Cours de JavaScript

Par Jacques Guizol

Date de publication : 15 décembre 2003

Dernière mise à jour : 12 juin 2013

CONFIRMÉ

Cette introduction au langage JavaScript est en cours d'élaboration et sera donc amendée au fur et à mesure de sa rédaction.

Au départ, c'est un pari qui fut pris, d'introduire très tôt en DEUG des enseignements portant sur des outils actuels, utilisés pour un développement à la fois sympa et fonctionnel des sites web. C'est ainsi que depuis l'année universitaire 2000-2001, les étudiants de deuxième année de MIAS (Mathématiques et Informatique Appliquées aux Sciences) « subissent » des enseignements de Perl et de JavaScript. En fait, quand je dis « subissent », c'est un petit clin d'œil, car, nous semble-t-il, nos étudiants sont au contraire satisfaits de cette nouveauté qu'ils mettent aussitôt en application dans leurs sites personnels pour obtenir des pages plus chouettes que ce qu'ils avaient (ou n'avaient même pas) auparavant.

Oh ! j'allais oublier… L'étude de ce cours présuppose une connaissance approximative de HTML. Si vous avez besoin d'y faire référence, vous pouvez le faire ici et positionner un marque-page. Le cours que vous trouverez **ici** est l'œuvre de Marc Bollard. Si vous le préférez, vous pouvez aussi télécharger la version PDF de la doc.

I - Aperçu rapide de JavaScript

I-A - Qu'est-ce que JavaScript ?

JavaScript est,comme son nom l'indique,un ***langage de scripts*** . De ce point de vue, il est d'un ***apprentissage simple et rapide*** et permet à des programmeurs débutants de réaliser leurs premières pages web sous une forme particulièrement attrayante et fonctionnelle.

Mais, JavaScript est aussi un ***langage de programmation à part entière*** , permettant de réaliser des applications complexes dès que l'on a acquis une connaissance suffisante du langage et de ses diverses possibilités. Deux restrictions qu'il convient toutefois de souligner :

- JavaScript ne dispose d'***aucune fonctionnalité graphique*** *;*
- pour des raisons bien compréhensibles de sécurité, JavaScript ne peut ni lire ni écrire un fichier.

JavaScript est un langage non typé (« loose typing »). Cela signifie qu'il n'est pas forcément utile de déclarer les variables qui vont être utilisées et encore moins d'indiquer leur type.

<a onclick="this.href=window.open('./fichiers/
Reml.1.1.html','R_11','width=500,height=350,scrollbars=yes');return false"
onmouseup="off('Bouton1')" onmousedown="on('Bouton1')" onmouseover="status='ou
plus exactement.....';return true" href="#"> <img width="20" height="20" border="0"
align="absmiddle" name="Bouton1" src="./images/bouton_o.gif">
En fait, une variable est implicitement déclarée dès son apparition et typée par son contexte d'utilisation (affectation).

```
var entier;
var i = 0;
valeur = i;
```

Dans l'exemple ci-dessus, chacune des variables a été déclarée ; les deux premières de façon explicite alors que la troisième l'a été de façon implicite. Concernant le typage, à chaque instant, le type d'une variable est celui qui a été induit par le contexte d'utilisation le plus récent. Si, bien que déclarée, une variable n'a pas été affectée, elle possède néanmoins un type : undefined.
Ici, les variables i et valeur sont de type numérique, la première par son initialisation (affectation lors de la déclaration), l'autre par affectation d'une variable elle-même de type numérique. Quant à la variable entier, son type est pour l'heure « undefined ». Pour bien comprendre ce mécanisme, reprenons le précédent exemple :

```
var entier;                // déclaration explicite, type undefined
var i = 0;                 // déclaration explicite, type numérique
valeur = i;                // déclaration implicite, type numérique
entier = "0123";        // typage chaîne de caractères
i = entier + valeur;
  // i devient la chaîne de caractères "01230". L'affectation a nécessité la conversion de valeur en "0" pour la
```

Finalement, parler de langage non typé peut prêter à confusion, voire à incompréhension. En effet, le typage est omniprésent, il y a des fonctions propres au type chaîne de caractères, d'autres spécifiques aux variables numériques, mais nous verrons que bien souvent par des mécanismes automatiques de conversion (tels que nous venons de le voir), nous pourrons employer des variables dans un contexte qui ne leur est pas naturel. Cette facilité sévèrement jugée par certains puristes est un des facteurs de puissance de ce langage. En fait, plutôt que non-typage, il vaudrait mieux parler de transtypage.

Un autre aspect de ce transtypage est qu'une variable déjà définie et typée comme nous venons de le voir, peut être redéfinie et/ou retypée à tout moment.

```
var entier = "0123";    // déclaration explicite, type chaîne de caractères
```

```
i = entier + "456";
  // déclaration implicite, type chaîne de caractères,i prend la valeur "0123456"
entier=i*1;                // transtypage chaîne de caractères -
> nombre, entier prend la valeur numérique 123456
var i=entier/2;          // redéfinition explicite & transtypage chaîne de caractères -
> nombre, i prend la valeur numérique 61728.
```

Sans nous étendre d'excès sur ce point que nous développerons en détail dans le prochain chapitre, le survol de ce premier paragraphe serait néanmoins incomplet si l'on ne précisait pas que JavaScript est un langage orienté objet. Pour le lecteur qui ne connaît que les langages orientés action (type Pascal ou C), dans lesquels les données avaient un rôle secondaire par rapport au rôle primordial concédé à l'aspect essentiellement algorithmique, il va lui falloir reconsidérer sa démarche de programmation. Dans ces langages orientés objets tels que C++, Java ou JavaScript, tout est pensé en fonction des données, les objets, à leur structure (les propriétés), aux manières grâce auxquelles on va y accéder (les méthodes), aux manières de les créer qui sont des méthodes particulières (les constructeurs). Cette façon de programmer est très bien adaptée à un souci de sécurité, particulièrement crucial dans le domaine d'internet, puisqu'on ne peut pas faire n'importe quoi sur n'importe quoi ! Cette façon de voir est caricaturale, mais pour une introduction qui se veut concise, elle conviendra…

Malgré ses faux airs de ressemblance dont, en particulier, son nom, JavaScript n'a absolument rien à voir avec Java. Alors que JavaScript, nous l'avons vu, est un langage interprété, « non typé », qui ne possède aucune possibilité graphique, ne peut manipuler des fichiers, Java, au contraire, est compilé, a une panoplie de typages particulièrement riche, autorise la mise en œuvre de dessins et même, le plus souvent, d'animations et dispose de nombreuses fonctionnalités d'exploitation de fichiers. À l'inverse, Java ne peut en aucune manière contrôler le fonctionnement d'un navigateur ou avoir accès aux objets contenus dans un document HTML, chose que fait à merveille JavaScript y compris pour contrôler les applettes Java !!!

Ce sont donc deux compères qui se complètent à merveille dans l'univers des nouveaux médias de communication, mais il ne faut voir aucun lien direct ou indirect dans la consonance de leur nom, si ce n'est de basses considérations commerciales qui sont venues substituer au nom de « LiveScript » prévu à l'origine celui de « JavaScript », peut être plus porteur dans le début des années 90 du fait de l'impact grandissant du produit de Sun Microsystems.

I-B - Les diverses formes de JavaScript

JavaScript est à la programmation sur Internet ce que Frigidaire est aux réfrigérateurs ;-)
En effet, c'est une marque déposée par Netscape. Dans la firme Microsoft, on ne connaît pas JavaScript !!! (Surtout depuis le procès retentissant qui a opposé Netscape à Microsoft, l'issue n'ayant pas été à l'avantage de ce dernier…)
Non, chez Microsoft, on ne connaît que JScript !!!

Sur la base de ces considérations, on voit poindre le spectre de l'incompatibilité qui va obliger le « programmeur de scripts » à tester le type de navigateur utilisé par le client, sa version, la plate-forme qui le supporte (Windows 16 ou 32, MacPPC, Unix…) et bien d'autres choses encore, afin d'adapter le script qu'il développe à tous les cas de figure envisageables s'il veut que son document puisse être accessible le plus largement possible.

Un script dont l'idée de base se veut simple peut donc devenir « une usine à gaz » si l'on doit tenter de prendre en compte toutes les solutions possibles. Dans ces conditions, le premier besoin du programmeur est de tester l'ensemble de son script dans les différentes situations. Là encore, les méthodes diffèrent :

- sous Netscape, rien de plus simple. Après chargement du document contenant le script à tester, il suffit, **en local**, d'entrer dans la ligne URL de la fenêtre du navigateur la commande « javascript: ». Une fenêtre s'ouvre vous indiquant, pour chaque erreur rencontrée, l'endroit où elle a été détectée, son type et la raison probable de son origine. Vous pouvez aussi récupérer par réseau un analyseur à l'adresse :

;

- sous Microsoft Internet Explorer, malgré l'éventail beaucoup plus modeste des fonctionnalités offertes par cet interpréteur, on ne peut espérer bénéficier d'aucune possibilité de débogage intégré. Il ne vous reste que la possibilité de télécharger un programme permettant de diagnostiquer les erreurs à l'adresse :

.

I-C - Vers une normalisation ?

Face aux problèmes que nous venons d'évoquer, l'utilisateur est en droit d'espérer une normalisation de ce langage de plus en plus utilisé, afin que sa tâche soit enfin simplifiée en n'ayant plus l'obligation de tripler ou quadrupler la plupart de ses programmes. Soyez satisfaits, un standard existe !!!!
En l'état actuel des choses il a pour nom ECMA-262 (European Computer Manufactures Association). Cette **norme** dans sa troisième version date de 1999 ! Depuis, une nouvelle norme ECMA-327 a été proposée en juin 2001. Une autre standardisation ISO (International Organization of Standardization) ayant pour code ISO/IEC-16262 a été définie. Mais ne vous réjouissez pas trop vite… Ces standards définissaient, il y a encore peu, un langage correspondant approximativement à JavaScript 1.1, alors que nous en sommes aujourd'hui à la version 1.4. Toutefois, il convient de noter qu'une **nouvelle version** de la norme ISO/IEC-16262 a été proposée en 2002.
En fait ces standards ne voulant avantager ni Netscape, ni Internet Explorer ont choisi d'adopter un niveau très bas (en gros, l'intersection des fonctionnalités offertes par les deux navigateurs), ce qui leur confère un intérêt de portabilité, mais passablement douteux sur le plan des fonctionnalités. (C'est pourquoi la plupart du temps les créateurs préfèrent adapter leurs pages aux deux navigateurs les plus utilisés en prévoyant les particularités de chacun.) Conscients de cette évidence, Netscape et Microsoft ont d'ores et déjà proposé une mise à jour du standard. Mais la procédure n'en est qu'à ses débuts et vous aurez le temps d'aligner de nombreuses lignes de programme avant que le nouveau standard soit adopté. Il y a fort à parier qu'il sera d'ailleurs devenu obsolète lorsqu'il entrera en vigueur ;-)
Dans la suite de ce cours, chaque fois qu'une nouvelle fonctionnalité sera introduite, la version et l'appartenance au standard seront précisées.

I-D - Que fait-on avec JavaScript ?

Un programme JavaScript s'intègre directement dans votre page HTML entre deux balises <script> et </script>. De plus, afin de solliciter l'interpréteur JavaScript, on précise dans la balise ouvrante : <script langage="JavaScript">.

JavaScript peut intervenir dans un document HTML essentiellement de deux façons :

- la première que je qualifierai de synchrone s'exécute au chargement de la page. Le programme JavaScript a pour objet d'écrire dans le document du code HTML pouvant s'adapter dynamiquement à plusieurs facteurs tels que le type de configuration matérielle et/ou logicielle du client (navigateur, plugins…), le contenu éventuel de certains cookies, la date, l'heure du moment présent, etc.
  Les parties du document dépendant de facteurs variables ne peuvent donc pas être écrites « en dur » dans le code HTML (le simple fait que HTML n'autorise pas de tests permet de comprendre cela). À chaque endroit où le contenu du document devra s'adapter à ces facteurs extérieurs, un script sera placé et exécuté au moment du chargement.
   <img width="20" height="20" border="0" align="absmiddle" name="Bouton2" src="./images/bouton_o.gif">
  Ces scripts prendront naturellement leur place dans le corps du document (entre <body> et </body>) ;
- la seconde, par opposition, aura un fonctionnement asynchrone. Il s'agira de scripts qui s'exécuteront non pas au moment du chargement de la page.

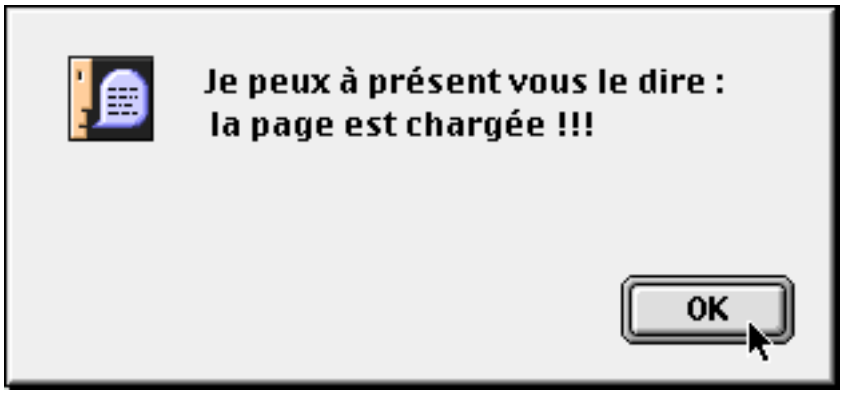
 <img width="20" height="20" border="0" align="absmiddle" name="Bouton3" src="./

images/bouton_o.gif"> <img width="20" height="20" border="0" align="absmiddle" name="Bouton4" src="./images/bouton_o.gif">

I-E - Comment JavaScript est-il invoqué ?

Comme nous l'avons dit précédemment, la façon la plus simple d'invoquer une séquence JavaScript consiste à inclure dans le corps du code HTML un script dont le résultat apparaîtra exactement à l'endroit où il s'exerce. C'est le fonctionnement que nous avons appelé synchrone et dont nous avons vu un exemple avec l'affichage de la date courante.
Dans le cas du fonctionnement asynchrone, nous avons vu qu'il intervenait à la suite d'un événement sur un des composants de la fenêtre active. Dans ce cas deux nouvelles possibilités peuvent être envisagées :

les actions (instructions) JavaScript que l'on désire associer à un type d'événement particulier sont indiquées en totalité dans la liste des caractéristiques de l'objet sur lequel l'événement s'est produit ;
Nous avons vu une utilisation de cette méthode dans l'apparition de la fenêtre d'alerte indiquant la fin du chargement de la page



Cela était obtenu par :

```
<body ...
onload="alert('Je peux &agrave; pr&eacute;sent vous le dire :\n la page est charg&eacute;e !!!')"...
```

où onload indique l'événement survenant sur la page (document) et auquel est associée l'instruction JavaScript qui suit.

Dans cette même liste des caractéristiques de l'objet apparaît une référence vers les instructions JavaScript que l'on désire associer à l'événement pris en compte sur ledit objet. Dans ce cas, la référence est en fait un nom de fonction (objet que nous introduirons plus tard) dont la définition contient donc les instructions (cette fonction sera, bien entendu, définie à l'intérieur de balises <script>).
Dans l'exemple précédent, nous avions en fait :

```
<body …
onload="alert('Je peux &agrave; pr&eacute;sent vous le dire :\n la page est charg&eacute;e !!!')"
onUnload="Fermer()">
```

L'événement de « déchargement » (correspondant à la fermeture) est traité dans une fonction, que l'on a choisi d'appeler Fermer, qui avant de fermer la fenêtre courante, ferme celle qui a permis de l'ouvrir elle-même. Cette fonction Fermer et éventuellement d'autres, seront définies à l'intérieur de balises de script. Ces balises de script (i.e. les scripts) avec celles que nous avons vues en 1 peuvent être en nombre quelconque dans une page. En fait ils constituent les parties d'un seul et même programme JavaScript. À ce titre, les variables ou fonctions de l'un seront disponibles et utilisables dans n'importe quel autre.

<a onclick="this.href=window.open('./fichiers/
Reml.5.1.html','R_51','width=700,height=500,scrollbars=yes');return false"
onmouseup="off('Bouton5')" onmousedown="on('Bouton5')" onmouseover="status='... et cela n
\'est pas sans conséquence.....';return true" href=""> <img width="20" height="20" border="0"
align="absmiddle" name="Bouton5" src="./images/bouton_o.gif">
Sous votre navigateur, vous pouvez aussi exécuter du JavaScript en tapant directement des instructions dans la ligne URL du navigateur à la suite du préfixe "javascript:" (attention, n'oubliez pas les deux points).
Ce type d'utilisation est plutôt réservé pour la mise au point de portions de programme.
À noter que sous Netscape, si vous tapez uniquement javascript: dans la ligne URL, une fenêtre console s'ouvre, vous permettant de tester des instructions entrées dans la zone de saisie.

Une entité HTML est une séquence de caractères débutant par &. Par exemple, un « é » dans une page HTML, sera en fait la traduction de « é ». Netscape, depuis sa version 3 autorise dans le code HTML, des entités JavaScript. Leur syntaxe est simple :

```
&{ liste d'instructions JavaScript };
```

Une seule restriction, ces entités JavaScript ne peuvent être utilisées que dans les valeurs d'objets HTML. Par exemple, dans le champ de texte ci-dessous, seuls ceux utilisant Netscape pourront voir le résultat de l'appel à une fonction nommée « Bidon » ; les autres ne verront que l'appel de celle-ci sous la forme décrite précédemment :

<form action="" method="post"> <input type="text" size="80" value="&{Bidon();};"
name="textfield"> </form>
Que les autres (ceux qui ne voient que le nom de la procédure appelée) se consolent, ce n'était pas bien important… De toute façon, dans les versions récentes des navigateurs de la famille Mozilla (Netscape, Mozilla, Camino, Safari…), cette fonctionnalité n'existe plus.

Cette liste, non exhaustive, s'arrêtera sur la possibilité qu'offre Netscape 4x d'inclure dans votre page HTML des commentaires conditionnels. Nous reviendrons dans le chapitre **suivant** sur ces types de commentaires. Précisons seulement ici qu'entre des balises habituelles de commentaires HTML : <!-- et -->, on trouve en premier lieu une expression booléenne JavaScript. Ce test est suivi de code HTML ou, pour ce qui nous intéresse ici, d'un script. Dans le cas où le résultat du test est false ou que le test n'est pas évalué, car le navigateur ne supporte pas cette fonctionnalité, l'ensemble conserve le caractère de commentaire. Dans le cas contraire, le script est exécuté.

Là encore, cette fonctionnalité peu utilisée a disparu sur les navigateurs de nouvelle génération.

Nous venons de montrer diverses façons par lesquelles JavaScript peut être invoqué. Cela nous a permis de découvrir une partie des différents modes d'insertion de JavaScript dans une page. Avant d'en terminer, il est nécessaire d'ajouter qu'au lieu de mettre les instructions JavaScript utilisées par une page en totalité dans celle-ci, on peut avoir recours à l'utilisation de fichiers contenant tout ou partie des divers éléments utilisés.

L'utilité d'un tel système est multiple :

1. Lorsque plusieurs pages d'un site utilisent des variables ou fonctions identiques, au lieu de dupliquer celles-ci dans chacun des documents HTML, il sera plus judicieux de regrouper toutes les parties communes dans un seul fichier et de préciser dans chacun des documents HTML, seulement le nom du fichier dont il s'agit. Cela réduit l'espace disque serveur nécessaire pour stocker les fichiers HTML et rend toute modification plus simple et plus sûre ;
2. Si l'ensemble des parties communes est important, le fait d'en soulager chacune des pages qui les référencent rendra plus rapide le chargement de celles-ci ;
3. Pour ce qui est du temps de chargement des parties communes, il n'interviendra que lors de la première utilisation puisqu'à partir de cet instant, l'ensemble sera dans le cache ;
4. Le mode d'accès au fichier par URL est tel que l'on peut tout à fait considérer que ce fichier est local au site serveur (cas le plus courant), mais peut aussi concerner n'importe quel serveur sur le web.

Les fichiers peuvent être de deux types. Le premier type concerne des fichiers qui contiennent du source JavaScript pur (sans HTML). Le fichier sera postfixé par l'extension .js et chaque page le référençant le fera par : <script src="path/nom de fichier.js"></script>.

```
<script src="../../Sources_JS/Sces_Chap1.js"></script>
```

Pour cela votre serveur devra être configuré afin d'exporter ces fichiers avec le type MIME application/x-javascript.

Le second type de fichier concerne les fichiers-archives ayant une extension .jar (Java archive) et contenant plusieurs fichiers du type de ceux que l'on vient de voir, mais compressés (Netscape propose un outil gratuit réalisant ces archives). La référence à ces fichiers se fera d'une façon proche de la précédente :

```
<script archive="../../Archives_JS/CoursJS.jar" src="Sces_Chap1.js"></script>
```

I-F - La place des scripts dans le document a-t-elle une importance ?

Nous avons déjà abordé ce problème au sujet de l'utilité de placer un script entre <head> et </head> afin de pouvoir le rendre activable avant le chargement complet de la page. Ce qui nous intéresse ici c'est de savoir s'il est possible d'utiliser un identificateur (par exemple une fonction ou une variable indiquant le type de navigateur utilisé par le client) à un endroit de la page HTML alors qu'il est défini dans un script placé plus loin dans la page.

La réponse est OUI !!! En fait, au moment du chargement, tout le code JavaScript est analysé quel que soit son emplacement dans la page. Si bien qu'en n'importe quel lieu de la page, on peut référencer toute variable pourvu qu'elle soit définie au plus haut niveau et donc qu'elle soit visible.

En conséquence, hormis les considérations que nous avons évoquées au sujet de la réaction précoce aux événements (avant le chargement complet de la page §4) et hormis les cas d'utilisation synchrone, les scripts peuvent être placés n'importe où dans la page HTML. Les références en avant seront bien traitées.

Au surplus, pour venir confirmer cela, rappelons ce que nous avons **dit** : tous les éléments JavaScript d'une page constituent les parties d'un seul et même programme JavaScript !!!

Ce premier chapitre visant à introduire les généralités de JavaScript trouve ici son épilogue qui n'est probablement que temporaire, car il y aura sûrement d'autres points à développer…

II - Variables, Objets, Types et Valeurs

<script language="JavaScript"> var S = "L\'utilisation d\'une chaine dans une\nexpression arithmetique affiche :\n \n" + 'Chaine'*1 + "\n \npour prevenir que ladite chaine \"is Not a Number\""; var Bouton_OFF; var Bouton_ON; onload=Declare(); </script> <script language="JavaScript"> function defaultHandler() {return false} function silentHandler() {return true} function customHandler(desc,page,line,chr) { alert('Une erreur JavaScript est apparue !\n' +'Elle a \u00e9t\u00e9 prise en compte par un gestionnaire propre a cette page.\n' +'Il s\'agit d\'une erreur de syntaxe dans un litt\u00e9ral.\n' +'\nDescription : \t'+desc +'\nPage : \t'+page +'\nNum\u00e9ro de ligne : \t'+line) return true } // window.onerror=customHandler; </script>

II-A - Les identificateurs

En tout premier lieu, en particulier pour ceux qui n'y sont pas habitués, il est important de signaler que le langage est sensible à la casse des caractères. Ainsi, si un identificateur est défini sous la forme MonIdent, il ne pourra être référencé que par cette écriture. Tout référencement ne respectant pas la casse des caractères composant un identificateur provoquera une erreur par JavaScript lors de l'exécution.

```
var MonIdent = 12;  // déclaration explicite, type numérique
var i = 3;  // déclaration explicite, type numérique
i = i * monIdent;  // monIdent non déclaré -> Erreur à l'exécution
```

Dans l'exemple ci-dessus, on a commis l'erreur, sans doute involontaire d'utiliser monIdent au lieu de MonIdent. Dans certains contextes bien particuliers, cette contrainte de respect de casse est levée, mais il vaut mieux prendre l'habitude de la respecter systématiquement. En particulier, lorsque nous utiliserons des fonctions prédéfinies de JavaScript, il faudra veiller à respecter cette contrainte faute de quoi on aura droit à l'erreur « fonction non définie ».

JavaScript ignore tout caractère d'espacement (blanc, tabulation, retour à la ligne, etc.) apparaissant entre les entités lexicales. Ainsi, pour des besoins de présentation d'un programme (pour une meilleure lisibilité) vous pouvez sans aucun problème insérer des blancs par exemple, de part et d'autre d'opérateurs), indenter des portions constituant des blocs ou couper une ligne trop longue. Par contre une entité lexicale ne peut en aucun cas contenir un caractère d'espacement sous peine de la transformer en deux entités lexicales et donc, vraisemblablement de voir apparaître une erreur. Par exemple, vous ne pouvez pas définir un identificateur comme étant Mon_Ident.

Les identificateurs servent à nommer des variables, des fonctions ou encore des étiquettes (JavaScript 1.2). La syntaxe d'un identificateur impose en première place un caractère alphabétique (majuscule ou minuscule) ou $ (dollar) ou _ (souligné). Les caractères suivants peuvent être n'importe quelle combinaison de ces mêmes caractères plus des chiffres.

```
Mon_Ident
$id31
_entier
valeur
```

Évitez de commencer un identificateur par deux « soulignés » (__) consécutifs, car il peut y avoir conflit avec certaines fonctions prédéfinies du langage.

II-B - Les littéraux

<script language="JavaScript"> function ShowHide(x,nim){ Img='Tool'+nim; with(document){ switch(x){ case 0 : if(navigns4)layers['Comment'].visibility='hide'; else {if (navigie) all.item("Comment").style.visibility='hidden'; else getElementById("Comment").style.visibility='hidden'; } status=''; images[Img].src='./images/Tool_up.gif'; return case 1 : if(navigns4)layers['Comment'].visibility='show'; else {if (navigie) all.item("Comment").style.visibility='visible'; else getElementById("Comment").style.visibility='visible'; } status='Visualisez les résultats...'; images[Img].src='./images/Tool_sel.gif'; return true } } } function Interp() { window.onerror=customHandler; alert('valeur = '+eval(document.EssLit.litteral.value)); } </script> <script language="JavaScript"> var msg, TDefil, Carac=' '; var index=0; function Defiler(i){ clearInterval(TDefil); switch(i){ case 1 : msg="Le d\u00EEner sera pr\u00EAt \u00E0 l'heure..... "; break; case 2 : msg="Le dîner sera prêt à l'heure..... "; } TDefil=setInterval("defil()",100); return false; } function defil(){ // nb_char=msg.length; // index = ++index % nb_char; with(document.$Exemple.Defiltxt) if(navigie)innerText=msg; else value=msg; Carac=msg.substr(0,1); msg=msg.substr(1)+Carac; } </script> <script language="JavaScript"> var Tab=new Array(1,2,3,4,5); function Init(){ Modifier(Tab); Afficher ('Valeurs contenues dans le tableau Tab :\n',Tab); } function Modifier(T){ var AutreT=new Array (9,8,7,6,5); T[2]=100; T=AutreT; Afficher('Valeurs contenues dans le tableau T :\n',T); } function Afficher(Deb,T){ var S=Deb; for(var i=0;i<T.length-1;i++) S+=T[i]+', '; S+=T[i]+'.'; alert(S); } </script>

Les valeurs constantes affectées à des variables ou intervenant dans des comparaisons à celles-ci sont appelées des littéraux. Ceux-ci peuvent être de type numérique (entiers ou réels), de type chaînes de caractères ou de type booléen.

À ces différents types de littéraux, il convient d'ajouter, en JavaScript, la constante null qui permet de représenter une valeur différente de toute autre signifiant ainsi qu'une variable possédant cette valeur n'en contient pas d'autre. Elle n'a donc pas été affectée.
Bien que la distinction soit subtile, il ne faut pas confondre null et undefined. D'abord l'un correspond, nous venons de le voir, à un littéral, ce qui n'est pas le cas pour l'autre. undefined est une valeur qui est retournée (en cas de demande d'écriture ou par appel de la fonction typeof()) pour une variable définie, mais non affectée (d'où la confusion), ou non définie ou pour une propriété non définie d'un objet (nous reviendrons là-dessus).


le littéral null n'est pas, comme en C ou C++ équivalent à 0. Dans certains cas, il pourra être converti en 0, mais ne lui sera jamais équivalent.

De façon générale, les littéraux de type numérique sont en représentation décimale. Le langage accepte néanmoins des littéraux entiers en représentation octale ou hexadécimale. Les premiers, dont les digits sont compris bien évidemment entre 0 et 7, ont pour caractéristique de débuter par le chiffre « 0 ». Les seconds débutent par les caractères « 0x » et ont pour digits n'importe quel chiffre entre 0 et 9 ainsi que les lettres A, B, C, D, E ou F (majuscules ou minuscules).

Les réels peuvent avoir une représentation en virgule fixe (du type 3.14159) ou en virgule flottante (du type 0.314 E+1 ou encore .314 e+1). Dans tous les cas, la marque décimale est le « . » et non la « , », quant au signe d'exponentiation, il peut être indifféremment en majuscule ou minuscule.

Les chaînes sont encadrées par des quotes simples (') ou doubles ("). Si à l'intérieur d'une chaîne encadrée par un de ces caractères, ce même caractère doit apparaître, il sera précédé du caractère « antislash » (\ ). Il existe bien d'autres caractères subissant ce traitement. Nous en reparlerons lorsque nous étudierons spécifiquement les chaînes de caractères et les expressions régulières.

```
3.14     // réel fixe
48       // entier décimal
.327E+4  // réel flottant
0357     // entier octal
0x4f7C   // entier hexadécimal
'C\'est une cha\u00eene'
"... celle-l\u00e0 aussi !!"
false    // booléen
null
0358    // entier décimal (chiffre 8)
```

Les littéraux peuvent avoir des formes plus complexes que pour les types primitifs que nous venons de voir. C'est le cas des « initialisateurs » de tableaux ou d'objets. Nous ne donnerons ici que deux exemples, sans rentrer dans les détails que nous aborderons lorsque nous étudierons ces deux concepts

```
[0,8,3,7]                       // tableau d'entiers
{Nom:"Terieur",Prenom:"Alain"}  // Propriétés d'objet
```

Comme en C, C++, Java, Pascal, les instructions se terminent généralement par le caractère « ; » afin de matérialiser la fin de l'instruction. Toutefois, en JavaScript, celle-ci est optionnelle pour chaque instruction suivie d'une fin de ligne. Cela peut sembler être un avantage, mais peut aussi représenter un piège en fonction de ce qui a été dit plus haut. En effet, il a été dit que l'on peut, pour des raisons de lisibilité, introduire des retours à la ligne dans des instructions très longues. Dans ce cas, il faut veiller à deux choses :

1. La coupure ne peut avoir lieu à l'intérieur d'une unité lexicale ; il est hors de question de couper en deux des identificateurs, des chaînes de caractères ou des littéraux : cela va irrémédiablement entraîner des erreurs d'analyse (identificateur non déclaré, chaîne non terminée) ou d'exécution ;
2. Certaines instructions de rupture de séquence peuvent être assorties ou non d'un identificateur. Elles sont particulièrement sensibles au problème et leur mauvaise interprétation peut avoir des conséquences fâcheuses sur la logique ou l'exécution du programme. C'est le cas des instructions return ou break que nous verrons plus loin

```
var MonIdent = 12
var i = 3                    //Instruction bien analysée
2 * MonIdent/2                // L'identificateur i est affecté à 3 (partie du littéral 32...)

  // Curieusement cette ligne ne provoque pas d'erreur  (elle semble ignorée... Il n'en est rien)
var j = 8                    // L'identificateur j est affecté à 8 (partie du littéral 81...)
1 * Mon                       //Erreur : identificateur Mon non défini !!!!... et c'est tout !!!!
Ident                        //Erreur : identificateur Ident non défini !!!!
var Nom = "Alain Terieur     //Erreur : chaîne de caractères non terminée.
"
```

De façon générale, il vaut mieux utiliser le point-virgule pour terminer les instructions. Cela rend le texte plus clair et permet de visualiser très rapidement les risques d'erreur tels que ceux qui viennent d'être signalés.

II-C - Les commentaires

On ne pouvait pas traiter ce chapitre sans parler de ce que l'on a utilisé plusieurs fois jusqu'ici… les commentaires. Les commentaires sont non seulement utiles, mais il vaudrait mieux dire indispensables lorsqu'on programme. Ils permettent d'indiquer la nature du traitement effectué à tout endroit d'un programme, de préciser une éventuelle astuce de programmation, de placer des pense-bêtes afin de se souvenir qu'il convient de prévoir tel ou tel cas non encore traité. Enfin et surtout, ils permettent de retrouver rapidement la logique du programme lorsqu'on doit le modifier quelques mois, voire quelques années après qu'on l'a écrit.

En JavaScript on dispose de deux types de commentaires : les commentaires ligne et les commentaires multilingues. Les premiers portent principalement sur une instruction, pour en préciser la fonction ou le résultat. Ils sont introduits à la suite de deux caractères « slash » (//) ; tout ce qui est inclus entre ces caractères et la fin de la ligne est considéré en commentaire. Les seconds permettent de préciser le contenu d'un bloc d'instructions, éventuellement, l'algorithme utilisé, etc. Ces types de commentaires réclament donc plusieurs lignes. Le début du commentaire est précisé par /*, tandis que la fin l'est par */. Ce type de commentaire disposant d'une marque de début et d'une marque de fin, il pourra aussi être utilisé pour des commentaires très courts, au contraire, en particulier à l'intérieur même d'une instruction (comme nous le verrons en exemple.

Il convient de noter aussi la grande utilité de ces commentaires pour déboguer les programmes. Ils permettent soit de transformer en commentaires certaines parties sujettes à provoquer des erreurs pour vérifier le reste, soit de prévoir des traces de mise au point que l'on peut à loisir faire disparaître simplement en les transformant en commentaires.

```
var Rayon = 3;            // Initialisation de Rayon

/
* On va à présent s'étendre dans un long commentaire afin d'expliquer ce que l'on va bien pouvoir faire avec Rayo

var S = 3.14 /* valeur de Pi */ * Rayon * Rayon;
```

Nous ne reviendrons pas ici sur les « commentaires conditionnels » que nous avons déjà présentés au chapitre précédent.

II-D - Les caractères spéciaux

Comme en C, C++, Java ou Perl, JavaScript autorise la représentation de caractères particuliers à l'intérieur d'une chaîne de caractères. Pour pouvoir obtenir ces représentations, on utilise le caractère backslash (\) suivi d'un autre caractère. Nous avons vu que JavaScript dispose de deux caractères différents pour limiter les chaînes, ce qui est bien agréable lorsque la chaîne elle-même nécessite l'utilisation d'un de ces caractères, par exemple dans « L'apprentissage de JavaScript est aisé », ou encore « Le professeur a dit : "Étudiez sérieusement !" ». Mais nous rencontrerons de nombreuses situations où cela ne suffira plus. Dans ce cas, nous pourrons utiliser les caractères \' ou \" pour signifier l'occurrence d'une simple apostrophe ou de guillemets comme dans : « Le professeur a dit : \"L \'apprentissage de JavaScript est aisé !\" ».

Voici la liste de ces caractères spéciaux ainsi que leur signification :

Séquence	Caractère représenté
\b	*Backspace - retour arrière d'un caractère*
\n	*Newline - Saut de ligne*
\f	*Form feed - Nouvelle page*
\t	*Tab - Tabulation*
\r	*Return - Retour chariot*
****	*Backslash - Le caractère backslash lui-même*
\'	*Single quote - Apostrophe*
\"	*Double-quote - Guillemets*
****□□□	*N'importe quel caractère dont* □□□ *représente le code octal (trois chiffres)*
\x□□	*N'importe quel caractère dont* □□ *représente le code hexadécimal (deux chiffres)*
\u□□□□	*N'importe quel caractère dont* □□□□ *représente le code Unicode (quatre chiffres hexa)*

Il est à noter que pour tout caractère autre que ceux indiqués ci-dessus, le fait qu'il apparaisse précédé d'un backslash ne change absolument rien. Le backslash est ignoré et le caractère apparaît normalement.
Une application très simple, mais cependant fort utile pour coder les caractères non standard… Elle vous donnera l'Unicode hexa d'un caractère.

<img width="22" height="22" border="0" align="absmiddle" name="Tool9" src="./images/Tool_up.gif">

Table des unicodes des caractères les plus couramment utilisés en français :

		é	00E9			**œ**	0153		
à	00E0	**è**	00E8					**ù**	00F9
â	00E2	**ê**	00EA	**î**	00EE	**ô**	00F4	**û**	00FB
ä	00E4	**ë**	00EB	**ï**	00EF	**ö**	00F6	**ü**	00FC

L'utilisation de ces unicodes dans les navigateurs récents est préférable dans certaines utilisations au bricolage du type é=é car ces unicodes sont interprétés de façon homogène dans toutes les utilisations. Par exemple, supposons que l'on affecte à une variable S la chaîne « Le dîner sera prêt à l'heure » et que la valeur de cette variable soit utilisée en défilement.

En utilisant l'Unicode, on aura S = "Le d\u00EEner sera pr\u00EAt \u00E0 l'heure " ;
avec l'autre codage, on aura S = "Le dîner sera prêt à l'heure ".

Voyons ce que cela donne en défilement (suppression de caractère de tête + rajout à la fin + écriture) :

<script language="JavaScript"> var msg, TDefil, Carac=' '; var index=0; function Defiler(i){ clearInterval(TDefil); switch(i){ case 1 : msg="Le d\u00EEner sera pr\u00EAt \u00E0 l'heure..... "; break; case 2 : msg="Le dîner sera prêt à l'heure..... "; } TDefil=setInterval("defil()",100); return false; } function defil() { // nb_char=msg.length; // index = ++index % nb_char; with(document.$Exemple.Defiltxt) if(navigie)innerText=msg; else value=msg; Carac=msg.substr(0,1); msg=msg.substr(1)+Carac; } </script> <form name="$Exemple"> <table> <tbody><tr> <td> <input type="text" maxlength="40" size="40" name="Defiltxt"> </td> <td> <p align="center"> <a onclick="return Defiler(1)" onmouseup="Tool10.src='./images/Tool_up.gif'" onmousedown="Tool10.src='./images/

Tool_dep.gif" onmouseout="ShowHide(0,10)" onmouseover="return ShowHide(1,10)" href=""><img width="22" height="22" border="0" align="absmiddle" name="Tool10" src="./images/Tool_up.gif">
 Unicode </td> <td> <img width="22" height="22" border="0" align="absmiddle" name="Tool11" src="./images/Tool_up.gif">
  code </td> <td> <img width="22" height="22" border="0" align="absmiddle" name="Tool12" src="./images/Tool_up.gif">
 Stop </td> </tr> </tbody></table> </form>

On constate que chaque groupe désignant un caractère Unicode est effectivement réduit à un seul caractère qui peut ainsi être ôté en début de chaîne à chaque itération, ce qui n'est pas le cas pour l'autre codage.

P.-S. : à noter que les codes Unicode correspondant à l'alphabet latin et à son extension pour les caractères accentués se situent entre les codes 0000 et 00FF et correspondent exactement aux codes ASCII que l'on connaît bien. Par exemple \u00ea peut être remplacé par \xea pour représenter "ê" lorsque le système de codage est l'Occidental-Latin.

II-E - Les valeurs numériques spéciales

L'objet JavaScript Number possède pour propriétés des valeurs permettant de représenter soit des grandeurs non représentables, soit les valeurs extrêmes pouvant être atteintes, soit une indication permettant de signaler ou de tester le fait qu'une variable n'est pas un nombre.

<table width="60%" cellpadding="3" bordercolor="996633" border="5" bgcolor="eeeeee" bordercolordark="663333" bordercolorlight="999966"> <tbody><tr> <td class="programme"> Afficher Number.MAX_VALUE </td> <td width="30" bgcolor="006699" class="programme"> <img width="20" height="20" border="0" name="Bouton1" src="./images/bouton_o.gif"> </td> </tr> <tr> <td class="programme">Afficher Number.MIN_VALUE
 </td> <td width="30" bgcolor="006699" class="programme"> <img width="20" height="20" border="0" name="Bouton2" src="./images/bouton_o.gif"> </td> </tr> <tr> <td class="programme">Afficher "chaine" * 1
 </td> <td width="30" bgcolor="006699" class="programme"> <img width="20" height="20" border="0" name="Bouton3" src="./images/bouton_o.gif"> </td> </tr> <tr> <td class="programme">Afficher 1/0
 </td> <td width="30" bgcolor="006699" class="programme"> <img width="20" height="20" border="0" name="Bouton4"

src="./images/bouton_o.gif"> </td> </tr> <tr> <td class="programme">Afficher -1/0</td> <td width="30" bgcolor="006699" class="programme"> <img width="20" height="20" border="0" name="Bouton5" src="./images/bouton_o.gif"> </td> </tr> </tbody></table>

II-F - Des booléens qui sortent de l'ordinaire…

Alors que les littéraux chaînes et les littéraux numériques peuvent avoir une infinité de valeurs, les booléens, eux, n'ont que deux valeurs possibles true ou false. Cela n'est pas une nouveauté et JavaScript, pour cela, ne se différencie pas des autres langages. Il convient toutefois d'ajouter que les littéraux autres que les littéraux booléens ont eux-mêmes une valeur booléenne et il vaut mieux la connaître.

Pour tester toutes les valeurs que vous voulez, vous n'avez qu'à cliquer ici…

<img width="20" height="20" border="0" align="absmiddle" name="Bouton6" src="./images/bouton_o.gif">

Ainsi que vous pouvez le constater, seule la valeur numérique nulle quelle que soit sa forme (entière, réelle en virgule fixe ou flottante) a une valeur booléenne false, toutes les autres ayant la valeur true. null a une valeur booléenne false et toutes les chaînes ont une valeur booléenne true. Ce dernier point mérite d'être souligné, car JavaScript est capable d'interpréter toute chaîne de caractères contenant des chiffres comme des nombres en fonction du contexte dans lequel la chaîne est utilisée. Par exemple, pour mettre en évidence la valeur NaN dans l'exemple ci-dessus, on a fait écrire "chaîne" * 1, ce qui a provoqué l'apparition de la valeur NaN. Mais si l'on avait exécuté

```
alert("12"*2/3)
```

on aurait bien obtenu le résultat 8. C'est donc bien que "12" a été interprété comme la valeur 12.

Ces explications pour mettre en garde contre le piège qui pourrait consister à penser qu'en conséquence, la chaîne "0" va avoir la valeur de vérité false comme le chiffre correspondant. Il n'en est rien ! "0" ne déroge pas à la règle indiquant que les chaînes quelles qu'elles soient ont pour valeur booléenne true.

Vous pouvez vérifier tout cela :

lien

II-G - Les objets

La plupart des langages récents (C++, Java, JavaScript…) ont pris le parti d'être orientés objet, par opposition aux langages plus anciens (Fortran, Basic, Pascal, C…) qui sont eux orientés action. Dans ces langages, la priorité est faite aux données (les objets), à leur structure (les propriétés), à la façon d'y accéder pour en extraire tout ou partie de l'information qu'elles contiennent (les méthodes), à la façon de les créer (les constructeurs - qui sont des méthodes particulières).

Cette organisation a un immense intérêt pour les langages utilisés sur internet, car s'exécutant sur les machines « clients », ils se doivent d'offrir un niveau de garantie de sécurité irréprochable afin qu'une applette (Java) ou qu'un script ne prenne pas des « libertés » préjudiciables au client (préservation de la confidentialité et de l'intégrité des données). En l'occurrence, ces langages et plus exactement leur compilateur (respectivement interpréteur), veillent à ce que seules les méthodes définies associées aux objets auxquels elles appartiennent et donc, sur lesquels elles s'exercent, puissent être activées. « *Ainsi, on ne peut pas faire n'importe quoi sur n'importe quoi !* »

En JavaScript, tout est objet ! On distingue toutefois parmi ceux-ci, deux types : les objets de type primitif et les objets de type composé. Les premiers sont ceux qui se résument à une valeur unique : les nombres, les booléens et les chaînes. Les seconds (tableaux, fonctions ou… objets) comportent plusieurs valeurs (propriétés), chacune bénéficiant d'un nom et d'une valeur, l'ensemble réuni en une seule entité (l'objet), elle aussi identifiée par un nom. L'accès à une propriété (pour lui affecter ou en lire sa valeur) devra obligatoirement préciser en premier lieu le nom de l'objet suivi du nom de la propriété concernée. Nous pourrons affecter à une propriété n'importe quel type d'objet (objets de type primitif ou de type composé).

Autre point d'intérêt différenciant les types primitifs des types composés : les premiers sont manipulés par valeur, tandis que les autres les sont par référence. Cela se justifie parfaitement si l'on considère que des objets de type composé peuvent contenir un très grand nombre de valeurs (les tableaux, par exemple) et qu'il serait donc totalement inefficace de les traiter par valeur, vu l'espace mémoire qui serait utilisé en pure perte.
La question peut toutefois se poser pour les chaînes, qui bien que pouvant comporter un nombre de caractères important, sont de type primitif. Effectivement, c'est une bonne question ! ;-))

> *Ce qui vient d'être dit mérite une mise au point ! Certains pourraient assimiler passage par valeur et passage par référence avec ce qu'ils ont déjà vu dans certains langages en ce qui concerne le passage de paramètres à des fonctions, à savoir : toute modification à l'intérieur d'une fonction d'une variable passée par valeur n'aura pas de répercussion à l'extérieur, tandis que pour une variable passée par référence, la modification sera retransmise.*
> *Ici, il s'agit de l'entité passée en paramètre. Quand on parle de référence, il s'agit d'un pointeur vers l'objet et non pas de l'objet lui-même. Si une ou plusieurs valeurs de cet objet sont modifiées via la référence, bien sûr, ces modifications affecteront l'objet. Par contre, si dans une fonction, cette référence est surchargée par une autre référence à un autre objet, cette modification demeurera locale. Pour ceux que cela perturbe, disons que le passage s'opère toujours par valeur, mais dans le cas des objets composés, cette valeur est une référence vers l'objet. C'est bon comme cela ?…*

Bien que nous n'ayons pas encore étudié les objets de type composé, voici un exemple simple à comprendre, qui illustre la précédente mise en garde. Nous allons considérer un tableau contenant les valeurs 1, 2, 3, 4 et 5. Ce tableau est passé en paramètre à une fonction qui dans un premier temps va substituer à la valeur 3 la valeur 100, puis affecter à ce tableau (via son pointeur de référence) un autre tableau contenant les valeurs 9, 8, 7, 6 et 5. Voici le programme et son exécution.

```
var Tab=new Array(1,2,3,4,5);
Modifier(Tab);
Afficher(Tab);
function Modifier(T){
  var AutreT=new Array(9,8,7,6,5);
  T[2]=100;
  T=AutreT;
  Afficher(T);
}
```

<img width="72" height="30" border="0" align="top" src="./images/Exec.jpg">

II-H - Les autres objets de type composé

Il convient de mentionner enfin deux types d'objets composés : les fonctions et les tableaux dont nous avons eu un avant-goût dans les exercices précédents. Nous ne rentrerons pas ici dans le détail de ces objets. Simplement, nous nous devions d'en parler, car ils constituent des données au même titre que les nombres ou les chaînes. En particulier, contrairement à tous les autres langages (dont Java), les fonctions ont à la fois un statut opérationnel exécutable, mais aussi un statut de données ce qui apporte une grande souplesse au langage en ce sens qu'elles peuvent être affectées à une variable, un élément de tableau, une propriété d'objet (méthode), passées en paramètre d'une autre fonction, etc. En ce qui concerne les tableaux, la particularité de JavaScript est d'autoriser ceux-ci à

contenir des objets de types différents. Un tableau peut ainsi contenir simultanément et indifféremment des nombres, des chaînes, des fonctions, des tableaux, bref n'importe quelles sortes d'objets.

Ajoutons enfin que fonctions et tableaux peuvent être définis sous une forme littérale comme nous avons pu le voir précédemment.

III - Expressions et opérateurs

Ce chapitre a pour but d'introduire les expressions et opérateurs utilisés en JavaScript. Les utilisateurs de Java, C ou C++ ne seront pas surpris par ce qui suit. Pour les autres, il faudra qu'ils prennent l'habitude de manipuler des opérateurs donnant à certaines instructions ou expressions des allures cabalistiques, mais ils se feront assez vite à ce formalisme qui ne présente, au bout du compte, aucune difficulté.

III-A - Les expressions

De façon générale, et quel que soit le langage utilisé, une expression est une suite de caractères pouvant être interprétée de façon à lui associer une valeur. JavaScript n'échappe pas à la règle !!!

Les expressions les plus simples sont celles ne faisant intervenir aucun opérateur. Elles ne contiennent donc qu'un littéral.

```
0.314e+1  // littéral numérique
"Expression"  // littéral chaîne
MonIdent  //Identificateur de variable
false  // littéral booléen
function(x){return x+1}  // littéral fonction
{Nom:"Terieur",Prenom:"Alex"}  // littéral objet
[3,"OK",true,[null,12],success]  // littéral tableau
```

Ces expressions simples peuvent être combinées entre elles (ou à d'autres), pour construire des expressions plus complexes, à l'aide d'opérateurs. Certains de ces opérateurs sont communs à différents types de littéraux (bien que leur sémantique soit différente), d'autres sont spécifiques. Par exemple, l'opérateur « + » peut être utilisé pour concaténer (mettre à la suite) deux chaînes de caractères ou pour additionner deux valeurs numériques. L'opérateur « == » est un opérateur booléen dont les deux opérandes peuvent être soit numériques, soit chaînes, soit booléens. Ils peuvent même être de types différents… On verra que dans le cas d'une comparaison nombre # chaîne, par exemple, JavaScript opère au préalable une conversion.

III-B - Les opérateurs

Ces opérateurs disposent de deux caractéristiques syntaxicosémantique : la priorité et le sens d'associativité (gauche-droite ou droite-gauche).
En voici la liste :

Opérateur	Priorité	Sens	Types opérandes	Fonction
. ( ) [ ]	15	GD	Objets / Propriétés Fonctions / Arguments Tableau / entier	Accès à une propriété Appel de fonction Indexage de tableau
++ -- - ~ !	14	DG Unaire	Nombre Nombre Nombre Entier Booléen	Pré ou post incrémentation Pré ou post décrémentation Moins (Inversion)

typeof delete void new			Indifférent Variable Indifférent Nom de constructeur	Complément à 1 binaire NON logique Rend le type de la donnée Rompt l'accès à une propriété Rend une valeur indéfinie Crée un nouvel objet
* / %	13	GD	Nombres	Multiplication Division Reste modulo
+ -	12	GD	Nombres	Addition Soustraction
+	12	GD	Chaînes	Concaténation
<< >> >>>	11	GD	Entiers	Décalage des bits à gauche Décalage à droite (extension signée) Décalage à droite (extension à zéro)
< <= > >=	10	GD	Nombres ou Chaînes	Inférieur à… Inférieur ou égal à… Supérieur à… Supérieur ou égal à…
== != === !==	9	GD	Indifférent	Test d'égalité Test d'inégalité Test d'identité Test de non identité
&	8	GD	Entiers	ET binaire
^	7	GD	Entiers	OU exclusif binaire
|	6	GD	Entiers	OU binaire
&&	5	GD	Booléens	ET logique
||	4	GD	Booléens	OU logique
? :	3	DG	Booléen / Indifférent / Indifférent (ternaire)	Opérateur conditionnel
=	2	DG	Variable / Indifférent	Affectation
*=, /=, %= +=, -= += <<=, >>=, >>>= &=, ^=, |=	2	DG	Variable / Nombre Variable / Nombre Variable / Chaîne Variable / Entier Variable / Entier	Affectation avec opération
,	1	GD	Indifférent	Évaluation multiple

III-B-1 - Priorité et sens d'évaluation

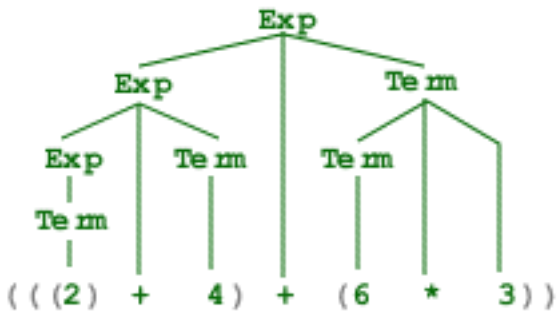
L'utilisation des opérateurs dans une expression impose, nous venons de le voir, des contraintes plus ou moins lâches sur les opérandes. Par exemple l'expression C1*C2 n'aura de sens que si C1 et C2 sont (ou peuvent se ramener) à des valeurs numériques. Par contre "C1"*"C2" n'aura pas de sens et provoquera donc une erreur, alors que "12"*"3" sera admis tout comme "12"*3 ou 12*"3". Par contre la valeur finale de l'expression, le résultat, sera d'un type bien précis. Dans le cas présent, quelle que soit l'écriture envisagée, la valeur du résultat sera le nombre 36 !!!

Ce qui vient d'être dit est valable, ainsi que nous allons le voir, pour bien d'autres opérateurs. Le seul opérateur jouant des tours inattendus est l'opérateur + dont la signification contient une ambiguïté.

```
"3" - 2     // soustraction : opérandes ramenés à des nombres
"3" / '2'   // division : opérandes ramenés à des nombres
'3' << '2' // décalage gauche : opérandes ramenés à des entiers
'3' == 3    // test : opérandes ramenés à des nombres
12 >> "2"   // décalage droite : opérandes ramenés à des entiers
3 % '2'     // modulo : opérandes ramenés à des entiers
"3" + 2     // concaténation : opérandes ramenés à des chaînes
3 & '2'     // ET binaire : opérandes ramenés à des entiers
"4" | 3     // OU binaire : opérandes ramenés à des entiers
```

Les caractéristiques des opérateurs (priorité ou sens d'associativité) ne sont pas une vue de l'esprit ou une convention prédéfinie qui les impose. **Cela découle directement de l'analyseur du langage** et en particulier, d'une part du niveau d'introduction des opérateurs dans la grammaire et, d'autre part, de la récursivité (gauche ou droite) des règles concernées. C'est dans ce sens que l'on a parlé plus haut de caractéristiques syntaxicosémantiques. Pour illustrer cela, considérons un exemple simple :

```
<Exp>::=<Exp> + <Term>
<Exp>::=<Term>
<Term>::=<Term> * <Num>
<Term>::=<Num>
<Num>::=<Num><Digit>
<Num>::=<Digit>
<Digit>::=0|1|2|....|9
```



Dans l'arbre de dérivation ci-dessus, nous avons négligé la partie purement lexicale qui n'apporte rien à notre propos. Néanmoins, cet arbre montre bien que seul le niveau d'introduction des opérateurs dans les règles syntaxiques définit leur priorité. **Plus bas est le niveau d'introduction d'un opérateur, plus il est prioritaire**. Si au lieu d'introduire l'opérateur + dans Exp et * dans Term, on avait procédé à l'inverse, + serait devenu prioritaire sur *. Par ailleurs, on peut constater que les règles sont en récursivité gauche : « *pour évaluer l'expression, il faut d'abord évaluer l'expression fille qui est à gauche pour laquelle il faut…* ». D'où un sens d'évaluation de la gauche vers la droite ! Nous avons rajouté en grisé des parenthèses pour bien montrer l'ordre des évaluations. Mais celles-ci sont tout à fait redondantes et la valeur de cette expression est bien 24.

III-B-2 - Les opérateurs arithmétiques

Nous ne nous étendrons pas ici sur les opérateurs arithmétiques habituels, + (addition), * (multiplication), / (division), - binaire (soustraction) ou - unaire (inversion). Nous préciserons toutefois que le % (modulo) ne se contente pas d'opérer sur des entiers, mais permet d'obtenir des restes de divisions rationnelles.
Par exemple,

```
5.6 % 3.21 = 2.3899999999999997.
```

Nous nous arrêterons, par contre, sur les opérateurs d'incrémentation (+1) ou de décrémentation (-1). Ces opérateurs, selon qu'ils sont placés avant ou après la variable concernée opèrent l'opération soit a priori, soit a posteriori. Par exemple l'instruction **j = i++;** est équivalente à **j = i; i = i+1;**. Si l'on avait voulu affecter à j la valeur de **i+1**, il aurait fallu écrire **j = ++i;** ce qui est équivalent à **i = i+1; j = i;**. Ce que l'on vient de dire s'applique de la même façon à l'opérateur **--**.

À noter que si l'utilisation de tels opérateurs intervient dans une chaîne de caractères, vous pouvez rencontrer un problème…
Ce problème est dû à la confusion qui existe hélas, dans JavaScript, entre le + de concaténation et les opérateurs arithmétiques comportant le signe +.

*Pour en revenir aux priorités des opérateurs : après avoir exécuté les instructions les instructions **i=3; j=7;** l'exécution de **i=i+++j** va affecter à i la valeur **10** tandis que j conserve la valeur **7**. En effet, ++ étant prioritaire sur +, l'évaluation va s'opérer selon **i=(i++)+j** et non **i=i+(++j)**. Donc i va bien prendre la valeur 10 **et la postincrémentation de i est perdue !!!!!** Cette opération se réalise en fait selon le mode suivant : **R=i; i=i+1; i=R+j** où R est un registre de travail. Dans le second cas qui aurait pu avantageusement s'écrire **i +=++j**, les valeurs finales auraient été **i : 11 et j : 8**.*

III-B-3 - Les opérateurs d'égalité et d'identité

<script language="JavaScript"> var S="Bien sur !!! \nQuelle que soit la chaine X,\na condition qu'elle soit differente de \"1\",\nLa valeur de l'expression bool\u00e9enne \"X\"==true sera toujours false."; var S6 = "L'execution de\nj=6;\nalert('j+1 vaut : '+++j)\n \n donne une erreur : \"invalid increment operand.\""; var S10 = "Valeur de l'expression \n true > \"0\" : " + (true>"0") + "\n \nLe booleen true est converti en la valeur 1."; var S11 = "Valeur de l'expression \n12 < \"Infinity\" : " + (12<'Infinity') + "\n \nLa cha\u00eene \"Infinity\" est bien interpr\u00e9t \u00e9e comme la valeur num\u00e9rique Infinity."; var S12 = "Valeur de l'expression \n \"13,6\" <= 17 : " + ('13,6'<=17) + "\n \nLa virgule n\'est pas accept\u00e9e."; var S13 = "Valeur de l'expression \nInfinity >= \"n'importe quoi\" : " + (Infinity>='n\'importe quoi') + "\n \nLa cha\u00eene ne peut \u00eatre convertie en un nombre\n \nComme quoi, l'infini n'est pas aussi grand qu'on pourrait le croire... ;-) "; var S14 = "Le resultat de l'evaluation est : \"6trouille\" (bof !...)\n \n En effet, l'operateur +, qu'il porte sur des chaines ou sur des nombres est gauche-droite.\nDonc on evalue en premier 4 + 2, ce qui donne bien 6, puis finalement, la concat\u00e9nation est op\u00e9r\u00e9e."; var S15 = "Effectivement, le resultat est bien true !\n \nLe parenthesage gauche-droite donne\n((\"4\" + 5) + 2), soit \"452\" et, apr\u00e8s conversion, il s'ensuit la comparaison 452 > 58."; function Exec(i) { switch (i) { case 1 : var s1 = ""; var s2 = "Bonjour"; s1 += s2; break; case 2 : var s1 = new String("Bonjour"); var s2 = "Bonjour"; break; case 3 : var s1 = new String("Bonjour"); var s2 = new String("Bonjour"); } var Out='s1 = ' + s1 + '\ns2 = ' + s2; if (s1==s2) Out += '\nEgalite : OK'; else Out += '\nEgalite : NON OK'; if (s1===s2) Out += '\nIdentite: OK'; else Out += '\nIdentite : NON OK'; alert(Out); } function Void(){ Indiv=new Object( ); Indiv.naiss=new Date(92,8,27); Indiv.copie=Indiv.naiss; Mois=Indiv.naiss.getMonth(); alert('naiss = ' + Mois); delete Indiv.naiss; (Indiv.naiss != void 0) ? alert('naiss2 = '+Indiv.naiss.getMonth()) : alert("Indiv.naiss est indefini"); Mois=Indiv.copie.getMonth(); alert('Indiv.copie = '+Mois); } </script>

L'opérateur **d'égalité** (==) renvoie une valeur booléenne. Le test s'opère différemment selon que les opérandes sont de type primitif ou de type composé. **Pour les types primitifs le test s'opère par valeur**, c'est-à-dire que le résultat du test sera true si et seulement si les arguments sont identiques (nombres ou booléens de même valeur, chaînes comportant une suite identique de caractères)ou peuvent l'être par conversion de chaînes ou de booléens vers des nombres. Dans le cas contraire, le résultat vaudra false. Pour les types composés (tableaux, objets, fonctions), **le test s'effectue par référence**, c'est-à-dire que le résultat sera true si les deux arguments font référence aux mêmes objets.

Il faut noter en outre les égalités suivantes :

- si les deux opérandes ont pour valeur null ou sont indéfinis, ils sont égaux ; (1)
- si l'un des opérandes a pour valeur null et l'autre est indéfini, ils sont égaux. (2)

Enfin, dans le cas où les arguments sont de types différents, les règles suivantes doivent être appliquées (reprenant en cela ce que nous avons vu **précédemment**) :

- si un opérande est une chaîne et l'autre un nombre, reconsidérer le test après avoir converti (si cela est possible) la chaîne en nombre ; (3)
- si l'un des opérandes est booléen, reconsidérer le test après avoir converti respectivement true en 1 et false en 0. (4)

On a vu précédemment que toute chaîne avait pour valeur logique true. En conséquence, on serait en droit de penser que l'expression booléenne "X"==true a toujours pour valeur true, quelle que soit la chaîne X.
Essayons donc d'évaluer "1" == true. <img width="20" height="20" border="0" align="absmiddle" name="Bouton7" src="./images/bouton_o.gif">
. **Tout paraît normal…**
Essayons "0"==true. <img width="20" height="20" border="0" align="absmiddle" name="Bouton8" src="./images/bouton_o.gif">
Tiens ! **La supposition précédente se révèle fausse !** Mais cela est bien cohérent avec les règles énoncées ci-dessus : par la règle (4) true interprété comme la valeur 1 vérifie donc bien l'égalité avec la chaîne "1" convertie elle-même en 1 par la règle (3) (mais pas la chaîne "0") **!!**
Dans ces conditions, essayons "0"==false. <img width="20" height="20" border="0" align="absmiddle" name="Bouton9" src="./images/bouton_o.gif">
. Correct ! false et "0" ramenés par la règle (4) [resp. (3)] à la valeur 0, on obtient bien pour valeur true.
Finalement, pouvez-vous répondre à la question du début : quelle est la valeur de l'expression booléenne "X"==true si la chaîne X est quelconque différente de "1" **?** Réfléchissez un peu avant de regarder la réponse.<img width="20" height="20" border="0" align="absmiddle" src="./images/bouton_o.gif">
L'opérateur **d'identité** (===) a un comportement quelque peu différent selon qu'il porte sur des objets de type primitif ou des objets de type composé.
Les objets de type composé : l'opérateur d'identité se comporte exactement comme l'opérateur d'égalité ;
Les objets de type primitif : l'opérateur d'identité se comporte exactement comme l'opérateur d'égalité, mais **il n'opère aucune conversion !**!!
Ainsi, tous les exemples du paragraphe précédent retourneront « false » à un test d'identité. De même « null » ne sera plus assimilé à l'état indéfini.

En définitive, **la relation d'identité sera satisfaite par ses opérandes si et seulement si ceux-ci sont de même type et ont même valeur.**

À la limite entre les types primitifs et composés, voyons comment se comportent ces opérateurs à l'aide de trois exemples où seule l'initialisation de deux variables, s1 et s2 diffère de l'un à l'autre. La suite, identique pour les trois, teste l'égalité et l'identité de ces deux variables…

```
var s1 = "";
var s2 = "Bonjour";
 s1 += s2;
```

```
Out = 's1 = ' +
 s1+'\ns2 = ' + s2;
if
 (s1==s2) Out += '\nEgalite : OK';
else
 Out += '\nEgalite : NON OK';
if (s1===s2) Out += '\nIdentite : OK';
else
 out += '\nIdentite : NON OK';
```

```
var
 s1 = new String("Bon
var s2 = "Bonjour";
```

```
var
 s1 = new String("Bon
var
 s2 = new String("Bon
```

```
<a onclick="Exec(1);return false"
onmouseover="status='Voyons
ce que cela donne...';return
true" href=""><span
class="programme">Alert(out)</
span></a>
<a onclick="Exec(2);return false"
onmouseover="status='Voyons
ce que cela donne...';return
true" href=""><span
class="programme">Alert(out)</
span></a>
<a onclick="Exec(3);return false"
onmouseover="status='Voyons
ce que cela donne...';return
true" href=""><span
class="programme">Alert(out)</
span></a>
```

Le premier exemple montre le comportement des opérateurs d'égalité et d'identité sur deux chaînes, le second, sur une chaîne et un objet chaîne, le troisième sur deux objets chaînes (nous reviendrons plus loin sur la distinction chaîne # objet chaîne).

III-B-4 - Les opérateurs de comparaison

Nous ne nous étendrons pas sur les **opérateurs de comparaison** dans le cas où les opérandes sont numériques. Chacun sait qu'ils vont être utilisés dans les expressions booléennes en général (comme les précédents) et en particulier dans les instructions **if** ou **while**, for ou do que nous verrons ensuite. Précisons néanmoins qu'ils n'admettent pour opérandes que des nombres ou des chaînes de caractères (et éventuellement, mais l'intérêt est médiocre, des booléens). Dans le cas d'opérandes de type différents, il y a tentative de conversion vers un nombre. Si la conversion est possible et la comparaison par voie de conséquence, le test est évalué, sinon l'évaluation est false quel que soit l'opérateur.

Évaluer les expressions…	*Résultat*
`true >= "0"`	`<a onclick="alert(S10);return false" onmouseover="status='Conversion booléen -> entier...';return true" href=""><img width="20" height="20" border="0" name="Bouton10" src="./images/bouton_o.gif"></a>`
`12 < "Infinity"`	`<a onclick="alert(S11);return false" onmouseover="status='La chaîne vaut l \'infinité...';return true" href=""><img width="20"`

	height="20" border="0" name="Bouton11" src="./images/bouton_o.gif">
"13,6" <= 17	<img width="20" height="20" border="0" name="Bouton12" src="./images/bouton_o.gif">
Infinity >= "n'importe quoi"	<img width="20" height="20" border="0" name="Bouton13" src="./images/bouton_o.gif">

Nous avons précédemment étudié le comportement des opérateurs d'égalité et d'identité lorsque les opérandes sont des chaînes. Les opérateurs de comparaison sont eux aussi utilisés pour les chaînes de caractères. Dans ce cas, la comparaison s'opère suivant l'ordre lexicographique. Par exemple, "Pascal" < "Perl" est évalué à true. Lorsque les deux chaînes comparées ont une partie commune en tête, c'est la longueur des chaînes qui sert pour l'évaluation. Par exemple, "Java" >= "JavaScript" est évalué à false. Un autre opérateur reçoit pour opérandes des chaînes de caractères : +, opérateur de concaténation. Cet opérateur a pour effet de construire une chaîne en réunissant les chaînes opérandes. Par exemple : "JavaScript, "+'c\'est '+"chouette" a pour valeur "JavaScript, c'est chouette".

Il est important de noter qu'en cas d'ambiguïté, ***priorité est donnée aux opérateurs de chaîne par rapport aux opérateurs de nombre***. Ainsi, "12"+7 prend pour valeur, non pas 19, mais "127" ! Dans le cas on l'on veut opérer un calcul de ce type, il faudra auparavant forcer la conversion de la chaîne en nombre. Ainsi, si la variable Str contient la chaîne "12", le calcul arithmétique précédent devra s'écrire Str * 1 + 7. La multiplication force l'évaluation à convertir Str en un nombre avant de l'additionner à 7. Le résultat est alors celui attendu : 19.

Inversement, les opérateurs de comparaison privilégient la conversion vers les nombres. Ainsi "12" > 7 sera bien évalué par true, preuve qu'il y a bien eu conversion de "12" vers 12, car la comparaison lexicographique aurait donné false.

Voyons à présent si vous avez bien assimilé ce que l'on a dit depuis le début de ce chapitre… Pouvez-vous donner la valeur de 4+2+"trouille" ?

 <img width="17" height="17" border="0" align="absmiddle" name="Bouton14" src="./images/bouton_o.gif">
Bien ! Et maintenant, vous n'aurez donc plus aucune difficulté pour évaluer l'expression "4"+5+2 > 58 ?!…

 <img width="17" height="17" border="0" align="absmiddle" name="Bouton15" src="./images/bouton_o.gif">

III-B-5 - Les opérateurs logiques

Les opérateurs logiques sont utilisés dans des expressions booléennes complexes mettant en jeu plusieurs variables. Pour ceux qui connaissent Pascal, ce langage comporte les d'opérateurs AND, OR, NOT qui réclament l'évaluation complète de l'expression dont ils font partie. Certaines versions (ou options de compilation) autorisent la génération d'un code optimisé qui retourne une évaluation de l'expression dès qu'elle est possible sans que tous les termes ou facteurs de ladite expression aient forcément été évalués. Les opérateurs logiques de JavaScript sont optimisés.

- **L'opérateur ET (&&)**, est-il besoin de le rappeler, retourne la valeur true si et seulement si ses deux opérandes ont eux-mêmes pour valeur true ? Il convient toutefois d'insister sur l'aspect optimisation. En effet, dans le cas de l'expression 3>4 && "4"+5+2 > 58, l'opérateur && étant gauche-droite, l'évaluation va tout d'abord s'opérer sur 3>4. Ce facteur ayant pour valeur false, point n'est besoin d'évaluer le second opérande puisque d'ores et déjà, on sait que la valeur de l'expression sera false. Si, par contre, le premier facteur avait été évalué à true, l'évaluation se serait poursuivie sur le deuxième opérande.


Certains programmeurs affectionnent particulièrement ce type de fonctionnement pour faire des effets de style. Par exemple, au lieu d'écrire if(val1==val2) alert('OK'), ils préfèrent var Bool=(val1==val2) && alert('OK'). Ce faisant, si val1 n'est pas égal à val2, Bool prendra la valeur false sans que alert('OK') n'ait été évalué et donc exécuté. Si par contre val1 est bien égal à val2, le deuxième facteur sera évalué et donnera donc lieu à l'apparition de la fenêtre d'alerte. Ces deux écritures sont donc équivalentes. Mais, outre le fait que l'intérêt est difficile à saisir (!!!), cette façon de faire est totalement déconseillée. En effet, si le deuxième facteur n'est pas évalué, cela pourra provoquer une erreur de programmation si l'on n'y prend garde. De façon générale, prenez l'habitude de ne pas utiliser dans une expression booléenne des expressions à effet de bord (appel de fonction, incrémentation, décrémentation…) au-delà du premier opérande.

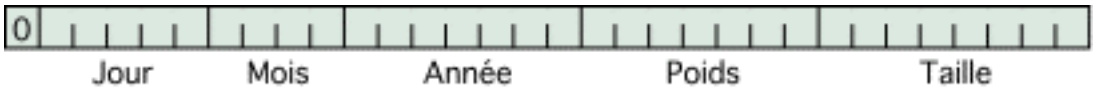
- **L'opérateur OU (||)** qui ne renvoie une valeur **false** que si ses arguments ont tous deux pour valeur false fonctionne de la même façon que l'opérateur précédent. Si l'évaluation du premier terme se révèle être **true**, l'expression aura pour valeur **true** sans avoir à évaluer la suite. Si par contre, le premier terme a la valeur **false**, le second sera évalué.

- **L'opérateur NON (!).** Rien de particulier à dire de cet opérateur unaire, si ce n'est qu'il délivre une valeur de vérité inverse de celle de son opérande.

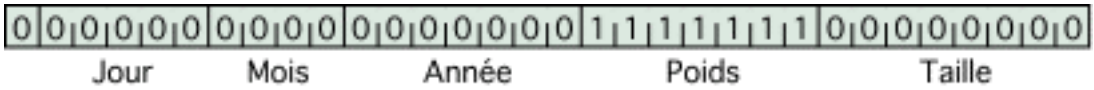
III-B-6 - Les opérateurs binaires

Nous avons montré plus haut quelques expressions utilisant des **opérateurs binaires**. Ils nécessitent des opérandes numériques (ou ayant subi une conversion adéquate) et interviennent directement sur la configuration binaire du codage en complément à 2 sur 32 bits de ceux-ci. Pour ceux qui ne comprennent rien à ce charabia, ce serait trop long et mal venu ici de l'expliquer. Ils pourront si cela les intrigue consulter mon cours à ce sujet.
Ces opérateurs interviennent dans des utilisations de bas niveau sur des représentations binaires. Cela autorise des réalisations spectaculaires d'efficacité comme l'algorithme de Transformée Rapide de Fourrier (butterfly). Mais pour ceux qui ne se sentent pas concernés par ce genre de subtilité, ils peuvent sauter ce paragraphe, il ne leur en sera pas tenu rigueur…

- **L'opérateur binaire ET (&)** est surtout utilisé pour mettre en place des masques. Un masque est un nombre binaire comportant des 1 aux emplacements où on veut récupérer l'information d'autres nombres. Par exemple, supposons que l'on code sur 32 bits des informations individuelles : Jour de naissance (5 bits -[0,31]-), Mois (4 bits -[0,15]-), Année (7 bits -[0,127]-), Poids (7 bits -[0,127]-), Taille en cm (8 bits -[0,255]-). On suppose que le bogue de l'an 2000 est passé, et qu'il n'y a pas de sumotori parmi ces personnes. Pour le reste, les contraintes sont supportables… ;-)



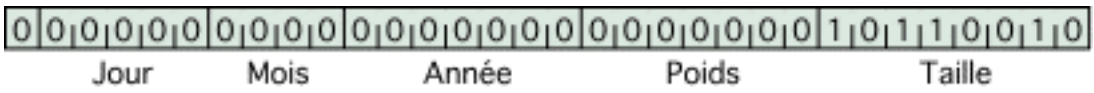
Supposons à présent que pour l'ensemble de ces individus on veuille faire une étude statistique sur le poids. Pour chaque personne nous allons donc devoir récupérer les sept bits utilisés pour coder cette caractéristique. Pour cela, nous allons utiliser le masque suivant :



(I)

Pour chaque bit autre que les bits de poids, le résultat sera 0, puisque 0 (du masque) & X a pour valeur 0 quel que soit X. Pour le champ Poids, pour un bit de donnée à 1, le résultat sera 1 (puisque 1 & 1 = 1) et pour un bit de donnée à 0, le résultat sera 0 (puisque 0 & 1 = 0). Ainsi on récupère pour résultat les seuls bits du champ Poids. Il ne restera alors plus qu'à opérer un décalage à droite de 8 bits (ce que nous allons voir) pour obtenir la valeur effective.

- **L'opérateur binaire OU (|)**, pour reprendre l'exemple ci-dessus, va nous permettre d'initialiser des champs. Supposons que le codage des caractéristiques de chaque individu soit initialisé à 0 et que nous ayons déjà chargé, dans la variable concernée, la taille de l'individu (178 cm). La variable a donc l'allure suivante :



(I)

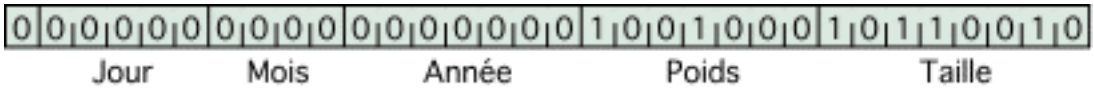
Affectons par exemple à cet individu I la valeur de son poids : 72 kg. Un mot de 32 bits va donc être initialisé à la valeur décimale 72, soit en binaire :

(Poids)

On procède ensuite à un décalage à gauche de 8 bits (place occupée par le champ « Taille »), pour obtenir :

(Poids)

Il ne reste plus, ensuite, qu'à opérer un OU entre cette valeur et la variable contenant déjà le champ « Taille ». Chacun des bits du résultat prendra pour valeur 1 chaque fois que le bit correspondant de la variable I ou de la variable Poids (ou des deux, *situation qui ne se présente pas dans notre exemple*) aura pour valeur 1 et 0 sinon. Le résultat final est donc :



(I)

- **L'opérateur binaire OUX (^)** délivre en résultat la valeur 1 chaque fois que l'un ou l'autre (mais pas les deux) des bits correspondants des opérandes est à 1. On peut donner une application de cet opérateur

dans la recherche du nombre minimum d'arêtes à suivre pour relier deux sommets d'un hypercube (certains reconnaîtront ici le calcul de la distance de Hamming). Soit donc deux points A et B dans un espace [0,1]3.

Le point A est codé 101 et le point B est codé 110. En effectuant un OU exclusif entre ces coordonnées, on obtient :

```
Res = 101 ^110 (011).
```

Il ne reste plus qu'à compter le nombre de bits à 1 du résultat pour déterminer le nombre d'arêtes qui doivent être traversées pour relier les deux points. Cela peut se faire avec le petit script suivant :

```
var S=0;          //Initialisation compteur

while (Resl>0) { // Tant qu'il reste 1 bit à 1..

  S+=(Res & 1);  // Ajout de la valeur du bit

  Res>>>=1;      // Décalage à droite

}
```

- **L'opérateur binaire NON (~)** est un opérateur unaire se plaçant avant son opérande. Il a pour fonction d'inverser l'ensemble des bits de l'opérande. Autrement dit, il délivre son complément à 1.

- **L'opérateur binaire de décalage à gauche (<<)** comporte deux opérandes : à gauche, le nombre à traiter et à droite le nombre de bits de décalage. Ce dernier doit être une valeur comprise entre 0 et 31. Si d'aventure il est plus grand, c'est la valeur modulo 32 qui sera prise en compte. La fonction de cet opérateur est de décaler vers la gauche l'ensemble des bits du premier opérande d'autant de places que le second l'indique en remplissant à droite par des 0 et en provoquant la perte des bits poids forts. Le fait de décaler d'un bit à gauche provoque une multiplication du nombre par 2. Le fait de le décaler de n bits provoque donc une multiplication par 2n.

- **L'opérateur binaire de décalage à droite (>>) signé** a les mêmes caractéristiques que le précédent quant à ses opérandes. Sa fonction consiste à décaler vers la droite l'ensemble des bits du premier opérande d'autant de places que le second l'indique. Selon que le bit poids fort de la valeur d'origine du premier opérande était à 1 ou à 0, le remplissage à gauche se fait par des bits à 1, respectivement, à 0. Les bits poids faibles sont perdus.

- **L'opérateur binaire de décalage à droite (>>>) non signé** a les mêmes caractéristiques que les précédents quant à ses opérandes. Sa fonction est identique à celle de l'opérateur de décalage à droite signé hormis le fait que le remplissage des bits poids forts s'opère systématiquement par des 0.

Dans les exercices qui suivent, essayez d'évaluer vous-même les expressions proposées avant de regarder le résultat (n'hésitez pas à vous référer au **tableau** du début, en particulier pour la priorité des opérateurs).

Évaluer les expressions…	*Résultat*
12-4%3!="1"+(0xD^014)	

	<img width="20" height="20" border="0" name="Bouton2" src="./images/bouton_o.gif">
0?3:null?'6':"1"?100:true?x:12	<img width="20" height="20" border="0" name="Bouton3" src="./images/bouton_o.gif">
-2+3^6>>1	>1 : \n \n' + (-2+3^6>>1));return false;" onmouseover="status='Opérateurs binaires...';return true" href=""><img width="20" height="20" border="0" name="Bouton4" src="./images/bouton_o.gif">
~-~3&-~4<<2	<a onclick="alert('Valeur de l\'expression \n~-~3&-~4<<2 : \n \n' + (~-~3&-~4<<2));return false;" onmouseover="status='Opérateurs binaires...';return true" href=""><img width="20"

	height="20" border="0" name="Bouton5" src="./images/bouton_o.gif">

III-B-7 - Les opérateurs d'affectation

On distingue deux types d'opérateurs d'affectation : l'opérateur d'affectation simple et ceux d'affectation avec calcul. Dans les deux cas, le premier opérande doit être une variable.

- Pour ce qui concerne l'**affectation simple**, le deuxième opérande est indifférent (variable ou littéral de type quelconque). Comme nous l'avons vu dans le chapitre précédent, le type de l'argument affecté détermine le type de l'argument récepteur. Nous pouvons constater dans le **tableau** récapitulatif que cet opérateur est droite-gauche. Il est essentiel d'insister sur le fait que l'affectation est donc effectuée via un opérateur. Cela va avoir des conséquences dans son utilisation puisqu'une affectation pourra donc être considérée comme une expression. Ainsi, nous allons pouvoir ainsi procéder à des **affectations multiples**.

Par exemple, l'expression a=b=c=d=3. Celle-ci va être analysée (en faisant apparaître le parenthésage : a=(b=(c=(d=3))). La première expression évaluée est d=3 qui non seulement a pour fonction d'affecter la valeur 3 à d, mais aussi de retourner la valeur 3 elle-même. Ainsi l'évaluation se poursuit sur c=3, etc. À l'issue de l'évaluation, les quatre variables a, b, c, et d auront ainsi été affectées à 3. Autre possibilité dérivée de ce qui vient d'être dit : (a=b)>3 constitue une expression booléenne tout à fait correcte qui à la fois affecte la valeur de b à a et vérifie si cette valeur est supérieure à 3.

- **Les opérateurs d'affectation avec calcul** réclament pour le deuxième opérande un type précis dépendant de l'opérateur concerné comme précisé dans le **tableau** récapitulatif. Ces opérateurs comportent deux caractères, le premier précisant l'opération le second étant le signe d'affectation. La fonction de ces opérateurs est d'effectuer l'opération précisée sur le premier et le second opérande, puis d'affecter le résultat au premier opérande. On a donc les équivalences suivantes :

Opérateur	Exemple	Signification	Résultat
+= (Num)	a=12; a+=3	a=a+3	a=15
+= (Chaîne)	a="12"; a+="3"	a=a+"3"	a=123
-=	a=12; a-=3	a=a-3	a=9
*=	a=12; a*=3	a=a*3	a=36
/=	a=12; a/=3	a=a/3	a=4
%=	a=13; a%=3	a=a%3	a=1
&=	a=12; a&=3	a=a&3	a=0
| =	a=12; a |=3	a=a|3	a=15
^=	a=14; a ^=3	a=a^3	a=13
<<=	a=12; a<<3	a=a<<3	a=48
>>=	a=-12; a>>=2	a=a>>2	a=-3
>>>=	a=-12; a>>>=2	a=a>>>2	a=1073741821

III-B-8 - Opérateurs divers

Nous allons voir ici les derniers opérateurs (mis à part ceux dédiés aux tableaux et aux objets) que propose JavaScript.

- Un opérateur dont vous apprendrez à apprécier la grande utilité est l'**opérateur conditionnel**. Celui-ci est un opérateur comportant trois opérandes (on dit qu'il est ternaire) et peut être utilisé en de multiples circonstances, chose que ne permet pas l'instruction if/else avec laquelle on pourrait faire le rapprochement. Sa syntaxe est la suivante : <*expression booléenne*> ? <*expression1*> : <*expression2*>.

```
Son fonctionnement : □ tout d'abord, l'expression booléenne (premier opérande) est évaluée.

                                □ si cette évaluation est true, l'expression résultat prend pour valeur
 l'évaluation de l'expression1

                                  (2e opérande)

                                □ sinon, l'expression prend pour valeur l'évaluation de l'expression2
 (3e opérande)
```

Avec cet opérateur, on peut, par exemple, construire une chaîne de caractères en choisissant des portions, prévoir dans la construction même de pages web plusieurs options, affecter différemment une variable selon sa valeur précédente ou autre, construire un lien vers une autre page s'adaptant à des informations recueillies, bref, donner une flexibilité très grande au travers d'une écriture particulièrement concise.

<a href="" OnClick="b=prompt('Donnez une valeur...');(b<=10) ? window.open('./fichiers/Inferieur.html','Inferieur','width=700,height=200') : window.open('./fichiers/Superieur.html','Superieur','width=700,height=200');return false"> Essai
Le choix de page de cet essai est réalisé par :

b=prompt('Donnez une valeur…');(b<=10) ? window.open('Inferieur.html') : window.open('Superieur.html')

- **L'opérateur ", "** est particulier… En fait, sa principale utilisation apparaît dans les boucles bornées (for) que nous aborderons plus loin. Disons simplement que grâce à lui et une telle boucle peut porter sur plusieurs variables qui évoluerons simultanément à chaque pas de la boucle.

Les opérateurs suivants n'ont pas le graphisme du type de ceux que l'on a vu jusqu'à présent. Ce sont en fait des mots réservés du langage, c'est-à-dire des suites de caractères alphabétiques qui ne peuvent être choisis comme identificateurs de variables.

- **L'opérateur typeof** est un opérateur unaire placé avant son opérande. La valeur retournée est une chaîne de caractères indiquant le **type de l'opérande**. Cette évaluation peut donc délivrer : « number », « string », « boolean », « object », « array », « function », « null » ou « undefined ».

- **L'opérateur de création d'objets, new** reçoit en opérande un **constructeur d'objets** et délivre une instance de ce type d'objet. Nous avons déjà vu des utilisations de cet opérateur au **chapitre** précédent, aussi, nous ne nous attarderons pas sur celui-ci.

- **L'opérateur delete** permet de supprimer une propriété d'un objet (ou un élément de tableau) passée en argument depuis la version 1.2 de JavaScript. Il faut insister sur le fait que le *delete* de JavaScript n'a rien à voir avec celui de C ! Nous avons déjà précisé que la récupération d'espace mémoire en JavaScript était un processus automatique. Ici, nous supprimons seulement une propriété. Il faut néanmoins indiquer que dans une page web, tout est propriété d'un objet window. Donc un objet défini par le créateur d'une page web constitue une propriété de l'objet window et peut donc à ce titre être supprimé… sauf s'il a été créé en utilisant var auquel cas la suppression n'aura pas lieu. Par contre, l'objet window lui-même ne pourra jamais être supprimé… et cela vaut mieux ;-)

- **L'opérateur void** comporte un argument indifférent (variable, expression, littéral…). Il renvoie systématiquement la valeur undefined. Nous avons vu la différence subtile qu'il y avait entre le littéral null et la valeur undefined et nous avons précisé que undefined ne pouvait pas être employé en tant que littéral. L'utilité essentielle de l'opérateur void est de pallier ce défaut. Appliquons cela à un petit exemple.
  Soit le programme :

```
Indiv=new Object( );
Indiv.naiss=new Date(92,8,27);
var Indiv.copie=Indiv.naiss;
Mois=Indiv.naiss.getMonth();
alert('naiss = ' + Mois);
```

```
delete Indiv.naiss;
(Indiv.naiss != void 0) ?
   alert('naiss2 = '+Indiv.naiss.getMonth()) :
   alert("Indiv.naiss est indefini");
Mois=Indiv.copie.getMonth();
alert('Indiv.copie = ' + Mois);
```

 <img width="17" height="17" border="0" name="Bouton15" src="./images/bouton_o.gif">

Ce petit exemple a plusieurs utilités : il illustre ce qui a été dit au chapitre 2 sur les objets et ce qui vient de l'être au sujet de l'opérateur delete. Celui-ci a bien supprimé la propriété naiss de l'objet Indiv, c'est-à-dire **l'accès** vers l'objet Indiv.naiss. Mais cet objet n'a lui-même pas été supprimé, car un autre accès, copie pointait vers lui. L'accès à l'objet Indiv.naiss n'existant plus, le test utilisant l'opérateur void nous le confirme et c'est bien alert("indefini") qui est exécuté.

IV - Les instructions

<script language=JavaScript> function ExecIf(){ V1 = V2 = 12; V3 = 16; if (V1 >= V2) if (V2 >= V3) alert(V3 + ' n\'est pas strictement superieur a ' + V1) else alert(V1 + ' est strictement inferieur a ' + V2) } function xcfor(){ var S = ""; for(var I = 0 ; I < 10 ; I++) S += I; alert("La boucle a construit la chaine : " + S); } function xcfor2(){ var arret=true,S="",Add='abc'; alert("Je demarre"); for(var i=1;arret;i+=3){ alert("Dans for, i = "+i); S+=Add; arret= !(S.length>8); } alert("J'en suis sorti !!!") } function xcfor3(){ var S = ""; for(var I = -1 ; I++ < 9 ; S += I); alert("La boucle a construit la chaine : " + S); } function xcfor4(){ var S = ""; for(var I = 0 ; I < 10 ; S += I ++); alert("La boucle a construit la chaine : " + S); } function xcfor5(){ var S = ""; for(var I = 1 ; I < 6 ; I++) for(var J = 2 ; J < 7 ; J+=2) for(var K = 3 ; K < 8 ; K+=3) S+=""+I+J+K; alert("Chaine construite : " + S); } function xcfor6(){ var S = ""; for(var I=1, J=2, K=3; I<6, J<7, K<8; I++, J+=2, K+=3) S += "" + I + J + K; alert("Chaine construite : " + S); } function PrObj(){ S="Proprietes de window :\n"; for(var i in window) S+=i+", "; alert(S); } function Break(){ S = "Valeurs successives d'index :\n"; Boucle1:for(var i=0;i<5;i++){ Boucle2:for(var j=0;j<6;j+=2){ if (j>2) break; if (i==1) break Boucle2; if (i==4) break Boucle1; S += "i = "+i+" j = "+j+"\n"; } } S += "En sortie : i = "+i+" j = "+j+"\n"; alert(S); } var CeLien=""; function Conti(){ var S = "Valeurs successives d'index :\n"; Boucle:for(var i = 0 ;i < 4; i++){ for(var j = 0; j < 5; j+=2){ if (j == 2) continue; if (i == 1) continue Boucle; S += "i = " + i + " j = " + j + "\n"; } } S += "En sortie : i = " + i + " j = " + j + "\n"; alert(S); } function ChgTxt(){ window.document.Formul.textfield.value=CeLien; // idem window.document.links[14] // adapté pour le site Developpez.com qui ajoute des liens dont je n'ai pas la maitrise } function NwProp(i){ var Ordinateur=new Object(); Ordinateur.Systeme=Ordinateur.Memoire=Ordinateur.Disque=null; with (Ordinateur){ S = "Proprietes de Ordinateur :\n"; Systeme = "WinXP"; Memoire = 1024; Disque = 80000; switch (i){ case 1 : Processeur = "Pentium IV";break; case 2 : Ordinateur.Processeur = "Pentium IV"; } } for (var i in Ordinateur) S += i + " : " + Ordinateur[i] + ",\n"; alert(S); } function enumimp(x){ for(i=8;i<12 ;i++) S+="("+x+","+i+")"; S+="\n"; } function enumexp(x){ for(var i=8;i<12 ;i++) S+="("+x+","+i+")"; S+="\n"; } </script>

Mis à part lorsqu'elles contiennent des opérateurs induisant des effets de bord (en particulier, les opérateurs d'affectation et d'incrémentation), les expressions qui ont été présentées au précédent chapitre se contentent de produire une valeur, mais n'ont aucun rôle actif dans l'exécution d'un programme. L'exécution proprement dite est réalisée grâce au répertoire d'instructions dont dispose le langage. L'objet de ce nouveau chapitre est d'introduire les diverses instructions utilisables sous JavaScript. Mais avant tout, il convient de préciser que JavaScript dispose aussi de deux types d'instructions :

- les instructions simples qui se résument à des instructions atomiques comme les instructions expression (s+=12; ou index--; ou encore Res=(index > 3) ? "OK" :'KO';)… de même que les appels de fonctions (prompt('Voulez-vous un cafe ?');)

- les instructions composées qui constituent donc un bloc composé de plusieurs instructions et bordé par deux accolades {…}.

IV-A - Les tests

Les instructions de test disponibles en JavaScript sont celles que l'on connaît pour beaucoup d'autres langages. Il s'agit de l'instruction if et de l'instruction **if…else**. La syntaxe de l'instruction **if** est la suivante :

```
if (<expression booléenne>) <instruction>
```

Son fonctionnement est le même que pour les autres langages : si le résultat de l'évaluation de l'expression est **true** ou peut être converti vers **true**, alors l'instruction est exécutée ; si le résultat de l'évaluation de l'expression est **false** ou peut être converti vers **false**, alors l'instruction n'est pas exécutée. Il convient de noter dans la syntaxe, l'obligation d'encadrer l'expression booléenne entre deux parenthèses. Par ailleurs, il est bien évident que l'instruction conditionnelle peut être simple ou composée.

La seconde forme de l'instruction est donc le **if…else** dont la syntaxe est la suivante :

```
if (<expression booléenne>) <instruction 1> ; else <instruction 2>
```

Son exécution, s'il est besoin de le préciser, est le suivant : si le résultat de l'évaluation de l'expression est **true** ou peut être converti en **true**, alors l'instruction 1 est exécutée ; si le résultat de l'évaluation de l'expression est **false** ou peut être converti vers **false**, alors c'est l'instruction 2 qui 'est exécutée. Les remarques concernant l'instruction **if** s'appliquent à l'instruction **if…else**. De plus, on remarquera dans la syntaxe le point-virgule précédant **else**. Cela ne sera pas sans surprendre ceux qui connaissent le langage Pascal où justement ce caractère placé à cet endroit était totalement en défaut par rapport à la syntaxe.

Pour être précis, ce caractère « ; » n'est absolument nécessaire que dans le cas d'une écriture linéaire d'une telle instruction. En effet, en JavaScript, **le retour à la ligne vaut le « ; »** (sauf quand l'instruction n'est pas syntaxiquement finie). Ainsi, une écriture du style :

```
if (<expression booléenne>)
  <instruction 1>
else
  <instruction 2>
```

est tout à fait correcte et admise par la syntaxe. Le retour à la ligne après l'expression booléenne appelle une suite constituée par l'instruction 1. Par contre, après celle-ci, l'instruction **if** est syntaxiquement achevée et donc, le retour à la ligne vaut le « ; » nécessaire à cet endroit. Conséquence de ce qui vient d'être dit : même si le **else** est, comme ici, suivi d'un retour à la ligne, l'instruction placée à la ligne suivante fera partie intégrante de l'instruction **else**. Si l'on veut que l'instruction qui lui est associée soit une instruction vide (ce qui peut sembler bizarre, car on peut alors se poser la question de l'utilité du **else**), il faudra prévoir effectivement un « ; » à la suite.

Les instructions **if** peuvent être **enchâssées**. Dans ce cas, si l'une ou plusieurs d'entre elles sont complétées d'instructions **else**, il apparaît souhaitable de connaître très précisément la syntaxe afin d'éviter toute erreur d'interprétation dans l'évaluation. Considérons, par exemple, l'écriture suivante :

```
V1 = V2 = 12;
V3 = 16;
if (V1 >= V2)
   if (V2 >= V3)
      alert(V3 + ' n\'est pas strictement superieur a ' + V1)
else
   alert(V1 + ' est strictement inferieur a ' + V2)
```

<img width="29" height="29" border="0" name="Bouton0" src="./images/bouton_o.gif">
 Exec

Étudiez ce programme. A priori il semble cohérent avec le contenu des fenêtres **alert** qui sont prévues !… Et pourtant ! … Essayez de l'exécuter.
L'indentation qui a été choisie représentait ce que l'utilisateur voulait signifier. En ce sens, effectivement, tout semble correct. Le problème vient du fait que le **else** se réfère au if le plus proche. Un moyen simple de se rappeler cela consiste à dire que **if** est à **else** ce que la parenthèse gauche est à la parenthèse droite, dans une expression correctement parenthésée, bien sûr ! Et là apparaît peut-être l'intérêt d'une instruction vide après un **else…** Il est néanmoins préférable d'utiliser la structure de bloc, ce qui donne :

```
V1 = V2 = 12;
V3 = 16;
if (V1 >= V2){
   if (V2 >= V3)
      alert(V3 + ' n\'est pas strictement superieur a ' + V1)}
else
   alert(V1 + ' est strictement inferieur a ' + V2)
```

IV-B - Les boucles

JavaScript, comme la plupart des langages de programmation, comporte deux types de boucle : les **boucles non bornées** et les **boucles bornées**.

Les premières sont celles dont on ne connaît pas le nombre de fois qu'elles seront exécuté, car le test de fin de boucle est une expression booléenne mettant en jeu des informations non connues, même de façon formelle, au moment de l'écriture du programme.

Les autres sont celles pour lesquelles, au moment de la rédaction du programme, on sait (au moins de façon formelle) combien de fois elles doivent boucler (**boucles muettes** où la valeur de l'indice de boucle peut n'avoir aucun rôle dans les instructions qui en font partie) ou à quelle valeur on doit commencer le calcul, à quelle valeur il se termine et quel est le pas de variation entre chaque itération (**boucles expressives** où la valeur d'indice a une fonction dans les calculs).

La première instruction de boucle non bornée est l'instruction while dont la syntaxe est la suivante :

```
while (<expression booléenne>)
  <corps de boucle>
```

Le corps de boucle peut se réduire à une seule instruction ou, au contraire, en comporter plusieurs. Dans ce cas, bien sûr, le bloc sera délimité par **{** et **}**. Comme pour les autres langages que vous connaissez peut-être, le fonctionnement de la boucle while est le suivant :

1 Évaluation de l'expression booléenne d'entrée ;
2 Si cette évaluation délivre la valeur **false**, la boucle est terminée et l'exécution se poursuit en séquence vers 4.
Si le résultat de l'évaluation est **true**, le corps de boucle est exécuté ;
3 Retour en 1 ;
4 Suite du programme.

En conséquence, une boucle while peut ne pas être exécutée dans le cas où le test d'entrée se révèle toujours faux. A contrario et comme dans tous les autres langages, la logique de programmation réclame que si l'entrée dans la boucle est possible, l'environnement servant à l'évaluation de l'expression booléenne subisse une modification dans le corps de boucle, faute de quoi le programme sera dans une boucle infinie.

La seconde instruction de boucle non bornée est le **do/while** qui, pour les habitués de Pascal, correspond, d'un point de vue logique, au **repeat/until**. Sa syntaxe est la suivante :

```
do
<corps de boucle>
while (<expression booléenne>);
```

Par rapport à l'instruction précédente, il est évident de constater que cette boucle s'exécute au moins une fois et ce n'est qu'après cette exécution qu'est évaluée l'expression booléenne afin de déterminer si la boucle doit être à nouveau exécutée ou pas. Par rapport au **repeat/until** de Pascal, on notera toutefois l'inversion du test puisqu'ici la boucle se maintient tant que l'expression à une valeur **true** alors qu'en Pascal le processus se réitère tant que l'expression a une valeur **false**.

La boucle bornée est la boucle for dont la syntaxe est décrite ci-dessous :

```
for([var] <ident>=<init>;<test de fin>;<opération>)
<corps de boucle>
```

Le fait de faire précéder l'identificateur d'index de boucle du mot réservé **var** est optionnel. Pour plus de détails, revenir au **§ I.1**. La commande **for** se compose donc de trois composants séparés par le caractère « ; ». En premier lieu, on trouve l'affectation de la variable index de boucle à sa valeur initiale. Ensuite se situe une expression booléenne évaluée avant d'entrer dans le corps et constituant un test d'arrêt (effectif si son évaluation délivre la valeur **false**). Enfin, le troisième élément est une action effectuée au sortir du corps de boucle et permettant, en règle générale, d'effectuer l'incrémentation de l'index. Comme précédemment, le corps de boucle peut soit se réduire à une seule instruction, soit, s'il en comporte plusieurs, nécessiter leur encadrement par **{** et **}** pour définir une structure de bloc.

Voici donc un exemple simple d'une telle boucle dont vous pouvez obtenir le résultat de l'exécution :

```
var S = "";
for(var I = 0 ; I < 10 ; I++)
  S += I;
alert("La boucle a construit la chaine : " + S);
```

<img src="./images/bouton_o.gif" width="29" height="29" border="0" name="Bouton1">
 Exec <script language="JavaScript"> function xcfor(){ var S = ""; for(var I = 0 ; I < 10 ; I++) S += I; alert("La boucle a construit la chaine : " + S); } function xcfor2(){ var arret=true,S="",Add='abc'; alert("Je demarre"); for(var i=1;arret;i+=3){ alert("Dans for, i = "+i); S+=Add; arret= !(S.length>8); } alert("J'en suis sorti !!!") } function xcfor3(){ var S = ""; for(var I = -1 ; I++ < 9 ; S += I); alert("La boucle a construit la chaine : " + S); } function xcfor4(){ var S = ""; for(var I = 0 ; I < 10 ; S += I++); alert("La boucle a construit la chaine : " + S); } function xcfor5(){ var S = ""; for(var I = 1 ; I < 6 ; I++) for(var J = 2 ; J < 7 ; J+=2) for(var K = 3 ; K < 8 ; K+=3) S+=""+I+J+K; alert("Chaine construite : " + S); } function xcfor6(){ var S = ""; for(var I=1, J=2, K=3; I<6, J<7, K<8; I++, J+=2, K+=3) S += "" + I + J + K; alert("Chaine construite : " + S); } function PrObj(){ S="Proprietes de window :\n"; for(var i in window) S+=i+", "; alert(S); } </script>

Mais au-delà d'une telle utilisation « canonique », on peut imaginer d'autres formes d'utilisations de la boucle **for**. Voici un exemple qui va nous permettre de montrer à la fois comment le pas d'incrémentation peut être quelconque (et en tout cas différent de 1) et comment décrire un **while** avec un **for** (on avait plutôt l'habitude inverse). Voici l'exécution.

```
var arret = true, S = "", Add = 'abc';
alert("Je demarre");
for(var i = 1 ; arret ; i += 3){
  alert("Dans for, i = " + i);
  S += Add
  arret = !(S.length > 8);
}
alert("J'en suis sorti !!!")
```

<img src="./images/bouton_o.gif" width="29" height="29" border="0" name="Bouton7">
Exec

On voit que l'incrément (ici, 3) a pu être pris en compte grâce à l'opérateur +=. Quant au test d'arrêt, on voit qu'il peut n'avoir aucun lien direct avec l'index. C'est en fait une expression booléenne quelconque qui, d'une part est évaluée avant l'entrée dans le corps de boucle, et, d'autre part, évaluée à **false** provoque l'abandon de la boucle… EXACTEMENT COMME POUR LE **while** !!!

Grâce aux opérateurs de pré ou postincrémentation, dans le cas d'un pas de 1, l'incrémentation peut être réalisée dans le test, laissant ainsi le troisième argument libre pour réaliser une opération quelconque. Reprenons l'avant-dernier exemple… On peut le modifier comme suit et vérifier que l'exécution reste inchangée modulo une translation compréhensible des valeurs de début et fin.

```
Var S = "";
for(var I = -1 ; I++ < 9 ; S += I)
alert("La boucle a construit la chaine : " + S);
```

<img width="29" height="29" border="0" name="Bouton2" src="./images/bouton_o.gif">
 Exec

En intégrant la postincrémentation dans le troisième champ de l'instruction, les valeurs de début et fin restaient inchangées… la preuve !

```
var S = "";
for(var I = 0 ; I < 10 ; S += I++);
alert("La boucle a construit la chaine : " + S);
```

<img width="29" height="29" border="0" name="Bouton3" src="./images/bouton_o.gif">
 Exec

Enfin, il convient de noter que l'opérateur « , » permet de multiplier le nombre de variables, de tests et/ou d'opérations dans une boucle **for**. Il ne faut pas confondre cette structure avec des boucles enchâssées où chacune gère une variable. Ici toutes les variables progressent simultanément. Voici un exemple qui va nous servir à illustrer cela :

```
var S = "";
for(var I = 1 ; I < 6 ; I++)
  for(var J = 2 ; J < 7 ; J +=2)
    for(var K = 3 ; K < 8 ; K +=3)
      S += "" + I + J + K;
alert("Chaine construite : " + S);
```

<img width="29" height="29" border="0" align="absmiddle" name="Bouton4" src="./images/bouton_o.gif"> Résultat

```
var S = "";
for(var I=1, J=2, K=3; I<6, J<7, K<8; I++, J+=2, K+=3)


  S += "" + I + J + K;
alert("Chaine construite : " + S);
```

<img width="29" height="29" border="0" align="absmiddle" name="Bouton5" src="./images/bouton_o.gif"> Résultat

Le résultat des boucles enchâssées correspond bien à ce à que l'on attendait. I étant initialisé à 1, J l'est à 2 tandis que K va balayer toutes les valeurs de 3 à 7 par pas de 3 (soit 3 et 6) avant que J ne passe à 4, etc. Pour la boucle multiple, les trois tests sont évalués avant l'exécution du corps ; dès que l'un d'eux est évalué à **false**, la boucle est stoppée. Ici, la première boucle sera effectuée pour I=1, J=2, K=3, la seconde pour I=2, J=4, K=6, mais la troisième ne sera pas exécuté, car bien que I (qui prend la valeur 3) et J (qui prend la valeur 6), soient bien dans les bornes autorisées, K qui vaut à présent 9 met en défaut K<8.

La boucle for revêt une autre forme adaptée à la structure objet. Sa syntaxe est alors la suivante :

```
for([var] <ident> in <objet>)
<corps de boucle>
```

La variable va énumérer tous les noms de propriétés associées à l'objet référencé. Nous avons utilisé de nombreuses fois ce type de boucle lorsque nous avons abordé les objets au § **II.7**. Nous allons en donner ici un rapide exemple qui va nous permettre d'énumérer toutes les propriétés/méthodes contenues dans cette fenêtre.

```
Var S = "Proprietes de window :\n";
for(var i in this)      // ou window
S += i + ", ";
alert(S);
```

<img width="29" height="29" border="0" align="absmiddle" name="Bouton6" src="./images/bouton_o.gif">
 Exec

On note au passage que i prend pour valeurs successives la chaîne désignant le nom de la propriété et non la valeur. Nous verrons dans le prochain chapitre traitant des objets que pour obtenir la valeur effective il faudra utiliser l'accès associatif à l'objet… à l'exception du cas de l'objet window !!!
On notera aussi que les propriétés énumérées contiennent celles introduites par le constructeur window, mais aussi celles qui ont été rajoutées ensuite (fonctions définies dans cette page, images, etc.).

IV-C - Les ruptures

Les **ruptures de séquences** constituent un concept qui fut largement (trop peut-être) utilisé dans les langages de programmation d'ancienne génération (Fortran, Basic, Cobol, etc.) Ensuite sont apparus des langages de programmation structurée tournant résolument le dos à cette notion (Algol W, Pascal…). Les langages récents (C, Java, JavaScript, Perl) réintroduisent les ruptures de séquences utilisant, dans certains cas, des **étiquettes** pour effectuer des sorties de blocs (boucles, fonctions, aiguillages).

La syntaxe des étiquettes suit celle des identificateurs de JavaScript (voir **§ II.1**). La syntaxe est particulièrement simple. Il suffit de faire précéder l'instruction que l'on désire étiqueter par une étiquette suivie de « : ».

```
<ident étiquette> : <instruction>
```

Nous avons déjà introduit auparavant des étiquettes particulières. En effet **case** et **default** apparaissant dans une instruction **switch** ont un statut d'étiquettes. Bien que syntaxiquement on puisse étiqueter n'importe quelle instruction, dans les faits, seules les boucles que nous venons d'étudier seront concernées afin d'effectuer, soit des abandons définitifs (**break**) soit des abandons de l'itération en cours et passage à l'éventuelle itération suivante (**continue**).


***goto** est un mot réservé de JavaScript, mais l'instruction correspondante n'est pas reconnue du langage.*

L'instruction break provoque la sortie immédiate d'une instruction d'aiguillage (switch) ou de la boucle dans laquelle elle apparaît (pas au-delà dans le cas de boucles imbriquées). Hors de ces contextes, l'occurrence de l'instruction break (seule) provoque une erreur.

Si l'instruction break comporte une étiquette, celle-ci doit impérativement désigner une instruction d'un niveau supérieur à celui où apparaît l'appel. Dans ce cas, c'est tout le bloc ainsi désigné qui sera abandonné. À noter que contrairement au cas précédent, le break étiqueté peut être employé dans n'importe quel contexte définissant de façon implicite ou explicite un bloc. On peut ainsi nommer, par exemple, une instruction if ou de façon plus générale une suite quelconque d'instructions encadrées d'accolades même si celles-ci sont mises dans le seul but de nommer la suite d'instructions qu'elles contiennent.

Voici un exemple mettant en jeu ce qui vient d'être expliqué. Essayez de déterminer ce que va être la sortie puis vérifiez.

L'instruction **break** provoque la sortie immédiate d'une **instruction d'aiguillage (switch) ou de la boucle dans laquelle elle apparaît (**pas au-delà dans le cas de boucles imbriquées). **Hors de ces contextes, l'occurrence de l'instruction** break **(seule) provoque une erreur**.

Si l'instruction break comporte une étiquette, celle-ci doit impérativement désigner une instruction d'un niveau supérieur à celui où apparaît l'appel. Dans ce cas, c'est tout le bloc ainsi désigné qui sera abandonné. À noter que contrairement au cas précédent, **le** break **étiqueté peut être employé dans n'importe quel contexte définissant de façon implicite ou explicite un bloc**. On peut ainsi nommer, par exemple, une instruction if ou de façon plus générale une suite quelconque d'instructions encadrées d'accolades même si celles-ci sont mises dans le seul but de nommer la suite d'instructions qu'elles contiennent.

Voici un exemple mettant en jeu ce qui vient d'être expliqué. Essayez de déterminer ce que va être la sortie puis vérifiez.

```
var S = "Valeurs successives d'index :\n";
Boucle1:for(var i = 0 ;i < 5; i++){
  Boucle2:for(var j = 0; j < 6; j+=2){
    if (j > 2) break;
    if (i == 1) break Boucle2;
    if (i == 4) break Boucle1;
    S += "i = " + i + " j = " + j + "\n";
  }
}
S += "En sortie : i = " + i + " j = " + j + "\n";
alert(S);
```

<img width="29" height="29" border="0" align="absmiddle" name="Bouton8" src="./images/bouton_o.gif">
 Exec

L'instruction **continue** comme **break**, interrompt la boucle en cours. Mais, alors que break termine définitivement la boucle, continue se contente d'abandonner l'itération en cours pour passer directement à celle qui suit. Cela impose donc, dans une boucle **while** ou **do/while**, que l'environnement servant à évaluer l'expression gérant la fin de boucle ait été réactualisé avant l'exécution de l'instruction **continue**, faute de quoi on sera en présence d'une boule infinie. Cette précaution est sans objet dans une boucle for puisqu'après une instruction continue, l'exécution se poursuit vers l'incrémentation, dans un premier temps, avant de procéder au test.

Dans Navigator 4, la condition du do/while n'est plus évaluée après l'exécution de l'instruction continue, si bien que l'itération suivante est lancée systématiquement !

L'instruction continue peut donc apparaître dans le corps d'une boucle while ou for ou encore for/in (l'utilisation dans un do/while étant fortement déconseillée). **Hors de ces contextes, l'occurrence de l'instruction** continue **provoque une erreur de syntaxe**.

Tout comme l'instruction **break** , **continue** peut être associée à une étiquette, celle-ci devant impérativement désigner une instruction d'un niveau supérieur à celui où apparaît l'appel. Dans ce cas, c'est au niveau de l'itération suivante de la boucle ainsi désignée que se poursuivra l'exécution.

Voici un exemple mettant en jeu ce qui vient d'être expliqué. Essayez de déterminer ce que va être la sortie puis vérifiez.

```
Var S = "Valeurs successives d'index :\n"; Boucle:for(var i = 0 ;i < 4; i++){
  for(var j = 0; j < 5; j+=2){
    if (j == 2) continue;
    if (i == 1) continue Boucle;
    S += "i = " + i + " j = " + j + "\n";
  }
}
S += "En sortie : i = " + i + " j = " + j + "\n";
alert(S);
```

<img width="29" height="29" border="0" align="absmiddle" name="Bouton10" src="./images/bouton_o.gif">
 Exec

IV-D - Préfixage d'objets

<script language="JavaScript"> var CeLien=""; function Conti(){ var S = "Valeurs successives d'index :\n"; Boucle:for(var i = 0 ;i < 4; i++){ for(var j = 0; j < 5; j+=2){ if (j == 2) continue; if (i == 1) continue Boucle; S += "i = " + i + " j = " + j + "\n"; } } S += "En sortie : i = " + i + " j = " + j + "\n"; alert(S); } function ChgTxt(){ window.document.Formul.textfield.value=CeLien; //idem window.document.links[14] // adapté pour le site Developpez.com qui ajoute des liens dont je n'ai pas la maitrise } </script>

L'écriture induite par un langage objet peut devenir lourde lorsqu'on désire, faire référence à une propriété définie à un niveau très profond dans la hiérarchie des objets. Par exemple, si on définit dans le document contenu dans cette fenêtre, un formulaire contenant un champ de texte dont la valeur est « Le lien … » et que nous voulons remplacer cette chaîne par le nom du lien qui va lui-même permettre de réaliser cette opération, il faudra exécuter l'instruction suivante :

```
window.document.Formul.textfield.value = window.document.links[14];
```

<img width="29" height="29" border="0" align="absmiddle" name="Bouton11" src="./images/bouton_o.gif">

Vous pouvez contrôler cela en observant la barre de statut au bas de votre fenêtre alors que la souris survole le bouton de déclenchement…

<form name="Formul"> <input type="text" size="60" value="Le lien qui va permettre de réaliser ce changement est..." name="textfield"> </form>

…où ChgTxt est une fonction JavaScript se résumant à l'instruction ci-dessus. ***(Pourquoi 14 ?… Parce que je sais que c'est le quinzième de cette page et que les indices de tableau débutent en 0, tout simplement !!!)***

En fait, pour les curieux qui vont vérifier le source, ils verront que pour le site « Developpez.com » qui rajoute des liens dont je n'ai pas la maitrise, j'ai opéré d'une autre manière.

L'exemple que nous venons de prendre, bien qu'il ne soit pas particulièrement compliqué, va nous permettre, néanmoins, de montrer l'utilité et le fonctionnement de l'instruction **with**. Encore une fois, la syntaxe de cette instruction est particulièrement simple :

```
with (<objet>)
<corps de with>
```

Comme d'habitude, le corps de cette instruction peut se limiter à une seule instruction ou peut en contenir plusieurs, auquel cas, elles seront encadrées par { et }. L'utilité de cette instruction est de définir temporairement un *objet standard* tel que toutes les références de propriétés de celui-ci apparaissant dans le corps peuvent être résolues sans qu'il soit besoin dudit objet. Ainsi, dans notre exemple précédent, nous aurions pu écrire :

```
with (window.document)
Formul.textfield.value = links[14];
```

L'intérêt est ici modeste, mais dans le cas où le corps de l'instruction contient un nombre non négligeable de références à l'objet mis en préfixe, on comprend que cela puisse devenir intéressant. Il est évident que dans le corps d'un with, pourront apparaître (et cela est heureux !…) des variables n'ayant aucun lien avec l'*objet standard* (indices de boucles ou autres). On en déduit donc que le préfixage n'interviendra que lorsqu'il sera fait référence à une propriété existante de cet objet.

Au risque d'anticiper sur le prochain chapitre traitant spécifiquement des objets, supposons que pour un objet que nous appellerons « Ordinateur » on ait déjà défini les propriétés « Systeme », « Memoire » et « Disque », et que

l'on veuille leur affecter une valeur et rajouter une nouvelle propriété : « Processeur ». Considérons les deux portions de programme suivantes :

```
with (Ordinateur){
  S = "Proprietes de Ordinateur :\n";
  Systeme = "WinXP";
  Memoire = 1024;
  Disque = 80000;
  Processeur = "Pentium IV";
}
for (var i in Ordinateur)
  S += i + " = " + Ordinateur[i] + ",\n";
alert(S);
```

<img width="29" height="29" border="0" align="absmiddle" name="Bouton12" src="./images/bouton_o.gif"> Résultat

```
with (Ordinateur){
  S = "Proprietes de Ordinateur :\n";
  Systeme = "WinXP";
  Memoire = 1024;
  Disque = 80000;
  Ordinateur.Processeur = "Pentium IV";
}
for (var i in Ordinateur)
  S += i + " = " + Ordinateur[i] + ",\n";
alert(S);
```

<img width="29" height="29" border="0" align="absmiddle" name="Bouton13" src="./images/bouton_o.gif"> Résultat

On constate que dans le premier cas, la liste des propriétés ne contient pas « Processeur ». En fait, on a simplement défini non pas une propriété de l'objet « Ordinateur », mais une variable de type chaîne à laquelle on a affecté la valeur « Pentium IV ». De la même manière que S, par exemple n'est pas, non plus, une propriété de ce même objet. Dans le second cas, par contre, on a spécifié, bien qu'étant dans le corps du with, Ordinateur.Processeur = « Pentium IV ». On vérifie que la propriété a alors bien été rajoutée à l'objet.

IV-E - Les définitions de fonctions

L'instruction **function** de JavaScript permet de définir un objet fonction en tant que propriété/méthode de l'objet dans lequel il est défini. Sa syntaxe est la suivante :

```
function <ident> ([<par1> [,<par2> [..., <parp>]...]){
  <corps de function>
}
```

Dans cette syntaxe, il apparaît que les parenthèses sont obligatoires (même s'il n'y a pas de paramètres), de même que les accolades (même si le corps se réduit à une seule instruction) à la différence de ce que nous avons vu précédemment pour les corps de boucle.

Le fait de classer la définition des fonctions dans les instructions se place sur le même plan que ce que nous verrons dans le prochain paragraphe : ajout d'une propriété d'un objet contenant ; mais, comme dans les autres langages, la définition n'engendre pas l'exécution. Vous pouvez même, si le cœur vous en dit, définir une fonction dont les instructions ne seront jamais exécutées. En fait, au chargement de la page, le code JavaScript est analysé, puis ce qui est au niveau global est effectivement exécuté (c'est ce qui a été appelé au **§ I.5** *fonctionnement synchrone*). L'exécution d'une fonction, et donc des instructions qu'elle contient, ne se fera que de façon *asynchrone* par appel de ladite fonction. Cela va avoir des répercussions surprenantes (dont il faudra tenir compte) sur l'exécution. Considérons par exemple un document contenant le script suivant :

```
<script language="JavaScript">
  alert(Succ(6));     // Appel de la fonction
  alert(Succ);        // Affich de la fonction
```

```
  Succ = 0;
  function Succ(x) { // Définition
    return x+1;
  }
  alert(Succ);       // Affich de la fonction ?
</script>
```

<img width="29" height="29" border="0" align="absmiddle" name="Bouton14" src="./images/bouton_o.gif">
 Exec

Le fait de cliquer sur le bouton Exec a provoqué l'ouverture d'une page (vous l'avez peut-être aperçue au centre de l'écran). La fonction Succ est donc une propriété de cette page (analyse au chargement) qui va être utilisée puis modifiée par le script.

IV-F - Les déclarations

<script language="javascript"> function enumimp(x){ for(i=8;i<12 ;i++) S+="("+x+","+i+")"; S+="\n"; } function enumexp(x){ for(var i=8;i<12 ;i++) S+="("+x+","+i+")"; S+="\n"; } </script>

Bien que nous ayons commencé ce cours par un paragraphe introduisant la notion de variable JavaScript, nous ne pouvions pas éviter d'en reparler dans ce chapitre, car la déclaration explicite d'une variable est effectivement une instruction qui peut, comme on l'a vu, apparaître n'importe où dans un script. On peut toutefois compléter ce qui a donc été dit au **§ I.1** en précisant que l'opérateur « , » va trouver un de ses emplois les plus fréquents dans ce type d'instruction. En effet, la syntaxe générale est de la forme :

```
var <ident1> [= <valeur1] [, ....., <identp> [= <valeurp>]....]
```

Avant d'en terminer, il convient de préciser que le fait de définir une variable va avoir pour effet de la rajouter en tant que propriété de l'objet qui la contient. Si c'est au niveau global qu'elle est définie, ce sera une propriété de l'objet window, si c'est dans une fonction, ce sera une propriété de cet objet fonction, lui-même objet de window (si la fonction est définie au niveau global)…

La latitude de déclarer ou pas de façon explicite des variables peut avoir des conséquences éventuellement fâcheuses sur l'exécution d'un programme. En effet, une déclaration implicite, même si elle est opérée à l'intérieur d'une fonction, se traduit par une déclaration au niveau le plus haut. Si à ce niveau, une variable de même nom existe, elle sera substituée à la nouvelle variable implicitement déclarée et sa valeur sera donc affectée par les modifications survenues au niveau inférieur. Pour bien comprendre cela, regardons les résultats obtenus par ces deux programmes où la seule différence réside dans la déclaration de la variable i dans la fonction effectuée soit de façon explicite, soit de façon implicite…

```
S="";
for(i=1;i<6;i++)
enumerer(i);
alert(S);
function enumerer(x){
  for(var i=8;i<12 ;i++) //decl. explicite
  S+="("+x+","+i+")";
  S+="\n";
}
```

<a onclick="javascript:S='';for(i=1;i<6;i++)enumexp(i);alert(S);return false" href="#"><img width="20" height="20" border="0" align="absmiddle" name="Bouton15" src="./images/bouton_o.gif">

```
S="";
for(i=1;i<6;i++)
```

```
enumerer(i);
alert(S);
function enumerer(x){
  for(i=8;i<12 ;i++) //decl. implicite
  S+="("+x+","+i+")";
  S+="\n";
}
```

<a onclick="javascript:S=";for(i=1;i<6;i++)enumimp(i);alert(S);return false" href="#"><img width="20" height="20" border="0" align="absmiddle" name="Bouton16" src="./images/bouton_o.gif">

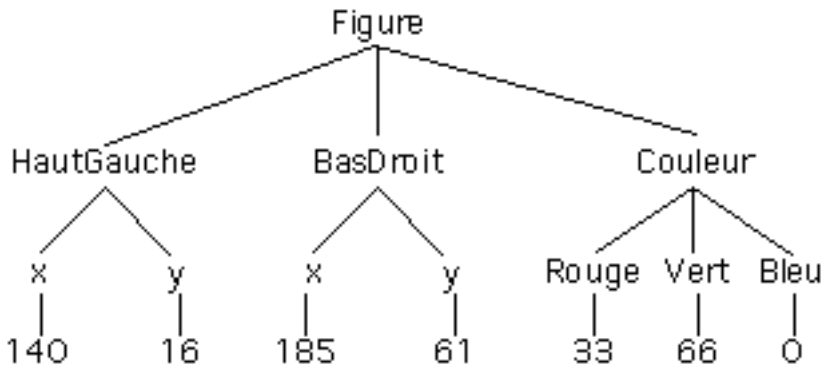
V - Les objets

V-A - Rappels

Nous avons déjà introduit, de façon succincte, la notion d'objet au **chapitre II**. Nous avons distingué deux types d'objets, les objets de **type primitif** qui font référence à une seule valeur (chaînes, nombres ou booléens) et ceux de **type composé** qui, par l'intermédiaire d'une référence unique, le nom de l'objet, permettent d'accéder à un ensemble d'informations multiples, propriétés nommées de l'objet en question. Nous nous intéresserons ici essentiellement à cette seconde catégorie.

V-B - Les constructeurs

La création d'un objet s'opère à l'aide de l'opérateur **new** suivi du nom d'un constructeur. Le constructeur le plus général est **Object**. À partir de celui-ci, on peut créer un nouvel objet en précisant ses propriétés et méthodes dont le (ou les) constructeur(s), de telle manière que ce nouvel objet puisse lui-même être invoqué avec l'opérateur **new** pour générer de nouveaux objets « clones » possédant les caractéristiques de l'objet d'origine. De proche en proche, on décrit une arborescence dans laquelle, quel que soit le niveau où l'on se place, toute propriété peut être un objet lui-même pouvant contenir des propriétés objets, etc.



Dans la figure ci-dessus, on a défini un objet **Figure** comportant trois propriétés de noms **HautGauche**, **BasDroit** et **Couleur**. Celles-ci sont elles-mêmes des objets. Pour les deux premières, il s'agit d'objets de type **point**, par exemple, objet préalablement défini et comportant deux propriétés de nom **x** et **y** définissant les coordonnées d'un point dans un repère donné. Pour la troisième, il s'agit aussi d'un objet **remplissage** possédant trois propriétés, **Rouge**, **Vert** et **Bleu**, permettant d'indiquer les intensités des couleurs primaires dans un codage RVB.

La création d'un tel objet peut se faire par les instructions suivantes :

```
var Figure = new Object();
Figure.HautGauche = new point(140,16);
Figure.BasDroit = new point(185,61);
Figure.Couleur = new remplissage(33,66,0);
```

Si, par exemple, au lieu de définir totalement cette variable on s'était contenté d'écrire **Figure.BasDroit = new point()**, les valeurs **x** et **y** de **BasDroit** auraient été indéfinies (undefined). Cette variable ayant été créée, on peut accéder à ses caractéristiques en lecture et écriture. Si l'on veut affecter les valeurs **x** et **y** de **BasDroit**, le membre gauche de l'affectation devra parcourir toute l'arborescence : **Figure.BasDroit.x = 185; Figure.BasDroit.y = 61;**(nous avons vu en étudiant l'instruction **with** comment raccourcir cette écriture).

Nous avons, par ailleurs, indiqué **plus haut** que JavaScript autorisait des littéraux pour définir des objets. La définition de **Figure** à l'aide de **littéraux objets** aurait été :

```
var Figure = { HautGauche : new point(140,16),
               BasDroit : new point(185,61),
               Couleur : new remplissage(33,66,0),
             };
ou encore....
var Figure = { HautGauche : { x : 140,
                              y : 16
                            },
               BasDroit :   { x : 185,
                              y : 61
               Couleur :    { Rouge : 33,
                              Vert : 66,
                              Bleu : 0
                            }
             };
```

Nous avons vu qu'un objet est défini par un constructeur, méthode particulière qui reçoit en paramètre tout ou partie des valeurs permettant d'affecter les diverses propriétés. Pour comprendre comment fonctionne un constructeur, cliquez ici…

<img width="20" height="20" border="0" align="absmiddle" name="Bouton7" src="./images/bouton_o.gif">

V-C - Les propriétés et les méthodes

Il est difficile de parler des constructeurs sans faire référence aux propriétés et méthodes, comme nous l'avons vu dans le paragraphe précédent. Aussi, dans ce paragraphe, nous nous contenterons de préciser ou de rappeler quelques points d'intérêt.

En JavaScript, un **constructeur** est réalisé grâce à une fonction, tout comme une **méthode**. La seule différence entre les deux, c'est que le constructeur est invoqué par l'opérateur **new** et qu'il a pour effet de créer un objet vide et éventuellement de l'initialiser en tout ou partie par l'intermédiaire du mot this qui constitue une référence vers cet objet, alors qu'une méthode est appelée en tant que **propriété** de l'objet dans lequel elle a été définie.

À ce niveau, il convient de préciser que l'utilisation du mot **this** n'est pas réservée aux seuls constructeurs. Employé dans une méthode, il fait référence à l'objet auquel appartient ladite méthode. Par exemple, en accédant à la méthode **MonCarre.Surface()**, on peut, dans le corps de la méthode **Surface()** utiliser **this** pour faire référence à l'objet **MonCarre**. C'est ainsi que l'on pourra définir cette méthode par

```
Surface()=function() {return this.cote * this.cote;}
```

Quelle différence entre méthode et fonction ?

Au moment où l'on définit une fonction, elle reçoit un nom en tant que variable qui n'est autre qu'une propriété d'un objet « englobant » (dans lequel est définie cette propriété). Ainsi, l'appel d'une fonction revient à invoquer une méthode d'un objet global et en conséquence la différence entre fonction et méthode apparaît très mince. Elle se résume en fait à ce que les méthodes sont véritablement dédiées pour opérer sur l'objet les contenant en tant que propriété, alors que le rôle des fonctions est totalement dissocié de l'objet dans lequel elles sont définies.

Dans une fenêtre HTML, l'**objet omniprésent** auquel tout ce que vous créez se réfère est l'objet **window :** il s'agit de la fenêtre (ou du cadre - « frame ») contenant. Cela est quasiment toujours vrai, si bien que c'est implicite : ce qui signifie que l'on n'est pas obligé de faire apparaître le terme **window** dans la hiérarchie des accès. Cela signifie aussi que la plupart du temps le terme **this** se réfère à **window**. Il y a deux exceptions à cette règle : les constructeurs et les gestionnaires (« handlers ») d'événements. Dans les fonctions permettant de mettre en œuvre respectivement, la construction d'un objet ou la prise en compte d'un événement, le terme **this** réfère respectivement à l'objet créé ou à l'événement traité et **window** n'est plus implicite. Si bien que dans ces cas, pour faire référence à un objet autre que celui traité, la hiérarchie d'accès devra être complète.

Il convient en outre d'ajouter que les objets étant atteints via une référence, il peut advenir que les valeurs référencées ne soient plus accessibles. Par exemple, dans le programme suivant :

```
var Acces = new Rectangle(100,45);
...
Acces = 12;
...
```

La variable **Acces**, dans un premier temps a été une référence vers un objet de type **Rectangle** qui a été créé. Puis plus loin dans le programme, le même identificateur a été affecté à une variable entière. L'objet précédemment créé n'est donc plus accessible. Son adresse d'implantation en mémoire est définitivement perdue. Cela peut se reproduire plusieurs fois dans un programme, ce qui peut engendrer une perte d'espace mémoire importante. Dans les langages Pascal, C ou C++, cette notion de référence existe et est totalement prise en charge par le programmeur au travers de pointeurs. Le bon programmeur, avant de rendre inaccessible un enregistrement, aura pris soin, explicitement, de libérer le pointeur d'accès, ce qui a pour effet de récupérer la mémoire occupée par l'enregistrement pointé. En Java et JavaScript (toujours pour des raisons de robustesse et de sécurité) il n'y a pas de pointeur !!!… ou du moins, ceux-ci ne sont pas accessibles au programmeur. Ils sont totalement gérés par le langage. De ce fait, la récupération d'espace mémoire est elle aussi automatiquement gérée par le **Garbage Collector.**

De plus, et cela permet de surpasser les autres langages cités, cette fonctionnalité ne se limite pas aux objets de type composé, mais s'étend à tous les objets !! Soit, pour exemple, le programme suivant :

```
var S1 = "javascript";      // affectation de S1
var S2 = S1.toUpperCase(); // S2 prend la valeur "JAVASCRIPT"
S1 = S2;                    // S1 prend la valeur de S2
```

… à la suite de cette portion de programme, la chaîne "**javascript**" ne peut plus être atteinte. Le Garbage Collector déterminera cela et récupérera donc la place occupée.

Une propriété d'un objet peut aussi être rendue inaccessible par suppression, tout simplement ! Depuis la version 1.2 de JavaScript, l'opérateur **delete** permet cela. Bien entendu, le Garbage Collector intervient ensuite pour la récupération d'espace. On va mettre cela en évidence. Un objet de nom « Obj » a été prédéfini et contient cinq propriétés de nom « prop1 » à « prop5 ». Vous pouvez vous en assurer en utilisant le bouton « VOIR ». À chaque action sur le bouton « DELETE », vous allez pouvoir supprimer une des propriétés. Vous pourrez vérifier à chaque fois l'état de l'objet par le bouton « VOIR ».

<center> <table width="178" bordercolor="#996633" border="6" bgcolor="#CCCC99" align="center" bordercolorlight="#9999FF" bordercolordark="#0000CC"> <tbody><tr> <td> <div align="center"><img width="72" height="30" border="0" align="absmiddle" src="./images/Voir.jpg"> </div> </td> <td> <div align="center"><img width="72" height="30" border="0" align="absmiddle" src="./images/Delete.jpg"> </div> </td> </tr> </tbody></table> </center> <script language="JavaScript"> var Obj=new Object; Obj.prop5=Obj.prop4=Obj.prop3=Obj.prop2=Obj.prop1=null; function Enum(){ var S = "Voici les noms des proprietes de l'objet Obj\n \n"; for (var i in Obj) S += i + ',\n'; S=S.slice(0,-2)+'.'; alert(S); } function Suppr(){ alert("Vous allez supprimer une des proprietes de Obj et vous

verifierez ensuite qu'elle n'est plus accessible..."); do{ var N = prompt("... Donnez une valeur entre 1 et 5 ; rien d'autre ne sera accepte !"); } while (N.match(/^\d$/) == null || N < 1 || N > 5) delete Obj["prop"+N]; if(confirm("Voulez-vous a present verifier que les proprietes choisies ont bien ete supprimees")) Enum(); } </script>

V-D - Prototype et héritage

Dans les pages d'exercices rencontrées plus haut, nous avons élaboré un constructeur d'objets de type Rectangle qui, outre les propriétés de Longueur et Largeur qui peuvent changer d'un tel objet à l'autre, contenait aussi les méthodes de calcul du Périmètre et de la Surface. Or, quelles que soient les dimensions d'un rectangle, la façon de calculer son périmètre ou sa surface sera toujours la même. Et pourtant, chaque Rectangle construit par ce constructeur disposera de ces méthodes, ce qui va avoir pour conséquence d'occuper inutilement de l'espace. Si au lieu de cela, ces méthodes, et plus généralement toutes les **propriétés constantes communes à une classe d'objets** étaient partagées par eux, l'économie d'espace en découlant constituerait un gain appréciable.

Pour cela, JavaScript propose la notion d'**objet prototype**. Tout objet dispose d'un objet prototype possédant toutes les propriétés constantes communes à la classe, propriétés dont il **hérite** au moment de sa création.

L'héritage, bien sûr, ne se traduit pas par une recopie dans l'objet (sinon, on n'aurait rien gagné). Vu de l'extérieur, tous les objets prototypés sembleront posséder l'ensemble de l'héritage, mais en fait ils ne pourront y accéder seulement en lecture en cas de besoin !…
Outre l'intérêt déjà mentionné concernant le gain d'espace, cette remarque a une autre conséquence intéressante : en cas de modification a posteriori du prototype, l'ensemble d'héritage des objets précédemment créés avec ce prototype sera modifié en conséquence.

Du fait de l'utilisation du prototype, l'accès aux propriétés selon qu'il s'opère en lecture ou en écriture deviendra un peu plus délicat :

- en **écriture**, tout d'abord, la modification, par l'utilisateur, d'une propriété ne pourra effectivement se faire que si celle-ci n'appartient pas au prototype. En effet, si une telle propriété pouvait être modifiée, elle le serait alors pour tous les objets ayant le même prototype ;
- en **lecture**, JavaScript vérifie si l'objet possède en propre la propriété invoquée. Dans la négative, la vérification se prolonge vers le prototype de l'objet. Si la recherche réussit (dans l'un ou l'autre cas), la valeur atteinte est fournie, dans l'autre cas, la valeur rendue est **undefined**.

Question : que se passe-t-il si l'utilisateur crée une propriété dont le nom apparaît dans le prototype ?

Cette propriété va tout simplement devenir propre à l'objet sur lequel la création aura eu lieu. Cet objet n'en héritera donc plus, mais il continuera à hériter du reste (éventuel) du prototype et les autres objets de la classe qui n'auront pas redéfini cette propriété continueront à en hériter. On voit donc l'importance de l'accès prioritairement sur les propriétés propres de l'objet avant de considérer le prototype. Les propriétés d'un objet **masquent** les propriétés de même nom de son prototype.

Pour créer une propriété dans le prototype d'un constructeur, la syntaxe sera la suivante :

```
<Nom constructeur>.prototype.<Nom propriété> = <expression> ;
```

Voici un exemple qui va nous permettre de mieux comprendre ce fonctionnement :

```
<script language="JavaScript">
  function Objet(x,y){                    // définition constructeur
    this.x=x;                             // propriété propre
    this.y=y;                             // propriété propre
  }
  function Moy(){                         // définition méthode
```

```
    return (this.x + this.y)/this.Cte;
  }
  Objet.prototype.Cte = 2;                // propriété prototypée
  Objet.prototype.calcul = Moy;           // méthode prototypée
  var O1 = new Objet(3,5);                // création
  var O2 = new Objet(7,9);                // création
  O2.Cte+=3; O1.star = '*';               // ajout propriétés propres
  alert(O1.x+O1.y+O1.Cte+O1.calcul());
  alert(O2.x+O2.y+O2.Cte+O2.calcul());
  Objet.prototype.New='new';              // ajout propriété prototypée
  delete O1.Cte;                          // suppression dans prototype
  delete O2.Cte;                          // suppression en propre
  S="Proprietes de O1 :\n";
  for(var i in O1)S+=i+', '; alert(S);    // affich O1
  S="Proprietes de O2 :\n";
  for(var i in O2)S+=i+', '; alert(S);    // affich O2
</script>
```

Ce script définit un constructeur **Objet** comportant deux propriétés **x** et **y**. Par ailleurs on *prototype* deux autres propriétés, une constante de nom **Cte** et une méthode opérant un calcul simple, **calcul**. On crée ensuite deux objets, **01** et **02**. La propriété **Cte** de **02** est masquée par une nouvelle propriété évaluée à partir du prototype. La propriété **02**.**Cte** vaut donc à présent 2 + 3, soit 5. Enfin on ajoute une nouvelle propriété au prototype.

Les résultats nous montrent bien que :

- les propriétés du prototype sont bien accessibles à partir des objets construits ;
- **01** a conservé la **Cte** égale à 2, tandis pour **02** on a atteint la valeur 5 puisque **Cte** a été redéfinie ;
- la propriété **calcul** prend en compte la variable **Cte** prototypée pour **01** et la variable **Cte** propre pour **02** ;
- l'opérateur delete est sans effet sur la propriété **Cte** de **01,** car elle est prototypée ;
- par contre dans O2, la propriété propre **Cte** qui valait 5 est bien supprimée et la propriété prototype de valeur 2 est alors démasquée ;
- enfin on constate bien, l'ajout a posteriori d'une propriété de nom **New** dans le prototype.

exécution

*On retrouve ce mécanisme de prototypage dans les classes d'objets prédéfinies de JavaScript, comme la classe **String**. Ainsi, dans ces classes, on aura le loisir de modifier le prototype en ajoutant, par exemple, des méthodes qui seront donc accessibles par tous les objets **String**.*

V-E - Les tableaux associatifs

De la même façon que l'on utilise l'opérateur « . » pour accéder aux propriétés d'un objet, on peut réaliser la même opération en utilisant le mécanisme des tableaux associatifs.
En fait au lieu d'utiliser la notation ***<ident objet>.<propriété>***, on peut la remplacer par ***<ident objet>*["*<propriété>*"]. Alors que dans la première forme, la propriété apparaît en tant qu'identificateur, c'est la chaîne de caractères correspondant à celui-ci qui est utilisée dans la seconde forme. Cela est très intéressant, car on peut utiliser toutes les méthodes accessibles à partir d'un objet de type **String** pour créer ou construire le nom de la propriété. Par exemple, dans l'illustration de l'opérateur delete ci-dessus, c'est le numéro **N** de la propriété qui a été fourni par l'utilisateur sous la forme de chaîne. Il a suffit de concaténer la chaîne « **prop** » à ce numéro pour opérer enfin **delete["prop"+N]** qui réalisait la suppression souhaitée.
Par ailleurs, alors que dans C++ ou Java, les propriétés d'une classe d'objets sont parfaitement définies avant la compilation, nous avons vu que dans JavaScript, l'ensemble des propriétés était totalement dynamique et "qu'il suffit de l'écrire pour qu'elle existe". Dans ces conditions, il peut être judicieux, dans une phase itérative, de créer des propriétés sous la forme d'une chaîne composée d'un préfixe suivi d'un numéro (comme précédemment) et de les manipuler avec ce formalisme de tableaux associatifs.

V-F - L'objet Object

<script language="JavaScript"> function Objet(x,y){ this.x=x; this.y=y; } var O1 = new Objet(3,5); </script>

L'objet **Object**, en JavaScript comme en Java est **la classe la plus générale à partir de laquelle tout objet est dérivé**. Les autres classes prédéfinies du langage ou les classes définies par l'utilisateur comportent donc les propriétés et méthodes qui leur sont spécifiques ainsi que celles dont dispose la classe **Object**. Il apparaît donc nécessaire de préciser ici ces dernières.

La propriété **constructor** fait référence à la fonction utilisée pour construire l'objet. Par exemple, **"JavaScript".constructor** est une expression dont voici l'évaluation…

<img width="15" height="15" border="0" align="absmiddle" name="Bouton2" src="./images/bouton_o.gif">

Il apparaît bien qu'il s'agit d'un constructeur prédéfini du langage. Voyons ce qu'il se passe pour un constructeur utilisateur en évaluant l'expression **01.constructor** associée à l'objet O1 utilisé plus haut…

<img width="15" height="15" border="0" align="absmiddle" name="Bouton3" src="./images/bouton_o.gif">

On retrouve bien le constructeur que l'on a présenté dans l'exemple précédent.

N.B. Les utilisateurs de **Safari 1.0** sous **Mac OS X** s'apercevront que le fonctionnement présente quelques problèmes. Dans les deux cas, on obtient **"Internal function"**.

Définissons un constructeur Individu, puis une instance Untel de la façon suivante :

```
function Individu(N,P){
   this.Nom=N;
   this.Prenom=P;
   this.Naiss={An:0,Mois:0,Jour:0};
 }
 var Untel = new Individu("Terieur","Alex");
 with (Untel.Naiss){
   An = 1972;
   Mois = 2;
   Jour = 29;
 }
 Untel.Job = "Enseignant";
```

<script language="JavaScript1.2"> function Individu(N,P){ this.Nom=N; this.Prenom=P; this.Naiss={An:0,Mois:0,Jour:0}; } var Untel = new Individu("Terieur","Alex"); with (Untel.Naiss) { An = 1972; Mois = 2; Jour = 29; } Untel.Job = "Enseignant"; // J'ai triche en alterant le nom de la fonction afin que les 2 coexistent sur cette meme page... Individu.prototype.toSlring = function(){ with (this) var S = "Il s'agit de M. "+ Nom +" "+ Prenom + " ne le "+ Naiss.Jour +"/"+ Naiss.Mois +"/"+ Naiss.An + " exer\u00e7ant la profession : "+ Job; return S; } Individu.prototype.valueOf = function(){ with (this.Naiss) return 100*(100*Jour + Mois) + An %100; } </script>

Les méthodes que nous allons présenter à présent ont des comportements qui diffèrent entre les navigateurs et entre les diverses versions d'un même navigateur. Essayons d'écrire l'objet **Untel** par la méthode **alert** puisque l'on est en asynchrone, mais cela pourrait aussi s'appliquer à la méthode **write** dans le cas synchrone…

alert(Untel)

On constate que cette écriture bien que guère explicite sur le contenu de l'objet *(en particulier, Internet Explorer reste muet sur cet appel)* nous renseigne, pour les versions de Netscape inférieures à 4.5 sur son type (objet) et le nom de sa classe (Object). Par contre pour les versions 4.x plus récentes, la structure apparaît.

Nous allons utiliser une nouvelle méthode, toString() qui étant définie dans la classe Object est donc accessible par tout objet. Voyons donc ce que cela donne en exécutant

alert(Untel.toString())
Le résultat reste le même pour Netscape, car en fait, **alert** a opéré une utilisation implicite de la méthode. À noter que pour les utilisateurs d'Internet Explorer, s'agissant d'un objet, la sortie sera effectivement toujours du type **[object *<nom de classe>*]**. Par contre, pour les utilisateurs de versions de Netscape supportant la version 1.2 de JavaScript, l'utilisation de **toString()** dans un script, produira déjà plus d'informations sur le contenu de l'objet puisque c'est en fait sa forme littérale qui sera donnée.

N.B. Aujourd'hui, la famille Mozilla a abandonné cette fonctionnalité apportée à toString().

Mieux, on peut redéfinir et prototyper la méthode **toString** pour qu'elle présente toutes les instances sous la forme que l'on veut et sous quelque navigateur que ce soit. Ajoutons par exemple la fonction suivante :

```
Individu.prototype.toString = function(){
  with (this)
  var S =
    "Il s'agit de M. "+ Nom + " "+ prenom + " ne le "+
    Naiss.Jour + "/" + Naiss.Mois + "/" + Naiss.An +
    " exerçant la profession "+ Job;
  return S;
}
```

<script language="JavaScript1.2"> function Individu(N,P){ this.Nom=N; this.Prenom=P; this.Naiss={An:0,Mois:0,Jour:0}; } var Untel = new Individu("Terieur","Alex"); with (Untel.Naiss) { An = 1972; Mois = 2; Jour = 29; } Untel.Job = "Enseignant"; // J'ai triche en alterant le nom de la fonction afin que les deux coexistent sur cette meme page... Individu.prototype.toSlring = function(){ with (this) var S = "Il s'agit de M. "+ Nom +" "+ Prenom + " ne le "+ Naiss.Jour +"/"+ Naiss.Mois +"/"+ Naiss.An + " exer\u00e7ant la profession : "+ Job; return S; } Individu.prototype.valueOf = function(){ with (this.Naiss) return 100*(100*Jour + Mois) + An %100; } </script>
… et voyons à nouveau ce que donne

alert(Untel.toString())
Désormais, grâce à cette redéfinition, sous Explorer comme sous Netscape, toute écriture d'un objet de type **Individu** aura cette allure.

La particularité signalée précédemment en ce qui concerne Netscape ne suit pas la norme ECMA. En conséquence, dans JavaScript 1.3, Netscape revient dans la norme en donnant à **toString()** la fonction de délivrer, comme pour Internet Explorer, le nom de classe. Par contre, la fonctionnalité délivrant l'objet sous sa forme littérale est à présent disponible.
Il s'agit de la méthode **toSource()** seulement disponible sur Netscape, mais qui le sera sur les prochaines versions d'Internet Explorer. Pour les utilisateurs de Netscape, voyons ce que donne

alert(Untel.toSource()).
À noter que là encore, **Safari** fait exception à la règle en ne se comportant pas comme ses alter ego (Netscape, Mozilla, Camino ou Navigator, etc.)

La philosophie de la prochaine méthode que nous allons voir se rapproche de celle de **toString()**. Alors que cette dernière a pour fonction de transformer un objet vers une chaîne, **valueOf()** tend à traduire un objet vers une valeur numérique. Lorsque cela n'est pas possible cette méthode délivre les mêmes résultats que toString(). Comme on l'a vu pour toString(), valueOf() peut être redéfini pour donner une valeur que l'utilisateur jugera représentative d'un objet. Par exemple, pour la classe **Individu** définie plus haut, on peut imaginer que la date de naissance sous une forme *jjmmaa* soit représentative de chaque objet de cette classe. La redéfinition de valueOf() sera donc de la forme :

```
Individu.prototype.valueOf = function(){
  with (this.Naiss)
   return 100*(100*Jour + Mois) + An%100;
}
```

Voici quelques exemples qui vont illustrer le comportement de cette méthode, en particulier sur deux types d'objets définis dans cette page : O1 dont on a rencontré précédemment la définition de la classe « objet » à laquelle il appartient et dans laquelle valueOf() reste « générique » et Untel de la classe Individu dont on vient de voir la redéfinition de la méthode en question :

<table width="350" cellpadding="3" bordercolor="996633" border="5" bgcolor="eeeeee" bordercolordark="666633" bordercolorlight="999966"> <tbody><tr bgcolor="#E3E3E3"> <td width="226" class="programme"> Comportement de valueOf()... </td> <td width="65"> Résultat </td> </tr> <tr> <td width="226">B = 3!=2; B.valueOf()
 </td> <td width="65" bgcolor="006699"> <img width="20" height="20" border="0" name="Bouton10" src="./images/bouton_o.gif"> </td> </tr> <tr> <td width="226">Untel.valueOf()
 </td> <td width="65" bgcolor="006699"> <img width="20" height="20" border="0" name="Bouton11" src="./images/bouton_o.gif"> </td> </tr> <tr> <td width="226">O1.valueOf()
 </td> <td width="65" bgcolor="006699"> <img width="20" height="20" border="0" name="Bouton12" src="./images/bouton_o.gif"> </td> </tr> <tr> <td width="226">"n'importe quoi".valueOf() </td> <td width="65" bgcolor="006699"> <img width="20" height="20" border="0" name="Bouton13" src="./images/bouton_o.gif"> </td> </tr> </tbody></table>

V-G - Exemples d'objets… et d'événements

Pour montrer un exemple de hiérarchie d'objets et accessoirement des gestionnaires d'événements qui leur sont attachés, voici une

page
qui « met en scène » une configuration d'objets couramment utilisés.

VI - Les tableaux

VI-A - Rappels

Nous avons introduit ce concept précédemment lorsque nous avons présenté les divers types d'objets, et en particulier, les objets dits **composés**. Nous avons aussi parlé de tableaux associatifs dans le précédent chapitre §5. Il est important de souligner qu'**il n'y a rien à voir entre les deux !** Alors que le second constitue une façon d'accéder à des objets/propriétés via une chaîne de caractères les désignant, le premier, auquel nous nous intéresserons dans ce chapitre, sont des tableaux au sens habituel du terme dans un langage de programmation, c'est-à-dire, des objets qui permettent d'associer une série de valeurs à une série de nombres entiers. On dit que la série de valeurs est indicée(***indexée***) et chacune est accessible au moyen d'un nombre, **la valeur de l'indice** (***index***) lui **correspondant**.

À noter toutefois qu'à la différence d'autres langages de programmation, en JavaScript, les éléments contenus dans un même tableau peuvent être de types différents.

VI-B - Comment définir un tableau

La création d'un objet de type Array (*que nous appellerons dorénavant tableau*) s'opère « canoniquement » à l'aide de l'opérateur **new** suivi du constructeur **Array**. Cette méthode peut revêtir trois allures différentes : soit on définit seulement un objet tableau, sans préciser sa taille ni son contenu, soit on précise sa taille, mais pas son contenu, soit son contenu et donc, implicitement, sa taille.

```
var MonTableau1 = new Array();
var MonTableau2 = new Array(15);
var MonTableau3 = new Array(1,"Salut",3.14,void 0);
```

À ce point il importe de préciser quatre choses :

- tout indice de tableau débute en 0 (comme en C, C++, Java, etc.) ;
- la taille d'un tableau n'est pas figée par sa définition ou son initialisation. Elle est **dynamique**. Le script qui suit vous en donne la preuve !!! ;
- à chaque instant, la taille d'un tableau peut être connue en invoquant la propriété **length** de l'objet correspondant à ce tableau. (*Nous verrons, à la fin de ce chapitre, un exercice mettant en jeu la possibilité d'accéder à cette propriété non seulement en lecture, mais aussi en écriture.) ;*
- l'accès à un élément d'un tableau **Tab** s'opère en faisant suivre l'identificateur représentant la référence vers ce tableau (ici Tab) d'une paire de crochets (ouvrant et fermant), **l'opérateur [ ]**, contenant une expression arithmétique s'évaluant par un entier positif ou nul et représentant la valeur de l'indice.

```
<script language="JavaScript">
  var T = new Array(3);
  var S ="";
  for (var i = 0; i < T.length; i++){
    T[i] = i;
  }
  T[5] = "Pourtant, j'y suis";
  for (var i = 0; i < T.length; i++){
    S += "En " + i + " on trouve " + T[i] + "\n";
  }
  alert(S);
</script>
```

Par ailleurs, on peut à la fois définir et initialiser un tableau par sa forme littérale en précisant les valeurs successives entre crochets.

```
var MonTableau1 = [1,"Salut",3.14,null];
var MonTableau2 = [['essai',3],{h:3,v:12},2.7183];
var MonTableau3 = [function suc(x){return x+1},3];
```

La deuxième ligne de l'exemple ci-dessus montre que l'on peut avoir des tableaux multidimensionnés ou plus exactement des tableaux de tableaux. Pour accéder à un élément de tableau de niveau inférieur, on aura recours à autant d'opérateurs [ ] que de niveaux à traverser. Ainsi, dans l'exemple ci-dessus, l'écriture **MonTableau2[1][1]** aura pour valeur '**essai**', tandis que **MonTableau2[1][2]** aura pour valeur **3**.

VI-C - Les méthodes de la classe Array

<script language="JavaScript"> function Join(){ var Tab = new Array(); var Sep; S = prompt("Donnez une chaine de caracteres :",""); Sep = prompt("Donnez une chaine separateur :",""); Tab = S.split(Sep); if(Tab.length==1) alert("Le s\u00e9parateur \""+Sep+"\" n'est pas contenu dans la chaine \""+S+"\" !\n\nLe tableau r\u00e9sultat ne comporte donc qu'un

\u00e9l\u00e9ment contenant ladite chaine") else{ alert ("Voici le tableau :\n\n" + Tab.valueOf()); Sep = prompt("Donnez une nouvelle chaine separateur :",""); alert('Voici la nouvelle mouture...\n\n"' + Tab.join(Sep) + '"'); } } </script>

La classe **Array** comporte un certain nombre de méthodes dont plusieurs s'inspirent du langage Perl. Elles permettent de manipuler les tableaux en les transposant en chaînes de caractères, de les inverser, de les trier, d'en extraire des sous-tableaux, de les gérer en tant que pile ou file, etc.

VI-C-1 - La méthode join()

Cette méthode s'applique à un tableau et a pour fonction de le **transformer en une chaîne de caractères** dans laquelle les éléments sont séparés par le caractère ",". La méthode join peut recevoir un caractère optionnel spécifiant la chaîne de caractères (éventuellement réduite à un seul caractère) qui servira à séparer les éléments. En conséquence, quel que soit le tableau **Tab**, **Tab.join()** est équivalent à **Tab.join(",")**.
Il convient de noter (nous la verrons dans le prochain chapitre), la méthode **join()** a une méthode inverse, **split()** qui s'applique donc à un objet de type chaîne, pour transformer celui-ci en éléments d'un tableau sur la base d'un séparateur fourni en paramètre.

Nous allons imager ce duo par un exemple dans lequel à partir d'une chaîne de caractères, puis d'un caractère (ou d'une suite de caractères) auquel sera attribué le rôle de séparateur, fournira alors les éléments du tableau ainsi constitué. Sur la base d'une nouvelle chaîne de séparation, la chaîne des éléments du tableau ainsi séparés sera alors affichée ;

<img width="15" height="15" border="0" align="absmiddle" name="Bouton1" src="./images/bouton_o.gif">

VI-C-2 - La méthode concat()

Cette méthode réclame en argument le ou **les éléments qui seront rajoutés en fin du tableau référencé**. L'absence d'argument ne provoque pas d'erreur, mais présente un intérêt limité ;-) Elle retourne comme résultat un tableau comportant en tête les éléments du tableau source et se terminant par les éléments rajoutés. Dans le cas où l'un des éléments est un tableau, ce sont ses éléments qui sont rajoutés dans l'ordre des indices croissants. Ce traitement n'est toutefois pas récursif et si l'un des éléments rajoutés est un tableau de tableau(x) l'aplatissement n'interviendra qu'au premier niveau. Voici quelques exemples :

Comportement de concat()…	*Résultat*
[3,"essai",pi=3.14].concat("ajout",.159)	 <img width="20" height="20" border="0" name="Bouton2" src="./images/bouton_o.gif">
[3,"essai",pi=3.14].concat([1,6],true)	 <img width="20" height="20" border="0" name="Bouton3" src="./images/bouton_o.gif">
[3,"essai",pi=3.14].concat([1,[6,true]])	 <img width="20" height="20" border="0" name="Bouton4" src="./images/bouton_o.gif">
[3,"essai"].concat(false,[[4,[6,2]],31])	 <img width="20" height="20"

	border="0" name="Bouton5" src="./images/bouton_o.gif">

VI-C-3 - La méthode reverse()

Comme son nom l'indique, cette méthode tout simplement **inverse l'ordre des éléments du tableau référencé**. Le traitement s'opère dans le tableau lui-même et ne nécessite donc pas une affectation comme c'était le cas pour **concat()** ou comme on le verra pour **slice()**.

VI-C-4 - La méthode sort()

La méthode **sort()** agit elle aussi sur le tableau référencé afin de le **trier**. Par défaut, le tri s'opère par **ordre alphabétique** après transformation de tous les éléments qui le nécessitent en chaîne de caractères.
Par exemple,

```
[3,"essai",true,25,["prénom","nom"],false,"in"].sort()
```

a pour résultat :

```
[25,3,["prénom","nom"],"essai",false,"in",true]
```

On notera que l'ordre alphabétique n'intervient qu'au niveau le plus haut, celui du tableau effectivement trié. Le tableau enchâssé n'est donc pas lui-même trié. En fait, l'ensemble formé par ce tableau enchâssé est pris tel quel comme une chaîne de caractères et classé en conséquence.

Soit un tableau de nombres,

```
TabN=[9,12,77,31,14,54,111]
```

et un tableau de chaînes de caractères,

```
TabC=["Pascal","java","C++","fortran","COBOL","algol"]
```

Examinons le tri utilisant diverses fonctions…

<table cellpadding="3" bordercolor="996633" border="5" bgcolor="eeeeee" bordercolordark="666633" bordercolorlight="999966"> <tbody><tr bgcolor="#E3E3E3"> <td width="226" class="programme"> Comportement de sort()... </td> <td width="65" class="programme"> Résultat </td> </tr> <tr> <td width="226" class="programme">TabN.sort()
 </td> <td width="65" bgcolor="006699" class="programme"> <img width="20" height="20" border="0" name="Bouton6" src="./images/bouton_o.gif"> </td> </tr> <tr> <td width="226" class="programme">TabN.sort(function(x,y){return x-y}) </td> <td width="65" bgcolor="006699" class="programme"> <img width="20" height="20" border="0" name="Bouton7" src="./images/bouton_o.gif"> </td> </tr> <tr> <td width="226" class="programme">TabN.sort(function(x,y)

{return x%2-y%2}) </td> <td width="65" bgcolor="006699" class="programme">
<a onclick="IniTabN();alert("Les pairs \u00e0 gauche les impairs \u00e0 droite...
\n \n"+TabN.sort(function(x,y){return x%2-y%2}));return false" href=""> <img
width="20" height="20" border="0" name="Bouton4" src="./images/bouton_o.gif"></
a> </td> </tr> <tr> <td width="226" class="programme">TabC.sort()</td> <td
width="65" bgcolor="006699" class="programme"> <a onclick="alert("Ordre
alphabetique sensible a la casse\n \n"+TabC.sort());return false" href=""> <img
width="20" height="20" border="0" name="Bouton8" src="./images/bouton_o.gif"></
a> </td> </tr> <tr> <td width="226" class="programme"> TabC.sort(function(x,y)<font class="programme">
   {return(x.toLowerCase()<=y.toLowerCase()?—1:1)})</
font> </td> <td width="65" bgcolor="006699" class="programme">
<a onclick="alert("Ordre alphabetique insensible a la casse\n \n"
+TabC.sort(function(x,y){return (x.toLowerCase()<=y.toLowerCase()?-1:1)}));return
false" href=""> <img width="20" height="20" border="0" name="Bouton9" src="./images/
bouton_o.gif"> </td> </tr> </tbody></table>

Les utilisateurs d'**Internet Explorer** vont s'apercevoir, dans les deux derniers exemples, que le comportement de la méthode ne correspond pas à ce qui est décrit précédemment. En effet, ce navigateur **gère mal le premier paramètre lorsqu'il est négatif**. L'utilisation de cette méthode doit donc prendre cela en compte.

VI-C-5 - La méthode slice()

<script language="JavaScript"> var TabN=new Array(9,12,77,31,14,54,111); var TabC=new
Array("Pascal","java","C++","fortran","COBOL","algol"); var Tab=new Array(); var V; var
Deux=Trois=Quatre=false; function IniTabN(){TabN=new Array(9,12,77,31,14,54,111);}; </
script>

Cette méthode est utilisée pour **extraire un sous-tableau du tableau référencé**. Elle reçoit deux arguments et le sous-tableau extrait contient les éléments du tableau source depuis l'indice désigné par le premier argument jusqu'à l'indice désigné par le second argument **non compris**. Dans le cas où un seul argument est spécifié, le sous-tableau contient tous les éléments depuis l'indice ainsi désigné jusqu'à la fin du tableau. Enfin, si un des arguments (ou les deux) est négatif, l'indice désigné est relatif à la fin. Par exemple -1 désigne le dernier élément, -2, l'avant-dernier, etc. Reprenons, par exemple, le tableau de nombres précédent,

```
TabN=[9,12,77,31,14,54,111].
```

 <table width="350" cellpadding="3" bordercolor="996633" border="5"
bgcolor="eeeeee" bordercolordark="666633" bordercolorlight="999966"> <tbody><tr
bgcolor="#E3E3E3"> <td width="226" class="programme"> <font size="4" face="Arial,
Helvetica, sans-serif" color="#333300">Comportement de slice()... </
td> <td width="65" class="programme"> <font size="3" face="Arial, Helvetica,
sans-serif" color="#333300">Résultat </td> </tr> <tr> <td
width="226" class="programme">TabN.slice(2,5)
 </td> <td width="65" bgcolor="006699"
class="programme"> <a onclick="IniTabN();alert("Sous tableau extrait :\n
\n["+TabN.slice(2,5)+"]");return false" href=""> <img width="20" height="20"
border="0" name="Bouton10" src="./images/bouton_o.gif"> </td> </tr> <tr>
<td width="226" class="programme">TabN.slice(4)</td> <td width="65" bgcolor="006699"
class="programme"> <a onclick="IniTabN();alert("Sous tableau extrait :\n
\n["+TabN.slice(4)+"]");return false" href=""> <img width="20" height="20"
border="0" name="Bouton11" src="./images/bouton_o.gif"> </td> </tr> <tr>
<td width="226" class="programme">TabN.slice(2,-3) </td> <td width="65" bgcolor="006699"

class="programme"> <img width="20" height="20" border="0" name="Bouton12" src="./images/bouton_o.gif"> </td> </tr> <tr> <td width="226" class="programme">TabN.slice(-5,5)</td> <td width="65" bgcolor="006699" class="programme"> <img width="20" height="20" border="0" name="Bouton13" src="./images/bouton_o.gif"> </td> </tr> <tr> <td width="226" class="programme">TabN.slice(-6,-2) </td> <td width="65" bgcolor="006699" class="programme"> <img width="20" height="20" border="0" name="Bouton14" src="./images/bouton_o.gif"> </td> </tr> </tbody></table> Les utilisateurs d'Internet Explorer vont s'apercevoir, dans les deux derniers exemples, que le comportement de la méthode ne correspond pas à ce qui est décrit précédemment. En effet, ce navigateur gère mal le premier paramètre lorsqu'il est négatif. L'utilisation de cette méthode doit donc prendre cela en compte.

Les utilisateurs d'**Internet Explorer** vont s'apercevoir, dans les deux derniers exemples, que le comportement de la méthode ne correspond pas à ce qui est décrit précédemment. En effet, ce navigateur **gère mal le premier paramètre lorsqu'il est négatif**. L'utilisation de cette méthode doit donc prendre cela en compte.

VI-C-6 - La méthode splice()

Comme pour **reverse()** et **sort()**, cette méthode agit directement sur le tableau référencé (1) . Elle a pour fonction d'**insérer et/ou retirer des éléments du tableau**. Le premier argument désigne l'indice dans le tableau de l'élément de départ à partir duquel la méthode va opérer *(comme pour **slice()** on peut ici utiliser des indices relatifs à la fin)*. Le second argument indique le nombre d'éléments qui vont être retirés à partir de l'indice désigné précédemment. En l'absence de cet argument, c'est la fin du tableau qui est supprimée. Les arguments suivants sont en fait les éléments qui doivent être insérés dans le tableau à partir de la position indiquée par le premier argument. La méthode délivre une valeur en retour : le tableau constitué des éléments supprimés (éventuellement vide).

Voyons quelques exemples d'utilisation de cette méthode sur le tableau de chaînes de caractères

```
TabC=["Pascal","java","C++","fortran","COBOL","algol"].
```

Cliquez de haut en bas afin de voir les transformations successives du tableau **TabC**.

 <table width="350" cellpadding="3" bordercolor="996633" border="5" bgcolor="eeeeee" bordercolordark="666633" bordercolorlight="999966"> <tbody><tr bgcolor="#E3E3E3"> <td width="226" class="programme"> Comportement de splice()... </td> <td width="65" class="programme"> Résultat </td> </tr> <tr> <td width="226" class="programme">TabC.splice(2,1)
 </td> <td width="65" bgcolor="006699" class="programme"> <img width="20" height="20" border="0" name="Bouton15" src="./images/bouton_o.gif"> </td> </tr> <tr> <td width="226" class="programme">TabC.splice(4)</td> <td width="65" bgcolor="006699" class="programme">

  <img width="20" height="20" border="0" name="Bouton16" src="./images/bouton_o.gif"> </td> </tr> <tr> <td width="226" class="programme">TabC.splice(1,1,"PL1","SmallTalk") </td> <td width="65" bgcolor="006699" class="programme"> <img width="20" height="20" border="0" name="Bouton17" src="./images/bouton_o.gif"> </td> </tr> <tr> <td width="226" class="programme">TabC.splice(3,0,["a",2],true)</td> <td width="65" bgcolor="006699" class="programme"> <img width="20" height="20" border="0" name="Bouton18" src="./images/bouton_o.gif"> </td> </tr> </tbody></table>
 <font size="-1">^(*)</ font> Cette méthode apparue sous Netscape 4 n'est supportée par Internet Explorer que pour ses versions les plus récentes (>5.0x). À l'inverse des méthodes précédentes, les résultats affichés ne sont pas issus de véritables calculs, mais sont des chaînes de caractères. Il faut bien que les "Exploreurs" voient ce qu'ils ratent !....

Les quatre méthodes qui vont suivre permettent une utilisation de l'objet tableau pour stoker et récupérer des valeurs, non pas par un accès aléatoire via un indice, mais en **gérant le tableau sous la forme d'une pile**. La structure de pile est bien connue des informaticiens. Il s'agit comme son nom l'indique d'un empilement de valeurs, chaque valeur empilée venant recouvrir la précédente en écriture et seule la dernière empilée étant accessible en lecture. Il convient de préciser que l'ensemble de ces quatre méthodes n'est pas supporté par Internet Explorer, seul Netscape en autorisant l'utilisation.

VI-C-7 - Les méthodes shift() & unshift()

Ces deux méthodes permettent respectivement d'empiler et de dépiler des valeurs quelconques dans un tableau en **utilisant systématiquement le début du tableau** pour insérer ou retirer des éléments. Cela impose donc obligatoirement une translation des éléments du tableau, soit pour laisser la place en indice 0 pour une valeur empilée, soit pour récupérer la place libérée en indice 0 par une valeur dépilée. Cela explique donc le nom de ces méthodes.

Alors que la méthode **unshift()** réclame des arguments (les valeurs empilées dans l'ordre de leur empilement), la méthode **shift()** n'a elle aucun argument puisque par définition, c'est toujours l'élément d'indice 0 qu'elle prélève. À noter que bien évidemment c'est le tableau référencé qui est lui-même affecté par l'utilisation de ces méthodes qui, par ailleurs, retournent une valeur : pour **unshift()** la valeur de retour indique la **taille de la pile ;** pour **shift()**, elle correspond à l'**élément prélevé en tête de pile**.

Soit un tableau de nombres **Tab=[]**, voyons comment il se transforme par l'action de ces méthodes. Là encore, pour respecter la chronologie, cliquez de haut en bas.

 <table width="350" cellpadding="3" bordercolor="996633" border="5" bgcolor="eeeeee" bordercolordark="666633" bordercolorlight="999966"> <tbody><tr bgcolor="#E3E3E3"> <td width="226" class="programme"> Comportement de shift()/ unshift()... </td> <td width="65" class="programme"> Résultat </

center> </td> </tr> <tr> <td width="226" class="programme">V=Tab.unshift(22,9)
</td> <td width="65" bgcolor="006699" class="programme"> <a
onclick="Deux=Trois=Quatre=false;alert("Tab devient :\n \n[22,9]\n \net la m\u00e9thode
retourne la valeur : 2");Deux=true;return false" href="#"> <img width="20" height="20"
border="0" name="Bouton19" src="./images/bouton_o.gif"> </td> </tr> <tr>
<td width="226" class="programme">V=Tab.shift()</td> <td width="65" bgcolor="006699"
class="programme"> <a onclick="if(Deux) {alert("Tab devient :\n \n[9]\n \net la
m\u00e9thode retourne la valeur : 22");Deux=false;Trois=true} else alert("Cliquez
dans l'ordre !");return false" href="#"> <img width="20" height="20" border="0"
name="Bouton20" src="./images/bouton_o.gif"> </td> </tr> <tr> <td width="226"
class="programme">V=Tab.unshift(V%3,[V,V+1]) </td> <td width="65" bgcolor="006699"
class="programme"> <a onclick="if(Trois) {alert("Tab devient :\n \n[1,
[22,23],9]\n \net la m\u00e9thode retourne la valeur : 3");Trois=false;Quatre=true} else
alert("Cliquez dans l'ordre !");return false" href="#"> <img width="20" height="20"
border="0" name="Bouton21" src="./images/bouton_o.gif"> </td> </tr> <tr>
<td width="226" class="programme">V=Tab.shift()</td> <td width="65" bgcolor="006699"
class="programme"> <a onclick="if(Quatre) {alert("Tab devient :\n \n[[22,23],9]\n
\net la m\u00e9thode retourne la valeur 1");Quatre=false} else alert("Cliquez
dans l'ordre !");return false" href="#"> <img width="20" height="20" border="0"
name="Bouton22" src="./images/bouton_o.gif"> </td> </tr> </tbody></table> </
div>

VI-C-8 - Les méthodes push() & pop()

Alors que les deux méthodes précédentes interviennent sur le début du tableau, **push()** insère une valeur en fin de tableau, de même que **pop()** prélève en fin de tableau. On évite ainsi le problème de translation rencontré précédemment, par contre, l'indice concerné est dynamique (la propriété length sera fort utile). À ce sujet, même si Internet Explorer ne dispose pas de ces méthodes, il sera aisé de les définir. Essayez, par exemple, de réécrire la méthode **pop()** avant de regarder une proposition de

<a onclick="this.href=window.open('./fichiers/
POProto.html','','width=800,height=600,scrollbars=yes');return false" href="[object
Window]">solution

.

À présent, prototypez la classe Array avec une méthode **push()**.

VII - Les chaînes de caractères

<script language="JavaScript"> var Bouton_OFF; var Bouton_ON; var
Refer_Down=Refer_Up=Refer_Norm=new Image(); var NS; onload=Declare();
Refer_Norm.src="./images/RefNorm.gif"; Refer_Down.src="./images/RefDown.gif"; function
changer(Act) { if(document.images) { document.images["Refer"].src = "./images/Ref"+Act+".gif";
eval("Refer_" + Act); } } var Temp; function Type(x){ switch (x){ case 1: var V=prompt('Entrez
une valeur'); Temp = V; alert('La variable V, qui contient "'+V+'",\nest de type:\n'+typeof V);
return; case 2: if(Temp == void 0) alert("Entrez d\'abord une valeur par le lien ci-dessus");
else{ V=new String(Temp); alert('La variable V, qui contient "'+Temp+'",\n est a present de
type : '+typeof V); } return; } } </script> <script language="javascript"> function Conc(x,y){ x
+=y; } function SimpTyp(){ var T = "GA"; var S = T; Conc(S,T); alert("S = "+S+" T = "+T); }
function AliasTyp(){ var T = new String("GA"); var S=T; Conc(S,T); alert("S = "+S+" T = "+T); }

function MaChaine(val){ this.valeur=val; } function ConcObj(x,y){ x.valeur+=y.valeur; } function ClassTyp(){ var T = new MaChaine("GA"); var S = T; ConcObj(S,T); alert("S="+S.valeur+" T="+T.valeur); } </script>

VII-A - Introduction

Nous avons, dans les chapitres qui précèdent, à maintes reprises utilisé des **chaînes de caractères** dans le sens dont nous en parlions au **§II.7**, c'est-à-dire, en tant qu'éléments de type simple. Nous allons voir dans ce chapitre que les chaînes ont pourtant l'apparence d'objets auxquels est associé ce qui semble être de nombreuses méthodes de traitement (modification d'apparence, extraction, recherche de caractère, de motif, etc.). En tout cas, la syntaxe servant à les invoquer suit exactement celle d'objets bénéficiant de propriétés et de méthodes… Qu'en est-il exactement ?

Directement liées aux chaînes de caractères, nous présenterons ensuite le formalisme lié aux **expressions régulières**, l'objet utilisant ce formalisme pour établir des modèles ainsi que les méthodes et propriétés qui sont liées.

VII-B - Finalement, les chaînes de caractères sont-elles des objets à part entière ?

En préambule, et pour se faire rapidement une opinion sur le sujet, nous allons utiliser l'opérateur **typeof** sur une chaîne. Vous allez donner une valeur quelconque dans une fenêtre de dialogue (qui a la propriété de toujours restituer une chaîne). Cette valeur sera affectée à une variable, V dont, en retour, vous obtiendrez le type.

Allons-y...

On voit bien que l'on n'obtient pas le type **Object**, comme ce serait effectivement le cas si cette chaîne était véritablement un objet de type composé, mais un type spécifique : **string**.

Nous allons à présent redéfinir la variable par **V=new String(<*la valeur que vous venez d'entrer*>)** et voyons ce que donne

 typeof V

En fait, à chaque type primitif, que ce soit pour les chaînes, mais aussi pour les nombres et les booléens, correspond une classe d'objets composés celle-ci constituant une **enveloppe** du type primitif correspondant. Pour une variable de type primitif, son *enveloppe* va contenir, outre la valeur affectée à la variable, toutes les propriétés et méthodes de la classe correspondante. Ainsi, si l'on définit une variable de type simple et que l'on essaie au travers d'elle d'accéder à des propriétés ou méthodes de la classe correspondant à son type, JavaScript opère une conversion implicite et l'objet ainsi créé est utilisé en ses lieu et place. Cet objet ainsi créé est temporaire. Dès qu'il n'est plus utilisé, sa place est récupérée et il n'existe donc plus.
Cette transformation s'opère aussi en sens inverse : si nous définissons un objet de type String et que nous désirons le faire intervenir dans une expression de concaténation de chaînes utilisant l'opérateur +, l'objet est transformé vers son alias de type primitif afin que l'expression soit correctement évaluée.

Ce mécanisme de transformation peut sembler très élégant, mais s'opérant au coup par coup, on peut craindre une perte d'efficacité. Mais que l'on se rassure (!…), JavaScript fait cette transformation très rapidement.

On a vu, toujours au **§II.7**, la distinction qui se faisait entre objets de type simple et objets de classe qui, pour les uns, étaient traités par valeur et, pour les autres, par référence. À présent que l'on sait que les objets de type simple ont leur alter ego en tant qu'objet de classe, on peut craindre un comportement différent selon que l'on utilise l'une ou l'autre forme… Voyons ce qu'il en est au travers d'un exemple.

Voici deux petits programmes totalement comparables hormis le fait que dans l'un, on utilise des variables de type simple et dans l'autre des objets. Il met en jeu des chaînes de caractères, car c'est de cela que nous traitons dans ce chapitre, mais vous pourrez vérifier que le fonctionnement serait identique avec des nombres (classe **Number()**).

```
<script language="JavaScript">
  function Conc(x,y){
```

```
<script language="JavaScript">
  function Conc(x,y){
```

```
    x+=y;
  }
  var T = "GA";
  var S = T;
  Conc(S,T);
  alert("S="+S+" T="+T);
</script>
```

```
    x+=y;
  }
  var T = new String("GA");
  var S = T;
  Conc(S,T);
  alert("S="+S+" T="+T);
</script>
```

<img width="44" height="18" border="0" align="top" src="./images/BoutEXEC.gif">

<img width="44" height="18" border="0" align="top" src="./images/BoutEXEC.gif">

On constate que les comportements sont totalement identiques et, en particulier, que les modifications opérées dans la fonction Conc ne se répercutent pas vers l'extérieur. Il s'agit bien d'un passage par valeur au sens où on l'entend habituellement. Il est heureux que les comportements soient identiques, car du fait des transformations implicites dont il a été question plus haut, si tel n'avait pas été le cas, cela aurait pu rendre les résultats aléatoires.

Voyons à présent pour un programme équivalent ce qui ce serait passé si l'on avait eu affaire à un véritable objet de classe.

```
<script language="JavaScript">
  function MaChaine(val){
    this.valeur=val;
  }
  function ConcObj(x,y){
    x.valeur+=y.valeur;
  }

  var T = new MaChaine("GA");
  var S = T;
  ConcObj(S,T);
  alert("S="+S.valeur+" T="+T.valeur);
</script>
```

<img width="44" height="18" border="0" align="top" src="./images/BoutEXEC.gif">

On constate que non seulement la valeur de **S.valeur** a bien changé, mais de plus, on s'aperçoit que celle de T.valeur a changé aussi. Est-ce un bogue ? Pas du tout ! C'est au contraire tout à fait normal si l'on se souvient que toute manipulation d'objets de classe se fait par référence. Dans cet exemple, que s'est-il passé ?

On a tout d'abord créé un objet de la classe **MaChaine** dont la propriété **valeur** a été affectée à "**GA**", puis cet objet ainsi initialisé a été affecté à la variable **T.T est donc une référence vers l'objet créé**. On a ensuite créé une nouvelle variable S à laquelle on a affecté la référence **T**. Ainsi, dès lors **S** et **T** pointent vers le seul et même objet qui vient d'être créé. Dans **ConcObj**, on modifie la propriété valeur de celui-ci via la référence S. Au passage on constate bien que cette modification sera pérenne et ne sera pas détruite au sortir de la fonction. En effet l'affichage de **S.valeur** nous le prouve. Lorsque nous allons lire pour affichage **T.valeur**, **S** et **T** pointant vers le même objet, il est donc normal que la valeur retournée soit elle aussi affectée par la modification intervenue dans **ConcObj**.

VII-C - Les propriétés et méthodes associées aux chaînes

<script language="JavaScript"> function Taille(){ var Chaine = prompt('Donnez une chaine de caracteres :'); alert('La chaine ' + Chaine + '\ncomporte ' + Chaine.length + ' caracteres'); } </ script>

Nous avons vu, dans le paragraphe précédent, que les chaînes avaient, selon leur contexte d'utilisation, soit un statut de type simple, soit celui d'objet de la classe **String**. À ce propos, signalons au passage que ce nom désigne, comme nous l'avons vu dans l'un des exemples précédents le **constructeur** de la classe. Parmi les méthodes qui seront présentées, nous distinguerons deux types : les méthodes **d'environnement** qui permettent d'obtenir une copie de la chaîne cible dans un environnement HTML et les méthodes de traitement proprement dites qui permettent diverses manipulations sur la chaîne cible elle-même.

VII-C-1 - La propriété length

Nous avons déjà rencontré ce terme lorsque nous avons étudié les tableaux. Dans ce contexte, la propriété length indiquait la taille dynamique du tableau concerné (le nombre de ses éléments de premier niveau). Nous avons vu aussi, dans ce même chapitre, qu'il existait une méthode qui, appliquée à un tableau, donnait pour résultat la chaîne composée de ses éléments séparés par un séparateur (par défaut « , » ou précisé). On voit donc que tableaux et chaînes ont des points de convergence très forts et on ne s'étonnera donc pas de rencontrer de nombreuses utilisations de cette ambivalence.
L'utilisation de cette propriété se présente sous la forme ***<chaîne>*.length**. Ici, nous vous demandons de fournir une chaîne quelconque et en retour, une fenêtre vous indiquera le nombre de caractères de celle-ci :

<img width="15" height="14" border="0" align="absmiddle" name="Bouton1" src="./images/bouton_o.gif">

VII-C-2 - Les méthodes d'environnement

Ces méthodes permettent de simplifier l'écriture d'un programme JavaScript devant générer du code HTML. Sans rentrer dans des détails inutiles qui feraient référence au langage HTML, on peut se limiter à citer :

- anchor(<nom>) ;
- big() ;
- blink() ;
- bold() ;
- fixed() ;

- fontcolor(<couleur>) ;
- fontsize(<taille>) ;
- italics() ;
- link(<reference>) ;
- small() ;
- strike() ;
- sub() ;
- sup().

VII-C-3 - La méthode charAt()

Avant toute chose, et comme nous l'avons vu pour les tableaux, il convient de préciser que le premier caractère d'une chaîne se situe à l'indice 0. De façon générale, une chaîne de n caractères (**length**) comporte des éléments depuis l'indice 0 jusqu'à l'indice n-1. La méthode dont il est ici question permet d'obtenir le **caractère de la chaîne cible dont l'indice est fourni en paramètre**. Par exemple, l'exécution de, **car="JavaScript".charAt(4)** aura pour effet d'affecter à la variable, car le caractère « S ». Si la valeur entière fournie en paramètre est en dehors de l'intervalle [0, length-1], la valeur rendue est une chaîne vide (de longueur 0).

VII-C-4 - La méthode charCodeAt()

<script language="JavaScript"> var TabN=new Array(9,12,77,31,14,54,111); var TabC=new Array("Pascal","java","C++","fortran","COBOL","algol"); var Tab=new Array(); var V; var Deux=Trois=Quatre=false; </script>

Cette méthode est semblable dans son fonctionnement à la précédente. Elle aussi à pour rôle de rechercher dans la chaîne cible le caractère dont l'indice est fourni en paramètre, mais ici, ce n'est pas le caractère lui-même qui est rendu par la méthode, mais **son code Unicode**, c'est-à-dire, une valeur entière comprise entre 0 et 65 535 (codage non signé sur 16 bits).

Nous allons montrer ici trois exemples : les deux premiers montreront des utilisations de cette méthode. Le troisième montrera l'utilisation d'une *méthode statique, propriété du constructeur String(), lui-même :* **fromCharCode()**. Celle-ci permet de **construire une chaîne de caractères à partir d'une liste de codes Unicode passée en paramètre**.

charCodeAt() & fromCharCode()...	*Résultat*
`"JavaScript".charCodeAt(3)`	 <img width="20" height="20" border="0" name="Bouton2" src="./images/bouton_o.gif">
`"JavaScript".charCodeAt(10)`	 <img width="20" height="20" border="0" name="Bouton3" src="./images/bouton_o.gif">
`String.fromCharCode(97,98,99,100)`	 <img width="20" height="20" border="0" name="Bouton4" src="./images/bouton_o.gif">

Le premier test délivre bien le code Unicode de 'a', 97 ; le second aboutit à une erreur, car l'indice 10 est au-delà des indices de caractères de la chaîne concernée ; quant au troisième, il reconstruit une chaîne à partir de la succession des codes donnés en paramètres et on retrouve bien 'a' de code 97 et les trois caractères suivants dans l'ordre alphabétique.

Notez la façon dont cette méthode est utilisée et en particulier le préfixe correspondant au nom du constructeur.

VII-C-5 - La méthode concat()

Cette méthode, comme son nom l'indique, permet de **concaténer (accoler) à la chaîne cible référencée la série de valeurs fournies en paramètre**.

Comportement de concat()…	*Résultat*
`"Java".concat("Sc",'ri',"pt")`	 <img width="20" height="20" border="0" name="Bouton5" src="./images/bouton_o.gif">
`"Java".concat("Script",1.3)`	 <img width="20" height="20" border="0" name="Bouton6" src="./images/bouton_o.gif">

On constate que la concaténation s'opère bien avec des chaînes (jusque-là, rien de bien étonnant), mais aussi avec des valeurs numériques. En effet, toute valeur numérique passée en paramètre est systématiquement transformée en chaîne de caractères, ce qui permet ainsi la concaténation.

Notons au passage que l'on peut obtenir les mêmes résultats d'une façon beaucoup plus simple : en employant l'opérateur de concaténation de chaîne +.

VII-C-6 - Les méthodes indexOf() et lastIndexOf()

<script language="JavaScript"> var Texte = "Il est certain que JavaScript est un langage simple ˆ apprendre, mais il n'en est pas pour autant rudimentaire..."; function NbOccur(){ var Compt = 0; var Deb = Texte.indexOf('e'); while (Deb != -1){ Compt++; Deb = Texte.indexOf('e',++Deb); } alert("Nbre d'occurrences de 'e' : " + Compt); } </script>
À l'opposé de **charAt()** qui permettait de déterminer le caractère apparaissant à une position particulière dans une chaîne, on pourrait imaginer une méthode permettant de déterminer l'indice de l'occurrence d'un caractère dans la chaîne référencée. En fait, cette méthode est beaucoup plus puissante que cela. Elle permet, en effet, d'**indiquer l'indice où débute une sous-chaîne fournie en paramètre dans la chaîne référencée** (bien sûr, si la sous-chaîne est réduite à un seul caractère, on obtient le résultat recherché précédemment). Mais la sous-chaîne recherchée peut apparaître plusieurs fois dans la chaîne référencée. Dans ces conditions, comment balayer toutes les occurrences sans devoir tronquer, au fur et à mesure, le début de la chaîne de référence (en supposant que la recherche s'opère de gauche à droite). La méthode comporte un second paramètre qui précise à partir de quel indice la recherche doit être entreprise. Ainsi, sans modifier la chaîne source, on pourra énumérer toutes les occurrences d'une sous-chaîne précisée et en indiquer l'indice de début.
Finalement, la forme générale d'utilisation de cette méthode est la suivante :

```
<chaine source>.indexOf(<sous chaine>[,<debut>])
```

On constate que le second argument précisant le début de la recherche est optionnel. En l'absence de celui-ci, la recherche s'opère à partir de l'indice 0. Si aucune occurrence n'est trouvée, soit parce qu'elle n'existe pas, soit parce que la valeur de début est plus grande que la longueur de la chaîne, cette méthode renvoie la valeur -1.

Dans les versions anciennes de Netscape (< 4), dans le cas où le début était hors des limites normales, la méthode retournait une chaîne vide au lieu de la valeur -1.
Voici un petit texte et le programme qui va nous servir à compter de nombre de fois où la lettre « e » apparaît…

Le texte
```
"Il est certain que JavaScript est un langage simple à apprendre, mais il n'en est pas pour autant
 rudimentaire... "
```

Le programme
```
var Compt = 0;
var Deb = Texte.indexOf('e');
while (Deb != -1) {
  Compt++;
  Deb = Texte.indexOf('e',++Deb);
}
alert("Nbre d'occurrences de 'e' : " + Compt);
```

 <img width="20" height="20" border="0" name="Bouton7" src="./images/bouton_o.gif">
Nous venons de voir la méthode **indexOf()** qui opère la recherche de gauche à droite. Il existe une autre méthode, **lastIndexOf() qui elle opère une recherche de droite à gauche**. L'utilisation de celle-ci a la même forme que la précédente ; comme elle, elle reçoit en second paramètre optionnel une valeur entière indiquant l'indice du début de recherche.

VII-C-7 - La méthode slice()

Cette méthode n'est pas sans rappeler celle, de même nom, qui a été présentée lors de l'**étude des tableaux**. Elle permet, à partir de la chaîne source référencée, d'**extraire la sous-chaîne débutant à l'indice désigné par le premier paramètre** et s'achevant à l'indice désigné par le second paramètre. Le second paramètre est optionnel. En l'absence de celui-ci, c'est toute la fin de la chaîne (à partir de l'indice indiquant le début) qui est rendue.

Par exemple, **"JavaScript".slice(4,6)** rend la valeur **'Sc'**, **"JavaScript".slice(4)** rend la valeur **'Script'**. Ces deux exemples montrent qu'alors que le caractère désigné par l'indice de début fait partie de la sous-chaîne rendue, celui désigné par l'indice de fin n'en fait pas partie. Les valeurs données en paramètres peuvent être négatives. Dans ce cas, elles indiquent des indexations partant de la fin de la chaîne. On doit interpréter cette valeur, x, négative, par length + x. Autrement dit, -1 indique le dernier caractère (indice length-1), -2, l'avant-dernier (indice length-2), etc.

Il est à noter que le fait d'attribuer une valeur négative au paramètre de début ne fonctionne pas sous Internet Explorer. Ce dernier ramène cette valeur à 0 et c'est donc tout le début de la chaîne qui est rendu.

Voyons quelques utilisations directes de cette méthode :

Comportement de slice()…	*Résultat*
`"JavaScript".slice(4)`	 <img width="20" height="20" border="0" name="Bouton8" src="./images/bouton_o.gif">
`"JavaScript".slice(4,6)`	 <img width="20" height="20" border="0" name="Bouton9" src="./images/bouton_o.gif">
`"JavaScript".slice(6,4)`	<a onclick="alert("JavaScript".slice(6,4));return

	false" href=""> <img width="20" height="20" border="0" name="Bouton10" src="./images/bouton_o.gif">
`"JavaScript".slice(3,-3)`	 <img width="20" height="20" border="0" name="Bouton11" src="./images/bouton_o.gif">
`"JavaScript".slice(-5,-1)`	 <img width="20" height="20" border="0" name="Bouton12" src="./images/bouton_o.gif">
`"JavaScript".slice(-1,-5)`	 <img width="20" height="20" border="0" name="Bouton13" src="./images/bouton_o.gif">

Les exemples 1, 2 et 4 fonctionnent bien comme annoncé précédemment. Sous Netscape, les exemples 3 et 6 provoquent une erreur, car l'indice de début est supérieur à celui de fin ; par contre, sous Internet Explorer, 3 retourne une chaîne vide, tandis que 6 s'évalue en fonction de la remarque faite plus haut, l'indice de début (-1) étant remplacé par 0. Cette particularité d'Internet Explorer se manifeste de la même façon pour l'exemple 5 qui se comporte donc lui aussi de manière différente selon le navigateur utilisé.

VII-C-8 - La méthode split()

<script language="JavaScript"> function NbMots(){ return 'Nbre de mots : '+Texte.split(' ').length; } </script>

Ces deux méthodes permettent, comme **slice()**, d'**extraire une sous-chaîne de la chaîne source. substring()** constitue l'ancêtre de **slice()** en ce sens qu'elle apparaissait déjà dans JavaScript 1.0 alors que slice() a été introduite dans JavaScript 1.2. Elle fonctionne exactement de la même manière (le caractère d'indice **<début>** fait partie de la sous-chaîne extraite alors que le caractère d'indice **<fin>** en est exclu), mais **elle ne supporte pas les paramètres négatifs** comme nous l'avons vu pour **slice()**.

La méthode **substr()** qui est apparue dans JavaScript 1.2 accepte en paramètre une, voire deux valeurs entières, la première indiquant, comme pour les précédentes, l'indice de début de la sous-chaîne à extraire, la seconde, facultative, indiquant la **longueur** de celle-ci. Toujours comme pour les précédentes, en l'absence de ce second paramètre, c'est toute la fin de la chaîne qui est extraite.

VII-C-9 - Les méthodes substring() et substr()

Nous avons étudié dans le chapitre précédent la méthode **join()** de la classe **Array()**. Rappelons que celle-ci permet de transformer un tableau en une chaîne de caractères dans laquelle les éléments du tableau source sont séparés, par défaut, par le caractère « , » ou, si elle est précisée, par une chaîne quelconque. La méthode **split()** est en quelque sorte la « méthode inverse » puisque **à partir de la chaîne source référencée** et en la découpant sur la base d'un séparateur fourni en paramètre, **elle construit un tableau** dont les éléments sont les sous-chaînes ainsi séparées. Précisons que le séparateur peut être une chaîne de caractères éventuellement vide. Dans ce cas particulier, la chaîne est découpée caractère par caractère. Il peut aussi prendre la forme d'une expression régulière, mais nous reviendrons là-dessus plus tard.

Pour montrer un exemple d'utilisation de cette méthode, reprenons le texte utilisé plus haut et calculons le nombre de mots.

```
"Il est certain que JavaScript est un langage simple à apprendre, mais il n'en est pas pour autant
 rudimentaire…"
```

```
var Compt = Texte.split(' ').length;
alert("Nbre de mots: " + Compt);
```

 <img width="20" height="20" border="0" name="Bouton14" src="./images/bouton_o.gif">
Efficace, n'est-ce pas ?… Oui, mais approximatif ! En effet, on n'est pas à l'abri de caractères d'espacement (' ') doublés, ce qui aurait pour conséquence de « créer » des mots fantômes. Par ailleurs, les élisions ne sont pas prises en compte ici : « n'en » compte pour un seul mot ce qui pour les puristes peut paraître une aberration. Nous verrons plus loin que l'utilisation d'une expression régulière nous permettra de pallier ces travers.

VII-C-10 - Les méthodes toLowerCase() et toUpperCase()

Enfin, pour en terminer, voici deux méthodes permettant de « normaliser » des chaînes de caractères, lorsque cela est nécessaire, en rendant une **copie de la chaîne source dont tous les caractères alphabétiques** sont convertis en **minuscules** pour l'une, en **majuscules** pour l'autre.

À titre d'exemple d'utilisation, on peut imaginer que parmi les données saisies dans un formulaire, certaines (initiale de prénom, nom, ville, etc.) puissent être systématiquement converties en minuscules. On peut aussi reprendre l'exemple précédent dans lequel on recherchait le nombre d'occurrences d'une certaine lettre par la méthode **indexOf()**. Dans cet exemple, on fournissait en paramètre une lettre minuscule, mais pour prévoir l'éventualité d'occurrence de cette même lettre sous la forme d'une majuscule, on aurait dû opérer cette recherche sur une copie du texte où toutes les lettres auraient été transformées en minuscules.

Il y a d'autres méthodes de la classe String, mais nous avons choisi de ne les présenter que dans le prochain chapitre, car, mettant en jeu les expressions régulières, il serait hasardeux d'en parler avant d'avoir pris connaissance de celui-ci.

VIII - Les expressions régulières

<script language="JavaScript"> var Bouton_OFF; var Bouton_ON; onload=Declare(); var NS=navigator.appName.indexOf('Nets')>=0; </script>

VIII-A - Structure des expressions régulières

En JavaScript, les expressions régulières, dont nous allons voir la grande utilité au travers des exemples qui émailleront ce paragraphe, constituent une classe d'objets, la classe **RegExp**. Nous verrons qu'elles agissent sur des chaînes de caractères pour permettre de les **analyser**, les **filtrer** et de chercher des **motifs** contenus dans celles-ci avec un niveau de précision adapté à la problématique (du plus grossier au plus fin). Notons que ce concept d'expressions régulières, très fortement inspiré (c'est un euphémisme) de celui du langage Perl, n'est apparu dans JavaScript que depuis sa version 1.2.

VIII-A-1 - Comment définit-on des expressions régulières ?

Un objet « expression régulière » peut être introduit et éventuellement défini grâce au constructeur de la classe, **RegExp()**, en écrivant simplement une instruction du type **var R = new RegExp()**. Dans cet exemple, l'expression régulière R existe, mais aucune valeur ne lui a été affectée. Comme pour les constructeurs que nous avons déjà rencontrés, on aurait pu définir sa valeur en la précisant, sous la forme d'une chaîne de caractères, en paramètre du constructeur. Un tel objet peut aussi être introduit de façon littérale en utilisant des marqueurs spécifiques délimitant l'expression : '/'. Nous pourrions ainsi avoir : **var R = /a/**. (Cette expression, dans le cadre de l'analyse d'une chaîne impose simplement que celle-ci contienne le caractère a). Une écriture équivalente utilisant le constructeur eut été : **var R : RegExp("a")**.

Dans une expression régulière, outre les caractères devant se retrouver physiquement dans la ou les chaînes qui seront traitées par celle-ci, on rencontre un nombre non négligeable de caractères « outils » dont il est important de bien saisir la sémantique pour pouvoir les utiliser de façon pertinente et réaliser ainsi des modèles efficaces et concis. Ces caractères sont de trois types : les caractères définissant des **littéraux**, les caractères **d'ensemble**, les caractères de **groupement** et enfin les caractères de **répétition**.

VIII-A-2 - Les littéraux

Ces caractères contiennent, outre tous les caractères alphanumériques, une représentation des caractères non imprimables (tabulation, retour chariot, saut de ligne, saute de page, etc.) ainsi que des caractères qui étant utilisés en tant que caractères « outils » doivent néanmoins avoir une représentation différente pour pouvoir être analysés dans une chaîne. Ces caractères littéraux seront en introduit par le caractère « backslash » : \.

En voici la liste :

Caractère	Signification
Alphanumérique	Lui-même

\ /
\ \
\ .
\ +
\ *
\ ?
\ |
\ }
\ {
\ (
\ )
\ [
\ ]
	/
	\
	.
	+
	*
	?
	|
	}
	{
	(
	)
	[
	]

\t \r \n \f \v	tabulation horiz. retour chariot saut de ligne saut de page tabulation vert.
\xxx \xhh	car. ASCII de code octal xxx, car. ASCII de code hexa hh

À titre d'exercice, vous allez donner un texte en entrée en tâchant d'y inclure la lettre dont le code ASCII octal 112 est attendu…

essai

Si vous voulez un peu tricher, consultez une table des codes ASCII… Mais elle est souvent en décimal et hexadécimal ! Ce sera une occasion pour jongler entre les bases de numération !!! (On n'a rien sans rien ;-))

VIII-A-3 - Les caractères d'ensemble

Les caractères d'ensemble permettent de **construire** à la demande une **collection de caractères** ou de **désigner des collections prédéfinies**. Ils pourront indiquer que l'un quelconque des caractères qu'ils représentent doit apparaître ou, au contraire, qu'aucun des caractères qu'ils représentent ne doit apparaître.
Il faut ajouter que dans le cas où les codes des caractères de la collection à désigner constituent une suite, on pourra représenter cette suite par seulement **le premier et le dernier caractères séparés par un tiret**. Citons, par exemple

les caractères de l'alphabet majuscules (**[A-Z]**), minuscules (**[a-z]**), les chiffres décimaux (**[0-9]**), hexadécimaux (**[0-9A-F]**), etc.

Si, au contraire on désire représenter un caractère pouvant être quelconque hormis appartenir à un ensemble parfaitement cerné, on utilisera le caractère de **complémentarité** ^ en début d'ensemble ; par exemple **[^!?.,;]** désigne tout caractère qui n'est pas un signe de ponctuation. Voici la liste de ces caractères :

Caractère	Signification
[….]	Un qcq des caractères contenus
[^….]	Aucun des caractères contenus
.	Caractère qcq sauf saut de ligne
\s	équivaut à [^ \n]
\S	Tout caractère de césure
\w	équivaut à [\t\r\n\f\v]
\W	Aucun caractère de césure
\d	équivaut à [^ \t\r\n\f\v]
\D	Tout caractère alphanumérique
	équivaut à [a-zA-z0-9]
	Aucun caractère alphanumérique
	équivaut à [^a-zA-z0-9]
	Tout chiffre (décimal)
	équivaut à [0-9]
	Aucun chiffre (décimal)
	équivaut à [^0-9]

Si, par exemple, on désire analyser le mot JavaScript contenant ou pas les majuscules et encadré de deux caractères de césure, nous définirons ainsi le modèle : **/\s[Jj]ava[Ss]cript\s/**. *Attention : avec cette expression on accepte l'écriture **JavaScript** qui n'est pas gênante, mais aussi **javaScript** qui peut ne pas être souhaitée.*

À l'intérieur d'un ensemble (ou d'un complémentaire d'ensemble), pour représenter des « caractères outils » du premier sous-groupe ("/", "+", "", "?", "(", ")", etc.), point n'est besoin de les faire précéder du caractère "\". Les seuls qui doivent l'être (car ils introduisent une ambiguïté) sont : "\" et "]"*

VIII-A-4 - Les caractères de groupement et référencement

Le caractère de groupement a pour fonction de **regrouper plusieurs éléments constituant un sous-motif du modèle en un seul élément**. Le groupement s'opère simplement en l'encadrant d'une paire de parenthèses. Il y a plusieurs utilités à pouvoir regrouper des éléments :

- on peut ainsi faire **porter les caractères de répétition** que nous verrons plus loin sur l'ensemble des éléments du modèle ainsi regroupés. Ainsi, c'est cet ensemble qui sera dupliqué et non les éléments constitutifs individuellement ;
- on peut aussi **référencer ce groupement** de façon à prévoir une autre occurrence des éléments de la chaîne analysés avec lesquels il a été apparié, plus loin dans le modèle.

Le référencement dont il vient d'être question s'opère par l'apparition dans l'expression régulière du caractère **\n** où n désigne le n° de parenthésage auquel il fait référence. Pour illustrer d'un exemple simple, supposons que l'on veuille analyser la chaîne JavaScript encadrée de simples ou doubles guillemets (les deux devant être identiques bien entendu).

Si l'on analyse à l'aide de l'expression régulière **/[""]JavaScript[""]/**, les écritures **'JavaScript"** et **"JavaScript'** seront reconnues correctes, ce que nous refusons. En fait, l'expression d'analyse devra être de la forme : **/([""])JavaScript \1/**.

Essayez
Grâce à cet essai, on constate que cette expression autorise en début de chaîne le caractère « ' » ou le caractère « " », mais une fois le premier analysé, il contraint le second à lui être identique.

VIII-A-5 - Les caractères de répétition

<script language="JavaScript"> function TestQuot(){ OK=(/^\s*([""])([^\1]*)\1\s* $/.exec(prompt("Donnez une chaine encadree de guillemets simples ou doubles",""))==null?'Incorrecte':'Correcte'; return OK; } </script>
Ce sont des caractères dont la fonction est de prévoir dans le modèle un nombre d'occurrences successives du littéral, du groupement ou d'élément d'ensemble sur lequel ils portent.

Caractère	Signification
*	Un nombre indéfini de fois
+	[0,x], x >= 0
?	Au moins une fois [1,x], x > 0
{n}	Éventuellement une fois [0,1]
{n,m}	Exactement n fois
{n,}	Au moins n et au plus m fois [n,m]
	Au moins n fois [n,x], x >= n

Chacun sait que tout individu français est immatriculé par un numéro INSEE plus connu pour certains d'entre eux (pas pour tous, hélas) sous le nom de numéro « sécu ». Ce numéro comporte un formatage bien précis : un chiffre pour le sexe (1 ou 2), deux pour l'année de naissance, deux pour le mois, deux pour le département de naissance, trois pour la ville, trois pour l'ordre, tous ces champs étant séparés par un espace, puis un slash (/) et enfin deux chiffres pour la clé. Vous allez proposer une expression régulière, la plus courte possible, permettant d'opérer l'analyse de ce type d'information, puis vous la testerez contre votre numéro INSEE afin de vérifier soit l'un, soit l'autre…

Veillez à bien refermer parenthèses, crochets, accolades, etc. si vous en utilisez, sinon, sous Netscape, la compilation de votre expression provoquera une erreur. Si d'aventure cela se produit, remontez simplement à la fenêtre précédente de l'historique du navigateur. <img width="44" height="18" border="0" align="top" src="./images/BoutEXEC.gif">

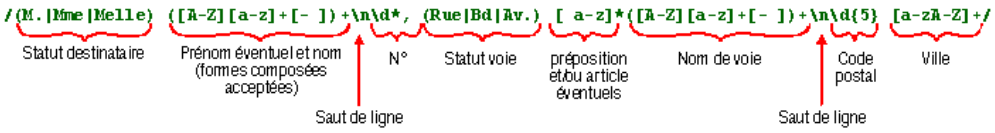
VIII-A-6 - Le caractère de choix

<script language="JavaScript"> var Model=new RegExp(); function TryINSEE(){ var INSEE; var StrExp; var Test="1 11 11 11 111 111/11"; var verif=true; var OK=true; Compt=0; do{ do{ StrExp=prompt("Donnez l'expression reguliere :",Model) } while (StrExp==null); Compt ++; Model=Model.compile(StrExp.replace(/^([\^\/]+)([^\$]+)([\$\/]+)$/,"$2")); while (INSEE==null) INSEE=prompt("Donnez votre n¡ INSEE",INSEE); if (!(Model.test(INSEE) || Model.test(Test))) { alert("C'est faux,\nvous allez devoir proposer une autre expression !"); if (Compt == 4) if (confirm("Peut-etre en avez-vous assez...\n\nVoulez-vous malgre tout continuer ?")) Compt=0; else break; verif = (verif && confirm("Votre n¡ INSEE est : "+INSEE+"\nVoulez-vous le modifier ?")); if (verif) INSEE=prompt("Donnez votre n¡ INSEE",INSEE); } else { OK=false; alert

("C'est bien, \nune expression type aurait pu \u00eatre :\n/[12]\\s*\\d{2}\\s*[01]\\d\\s*\\d{2}(\\s*\\d{3}){2}\s*\\/\\s*\\d{2}/"); } } while (OK); } </script>

Le caractère de choix **permet de prévoir**, dans l'expression régulière, **le choix entre différents sous-motifs**, séparés par le caractère « | ». Comme pour les caractères de répétitions, celui-ci peut porter sur un ou plusieurs littéraux, ensembles ou groupements.

Par exemple, pour analyser une adresse, on peut imaginer l'expression suivante :



Dans cette expression apparaissent plusieurs groupements. Parmi eux, certains constituent des choix (statut du destinataire, de la voie), d'autres permettent de faire porter un caractère de répétition sur plusieurs sous-motifs successifs. Par exemple, 'identité du destinataire est composée d'au moins un mot, son nom (d'où le caractère de répétition + qui impose au moins 1) ; ce nom peut être composé ; il peut être précédé ou suivi d'un prénom, lui-même pouvant être composé chacun de ces éléments répondant au même modèle. Avec une telle expression, on peut analyser une adresse du type :

```
M. Jean-Claude Vandamme
17, Rue de la Castagne
31500 TOULOUSE
```

VIII-A-7 - Les caractères de positionnement

Nous avons seulement vu, jusqu'ici, des appariements de caractères. Il existe aussi la possibilité **d'apparier** et donc, de contraindre encore, **selon la position dans la chaîne**. Cela est d'un grand intérêt, car on va pouvoir mettre en place des ancrages de motifs. Prenons l'exemple d'un identificateur qui doit commencer forcément par une lettre éventuellement suivie de lettres ou de chiffres.
Vous seriez tenté de proposer, avec beaucoup d'assurance, l'expression régulière suivante : **/[A-Za-z]\w*/**. Considérons donc la chaîne **03Trois1**. Je suis au regret de vous dire que l'appariement va réussir et donc que **03Trois1** sera bien pris comme un identificateur !!!

En fait pour toutes les expressions régulières que l'on a vues jusqu'ici, la vérification se contentait de tester si dans la chaîne fournie, une partie de celle-ci répondait au modèle décrit par l'expression régulière. Si vous avez réussi à faire l'exercice sur le n° INSEE, proposez à présent quelque chose du style **"Tartempion *1 85 06 13 055 312/51*.. Ouf!!!"**. Vous allez constater que tout baigne !!! Précisément, la partie centrale qui apparaît ici en italique aura répondu au modèle. De la même manière, dans **03Trois1**, c'est en fait **Trois1** qui aura constitué un parfait identificateur !!

Il faudrait donc contraindre l'analyse à appliquer le modèle dès le début de la chaîne afin que **[A-Za-z]** ne pouvant s'apparier à **0 , 03Trois1** ne soit pas retenu comme identificateur. Voici ces caractères de positionnement :

Caractère	Signification
^	Ancre en début de chaîne
$	Ancre en fin de chaîne
\b	Ancre sur limite de mot
\B	(entre \w et \W) Ancre sur non-limite de mot

Ainsi, en utilisant l'expression **/^[A-Za-z]\w*/**, l'analyse d'un identificateur sera pertinente. Pour le numéro INSEE, et seulement lui, on utilisera le caractère ^ en début d'expression tandis que le caractère $ la terminera.

Ces caractères de positionnement, dont on comprend bien l'énorme importance, sont hélas, trop peu nombreux. Ce défaut disparaîtra avec JavaScript 1.3 qui permettra de définir le type de positionnement que l'on veut. Par exemple

(?=[0-9]) permet de définir un ancrage avant un chiffre, tandis que **(?![0-9])** désigne un ancrage avant tout caractère qui ne soit pas un chiffre **(Â(?=[^0-9]))**.

***Ne pas confondre le caractère d'ancrage en début avec le caractère de complémentarité**. Même s'ils ont une représentation similaire, il n'y a aucune ambiguïté, car l'un se situe obligatoirement en début d'expression (après /) tandis que l'autre ne peut apparaître qu'en début de définition d'ensemble (après [).*

VIII-A-8 - Les attributs d'appariements

Les attributs d'appariement sont au nombre de deux. Ils se placent après l'expression régulière sur laquelle ils portent.

On dispose de l'attribut **i** qui indique s'il est présent que l'analyse de la chaîne doit se faire sans tenir compte de la casse des caractères. Cela permet, entre autres, de simplifier les expressions et si nous reprenons les exemples précédents, l'expression décrivant un identificateur pourra s'écrire **/^[A-Z]\w*/**, celle des nombres hexadécimaux, **/^[A-F0-9]+$/i**.

Nous allons présenter dans les prochains paragraphes des méthodes permettant, sur la base d'une expression régulière, de rechercher des sous-chaînes de la chaîne de caractères source. En présence de l'attribut g, ce sont effectivement toutes les portions s'appariant avec l'expression qui seront recherchées. Dans le cas contraire, la recherche s'arrêtera dès la première rencontrée.

VIII-B - Les propriétés et méthodes de la classe RegExp

VIII-B-1 - La propriété lastIndex (non implanté sous IE)

C'est une propriété accessible en lecture/écriture dans laquelle est enregistré, dans le cas de recherches globales, l'**indice dans la chaîne source, à partir duquel la recherche devra reprendre**. Cette propriété est en particulier utilisée par les méthodes **test()** et **exec()** que nous allons voir.

VIII-B-2 - La propriété source

Accessible en lecture seulement, cette propriété est une **chaîne de caractères contenant le texte de l'expression régulière référencée**. Par exemple, si vous avez fourni une expression régulière, dans l'exercice sur le numéro INSEE, vous pouvez la revoir en cliquant

ici
.

VIII-B-3 - Les propriétés global et ignoreCase (non implanté sous IE)

Ces deux propriétés - qui sont elles aussi accessibles seulement en lecture -sont deux booléens permettant de récupérer la **valeur des attributs de l'expression régulière référencée**. Si **global** est vrai, cela signifie que la recherche d'appariements est globale à toute la chaîne ; si **ignoreCase** est vrai il n'est pas tenu compte de la casse des caractères.

Nous allons voir à présent les propriétés statiques de la classe **RegExp**. Rappelons qu'à la différence des propriétés d'instance qui s'appliquent à tout objet instance de la classe, chaque propriété statique, est unique dans la classe et donc commune à tous les objets de cette classe.

VIII-B-4 - Les propriétés RegExp.leftContext et RegExp.rightContext (non implanté sous IE)

À chaque fois qu'une recherche d'appariement entre une expression régulière et une chaîne de caractères est opérée, par des méthodes que nous verrons plus loin, qu'elles soient de la classe RegExp ou de la classe String, ces indicateurs seront positionnés. **Le premier contiendra la partie de la chaîne située à gauche du dernier appariement** effectué, tandis que le **second contiendra la partie droite**.

À noter que ces propriétés sont équivalentes respectivement à RegExp["$`"] **et** RegExp["$'"].

VIII-B-5 - Les propriétés RegExp.lastMatch et RegExp.lastParen (non implanté sous IE)

Les deux précédentes propriétés permettent de récupérer les contextes gauche et droit du dernier appariement opéré. Ce qui se trouve entre les deux, c'est-à-dire **la partie de la chaîne source mise en concordance avec l'expression régulière sera disponible dans la propriété lastMatch**.

À l'intérieur de la sous-chaîne appariée contenue dans **lastMatch**, on peut de plus dégager la **partie qui s'est appariée au dernier groupement du précédent appariement**. Celle-ci est contenue dans lastParen. Il est intéressant de noter que l'on peut obtenir les sous-chaînes mises en appariement avec les neuf premiers groupements par l'intermédiaire des propriétés **RegExp.$1, RegExp.$2…, RegExp.$9.**

À noter aussi que ces propriétés sont équivalentes respectivement à RegExp["$&"] **et** RegExp["$+"].

Un bon exemple valant plus que de longs discours, étudiez le comportement de ces différentes propriétés sur la base d'une expression régulière de la forme **/([A-Z]([a-z]+)\d([,.]))/g** et d'une chaîne de caractères ayant pour valeur : **"Un1,Deux2,Trois3,Quatre4,Cinq5."**.

<img width="44" height="18" border="0" align="top" src="./images/BoutEXEC.gif">

VIII-B-6 - La propriété RegExp.multiline (non implanté sous IE)

<script language="JavaScript"> function Context1(){ if(!NS) {alert('Fonctionnement impossible sous Internet Explorer, car les diverses propriétés de classe ne sont pas implantées.'); return} var Texte="Un1,Deux2,Trois3,Quatre4,Cinq5.";S=Texte; var Model = /([A-Z]([a-z]+)\d([,.]))/g; Model.lastIndex=0; while (Model.test(Texte)) with (RegExp){ S= "lastIndex = "+Model.lastIndex+"\nleftContext = "+leftContext+"\nrightContext = "+rightContext +"\nlastMatch = "+lastMatch+"\nlastParen = "+lastParen; S+="\n$1 = "+$1+" $2 = "+$2+" $3 = "+$3+"\n$4 = "+$4+" $5 = "+$5+" $6 = "+$6; alert(S); } } function Context2(num,ml){ if(!NS) {alert('Fonctionnement impossible sous Internet Explorer, car les diverses propriétés de classe ne sont pas implantées.'); return} var Texte="Un1,\nDeux2,Trois3,\nQuatre4,Cinq5."; var Model = num==1 ? /^([A-Z]([a-z]+)\d([,.]))+$/g : /^([A-Z]([a-z]+)\d([,.]\n?))+$/g; with (RegExp) { multiline=ml; if (!Model.test(Texte)) alert("Dans cette configuration, aucun appariement n'est possible !"); else { Model.lastIndex=0; while (Model.test(Texte)) { S= "lastIndex = "+Model.lastIndex+"\nleftContext = "+leftContext+"\nrightContext = "+rightContext+"\nlastMatch = "+lastMatch+"\nlastParen = "+lastParen; S+="\n$1 = "+$1+" $2 = "+$2+" $3 = "+$3+"\n$4 = "+$4+" $5 = "+$5+" $6 = "+$6; alert(S); } } } } </script>

Ce booléen permet de préciser si la chaîne sur laquelle agit l'expression régulière contient une seule ou plusieurs lignes. Selon le cas, le comportement pourra être différent et c'est en particulier le cas pour les ancrages de début et de fin. Dans le cas où le texte n'est pas précisé multiligne, ^ représente le début de la chaîne et $, la fin de la chaîne, même s'il contient des littéraux \n. Dans le cas où ce même texte est déclaré multiligne, ^ représente le début et $, la fin **de chaque ligne**. Modifions légèrement les chaînes et expressions régulières du précédent exemple.

La chaîne utilisée va contenir plusieurs lignes : **"Un1,\nDeux2,Trois3,\nQuatre4,Cinq5."**.

Par ailleurs, l'expression régulière va pouvoir revêtir deux formes :

```
Forme 1 = /^([A-Z]([a-z]+)\d([,.]))+$/g  et  Forme 2 = /^([A-Z]([a-z]+)\d([,.]\n?))+$/g
```

Enfin, nous allons considérer le cas où **multiline** est vrai et celui où **multiline** est faux… Voyons ce que cela donne…

	Forme 1	**Forme 2**
multiline VRAI	<img width="20" height="20" border="0" align="top" name="Bouton0" src="./images/bouton_o.gif">	<img width="20" height="20" border="0" align="top" name="Bouton1" src="./images/bouton_o.gif">
multiline FAUX	<img width="20" height="20" border="0" align="top" name="Bouton2" src="./images/bouton_o.gif">	<img width="20" height="20" border="0" align="top" name="Bouton3" src="./images/bouton_o.gif">

À noter que cette propriété est équivalente à RegExp["$*"].

VIII-B-7 - La méthode compile(<chaîne>[, <attributs>])

Lorsque dans un script interviennent plusieurs expressions régulières, soit qu'elles soient fournies en paramètre soit même par l'utilisateur comme ce fut le cas dans l'exercice sur le numéro INSEE, plutôt que de créer pour chacune un objet **RegExp**, il est sûrement avantageux d'utiliser, si cela s'y prête, un seul objet. Dans ces conditions, les expressions régulières successives qui lui seront affectées seront tout d'abord fournies sous la forme de chaîne qu'il faudra donc transformer ensuite en un objet de type **RegExp**. **Cette transformation d'une chaîne de caractères vers une expression régulière est opérée par la méthode compile()**. On peut déplorer que cette méthode ne récupère pas les erreurs de syntaxe qui peuvent apparaître dans le modèle fourni sous la forme de chaîne. En particulier, une erreur de parenthésage dans les groupements rend Netscape fou de rage !…

Les deux méthodes de la classe **RegExp** qu'il nous reste à voir ont une utilité voisine et complémentaire. Les deux opèrent un appariement entre l'expression régulière référencée et la chaîne de caractères passée en paramètre, mais l'une renvoie seulement un booléen signifiant qu'au moins un appariement a pu avoir lieu, alors que l'autre renvoie, sous la forme d'un tableau, des informations plus complètes sur les sous-expressions appariées.

Ces deux méthodes affectent et utilisent deux propriétés qui n'ont pas été citées plus haut : **index** et **input**. **index** contiennent la position du caractère de la chaîne à partir duquel l'appariement a eu lieu et **input** fait référence à la chaîne elle-même. Si l'une ou l'autre de ces méthodes est rappelée sans paramètre, par défaut, c'est la chaîne référencée par **input** qui sera utilisée.

VIII-B-8 - La méthode test(<chaîne>) (bogué sous IE)

Cette méthode essaie d'opérer un appariement entre l'expression régulière référencée et la chaîne de caractères donnée en paramètre à partir de l'indice spécifié par la propriété **lastIndex**. Si un appariement est possible, **lastIndex** est réactualisé et la valeur booléenne renvoyée est **true**. Dans le cas contraire, la valeur renvoyée est **false** et **lastIndex** est remis à 0.

VIII-B-9 - La méthode exec(<chaîne>) (bogué sous IE)

L'autre méthode d'appariement s'évalue sur le même type de donnée que la précédente (expression régulière référencée et chaîne de caractères en paramètre). Comme elle, elle essaie donc de procéder à l'appariement de ces deux données. En cas d'échec, la valeur de retour est **null**. Dans le cas contraire, la valeur retournée est un tableau contenant en 0, la sous-chaîne appariée avec l'expression régulière et dans les éléments suivants les sous-chaînes appariées avec les éventuels groupements parenthésés. Par ailleurs, en cas d'appariement, cette méthode actualise **lastIndex**, si bien que si elle est rappelée avec la même référence, elle procède à une nouvelle recherche à partir de cet emplacement. Comme pour la méthode précédente, en cas d'échec, **lastIndex** est remis à 0.

VIII-C - Les méthodes de la classe String mettant en jeu les expressions régulières

VIII-C-1 - La méthode match(<expr. régul.>)

<script language="JavaScript"> function Visible(Calque,i){ with(document){ if(NS && parseInt(navigator.appVersion)<5) Calque.visibility=(i?'show':'hide'); else getElementById(Calque).style.visibility=(i?'visible':'hidden'); } } function Exec_Match(E_M,g) { var NSReg=NS && navigator.appVersion.indexOf('Safari')==-1; var Tab; var Texte="Un1, \nDeux2,Trois3,\nQuatre4,Cinq5."; var Model= g ? /([A-Z]([a-z]+)\d([,.]))+/g : /([A-Z]([a-z]+)\d([,.]))+/; switch (E_M) { case 'E' : if(!NSReg && g) alert('Le fonctionnement n\'est pas satisfaisant sous Internet Explorer, car diverses propriétés de classe ne sont pas implantées.'); if(NS) lastIndex=0; Tab=Model.exec(Texte); break; case 'M' : Tab=Texte.match(Model); break; } if (Tab==null) alert("Dans cette configuration, aucun appariement n'est possible !"); else{ S=""; for(var i=0;i<Tab.length;i++) S+= "En "+i+" : "+Tab[i]+"\n"; alert(S); } } </script>

Cette méthode qui est à rapprocher de la méthode exec() que l'on vient de voir, hormis le fait que celle-ci fait référence à une chaîne et a pour paramètre une expression régulière. Elle permet de **rechercher dans la chaîne référencée la ou les portions de celle-ci qui répondent à un modèle** (expression régulière) fourni en paramètre. Dans le cas où aucune portion de la chaîne ne répond au modèle, la valeur retournée est **null**. Dans le cas contraire, la valeur rendue est un tableau construit de façon différente selon que l'on ne recherche qu'une (la première à gauche) mise en concordance ou chacune d'elles. Dans le premier cas, le premier élément du tableau comporte la sous-chaîne répondant au modèle et les éléments suivants contiennent les sous-parties correspondant aux éventuelles sous-expressions parenthésées du modèle. Dans le cas où toutes les concordances sont recherchées, le tableau rendu se limite à chacune des parties de la chaîne source répondant au modèle.

On voit donc que la similitude que l'on trouvait au début entre **exec()** et **match()** n'est pas totale puisque alors que **exec()** n'évalue qu'un seul appariement, **match()** effectue tous les appariements que l'expression autorise sur la chaîne passée en paramètre. En particulier, l'attribut g prendra pleinement sa signification dans **match()** ce qui n'est pas le cas dans **exec()**.

Dans l'exemple qui suit, nous allons comparer les comportements de **match()** et **exec()** dans un contexte de recherche globale ou pas.

Le programme

```
var Tab;
var Texte="Un1,\nDeux2,Trois3,\nQuatre4,Cinq5.";
```

Le programme
```
var Model=/([A-Z]([a-z]+)\d([,.]))+/g;
Tab = Model.exec(Texte);

if (Tab==null) alert("Aucun appariement n'est possible !");
else {
  S="";
  for(var i=0;i<Tab.length;i++)
    S+= "En "+i+" : "+Tab[i]+"\n";
  alert(S);
}
```

Ou

Le programme sans attribut g
```
var Tab;
var Texte="Un1,\nDeux2,Trois3,\nQuatre4,Cinq5.";

var Model=/([A-Z]([a-z]+)\d([,.]))+/;
Tab = Texte.match(Model);

if (Tab==null) alert("Aucun appariement n'est possible !");
else {
  S="";
  for(var i=0;i<Tab.length;i++)
    S+= "En "+i+" : "+Tab[i]+"\n";
  alert(S);
}
```

Voici le canevas du programme qui va être utilisé, dans lequel, est mise en évidence (enfin, j'espère !…) la combinatoire des tests qui vont être faits…

	match()	***exec()***
avec attribut g	<img width="20" height="20" border="0" align="top" name="Bouton4" src="./images/bouton_o.gif">	<img width="20" height="20" border="0" align="top" name="Bouton5" src="./images/bouton_o.gif">
sans attribut g	<img width="20" height="20" border="0" align="top" name="Bouton6" src="./images/bouton_o.gif">	<img width="20" height="20" border="0" align="top" name="Bouton7" src="./images/bouton_o.gif">

N.B. Cette méthode met à jour les propriétés statiques de **RegExp**.

VIII-C-2 - La méthode replace(<expr. régul.>,<remplacement>)

Cette méthode référence une chaîne de caractères dans laquelle une sous-chaîne appariée avec l'expression régulière donnée dans le premier paramètre va être remplacée par le second paramètre. Généralement, le

remplacement s'opère par une chaîne de caractères. Mais il peut advenir que ce soit la sous-chaîne extraite de l'appariement que l'on veuille modifier tout en gardant tout ou partie des éléments qui la composent. Dans ces conditions, il est souhaitable de pouvoir accéder à ces parties sous la forme paramétrée afin de les répercuter dans le paramètre de remplacement sous la forme et dans l'ordre désirés. Cela est possible si l'on prévoit dans l'expression régulière des groupements parenthésés que l'on peut ensuite référencer dans le paramètre remplacement par les caractères **$1, $2…,$9**. Plusieurs caractères faisant référence à diverses portions de la chaîne sont ainsi disponibles :

Caractère	Signification
$1, $2…, $9	sous-chaînes appariées aux 9 premiers groupements
$&	sous-chaîne appariée à l'expression régulière
$'	sous-chaîne à droite de l'appariement (idem rightContext)
$`	sous-chaîne à gauche de l'appariement (idem leftContext)
$+	dernier groupement apparié


Les dates sont codées différemment dans les pays anglo-saxons et en France. Sur la base de deux chiffres pour le jour (jj), le mois(mm) et l'année (aa), une date sera codée en France jjmmaa, alors qu'en Angleterre, par exemple, elle sera codée mmjjaa. Quant au séparateur, qu'il soit « , », « / » ou « - », il demeurera identique. En supposant que le texte sur lequel porte l'opération soit référencé par la variable Txt, le problème sera résolu par :

```
Txt.replace(/\s*(\d{1,2})([ \/-])(\d{1,2})\2(\d{1,2})\s*/,"$3$2$1$2$4")
```

<img width="29" height="29" border="0" align="absmiddle" name="Bouton10" src="./images/bouton_o.gif">


*Ces caractères n'ont de validité qu'à l'intérieur de l'appel à la méthode **replace()**. Si vous essayez de les utiliser à l'extérieur, vous aurez de grosses surprises !… À moins que vous preniez en compte qu'ils sont aussi des propriétés statiques de la classe **RegExp** et que vous les utilisiez sous la forme qui convient.*

Par ailleurs, de façon logique, lorsqu'aucun appariement n'est possible, la chaîne référencée reste inchangée.

VIII-C-3 - La méthode search(<expr. régul.>)

<script language="JavaScript"> function Repl(){ var Str=prompt('Date'); return Str.replace(/\s*(\d{1,2})([\/-])(\d{1,2})\2(\d{1,2})\s*/,'$3$2$1$2$4'); //alert(Str.match(/\s*(\d{1,2})([\/-])(\d{1,2})\2(\d{1,2})\s*/)); //return false } </script> <script language="JavaScript"> var Texte = "Il est certain que JavaScript est un langage simple ˆ apprendre. Il n'en est pas pour autant rudimentaire..."; var Tartuffe="Ho c\u00e0 ! n'ai-je pas lieu de me plaindre de vous ? Et, pour n'en point mentir, n'\u00eates vous pas m\u00e9chante De vous plaire \u00e0 me dire une chose affligeante ?"; function NbOccur_I(){ var Compt = 0; var Deb = Texte.indexOf('i'); while (Deb != -1){ Compt++; Deb = Texte.indexOf('i',++Deb); } alert('Nombre d\'occurrences de "i" trouvees avec indexOf() : ' + Compt); } function NbOccur_S(){ var Compt = 0; var Deb = Texte.search(/i/i); if(!RegExp.rightContext){ alert('Fonctionnement impossible sous votre navigateur, car les diverses propriétés de classe ne sont pas implantées.'); return; } while (Deb != -1) { Compt++; Deb = RegExp.rightContext.search(/i/i); } alert('Nombre d\'occurrences de "i" trouvees avec search() : ' + Compt); } </script>

Nous avons vu, dans le chapitre précédent, la méthode **indexOf()** qui permettait de déterminer l'emplacement d'une sous-chaîne dans une chaîne donnée en référence. En particulier, nous avons mis en évidence ses faiblesses en

l'utilisant dans un exemple où l'on cherchait à compter le nombre d'occurrences d'un caractère, car pour prendre en compte la casse de celui-ci, il nous aurait fallu procéder à deux appels, l'un pour rechercher les occurrences minuscules et l'autre pour les majuscules. La méthode présentée ici bénéficie des possibilités offertes par les expressions régulières pour **rechercher une sous-chaîne vérifiant un modèle particulier**. Reprenons l'exemple auquel il était fait référence à l'instant :

Le texte…

« Il est certain que JavaScript est un langage simple à apprendre. Il n'en est pas pour autant rudimentaire… »

Le programme *(version search)…*

```
var Compt = 0;
var Deb = Texte.search(/i/i);
while (Deb != -1) {
  Compt++;
  Deb = RegExp.rightContext.search(/i/i);
}
alert("Nbre d'occurrences de 'i' : " + Compt);
```

Les résultats…	 <img width="20" height="20" border="0" name="Bouton8" src="./images/bouton_o.gif"> indexOf()	 <img width="20" height="20" border="0" name="Bouton9" src="./images/bouton_o.gif"> search()

On constate qu'avec indexOf("i"), la valeur retournée correspond au nombre de caractères très précisément identiques à celui donné en paramètre. Tandis qu'avec search() on peut préciser, grâce à l'attribut i que la casse est indifférente. On voit aussi que malgré le fait que search() ne dispose pas de paramètre indiquant l'indice de début de recherche, comme c'était le cas pour indexOf(), grâce à la propriété statique rightContext, on peut balayer la totalité de la chaîne fournie. Ces propriétés de classe n'existant pas sous Internet Explorer, cet exemple ne fonctionnera pas de façon satisfaisante sous ce navigateur.

N.B. Cette méthode met à jour les propriétés statiques de **RegExp**.

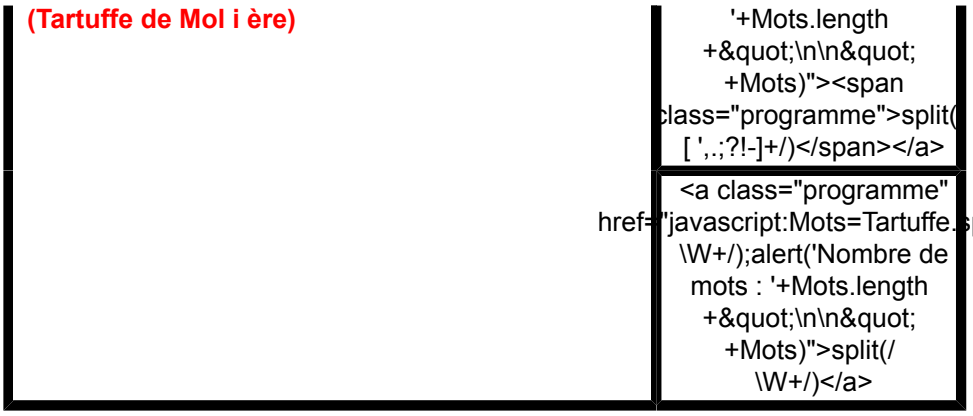
Pour terminer signalons une autre méthode de la classe **String** pouvant utiliser une expression régulière. Nous avons vu dans le chapitre précédent la méthode split(), qui, sur la base d'un séparateur fourni en paramètre, transformait la chaîne référencée en un tableau dont les éléments étaient la succession des sous-chaînes ainsi séparées.

Cette méthode, en l'état, atteint vite ses limites. Si, par exemple, nous voulons obtenir un tableau constitué des différents mots d'un texte sur la base d'un seul séparateur, on va se heurter à plusieurs problèmes. Tous les caractères de ponctuations vont soit faire partie de mots (c'est le cas de la virgule ou du point, toujours accolés au mot qui les précède), soit constituer des mots eux-mêmes (c'est le cas de ; ? ! qui réclament une espace de part et d'autre). Cela sans compter les espaces doublées ou oubliées après la virgule ou le point, les apostrophes, etc. Bref, les mots dégagés et donc le nombre de mots trouvés seraient entachés d'erreur.

En utilisant l'expression régulière **/[',.;?!-]+/** en tant que séparateur, voire, beaucoup plus simplement **/\W+/**, le problème sera résolu !….

Considérons,par exemple, le texte suivant :

Ho cà ! n'ai-je pas lieu de me plaindre de vous ? Et, pour n'en point mentir, n'êtes-vous pas méchante De vous plaire à me dire une chose affligeante ?	<a href="javascript:Mots=Tartuffe.split(/[',.:;?!-]+/);alert('Nombre de mots :



Pour les deux expressions régulières, le résultat est le même. Par contre, selon que vous serez sous Netscape ou Explorer, le résultat sera 33 pour l'un, ce qui est faux, ou 32 pour l'autre, ce qui est exact. Effectivement, regardez bien l'énumération des mots répertoriés. Celle-ci se termine par une virgule sous Netscape, preuve qu'il y a un mot vide comptabilisé, ce qui n'est pas le cas avec Explorer.

N.B. Vous remarquerez aussi, sous Explorer, que dans la deuxième méthode, les caractères accentués ne sont pas reconnus comme appartenant à \w. Ils sont donc inclus dans \W et donc interprétés comme séparateurs. Si bien qu'un mot contenant n caractères accentués sera décompté pour n+1 mots. En particulier dans cet exemple, vous pouvez constater que "méchante" compte pour deux mots à cause du « é », alors que « à » (de « …plaire à me dire… »), disparaît. Par ailleurs dans le texte analysé, « çà » a en fait été orthographié « cà », ce qui explique que le « c » subsiste et qu'un mot soit ainsi détecté. Si l'orthographe avait été correcte, il manquerait un mot au décompte final… Conclusion ?

 Testez vos acquis concernant les expressions régulières !!!!

IX - Les liens

<script language="JavaScript"> var Bouton_OFF; var Bouton_ON; var NS; onload=Declare(); </script> <script language="JavaScript"> NS = navigator.appName.indexOf('Explo')==-1; var NumEssai=0; function choix(){ var Num=prompt("Quel numÈro de journal choisissez-vous ?\n(1, 2 ou 3)",""); switch(Num){ case '1' : return "http://www.laprovence-presse.fr/"; case '2' : return "http://www.liberation.fr/"; case '3' : return "http://www.lemonde.fr/"; default : if (NumEssai++) {alert('Tant pis pour vous ! Vous ne lirez pas le journal !'); return "#";} } } </script>

Sans eux, la **toile** (Internet) n'existerait pas. En effet, les milliards de référencements de pages permettant de naviguer, de surfer, de butiner au travers de l'immense quantité d'informations diverses et variées, accessibles par chacun pour le meilleur et pour le pire, ne seraient pas possibles. À une échelle plus modeste, au niveau d'un site, le renvoi à une explication donnée dans une autre page est une technique couramment utilisée permettant de mieux structurer un texte tout en offrant au lecteur un moyen simple de revenir sur des connaissances annexes. C'est dire l'importance de ces objets. Bien sûr leur utilisation peut être opérée en HTML pur, mais ce chapitre va nous montrer comment JavaScript complète et accroît les diverses manières de créer de tels objets.

IX-A - Création et définition des liens

En JavaScript, tous les **liens** apparaissant dans une page sont introduits par la balise HTML <A>…</A>. Pour chaque occurrence de ces balises, un objet est créé et tous les liens apparaissant dans une page sont contenus dans un objet **links** qui est un tableau propriété de l'objet **document**. On notera le fait que l'on a utilisé le terme **introduits** et non pas **définis**. En effet, si un lien peut être directement défini en HTML, il peut aussi n'être qu'introduit puis défini sous JavaScript.

Un lien, qu'il soit totalement défini ou pas, comporte deux parties : la partie opérationnelle qui établit le lien physique recherché et la partie interface (texte, image, zone sensible d'image…) sur laquelle l'utilisateur doit agir pour que la partie opérationnelle soit activée.

Résumons ces deux aspects dans l'exemple suivant. Voici deux liens, le premier matérialisé par un texte, et le second par une image. En outre, le premier est totalement défini, alors que le second va dépendre d'un choix que vous allez devoir faire. Celui-ci ne peut donc pas être figé et JavaScript va nous permettre de répondre à votre choix.

Site du Département d'Informatique de Marseille-Luminy	En cliquant sur l'image ci-dessous, un dialogue vous permettra de choisir entre les trois sites suivants : 1 Le journal « La Provence » ; 2 Le journal « Libération » ; 3 Le journal « Le Monde ». <img width="64" height="64" border="0" src="./images/News.gif">

Regardez la barre d'état en bas de la fenêtre de votre navigateur. Lorsque vous passez le pointeur de souris au-dessus du lien de gauche, l'URL à atteindre s'affiche. Le lien est prédéfini ! À présent, survolez le bouton de choix ; l'URL qui s'affiche est celle de la page courante. En effet, dans le cas où la référence ne peut être résolue à l'appel, autrement dit, si le programme chargé d'affecter la référence renvoie la valeur undefined, la page active le reste. Essayez par exemple de donner un numéro non proposé et regardez ce qui se passe. Un message d'erreur vous signale que la référence n'a pu être résolue, ce qui dénote un site mal géré.

Afin d'éviter le message d'erreur, il vaut mieux prévoir l'éventualité et renvoyer la chaîne "#" qui correspond à l'URL courante. Essayez à nouveau ! C'est mieux, non ?

Voici comment ont été réalisés ces deux types de liens :

- le premier, celui de gauche, est obtenu directement via une balise HTML et a la forme suivante :
```
<a href="http://www.dil.univ-mrs.fr/"><font face="sans-serif" size="2">Site du Département
 d'Informatique de Marseille-Luminy</font></a>
```
- le second, celui de droite, dispose dans sa partie interface d'une image et passe par une fonction pour sa partie fonctionnelle :

```
<a href="" OnClick="this.href=choix()"><img src="./images/
Choisir.jpg" width="72" height="30" border="0></a>
```

La fonction choix est de la forme :

```
function choix(){
  var Num=prompt(<Message>);  //Choix de l'utilisateur
  switch(Num){                //Aiguillage selon ce choix
    case 1 : return <URL 1>;
    case 2 : return <URL 2>;
    ...
    case <i> : return <URL i>;
    ...
    default : return "#";      //Renvoi de # si réponse non prévue
  }
}
```

On remarque que l'appel de la fonction ne peut en aucune façon s'opérer tel quel dans href. En effet, celui-ci réclame un argument dont la syntaxe est bien définie et à laquelle ne répond pas l'appel de la fonction. On est donc contraint d'opérer de deux façons possibles :

1 Soit préciser en argument de **href** que l'on va exécuter une séquence JavaScript simplement en faisant précéder celle-ci de **javascript** :. Ici la séquence se réduirait donc à l'appel de la fonction **choix()** et on aurait alors :

```
<a href="javascript:choix()"><img src="./images/
Choisir.jpg" width="72" height="30" border="0></a>
```

2 Soit, comme on l'a proposé plus haut, passer par la prise en compte d'un événement « **click** » pour pouvoir affecter une URL au **href** via cette fonction. Nous reviendrons plus loin sur les événements associés aux liens.

IX-B - Les propriétés des liens

Nous avons vu que tous les liens hypertextes contenus dans une page étaient rassemblés dans un tableau **links**, propriété de l'objet **document**. Chacun des éléments de ce tableau (i.e. chaque lien hypertexte) est un objet comportant un certain nombre de propriétés que nous allons passer en revue.

Tout d'abord, définissons la syntaxe d'une URL. Celle-ci doit être de la forme :

```
<protocole>//<nom de domaine>[[:<port>]<chemin d'accès>[?<recherche>][#<ancre>]]
```

- parmi les protocoles les plus courants : **http:, ftp:, mailto:, file:/, news:, gopher:, about:, javascript:, mocha:**, où les quatre derniers ont une utilisation spécifique ;
- on appelle <domaine> le couple <nom de domaine>:<port> lorsque ce dernier existe. En son absence, **<domaine>=<nom de domaine>**. Nous ne nous appesantirons pas sur le champ :**<port>**, d'une utilisation très marginale et qu'il convient de ne pas mettre entre toutes les mains !;
- le **<nom de domaine>** correspond à l'adresse du serveur. Lorsque l'URL se limite à la donnée du protocole suivi du nom de domaine, un chemin d'accès implicite vers un fichier cible est prévu : le fichier **index.html**. En l'absence de ce fichier, le navigateur donne accès à toute la hiérarchie du domaine et en particulier à des fichiers qui théoriquement ne devraient pas être accessibles au client. Ceci est d'ailleurs vrai dans toute la hiérarchie si l'URL n'a pas pour cible un fichier, mais le nom d'un répertoire. C'est la raison pour laquelle, par mesure de sécurité, il convient de prévoir dans chaque répertoire un fichier **index.html**. Essayez, par exemple d'accéder à l'adresse **http://www.dil.univ-mrs.fr/~guizol/Le_Site_JS/Chapitre_IX/** ou tout autre répertoire de mon site… Si je n'ai rien oublié, vous allez toujours trouver un fichier qui vous empêche de musarder chez moi ! Non, mais ! ;
- le champ **?<recherche>** se rencontre lorsque vous avez lancé une recherche sur un site. Essayez par exemple de lancer une requête sur un moteur de recherche ;
- enfin, le champ **#<ancre>** permet d'accéder à l'intérieur d'un fichier cible à un emplacement particulier qui a été prévu par le créateur en positionnant une étiquette.

Par exemple, l'URL : **http://www.dil.univ-mrs.fr/~guizol/Le_Site_JS/Chapitre_VI/Chapitre_6.html#SLICE** comporte le protocole (*http:*), le nom de domaine (*www.dil.univ-mrs.fr*), le chemin d'accès (*~guizol/Le_Site_JS/Chapitre_VI/Chapitre_6.html*) et une ancre dans ce fichier (*#SLICE*) qui positionne automatiquement le document sur le paragraphe concerné.

IX-B-1 - La propriété hash

C'est une propriété accessible en lecture/écriture spécifiant, si elle existe, le nom de **l'ancre** faisant partie de la cible de l'URL. La chaîne contient le caractère #.

<img width="29" height="29" border="0" align="bottom" name="Bouton0" src="./images/bouton_o.gif">

IX-B-2 - La propriété host

Accessible elle aussi en lecture/écriture, cette propriété permet de spécifier l'ensemble **nom de domaine + port**.

<img width="29" height="29" border="0" align="bottom" name="Bouton1" src="./images/bouton_o.gif">

IX-B-3 - La propriété hostname

Alors que la précédente concernait l'association nom de domaine/port, celle-ci, accessible en lecture/écriture, ne s'intéresse qu'au **nom de domaine**.

<img width="29" height="29" border="0" align="bottom" name="Bouton2" src="./images/bouton_o.gif">

IX-B-4 - La propriété href

La totalité de l'URL est spécifiée par cette propriété, elle encore accessible en lecture/écriture.

<img width="29" height="29" border="0" align="bottom" name="Bouton3" src="./images/bouton_o.gif">

IX-B-5 - La propriété pathname

Le pathname concerne la **cible à atteindre sur le serveur** désigné par le nom de domaine. On a vu que cette cible pouvait être soit un fichier, soit un répertoire et dans ce cas le chemin d'accès était complété par index.html. Si ce fichier existe, c'est lui qui est chargé, sinon, le navigateur affiche le contenu du répertoire et à partir de lui, l'arborescence du site. Cette propriété est à nouveau accessible en lecture/écriture.

<img width="29" height="29" border="0" align="bottom" name="Bouton4" src="./images/bouton_o.gif">

IX-B-6 - La propriété port

De la même façon que la propriété hostname s'intéresse à la partie nom de domaine de la propriété host, la propriété port spécifie le **port de communication de l'URL**.

<img width="29" height="29" border="0" align="bottom" name="Bouton5" src="./images/bouton_o.gif">

IX-B-7 - La propriété protocol

Cette propriété de lien accessible en lecture/écriture permet de **spécifier le type de protocole** utilisé pour accéder à la cible désignée par le lien.

<img width="29" height="29" border="0" align="bottom" name="Bouton6" src="./images/bouton_o.gif">

IX-B-8 - La propriété search

Cette portion de l'URL permet de faire transiter **une liste de couples nom/valeur introduite par le signe ? résultant d'une requête**. Cette liste sera interprétée selon le même mode que les cookies que nous verrons plus loin.

Par exemple, pour **http://www.google.fr/search?q=javascript&hl=fr&btnG=Recherche** on procédera tout d'abord à l'extraction de la zone *search*, puis des différents *couples* en utilisant la méthode **split** paramétrée par le séparateur des couples (ici, '&') et enfin, pour chacun d'eux, le *nom* et la *valeur* en appliquant à nouveau la méthode **split** paramétrée par le séparateur '=';
Cette propriété est encore accessible en lecture/écriture.

<img width="29" height="29" border="0" align="bottom" name="Bouton7" src="./images/bouton_o.gif">

IX-B-9 - La propriété target

Cette propriété permet de **spécifier la fenêtre** dans laquelle sera chargée la page désignée par le lien. La valeur qui lui est affectée peut être une valeur prédéfinie : "**_blank**" qui opère le chargement dans une nouvelle page, "**_self**", valeur par défaut qui conduit à charger la nouvelle page dans la fenêtre contenant le lien, "**_parent**" qui charge la page référencée dans le cadre (*frame*) immédiatement supérieur à celui contenant le lien supprimant ainsi tous les cadres de même niveau que celui de départ, et enfin "**_top**" qui place la page chargée au plus au niveau, occupant ainsi toute la fenêtre du navigateur et supprimant du même coup tous les cadres qu'elle contenait.

<img width="29" height="29" border="0" align="bottom" name="Bouton8" src="./images/bouton_o.gif">
Par ailleurs, target peut être affecté avec un nom correspondant à celui qui a été donné à une fenêtre préalablement ouverte par la méthode **window.open(***<url>***,***<NOM DE FENÊTRE>***,***<caractéristiques>***)** ou celui d'un cadre défini dans la page courante. Si aucune fenêtre (ou cadre) désignée par le nom précisé n'existe, le navigateur se comporte comme si target avait été affecté à "**_blank**".

IX-B-10 - La propriété text

<script> function Texte(){ with (document){ if(NS) alert('Pour le lien '+links[0]+',\non a sous Netscape :\ntext = '+links[0].text); else alert('Pour le lien '+links[0]+',\non a sous Explorer :\ninnerText = '+links[0].innerText+'\net\ninnerHTML = '+links[0].innerHTML); } } </script>
Cette propriété permet de spécifier le **texte utilisé pour la partie interface du lien**, le texte contenu entre les balises <A> et </A> qui sera visible sur la page. *Cette propriété est seulement accessible sous Netscape*. Toutefois, du fait que les objets liens héritent des propriétés d'éléments HTML, Internet Explorer permet non seulement d'obtenir le texte en utilisant la propriété **innerText**, mais aussi l'environnement HTML propre à ce texte grâce à la propriété **innerHTML**.

<img width="29" height="29" border="0" align="bottom" name="Bouton9" src="./images/bouton_o.gif">

IX-B-11 - Les propriétés x et y

Enfin, encore une particularité de Netscape d'une utilité marginale (hormis le fait que l'on puisse effectuer un autoscroll vers le lien), ces deux propriétés **situent l'apparition du lien dans la page** en spécifiant les coordonnées horizontale (x) et verticale (y).

<img width="29" height="29" border="0" align="bottom" name="Bouton10" src="./images/bouton_o.gif">

IX-C - La propriété location de l'objet window

Nous venons de voir les propriétés d'un objet de type lien, élément du tableau **links**, lui-même propriété de l'objet **document**. L'objet **window** dispose d'une propriété, **location**, elle-même comportant la plupart des propriétés que nous venons de voir : **hash, host, hostname, href, pathname, port, protocol, et search**. Pourquoi avoir créé deux entités aussi ressemblantes ? En fait, elles ne s'intéressent pas aux mêmes objets ! Alors que la première, nous venons de le voir, s'applique à TOUS les liens hypertextes apparaissant dans une page, la seconde, elle, **représente l'adresse de la page courante** affichée dans la fenêtre du navigateur.

Cette propriété peut être modifiée. Ainsi, si l'on affecte à **location** ou **location.href** une nouvelle adresse, la page correspondante va être chargée dans la fenêtre courante et donc se substituer à la page actuellement affichée (c'est une autre façon d'opérer un lien par programme). On peut aussi ne modifier qu'une propriété de **location**. Par exemple, pour se déplacer dans une page sans la recharger, on va seulement affecter location.hash.

Essai location.hash="hash"

Alors qu'Internet Explorer effectivement ne fait que se déplacer dans la page, Netscape, pour ses versions 2 et 4.5 (au moins) recharge la page !

Alors qu'en lecture, la propriété hash associe le caractère « # » à l'étiquette choisie, comme cela apparaît dans l'exemple précédent, il ne faut pas mentionner ce caractère en écriture !

De la même façon, en modifiant la propriété search, vous pouvez forcer le navigateur à recharger l'URL avec une nouvelle requête, ce qui donnera vraisemblablement une apparence différente du document.

location dispose en outre de deux méthodes que nous allons présenter : **reload()** et **replace()**.

IX-C-1 - La méthode reload()([<obligatoire>])

Cette méthode comporte un argument facultatif booléen. En l'absence d'argument ou si celui-ci a la valeur **false**, cette méthode agit exactement comme si l'utilisateur cliquait sur le bouton « Actualiser » du navigateur. Si la page a été modifiée sur le serveur depuis le précédent chargement, elle est de nouveau chargée depuis ce dernier, sinon, elle est rechargée depuis le cache.

Si l'argument a la valeur **true**, la page est systématiquement rechargée depuis le serveur.

IX-C-2 - La méthode replace()(<URL>)

Cette méthode permet de remplacer l'URL affichée dans la fenêtre du navigateur par l'URL fournie en paramètre. Si la fonction se limite à cela, quelle différence avec la solution d'affecter **location** avec la nouvelle URL ?

En fait, alors que dans la seconde solution, l'URL précédente demeure dans l'historique de navigation, pouvant ainsi être de nouveau atteinte en cliquant sur le bouton « *Précédente* » du navigateur, avec la méthode **replace()**, la substitution est totale ; non seulement dans la fenêtre, mais aussi dans l'historique ! Ainsi l'URL précédente n'est plus accessible par le bouton « *Précédente* ».

IX-D - Les événements associés

Les événements que nous allons passer en revue ne sont pas spécifiques des liens, mais a contrario, ceux-ci peuvent faire usage de ceux-là de diverses manières et en particulier pour animer des textes ou des boutons commandant l'activation de liens ou tout simplement, comme nous l'avons vu au début de ce chapitre dissimuler derrière l'interface d'un lien un aiguillage vers une URL finale choisie par l'utilisateur. Ce ne sont là que quelques exemples qui n'ont pas la prétention de couvrir l'ensemble des possibilités et nous laisserons libre cours à l'imagination du lecteur pour trouver des utilisations judicieuses de ces outils.

De façon générale, tous ces événements apparaîtront dans la définition du lien (entre les balises <A> et </A>) sous la forme :*<événement>*="*<action>*" où l'action correspond en fait à des instructions JavaScript.

Ajoutons enfin que **les trois premiers que nous allons présenter sont accessibles en tant que propriétés d'un objet de type lien.**

IX-D-1 - L'événement-propriété OnClick

Dès l'apparition de l'événement **OnClick**, l'action JavaScript qui lui est liée est exécutée, avant même que la valeur associée à **href** soit considérée. Ce n'est qu'après avoir exécuté l'action liée à l'événement que le navigateur va charger prendre en compte **href**. Si celui-ci spécifie une URL, la page correspondante va être chargée dans la fenêtre ; si par contre sa valeur est null (si par exemple on a **<a href="" … >**) une erreur va être signalée selon laquelle l'URL ne peut être trouvée. Afin d'éviter ce problème, ou plus généralement, si l'exécution de l'action fait que l'on veut renoncer à considérer la valeur de **href** (même si celle-ci est définie), il suffit que l'action associée à **OnClick** retourne la valeur booléenne **false**.

Voici un exemple utilisant deux liens dont la définition est totalement identique hormis le fait que seule l'action du second renvoie la valeur false :

Définition du lien	Le lien
Onclick sans false	Onclick sans false
 Onclick avec false	Onclick avec false

IX-D-2 - L'événement-propriété OnMouseOver

<script language="JavaScript"> with(document) if(!NS) write(' Vous survolez ce lien '); function Montrer1(){ if(NS) document.layers['Over1'].visibility='show'; else document.all['Over1'].style.visibility='visible'; } </script> <script language="JavaScript"> with(document) if(!NS) write(' Vous survolez cet autre lien '); function Montrer(x){ if(NS) document.layers['Over'+x].visibility='show'; else document.all['Over'+x].style.visibility='visible'; } </script>

Cet événement apparaît dès que le pointeur de souris survole un lien. Vous en avez sûrement déjà vu les effets à maintes reprises. En effet, l'action par défaut prévu dans tous les navigateurs consiste à afficher dans la barre d'état, au bas de la fenêtre, l'URL correspondant au lien survolé. Par exemple, passez la souris au-dessus d'un quelconque des deux liens ci-dessus. Vous voyez apparaître "http://www.univ-mrs.fr" correspondant à la valeur de **href**.
À présent, survolez le lien ci-contre : **essai1 de OnMouseOver**.

Dans tous les cas, vous voyez apparaître un calque. Cette apparition est le résultat d'une des actions associées à l'événement en question. De plus, si votre navigateur est Explorer vous voyez dans la barre d'état non plus l'URL du lien, ce qui est le fonctionnement normal par défaut du navigateur, mais un message indiquant, par exemple, la fonction associée à l'événement (mais vous auriez pu tout aussi bien afficher l'âge du capitaine…). Dans Netscape, par contre, il n'y a aucun changement dans la barre d'état.

Recommençons l'expérience avec ce second lien : **essai de OnMouseOver**.

Le fonctionnement est identique au précédent pour ce qui concerne Explorer. Par contre, sous Netscape, nous voyons cette fois apparaître dans la barre de menu, non pas l'URL, mais le message choisi. La seule différence qu'il y a entre le premier et le second exemple est que nous avons retourné de l'action associée à l'événement **OnMouseOver**, un booléen de valeur **true**. Cela impose au navigateur de ne pas effectuer l'action par défaut qui consiste à afficher l'URL.

Pourquoi doit-on renvoyer **false** *dans* **OnClick** *et* **true** *dans* **OnMouseOver** *pour forcer le navigateur à faire seulement ce que nous voulons ? Ne cherchez pas à comprendre pourquoi… C'est un héritage du passé !*

IX-D-3 - L'événement-propriété OnMouseOut

<script language="JavaScript"> with(document) if(!NS) write(' Vous survolez encore un lien </ div>'); function Cacher(){ if(NS) document.layers['Over3'].visibility='hide'; else document.all['Over3'].style.visibility='hidden'; } </script> <script language="JavaScript"> with(document) if(NS) { write('<layer id="Over1" left="'+(links[20].x+100)+'" top="'+links[20].y +'" width="178" height="15" z-index="1" bgcolor="#FFFFCC" visibility="hide"> Vous survolez ce lien </layer>'); write('<layer id="Over2" left="'+(links[21].x +100)+'" top="'+links[21].y+'" width="178" height="15" z-index="1" bgcolor="#FFFFCC" visibility="hide"> Vous survolez cet autre lien </layer>'); write('<layer id="Over3" left="'+(links[22].x+100)+'" top="'+links[22].y+'" width="178" height="15" z-index="1" bgcolor="#FFFFCC" visibility="hide"> Vous survolez encore un lien </ layer>'); } </script>

Les exemples précédents ont montré, au travers des actions prévues, que l'on pouvait faire apparaître des objets. On a vu que Netscape efface le contenu de la barre d'état dès lors que le pointeur de souris ne survole plus le lien, ce qui n'est pas le cas pour Explorer. Pour cela, Netscape capte, en fait, l'événement OnMouseOut. Il suffit que nous fassions de même pour rendre invisibles les calques ou effacer la barre d'état sous Explorer.

Faisons l'essai : **essai de OnMouseOut**.

Effectivement, les textes et objets apparaissant lorsque le pointeur survole le lien disparaissent dès que ce n'est plus le cas.

Vous survolez ce lien.
Vous survolez cet autre lien.
Vous survolez encore un lien.

IX-D-4 - L'événement OnMouseDown

Cet événement apparaît à l'instant précis où le bouton de souris est appuyé et disparaît dès qu'il est relâché. Cela autorise une intervention supplémentaire pendant l'action du clic. En fait celui-ci est décomposé en deux phases, bouton appuyé/bouton relâché.

Pour appliquer cette nouvelle fonctionnalité, considérons un « bouton » ayant trois apparences : un état de repos, un état pointé et un état appuyé ces trois états étant respectivement représentés par les images suivantes :

Repos	Pointé	Appuyé
		

Ces trois images seront utilisées de la façon suivante :

- la première sera celle contenue dans le lien en tant qu'interface par défaut ;
- la seconde se substituera à la première sur événement **OnMouseOver**, la précédente reprenant sa place sur événement **OnMouseOut** ;
- enfin, la troisième apparaîtra lorsque le bouton de souris sera appuyé, sur événement **OnMouseDown**.

Et voilà le travail... ce **lien** vous conduira vers une utilisation de cette technique dans le chapitre VII.

IX-D-5 - L'événement OnMouseUp

Ce dernier événement permet de prendre en compte le fait que le bouton de souris est relâché. Dans l'exemple précédent, sur événement **OnMouseUp**, nous avons rétabli l'image correspondant à l'état de repos. Si tel n'avait pas été le cas, l'image correspondant au bouton appuyé aurait été maintenue. Dans la réalisation du chapitre VII à laquelle on accède par le lien ci-dessus, vous pouvez voir que l'événement **OnMouseOver** n'a pas été géré (rien ne se passe lorsqu'on survole le bouton). Par contre, on peut voir que l'événement **OnMouseDown** provoque une modification de l'allure du bouton et l'événement **OnMouseUp** permet de faire apparaître le schéma explicatif animé.

Avant de terminer, signalons la propriété referrer de l'objet document qui fait référence à l'URL du document (s'il existe) à partir duquel le document présent a été atteint. Par exemple, vous êtes parvenu à cette page à partir de…

```
<a href="javascript:alert(document.referrer ? document.referrer : 'Historique vide')">cette URL.</a>
```

Bien entendu, cette propriété n'est accessible qu'en lecture… ***«On ne peut refaire l'histoire»***.

1 : Cette méthode apparue sous Netscape 4 n'est supportée par Internet Explorer que pour ses versions les plus récentes (>5.0x). À l'inverse des méthodes précédentes, les résultats affichés ne sont pas issus de véritables calculs, mais sont des chaînes de caractères. Il faut bien que les « Exploreurs » voient ce qu'ils ratent !