Le langage ECMAScript

Table des matières

Introduction

JavaScript est un langage de programmation utilisé principalement sur le Web, développé par Netscape et repris par Microsoft sous le nom de JScript. ECMAScript est une tentative de normalisation du noyau du langage : sa syntaxe, ses mots-clés et ses composants natifs. La troisième édition du standard ECMA-262 a été publiée en décembre 1999.

Les commentaires

Avant d'aborder la syntaxe du langage, voyons où nous pourrons l'enfreindre à loisir… Les commentaires vous permettent en effet de loger votre prose presque n'importe où dans un script, sans en modifier le comportement lors de l'exécution. Il existe deux types de commentaires en ECMAScript ; dans le premier cas, le commentaire commence par les caractères // et se termine à la fin de la ligne ; dans le second, il débute avec les caractères /*, peut se prolonger sur plusieurs lignes et s'achève dès que les caractères */ sont rencontrés par l'interpréteur.

```
// Un premier commentaire sur une ligne

/* Un deuxième commentaire */

/* Un troisième commentaire sur plusieurs lignes */
```

Les commentaires ne s'imbriquent pas l'un dans l'autre ; les exemples suivants provoquent une erreur de syntaxe :

```
/* ce texte est compté /* dans le commentaire */ mais pas celui-ci */

/* dans // le commentaire */ hors du commentaire
```

Les variables

Une variable est un emplacement réservé en mémoire, désigné par un nom particulier, qui permet de stocker puis de réutiliser des données. Une variable est créée avec le mot-clé var suivi du nom qui doit lui être donné et d'un point-virgule, qui termine toute instruction ECMAScript.

```
var réponse;
```

Vous pouvez donner n'importe quel nom à vos variables, tant que ce dernier respecte certaines règles : il doit commencer par une lettre ou par les caractères $ ou _, et peut se poursuivre avec des chiffres, mais pas avec des espaces ni des sauts de lignes. Il doit également être unique, c'est-à-dire que deux variables différentes ne peuvent pas porter le même nom, et il

est sensible à la casse des caractères. De plus vous ne pouvez utiliser ni les mots-clés, que nous verrons par la suite, ni les mots réservés pour nommer vos variables.

Mots réservés par ECMAScript :

abstract ; boolean ; byte ; char ; class ; const ; debugger ; double ; enum ; export ; extends ;	final ; float ; goto ; implements ; import ; int ; interface ; long ; native ; package ;	private ; protected ; public ; short ; static ; super ; synchronized throws ; transient ; volatile.

Il existe cinq types de données primitifs en ECMAScript. Les deux premiers sont quelque peu différents des autres : ce sont les types Undefined et Nullqui n'admettent d'une seule valeur, respectivement undefined et null. Toute variable qui n'a pas encore reçu de valeur spécifique vaut undefined alors que la valeur null peut, selon le contexte, représenter une variable explicitement vide ou une fonction non gérée. Les trois autres types de données, Boolean, Number et String, admettent en revanche plusieurs valeurs différentes. Le type Boolean, qui représente les nombres booléens, accepte en effet exactement deux valeurs qui s'excluent mutuellement : true(vrai) et false (faux).

```
var bonne_réponse;

bonne_réponse = true; /* Un booléen */
```

Notez que vous pouvez rejoindre les deux instructions en une seule autre équivalente :

```
var bonne_réponse = true;
```

Le type Number permet de manipuler les nombres à virgule flottante, en valeur approchée, comme décrits dans la norme IEEE 754-1985. Le point ici agit en séparateur des parties entière et décimale. Ce type possède quelques valeurs notables, comme Infinity pour l'infini positif, NaN pour désigner ce qui n'est pas un nombre (le résultat d'une division par zéro par exemple), ou un zéro négatif, que l'on saura utiliser si nécessaire et ignorer dans les autres cas.

```
var taux_tva = 0.196;

var superficie_de_la_terre = 134.288e6; // 134 288 000 km²
```

```
/* Le 'e6' (exposant 6) signifie "10 puissance 6" et ne s'emploie
que dans ce cas précis, où il définit un nombre */
var députés = 0x241; /* Le nombre de députés à l'Assemblée Nationale, en
hexadécimal comme l'indique le préfixe '0x' */
var univers = Infinity; /* L'univers est infini, tout le monde le sait :-) */
var zéro_négatif = -0; /* Oui, un zéro négatif, distinct de +0 */
```

Enfin le type String permet d'utiliser des chaînes de caractères, dont la longueur n'est, a priori, pas limitée.

```
var question = "Quelle est la Réponse ?"
```

Dans l'exemple précédent, nous avons créé une chaîne de caractères qui contient le texte Quelle est la Réponse ? ; les guillemets servent uniquement à délimiter la chaîne et n'en font pas partie. Cependant certains caractères ne peuvent pas apparaître tels quels quand vous créez la chaîne : il s'agit notamment des sauts de ligne et des guillemets eux-mêmes. Vous devez alors utiliser une séquence d'échappement : c'est un groupe de caractères automatiquement remplacé par le caractère, unique, qu'il symbolise. Les guillemets sont échappés avec la séquence \" et le saut de ligne avec \n.

```
var dialogue = "Il a dit : \"…\"\nJ'ai répondu : \"…\"";
```

La variable dialogue contient le texte suivant :

```
Il a dit : "…"
J'ai répondu : "…"
```

Cet exemple pourrait cependant être simplifié en utilisant des apostrophes (ou guillements simples) à la place des guillements doubles pour délimiter la chaîne, le résultat étant totalement identique. Cette solution a le mérite d'être plus lisible mais vous ne pouvez plus insérer d'apostrophes dans le texte sans les échapper.

```
var dialogue = 'Il a dit : "…"\nJ\'ai répondu : "…"';
```

Il existe quelques autres séquences d'échappement simples, c'est-à-dire ne comportant qu'un seul caractère après \, mais ECMAScript permet surtout d'insérer n'importe quel caractère de l'Unicode dans une chaîne en l'échappant sous la forme \uXXXX où XXXX est son code hexadécimal dans la table Unicode.

```
var ego = "\u0046\u006c\u006f\u0072\u0069\u0061\u006e";
```

Voici un moyen particulièrement pervers pour simplement écrire mon prénom… Le tableau suivant présente les séquences d'échappement simples définies par le standard ECMA-262.

Séquences d'échappement ECMAScript

Séquence d'échappement	Séquence Unicode	Nom	Symbole	Échappement obligatoire ?
\b	\u0008	backspace	<BS>	Non
\t	\u0009	tabulation horizontale	<HT>	Non
\n	\u000A	fin de ligne	<LF>	Oui
\v	\u000B	tabulation verticale	<VT>	Non
\f	\u000C	envoi de formulaire	<FF>	Non
\r	\u000D	retour charriot	<CR>	Oui
\"	\u0022	guillemets	"	Parfois
\'	\u0027	apostrophe	'	Parfois
\\	\u005C	anti-slash	\	Oui

Signalons enfin qu'ECMAScript est amené à réaliser automatiquement des conversions quand elles sont nécessaires, dans des situations où un booléen est attendu et qu'un nombre est donné par exemple. Les tableaux suivants, extraits de la section 9 du standard ECMA-262, présentent les règles utilisées pour de telles conversions.

Conversion vers le type Boolean

Type d'origine	Résultat
Undefined	false
Null	false
Boolean	Le résultat est identique à la valeur d'origine (pas de conversion).
Number	Le résultat vaut false si le nombre vaut +0, -0 ou NaN ; sinon il vaut true.
String	Le résultat est false si la chaîne est vide (de longueur nulle) ; sinon le résultat est true.

Conversion vers le type Number

Type d'origine	Résultat
Undefined	NaN
Null	+0
Boolean	Le résultat est 1 si le booléen vaut true, et +0 si le booléen vaut false.
Number	Le résultat est identique à la valeur d'origine (pas de conversion).
String	Le langage tente d'interpréter la chaîne en suivant la syntaxe des nombres.

Conversion vers le type String

Type d'origine	Résultat

Conversion vers le type String	
Type d'origine	**Résultat**
Undefined	"undefined"
Null	"null"
Boolean	Si le booléen vaut true, le résultat est "true". Si le booléen vautfalse, le résultat est "false".
Number	L'interpréteur donne une réprésentation du nombre tel qu'il aurait pu être écrit dans le code source du programme.
String	Le résultat est identique à la valeur d'origine (pas de conversion).

Les expressions

L'expression est un point fondamental du langage : presque toute instruction en contient, au moins, une. Une expression est une suite d'opérateurs et de valeurs opérandes. On dit qu'elle est évaluée quand l'interpréteur effectue les différentes opérations, en respectant les priorités des opérateurs, un peu à la manière d'un calcul algébrique.

```
3 + 4 * 7 / 2 - 9
```

Pour évaluer l'expression précédente, l'interpréteur va d'abord regrouper chaque opérande avec son opérateur, ce qui peut être symbolisé par l'ajout de parenthèses :

```
(3 + ((4 * (7 / 2)) - 9))
```

Notez que seul 2 est compté comme diviseur et non 2-9, car la division a une priorité plus importante que la soustraction. L'interpréteur effectue ensuite les calculs en partant de la parenthèse la plus intérieure.

```
(3 + ((4 * (7 / 2)) - 9)) // La division
(3 + ((4 * 3.5) - 9)) // La multiplication
(3 + (14 - 9))// La soustraction
(3 + 5) // L'addition
```

```
8
```

Quand toutes les opérations ont été effectuées, le résultat, ici 8, devient la valeur de l'expression tout entière. Ainsi les deux variables suivantes sont strictement égales :

```
var i = 3 + 4 * 7 / 2 - 9;

var j = 8;
```

Bien sûr l'intérêt des expressions réside dans le fait de pouvoir utiliser des variables comme opérandes, et non dans ce type de calculs où toutes les valeurs sont fixées.

Opérateurs mathématiques

Avec l'exemple précédent nous avons vu quatre opérateurs mathématiques courants : ce sont des opérateurs binaires, c'est-à-dire qui acceptent deux opérandes. De plus leur opération s'effectue avec deux nombres et renvoie une valeur du type Number. Chaque opérateur possède également une certaine priorité qui peut se traduire par un entier : plus ce dernier est grand, plus la priorité de l'opération est élevée. Nous pouvons ainsi commencer à remplir notre tableau des opérateurs du langage ECMAScript.

Opérateurs mathématiques binaires

Opérateur	Résultat	Priorité	Description
Number +Number	Number	12	Addition de deux nombres
Number - Number	Number	12	Soustraction
Number * Number	Number	13	Multiplication
Number / Number	Number	13	Division
Number %Number	Number	13	Reste de la division euclidienne (modulo)

Pour changer l'ordre d'évaluation des opérations, par exemple pour effectuer une addition avant une multiplication, utilisez simplement les parenthèses comme dans les exemples précédents pour regrouper les calculs concernés. Dans un de ces calculs, il peut arriver par exemple qu'un opérateur attende un nombre comme opérande, mais qu'il reçoive en fait un

booléen ou une chaîne de caractères. Dans ce cas, la valeur est convertie en nombre en suivant les règles de conversion du paragraphe précédent.

Parmi ces cinq opérateurs, il en existe un qui nécessite une attention particulière : c'est l'opérateur +, ou plutôt les opérateurs +, car il en a bien plusieurs. Le premier ne change pas, c'est toujours l'addition. Quant au second, c'est l'opérateur de concaténation de chaînes, qui met bout à bout deux chaînes de caractères. Tous deux sont binaires, mais l'opérateur de concaténation réclame qu'au moins un de ses arguments soit une valeur du type String. Si ce n'est pas le cas, alors c'est la simple addition qui est effectuée.

```
var i = "20" + 3;
```

Dans cet exemple, la variable **i** est une chaîne de caractères qui contient le texte 203, et non le nombre 23.

Il ne nous reste plus que deux opérateurs mathématiques, unaires pour leur part. Ce sont les opérateurs de signe + et -, mais ils ne peuvent pas être confondus avec leurs homologues binaires, car ces derniers sont toujours précédés d'un opérande, ce qui n'est pas le cas des opérateurs de signe. L'opérateur - convertit son opérande en nombre et retourne l'opposé de ce nombre, tandis que l'opérateur + le convertit uniquement en nombre.

Les opérateurs de concaténation et de signes

Opérateur	Résultat	Priorité	Description
String + String	String	13	Concaténation de chaînes
+ Number	Number	14	Conversion en nombre
- Number	Number	14	Opposé du nombre

Opérateurs d'affectation

Nous avons utilisé à plusieurs reprises jusqu'ici le symbole « = » en lui donnant implicitement sa signification mathématique. Mais c'est en fait (en doutiez-vous ?) l'opérateur d'affectation simple, qui se contente de prendre, sans aucune conversion et quel que soit son type, la valeur qui le suit pour la stocker dans la variable qui le précède. Mais n'avais-je pas laissé entendre que les variables étaient remplacées par leur valeur ? Sûrement, mais c'était un mensonge… Les noms de variables sont en fait des valeurs d'un type particulier, interne à l'interpréteur, et

que vous ne pourrez jamais manipuler directement dans vos scripts : c'est le type Reference. Un nom de variable est une référence, un lien ou encore un pointeur (oui, même en ECMAScript), vers la valeur de la variable. Lorsqu'ils rencontrent une Reference, tous les opérateurs vont automatiquement chercher la valeur associée, tous sauf les opérateurs d'affectation. Voilà pourquoi vous ne pourrez jamais écrire **3 = 2**, car 3 est du type Number et non Reference.

L'opérateur « = » possède plusieurs dérivés comme l'opérateur « += ». L'écriture **a += b** est strictement équivalente à **a = a + b**. Des opérateurs similaires existent pour les soustraction, multiplication, division et modulo.

ECMAScript possède enfin les traditionnels opérateurs unaires d'incrémentation, « ++ » et de décrémentation « -- ». L'opérateur « ++ » convertit son opérande en nombre et lui ajoute 1, alors que « -- » lui enlève 1. Ces deux opérateurs renvoient l'ancienne valeur de la variable.

```
var i = 8;

var j = i++; /* i vaut 9, j vaut 8 */
```

Opérateurs d'affectation et d'incrémentation

Opérateur	Résultat	Priorité	Description
Reference =(valeur)	(valeur)	2	Affectation simple
Reference +=Number	Number	2	Addition et affectation (similaire avec -=, *=, /= et %=)
Reference++	Number	15	Incrémentation
Reference--	Number	15	Décrémentation

Opérateurs relationnels

Les opérateurs relationnels sont des opérateurs qui servent à comparer deux nombres entre eux, pour déterminer les relations d'infériorité stricte avec « <», d'infériorité large « <= », de supériorité stricte « > », ou de supériorité large « >= ». Ces opérateurs binaires comparent

deux nombres et renvoient un booléen : ce dernier vaut true si la relation exprimée est vérifiée, et falsesinon. De plus il existe l'opérateur « == » (attention à bien doubler le signe) qui teste l'égalité de deux valeurs, après les avoir converties dans le même type. L'opérateur « === » teste également l'égalité, mais sans aucune conversion.

```
12 == 12; /* Vrai */

false == 0; /* Vrai, avec une conversion */

false === 0; /* Faux avec cet opérateur */

null == undefined; /* Vrai, par convention */

null === undefined; /* Faux */

"12" == 12 /* Vrai */

12 == "12" /* Évidemment vrai */

12 === "12" /* Faux, dans tous les sens :-) */
```

Ces deux opérateurs possèdent chacun une négation : « != » renvoie truequand « == » renvoyait false et inversement. Il en est de même avec « !== » et « === ».

Opérateurs relationnels

Opérateur	Résultat	Priorité	Description
Number > Number	Boolean	10	Strictement supérieur
Number >= Number	Boolean	10	Supérieur ou égal
Number < Number	Boolean	10	Strictement inférieur
Number <= Number	Boolean	10	Inférieur ou égal
(valeur) == (valeur)	Boolean	9	Égalité (large)
(valeur) != (valeur)	Boolean	9	Différence (large)
(valeur) === (valeur)	Boolean	9	Égalité (stricte)
(valeur) !== (valeur)	Boolean	9	Différence (stricte)

Opérateurs logiques

Les opérateurs logiques sont exclusivement réservés aux opérations entre booléens. Le premier d'entre eux l'opérateur, unaire, de négation « ! », qui renvoie true quand son opérande vaut false et inversement. Quant aux deux autres opérateurs, ils sont binaires et correspondent aux deux fonctions courantes « ET » et « OU ». Le « ET logique » est symbolisé par «&& », alors que le « OU logique » s'écrit « || ». Leurs comportements sont définis par les trois tableaux suivants.

ET logique

Opérande 1	Opérande 2	Résultat
false	false	false
true	false	false
false	true	false
true	true	true

OU logique

Opérande 1	Opérande 2	Résultat
false	false	false
true	false	true
false	true	true
true	true	true

Utilisation des opérateurs logiques

Opérateur	Résultat	Priorité	Description

Utilisation des opérateurs logiques

Opérateur	Résultat	Priorité	Description
!Boolean	Boolean	14	Négation logique
Boolean && Boolean	Boolean	5	ET logique
Boolean || Boolean	Boolean	4	OU logique

Contrôle de flux

Tout programme un peu évolué doit tôt ou tard prendre des décisions selon la valeur de certaines variables et savoir adapter son comportement. De même il doit avoir la capacité de répéter certaines actions ou d'interrompre son déroulement normal quand une erreur se produit : toutes ces fonctions sont gérées par des structures de contrôle de flux.

Savoir décider

Il existe deux structures dédiées à la prise de décision :if et switch. La structure if suit le schéma si-alors-sinon et s'utilise de la façon suivante :

```
if ( …expression à tester… )
{
 instructions si vrai
}
else {
 instructions si faux
}
```

L'expression entre parenthèses est d'abord évaluée, et son résultat est converti en booléen. S'il vaut true, le premier bloc d'instructions est exécuté et le second ignoré ; s'il vaut false, seul le second est exécuté. La structure peut cependant être réduite en enlevant le mot-clé else et le bloc qui le suit, s'il n'y a rien à faire quand la condition est fausse. Les accolades définissent un bloc d'instructions, qui est obligatoire avec toutes les structures de contrôle dès qu'il y a

plus d'une instruction à exécuter. Pour résumer : une seule instruction, les accolades sont facultatives ; plusieurs instructions, les accolades sont obligatoires.

La structure if ne permet de tester que deux possibilités, cependant il est possible d'imbriquer plusieurs if l'un dans l'autre pour effectuer des tests plus complets.

```
if (un_nombre == 1)
{
 // un_nombre vaut 1, faire quelque chose
}
else{
 if (un_nombre == 2){
   // un_nombre vaut 2, faire autre chose
 }
 else{
   // un_nombre ne vaut ni 1, ni 2, que faire ?
 }
}
```

Si cela fonctionne, il est pourtant déconseillé d'utiliser ainsi if, car la structure switch est plus adaptée à ce genre de tests. switch ne réagit pas selon la vérité logique d'une expression mais selon les différentes valeurs que cette dernière peut retourner. Voici comment elle s'utilise avec un exemple.

```
switch ( un_nombre )// Une expression quelconque
{ /* L'accolade, toujours obligatoire ici,
 il ne s'agit pas d'un bloc d'instructions */
 case 1: /* un_nombre vaut 1 */
   // des instructions sans accolades
   break;/* Le mot-clé break pour sortir de la structure */
 case 2:
 case 3:
   /* On exécute les mêmes instructions pour les valeurs 2 et 3
   car il n'y a pas eu de "break;" */
```

```
  break;
 default:/* Que faire quand c'est un imprévu ? */
  // Des instructions
  break;
}
```

Une structure switch fonctionne d'après un principe de « branchements » : selon la valeur de l'expression dans l'en-tête, l'interpréteur ira au casecorrespondant et exécutera toutes les instructions qui suivent, tant qu'aucun break n'est rencontré. Ainsi dans l'exemple précédent, les mêmes instructions sont exécutées pour les valeurs 2 et 3 puisqu'il n'y a aucunbreak avant la ligne case 3 :. Le branchement default permet enfin d'intercepter toutes les valeurs, sauf celles pour lesquelles il existe déjà un branchement case.

Travailler à la chaîne

ECMAScript possède trois structures pour mettre en place un mécanisme de bouclage dans un script. La première, la structure while, s'utilise ainsi :

```
while ( …expression à tester… )
{
 instructions à exécuter
}
```

Cette structure a un fonctionnement similaire à celui de if : l'interpréteur évalue l'expression entre parenthèses et exécute les instructions si cette dernière est vraie. A la fin du bloc d'instructions, l'expression est de nouveau testée et le bloc est de nouveau exécuté si elle est toujours vraie : pour éviter de créer une boucle infinie, il faut logiquement que l'expression à tester devienne fausse. L'exemple suivant donne un exemple d'une boucle infinie.

```
while (true)
{
 /* Une boucle infinie qui ne fait rien, mais elle sera probablement
 rapidement rejetée par les navigateurs */
}
```

Si au premier passage dans la structure while la condition est fausse, le contenu de la boucle ne sera jamais exécuté : c'est ce qui la différencie de la structure do while. Avec cette

structure, les instructions sont d'abord exécutées une première fois, puis la condition est testée pour savoir si l'on doit de nouveau exécuter le bloc.

```
do
{
 instructions à exécuter
}
while ( …expression à tester… ); /* Attention au point-virgule ici */
```

La dernière structure, for, peut se concevoir comme un while évolué, et elle est particulièrement adaptée au cas où un certain nombre d'itérations doit être réalisé. Voici un exemple d'une boucle for qui calcule la factorielle d'un nombre.

```
var nombre, factorielle;
nombre = 9;
factorielle = 1; /* Ne surtout pas oublier, pour la multiplication qui suit */
for (var i = 1; i <= nombre; i++)
{
 factorielle *= i;
}
```

L'en-tête de la boucle se compose de trois parties séparées par des points-virgules. La première est évaluée une seule fois à l'entrée de la boucle ; ici elle initialise une variable **i** qui servira à l'intérieur. Puis la seconde partie est évaluée : c'est la condition dont dépend l'exécution du bloc. Si elle vaut true, le bloc est exécuté, sinon on sort de la boucle for. La troisième partie de l'en-tête est évaluée après chaque passage dans la boucle. Dans l'exemple précédent, nous exécutons neuf fois le bloc d'instructions, puis i atteint la valeur 10, ce qui rend fausse la deuxième expression. Nous aurions pu utiliser pour la même tâche une boucle while.

```
var nombre, factorielle, i;
nombre = 9;
factorielle = i = 1;
while(i <= nombre)
{
 factorielle *= i;
```

```
 i++;
}
```

Ou même une boucle do while qui fonctionne aussi pour le calcul de la factorielle :

```
var nombre, factorielle, i;
nombre = 9;
factorielle = i = 1;
do
{
 factorielle *= i;
 i++;
}
while (i <= nombre);
```

Les fonctions

Le paragraphe précédent montrait comment calculer la factorielle d'un nombre. Pour calculer plusieurs factorielles différentes, vous pourriez recopier le petit morceau d'instructions, mais cette solution, triviale, possède au moins deux inconvénients immédiats : le code peut très vite devenir illisible et long, tandis que la modification de l'algorithme s'avère plus laborieuse. En programmeurs raisonnables, nous utiliserons donc une fonction ! C'est en effet un regroupement d'instructions, chargé d'accomplir une tâche spécifique et qui possède un nom. Ce dernier est utilisé par le reste du script pour exécuter la fonction autant de fois que nécessaire et il obéit aux mêmes règles de nommage que les variables. Pour créer une fonction, le mot-clé function s'utilise ainsi :

```
function nom_de_la_fonction ()
{
 // instructions à exécuter
}
```

Quand l'interpréteur rencontre une telle déclaration, il n'exécute pas les instructions mais se souvient seulement qu'il existe une fonction qui possède tel nom à tel endroit du script, pour y

revenir quand elle sera appelée. Après avoir déclaré votre fonction, vous pouvez l'exécuter simplement comme ceci :

```
nom_de_la_fonction();
```

Le corps de la fonction peut contenir n'importe quelle instruction. Cependant il existe un mot-clé particulier, return, qui termine immédiatement l'exécution de la fonction et définit sa valeur de retour. Une fonction renvoie en effet toujours une valeur au code qui l'a appelée : cette dernière remplace l'appel de la fonction, un peu à la manière d'une variable, et peut être utilisée comme opérande dans une expression. Le mot-cléreturn est habituellement suivi d'une expression : le résultat de son évaluation détermine la valeur de retour. Mais s'il est utilisé seul, il équivaut à return undefined;. Enfin, une fonction qui ne s'est pas terminée parreturn retourne toujours implicitement undefined.

Dans de nombreuses situations, l'exécution d'une fonction requiert certaines valeurs qui vont l'influencer : une fonction factorielle doit par exemple connaître le nombre dont il faut justement calculer la factorielle. Ce rôle de passage de valeurs est assuré par les paramètres, qui s'ajoutent entre les parenthèses qui suivent le nom de la fonction. À sa déclaration, vous y mettez les noms des paramètres, pour les utiliser dans le corps de la fonction comme des variables, mais qui ne seront pas connues à l'extérieur.

```
function ma_fonction(mon_param1, blabla, bouh, arg4)

{ /* etc., etc., etc. */ }
```

À l'appel de la fonction, les valeurs à donner à chaque paramètre prennent naturellement leur place entre les parenthèses, en suivant l'ordre de la déclaration.

```
ma_fonction(42, "un texte", true, null);
```

Lors de cette exécution de la fonction, **mon_param1** vaudra 42, **blabla**vaudra "un texte", etc. ECMAScript n'est pas très strict quant au passage des paramètres, c'est-à-dire que vous pouvez parfaitement passer moins de paramètres, ou au contraire plus qu'il n'en existe dans la déclaration. Les valeurs manquantes sont alors remplacées par undefined, tandis que les valeurs en trop sont ignorées (sauf si vous savez y accéder…).

Ce petit tour d'horizon rapide des fonctions nous permet maintenant d'écrire notre fameuse fonction factorielle : elle utilisera un paramètre, normalement un entier positif, et renverra la factorielle de cet entier.

```
function factorielle (nombre)
```

```
{
 var fact = 1;
 for (var i = 1; i <= nombre; i++)
   fact *= i;
 return fact;
}
```

Ce n'est finalement pas très compliqué, ni robuste d'ailleurs, mais terriblement efficace à l'utilisation, car vous pouvez désormais calculer autant de factorielles qu'il vous plaîra.

```
var factorielle_de_9 = factorielle(9);
factorielle(42); // Un grand nombre
factorielle(1e42); // Un nombre trop grand : on obtiendra +Infinity
```

Le dernier point de cette section concerne la récursivité, qui fait évidemment partie de tout langage qui se respecte un peu. Ce concept permet à une fonction de s'appeler elle-même pour effectuer une tâche récurrente, comme c'est parfaitement le cas de notre fonction factorielle.

```
function factorielle (nombre)
{
 return nombre ? nombre * factorielle (nombre - 1) : 1;
}
```

Ce genre de fonction qui se mord la queue peut être très utile, voire indispensable dans certaines situations. Je vous laisser méditer l'exemple précédent, qui fonctionne tout aussi bien que l'ancienne fonction, avant de passer aux objets ECMAScript.

Conclusion

Cet article a permis, je l'espère, d'acquérir quelques rudiments de la programmation avec ECMAScript. Il est cependant loin d'être exhaustif et plusieurs aspects du langage n'ont pas été évoqués. C'est par exemple le cas des objets, sur lesquels repose une grande partie du noyau ECMA-262. Finalement, seule la syntaxe de base trouve sa place ici. De plus ECMAScript ne peut pas se résumer qu'à son noyau, et pour créer des scripts vraiment utiles, il faudra utiliser d'autres technologies qui étendent le noyau, comme principalement le DOM.