Licence 1

Informatique, Développement d'Application

Cours : Introduction au JavaScript

Séquence 1 : Le langage JavaScript

UVS
UNIVERSITÉ VIRTUELLE DU SÉNÉGAL

1. Chapitre I : Le langage JavaScript

1.1. Généralités

JavaScript (communément appelé JS) est le langage de programmation du Web. La majorité des sites Web modernes utilisent JavaScript, et tous les navigateurs Web modernes, sur les ordinateurs, les consoles de jeux, les tablettes et les téléphones intelligents, comprennent et interprètent du JavaScript, ce qui rend JavaScript le langage de programmation le plus répandu dans l'histoire. JavaScript fait partie de la triade de technologies que tous les développeurs Web doivent apprendre: HTML pour spécifier le contenu des pages Web, CSS pour spécifier la présentation des pages Web et JavaScript pour spécifier le comportement des pages Web.

Le JavaScript était à la base un langage côté client standardisé par l'ECMA (European Computer Manufacturer's Association) sous le nom de ECMAScript. Depuis 2009, l'utilisation du JavaScript a beaucoup évoluée aussi bien au niveau client qu'au niveau serveur (avec l'engouement autour de NodeJS). Afin de suivre ces évolutions une nouvelle version de l'ECMAScript a été ratifiée en Juin 2015 ; l'ECMAScript 2015 (aussi appellé ES6). A la date de rédaction de ce cours , l'ECMAScript 5 est la version la mieux supportée par la plupart des navigateurs.

Ces caractéristiques globales à retenir sont :

- c'est un langage de script
- c'est un langage orienté objet
- c'est un langage interprété par le navigateur

Un script, en informatique, est, par opposition à un langage compilé, un langage qui s'interprète. Ici, l'interprète du JavaScript, c'est le navigateur du visiteur (le client).

Il est important de préciser que JavaScript est différent du langage de programmation Java.

1.2. Débuter en JavaScript

1.2.1. Logiciels nécessaires

JavaScript est un langage facile à mettre en oeuvre. De même que pour créer une page HTML, il vous suffit :

- d'un éditeur de texte : Bloc-notes est suffisant, mais d'autres logiciels offrent plus d'options, comme la coloration syntaxique, qui est très appréciable. Nous allons donc utiliser Visual Studio Code .
- d'un navigateur : vous en avez tous sur votre ordinateur

Rendez-vous donc à l'adresse https://code.visualstudio.com puis téléchargez et installez l'éditeur (voir figure 1.2.1) .

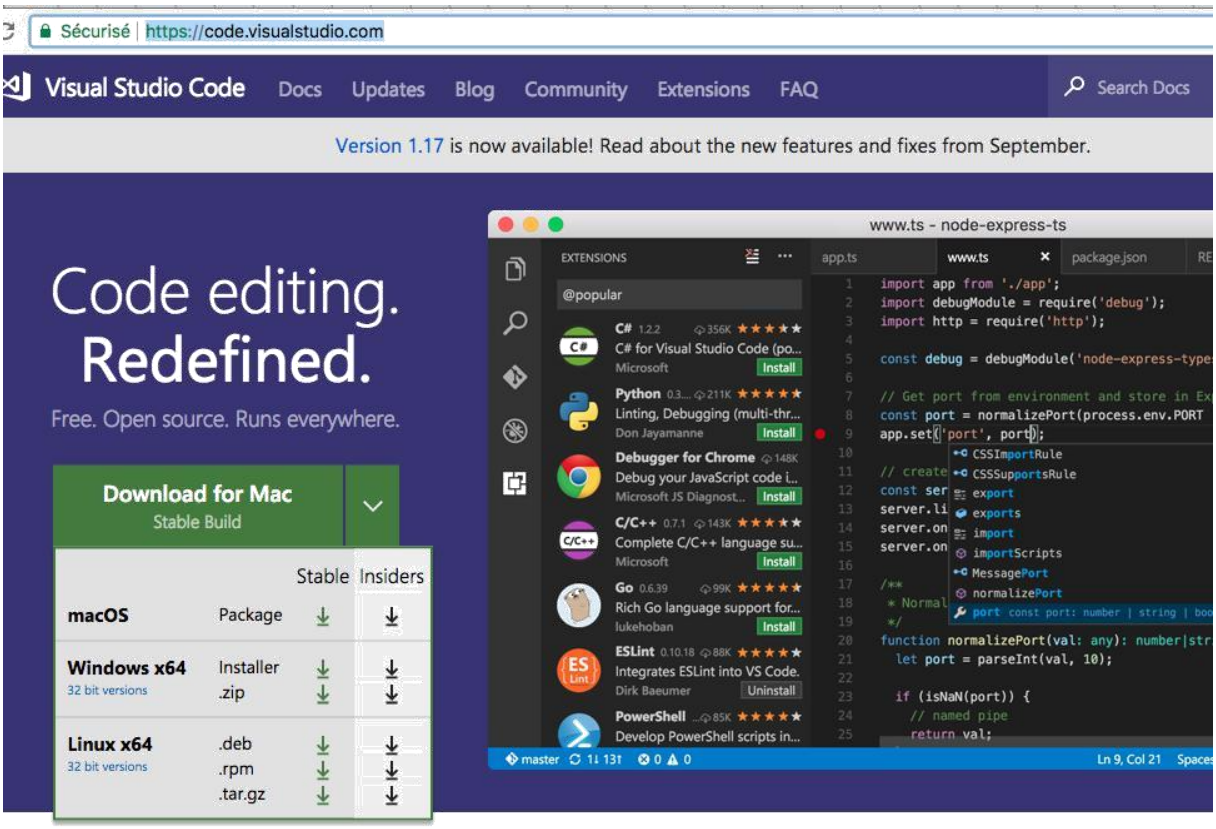

Figure 1.2.1 : Editeur Visual Studio Code

1.2.2. Intégration dans une page HTML

Le code Javascript s'insère dans une page HTML. Pour rappel, voici la structure minimale d'une page HTML.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
     <!-- en-tete du document -->
</head>
<body>
    <!-- corps du document -->
</body>
</html>
```

Il y a trois méthodes différentes pour insérer du JavaScript dans une page Web.

1.2.3. Directement dans les balises

La première méthode consiste à écrire le code directement **à l'intérieur de la balise** qui va être concernée par le script. Pour insérer le code dans une balise, une nouvelle propriété est nécessaire. Il s'agit du gestionnaire d'événements (en anglais : event handler) :

Cette propriété est caractéristique du JS
Elle porte le **nom de l'événement** qui doit **déclencher le script** (c'est pour cela qu'on parle de "gestionnaire d'événements").
Elle contient comme valeur **le script à exécuter** lors de cet événement.

Examinons tout de suite un exemple.
Nous allons utiliser **un lien**.
On ne veut pas qu'il nous envoie sur une autre page : c'est pourquoi nous utiliserons le symbole # en guise d'adresse.
On part donc de ceci : **lien**
L'événement déclenchant le script sera un **clic de souris** sur ce lien.
Le nom de cet événement est : onclick.
Le script à exécuter sera l'ouverture d'une **boîte de dialogue**. Le code sera donc : **alert('Bonjour !');**

```
    <a href="#" onclick="alert('Bonjour !');">lien</a>
```

1.2.4. Entre les balises <script> et </script>

La seconde solution consiste à écrire le script entre deux balises spécifiques, <script> et </script>. On peut les placer soit dans l'en-tête (<head> ... </head>), soit dans le corps (<body> ... </body>) de la page.
Il faut commencer par préciser au navigateur que notre script est du JavaScript.
Pour cela, on rajoute la propriété type="text/javascript", ce qui nous donne ceci :

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
     <!-- en-tete du document -->
     <script type="text/javascript">

          /* votre code javascript se trouve ici
          c'est deja plus pratique pour un script de plusieurs lignes */

    </script>
</head>
```

1.2.5. Placer le code dans un fichier séparé

Comme pour le CSS, on peut très bien placer notre code dans un fichier indépendant. On dit que **le code est importé depuis un fichier externe**. L'extension du fichier externe contenant du code JavaScript est **.js**.

On va indiquer aux balises le nom et l'emplacement du fichier contenant notre (ou nos) script(s), grâce à la propriété src (comme pour les images).

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
     <!-- en-tete du document -->
     <script type="text/javascript"
src="script.js"></script> </head>
```

1.2.6. Exemples de codes JavaScript

Souvent, la meilleure façon d'apprendre est de faire. Le but de ce paragraphe n'est pas d'expliquer tout ce qui se passe. Il y a beaucoup de choses qui ne seront pas familières et confuses, et je vous conseille de ne pas essayer de tout comprendre pour le moment. Le but de ce chapitre est de vous montrer ce que vous pouvez faire avec JavaScript.

Pour nos premières applications, nous allons utiliser un seul fichier contenant le HTML et le JavaScript.

Télécharger les fichiers ci-dessous puis ouvrez-les avec un navigateur pour voir quelques applications du langage JavaScript.

https://raw.githubusercontent.com/freemanpolys/javascript/master/helloworld/demo1.html
https://raw.githubusercontent.com/freemanpolys/javascript/master/helloworld/demo2.html
https://raw.githubusercontent.com/freemanpolys/javascript/master/helloworld/demo3.html

Vous pouvez directement voir le résultat de ces démos aux adresses suivantes :

Demo 1 : http://course-static.mygglo.com/javascript/helloworld/demo1.html
Demo 2 : http://course-static.mygglo.com/javascript/helloworld/demo2.html
Demo 3 : http://course-static.mygglo.com/javascript/helloworld/demo3.html

1.3. Les variables et les structures de contrôles

La structure d'un langage de programmation est l'ensemble des règles élémentaires qui spécifient comment vous écrivez des programmes dans cette langue. C'est la syntaxe de niveau inférieur d'une langue; il spécifie des choses telles que les noms de variables, les

caractères de délimiteur pour les commentaires et la façon dont une instruction de programme est séparée de la suivante.

Comme beaucoup de langages de programmation, JavaScript utilise le point-virgule (;) pour séparer les énoncés les uns des autres. En JavaScript, vous pouvez généralement omettre le point-virgule entre deux instructions si ces instructions sont écrites sur des lignes distinctes.

1.3.1. Valeurs et variables

Les programmes informatiques fonctionnent en manipulant des valeurs telles que le nombre 3.14 ou le texte «Hello World». Les types de valeurs qui peuvent être représentés et manipulés dans un langage de programmation sont connus sous le nom de types et l'une des caractéristiques les plus fondamentales d'un Le langage de programmation est l'ensemble de types qu'il supporte. Lorsqu'un programme doit conserver une valeur pour une utilisation future, il attribue la valeur à (ou "stocke" la valeur dans) une variable. Une variable définit un nom symbolique pour une valeur et permet à la valeur d'être appelée par le nom. La façon dont les variables fonctionnent est une autre caractéristique fondamentale de tout langage de programmation.
Voici un exemple de déclaration de variable :

```
var nom;
```

nom est le nom de la variable

Le nom peut contenir les 26 lettres de l'alphabet, en majuscules et en minuscules, les 10 chiffres ainsi que le underscore (le tiret du bas, touche 8 sur les claviers français). Mais il ne doit pas commencer par un chiffre.

Il y a également des mots "interdits" : ce sont des mots clés de JavaScript, mais pour autre chose. Parmi cette liste de mots, les plus courants seraient "break", "case", "char", "continue", "delete", "double", "final", "long", "new", "public" et "super".

Le JavaScript est sensible à la casse : autrement dit, il fait la différence entre lettres majuscules et minuscules. Ainsi, nom, Nom et NOM sont trois variables différentes !

Pour modifier la valeur d'une variable, on utilise pour le symbole d'affectation = de cette manière :

```
var annee;
var message;

annee = 2006;
message = "Bonjour, visiteur";
```

Le mot-clé **var** est utilisé pour déclarer une variable jusqu'à la spécification ES5.

ES6 (aussi appelé ES2015) vous apporte de nouvelles façons de déclarer vos variables grâce à **let** et **const** mais garde aussi la déclaration par var dans la spécification du langage.

const vous permet de déclarer une variable à assignation unique, ça veut simplement dire que vous pouvez déclarer une variable qui ne contiendra qu'une valeur et qu'elle sera accessible (ce qu'on appelle sa portée) au niveau du bloc.

1.3.1.1. Les chaînes de caractères

C'est un type de donnée, une suite de caractères qui a un ordre précis.
Les guillemets " (dits "double quotes") ou les apostrophes ' (dites "simple quotes") sont utilisés pour délimiter la chaîne de caractères. Un exemple :

```
// deux chaines de caracteres
var message1 = 'Une autre chaine de caracteres'; var
message2 = "C'est une chaine de caracteres ...";

// maintenant, on les
affiche alert(message1);
alert(message2);
```

Partant d'un "exercice" dans lequel on a une variable age qui contient l'âge du visiteur et qu'on veut afficher un message annonçant : "Vous avez XX ans" (XX est l'âge).
Il faudra mettre bout-à-bout nos 3 morceaux de chaîne (le premier morceau est "Vous avez ", ensuite la variable age, et enfin " ans", sans oublier les espaces après "avez" et avant "ans"). Concaténer, c'est en fait "mettre bout-à-bout" plusieurs chaînes de caractères pour n'en former qu'une seule.

```
        var age = 18; // on crée la variable pour pouvoir tester
        alert("Vous avez " + age + " ans"); // on affiche les chaines mises bout-à-
bout
```

1.3.1.2. Les nombres

Les nombres, au même titre que les chaînes de caractères, sont un **type** de variable.
Comme ce sont des nombres , on ne met pas de guillemets .
Ils se classent en deux catégories :

- les nombres entiers : ce sont des valeurs exactes ;
- les nombres à virgule : il est à noter qu'on utilise un point pour représenter la virgure .

Exemple :

```
var nombre = 1.234;
alert(nombre);
```

Voici les opérateurs de base qu'il est possible de faire avec les nombres:

+ (addition), exemple : 52 + 19 = 71
- (soustraction), exemple : 52 - 19 = 33
***** (multiplication), exemple : 5 * 19 = 95
/ (division), exemple : 5 / 3 = 1,666...667
% (modulo) : **a % b** (si **b** est positif, ce qui est le cas dans presque toutes les applications) est le reste de la division de **a** par **b**.
Exemple : 52 % 19 = 14 (car **52** = **19***2 + **14**)

Il est possible de convertir une chaîne de caractères en nombre grâce aux fonctions :
parseInt(chaine_a_convertir) : convertir en un nombre entier
parseFloat(chaine_a_convertir) : convertir en un nombre décimal

Il est à noter que la conversion échouera et on obtiendra une erreur de type "NaN" (Not a Number) si la chaine de caractères à convertir n'est pas un nombre.

A part les opérateurs de base cités plus haut (+, -, *, / , %) , ils en existent d'autres qui simplifient certaines opérations.

L'opération ci-dessous peut être simplifiée en utilisant l'opérateur **+=** :

```
resultat = resultat + X;      // on ajoute X à la variable resultat
resultat += X;      // on augmente la valeur de resultat de X
```

Il existe, de la même manière, les opérateurs **-=** (on retranche la valeur de la deuxième variable à celle de la première), ***=** (on multiplie la valeur de la première variable par celle de la deuxième), **/=** (idem mais avec une division) et **%=**.

Lorsque l'on veut augmenter de 1 la valeur d'une variable (on dit incrémenter), par exemple pour un compteur, on utilise la notation :

```
variable++;
```

De même, pour décrémenter (diminuer la valeur de 1) une variable, le code est le suivant :

```
variable--;
```

1.3.1.3. La portée des variables

La portée d'une variable est la région du code source de votre programme dans lequel elle est définie. Une variable globale a une portée globale; il est défini partout dans votre code JavaScript. En revanche, les variables déclarées dans une fonction sont définies uniquement dans le corps de la fonction. Ce sont des variables locales et ont une portée locale. Les paramètres de fonction comptent également comme variables locales et ne sont définis que dans le corps de la fonction.

Dans le corps d'une fonction, une variable locale est prioritaire sur une variable globale portant le même nom. Si vous déclarez une variable locale ou un paramètre de fonction portant le même nom qu'une variable globale, vous masquez la variable globale:

```
var scope = "global"; // Declarer une variable globale

function checkscope() {
    var scope = "local"; // Declarer une variable locale avec le même nom
    return scope; // c'est la variable locale est retournée et non la globale
}
checkscope() // => "local"
```

1.3.2. Instructions

Les programmes JavaScript ne sont rien de plus qu'une séquence d'instructions à exécuter. Par défaut, l'interpréteur JavaScript exécute ces instructions les unes après les autres dans l'ordre dans lequel elles sont écrites. Une autre façon de «faire les choses» est de modifier cet ordre d'exécution par défaut, et JavaScript a un certain nombre d'instructions ou de structures de contrôle qui font cela:

Les instructions conditionnels comme **if** et **switch** font que l'interpréteur JavaScript s'exécute ou ignore les autres instructions en fonction de la valeur d'une expression.

Les boucles sont des instructions comme **while** et **for** exécutent d'autres instructions de façon répétitive.

Les sauts sont des instructions comme **break** , **return** et **throw** qui font que l'interpréteur saute à une autre partie du programme.

Les sections qui suivent décrivent ces différentes instructions en JavaScript et expliquent leur syntaxe.

1.3.2.1. Conditions

Les instructions conditionnelles exécutent ou ignorent les autres instructions en fonction de la valeur d'une expression spécifiée. Ces instructions sont les points de décision de votre code et sont parfois appelées «branches». Si vous imaginez un interpréteur JavaScript suivant un chemin à travers votre code, les instructions conditionnelles sont les endroits où le code se divise en deux ou plus chemins et l'interprète doit choisir le chemin à suivre.

Les sous-sections ci-dessous expliquent les conditions avec les instructions **if/else** et **switch**.

1.3.2.2. if

L'instruction if est l'instruction de contrôle fondamentale qui permet à JavaScript de prendre des décisions, ou, plus précisément, d'exécuter des instructions de manière conditionnelle. Cette déclaration a deux formes. La première est:

```
if (expression) {
        instruction 1 ;
        instrunction 2 ;
        …..
        instruction n ;
}
```

Dans cette forme, **expression** est évaluée. Si la valeur résultante est vraie, la déclaration est exécutée. Si **expression** est faux, l'instruction n'est pas exécutée.

Exemple:

```
if (username == null){
// Si la variable username est nulle ou non définie, on la définie.
        username = "John Doe";
}
```

De façon similaire :

```
if ( !username){
// Si la variable username est null, undefined, false, 0, "" ou NaN, on lui affecte // une nouvelle valeur
        username = "John Doe";
}
```

La deuxième forme de l'instruction **if** introduit une clause **else** qui est exécutée lorsque l'expression est fausse. Sa syntaxe est:

```
if (expression) {
        instruction 1 ;
}else {
        instruction 2 ;
}
```

Cette forme de l'instruction exécute **instruction1** si l'expression est vraie et exécute **statement2** si l'expression est fausse.
Par exemple:

```
if(n == 1){
        console.log("Vous avez 1 nouveau message.");
} else {
        console.log("Vous avez " + n + " nouveaux messages.");
```

```
}
```

1.3.2.3. else if

L'instruction **if/else** évalue une expression et exécute l'un des deux blocs de code, en fonction du résultat. Mais qu'en est-il lorsqu'il il y a plusieurs blocs de code ? Une façon de le faire est d'utiliser une instruction **else if**.

```
if (n == 1) {
        // Executer block
#1 }  else if (n == 2) {
        // Executer block #2
} else if (n == 3) {
        // Executer block #3
}  else {
        // si toutes expressions échouent, executer block #4
}
```

1.3.2.4. switch

Une instruction if provoque une branche dans le flux de l'exécution d'un programme, et vous pouvez utiliser **if** si vous utilisez un chemin d'accès multiple. Cependant, la solution n'est pas la même lorsque toutes les branches dépendent de la valeur de la même expression. Dans ce cas, il est inutile d'évaluer à plusieurs reprises cette expression dans plusieurs instructions if. L'instruction **switch** gère exactement cette situation. Le mot clé **switch** est suivi d'une expression entre parenthèses et d'un bloc de code dans les accolades:

```
switch(expression){
  case valeur 1 :
        // Executer block
        #1 break ;
  case valeur 2 :
        // Executer block
        #1 break ;
  case valeur n-1 :
        // Executer block #n-1
        break ;
  default:
        // Executer block #n
        break ;
}
```

Lorsque **switch** s'exécute, il calcule la valeur de l'expression, puis recherche une étiquette **case** dont l'expression évaluée donne la même valeur.

S'il ne trouve pas un **case** avec une valeur correspondante, il recherche une instruction nommée **default** . S'il n'y a pas de bloc **default**, l'instruction **switch** ignore complètement le bloc de code.

Voici un exemple de l'utilisation de l'instruction **switch**; il convertit une valeur en une chaîne de caractères selon type de la valeur de **x**:

```
switch(typeof x) {
  case 'number':
    return x.toString(16);
  case 'string':
    return '"' + x + '"';
  default:
    return String(x);
}
```

1.3.3. Les boucles

Pour comprendre les instructions conditionnelles, nous avons imaginé l'interpréteur JavaScript suivant un chemin de branchement à travers votre code source. Les instructions en boucle sont celles qui replient ce chemin sur lui-même pour répéter des parties de votre code. JavaScript a quatre instructions en boucle: **while**, **do/while**, **for**, and **for/in**. Les sous-sections ci-dessous expliquent chacune d'elle. Une utilisation courante pour les boucles est d'itérer sur les éléments d'un tableau.

1.3.3.1. while

Tout comme l'instruction if est le conditionnel de base de JavaScript, l'instruction while est la boucle de base de Java- Script. Il a la syntaxe suivante:

```
while (expression) {
        // instructions
}
```

Pour exécuter une instruction **while**, l'interpréteur commence par évaluer l'expression. Si la valeur de l'expression est fausse, l'interpréteur ignore l'instruction qui sert de corps de boucle et passe à l'instruction suivante dans le programme. Si, par contre, l'expression est vraie, l'interprète exécute l'instruction et répète, en revenant en haut de la boucle et en évaluant à nouveau l'expression. Une autre façon de dire ceci est que l'interprète exécute l'instruction à plusieurs reprises tant que l'expression est vraie. Notez que vous pouvez créer une boucle infinie avec la syntaxe **while** (**true**).

Dans la plupart des cas, vous ne voudriez pas que JavaScript exécute exactement la même opération encore et encore. Dans presque chaque boucle, une ou plusieurs variables changent à chaque itération de la boucle. Puisque les variables changent, les actions exécutées par l'instruction d'exécution peuvent différer à chaque fois dans la boucle. De plus, si la ou les variable (s) changeante (s) sont impliquées dans l'expression, la valeur de l'expression peut être différente à chaque fois dans la boucle. C'est important sinon si

l'expression , qui est vraie au début de la boucle, n'évoluera pas ,et la boucle ne finirait jamais! Voici un exemple de boucle while qui imprime les nombres de 0 à 9:

```
var count = 0;
while (count < 10) {
        console.log(count);
        count++;
}
```

Comme vous pouvez le voir, la variable **count** commence à 0 et est incrémentée chaque fois que le corps de la boucle s'exécute. Une fois la boucle exécutée 10 fois, l'expression devient fausse (c'est-à-dire la variables **count** n'est plus inférieur à 10), l'instruction **while** se termine et l'interpréteur peut passer à l'instruction suivante dans le programme. De nombreuses boucles ont une variable compteur comme **count**. Les noms de variables i, j et k sont couramment utilisés comme des compteurs de boucles, bien que vous deviez utiliser des noms plus descriptifs si cela facilite la compréhension de votre code.

1.3.3.2. do/while

La boucle **do/while** est comme une boucle **while**, sauf que l'expression de boucle est testée en bas de la boucle plutôt qu'en haut. Cela signifie que le corps de la boucle est toujours exécuté au moins une fois. La syntaxe est:

```
do {
        // instructions
} while (expression);
```

La boucle **do/while** est , dans la pratique, moins utilisée que **while**, il est assez rare d'être certain que vous voulez qu'une boucle s'exécute au moins une fois. Voici un exemple de boucle **do/while**:

```
function printArray(a) {
        var len = a.length, i = 0;
         if (len == 0){
                console.log("Empty Array");
         else {
                do {
                        console.log(a[i]);
                } while (++i < len);
        }
}
```

1.3.3.3. for

L'instruction **for** fournit une construction en boucle qui est souvent plus pratique que l'instruction **while**. L'instruction **for** simplifie les boucles qui suivent un modèle commun. La

plupart des boucles ont une variable de compteur quelconque. Cette variable est initialisée avant le démarrage de la boucle et est testée avant chaque itération de la boucle. Enfin, la variable compteur est incrémentée ou mise à jour à la fin du corps de la boucle, juste avant que la variable ne soit à nouveau testée. Dans ce type de boucle, l'initialisation, le test et la mise à jour sont les trois manipulations cruciales d'une variable de boucle. L'instruction for code chacune de ces trois manipulations comme une expression et fait de ces expressions une partie explicite de la syntaxe de la boucle:

```
for(initialisation ; test ; incrémentation) {
        // instructions
}
```

initialisation, **test** et **incrémentation** sont trois expressions (séparées par des points-virgules) qui sont responsables de l'initialisation, du test et de l'incrémentation de la variable de boucle. En les mettant tous dans la première ligne de la boucle, il est facile de comprendre ce que fait la boucle **for** et d'éviter les erreurs telles que l'oubli d'initialiser ou d'incrémenter la variable de boucle.

La manière la plus simple d'expliquer le fonctionnement d'une boucle **for** est de montrer la boucle **while** équivalente :

```
initialisation;
while(test) {
        //instructions
        incrémenter;
}
```

En d'autres termes, l'expression **initialisation** est évaluée une fois avant le début de la boucle. Pour être utile, cette expression est généralement une affectation. JavaScript autorise également l'initialisation à être une déclaration de variable **var** afin que vous puissiez déclarer et initialiser un compteur de boucles en même temps. L'expression de **test** est évaluée avant chaque itération et contrôle si le corps de la boucle doit être exécuter. Si le **test** évalué est vrai, **l'instruction** qui est le corps de la boucle est exécutée. Finalement, l'expression **d'incrémentation** est évaluée. Généralement, il s'agit soit d'une expression d'affectation, soit de l'utilisation des opérateurs ++ ou --.

Nous pouvons afficher les nombres de 0 à 9 avec une boucle **for** comme suit. Comparez-le avec l'équivalent avec celle de la boucle **while** de la section précédente:

```
for(var count = 0; count < 10; count++){
        console.log(count);
}
```

Les boucles peuvent devenir beaucoup plus complexes que cet exemple simple, bien sûr, et parfois plusieurs variables changent à chaque itération de la boucle. Cette situation est le seul cas où l'opérateur de virgule est couramment utilisé en JavaScript; il fournit un moyen de combiner plusieurs expressions d'initialisation et d'incrément dans une seule expression pouvant être utilisée dans une boucle for:

```
var i,j;
for(i = 0, j = 10 ; i < 10 ; i++, j--){
        sum += i * j;
        console.log(sum) ;
}
```

L'instruction **for/in** utilise le mot clé for, mais il s'agit d'un type de boucle complètement différent de la boucle **for** habituelle. Une boucle **for/in** ressemble à ceci:

```
for (variable in object) {
        statement
}
```

Pour exécuter une instruction for/in, l'interpréteur JavaScript évalue d'abord l'expression de l'objet. Si elle est évaluée à null ou indéfinie, l'interpréteur ignore la boucle et passe à l'instruction suivante.

L'instruction **for/in** boucle à travers les propriétés d'un objet.

Le bloc de code à l'intérieur de la boucle sera exécuté une fois pour chaque propriété.

```
var person = {fname:"John", lname:"Doe", age:25};

var text = "";
var x;
for (x in person) {
   text += person[x] + " ";
}
```

Le résultat de text sera: John Doe 25.