JavaScript

10 minutes pour comprendre

Core JavaScript

Client-side JavaScript

Applications

http://campus.ec-lyon.fr/options/tic/js.pdf

JavaScript

10 minutes pour comprendre

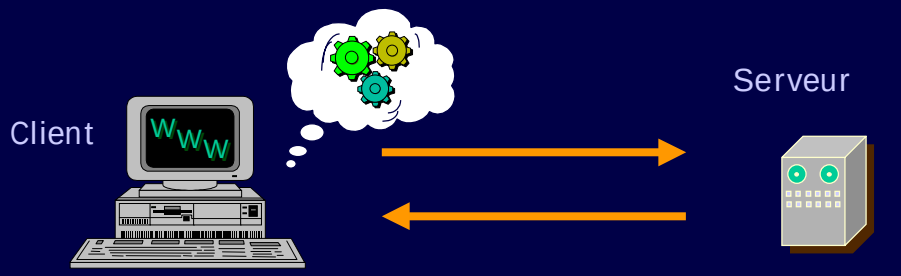
Qu'est-ce que JavaScript

JavaScript est un langage de programmation de scripts.

Un script est un programme dont le code source est inclus dans un document HTML. Ce programme est interprété et s'exécute sur la machine du client lorsque le document est chargé ou lors d'une action de l'utilisateur *(clic ou déplacement du curseur par exemple)*.

Les principes, l'implémentation, le fonctionnement, les domaines d'application de JavaScript n'ont strictement rien à voir avec ceux de Java. Seule une éventuelle ressemblance superficielle au niveau de la syntaxe et la volonté de profiter d'un effet de mode relatif à Java ont valu à ce langage de s'appeler JavaScript.

Fonctionnement de JavaScript



Le client demande un document HTML au serveur.

Le serveur envoie le document au client

Le document contient un script.

Le client interprète le script

```
<SCRIPT LANGUAGE="JavaScript">
…
</SCRIPT>
```

Exemple de programme JavaScript

```
<HEAD>
<SCRIPT LANGUAGE="JavaScript"><!--
function maj() {
      document.write("Dernière mise à jour : "
            + document.lastModified);
}
//-->
</SCRIPT>
</HEAD>
<BODY>
Ceci est un essai
<P>
<SCRIPT><!--
maj();
//-->
</SCRIPT>
<NOSCRIPT>Dommage !</NOSCRIPT>
```

Possibilités de JavaScript

Contrairement à Java, JavaScript est complètement intégré au navigateur. Un programme JavaScript a accès à des informations internes au navigateur *(version, plugins…)*, et à tous les éléments de la page HTML courante *(cf. DOM)*.

Les principales applications de JavaScript sont :

- la validation de formulaires avant envoi au serveur,
- la détection de type et de version de navigateur,
- la gestion de fenêtres *(pop-ups, cadres…)*,
- la préservation d'informations contextuelles,
- la génération automatique de date,
- la détection de plug-in,
- les calculs côté client,
- les effets d'images...

Versions JavaScript

JavaScript a été inventé par Netscape et implémenté à partir de Navigator 2.0.

Bien entendu (!) JavaScript a évolué au fur et à mesure des versions de navigateurs. La correspondance entre les versions du langage et les versions des navigateurs est rappelée ci-dessous :

Version	Navigator	Explorer
`JavaScript 1.0`	Navigator 2.0	I. Explorer 3.0
`JavaScript 1.1`	Navigator 3.0	-
`JavaScript 1.2`	Navigator 4.0	I. Explorer 4.0
`JavaScript 1.3`	Navigator 4.06	I. Explorer 5.0

JavaScript

Core JavaScript

Environnements JavaScript

Core JavaScript *(le noyau de JavaScript)* est un langage de script orienté objet qui, comme tout langage, reconnaît un certain nombre d'opérateurs, de structures de contrôle et d'instructions. Certains objets tels Array, Date ou Math font partie intégrante du langage, mais il est possible de compléter le noyau de base par des objets supplémentaires.

- Client-side JavaScript est une implémentation qui complète les éléments de base par des objets permettant de contrôler un navigateur *(Netscape ou autre)* et l'apparence du document courant.
- Server-side JavaScript ajoute des objets pertinents pour une exécution côté serveur, permettant par exemple de communiquer avec des bases de données, de gérer des sessions ou d'accéder aux fichiers du serveur *(Essentiellement disponible sur serveurs Netscape).*

Environnements JavaScript

Le principe qui consiste à étendre Core JavaScript avec des fonctionnalités spécifiques peut être généralisé pour la programmation de tout type de système.

C'est sur cette base que JavaScript *(Core JavaScript)* a été standardisé par l'ECMA *(European Computer Manufacturers Association)* sous l'appellation ECMAScript :

ECMA-262 = ISO-616262 = JavaScript 1.3 (cf N 4.06, IE5)

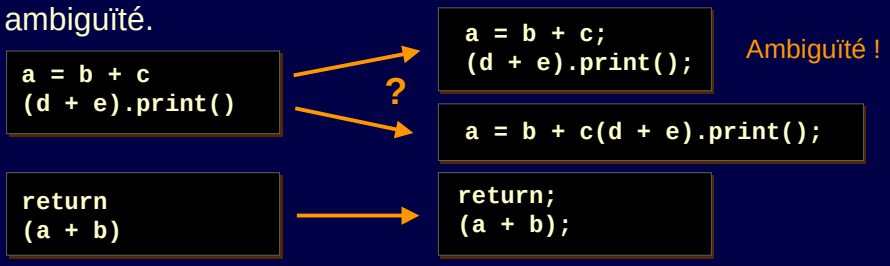
Le standard ECMAScript a été adopté en août 1998.
Les navigateurs implémentant JavaScript 1.3 datent de 1999.

Syntaxe

A priori, une instruction JavaScript se termine toujours par un point-virgule :

```
pi = 3.141592;
```

Toutefois, l'interpréteur est très permissif, et il fait partie des spécifications d'ECMAScript d'accepter l'absence de point-virgule dans tous les cas où cela ne conduit pas à une ambiguïté.



Un conseil : toujours finir une instruction par un point-virgule.

Commentaires

En JavaScript les commentaires peuvent prendre deux formes correspondant à la syntaxe C ou C++.

```
/*
** initialisation de tableau
*/
var centralien = {}; // déclaration du tableau
centralien.nom = "Deubaze";
centralien.prenom = "Raymond";
centralien.promo = 2000;

// initialisation d'objet

var centralienne = {
      prenom:"Raymonde", nom:"Deubaze", promo:2000};
```

Types de données

JavaScript accepte les types de données suivants :

- des nombres *(cf. 1, 2, 3, 42, 3.141592 …)*
- des valeurs logiques , (true ou false)
- des chaînes *(cf. "Hello !", 'Ca marche aussi', "aujourd'hui"…)*
- null
- undefined

Les conversions de type sont automatiques :

```
"Pi vaut : " + 3.141592 + " (ou à peu près)"
```

résulte bien en la chaîne escomptée, mais ceci ne fonctionne que pour l'opérateur +, ainsi :

"64" - 4 donne 60, mais
"64" + 4 donne "644" ...

null et undefined valent false
null vaut 0

Variables

Les variables peuvent être déclarées explicitement :

```
var pi = 3.141592;
```

ou implicitement lors de leur première utilisation :

```
pi = 3.141592;
```

L'évaluation *(i.e. à droite du signe =)* d'une variable non initialisée produit une erreur si celle-ci n'est pas déclarée, undefined ou NaN si elle l'a été *(suivant le contexte)* …

La portée d'une variable déclarée en dehors du corps d'une fonction est globale, et disponible dans l'ensemble du document courant. Si elle est déclarée dans le corps d'une fonction elle est locale à cette fonction. Les variables locales doivent être déclarées de manière explicite.

Valeurs littérales

Nombres entiers : *314, 0xbebe, 011 (en octal, vaut 9)*

Nombres à virgule flottante : *1, 1., 1.0, -3.2e5, .1e10*

Booléens : true, false

Chaînes : *"bla", 'bla', "J'ai dit \"non !\"",*

caractères spéciaux :

\b	backspace		
\f	form feed	\'	apostrophe simple
\n	new line	\"	apostrophe double
\r	carriage return	*\040*	code octal (ici espace)
\t	tab	*\0x20*	code hexadecimal (ici espace)

Tableaux

Un tableau peut être initialisé avec une liste de valeurs *(éventuellement vide)* spécifiées entre crochets []. Par exemple :

```
jours = [ "Dimanche", "Lundi", "Mardi",
          "Mercredi", "Jeudi", "Vendredi", "Samedi" ];
```

initialise un tableau de 7 éléments *(jours.length = 7).*

L' indice du premier élément est 0 :

jours[0] vaut "Dimanche" jours[1] vaut "Lundi" ...

N.B. Un tableau est un objet du type Array.

Opérateurs de comparaison

Ces opérateurs renvoient une valeur true ou false. Les opérandes peuvent être des nombres ou des chaînes.

Les nombres sont comparés numériquement, les chaînes alphabétiquement.

a == b	a égale b ?
a != b	a différent de b ?
a === b	a strictement égal à b ? *(opérandes égaux et du même type)*
a !== b	a non strictement égal à b ? *(vrai si valeur ou type différents)*
a > b	a supérieur à b ?
a >= b	a supérieur ou égal à b ?
a < b	a inférieur à b ?
a <= b	a inférieur ou égal à b ?

Opérateurs arithmétiques

Les quatre opérateurs fonctionnent de manière classique :

+	addition,
-	soustraction,
*	multiplication
/	division *(toujours à virgule flottante, jamais entière)*

Les autres opérateurs sont :

%	modulo, deux arguments, reste de la division entière
++	incrément, un argument, préfixe ou postfixe
--	décrément, un argument, préfixe ou postfixe
-	opposé, un argument, préfixe

Opérateurs bit à bit

Les opérateurs bit à bit considèrent leurs opérandes comme des mots de 32 bits, mais retournent des valeurs numériques standard :

&	ET logique, bit à bit
|	OU logique, bit à bit
^	OU exclusif, bit à bit
~	complément bit à bit
<<	décalage à gauche (on rentre des zéros à droite)
>>	décalage à droite (bit de signe conservé)
>>>	décalage à droite (on rentre des zéros à gauche)

Exemple : *3 << 2* donne *12*,

Opérateurs logiques

Les opérateurs logiques portent sur des variables booléennes. Toutefois leur évaluation est inhabituelle :

a && b	ET logique, (a == false) ? a : b
a || b	OU logique, (a == true) ? a : b
! a	NON logique, (a == true) ? false : true

N.B. == s'interprète de fait comme "peut être converti en".

Par conséquent, si les opérateurs && et || sont appliqués à des variables booléennes, ils retournent un booléen conforme au résultat attendu. S'ils sont appliqués à des variables non booléennes le résultat est non booléen :

par exemple : *1 && "Hello"* vaut *"Hello"*

Opérateurs logiques

L'évaluation des expressions logiques est interrompue dès que la valeur du résultat est connu. Dans l'expression ci-dessous, la fonction *essai( )* n'est pas appelée :

```
art = false;
x = art && essai();
```

De même, dans la situation suivante la fonction *la_vie ( )* n'est pas appelée :

```
la_bourse = true;
y = la_bourse || la_vie();
```

Ce comportement est garanti. Il fait partie des spécifications du langage.

Autres opérateurs

S'il est utilisé dans un contexte où l'un au moins des opérandes est une chaîne, + est interprété comme étant l'opérateur de concaténation :

`"Pi vaut : " + 3.141592 + " (ou à peu près)"` est une chaîne.

? est le seul opérateur JavaScript à admettre trois arguments :

condition ? val1 : val2

vaut *val1* si *condition* est vraie *(true)*, *val2* sinon.

L'opérateur , *(virgule)* évalue ses deux opérandes et retourne le second *(principalement utilisé avec l'instruction de boucle for)*.

Exemples : `i=1, j=2` `for(i=1, j=2; i++, j++; i<10)`

L'opérateur typeof

L'opérateur typeof renvoie le type de la variable ou de l'expression à laquelle il s'applique :

```
var pi = 3.141592;
var str = "Coucou";
var now = Date();

var check = "typeof pi : " + typeof(pi) + "\n"
      + "typeof str : " + typeof(str) + "\n"
      + "typeof now : " + typeof(now) + "\n"
      + "typeof foo : " + typeof(foo) + "\n";
```

Le type d'un objet non défini est *undefined*

Opérateurs d'affectation

Un opérateur d'affectation affecte la valeur de l'opérande situé à sa droite *(généralement une expression)* à l'opérande situé à sa gauche *(généralement une variable)*. L'opérateur d'affectation le plus simple est = :

x = *y* affecte la valeur de *y* à la variable *x*.

Les autres opérateurs d'affectation sont en fait des raccourcis pour quelques opérations standards :

x += *y*	*x* = *x* + *y*	*x* %= *y*	*x* = *x* % *y*	*x* &= *y*	*x* = *x* & *y*
x -= *y*	*x* = *x* - *y*	*x* <<= *y*	*x* = *x* << *y*	*x* |= *y*	*x* = *x* | *y*
x *= *y*	*x* = *x* * *y*	*x* >>= *y*	*x* = *x* >> *y*	*x* ^= *y*	*x* = *x* ^ *y*
x /= *y*	*x* = *x* / *y*	*x* >>>= *y*	*x* = *x* >>> *y*		

Précédence des opérateurs

La précédence des opérateurs donne l'ordre de priorité dans lequel ils sont évalués. Cet ordre permet d'écrire des expressions comme :

```
x = a + b * c - d == 23 ? 1 : ++j , ++i;
```

et d'obtenir le résultat escompté, qui est : ??

```
(x = (((a+(b*c)-d) == 23) ? 1 : ++j)), ++i;
```

Un conseil : il est plus sûr, plus rapide et plus efficace de spécifier l'ordre de priorité des opérations à effectuer à l'aide de parenthèses bien placées *(même si celles-ci sont éventuellement inutiles)* que de consulter la documentation relative à la précédence des opérateurs et d'accoucher d'une séquence de code illisible et non maintenable ...

Instructions conditionnelles

Une instruction conditionnelle permet de spécifier une séquence d'instructions qui ne s'exécute que si une certaine condition est remplie.

```
if (condition ) {
      instructions;
}
else {
      instructions;
}
```

Exemple :

```
if ( s  % 2 ) {
  alert("seconde impaire : "+s);
}
else {
  alert("seconde paire : "+s);
}
```

Il est bien entendu possible de faire directement suivre l'instruction *else* par une autre instruction *if*, pour obtenir une construction *if - else if - else* classique.

Instruction conditionnelle (2)

La seconde instruction conditionnelle reconnue par JavaScript est l'instruction switch. Cette instruction permet de tester un éventail de valeurs plus important que simplement true / false.

```
switch(expression){
   case valeur_1 :
      instructions;
      break;
   case valeur_2 :
      instructions;
      break;
   default :
      instructions;
 }
```

Exemple :

```
switch(Math.floor(s/10)) {
  case 0 :
    alert("tout petit : "+s );
    break;
  case 1 :
    alert("petit : " + s );
    break;
  case 2 :
    …
}
```


Boucles

Une boucle est une séquence d'instructions qui s'exécute de manière répétitive jusqu'à ce qu'une certaine condition soit remplie.

```
do {
      instructions;
} while (condition);
```

Exemple :

```
do {
      time = new Date();
      s = time.getSeconds();
} while ( s % 10 );
alert("nouvelle dizaine : "+s);
```

Cette boucle est parcourue une fois, à la suite de quoi la condition est évaluée.

La boucle est alors exécutée à nouveau jusqu'à ce que la condition ne soit plus remplie. Si la condition est initialement fausse, la boucle est parcourue une fois et une seule. *(preuve)*

Boucles (2)

Une boucle basée sur l'instruction while n'est pas exécutée si la condition est initialement fausse.

```
while (condition){
      instructions;
}
```

Exemple :

```
      while ( n < 5 ) {
            alert("n = " + n++);
      }
```

La boucle est parcourue tant que la condition est remplie.

La condition est évaluée avant toute exécution des instructions de la boucle. Si la condition est initialement fausse, la boucle n'est pas parcourue. *(preuve)*

Boucles (3)

L'instruction for permet de spécifier, en plus de la condition, une instruction d'initialisation et une instruction de boucle. La boucle n'est pas exécutée si la condition est initialement fausse.

Exemple :

```
for(init; condition; repeat){
      instructions;
}
```

```
for( n = 1; n < 5; n++ ) {
     alert("n = " + n);
}
```

L'instruction for ci-dessus est sémantiquement équivalente à la boucle :

```
init;
while(condition){
      instructions;
      repeat;
}
```

Cette instruction est très souvent utilisée,comme dans l'exemple mentionné, dès lors où l'on gère un compteur de boucle.

Boucles (4)

L'instruction break permet d'interrompre prématurément l'exécution d'une boucle.

```
while(true){
      instructions;
      if (condition) break;
      instructions;
}
```

Exemple : *(quelle est la dernière valeur de n affichée ?)*

```
while (true) {
      alert("n = " + n++);
      if ( n == 5 ) break;
      alert("On continue");
}
```

L'instruction continue permet d'interrompre le déroulement du corps d'une boucle, en exécutant directement l'itération suivante.

```
for ( n = 0; n < 10; n++ ){
      alert("n = " + n++);
      if ( n % 2 ) continue;
      alert("n pair");
}
```


Boucles (5)

Dans le cas de boucles imbriquées il est possible de les nommer à l'aide d'un label qui permet de spécifier celle à laquelle s'appliquent les instructions break ou continue.

```
label: while(true){
      instructions;
      if (condition) break label;
      instructions;
}
```

Exemple :

```
loop_i: for (i=1;  i < 5; i++) {
      loop_j: for (j=1;  j < 5; j++) {
            alert("(i,j) = (" + i + "," + j + ")");
            if ( j == i ) continue loop_i;
      }
}
```

Fonctions

En JavaScript la définition d'une fonction se fait à l'aide de l'instruction function. L'appel se fait de manière classique.

Depuis le corps d'une fonction il est possible d'accéder à la liste des arguments à l'aide du tableau arguments[]. Le premier argument a pour indice 0.

Une fonction peut renvoyer une valeur à l'aide de l'instruction return.

JavaScript possède un certain nombre de fonctions prédéfinies :

- eval
- isFinite
- isNaN
- parseInt, parseFloat
- Number, String
- escape, unescape

Tableaux associatifs

Comme dans d'autres langages *(cf. perl)* il est possible en JavaScript d'indexer des tables non pas avec un entier mais avec une chaîne de caractères *(tableaux associatifs)*.

Ainsi, la séquence :

```
ascendant[0] = "Raymond";
ascendant[1] = "Marcelle";
ascendant[2] = "Raymond";
ascendant[3] = "Ginette";
ascendant[4] = "Marcel";
ascendant[5] = "Raymonde";
```

Gagnerait en lisibilité à être codée :

```
ascendant["père"] = "Raymond";
ascendant["mère"] = "Marcelle";
ascendant["grand'père paternel"] = "Raymond";
ascendant["grand'mère paternelle"] = "Ginette";
ascendant["grand'père maternel"] = "Marcel";
ascendant["grand'mère maternelle"] = "Raymonde";
```

Objets

Si la valeur de l'index ne comprend que des caractères alphanumériques *(cf. nom de variable)*, la notation précédente peut être simplifiée :

```
ascendant["p"] = "Raymond";
ascendant["m"] = "Marcelle";
ascendant["pp"] = "Raymond";
ascendant["pm"] = "Ginette";
ascendant["mp"] = "Marcel";
ascendant["mm"] = "Raymonde";
```

```
ascendant.p = "Raymond";
ascendant.m = "Marcelle";
ascendant.pp = "Raymond";
ascendant.pm = "Ginette";
ascendant.mp = "Marcel";
ascendant.mm = "Raymonde";
```

En JavaScript, un objet est un tableau associatif :

```
centralien.prenom = "Raymond";
centralien.nom = "Deubaze";
centralien.promo = 2000;
```

Cette séquence crée un objet *centralien* possédant les trois propriétés *prenom*, *nom* et *promo*.

Initialisation d'un objet

Une variable du type objet peut être initialisée lors de sa déclaration :

```
var centralien = {prenom:"Raymond", nom:"Deubaze", promo:2000};
```

En général, on définit une fonction appelée constructeur, permettant de créer des instances d'objets similaires.

```
function Centralien(prenom,nom,promo) {
      this.prenom = prenom;
      this.nom = nom;
      this.promo = promo;
}
var moniteur_H10 = new Centralien("Raymond","Deubaze",2000);
var moniteur_F7 = new Centralien("Raymonde","Deubaze",2001);
```

Noter la variable this qui se réfère à l'objet que l'on est en train d'initialiser

Prototype

Il est possible d'initialiser ou de modifier une propriété pour l'ensemble des objets d'un type donné *(en C++ on dirait pour toutes les instances d'une classe)*.

```
function Centralien(prenom,nom,promo) {
	this.prenom = prenom;
	this.nom = nom;
	this.promo = promo;
}
var moniteur_H10 = new Centralien("Raymond","Deubaze",2000);
var moniteur_F7 = new Centralien("Raymonde","Deubaze",2001);

Centralien.prototype.option = "TIC";
```

La dernière instruction ajoute et initialise la propriété option aux deux objets du type Centralien : moniteur_H10 et moniteur_F7.

Méthodes

Lorsqu'une propriété d'un objet est une fonction, on appelle cette fonction une méthode de l'objet considéré. La définition d'une méthode peut être intégrée au constructeur.

```
function Centralien(prenom,nom,promo) {
	...
	this.print = list_props;
}
function list_props() {
	var str = ""; var prop;
	for ( prop in this ) {
		str += prop + " = " + this[prop] + "\n";
	}
	return str;
}
```

Noter encore la variable this qui se réfère à l'objet appelant.

Opérateurs objet

JavaScript possède des opérateurs qui s'appliquent plus particulièrement à des objets : new *(création d'un objet)*, this *(se réfère à l'objet appelant dans une méthode)*, et delete qui permet de supprimer un objet.

L'opérateur delete peut s'appliquer à toutes les variables déclarées implicitement. Ceci recouvre en particulier une propriété ou un élément de tableau mais aussi une variable ou un objet déclarés implicitement *(sans utiliser l'opérateur var)*.

delete renvoie *true* si l'opération s'est bien passée, *false* sinon.

Instructions objet

Il existe également des instructions qui ne s'appliquent qu'à des objets. La première est for.. in. Elle permet d'effectuer une boucle sur toutes les propriétés d'un objet.

```
function list_props(obj) {
       var str = ""; var prop;
       for ( prop in obj ) {
              str += prop + " = " + obj[prop] + "\n";
       }
       return str;
}
```

L'instruction with permet de travailler dans le contexte d'un objet particulier *(confort d'écriture)*.

```
x = r * Math.sin(Math.PI / 2);
```

```
with ( Math ) {
       x = r * sin(PI / 2);
}
```

Objets prédéfinis

Boolean

constructeur : *Boolean(valeur)*

méthode : *valueOf( )*

Attention : un objet du type Boolean, non *null* et non *undefined* sera toujours converti à la valeur *true* dans une expression logique :

```
var mybool = new Boolean(false);

if ( mybool ) {
       alert("mybool is true");
}
alert("mybool's value is " + mybool.ValueOf());
```

Sauf cas particulier, il vaut mieux utiliser une variable booléenne qu'un objet.

Number

Number

constructeur : *Number(valeur)*

méthodes : *valueOf( ), toString( )*

Comme dans le cas de l'objet Boolean, il est peu courant de créer des objets du type Number. Toutefois, le prototype de l'objet Number possède des propriétés intéressantes *(en C++ on parlerait de variables de classe)* :

Number.MAX_VALUE	: le plus grand nombre
Number.MIN_VALUE	: le plus petit nombre
Number.NaN	: le code NaN (Not a Number)
Number.NEGATIVE_INFINITY	: l'infini positif
Number.POSITIVE_INFINITY	: l'infini négatif

Pour le détail de ces codes, cf. double précision IEEE 754 *(64 bits).*