Syntaxe JavaScript

La syntaxe de JavaScript est un ensemble de règles qui définissent ce qui constitue un programme valide en langage JavaScript.

⚠ **Attention :** Les exemples présentés dans cet article utilisent souvent une fonction appelée `println`. Ce terme représente la fonction de sortie standard qui n'existe pas dans la bibliothèque standard de Javascript. Dans un navigateur, ce terme peut être remplacé par `document.write`.

Origine

Brendan Eich a résumé le lignage de la syntaxe dans le premier paragraphe JavaScript 1.1 [1] des spécifications ainsi : “JavaScript emprunte la plupart de sa syntaxe à Java, mais hérite aussi d'Awk et Perl, avec une influence indirecte de Self pour son système de prototype objet.”

Variables

Les variables en JavaScript n'ont pas de type défini, et n'importe quelle valeur peut être stockée dans n'importe quelle variable. Les variables peuvent être déclarées avec `var` . Ces variables ont une portée lexicale et une fois que la variable est déclarée, on peut y accéder depuis la fonction où elle a été déclarée. Les variables déclarées en dehors d'une fonction et les variables utilisées sans avoir été déclarées en utilisant 'var', sont globales (peuvent être utilisées par tout le programme).

Voici un exemple de déclaration de variables et de valeurs globales :

```
x = 0; // Une variable globale
var y = 'Hello!'; // Une autre variable globale

function f(){
  var z = 'foxes'; // Une variable locale
  twenty = 20; // Globale car le mot-clef var n'est pas utilisé
  return x; // Nous pouvons utiliser x ici car il s'agit d'une variable
 globale
}
// La valeur de z n'est plus accessible à partir d'ici
```

Types de données de base

Nombres

Les nombres en JavaScript sont représentés en binaire comme des IEEE-754 Doubles, ce qui permet une précision de 14 à 15 chiffres significatifs JavaScript FAQ 4.2 [2] **(en)**. Comme ce sont des nombres binaires, ils ne représentent pas toujours exactement les nombres décimaux, en particulier les fractions.

Ceci pose problème quand on formate des nombres pour les afficher car JavaScript n'a pas de méthode native pour le faire. Par exemple:

```
alert(0.94 - 0.01); // affiche 0.9299999999999999
```

En conséquence, l'arrondi devrait être utilisé dès qu'un nombre est formaté pour l'affichage [3] **(en)**. La méthode toFixed() ne fait pas partie des spécifications de l'ECMAScript et est implémentée différemment selon l'environnement, elle ne peut donc être invoquée.

Les nombres peuvent être spécifiés dans l'une de ces notations :

```
345;    // un "entier", bien qu'il n'y ait qu'un seul type numérique en
 JavaScript
34.5;   // un nombre flottant
3.45e2; // un autre nombre flottant, équivalent à 345
0377;   // un entier octal égal à 255
0xFF;   // un entier hexadecimal égal à 255, les lettres A-F peuvent
être en minuscules ou en majuscules
```

Dans certaines implémentations de l'ECMAScript comme l'ActionScript, les couleurs sont parfois spécifiées avec des nombres entiers en écriture hexadécimale :

```
var colorful = new Color( '_root.shapes' );
colorful.setRGB( 0x003366 );
```

Le constructeur Number peut être utilisé pour réaliser une conversion numérique explicite :

```
var myString = "123.456"
var myNumber = Number( myString );
```

Quand il est utilisé comme un constructeur, un objet *wrapper* numérique est créé (toutefois il est peu employé) :

```
myNumericWrapper = new Number( 123.456 );
```

Tableaux

Un tableau est un ensemble d'éléments repérés par leur indice, qui est un nombre entier. En JavaScript, tous les objets peuvent être formés d'un ensemble d'éléments, mais les tableaux sont des objets spéciaux qui disposent de méthodes spécifiques (par exemple, `join`, `slice`, et `push`).

Les tableaux ont une propriété `length` qui représente la longueur du tableau, c'est-à-dire le nombre d'éléments qu'il peut contenir. La propriété length d'un tableau est toujours supérieure à l'indice maximal utilisé dans ce tableau (N.B. Les indices de tableaux sont numérotés à partir de zéro). Si on crée un élément de tableau avec un indice supérieur à `length`, `length` est automatiquement augmentée. Inversement, si on diminue la valeur de `length`, cela supprime automatiquement les éléments qui ont un indice supérieur ou égal. La propriété `length` est la seule caractéristique qui distingue les tableaux des autres objets.

Les éléments de tableau peuvent être accédés avec la notation normale d'accès aux propriétés d'objet :

```
monTableau[1];
monTableau["1"];
```

Les 2 notations ci-dessus sont équivalentes.

Par contre, il n'est pas possible d'utiliser la notation "point" : objet.propriété, ni d'autres représentations du nombre entier de l'indice :

```
monTableau.1;       // déclenche une erreur de syntaxe
monTableau["01"];   // n'est pas identique à monTableau[1]
```

La déclaration d'un tableau peut se faire littéralement ou par l'utilisation du constructeur `Array` :

```
monTableau = [0,1,,,4,5];             // crée un tableau de longueur 6
avec 4 éléments
monTableau = new Array(0,1,2,3,4,5);  // crée un tableau de longueur 6
avec 6 éléments
monTableau = new Array(365);          // crée un tableau vide de
longueur 365
```

Les tableaux sont implémentés de façon à ce que seuls les éléments utilisés utilisent de la mémoire ; ce sont des tableaux *sporadiques*. Si l'on crée un tableau monTableau de longueur 58, puis qu'on attribue `monTableau[10] = 'uneValeur'` et `monTableau[57] = 'uneAutreValeur'`, l'espace occupé correspondra seulement à ces 2 éléments, comme pour tous les objets. La propriété `length` du tableau restera quand même à 58.

Grâce à la déclaration littérale, on peut créer des objets similaires aux tableaux associatifs d'autres languages :

```
chien = {"couleur":"brun", "taille":"grand"};
chien["couleur"];           // rend la valeur "brun"
```

On peut mélanger les déclarations littérales d'objets et de tableaux pour créer facilement des tableaux associatifs, multidimensionnels, ou les deux à la fois :

```
chats = [{"couleur":"gris", "taille":"grand"},
         {"couleur":"noir", "taille":"petit"}];
chats[0]["taille"];         // rend la valeur "grand"
chiens = {"azor":{"couleur":"brun", "taille":"grand"},
          "milou":{"couleur":"blanc", "taille":"petit"}};
chiens["milou"]["taille"];  // rend la valeur "petit"
```

Chaînes de caractères

En Javascript la chaîne de caractères est considérée comme une suite de caractères. Une chaîne de caractères en JavaScript peut être directement créée en plaçant des caractères entre quotes (doubles ou simples) :

```
var salutation = "Hello, world!";
var salut_martien = 'Bonjour, amis terriens';
```

Dans les navigateurs basés sur Mozilla, les caractères d'une chaîne peuvent être accédés individuellement (ils sont considérés comme des chaînes d'un seul caractère) avec la même notation que pour les tableaux :

```
var a = salutation[0];        // a vaut 'H' - dans un navigateur basé
sur Mozilla seulement
```

Dans Internet Explorer, il est nécessaire d'utiliser la méthode `charAt()` issue de la classe `String`. C'est la technique recommandée étant donnée qu'elle fonctionne aussi sur les navigateurs issus de Mozilla :

```
var a = salutation.charAt(0); // a vaut 'H' - fonctionne avec Internet
Explorer
                              //              et avec les navigateurs
basés sur Mozilla
```

En JavaScript les chaînes sont immuables :

```
salutation[0] = "H";      // erreur
```

L'opérateur d'égalité == permet de comparer le contenu de deux chaînes de caractères (en tenant compte de la casse), et retourne un booléen :

```
var x = "world";
var compare1 = ("Hello, " + x == "Hello, world"); // compare1 vaut true
var compare2 = ("Hello, " + x == "hello, world"); // compare2 vaut
false car les
                                                  // premiers
caractères des arguments
                                                  // n'ont pas la même
casse
```

Opérateurs

L'opérateur '+' est surchargé; il est utilisé pour la concaténation de chaîne de caractères et l'addition ainsi que la conversion de chaînes de caractères en nombres. Il a également un rôle spécial lorsqu'il est utilisé dans une expression régulière.

```
// Concatène 2 chaînes
var a = 'Ceci';
var b = ' et cela';
alert(a + b);  // affiche 'Ceci et cela'

// Additionne deux nombres
var x = 2;
var y = 6;
alert(x + y); // affiche 8

// Concatène une chaîne et un nombre
alert( x + '2'); // affiche 22

// Convertit une chaîne en nombre
var z = '4';   // z est une chaîne (le caractère 4)
alert( z + x); // affiche 42
alert( +z + x);// affiche 6
```

Arithmétiques

Opérateurs binaires

```
+     Addition
-     Soustraction
*     Multiplication
/     Division (rend une valeur en virgule flottante)
%     Modulo (rend le reste de la division entière)
```

Opérateurs unaires

```
-     Négation unaire (inverse le signe)
++    Incrémentation (peut être utilisé en forme préfixée ou postfixée)
--    Décrémentation (peut être utilisé en forme préfixée ou postfixée)
```

Affectations

```
=     Affectation
+=    Ajoute et affecte
-=    Soustrait et affecte
*=    Multiplie et affecte
/=    Divise et affecte
```

```
var x = 1;
x *= 3;
document.write( x );  // affiche: 3
x /= 3;
```

```
document.write( x );  // affiche: 1
x -= 1;
document.write( x );  // affiche: 0
```

Comparaisons

```
==    Égal à
!=    Différent de
>     Supérieur à
>=    Supérieur ou égal à
<     Inférieur à
<=    Inférieur ou égal à

===   Identiques (égaux et du même type)
!==   Non-identiques
```

Booléens

Le langage Javascript possède 3 opérateurs booléens :

```
&&    and (opérateur logique ET)
||    or (opérateur logique OU)
!     not (opérateur logique NON)
```

Dans une opération booléenne, toutes les valeurs sont évaluées à `true` (VRAI), à l'exception de :

- la valeur booléenne `false` (FAUX) elle-même
- le nombre 0
- une chaîne de caractères de longueur 0
- une des valeurs suivantes : *null undefined NaN*

On peut faire explicitement la conversion d'une valeur quelconque en valeur booléenne par la fonction `Boolean` :

```
Boolean( false );     // rend false (FAUX)
Boolean( 0 );         // rend false (FAUX)
Boolean( 0.0 );       // rend false (FAUX)
Boolean( "" );        // rend false (FAUX)
Boolean( null );      // rend false (FAUX)
Boolean( undefined ); // rend false (FAUX)
Boolean( NaN );       // rend false (FAUX)
Boolean ( "false" );  // rend true (VRAI)
Boolean ( "0" );      // rend true (VRAI)
```

L'opérateur booléen unaire NOT *!* évalue d'abord la valeur booléenne de son opérande, puis rend la valeur booléenne opposée :

```
var a = 0;
var b = 9;
!a; // rend true, car Boolean( a ) rend false
!b; // rend false, car Boolean( b ) rend true
```

En doublant l'opérateur *!*, on peut *normaliser* une valeur booléenne :

```
var arg = null;
arg = !!arg; // arg est désormais à false au lieu de null

arg = "ChaîneQuelconqueNonVide";
arg = !!arg; // arg est désormais à true
```

Dans les premières implémentations de Javascript et JScript, les opérateurs booléens *&&* et || fonctionnaient comme leurs homologues dans d'autres langages issus du C, c'est-à-dire qu'ils rendaient toujours une valeur booléenne :

```
x && y; // rend true si x ET y est VRAI, sinon il rend false
x || y; // rend true si x OU y est VRAI, sinon il rend false
```

Dans les implantations récentes, les opérateurs booléens *&&* et || rendent un de leurs deux opérandes :

```
x && y; // rend x si x est FAUX, sinon il rend y
x || y; // rend x si x est VRAI, sinon il rend y
```

Ce fonctionnement provient du fait que l'évaluation s'effectue de gauche à droite. Dès que la réponse est déterminée, l'évaluation s'arrête :

- x && y est nécessairement FAUX si x est FAUX, ce n'est pas la peine d'évaluer y
- x || y est nécessairement VRAI si x est VRAI, ce n'est pas la peine d'évaluer y

Ce nouveau comportement est peu connu, même parmi les développeurs Javascript expérimentés, et peut causer des problèmes si on s'attend à recevoir une valeur booléenne.

Binaires

Opérateurs binaires

```
&     And (et)
|     Or (ou inclusif)
^     Xor (ou exclusif)

<<    Shift left  (décalage à gauche avec remplissage à droite
avec 0)
>>    Shift right avec propagation du signe (le bit de gauche,
celui du signe, est décalé vers la droite)
>>>   Shift right (décalage à droite avec remplissage à gauche
 avec 0)

      Pour les nombres positifs, >> et >>> donnent le
même résultat
```

Opérateur unaire

```
~     Not (inverse tous les bits)
```

Chaînes de caractères

```
=     Assignation
+     Concaténation
+=    Concatène et assigne
```

```
str = "ab" + "cd";   // rend "abcd"
str += "e";          // rend "abcde"
```

Structures de contrôle

Si Sinon

```
 if (expression1)
 {
   //instructions réalisées si expression1 est vraie;
 }
 else if (expression2)
 {
   //instructions réalisées si expression1 est fausse et expression2
est vraie;
 }
 else
 {
   //instructions réalisées dans les autres cas;
 }
```

Dans chaque structure *if..else*, la branche *else* :

- est facultative
- peut contenir une autre structure *if..else*

Opérateur conditionnel

L'opérateur conditionnel crée une expression qui, en fonction d'une condition donnée, prend la valeur de l'une ou l'autre de 2 expressions données. Son fonctionnement est similaire à celui de la structure *si..sinon* qui, en fonction d'une condition donnée, exécute l'une ou l'autre de 2 instructions données. En ce sens on peut dire que l'opérateur conditionnel est aux expressions ce que la structure *si..sinon* est aux instructions.

```
 var resultat = (condition) ? expression1 : expression2;
```

Le code ci-dessus, qui utilise l'opérateur conditionnel, a le même effet que le code ci-dessous, qui utilise une structure *si..sinon* :

```
 if (condition)
 {
   resultat = expression1;
 }
 else
 {
   resultat = expression2;
```

```
}
```

Contrairement à la structure *si..sinon*, la partie correspondant à la branche *sinon* est obligatoire.

```
var resultat = (condition) ? expression1; // incorrect car il manque
la valeur à prendre si condition est fausse
```

Instruction switch

L'instruction switch remplace une série de *si..sinon* pour tester les nombreuses valeurs que peut prendre une expression :

```
switch (expression) {
  case UneValeur:
    // instructions réalisées si expression=UneValeur;
    break;
  case UneAutreValeur:
    // instructions réalisées si expression=UneAutreValeur;
    break;
  default:
    // instructions réalisées dans les autres cas;
    break;
}
```

- l'instruction `break;` termine le bloc `case` (le *cas* à traiter). Elle est facultative dans la syntaxe Javascript. Cependant, elle est recommandée, car en son absence l'exécution continuerait dans le bloc `case` suivant. Dans le dernier bloc `case` d'un switch l'instruction `break;` ne sert à rien, sauf à prévoir l'éventualité où on rajouterait plus tard un bloc `case` supplémentaire.
- la valeur du `case` peut être une chaîne de caractères
- les accolades { } sont obligatoires

Boucle For

Syntaxe pour plusieurs lignes d'instructions :

```
for (initialisation;condition;instruction de boucle) {
  /*
     instructions exécutées à chaque passage dans la boucle
     tant que la condition est vérifiée
  */
}
```

Syntaxe pour une seule instruction :

```
for (initialisation;condition;instruction)
```

Boucle For..in

```
for (var Propriete in Objet) {
  //instructions utilisant Objet[Propriete];
}
```

- L'itération s'effectue sur toutes les propriétés qui peuvent être associées à un compteur
- *Les avis divergent sur la possibilité d'utiliser cette boucle avec un tableau*[4] .
- L'ensemble des propriétés balayées dans la boucle diffère selon les navigateurs. Théoriquement, cela dépend d'une propriété interne appelée "DontEnum" définie dans le standard ECMAscript, mais dans la pratique chaque navigateur renvoie un ensemble différent.

Boucle While

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

Boucle Do..while

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

Boucle With

```
with(Objet) {
  var a = getElementById('a');
  var b = getElementById('b');
  var c = getElementById('c');
};
```

- on n'écrit pas `Objet.` avant chaque `getElementById()`

Fonctions

Une fonction est un bloc d'instructions avec une liste de paramètres (éventuellement vide). Elle possède généralement un nom et peut renvoyer une valeur.

```
function nom-fonction(argument1, argument2, argument3) {
  instructions;
  return expression;
}
```

Syntaxe pour l'appel d'une fonction anonyme, c'est-à-dire sans nom :

```
var fn = function(arg1, arg2) {
  instructions;
  return expression;
};
```

Exemple de fonction : l'algorithme d'Euclide. Il permet de trouver le plus grand commun diviseur de deux nombres, à l'aide d'une solution géométrique qui soustrait le segment le plus court du plus long :

```
function gcd(segmentA, segmentB) {
  while (segmentA != segmentB) {
    if (segmentA > segmentB)
      segmentA -= segmentB;
    else
      segmentB -= segmentA;
  }
  return segmentA;
}
println(gcd(60, 40)); // imprime 20
```

- le nombre des paramètres passés à l'appel de la fonction n'est pas obligatoirement égal au nombre des paramètres nommés dans la définition de la fonction
- si un paramètre nommé dans la définition de la fonction n'a pas de paramètre correspondant dans l'appel de la fonction, il reçoit la valeur `undefined`
- à l'intérieur de la fonction, les paramètres peuvent être accédés par l'objet `arguments`. Cet objet donne accès à tous les paramètres en utilisant un indice : `arguments[0]`, `arguments[1]`, `... arguments[n]`, y compris ceux qui sont au-delà des paramètres nommés. Attention, `arguments` n'est *pas* un tableau. Il a une propriété length mais il n'a pas les méthodes des tableaux comme slice, sort, etc.
- tous les paramètres sont passés par valeur, sauf pour les objets qui sont passés par référence.

```
var Objet1 = {a:1}
var Objet2 = {b:2}
function bizarre(p) {
  p = Objet2;          // on ignore la valeur reçue dans le paramètre p
  p.b = arguments[1]   // on remplit Objet2.b avec la valeur du 2ème
paramètre reçu
}
bizarre(Objet1, 3)     // Objet1 est ignoré, 3 est un paramètre
additionnel
println(Objet1.a + " " + Objet2.b); // imprime 1 3
```

On peut déclarer une fonction à l'intérieur d'une autre. La fonction ainsi créée pourra accéder aux variables locales de la fonction dans laquelle elle a été définie. Elle se souviendra de ces variables, même après qu'on soit sorti de la fonction dans laquelle elle a été définie, ce qui constitue une fermeture.

```
var parfum = "vanille"
var glace
function dessert() {
  var parfum = "chocolat"
  glace = function() {println(parfum)}
}
dessert()         // imprime "chocolat" parce qu'on est dans dessert
```

```
glace()           // imprime "chocolat" et pas "vanille", même si on
est sorti de dessert
```

Objets

Par commodité, on sépare habituellement les types de données en *primitives* (ou types de données de base) et *objets*. Les objets sont des entités qui ont une identité (il ne sont égaux qu'à eux-mêmes) et qui associent des noms de propriété à des valeurs. Les propriétés sont appelées *slots* dans le vocabulaire de la programmation orientée prototype.

Le langage Javascript propose divers types d'objets prédéfinis : tableau, booléen, date, fonction, maths, nombre, objet, expression régulière et chaîne de caractères. D'autres objets prédéfinis sont des objets *hôte* issus de l'environnement d'exécution. Par exemple dans un navigateur les objets hôtes appartiennent au modèle DOM : fenêtre, document, lien hypertexte, etc.

On peut voir les objets les plus simples comme des dictionnaires. En tant que dictionnaires, ils contiennent des couples clé-valeur, où la clé est une chaîne de caractères et la valeur peut être de n'importe quel type. Les objets simples sont souvent implémentés comme des dictionnaires. Cependant, les objets possèdent des fonctionnalités que n'ont pas les dictionnaires, comme une chaîne prototype.

Exemple de déclaration littérale d'un objet contenant des valeurs :

```
var monObjet = {nom: 'vistemboir', utilite: 'cet objet ne sert à rien',
 prix: 5000};
```

Les propriétés des objets peuvent être créées, lues et écrites, soit avec la notation "point" objet.propriété, soit avec la syntaxe utilisée pour les éléments de tableau :

```
var etiquette = monObjet.nom;    // etiquette contient désormais
'vistemboir'
var vente = monObjet['prix'];    // vente contient désormais 5000
```

Création

Les objets peuvent être créés à l'aide d'un constructeur ou de façon littérale. Le constructeur peut utiliser la fonction prédéfinie Object ou une fonction sur mesure. Par convention les noms des fonctions constructeurs commencent par une majuscule.

```
// création à l'aide du constructeur prédéfini :
var premierObjet = new Object();
// création littérale :
var objetA = {};
var monObjet = {nom: 'vistemboir', utilite: 'cet objet ne sert à rien',
 prix: 5000};
```

Les déclarations littérales d'objets et de tableaux permettent de créer facilement des structures de données adaptées aux besoins :

```
var maStructure = {
  nom: {
     prenom: "Sylvain",
     patronyme: "Etiré"
```

```
  },
  age: 33,
  distractions: [ "oenologie", "beuveries" ]
};
```

C'est la base de JSON, un format d'échanges de données utilisant une syntaxe inspirée des objets Javascript.

Constructeur

Les constructeurs assignent simplement des valeurs aux slots de l'objet nouvellement créé. Les valeurs assignées peuvent être elles-mêmes d'autres fonctions.

Contrairement à ce qui se passe dans d'autres langages à objets, en Javascript les objets n'ont pas de type. Le constructeur qui sert à créer l'objet n'est pas mémorisé. C'est simplement une fonction qui remplit les slots et le prototype du nouvel objet. Donc le constructeur ne correspond pas à la notion de classe qu'on trouve dans d'autres langages.

Les fonctions sont elles-mêmes des objets, ce qui peut servir à obtenir le même fonctionnement que les "static properties" des langages C++ et Java (cf exemple ci-dessous).

Les fonctions ont une propriété spécifique appelée `prototype` (cf paragraphe Héritage ci-dessous).

La destruction d'objet est rarement pratiquée car elle est peu nécessaire. En effet le ramasse-miettes détruit automatiquement les objets qui ne sont plus référencés.

Exemple : manipulation d'objet

```
function MonObjet(attributA, attributB) {
  this.attributA = attributA;
  this.attributB = attributB;
}

MonObjet.statiqueC = " bleu";   // ajoute à la fonction une propriété
statique avec une valeur
println(MonObjet.statiqueC);    // imprime bleu

objet = new MonObjet(' rouge', 1000);

println(objet.attributA);      // imprime rouge
println(objet["attributB"]);   // imprime 1000

println(objet.statiqueC);      // imprime undefined

objet.attributC = new Date();  // ajoute à l'objet une nouvelle
propriété avec une valeur

delete objet.attributB;        // enlève une propriété à l'objet
println(objet.attributB);      // imprime undefined

delete objet;                  // supprime l'objet entier (rarement
```

```
utilisé)
println(objet.attributA);      // déclenche une exception
```

Méthodes

Une méthode est simplement une fonction qui est assignée à la valeur d'un slot d'objet. Contrairement à ce qui se passe dans de nombreux langages à objets, dans Javascript il n'y a pas de distinction entre la définition d'une fonction et la définition d'une méthode. La différence peut apparaître lors de l'appel de la fonction : une fonction peut être appelée comme une méthode.

Dans l'exemple ci-dessous, la fonction Gloup assigne simplement des valeurs aux slots. Certaines de ces valeurs sont des fonctions. De cette façon Gloup peut assigner des fonctions différentes au même slot de différentes instances du même objet. N.B. Il n'y a pas de prototypage dans cet exemple.

```
function y2() {return this.xxx + "2 ";}

function Gloup(xz) {
  this.xxx = "yyy-";
  if (xz > 0)
    this.xx = function() {return this.xxx +"X ";};
  else
    this.xx = function() {return this.xxx +"Z ";};
  this.yy = y2;
}

var gloup-un = new Gloup(1);
var gloup-zero = new Gloup(0);

gloup-zero.xxx = "aaa-";

println(gloup-un.xx() + gloup-zero.xx());              // imprime yyy-X
 aaa-Z

gloup-un.y3 = y2;           // assigne à y3 la fonction y2 elle-même et
 non pas son résultat y2()

var bloub={"xxx": "zzz-"}   // création d'un objet sans utilisation de
constructeur
bloub.y4 = y2               // assigne à y4 la fonction y2 elle-même et
 non pas son résultat y2()

println(gloup-un.yy() + gloup-un.y3() + bloub.y4());  // imprime yyy-2
yyy-2 zzz-2

gloup-un.y2();              // déclenche une exception car gloup-un.y2
n'existe pas
```

Héritage

Javascript supporte les hiérarchies d'héritage par prototypage de la même façon que le langage self. Chaque objet contient un slot implicite appelé prototype.

Dans l'exemple ci-dessous, la classe Bagnole hérite de la classe Vehicule. Quand on crée l'instance b de la classe Bagnole, une référence à l'instance de base de la classe Vehicule est copiée dans b. La classe Bagnole ne contient pas de valeur pour la fonction roule. *Quand la fonction roule est appelée* cette fonction est recherchée dans l'arborescence d'héritage et elle est trouvée dans la classe Vehicule. Cela se voit clairement dans l'exemple : si on change la valeur de vehicule.roule, on retrouve cette valeur dans b.roule.

Les implémentations de Javascript basées sur Mozilla permettent d'accéder explicitement au prototype d'un objet grâce à un slot appelé _proto_ comme dans l'exemple ci-dessous.

```
function Vehicule() {
  this.roues = function() {println("nombre pair de roues");};
  this.roule = function() {println("ça roule");};
}

function Bagnole() {
  this.roues = function() {println("quatre roues");};
// surcharge de la fonction roues pour la classe Bagnole
}

vehicule = new Vehicule();
Bagnole.prototype = vehicule;    // doit être exécuté avant
d'instancier Bagnole

b = new Bagnole();               // crée b ce qui copie vehicule dans
le slot prototype de b

vehicule.roule = function() {println("ça roule sur terrain plat")}
// surcharge de la fonction roule pour l'objet vehicule

b.roues();                                    // imprime quatre roues
b.roule();                                    // imprime ça roule sur
terrain plat
println(b.roule == Bagnole.prototype.roule); // imprime true
println(b.__proto__ == vehicule);             // imprime true seulement
avec Mozilla
```

L'exemple suivant montre clairement que les références aux prototypes sont copiées lors de la création de l'instance et que les changements appliqués au prototype se répercutent dans toutes les instances qui s'y réfèrent.

```
function m1() {return "un ";}
function m2() {return "deux ";}
function m3() {return "trois ";}

function Base() {}
```

```
Base.prototype.y = m2;
alpha = new Base();
println(alpha.y());                    // imprime deux

function Special(){this.y = m3}
special = new Special();

beta = new Base();
Base.prototype = special;              // n'a pas d'effet sur alpha
et beta qui sont déjà créés
println(beta.y());                     // imprime deux

gamma = new Base();                    // gamma est créé avec special
 dans son slot prototype
println(gamma.y());                    // imprime trois

special.y = m1;                        // impacte gamma et ses
éventuelles classes dérivées
println(gamma.y());                    // imprime un
```

En pratique de nombreuses variations d'héritage sont utilisées, ce qui peut être très puissant mais aussi prêter à confusion.

Exceptions

A partir de Internet Explorer 5 et Netscape 6, les implémentations de Javascript comportent une instruction `try ... catch ... finally` pour la gestion des exceptions, c'est-à-dire les erreurs en cours d'exécution. Cette instruction traite les exceptions provoquées par une erreur ou par une instruction de levée d'exception.

La syntaxe est la suivante :

```
try {
  // instructions pouvant déclencher une exception
} catch(erreur) {
  // instructions à exécuter en cas d'exception
} finally {
  // instructions à exécuter dans tous les cas
}
```

D'abord, les instructions du bloc `try` s'exécutent.

- si pendant l'exécution du bloc `try` une exception est déclenchée, l'exécution passe au bloc `catch` et le code de l'exception est passé dans le paramètre. A la fin du bloc `catch`, l'exécution continue dans le bloc `finally`
- si aucune exception ne se déclenche, le bloc `catch` est ignoré et l'exécution passe au bloc `finally`

Le bloc `finally` sert souvent à libérer la mémoire, pour éviter qu'elle ne reste bloquée lors d'une erreur fatale, même si ce souci ne se pose guère dans Javascript.

Exemple :

```
try {
  tablo = new Array();                // création d'un tableau
  fonctionRisquee(tablo);             // appel d'une fonction qui peut
ne pas fonctionner
}
catch (...) {
  logError();                         // traitement des erreurs
éventuelles
}
finally {
  delete tablo;                       // libération de la mémoire
occupée par tablo même en cas d'erreur fatale
}
```

Le bloc `finally` est facultatif :

```
try {
  // instructions
}
catch (erreur) {
  // instructions
}
```

Le bloc `catch` est lui aussi facultatif. Dans ce cas, si une exception se déclenche, l'exécution quitte le bloc `try` et passe au bloc `finally` sans que l'erreur soit traitée.

```
try {
  // instructions
}
finally {
  // instructions
}
```

Attention : il est obligatoire d'avoir au moins l'un des deux blocs `catch` ou `finally`, ils ne peuvent pas être absents tous les deux.

```
try { instruction; }                              // erreur
```

Si vous utilisez le bloc `catch`, le paramètre est obligatoire, même si vous ne l'utilisez pas dans le bloc.

```
try { instruction; } catch() { instruction; }    // erreur
```

Sous Mozilla il est permis d'avoir plusieurs instructions `catch`. Il s'agit d'une extension au standard ECMAScript. Dans ce cas, la syntaxe est similaire à celle de Java :

```
try { instruction; }
catch ( e if e == "InvalidNameException"  ) { instruction; }
catch ( e if e == "InvalidIdException"    ) { instruction; }
catch ( e if e == "InvalidEmailException" ) { instruction; }
catch ( e ) { instruction; }
```

Divers

Casse

Javascript est sensible à la casse.

L'usage est de donner aux objets un nom qui commence par une majuscule et de donner aux fonctions ou variables un nom qui commence par une minuscule.

Espaces vides et points-virgules

Dans le langage Javascript les instructions se terminent par un point-virgule.

Cependant Javascript comporte un mécanisme d'insertion automatique de point-virgule : lors de l'analyse d'une ligne de code qui ne se termine pas par un point-virgule, si le contenu de la ligne correspond à une instruction correcte, la ligne peut être traitée comme si elle se terminait par un point-virgule.

Les caractères espace, tabulation, fin de ligne et leurs variantes, quand ils ne sont pas inclus dans une chaîne de caractères, sont désignés sous le terme général *whitespace*. Les caractères whitespace peuvent avoir un effet sur le code à cause du mécanisme d'insertion automatique de point-virgule.

Pour éviter les effets non désirés dûs au mécanisme d'insertion automatique de point-virgule, il est conseillé d'ajouter systématiquement un point-virgule à la fin de chaque instruction, même si cela diminue la lisibilité du code.

Exemple d'effet non désiré :

```
return
a + b;
//   ces 2 lignes renvoient undefined car elles sont traitées comme :
//   return;
//   a + b;
```

Autre exemple :

```
a = b + c
(d + e).fonc()
//   pas de point-virgule ajouté, les lignes sont traitées comme :
//   a = b + c(d + e).fonc()
```

D'autre part, les caractères whitespace accroissent inutilement la taille du programme et donc du fichier .js. La solution la plus simple à ce problème est de faire comprimer les fichiers par le serveur, ce qui réduira la taille de tous les fichiers source téléchargés sur le serveur. La compression en zip fonctionne mieux que les logiciels spécialisés en suppression de whitespace (*whitespace parser*).

Commentaires

La syntaxe des commentaires est la même qu'en C++.

```
// commentaire

/* commentaire
   multiligne */
```

Il est interdit d'imbriquer les commentaires :

```
/* cette syntaxe
   /* est interdite */
*/
```

Voir aussi

- JavaScript

Notes

[1] http://hepunx.rl.ac.uk/~adye/jsspec11/intro.htm#1006028
[2] http://www.jibbering.com/faq/#FAQ4_2
[3] http://www.jibbering.com/faq/#FAQ4_1
[4] La question est de savoir si l'itération se fera non seulement sur l'indice du tableau, mais aussi sur d'autres propriétés :
- Un article du site Microsoft Developer Network (http://msdn.microsoft.com/en-us/library/kw1tezhk(VS.85).aspx) assure que la boucle `For...In` peut être utilisée pour itérer sur tous les éléments d'un tableau. L'article du MSDN se réfère à JScript, qui est utilisé par Internet Explorer pour les scripts en JavaScript.
- Le site W3Schools (http://www.w3schools.com/js/js_loop_for_in.asp) donne les tableaux comme exemple d'utilisation de la boucle `For...In`.
- Un article du Mozilla Developer Centre (https://developer.mozilla.org/en/Core_JavaScript_1.5_Guide/Object_Manipulation_Statements#for…in_Statement) explique le problème : il est tentant d'utiliser la boucle `For...In` pour balayer les éléments d'un tableau, mais il faut savoir qu'elle balaie aussi les propriétés définies par l'utilisateur. Si vous modifiez l'objet `Array` en y ajoutant des propriétés ou des méthodes, la boucle `For...In` rendra le nom de vos propriétés en plus des indices numériques. De ce fait, pour boucler dans un tableau, il vaut mieux utiliser une boucle `For` classique avec un compteur numérique.

References

- **(en)** Cet article est partiellement ou en totalité issu d'une traduction de l'article de Wikipédia en anglais intitulé « *JavaScript_syntax* (http://en.wikipedia.org/wiki/JavaScript_syntax) ».
- David Flanagan, Paula Ferguson: *JavaScript: The Definitive Guide*, O'Reilly & Associates, ISBN 0-596-10199-6
- Danny Goodman, Brendan Eich: *JavaScript Bible*, Wiley, John & Sons, ISBN 0-7645-3342-8
- Thomas A. Powell, Fritz Schneider: *JavaScript: The Complete Reference*, McGraw-Hill Companies, ISBN 0-07-219127-9
- Emily Vander Veer: *JavaScript For Dummies, 4th Edition*, Wiley, ISBN 0-7645-7659-3

Liens externes

Reference Material

- Core References for JavaScript versions 1.5 (http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference), 1.4 (http://research.nihonsoft.org/javascript/CoreReferenceJS14/), 1.3 (http://research.nihonsoft.org/javascript/ClientReferenceJS13/) and 1.2 (http://research.nihonsoft.org/javascript/jsref/)

Ressources

- Mozilla JavaScript Language Documentation (http://developer.mozilla.org/en/docs/JavaScript)
- A re-introduction to JavaScript - Mozilla Developer Center (http://developer.mozilla.org/en/docs/A_re-introduction_to_JavaScript)

Sources et contributeurs de l'article

Syntaxe JavaScript *Source*: http://fr.wikipedia.org/windex.php?oldid=41822592 *Contributeurs*: Aproche, Clarus, DVD88, Emegamanu, Joeldumonteil, Koko90, Laddo, Manu1400, Michel.plomteux, NicoV, PicMirandole, Pok148, Poss Jean-Louis, Pymouss44, Seb35, Sebleouf, Shawn, Sigma 7, Silex6, Spartakus-fr, Stanlekub, Tanguy Ortolo, 34 modifications anonymes

Source des images, licences et contributeurs

image:Nuvola apps important.svg *Source*: http://fr.wikipedia.org/windex.php?title=Fichier:Nuvola_apps_important.svg *Licence*: GNU Lesser General Public License *Contributeurs*: User:Bastique

Licence