, modele } + { accelerer, freiner }

En première approche, il n'y a pas vraiment de distinction entre attribut et méthode :

- une méthode est un attribut de type fonction
- on dispose de la variable **this**

Notion de constructeur d'objet


```
function Voiture(marque, modele) {
   this.marque = marque;
   this.modele = modele;
   this.afficher = function() {
      alert(this.marque + " " + this.modele);
   }
}
```

```
var maVoiture = new Voiture("Ford", "Fiesta");
maVoiture.afficher();
console.log(maVoiture.marque);
```

Toute fonction appelée avec new **est un constructeur d'objet et peut donc utiliser** this**.**

On peut utiliser **this** dans toutes les fonctions. En dehors d'un contexte d'objet, **this** représente l'espace global.

Accès aux membres


```
// Accès aux attributs et méthodes
console.log(maVoiture.marque);
maVoiture.afficher();

// Ajout de membre a posteriori
maVoiture.km = 8000;
maVoiture.rouler = function(distance) {
   this.km += distance;
}

// Utilisation des crochets
alert( maVoiture["km"] );
```

exemple06.js

- Tous les membres d'un objet sont publics.
- On peut ajouter des membres a posteriori.
- Utilisation de l'opérateur . (point) et des [] (crochets).
- On peut mettre une variable entre les crochets (de type chaîne de caractères...)

Format JSON


```
// déclaration d'un objet selon la syntaxe JSON
var monObjet = {
    "nom" : "Fred",
    "age" : 28,
    "afficher" : function(id) {
        document.getElementById(id).innerHTML = this.nom;
    }
};
```

```
// utilisation de l'attribut nom
alert(monObjet.nom);
// utilisation de la méthode afficher
monObjet.afficher("bloc");
```

Exemple en utilisant la syntaxe JSON

~ couples clés/valeurs

Très pratique, à utiliser abondamment.

Format JSON


```
// déclaration d'un objet selon la syntaxe JSON
var monObjet = {
    "nom" : "Fred",
    "age" : 28,
    "afficher" : function(id) {
        document.getElementById(id).innerHTML = this.nom;
    }
};
```

Syntaxe JSON (JavaScript Object Notation) :

- liste de couples attributs/valeurs dans une paire de { }
- notez les séparateurs (deux-points et virgules)
- guillemets conseillés pour les attributs mais non obligatoires (ici : nom, age, afficher)

Discussion


```
function affichageObjet(id) {
    document.getElementById(id).innerHTML = this.nom;
}
var monObjet1 = {
    "nom" : "Fred",
    "afficher" : affichageObjet
};
var monObjet2 = {
    "nom" : "Paul",
    "afficher" : affichageObjet
};
```

Ici les méthodes sont dupliquées en mémoire.
Lorsque plusieurs objets ayant la même structure doivent être construits, on préfère utiliser la notion de **prototype** (slides suivants).

Notion de prototype


```
function Voiture(marque, modele) {
   this.marque = marque;
   this.modele = modele;
}
Voiture.prototype.afficher = function() {
   alert(this.marque + " " + this.modele);
}
var maCaisse1 = new Voiture("Ford", "Fiesta");
var maCaisse2 = new Voiture("Renault", "Espace");
maCaisse1.afficher();
maCaisse2.afficher();
```

exemple07.js

Un constructeur C peut être associé à un prototype P, qui est un objet «modèle». Tous les objets construits avec C possèdent alors les membres de P.

Notion d'héritage

On peut simuler un héritage en JavaScript :

- Le prototype est un objet « modèle »
- Donc le prototype du fils doit être une instance du père, à laquelle on ajoute des membres spécifiques
- Et le constructeur du fils doit appeler celui du père

Opérateur instanceof **:**

```
console.log( maVoiture instanceof Voiture ); // true
```

Voir

exemple08.js

Structure de contrôle for..in


```
for( var prop in maVoiture ) {
   console.log( prop + ": " + maVoiture[prop] );
}
```

Pour itérer sur chacun des membres d'un objet :

- L'ordre de l'itération est **non fiable**
- Ne pas ajouter de membres à l'objet pendant l'itération
- Passe en revue les attributs et les méthodes

Structure de contrôle for..in


```
var maCaisse = new Voiture("Ford", "Fiesta");

for(var prop in maCaisse) {
   console.log( prop + ": " + maCaisse[prop] );
}
```

exemple09.js

Pour itérer sur chacun des membres d'un objet :

- Attributs et méthodes
- Y compris les membres « hérités » par prototype
- Dans un ordre non spécifié
- Ne pas ajouter de membres à l'objet pendant l'itération

Structure de contrôle for..in


```
var maCaisse = new Voiture("Ford", "Fiesta");

for(var prop in maCaisse) {
   if( maCaisse.hasOwnProperty(prop) ) {
      console.log( prop + ": " + maCaisse[prop] );
   }
}
```

exemple09.js

- Il est possible de distinguer les membres «propres» de ceux qui viennent du prototype.

Notion de tableau


```
var monTableau = new Array();

monTableau[0] = "Toto";
monTableau[1] = 17;
monTableau[2] = true;
monTableau[4] = "hello";

console.log(monTableau.length + " éléments :");
for(var i = 0; i < monTableau.length; i++) {
   console.log(monTableau[i]);
}

// Ici monTableau[3] == undefined
```

exemple10.js

JavaScript propose un constructeur de tableaux

- Utilisation des crochets pour l'accès
- Propriété **length** pour avoir le nombre d'éléments

Tableau au format JSON


```
// déclaration d'un tableau selon la syntaxe JSON
var monTableau = [
    "Fred",
    "Gina",
    "Mehdi"
];

// les éléments sont numérotés à partir de zéro
alert(monTableau[0]); // Fred

// nombre d'éléments dans le tableau
alert(monTableau.length); // 3
```

Syntaxe raccourcie pour la déclaration d'un tableau

- Utilisation des crochets pour la déclaration
  (au lieu des accolades pour les objets)

Notion de type (fin)


```
// déclaration de quatre variables
var monMessage = "<p>Du contenu bien frais !</p>";
var monNombre  = 17.2;
var maFonction = function(id, message) {
    document.getElementById(id).innerHTML = message;
}
var monObjet = {
    "nom" : "Fred",
    "diplomes" : ["BAC", "BTS", "IMI"],
    "afficher" : function(id) {
        document.getElementById(id).innerHTML = this.nom;
    }
}
```

Types de données :

- Undefined
- Booléen
- Chaîne de caractères
- Nombre
- Fonction
- Objet
- (Tableau) = objet particulier

Objets natifs du navigateur

window

- l'objet racine
- toute variable globale est un attribut de **window**
- `var x = 0;` <=> `window.x = 0;`
  en contexte global

Objets natifs du navigateur

document

- représente le document actuellement affiché
- possède un certain nombre de méthodes et d'objets sous-jacents documentés par le W3C : le DOM (Document Object Model)
- exemple :

```
document.getElementById("monId").innerHTML
    = "<p>Mon contenu</p>";
```

Objets natifs du navigateur

console

- la console de debuggage (CTRL+MAJ+K)
- exemple :

```
console.log("Texte d'essai");
```

Objets natifs du navigateur

Math

- un objet agrégeant des fonctions mathématiques
- exemple :

```
var x = Math.round(18.22);
```

Timeouts


```
function action() {
   alert("Tadam !");
}

// Toutes les 1000 millisecondes, Tadam !
var timeout = setInterval(action, 1000);

// Au bout de 3500 millisecondes, on arrete ça.
setTimeout(
    function() {
       clearTimeout(timeout);
    },
    3500
);
```

exemple11.js

Un timeout est une fonction qui s'exécutera périodiquement ou au bout d'un temps défini.

- se déclare avec **setInterval** ou **setTimeout**
- s'annule avec **clearInterval**

Résumé : notions abordées

Types en JavaScript

- Undefined
- Simples : booléen, nombre, chaîne
- Complexes : fonction, objet(, tableau)

Notions

- Visibilité des variables et fermeture
- Objet, prototype
- Syntaxe JSON
- Objets natifs : window, document, console, Math…
- Timeouts

Encore un exemple


```
function makeFunctions()
{
   var arr = new Array();
   for(var i = 0; i < 5; i++) {
      arr[i] = function(j) {
         console.log(i+j);
      }
   }
   return arr;
}

var f = makeFunctions();
f[2](10);
```

Question : quel affichage dans la console ?

Merci !

Questions, commentaires, etc…

Contactez-moi sur Twitter : @novlangue