Table des matières

00 : JavaScript

1. historique.

1.1. buts d'un historique.

L'objectif du dossier JavaScript est de comprendre le fonctionnement du langage.

Si nous y plaçons une rubrique «historique», ce n'est pas pour faire œuvre de chroniqueur, mais pour comprendre comment ce langage est né, quels sont les choix informatiques qui ont été faits lors de sa conception, puis au cours de son évolution.

Ceci devrait remettre à leur place les concepts sur lesquels il repose et les représentations qui en ont été données.

1.2. le HTML avant JavaScript.

1990-1993 : Le **World Wide Web** se met en place à partir de trois inventions : le HTML, le HTTP (*Hypertext Transfer Protocol)* qui est un protocole de transfert de texte, et le système des adresses web (*URL*). Le HTML (*Hypertext Markup Language)* est un langage de balisage d'hypertexte. Créé par **Tim Berners-Lee** du CERN et conçu pour représenter les pages web, il devient le principal format d'accès aux données du réseau. Deux apports font décoller le HTML initial : celui de l'intégration des images (formats GIF et XBM) et celui de l'intégration des formulaires permettant d'utiliser le web pour faire du commerce électronique. Bientôt suivi par celui des tableaux à fins de mise en page ! Voir pour cela le site suivant, qui pérennise les premières pages de l'internet !

`http://www.w3.org/History/19921103-hypertext/hypertext/WWW/TheProject.html`

Il y manque un **logiciel navigateur** pour populariser tout cela : **NCSA Mosaic** est un navigateur web multi plate-forme qui est développé à partir de fin 1992 au centre de recherches américain NCSA (*National Center for Supercomputing Applications*).

1994-1996 : L'essentiel de l'équipe à l'origine de Mosaic quitte le NCSA dès 1994 pour rejoindre Netscape Communications Corporation et développer **Netscape Navigator**.

Il y a alors trois problèmes posés par la rédaction des pages HTML :

- le premier est que le HTML apparaît de fait comme un bricolage de balises (*tag soap*), sans véritable structure ; le premier travail est donc de marquer la structuration sémantique du texte (titres, paragraphes, liens, image...) . Suant à la structure de la page, elle est décrite selon un modèle théorique, le DOM (*Document Object Model*). Le W3C naît en 1995 et commence le travail de structuration, correspondant aux 4 premières versions publiées du HTML.

- le second est que le HTML, suivant en cela une recommandation éditée pour son ancêtre le SGML (repris par son descendant, le XML), sépare le style (modalités d'affichage) du contenu sémantique. Apparaît l'idée d'utiliser un langage de mise en page, les feuilles de style CSS, qui sera mis en œuvre de façon très progressive.

- le troisième est que l'inter-activité des pages web est très restreinte : on sait changer de page dans le navigateur (liens), envoyer un formulaire (form/action), et c'est tout. Pas de modification de la page une fois affichée, pas de contrôle des formulaires avant l'envoi.

1.3. HTML interactif.

Pour répondre au dernier problème, Netscape engage une équipe pour doter son navigateur d'un langage de script ; ce langage est proposé en 1995 par Brendan Eich sous le nom de **LiveScript**.
En décembre 1995, Sun et Netscape annoncent la sortie de «**JavaScript**», nom qui est sensé faire profiter le nouveau langage de l'engouement du moment pour le nouveau langage multi plate-forme Java, avec lequel **il n'a par ailleurs pas grand chose de commun**.

En mars 1996, Netscape met en œuvre le moteur JavaScript dans son navigateur Web Netscape Navigator 2.0. Le succès de ce navigateur contribue à l'adoption rapide de JavaScript dans le développement web orienté client. Microsoft réagit alors en développant JScript qu'il inclut ensuite dans Internet Explorer 3.0 en août 1996 pour la sortie de son navigateur. JScript est très semblable à JavaScript en ce qui concerne les spécifications du langage, mais ses primitives web sont largement incompatibles .

Netscape soumet alors JavaScript à Ecma International pour standardisation. Les travaux se terminent en juin 1997 par l'adoption du nouveau standard ECMAScript décrit dans le document *Standard ECMA-262*.

2. Les choix du langage.

2.1. Les préalables.

Nous sommes en 1995. Depuis les années 1970, une forte recherche s'est développée dans le domaine des langages, faisant émerger progressivement des paradigmes qui sont alors assez stabilisés :

- langages compilés (C, Pascal) / interprétés (Smalltalk) / p-codés (Java, Pascal)
- langages procéduraux (C) / langages fonctionnels (LISP) / langages objet (Smalltalk)
- instructions expressions (C) / instructions action (Pascal)
- langages textuels (Lisp)
- identification fortement typée (Ada) / faiblement typée (Lisp)
- changement de type / transtypage ou *cast.*

2.2. Un langage interprété.

Le choix pour JavaScript est celui d'un langage **interprété**, bien adapté au travail multi plate-forme. L'inconvénient rédhibitoire traîné par les langages interprétés est leur lenteur et leur manque d'optimisation. JavaScript pâtit encore de sa mauvaise réputation en la matière. Des performances sans cesse améliorées des composants matériels ainsi que l'affinement des algorithmes de la machine virtuelle font que cet inconvénient tend à disparaître : JavaScript s'est invité dans la programmation des jeux interactifs.

2.3. Un langage objet.

Le paradigme objet a été retenu pour ce langage. La programmation objet s'est imposée dans l'ingénierie logicielle à cause surtout du mode de structuration modulaire qu'il permet pour les données et les algorithmes. Ce type de programmation facilite également les extensions (appelées librairies, bibliothèques, modules) mais aussi la **maintenance et l'adaptation** des composants.

Mais en 1995, les machines réelles pâtissent de leur manque de puissance : aussi, contrairement aux choix qui seront faits quelques années plus tard pour Python, le programme n'est pas intégralement objet. Comme pour Java, les concepteurs ont maintenu une part non objet pour les données de base : les nombres, les textes, les valeurs de vérité (vrai/faux). Ce sont les données primaires, qui ne relèvent pas de la programmation objet et dont le traitement direct peut faire gagner du temps d'exécution.

En JavaScript, tout ce qui n'est pas primaire est objet. Une fonction ? Une procédure ? **`null`** ? ce sont des objets et ils relèvent du traitement général de tous les objets.

2.4. Un langage sans classes.

Les tenants du paradigme objets estiment à juste titre que l'une des caractéristiques qui donne de la force à ce mode de structuration est **l'héritage**. Mais ils se partagent en deux groupes sur la façon de la gérer :

- soit on définit des "super objets", des **classes**, envisagées comme des «moules» pour engendrer les objets intéressants que l'on appelle "**instances**" ; on confère à ces classes à la fois structures de données (propriétés / champs / attributs) et algorithmes (les méthodes). Ce qui dans les langages sophistiqués n'empêche pas de considérer aussi les classes comme des instances de super classes... Les classes sont hiérarchisées, et chacune participe aux instances finales en leur apportant leur part de structures de données et de méthodes.

- soit on définit un objet comme le clones d'un autre objet, avec la possibilité pour le clone de s'enrichir de données et de méthodes propres, qu'il peut transmettre ensuite à ses propres clones. On appelle cette méthode le **prototypage**, reprenant une expression de l'industrie où un objet est produit à partir d'un **prototype**, peut évoluer, et à son tour devient un prototype. C'est ce parti qu'ont pris les concepteurs de JavaScript, prolongeant une idée développée

dans les mêmes milieux de la recherche à cette époque : le projet **Self**, dont l'intérêt est resté tout théorique se rattache à cette mouvance (on trouve encore la documentation sur le site de Oracle).

2.5. Un langage faiblement typé.

Les langages "durs" comme Ada, C, Pascal, Java sont marqués par **l'obsession du contrôle** : on définit des noms bien structurés, les identificateurs, pour désigner les données et les algorithmes. Puis on déclare qu'à ces noms peuvent correspondre des données et des algorithmes parfaitement calibrés : **structures de données prédéfinies** (**`integer, float, single, double, small`** etc rien que pour les nombres entiers), **signatures** des fonctions et procédures...

On ne peut plus en changer dans le cours des programmes. Les objets ont également une structure bien définie et ne peuvent être enrichis qu'au prix d'une lourde extension du programme, pas d'une **manière dynamique** (au cours de l'exécution du programme). **Ce typage fort** répond à des impératifs de rigueur dans le développement et la maintenance des programmes. Il a son prix : lourdeur de la programmation, vérifications systématiques qui dépensent un temps de calcul important, tant au développement qu'à l'exécution.

On ne peut éviter **les questions de typage** : si on fait une division, les opérandes attendus sont des nombres et pas des textes, encore moins des valeurs de vérité ! Mais, si on veut travailler légèrement, on n'est plus obligé d'utiliser des identificateurs respectant des règles restrictives de formation ; un nom (donnée textuelle avec au moins un caractère) suffit. Ce procédé est utilisé par Lisp par exemple (et un de ses avatar, *logo* qui était entré dans l'éducation dans les années 1980).

De plus, on peut abandonner la multiplicité des structures de données prédéfinies, mais surtout la liaison rigide entre un identifiant et son type de données, déclarée une fois pour toutes. C'est le principe du typage dynamique.

S'il y a risque de problème, on peut faire **un contrôle a priori** ; mais il vaut mieux avoir à s'excuser une fois que de fureter sans cesse pour voir si tout se passe bien. Il faut donc un système de gestion d'erreur qui surveille les cas difficiles : c'est le principe des exceptions.

2.6. La force de l'habitude.

Les concepteurs de JavaScript avaient fait des choix adaptés, que l'on retrouve au cœur du langage. **Tout, ou presque, est objet**. Il y a **quatre types essentiels** et quatre seulement : nombres, textes, valeurs de vérité, objets. Il n'y a plus de fonctions qui seraient spécifiquement des fonctions : ce sont des objets qui relèvent du traitement général des objets. On peut également supprimer la notion d'identificateur (avec ses restrictions : caractères alphabétiques, chiffres, undescore / pas d'espace, pas de chiffre en tête) : **le nommage** est suffisant.

Une syntaxe «à la Lisp» est possible : des déclarations réduites au minimum, pas de mots réservés, tout texte est un code exécutable... C'est assez déroutant. Alors, les concepteurs ont concocté une syntaxe un peu particulière : à première vue, elle ressemble à celle des langages procéduraux de l'époque, le C en étant l'archétype. Il y a toujours des identificateurs, mais on peut souvent s'en passer ; et pour une même déclaration, il existe **des syntaxes concurrentes mais sémantiquement équivalentes** (c'est le parti pris opposé qui a été fait pour Python : il n'y a une façon de bien coder et une seule... enfin presque) **.**

Ainsi, il y a trois syntaxes distinctes pour déclarer une fonction/procédure/méthode. Cela complique singulièrement le jeu sous prétexte de ne pas dérouter les premiers programmeurs. Un des gros problèmes de l'enseignement de JavaScript est que les apprenants plaquent, **sur les productions textuelles, des concepts issus d'autres langages et qui ne sont pas les bons. La similitude syntaxique devient un obstacle à la compréhension.**

Il y a deux syntaxes pour **qualifier une propriété** ; l'une qui ne fonctionne qu'avec des identificateurs, **la notation qualifiée**, et l'autre plus générale, **la notation associative**, avec la possibilité de mélanger les deux : **`monObjet.maPropriete`** ou **`monObjet["maPropriété"]`**. La notation associative autorise par exemple : **`window["003 : mon objet"][" ma propriété en +"].`**

2.7. La bataille du point-virgule.

Une opposition conceptuelle entre les tenants du Pascal et les tenants du C s'est manifestée dans les années 1970. Les pascaliens concevaient l'instruction élémentaire comme une modification de l'environnement : une affectation comme **`a := 3 + 5`** change la variable **`a`** mais **`3 * a – 5`** qui

est bien une expression ne peut être une instruction. D'un point de vue conceptuel, c'est une bonne démarche que de séparer instruction (modification de données, en principe) et expression (valeur issue d'un calcul ou d'une déclaration). Ce n'est pas le point de vue du langage C, pour lequel toute expression est une instruction ! Dans le premier cas, la syntaxe permet d'isoler une instruction correctement écrite ; il convient pour rédiger une séquence de **séparer** les instructions (le point-virgule pascalien).

En C, il faut **terminer** l'instruction : une expression qui ne modifie pas l'environnement et qui est terminée est perdue (la terminaison d'instruction est aussi le point-virgule). C'est ce dernier point de vue qui est conservé par JavaScript, et malheureusement, cela a une conséquence pernicieuse sur l'écriture des programme ; on verra ce point dans le développement ultérieur (boucle **`while,`** déclaration de **`fonction`**).

2.8. Un langage versatile.

L'objectif premier de JavaScript ne doit pas être perdu de vue : **un langage à tout faire**, sans contrainte particulière, **aussi simple que possible**. JavaScript est un langage versatile. On peut créer des modules qui simulent la programmation des langages avec classes. L'exercice peut être amusant ; et les programmeurs de JavaScript en ont fait un véritable sport. Mais ceci est risqué, dans la mesure où on glisse rapidement dans des modes d'expositions ou de programmation qui ne sont pas ceux de la logique JavaScript et compliquent inutilement la pensée. Il est fréquent aussi que par exemple, la littérature parle de classe au lieu de constructeurs, sous prétexte que la syntaxe des constructeur en JavaScript ressemble à celle des classes dans les autres langages (**`new monConstructeur()`**). Mais la logique n'est pas la même.

2.9. Le transtypage.

Dans les langages typés, il est fréquent qu'un résultat d'un certain type puisse être interprété, dans un contexte défini, comme une donnée d'un autre type :

- un résultat nul (nombre 0) se voit interprété comme le booléen **`faux`** (et non nul est **`vrai`**) dans une condition : le langage C fonctionne ainsi (il est vrai qu'il n'a pas de type booléen).

- dans une commande d'affichage, qui normalement demande des argument textuels, un nombre se voit transformé en texte : **`print ("résultat : ", 255)`**. La majorité des langages fonctionnent sur ce mode (C, Pascal, Python...).

Pour les tenants des langages fortement typés, il convient de toujours utiliser des fonctions de conversion de type **`nombre → booléen`** ; **`nombre → texte`** etc. Les langages qui veulent rester lisibles et ne pas encombrer leurs sources de fonctions qui "*vont de soi*" ont développé un procédé : le **transtypage** (*cast*) automatique.

JavaScript qui vise toujours à "faire court" utilise systématiquement ce procédé. C'est ainsi que dans un contexte qui «attend» du texte, tous les arguments qui n'en sont pas se voient appliquer une fonction de transformation en texte. Ou encore un contexte où on attend une valeur de vérité (vraie/faux ou **`true/false`**) implique l'application d'une fonction adaptée au type de l'argument pour le transformer en booléen. JavaScript va beaucoup plus loin que les autres langages en la matière. L'exemple suivant est spécifique : un identificateur sans valeur affectée (non initialisé) se voit transcodé en valeur booléenne **`false`** dans un contexte où un booléen est attendu . On est évidemment très loin des usages classiques !

3. Intégration.

3.1. Pas de librairies ni de modules : tout est objet et propriété d'objet.

Les langages comme Java ou Python disposent d'un interpréteur, qui peut être lancé à partir d'une console. Ils comportent un certain nombre de librairies natives (bibliothèques/modules), intégrées, et peuvent importer d'autres librairies ; toutes les données nouvelles peuvent être utilisées partout dans le programme (elles sont "globales"). Le programme "principal" et ses annexes peuvent les utiliser moyennant quelques précautions d'usage comme par exemple l'indication du module d'importation. Rien de tel en JavaScript : le système JavaScript est vu comme un arbre où, tout objet (sauf la racine et les feuilles) est **à la fois objet propriétaire et propriété d'objet**.

note : ceci serait vrai sans une catégorie spéciale d'objets, **les objets d'appel** qui ne sont pas des

propriétés. La racine fait partie de ces objets spéciaux.

3.2. Un noyau "embarqué".

La racine est appelée aussi **l'objet global** ; elle caractérise l'environnement de programmation et n'appartient pas au sens strict au noyau de JavaScript. Par exemple, dans l'environnement web client qui est à l'origine de la création du langage, l'objet global s'appelle **`window`**.

L'objet global doit obéir à un certain nombre de spécifications pour être celui d'une implémentation JavaScript (plus exactement on devrait dire **ECMAScript**) : il doit posséder un certain nombre de propriétés qui sont au cœur du langage : **`Math, Object, String, Number, Booelan, Regex, Function, parseInt, parseFloat`** etc.

3.3. Spécificités du JavaScript web client.

JavaScript fait partie du navigateur, et il est en interaction avec le moteur du HTML : pour réaliser cela, il implémente un certain nombre de propriétés qui enrichissent l'objet **`window`** et que ne possède pas un environnement comme **ActionScript**, implémenté par Adobe dans Flash. Parmi ces propriétés, citons la propriété **`document`** qui donne une représentation du **`DOM`**, utilisable par le langage, et en interaction avec le moteur du HTML. Ou encore la propriété **`location`**, qui concerne l'url de la page actuelle.

3.4. framework.

Tel quel, JavaScript n'est qu'une coquille vide : pour l'utiliser, il faut lui ajouter des scripts. Un script est un bloc de code qui enrichit l'objet global, ou est simplement exécuté dans la foulée du lancement de la machine JavaScript. Dans une page web, cela se produit à chaque fois qu'une balise **`<script>`** est rencontrée. Un framework est simplement un script qui comporte des objets qui enrichissent **`window`** et un morceau de code qui réalise l'opération (on dit souvent : qui initialise le framework). Ainsi le framework **jQuery** ajoute une propriété à **`window`**, la propriété nommée **`jQuery`** qui bien entendu est un objet...

3.4. question de nommage.

Rien n'oblige un **objet/propriété** JavaScript a être nommé. Mais non nommé, il est simplement impossible de le manier : on ne crée pas explicitement d'objets non nommés. Sauf dans le cas de ce que l'on appelle les fonctions anonymes globales, qui ne sont que du code exécutable un peu sophistiqué.

JavaScript crée cependant en interne des objets non nommés, les "**objets d'appel**". Si on utilise une propriété sans spécifier l'objet dont il est propriété, c'est que cet objet est ce que nous appellerons **un objet d'appel** ; pour l'instant il suffit de savoir que **`window`** est l'objet d'appel global (c'est le seul objet d'appel nommé !) et que chaque corps de fonction appelée (donc le code est interprété) possède aussi un objet d'appel qui lui est propre et renferme paramètres et variables locales. Une variable globale est donc une propriété de **`window`**.

Les trois expressions suivantes en dehors d'un corps de fonction sont équivalentes dans un script :

```
var maVariable = 3.1416 / maVariable = 3.1416 / window.maVariable = 3.1416
```

4. un langage qui s'apprend.

Dire qu'un langage est simple et léger ne signifie pas qu'il soit aisé à apprendre et à utiliser : il possède seulement peu de paradigmes. Mais ceux-ci peuvent se révéler très puissants et contenir des chausse-trappe. Surtout, et c'est le cas ici, si d'une part sa syntaxe apparaît voisine de ce dont on a l'habitude, ce qui entraîne **des comportements inappropriés** si on ne sait pas précisément ce que l'on fait. En plus, le fonctionnement interne est suffisamment différent des autres langages pour que l'intuition ne suffise pas : c'est par exemple le cas pour la portée des variables qui répond à un mécanisme qui est propre au langage.

01 : JavaScript : le dispositif

1. Les logiciels.

1.1. Langage de script pour navigateur.

JavaScript et sa variante JScript de Microsoft sont des langages créés pour à écrire des scripts destinés à l'intégration dans des pages web. Il existe des environnements de développement spécialisés qui permettent un travail professionnel tant sur JavaScript que son intégration dans des pages web. Ces derniers ne sont pas adapté à notre objectif :

- nous voulons décrire **une approche rigoureuse du Javascript** ;
- nous en tenir à des outils disponibles pour tous, ayant leur équivalent dans le monde du libre ;
- proposer des activités multi plates-formes.
- *la question de la création de sites web professionnels est exclue.*

1.2. Recommandations.

En matière de développement : les éditeurs classiques légers qui disposent d'un environnement de développement (IDE), avec la coloration syntaxique et l'appel d'exécution sont recommandés. Exemple : Geany, PSPad.

Nous travaillerons par défaut en Unicode UTF-8. Outre que JavaScript suppose un code écrit en Unicode, l'utilisation d'autres systèmes d'encodage (ISO-latin1 par exemple) est sans intérêt et peut même s'avérer générateur d'erreur.

En matière de navigateur : Nous supposons que l'on dispose d'un navigateur récent et fonctionnant autant que possible selon les recommandation du W3C. Ce qui restreint à l'utilisation de Firefox/IceWeasel (version >3), de Chrome et Safary. L'utilisation de IE (version 9) est possible, comme celle de Epiphany ou Opera. Dans ces derniers cas, les comportements ne sont pas toujours garantis, essentiellement à cause de défaillances du CSS.

En matière de débogage : il existe un plug-in de Firefox, appelé Firebug qui fonctionne aussi bien sous Windows que Linux avec Firefox. Ce débogueur n'est pas indispensable, mais ce véritable couteau suisse du web peut s'avérer intéressant. Notons que les navigateurs assez récents disposent d'une console web, qui autorise au moins la détection d'erreurs et le contrôle syntaxique des scripts.

En matière d'outils complémentaires : Pour des raisons qui seront explicitées en leur temps, l'usage efficace de la programmation JavaScript dans les pages web passe par l'utilisation de framework (librairies JavaScript). Nous nous en tiendrons à un seul : jQuery, qui est léger, disponible et gratuit. Son télé-chargement et son utilisations feront l'objet de plusieurs études dans l'exposé (deuxième partie : **JavaScript pour le web)**.

2. Processus du travail.

2.1. Insertion dans une page web.

Étant donné la destination des scripts (illustrer le fonctionnement de JavaScript), la démarche suivant est proposée :

1- les scripts sont écrits dans une page web au format donné dans le paragraphe suivant.

2- la page web est sauvegardée.

3- la page web est exécutée dans le navigateur, soit par un appel depuis l'IDE, soit par exécution comme fichier (Fichier > Ouvrir un fichier …). On rappelle que lorsque l'on exécute plusieurs fois de suite un fichier, le rechargement du fichier ne réinitialise pas le cache d'exécution : on peut donc avoir des surprises liées au fait que les initialisations de variables (et d'images) ne sont pas faites.

La réinitialisation se fait simplement par F5 ; la remise à dimension par Ctrl-0.

2.2. Le cadre recommandé pour la page web.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

<head>
  <title>le titre</title>
  <meta http-equiv="content-type" content="text/html;charset=utf-8" />

<!-- *** feuilles de style *** -->
  <style type="text/css">
    #page {
      width : 1024px ;
      margin-left : auto ;
      margin-top : auto ;
    }
  ...
  </style>

<!-- *** code javascript *** -->
  <script type="text/javascript">
  ...
  </script>
</head>
<body>
  <div id = "page">
  ...
  </div>
</body>
</html>
```

L'en-tête est le standard actuel (en 2012) du XHTML (programmation HTML 4). Le conteneur ** ** d'identificateur **"page"** (ou un identificateur similaire) est mis systématiquement, avec sa feuille de style pour assurer une présentation pratique à l'écran, par exemple le centrage de l'affichage.

2.3. Où peut-on placer la balise <script> ?

On peut mettre la balise script à tout endroit où une balise non obligatoire est acceptée : dans **<head>** comme dans **<body>.** On peut mettre autant de balise **<script>** qu'on le veut.

Il faut prendre garde cependant que la balise **<script>** s'inscrit dans le flot HTML. Elle est exécutée au cours du chargement de la page, au moment où elle est lue dans la page HTML. La balise **<script>** ne peut donc pas prendre en compte des données qui sont définies après elle. Cela est important pour les instructions de JavaScript qui travaillent sur le HTML, ce qui représente la majorité des cas dans JavaScript côté client !

Dans le cas où il y a plusieurs balises **<script>**, les textes en JavaScript se comportent comme s'ils étaient placés les uns à la suite des autres, **en séquence**. Lors du chargement de la page, on ne peut utiliser que des choses qui ont déjà été définies. Mais **une fois la page chargée**, tout les éléments définis sont utilisables.

Il faut cependant bien faire attention à **ne pas redéfinir un élément** : la surcharge fait partie des spécification du langage, et l'erreur commune est de réaliser une surcharge sans le vouloir. Dans ce cas c'est la dernière définition et elle seule qui est prise en considération. **Le cas d'école** est celui des fonctions à exécuter à la fin du chargement de la page : dans la programmation proposée par les navigateurs du début des années 2000, il n'y avait qu'un seul identificateur de fonction disponible pour

une telle exécution, la fonction **`onload()`**. Si on redéfinissait cette fonction dans un script de la page web, toute utilisation dans un framework déjà chargé se voyait oblitérée sans que l'on en sache rien. D'autres procédés d'initialisation sont aujourd'hui proposés, et on les verra en leur temps.

2.4. Un schéma de site sur le disque.

Pour éviter les problèmes d'accès aux ressources, nous proposons par défaut, la structure arborescente suivante ; elle servira surtout dans la seconde partie, **JavaScript pour le web**.

racine	sous répertoires	
`texteshtml`		les fichiers HTML sont placés dans ce répertoire
	`css`	les fichiers de feuilles de style (.css) sont placés dans ce répertoire
	`img`	des sous répertoires plus ou moins spécifiques aux cas traités sont créés ici pour les fichiers images (.png, .jpg, .gif, .svg)
	`js`	les fichiers de script JavaScript (.js) sont placés dans ce répertoire ;
	`jquery`	le framework jQuery (des fichiers .js) sera placé ici pour des raisons de clarté ;
	`arc`	un répertoire fourre-tout pour des données archivées par exemple.

3. `alert()`.

3.1. L'instruction alert().

Tous les navigateurs disposent d'une instruction JavaScript **`alert()`**. Cette fonction prend **obligatoirement un paramètre** et lors de son exécution provoque l'apparition d'une popup modale (Il faut cliquer le OK ou faire un **`<Entrée>`** pour la fermer). Cette instruction sert beaucoup lors de la mise au point des scripts ou leur contrôle.

Comme le script est interprété, il n'est analysé que lors du chargement de la page web. Cette analyse se fait dans l'ordre des instructions, et s'arrête lors d'une erreur. L'analyse de la page se continue en «sortant» de la balise script (**`</script>`**). Si la popup ne se déclenche pas, c'est que l'analyseur n'est pas parvenu jusque la balise.

Le paramètre est en principe une chaîne de caractère mais les transcodages automatiques rendent son usage très souple.

commande	affichage
`alert("ceci est un message") ;`	ceci est un message
`alert("");`	
`alert(35 + 89);`	114
`alert(5 < 7);`	true
`alert("addition de 5 et 7: " + (5 + 7));`	addition de 5 et 7: 12
`alert("addition de 5 et 7: " + 5 + 7);`	addition de 5 et 7: 57
`alert() ;`	erreur fréquente : il faut obligatoirement un argument valide

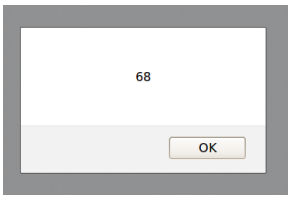
3.2. exemple.

```
1.script type="text/javascript">
2.     var a ; /* la variable a n'est pas initialisée */
3.             /* la variable b est  numérique, et c est un texte */
4.     var b = 68, c = "90" ;
5.     alert (b) ;
6.     alert (a) ;
7.     alert (b+c) ;
8.     alert (b*c) ;
9.</script>
10.<!-- *** h100_alert.html *** -->
```

Ce script s'effectue «dans la foulée», avant même que la page soit chargée. Les variables JavaScript sont déclarées à l'aide du mot clef **`var`** ; elles sont ou non initialisées. La virgule sépare les éléments d'une liste (initialisée ou pas). Il y a donc une rafale de quatre popup qui s'affichent l'une après l'autre.

Pour le travail sous Firefox (recommandé), on conseille d'utiliser le plug-in Firebug : dans ce cas, au lieu de **`alert()`**, on peut utiliser la commande **`console.log()`**, et même redéfinir **`alert`** en **`console.log`**. L'affichage se fait dans la console de Firebug. Les autres navigateurs connaissent aussi **`console.log`**, mais n'acceptent pas la surcharge.

3.3. Résultats.

copies d'écran	commentaire
	Le résultat est attendu : la valeur de la variable est affichée
	Il n'y a pas d'erreur due à la non initialisation de la variable a. Lorsque l'on évoque la valeur d'une variable non initialisée, une donnée primaire traduite par **`undefined`** est affichée. La seconde popup affichée n'est pas semblable à la première : une case à cocher est ajoutée ; ce point sera expliqué ci-dessous.
	Le résultat est un peu déroutant. En fait, le signe + est le signe de la concaténation (jonction bout à bout) des chaînes. Si l'un des arguments est une chaîne, tous les arguments sont transformés en chaîne avant concaténation.
	Ici également, le résultat est déroutant ; mais la logique précédente s'applique à la multiplication : JavaScript essaie de transformer tous les arguments en nombre et effectue la multiplication.

3.4. Empêcher cette page d'ouvrir des dialogues supplémentaires.

Cette indication apparue avec Firefox 3 est à première vue incongrue. Elle participe pourtant à la sécurité du navigateur : en effet, si une instruction **`alert()`** est placée dans une boucle, il n'y a aucun moyen de sortir de la boucle. Dans ce cas, il est intéressant de fournir le moyen de ne pas afficher les

popup tout en exécutant le reste de la boucle ; **on voit l'intérêt du procédé lors du débogage**. Évidemment, en cas de boucle infinie, volontaire ou non, l'intérêt est plus restreint : il faut mettre fin à l'exécution de la page (cliquer la fermeture par l'onglet correspondant).

4. prompt() et confirm().

4.1. Autres boîtes de dialogue.

La boîte de dialogue **`alert()`** n'est pas la seule à être disponible et utilisables pour la mise au point des scripts. Il existe deux autres boîtes de dialogues qui, à la différence de **`alert(),`** retournent une valeur exploitable, : la boîte **`prompt()`** et la boîte **`confirm()`**.

4.2. `prompt().`

La syntaxe est la suivante :

`prompt ( `*`chaîne d'information, chaîne valeur par défaut`*` )`

Retourne : ***`une chaîne de caractères ou null`***

La valeur par défaut est facultative. Le prompt est utilisé pour la saisie de données par le client. Ajoutons que les boîtes de dialogues n'apparaissent pas dans les pages web actuelles : elles subsistent comme outils simples de débogage et de mise au point.

Cette boîte de dialogue est dotée de deux boutons **`Annuler (Cancel)`** et **`OK`**. Il est impossible de changer le libellé des boutons. **`OK`** valide l'entrée ; **`Annuler`** renvoie **`null`**.

4.3. `confirm().`

La syntaxe est la suivante :

`confirm ( `*`chaîne d'information`*` )`

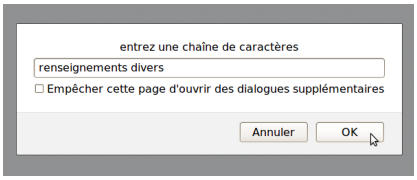
Retourne : ***`un booléen`***

Cette boîte de dialogue est dotée de deux boutons **`Annuler`** et **`OK`**. Il est impossible de changer le libellé des boutons. Il faut donc bien adapter le libellé de la chaîne d'information.

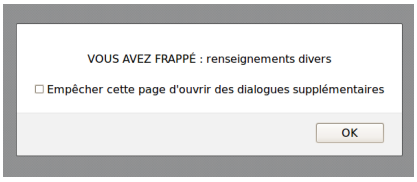
Exemple :

```
1.<script type="text/javascript">
2.    do {
3.         var valeur = prompt ('entrez une chaîne de caractères') ;
4.         alert ("VOUS AVEZ FRAPPÉ : "+valeur);
5.    }
6.    while (confirm ("pour continuer cliquez OK"));
7.</script>
8.<!-- h0101_confirm.html →
```

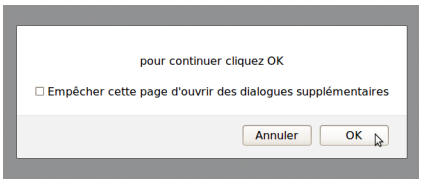
`prompt()`



`alert()`



`confirm()`


pour continuer cliquez OK
Empêcher cette page d'ouvrir des dialogues supplémentaires
Annuler
OK

02 : les données primaires

1. Types de données.

1.1. Une classification stricte : les types.

Dans tous les langages de programmation, on distingue **le code exécutable** et **les données** qui sont les éléments créés ou modifiés par le code exécutable. À la base, les données sont des emplacements de la mémoire : la façon dont ces éléments modifiables sont stockés en mémoire ou sont traités par le code exécutable est diversifiée. On classe les données selon ce que l'on appelle des **types** : tous les items qui relèvent d'un même **type** sont stockés selon un même processus et peuvent être soumis aux mêmes genres de transformations (opérateurs ou fonctions).

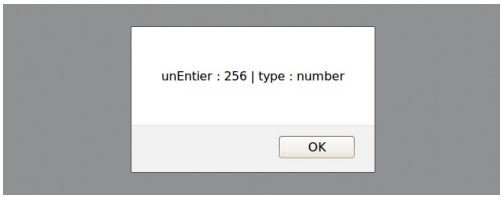
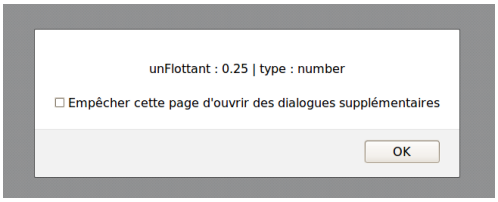
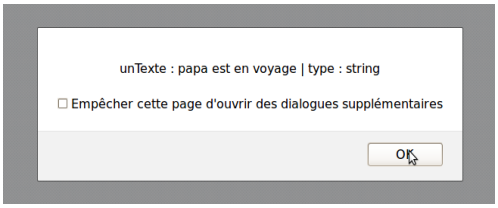
Une donnée appartient à un type et un seul. Les types son nommés. Tous les langages disposent de types de données «naturels» : certaines données sont numériques, d'autres textuelle, d'autres sont des valeurs de vérité (vrai/faut) etc.

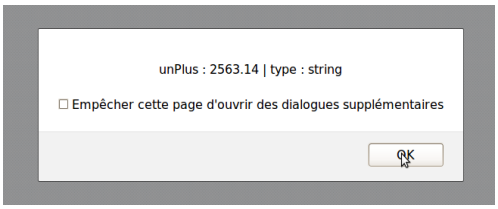
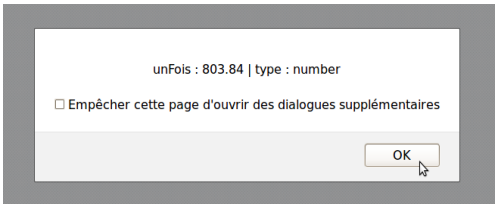
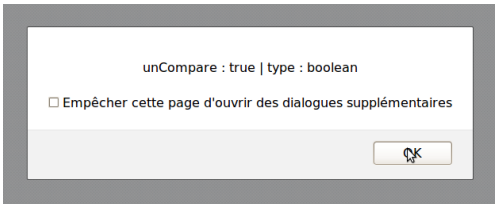
1.2. accès au type de donnée : `typeof`.

L'opérateur **`typeof`** permet d'accéder au nom du type de la donnée étudiée.

exemple :

```
1.<script type="text/javascript">
2.    /* des variables initialisées */
3.    var unEntier = 256, unFlottant = 0.25 ;
4.    var unTexte = "papa est en voyage", unPlus = 256 + "3.14";
5.    var unFois = 256 * "3.14", unCompare = unPlus < unFois ;
6.    /* les tests */
7.    alert ("unEntier : "   + unEntier   + " | type : " + typeof unEntier);
8.    alert ("unFlottant : " + unFlottant + " | type : " + typeof unFlottant);
9.    alert ("unTexte : "    + unTexte    + " | type : " + typeof unTexte);
10.   alert ("unPlus : "     + unPlus     + " | type : " + typeof unPlus);
11.   alert ("unFois : "     + unFois     + " | type : " + typeof unFois);
12.   alert ("unCompare : "  + unCompare  + " | type : " + typeof unCompare);
13.</script>
14.<!-- h200_typeof.html *** -->
```

	Le type est **`number`** et non **`integer`** (entier) , comme dans beaucoup de langages (C, Java, Python). Attention : en JavaScript, la casse est signifiante : **`number`** est en minuscules.
	Le type est également **`number`** et non **`float.`**
	Ici le type est **`string.`** En français, nous dirons chaînes de caractères ou chaîne -tout court-.

	Le signe plus est ici le signe de concaténation, d'où le résultat.
	Le signe * implique une multiplication.
	Ici, l'opérateur de comparaison implique un changement de type des arguments. Mais en priorité en chaîne : d'où le résultat assez paradoxal (c'est l'ordre lexicographique qui sert dans la comparaison).

2. Les types primaires.

2.1. des types de la programmation procédurale.

JavaScript définit quatre types appelés «primaires». Les items qui relèvent de ces types ont un comportement similaire à ce qu'il serait en C ou en BASIC. Une variable de l'un de ces types est une référence directe à un contenu mémoire de forme prédéfinie. Chaque type a ses opérateurs prédéfinis et dispose de quelques fonctions très classiques. Les types primaires sont similaires à ce qu'ils sont en Java : ils relèvent de la programmation procédurale, pas de la programmation objet.

2.2. les quatre types primaires :

* le type **`number`** : il n'y a qu'un seul type de nombres en JavaScript. Il n'y a pas les types entiers (**`integer, single, long, double`**) des autres langages. Un **`number`** est un nombre à virgule flottante. Un tel nombre est codé sur 64 bits. Un nombre à virgule flottante f s'écrit sous la forme suivante :

$$f = \pm m \times 2^{\pm e}$$

m est appelé la mantisse et ±e l'exposant ; ce sont deux entiers « machine ».

La valeur ± m est codée sur 54 bits et la valeur ±e sur 10 bits. Si l'exposant est nul, f est un nombre entier au sens usuel de l'appellation. On dispose ainsi d'entiers compris entre -9007199254740992 et +9007199254740992. En gros, 16 chiffres décimaux.

Quant à l'exposant, il est compris entre -1024 et +1024. Ce qui en décimal correspond à une valeur de $2^{\pm e}$ égale à peu près à $10^{\pm 308}$.

Deux valeurs spéciales de type **`number,`** ne respectent pas ce qui vient d'être dit : **`NaN`** le « non-nombre » et **`Infinity`** «nombre infini ».

* le type **`string`** : un item de type string est une séquence de caractères Unicode (en mémoire, le caractère Unicode est codé sur deux octets). Contrairement à la plupart des autres langages, JavaScript ne dispose pas de type caractères. Si on a besoin de représenter un seul caractère, il faut passer par une chaîne à un seul caractère.

Noter que quand on dit «séquence», on se réserve la possibilité de n'avoir aucun caractère : on dit que la chaîne est vide. Ceci très utile pour une fonction ou un opérateur qui réclame une chaîne et qu'aucune chaîne explicite ne convient : le cas typique est celui du transcodage que l'on a déjà rencontré lors de la concaténation. Si on écrit : **`var a = "" + 56`** , la variable **`a`** prend le type

`string` à peu de frais ! (56 est automatiquement transcodé dans le même type que **`""`**).

* le type **`boolean`** : comme tous les langages JavaScript a besoin de caractériser **`le vrai et le faux`**. C'est la base de la programmation conditionnelle (**`si <condition> alors … sinon`**) et de la boucle (**`tant que <condition> faire …`**).

- le type **`indefined`** : ce type est assez singulier, à tous les sens du terme : il n'a qu'un seul item et JavaScript est un des rares langages à l'utiliser. Cela vient du fait qu'en JavaScript, il est nécessaire que la non initialisation d'une variable ou d'un paramètre de fonction ne soit pas sanctionnée et qu'elle puise être reconnue. Il faut au contraire que l'on puisse tester si une variable ou un paramètre a été initialisé (avec les événements, l'initialisation ne dépend pas nécessairement du programmeur). Ce type spécial autorise le test.

3. Le type `number`.

3.1. littéraux décimaux.

On rappelle qu'un littéral est un texte qui sera interprété par l'analyseur lexico-syntaxique (*parser*) comme la valeur d'un élément d'un certain type. Qu'en est-il des nombres en écriture décimale ?

* écriture en nombres entiers décimaux :

0 // 1 // 12 // 1234567891236 // -1234567891236 // 1234567891236

sont des écritures valides et qui sont interprétés en valeurs exactes. Le signe **`-`** et le signe **`+`** sont des opérateurs unaires, mais il ne faut pas oublier que **`++`** et **`--`** sont aussi des opérateurs ; ainsi l'expression (**`3 ++5)`** est illicite alors que **`(3 + +5)`** est valide.

À partir de ~16 chiffres (voir ci-dessus) et jusque ~307 chiffres, l'écriture conduit à une valeur arrondie.

Il faut éviter de commencer une écriture par un 0 : ce mode est réservé aux écritures octales ou hexadécimales.

* écriture octale :

Il s'agit de l'écriture en forme d'entier interprété comme un octal. L'écriture commence par un zéro suivi de chiffres de 0 à 7. Exemple : 0123 en octal correspond au nombre 83 en décimal. L'implémentation est quelquefois incomplète.

* écriture hexadécimale :

La séquence commence par 0x et elle se poursuit par des chiffres hexadécimaux :

0xff // 0x1236 // 0x123aef // 0X1Ab5F (*respectivement à 255 ; 4662 ; 1194735 en décimal*)

Les caractères x, a, b, c, d, e, f peuvent être indifféremment en majuscules ou minuscules.

*écriture en nombre à *virgule* :

La marque décimale est le point. Faire attention de ne pas mettre de virgule, qui en JavaScript est un séparateur. Voici des écritures valides :

3.1416 // -3.14 // .33333 // 6e23 // 6.e23 // -6e23 //-6e+23 // 6.56e-24 //

La lettre e peut être majuscule ou minuscule.

Voici la formulation générale de la règle :

`[séquence de décimaux][.séquence de décimaux][(e|E)[(+|-)] séquence de décimaux]`

Si la première séquence n'est pas vide, il est admis que la seconde puisse l'être : 6. // 6.e-23

3.2. `NaN` et `Infinity`.

Il existe deux identificateurs, **`NaN`** (Not a Number) et **`Infinity`** (infini) pour deux éléments particuliers du type **`number`**. Le fait qu'ils appartiennent à ce type fait que certaines opérations qui dans d'autres langages se traduiraient par une erreur, sont licites en JavaScript :

- le dépassement de capacité : ce qui est un nombre du point de vue mathématique peut ne pas entrer dans le cadre fini des valeurs représentables ; Par exemple l'expression **`1e307 * 1e10`** multiplie deux nombres représentables ; mais le résultat ne l'est pas ! L'identificateur **`Infinity`** est alors requis. De même 1/0 retourne la même valeur. Ce n'est pas une erreur qui déclenche une exception. On a aussi **`-Infinity`** pour les infinis négatifs.

- l'écriture illicite : l'expression **`0/0`** est syntaxiquement correcte, mathématiquement inacceptable. Le

calcul retourne **NaN** alors que dans les autres langages il y aurait erreur. **La valeur NaN n'est comparable à rien** (à la différence de **Infinity**), même pas à elle même : **0/0 == NaN** n'est pas admis. Pour tester cette valeur spéciale, il existe une fonction prédéfinie : **isNaN()**.

Notons aussi la fonction **isFinite() qui** retourne la valeur vrai si son argument admet une représentation en JavaScript (ni **NaN**, ni **Infinity**, ni **-Infinity).**

3.3. Opérateurs.

Deux opérateur unaires **++** et **--** peuvent être préfixés ou postfixés ; ce sont les opérateurs d'incrémentation et de décrémentation.

Les valeurs de type **number** supportent les opérateurs binaires classiques, avec les règles usuelles de priorité : addition (**+**), soustraction (**-**), multiplication (*****), division (**/**).

L'opérateur reste (%) pose problème car le type entier n'existe pas.

Le résultat de a%b (ou a et b sont 2 nombres et donc peuvent avoir une partie décimale) est le plus petit nombre à soustraire de a pour que la division par b de la différence entre a et b donne un nombre entier, c'est à dire sans partie décimale. Dans le cas où a et b ont une expression entière, le résultat est bien celui attendu. C'est moins évident dans le cas contraire et il vaut surveiller les calculs de près.

Les opérateurs arithmétiques sont également présents :

décalage à gauche (**<<**), à droite (**>>**), décalage avec extension de zéros (**<<<** et **>>>**), le **Et**(**&**) bit à bit , **XOR** (**^**), le **Ou** (**|**), mais l'implémentation ne se fait que sur les nombres à représentation entière sur 32 bits.

Les opérateurs logiques seront vus avec le type booléen.

4. Les chaînes de caractères.

4.1. les littéraux chaînes de caractère.

- règle 1 : les littéraux chaînes s'écrivent lorsque c'est possible comme séquence de caractères affichables, encadrés de la double quote (") ou de la simple quote('). Si une chaîne est encadrée de double quotes, la simple quote est admise dans la chaîne et réciproquement. Voici des exemples :

Une chaîne vide : **""** *// une autre chaîne vide :* **''** // **"papa est en voyage d'affaires"** //

'on nomme "chaîne" une suite de caractères' //

** **

- règle 2 échappement : lorsqu'un caractère n'est pas affichable, ou pas disponible, ou interdit dans la chaîne, il faut utiliser un échappement. Le caractère d'échappement est l'anti-slash (\).

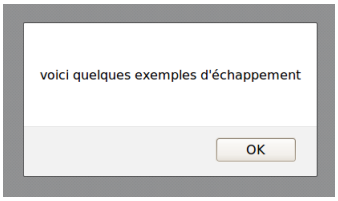
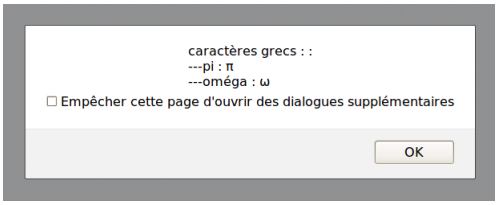
- règle 3 : la chaîne littérale est mono-ligne ; la ligne peut être très longue. Les éditeurs modernes facilitent l'édition de lignes longues (affichage multi-lignes, mais génération mono-ligne).

la séquence	équivalence	sa signification
\0	**\u0000**	le caractère NULL
\b	**\u0008**	back-space
\t	**\u0009**	tabulation horizontale
\n	**\u000A**	nouvelle ligne
\v	**\u000B**	tabulation verticale
\f	**\u000C**	nouvelle page
\r	**\u000D**	retour chariot
\"	**\u0022**	double quote
\'	**\u0027**	simple quote
****	**\u005C**	anti-slash
\xXY	**\u00XY**	caractère Iso machine (ici : latin-1)

\uXYZT		Caractère quelconque : XYZT est l'écriture hexadécimale du code Unicode du caractère. Majuscules ou minuscules admises.

exemples.

```
1.<script type="text/javascript">
2.    alert ('voici quelques exemples d\'échappement') ;
3.    alert ('caractères grecs : :\n---pi : \u03c0\n---oméga : \u03c9\n');
4.</script>
5.<!-- *** h0201_echappement.html *** -->
```

	échappement sur la quote simple
	saut de ligne par \n et caractères grecs (le code Unicode est trouvé à partir des tables de caractères, disponibles sur tous les SE). La seconde partie donne en annexe un tableau des valeurs utiles (celles que l'on trouve dans le fichiers de fonte classique, comme Arial, DéjaVu).

4.2. Concaténation de chaînes.

L'opérateur **+** permet de concaténer les chaînes. On a déjà rencontré plusieurs exemples. C'est à peu près tout ce que l'on peut faire sur les chaînes. On reviendra sur un autre type, l'objet **`String`** qui lui est beaucoup plus riche (différent du type **`string`**).

4.3. eval().

Une chaîne qui contient une expression valide ou un segment de code valide peut être passée à **`eval()`**. La fonction **`eval`** l'évalue comme si c'était du code. Si le résultat de la dernière évaluation est défini, la fonction retourne cette valeur. Sinon, elle est **`indefined`**.

`eval ("4+3")` retourne 7 ;

`eval ("alert('ceci est un message')")` évalue la fonction boîte de dialogue **`alert()`** et reste **`undefined`** car la fonction alert ne retourne rien.

`eval ("var a = 8 ; alert(a+1)")` retourne 9.

L'intérêt de cette fonction est tout théorique : elle sert par exemple lorsqu'un programme fabrique du code JavaScript et qu'il a ensuite besoin de l'évaluer.

4.4. `parseInt()` et `parseFloat()`.

Ces deux fonctions prennent une chaîne comme argument, ou aussi bien une chaîne qu'un nombre pour **`parseInt()`**, et retournent une valeur de type **`number`** : une valeur entière pour **`parseInt()`**, un flottant pour **`parseFloat()`**, et **`NaN`** en cas d'échec.

La façon de fonctionner est pratique, mais pas intuitive : en effet, le parser lit la chaîne, caractère par caractère, et s'arrête au premier caractère qui fait problème ; la fonction retourne alors le résultat obtenu en l'état, ignorant éventuellement la fin de la chaîne. Le second argument, facultatif, désigne la base où la recherche doit être faite. Si **`parseInt()`** détecte un octal ou un hexadécimal, il en retourne la valeur.

expression	valeur retournée et explication	
`parseInt("toto")`	NaN	(Not a Number)
`parseInt("044xyz")`	36	(4x8 +4)
`parseInt("0xffxyz")`	255	(15x16 +15)
`parseInt("25.15px")`	25	(valeur entière)
`parseInt("25px")`	25	
`parseFloat("25.15px")`	25.15	(valeur décimale)
`parseInt("25.15px", 16)`	37	(2x16 + 5)
`parseInt("25.15px", 6)`	17	(2 x6 + 5)

5. le type boolean.

5.1. les identificateurs réservés : `true` et `false`.

Pour les nombres et les chaînes, on dispose de codes inscrits dans notre culture de l'écrit. Il n'en est pas de même avec les valeurs de vérité (vrai/faux). Il faut donc utiliser **des identificateurs** pour caractériser les valeurs de vérité. Ce sont les mots réservés du langage **`true`** et **`false`**, en minuscules.

5.2. les opérateurs booléens.

On dispose en JavaScript des opérateurs booléens classiques :

opérateurs	commentaire
`!`	opérateur unaire de négation
`&&`	le ET booléen ; attention à ne pas confondre avec le ET arithmétique. Le second argument est évalué uniquement si le premier vaut **`true`**
`||`	le OU booléen : **`false`** si les deux arguments sont **`false.`** Le second argument est évalué uniquement si le premier vaut **`false`**.

5.3. les expressions à valeurs booléennes.

De nombreux opérateurs permettent d'obtenir des valeurs booléenne :

opérateurs	arguments	commentaires
`<`, **`<=`**, **`>`**, **`>=`**	deux nombres	inégalités entre nombres
`<`, **`<=`**, **`>`**, **`>=`**	deux chaînes	ordre lexicographique. Si l'un des arguments est un nombre, il est automatiquement transcodé en chaîne.
`==`	deux expressions quelconques	évalue les expressions et teste l'égalité de valeurs
!=	deux expressions quelconques	teste l'inégalité de résultats
`===` et **`!==`**		teste l'identité des arguments (même adresse en mémoire, ou alias).

5.4. Transcodage automatique.

JavaScript est un langage où on en écrit le moins possible. Aussi le transcodage automatique y prend-il une place très importante, au détriment quelquefois de la sécurité du code, puisque le contrôle du code est restreint. C'est là une différence fondamentale avec C++, Pascal, Java ou Python, ou encore de Ada, qu'il faut citer comme un langage où tout transcodage automatique est interdit. On dit que le typage de JavaScript est un typage faible.

données	interprétation	commentaire
`""`	**`false`**	ou toute chaîne vide
`toute chaîne non vide`	**`true`**	attention à l'erreur classique : **`if("false") ...`**
`0`	**`false`**	la valeur nulle
`tout number non nul`	**`true`**	
`NaN`	**`false`**	
`undefined`	**`false`**	absence d'initialisation
`null`	**`false`**	objet null
`tout autre objet`	**`true`**	

6. Variable non initialisée.

Le type **`undefined`** n'a qu'une seule valeur possible, de même nom, **`undefined`**. Il n'a pas d'intérêt en tant que type et on oubliera désormais que **`undefined`** appartient aux types de JavaScript.

Avec son transcodage automatique à false, cette variable est d'une grande utilité lorsque JavaScript est utilisé dans le contexte du web : son environnement de programmation dépend alors du navigateur utilisé par le client, et les fonctions définies dans cet environnement peuvent différer d'un navigateur à l'autre, voire d'une version du navigateur à l'autre.

L'exemple typique est le contrôle de l'existence d'une fonction qui est caractéristique d'une implémentation sur un navigateur afin de savoir sur quel environnement on peut tabler. Et donc si la fonction est utilisable ou non. Le framework jQuery utilise systématiquement ce type de procédé.

```
if (addEventListener) ….
else if(attachEvent) …
```

La première conditionnelle est valide sur les navigateurs Firefox pas trop anciens, Chrome... Et la seconde par Internet Explorer depuis la version 5.

Cette propriété reste difficile à manier dans des cas plus complexes.

7. null.

La norme ECMAscript donne **`null`** comme un type à un seul élément, **`null`** ! Cette données correspond à une initialisation (à la différence de **`undefined`**), avec laquelle on ne peut rien faire. C'est la valeur retournée par **`prompt()`** en cas d'annulation. Elle se teste facilement. Mais Firefox le donne comme type **`object`**. Ce qui est cohérent et en fait n'a pas une importance capitale.

03 : les types `object` et `function`

1. Rappel sur la notion d'objet.

1.1. Une donnée structurés.

Dans le vocabulaire de la programmation, **un objet est d'abord une donnée structurée**. Alors que les données primaires peuvent être vus comme des atomes, des constituants ultimes, les objets sont des collections de constituants qui peuvent être des données primaires, des objets eux-mêmes, ou du code exécutable (fonction).

Dans des langages à objets comme C++, Pascal ou Python, il existe deux sortes de constituants : **des données proprement dites et du code exécutable**, que l'on appelle en général des fonctions et plus précisément des **méthodes**. **Cette discrimination stricte n'a pas lieu d'être en JavaScript**. Un code exécutable, que l'on appelle aussi une fonction est lui aussi un objet, avec toutes les propriétés d'un objet.

1.2. Propriétés ou champs.

Chaque constituant d'un objet est appelé une propriété de l'objet. On utilise aussi le mot champ et le mot attribut pour désigner une propriété ; mais le mot attribut ayant déjà une signification en HTML, nous n'emploierons pas ce terme.

Une propriété est un couple identifiant/valeur. L'identifiant sert à désigner la propriété, afin de pouvoir l'utiliser soit comme donnée, soit comme code exécutable selon les besoins.

1.3. Identifiant et identificateur.

La plupart des langages usuels connaissent la notion d'identificateur : il s'agit d'une suite de caractères pris dans un alphabet restreint : [A-Z] [a-z] [0-9] et _ (underscore). Le premier caractère n'est jamais un chiffre. Pour certains langages (Java, Python), l'ensemble des caractères alphabétiques peut être étendu, mais ce n'est pas une bonne pratique que d'utiliser les caractères accentués ou de langues étrangères (les feuilles de style CSS ne les acceptent pas dans leurs intitulés -noms d'identificateurs ou de classes- qui sont communs au CSS et au code JavaScript ; noter cependant que le CSS admet le tiret et pas JavaScript !!!).

Voici quelques identificateurs valides :

j**`avascript`** // **`JavaScript`** // **`_bibi_`** // **`un_exemple`** // **`unExemple`**

Un identificateur ne peut comporter d'espace, de trait d'union, de point, ni aucun signe de ponctuation.

JavaScript admet cependant dans son alphabet le caractère **$** (dollar) ; mais il faut le réserver à des cas très particuliers, comme l'usage dans des frameworks (Les CSS l'ignorent). Certains identificateurs sont réservés par le système, et leur usage est prohibé :

mots réservés utilisés actuellement :

```
break case catch continue debugger default delete do else finally
for function if in instanceof new return switch this throw try
typeof var void while with
```

mots réservés pour les versions ultérieures :

```
class const enum export extends import super implements interface
let package private protected public static
```

Un identifiant est une chaîne de caractère non vide ; comme un identifiant peut contenir des espaces, des virgules et autres séparateurs utilisés en JavaScript on l'écrit nécessairement sous forme d'un littéral lorsqu'il n'est pas un identificateur. Voici quelques identifiants valides :

`"toto"` ou **`toto`** // **`"continue"`** // **`"Van der Valls"`** // **`"Mont-Saint-Éloi"`** // **`"{a, b, c}"`**

Les mots réservés ne peuvent pas être des identifiants : **`"do"`**, **`"continue"`**

Pour des raisons de clarté de l'exposé, **nous ferons la différence entre identificateurs et identifiants**. Cette différenciation n'est pas reconnue dans la littérature sur JavaScript.

Donnons tout de suite la règle d'usage qui différencie **identificateur** et **identifiant** :

- l'usage de la qualification exige des identificateurs : **`monObjet.laProp ;`**
- une variable est obligatoirement un identificateur ;
- pour utiliser une propriété on peut utiliser indifféremment identificateur et identifiant dans la notation avec crochets : **`monObjet[laProp]`** équivaut à **`monObjet["laProp"] ;`**
- un identifiant ne peut être qu'une propriété d'objet : **`montObjet['485']`** est valide ;
- pour les variables globales, il est équivalent de définir **`var id = 3.14`** ou **`window["id"] = 3.14`** ou **`window[id] = 3.14`**

2. Une définition littérale pour les objets à bloc de code exécutable.

2.1. fonction anonyme.

Un objet à bloc de code exécutable s'écrit sous forme de ce que JavaScript appelle une fonction anonyme. Cela ressemble à une fonction dans les autres langages :

1- un mot de déclaration **`function`**,

2- une suite de paramètres qui sont des identificateurs. La suite peut être vide, et donc être réduite aux parenthèses (). La virgule est le séparateur de paramètres s'il y en a plusieurs.

3- une séquence de code exécutable : cette séquence de code exécutable est une suite d'instructions avec des accolades { et } qui marquent le début et la fin de la séquence. Les instructions admettent le point-virgule comme séparateur. La partie exécutable n'est pas un bloc au sens de bloc/instruction (chapitre 6)

```
function (par1, par2, par3 . . .) { code exécutable }
```

2.2. nommer une fonction.

Un objet a pratiquement toujours besoin d'être identifié pour que l'on puisse l'utiliser ; la manière de faire ne présente pas grande originalité : la façon la plus simple est d'utiliser un identificateur (on nuancera au paragraphe 3.3. cette pratique qui recouvre la presque totalité des usages).

Exemple :

```
var maFonction = function (par1, par2, par3 . . .) { code exécutable }
```

Pour des raisons de continuité avec le langage C ou Java, on utilise un équivalent plus classique,

```
function maFonction (par1, par2, par3 . . .) { code exécutable }
```

Le texte de définition ci-dessus est identique à **tout** le texte de la définition donnée en premier lieu, la déclaration **`var`** comprise ! Seul diffèrent le mécanisme de création de la variable et son initialisation. Cependant, étant donnée le mode de gestion des variables en JavaScript, cette équivalence peut se révéler déroutante : on verra l'importance de cette remarque lors de l'étude de la portée des variables. Ceci implique également que si une instruction commence par le mot clef **`function`**, on a affaire à une déclaration de variable.

2.3. appeler une fonction.

Appeler une fonction, c'est demander à la machine virtuelle JavaScript de l'exécuter. La manière d'appeler est traditionnelle : on fait suivre le bloc de code, ou son substitut (son identificateur) de la liste (parenthèses, la virgule comme séparateur) des paramètres réels. En fait **`()`** est un opérateur d'exécution et la liste de ses paramètres ses opérandes. Dans le cas d'appel d'une fonction anonyme, un parenthésage du code est obligatoire car une instruction qui commence par **`function`** est une déclaration de variable ; si on veut faire une instruction qui est un appel de fonction anonyme, il est obligatoire d'expliciter le fait que l'on n'a pas affaire à une déclaration de fonction, mais une expression (appel de fonction) : on s'en tire en parenthésant :

```
(function (…) {….})() ;
```

ou encore :

```
(function (…) {….}()) ;
```

Attention cependant : ***il y a un piège ; si un déclaration de fonction est suivie de parenthèses, ce qui est entre parenthèse est considéré comme argument d'appel de la fonction.***

Si on a le schéma suivant :

```
var qqch = function () {…}
(function () {})() ;
```

la seconde ligne est considérée comme argument de la fonction du dessus ! L'accolade fermante n'est pas une fin d'instruction. Il faut alors soigneusement séparer les lignes ! Une règle de sécurité pourrait être : commencer un appel de fonction anonyme par le séparateur traditionnel, **le point virgule** :

```
var qqch = function () {…}
;(function () {})() ;
```

paramétrage :

Si un paramètre réel est une donnée primaire, cette donnée devient la valeur du paramètre formel, qui se comporte alors comme une variable déclarée dans le corps de la fonction, que cette donnée soit représentée par un identificateur ou un littéral ; si c'est **un objet** donné par son identificateur, c'est la référence de cette objet qui est copiée, c'est-à-dire en pratique que si une modification se produit sur une propriété de l'objet paramètre, elle se répercute sur l'objet passé en paramètre.

Faut-il respecter le nombre de paramètres lors de l'appel ? Il n'y a aucun contrôle sur la liste des paramètres : on peut donc syntaxiquement oublier des paramètres ou en mettre en plus. Ce n'est qu'à l'exécution du bloc de code que l'on peut avoir des problèmes : un paramètre réel absent revient à affecter **`undefined`** au paramètre formel. Il est courant en JavaScript d'utiliser cette disposition. Mais attention à l'erreur qui se produit si on utilise sans contrôle un paramètre formel qui peut ne pas être affecté lors de l'appel. Ainsi, la fonction **`alert()`** sans paramétrage effectif provoque une erreur car son paramètre réel (une chaîne de caractère) est explicitement utilisé.

En programmation WEB, le programmeur ignore quel environnement d'exécution sera celui de la fonction, puisque celui-ci dépend du client WEB, c'est-à-dire du navigateur utilisé. Telle fonction qui avec Firefox est appelée avec un paramètre réel peut très bien sous Internet Explorer être appelée sans paramètre. Le programmeur dans ce cas doit, dans le code de la fonction, vérifier si le paramètre formel a été initialisé ou non : le paramètre non initialisé a la valeur **`undefined`** !

```
1.<script type="text/javascript">
2.    var fonct = function(param) {
3.             if (param == undefined)
4.                 return  "paramètre absent";
5.             else return "paramètre égal à "+ param ;
6.        }
7.     alert (fonct()) ; /*affiche « paramètre absent » */
8.     alert (fonct(3.14)); /*affiche « paramètre égal à 3.14 » */
9.</script>
10.<!-- *** h0300_parametre.html *** -->
```

rappel : on peut écrire indifféremment **`if (param == undefined)`** ou **`if (param)`**. La valeur **`undefined`** est transcodée à **`false`**. C'est ce qu'on trouve en programmation courante.

3. Définition littérale d'un objet sans code exécutable.

3.1. Définition des champs avec des identificateurs.

```
1. <script type="text/javascript">
2. var monObjet = {
3.     propUn : "une chaîne " ,
4.     propDeux : 3.14 ,
5.     propTrois : function () { alert ("propriété trois")} ,
6.     propQuatre : { x : 1, y : 4.56 } ,
7.     propCinq : function () {
8.         alert( "la seconde propriété vaut " + this.propDeux)}
9.     } ;
```

```
10.     alert (monObjet.propUn) ;
11.     monObjet.propTrois() ;
12.     alert (monObjet.propQuatre.y) ;
13.     monObjet.propCinq() ;
14.</script>
15.<!-- *** h0301_objet.html *** -->
```

Chaque propriété se décline sur le schéma **identificateur/valeur** ; le séparateur est le signe : (deux-points). Les propriétés formant une suite dont le séparateur est la virgule. L'objet est défini comme un conteneur encadré par des accolades. **Les valeurs peuvent être des données de tous types**. Ici on n'a employé que des littéraux ; mais évidemment **toute variable déclarée** est admise (même non initialisée, ce qui peut faire problème car la donnée **undefined** n'est pas évidente à manier).

JSON

La définition littérale qui vient d'être évoquée est en fait une commodité syntaxique : **c'est un format léger d'échange de données** non spécifique à JavaScript, qui peut être utilisé pour les échanges de données entre langages tels que C, C++, C#, Java, JavaScript, Perl, Python. Il s'appelle JSON. Historiquement, il est bien issu des travaux sur JavaScript et son nom : **JavaScript Object Notation** (Notation Objet issue de JavaScript) le montre. Il est facile à lire ou à écrire pour des humains, bien plus que XML par exemple.

Il n'utilise que quelques structures bien partagées par les langages modernes :

* Un objet : ensemble de couples nom/valeur non ordonnés. Un objet commence par **{** (accolade gauche) et se termine par **}** (accolade droite). Chaque nom est suivi de **:** (deux-points) et les couples nom/valeur sont séparés par **,** (virgule).

* Un tableau : collection de valeurs ordonnées. Un tableau commence par **[** (crochet gauche) et se termine par **]** (crochet droit). Les valeurs sont séparées par **,** (virgule).

* Une valeur peut être soit une chaîne de caractères entre guillemets, soit un nombre, soit **true** ou **false** ou **null**, soit un objet, soit un tableau. Ces structures peuvent être imbriquées.

* Une chaîne de caractères est une suite de caractères Unicode, entre guillemets, et utilisant les échappements avec antislash. On admet qu'une chaîne soit vide. Un caractère est représenté par une chaîne d'un seul caractère.

3.2. L'identificateur qualifié.

Pour accéder à une propriété, le mode d'accès est celui de tous les langages objets : la qualification.

Un identificateur qualifié comporte l'identificateur de l'objet, suivi d'un point, suivi de l'identificateur de la propriété. Exemple : **monObjet.propUn**. Si la **valeur de la** propriété est elle-même un objet, on peut continuer le procédé **monObjet.propQuatre.y**.-Dans le cas d'une fonction que l'on veut voir exécuter, on ajoute l'opérateur d'exécution (parenthèses contenant la liste des paramètres réels).

monObjet.propCinq() : dans le code exécutable de la fonction, on a utilisé la variable **this** (mot réservé). **this** identifie l'objet qui a la fonction sollicitée comme propriété : soit ici **monObjet** . On désigne cet objet comme **objet appelant** (*caller*) ou **objet propriétaire** (*owner*). Le fonctionnement de JavaScript fait qu'en programmation web on ignore souvent lors de la programmation de la fonction quel sera en fait l'objet appelant. On reviendra ultérieurement sur cette question importante.

3.3. Définition des champs avec des identifiants.

Le cas défini précédemment n'est qu'un cas particulier d'un cas plus général : on peut en effet avoir comme propriété n'importe quelle chaîne identifiante.

L'exemple précédent peut s'écrire sans difficultés, sans avoir à modifier les appels qualifiés :

```
var monObjet = {
        "propUn" : "une chaîne " ,
        "propDeux" : 3.14 ,
```

```
            "propTrois" : function () { alert ("propriété trois")} ,
            "propQuatre" : { x : 1, y : 4.56 } ,
            "propCinq" : function () {
                alert( "la seconde propriété vaut " + this.propDeux)
        }
    } ;
```

Cela ne présente que peu d'intérêt sauf si on désire sophistiquer, en utilisant des identifiants qui ne sont pas des identificateurs :

```
1. <script type="text/javascript">
2.     var monObjet = {
3.             "propUn" : "une chaîne " ,
4.             "propDeux" : 3.14 ,
5.             "propriété fonctionnelle n°3" : function () {
6.                 alert ("propriété trois")} ,
7.             "proposition Quatre" : { x : 1, y : 4.56 } ,
8.             "propriété fonctionnelle n°5" : function () {
9.                 alert( "la seconde propriété vaut " + this.propDeux)
10.        }
11.    } ;
12.    alert (monObjet.propUn) ;
```

On ne peut alors utiliser la qualification que sur les propriétés ayant un identificateur.

3.4. Accès aux propriétés sans la qualification.

On utilise une syntaxe habituellement retenue pour les tableaux ordinaires (C, Pascal, Java) ou pour les tableaux associatifs (listes de Python, tableaux du PHP). Voici sur l'exemple précédent ce que cela donne (on n'a repris que l'appel) :

```
13.    monObjet["propriété fonctionnelle n°3"]() ;
14.    alert (monObjet["propriété Quatre"]["y"]) ;/* ou encore : */
15.    alert (monObjet["propriété Quatre"].y) ;
16.    monObjet["propriété fonctionnelle n°5"]() ;
17.</script>
18.<!-- *** h0303_objet.html *** -->
```

Il n'y a pas de problème particulier, si ce n'est un changement d'habitude du programmeur. Cette syntaxe est beaucoup moins utilisée que l'écriture qualifiée.

Tous les objets peuvent accéder ainsi à une propriété :
`{ prop1 : 3, prop1 : "toto" } ["prop1"]` est tout à fait sensé.

3.5. Questions de types.

Il existe deux genres d'objets : le genre **`function`** et le genre **`object`** qui correspondent aux deux modalités de définition qui viennent d'être décrites. L'opérateur **`typeof`** distingue les objets sans code exécutable (type **`object`**) et les fonctions (type **`function`**). Mais il faut bien se convaincre que **tout ce qui s'applique au type `object` s'applique également au type `function`**.

Quelques mises en garde doivent être formulées :

- on a déjà indiqué que **`null`** était de type **`object`** avec Firefox;

-un objet vide **`{}`** n'est pas l'objet **`null`** mais ils sont tous deux de type **`object`**. Si on crée deux objets avec **`{}`**, on obtient deux objets distincts.

4. enrichissement des objets.

4.1. le problème.

Dans les langages objets de haut niveau (C++, Pascal, Java), on ne peut en principe ni ajouter, ni retrancher une propriété à un objet. Il y va en fait de la consistance des objets qui doit être garantie

dans les projets d'envergure. Mais des langages comme Python ou JavaScript ont au contraire vocation à être très versatiles, à s'adapter aux situations locales de programmation, quitte à perdre en sécurité quant au code produit. On peut donc modifier un objet, lui ajouter une propriété ou lui en supprimer une de façon dynamique.

4.2. ajouter une propriété.

Il y a évidemment deux approches syntaxiques pour l'adjonction d'une propriété :

* méthode avec identificateur : **`objet.nouvellePropriete = valeur`**

* méthode avec identifiant quelconque : **`objet[nouvellePropriete] = valeur`**

Ce qui conduit à des instructions du genre suivant :

```
monObjet.nouveau    = "une chaîne de caractères comme valeur" ;
monObjet[nouveau]   = "une chaîne de caractères comme valeur" ;
monObjet["nouveau"] = "une chaîne de caractères comme valeur" ;
monObjet.nouveau    = null ;
monObjet["nouvelle propriété"] = {x1 : 3.14, x2 : 2.72} ;
monObjet.newFonction = function (par1, par2) { return par1*par2}
monObjet["nouvelle fonction"] = function (par1, par2) { return par1*par2}
```

Note : l'objet **`monObjet`** peut être aussi bien une fonction qu'un objets sans code exécutable !

Le signe égale = :

Le signe égal d'affectation appartient au langage C ; dans les langages procéduraux (C, Pascal) et typés, il signifie que la valeur obtenue en évaluant le membre de droite doit être stockée dans la case mémoire réservée à l'identificateur qui forme le membre de gauche. C'est un tout autre mécanisme qui est en jeu ici. Le membre de droite est bien évalué.

- Dans le cas où la valeur trouvée est **un objet** pré-existant (A = B où B est un identificateur d'objet), un lien est établi entre l'identificateur A et la valeur liée à l'identificateur B : A et B désignent désormais le même objet ! A est **un alias** de B. Il le demeure tant qu'il n'est pas de nouveau affecté. Toute modification d'une propriété de A se répercute dans B ; tout ajout ou suppression de propriété dans A se répercute dans B. Et réciproquement.

- Si le résultat n'est pas préexistant, le mécanisme d'évaluation **crée** une zone mémoire pour y mémoriser la valeur trouvée. L'affectation lie l'identificateur du membre de gauche à cette zone mémoire. En pratique, on accède à la valeur par l'identificateur affecté. Lors d'une affectation il y a réellement création d'une "variable". Si l'identificateur n'a jamais été ni déclaré ni déjà affecté, il est créé en tant qu'identificateur dans une table d'identifiant avant que la liaison avec la valeur soit effectuée.

4.3. retirer une propriété.

Il faut prendre garde à **ce que l'on veut faire** : très souvent, c'est la valeur actuelle que l'on veut supprimer. Dans ce cas, il vaut mieux affecter à la propriété une valeur "neutre" qui sera transcodée à false dans un test : 0 pour les nombres, **`""`** (ou **`''`**) la chaîne vide pour les chaînes, **`false`** pour les booléens, **`null`** pour les objets.

La suppression de propriété est réellement **une suppression radicale** : elle ne conserve pas l'identifiant de la propriété.

```
delete monObjet.propriete ;
delete monObjet[propriete] ;
delete monObjet[ "nouvelle propriété"] ;
```

On peut évidement affecter la valeur **`undefined`**. Mais attention dans ce cas à l'interprétation du test d'existence et d'initialisation. Si on teste une propriété (ou toute variable) qui n'existe pas, la propriété **`undefined`** est retournée. Il faut donc utiliser un autre moyen pour savoir si une propriété est définie ou pas. C'est l'objet du paragraphe qui suit.

note : l'opérateur **`delete`** n'opère la suppression que sur des propriétés «utilisateur» (non natives), pas sur des variables déclarées (**`var`**). Cet opérateur retourne un booléen, mais la valeur donnée du retour est tellement bizarre qu'elle n'est pas utilisée.

4.4. contrôler si une propriété est définie.

Il existe un opérateur qui dit si une propriété existe dans un objet, l'opérateur booléen **`in`**. La syntaxe est : **`chaîne in objet`**.

```
1. <script type="text/javascript">
2.     var monObjet = {
3.             propUn : "une chaîne " ,
4.             propDeux : 3.14 ,
5.         }
6.     /* une nouvelle propriété */
7.     monObjet.propTrois = "une nouvelle propriété  " ;
8.     monObjet.propQuatre = 2.75 ;
9.     alert ( monObjet.propTrois+monObjet.propQuatre ) ;
10.    /* on supprime propTrois et on teste */
11.    delete monObjet.propTrois ;
12.    if ( monObjet.propTrois == undefined )
13.         alert ("monObjet.propTrois est non défini") ;
14.    else alert ("monObjet.propTrois ???") ;
15.    /* on met monObjet.propQuatre à undefined */
16.    monObjet.propQuatre = undefined ;
17.    if ( monObjet.propQuatre == undefined )
18.        alert ("monObjet.propQuatre est non défini") ;
19.    else alert ("monObjet.propQuatre ???") ;
20.    /* on teste maintenant l'identité */
21.    if ( "propTrois"  in monObjet ) alert ("monObjet.propTrois existe") ;
22.    else alert ("monObjet.propTrois n'existe pas") ;
23.    if ( "propQuatre" in monObjet ) alert ("monObjet.propQuatre existe") ;
24.    else alert ("monObjet.propQuatre n'existe pas") ;
25.</script>
26.<!-- *** h0304_delete *** -->
```

5. Représenter l'occupation des objets en mémoire.

5.1. un objet sans code exécutable (type `object`).

Le schéma qui suit n'a aucune prétention à décrire comment fonctionne effectivement la gestion de la mémoire des données en JavaScript, mais simplement de **donner une petite idée des mécanismes mis en jeu**. On y trouvera donc une symbolisation de l'implémentation simplifiée de deux objets, le second étant une propriété du premier. On n'a pas figuré de fonctions. Voir le schéma ci-dessous.

La table d'un objets :

Tout objet est présent en mémoire grâce à une table. On peut l'appeler table de hachage (*hash table*), car c'est effectivement ce qu'elle est : une table avec deux colonnes. La première contient les identifiants des propriétés et la seconde, **la référence** de leurs valeurs si elles sont initialisées. Pour «référence», on dit aussi pointeur, ou adresse ; une référence est effectivement une façon de retrouver des données en mémoire, où chaque octet a un numéro, son adresse. La référence est ici symbolisée par un petit cercle suivit d'un lien fléché qui va là où la donnée intéressante se trouve.

La table d'allocation mémoire :

La table d'allocation mémoire est unique. Elle est bien évidemment sans commune mesure avec la représentation schématique donnée ci-dessus. C'est cette table qui regroupe 4 signifiants par item :

- le nombre d'occurrences : c'est le nombre de liens qui y arrivent. Quand un nouveau lien est créé, le compteur est incrémenté. Quand un lien est supprimé à cause d'un changement de valeur d'une propriété, le compteur est décrémenté. Lorsque la valeur devient nulle, c'est que la partie de mémoire allouée ne sert plus à rien : le ramasse-miettes peut alors se charger de récupérer des zones mortes.

- le type est symbolisé par les quatre lettres **`n, s, b, o`** pour chacun des types du langage qui consomment de la mémoire. Le type **`undefined`** n'en consomme pas. Le type indique quelle est la structure de la zone mémoire allouée.

  note : L'objet **`null`** est un objet qui lui non plus ne consomme pas de mémoire, c'est un objet qui n'a pas de propriété (même prédéfinie).

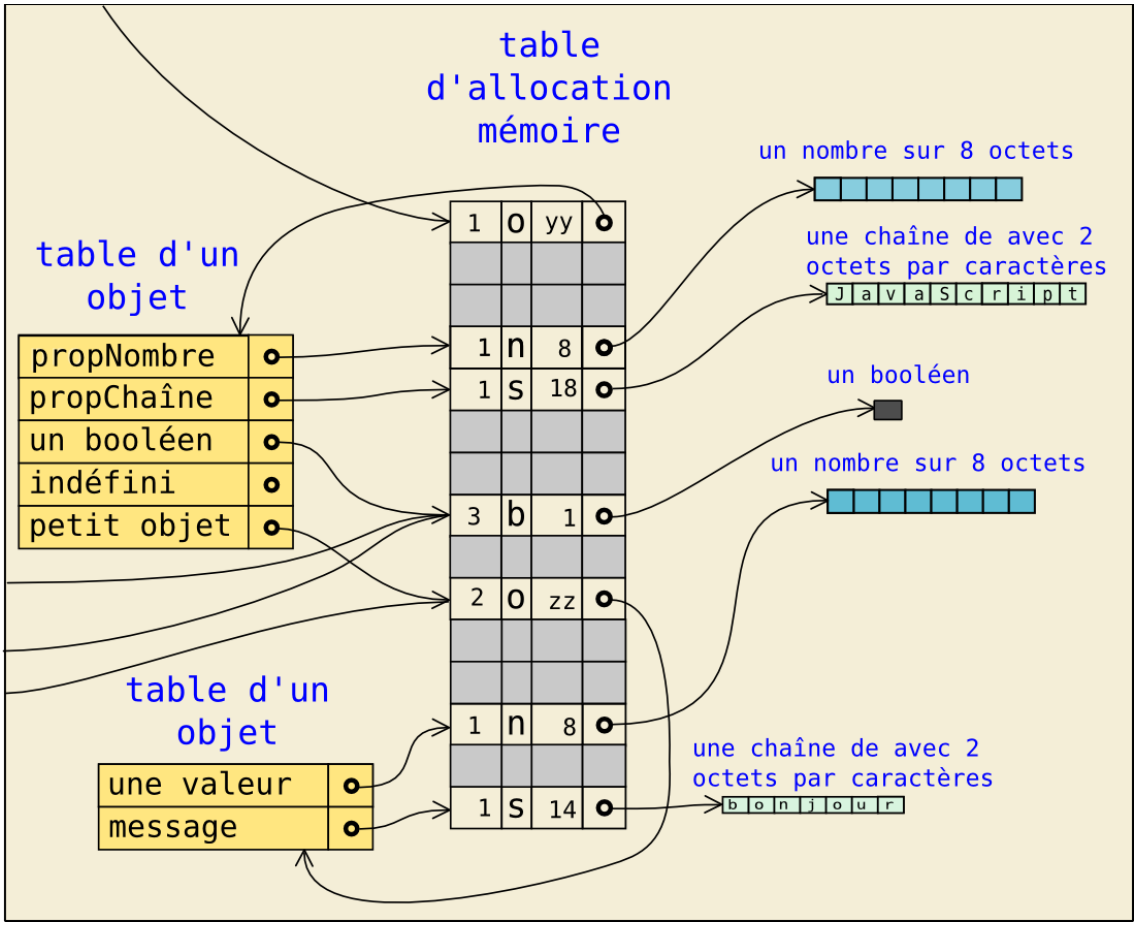
- la quantité de mémoire consommée par la donnée ; si cette quantité est stable pour les nombres et les booléens, il n'en est pas de même pour les chaînes et le tables des objets. On rappelle que l'unité de mémoire adressable est l'octet.

- la référence à la zone mémoire utilisée.

* Une même donnée peut être utilisée par plusieurs propriétés, appartenant ou non à des objets distincts : d'où la nécessité du **compteur d'occurrences**.

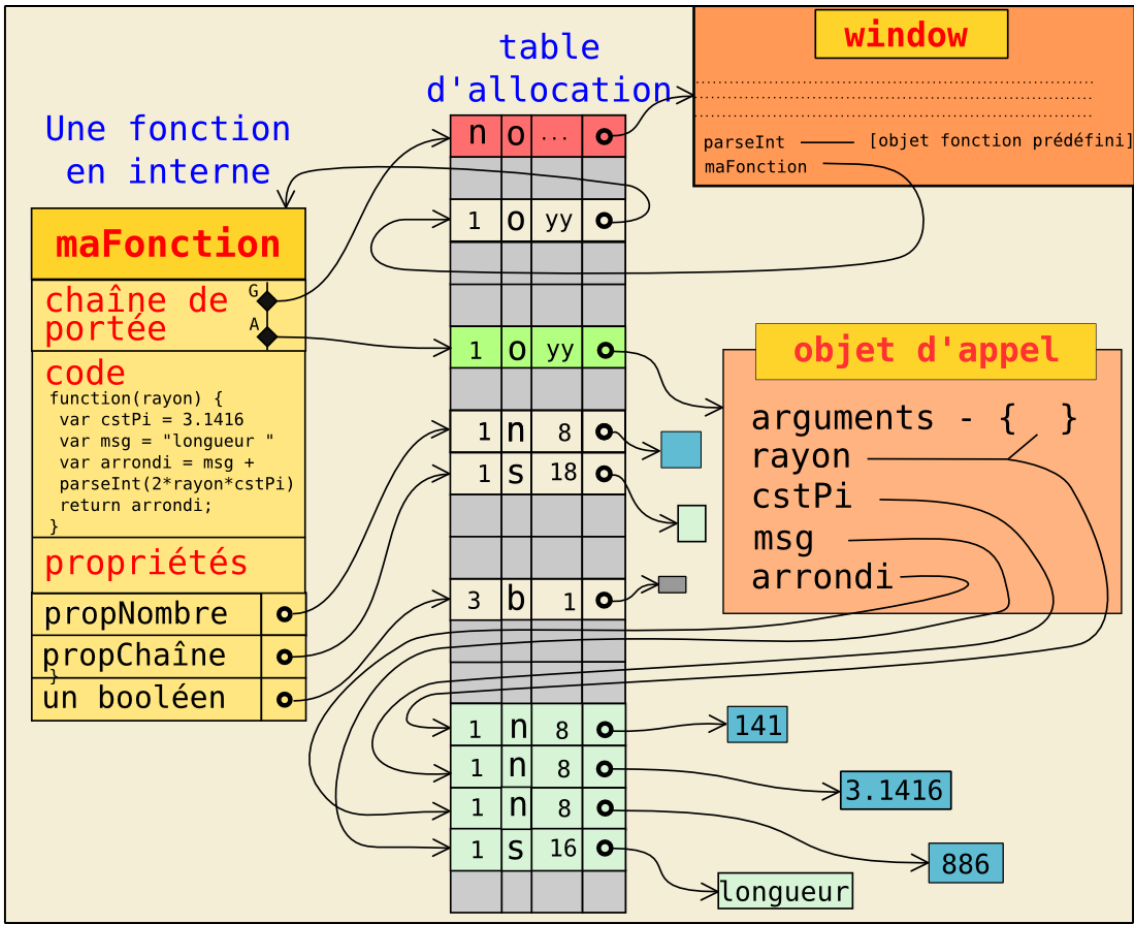
* Tout objet (sauf l'objet global) est aussi une propriété : non seulement il est un objet de la mémoire, mais il possède un lien entrant, provenant de la table d'allocation mémoire...

* Le type est important pour la gestion mémoire : les objets primaire sont des feuilles, et leur suppression entraîne uniquement la récupération de leur consommation de mémoire. Les objets sont connus de la table d'allocation mémoire par leur *hash table*. Lorsqu'une occurrence disparaît, l'interpréteur décrémente le compteur dans la table d'allocation ; mais il convient également de voir quelles conséquences cela a sur les valeurs des propriétés, et cela de façon récursive (objet qui contient des objets, qui contient des objets...). La chapitre sur les fermetures comporte un exemple de cette propagation lors de la suppression d'un objet.



5.2. Représentation d'un objet fonction (type `function`).

Les fonctions présentent une structure plus complexe. La table d'identifications, que l'n trouve habituellement pour les objets sans code exécutables est présente ; mais il y a deux structures supplémentaires : la zone de code et la zone de recherche des valeurs des «variables» (de touts types) utilisées dans le corps de fonction. Cette disposition pour la recherche des variables est appelée chaîne de portée.



La table d'allocation de mémoire.

Cette zone occupe toujours **une position centrale**. Tous les liens ne sont pas figurés. Mais on voit apparaître l'objet **`window`**, et son item d'allocation. Évidemment une quantité de liens devraient apparaître en entrée ; un seul a été représenté.

Quant à la fonction, **`maFonction`**, elle est représentée comme une fonction «globale», c'est-à-dire propriété de l'objet **`window`**.

La zone de code.

Elle est symbolisée par un script de fonction anonyme, ce qui n'est pas loin de la réalité.

On suppose que la représentation de l'occupation mémoire est une «photo» juste avant le **`return`**.

La chaîne de portée.

On peut se représenter la chaîne de portée comme une succession de références aux diverses structures où va se faire successivement la recherche des propriétés mise en œuvre dans le code.

Ici, il y en a deux : l'un qui est titré **objet d'appel** et l'autre est l'espace global, **`window`**. La recherche se fait d'abord dans l'objet d'appel de la fonction, puis dans **`window`**. C'est dans ce dernier qu'est trouvée la fonction (prédéfinie) **`parseInt()`**, qui n'utilise pas la table d'allocation car **`parseInt`** fait partie du noyau de JavaScript.

L'objet d'appel est un objet créé par l'opérateur d'appel de la fonction (la parenthèse double). C'est un

objet presque comme les autres : simplement il stocke les propriétés déclarées (var) et les paramètres de la fonction.

Il comporte toujours une propriété particulière, **`arguments`**, sur laquelle nous reviendrons. Pour le reste, on voit que les paramètres et variables sont traitées comme propriétés de cet objet.

note pratique : On voit que l'on est à la limite de la schématisation : des liens dans tous les sens, et pourtant, il n'y a pas grand chose de représenté ! Aussi, à l'avenir lorsque nous représenterons les fonctions, nous omettrons la table d'allocation, en court-circuitant les liens qui passent par ses items. Nous nous limiterons à l'essentiel : **code, chaîne de portée et objet d'appel**. Les autres données deviennent implicites, ou leur représentation symbolique est réduite à l'essentiel pour la compréhension :

Par exemple, dans l'objet d'appel, on choisit de représenter les couple identificateur/valeur par une symbolique du type **`cstPi - 3.1416`**. Déjà ici, l'objet **`arguments`** a été symbolisé par une accolade double au lieu de lien vers un item de la table d'allocation et un symbole d'objets sans code...

04 : exception et erreurs

1. Notion d'exception.

1.1. erreurs et exceptions.

Il y a exception dans le déroulement d'un programme quand une condition non habituelle survient : une division par zéro, saisie d'une valeur non traitable par la suite du programme, dépassement de capacité de la mémoire, absence d'un fichier à traiter. Dans le pire des cas, il y a plantage du programme, voire de la machine. Au début de la programmation, les exceptions étaient des erreurs fortuites dans le programme, et tout était fait pour se prémunir des circonstances exceptionnelles qui pouvaient se produire ; en général, il s'agissait d'un contrôle strict de tous les paramètres entrant dans un calcul par exemple.

Mais tout n'est pas totalement prévisible ou encore les contrôles ralentissent inutilement la machine pour des cas qui ne devraient pas se produire. On a donc admis qu'il pouvait y avoir des failles dans les dispositions prises, mais pour éviter le plantage, on a, en cas d'erreurs détectables par la machine, dévié le déroulement du programme ver un gestionnaire d'erreur. C'est l'époque du **`onerrorgoto`** du Basic.

1.2. gestionnaire d'exception.

Un gestionnaire d'exception est un segment de programme qui doit être capable de relancer le programme, en fournissant la nature de l'exception et le comportement souhaité.

Le signalement d'une exception peut être automatique (il correspond à une exception connue par le langage de programmation).Ou bien déclenché par le programmeur (utilisation d'une primitive de signalement).

La souplesse de la gestion des exceptions a conduit les langages de programmation à laisser au programmeur la possibilité de construire des exceptions non prévues et de les signaler. C'est même devenu un élément important de la construction des algorithmes, partant du principe qu'il vaut mieux *s'excuser une fois que d'être constamment en train de surveiller tout ce qu'on fait de façon tatillonne*.

2. l'instruction `throw`.

2.1. syntaxe.

L'instruction **`throw`** a la syntaxe suivante :

`throw` ***`argument`***

L'argument peut être à peu près tout : une chaîne de caractère (un message d'erreur !!!) , un nombre fini (un code d'erreur), un objet quelconque. Certains objets sont préprogrammés, les objets **`Error`**.

L'instruction permet de programmer **le signalement d'une exception** dans un script. Le traitement (la capture de l'exception) est réalisé par une instruction **`catch(`*`argument du throw`*`)`**.

Normalement, cette instruction se situe dans un corps de fonction. Elle n'est pas gérable au niveau "global".

2.2. fonctionnement.

Lorsque l'instruction **`throw`** à l'intérieur de la fonction f_a est exécutée, le déroulement normal du programme est interrompu. L'interpréteur cherche s'il lui correspond (voir au paragraphe suivant comment se fait la correspondance) un gestionnaire **`catch()`** dans la fonction f_{a-1} où a été appelée la fonction f_a. En cas de succès, la capture se fait. Sinon, on recherche de même dans f_{a-2} où f_{a-1} a été appelée et ainsi de suite, jusqu'au niveau global. Si aucun **`catch()`** n'est trouvé, il y a erreur du script (en général, le reste du script est ignoré ; l'interprétation s'arrête).

3. les instruction try/catch/finally.

3.1. syntaxe.

try *bloc de code*
catch(*paramètre*) bloc de code
finally bloc de code

Le bloc de code du catch peut être vide.

L'instruction **try** définit le code sous surveillance. Elle est donc nécessaire.

L'instruction **finally** définit un bloc de code qui sera de toute façon exécuté.

L'instruction **try** est suivie d'au moins l'une des deux instructions **catch** et **finally**.

3.2. Code sous surveillance.

- premier cas : aucune exception n'est signalée, ni dans le bloc, ni dans les fonctions appelées. Le bloc **try** arrive à son terme. S'il y a un bloc **catch**, il le saute. Si le bloc **finally** est présent, il est exécuté.

- deuxième cas : une instruction n'arrive pas à son terme parce qu'une exception non gérée est signalée dans une fonction appelée. Alors, s'il y a une instruction **catch**, l'exception est gérée. Sinon, il reste à espérer que la fonction actuelle a dans ses ascendants une fonction avec un **try/catch** qui la protège ; ce cas **try** sans **catch** est rare !. S'il y a un bloc **finally**, il est exécuté après le traitement de l'exception.

3.3. le paramètre du catch.

catch dispose d'un paramètre ; si l'exception est signalée par un **throw**, c'est le paramètre du **throw** qui devient automatiquement le paramètre réel du catch. Sinon c'est un objet **Error** fourni par le système ou créé par le programmeur. Un objet de ce type possède les propriétés **message**, **name**, **toString()**. On verra ultérieurement comment créer de tels objets de façon générique.

4. un exemple d'école.

4.1. moyenne harmonique.

note : l'exemple qui suit est d'abord une illustration, ce n'est certainement pas un modèle de code de production, ne serait-ce que par l'utilisation des instructions **alert()**, **confirm()** et **prompt().**

La moyenne harmonique de deux nombres a et b est le nombre h qui vérifie : $\frac{2}{h} = \frac{1}{a} + \frac{1}{b}$

Son calcul se fait par le formule : $h = \frac{2*a*b}{a+b}$

Le script demande d'entrer par un **prompt()** les valeurs a et b.

Les contraintes à surveiller son nombreuses :

- a et b doivent être des valeurs finies au sens de JavaScript obtenues à partir de chaînes de caractères ;
- ni a ni b ne peuvent être nuls ;
- a + b ne doit pas être nul ;
- a+b doit être un nombre fini, au sens de JavaScript.

4.2. le script.

```
1.<script type="text/javascript">
2.   /* moyenne harmonique */
3.   var promptValeur1 = function  () {
4.      var valeur1 = prompt ('entrez le premier nombre') ;
5.      valeur1 = parseFloat (valeur1) ;
6.      if (valeur1==0 || !isFinite(valeur1))
7.          throw "la première valeur n'est pas un nombre valide" ;
```

```
8.      return valeur1 ;
9.   } // promptValeur1
10.
11.  var promptValeur2 = function  () {
12.     var valeur2 = prompt ('entrez le second nombre') ;
13.     valeur2 = parseFloat (valeur2) ;
14.     if (valeur2==0 ||  !isFinite(valeur2))
15.         throw "la seconde valeur n'est pas un nombre valide" ;
16.     return valeur2 ;
17.  } // promptValeur2()
18.
19.  var calculValeur = function  () {
20.     var v1 = promptValeur1(), v2 = promptValeur2();
21.     var denominateur = v1+v2 ;
22.     if (!isFinite(denominateur))
23.         throw "la somme des valeurs est trop grande" ;
24.     if (denominateur==0)
25.         throw "la somme des valeurs ne doit pas être nulle" ;
26.     return 2*v1*v2/denominateur ;
27.  } // calculValeur()
28.
29.  var calculMoyenne = function () {
30.     while (true){
31.         try {
32.             var moyenne = calculValeur() ;
33.             alert ("la moyenne harmonique est : "+moyenne) ;
34.         } // try
35.         catch(erreur) {
36.             alert(erreur)
37.         } // catch
38.         finally {
39.             var continuer = confirm ("continuer les calculs : OK") ;
40.             if (! continuer) break;
41.         } // finally
42.     } // while
43.  } // calculMoyenne
44.
45.    calculMoyenne() ;
46.</script>
47.<!-- h0500_exception.html -->
```

* ligne 6 : la première valeur est entrée puis transformée en flottant ; on teste alors si la valeur existe comme nombre et n'est pas nulle. On signale une exception dans le cas contraire, le paramètre de **`throw`** étant une chaîne.

* ligne 19 : La fonction d'appel des deux fonctions précédentes est **`calculValeur()`**. Mais il n'y a aucune protection dans cette fonction ; bien plus elle signale deux exceptions, aux lignes 23 et 25.

* ligne 29 : c'est dans cette fonction qu'on trouve l'appel de **`calculValeur()`**. Cette fois, il y a une protection (**`try`**) sur **`calculValeur()`**, avec une capture d'exception (**`catch`**). Si une exception est détectée, indifféremment dans l'une des trois premières fonctions, elle «remonte» dans la fonction **`calculMoyenne()`** où elle est traitée.

On rappelle que quand l'erreur est détecté, l'interprétation s'arrête et le programme bifurque directement sur le bloc de traitement.

5. exemple avec une erreur détectée par la machine.

5.1. la fonction eval().

La fonction **eval()** prend comme argument une chaîne de caractères. En principe, c'est une expression ou un segment de code JavaScript.

Dans le script qui suit, le contenu de la chaîne est entré à l'aide une fonction prompt().

On pourra essayer : **alert("ceci est un message")** // **3+8** // **var a=3;alert(a+1)** // etc

Essayer le prompt avec **window // document // document.body**

On pourra commettre quelques erreurs : **alert()** // **alert("message"** // **var a=3 , a+1** //

5.2. le script.

```
1.<script type="text/javascript">
2.     var resultat, code, continuer ;
3.     while (true) {
4.         try {
5.             code = prompt ("entrez une expression à évaluer ") ;
6.             resultat = eval(code) ;
7.             if (resultat) alert (resultat) ;
8.         }
9.         catch(erreur) {
10.            alert ("ERREUR " + erreur ) ;
11.        }
12.        continuer = confirm ("vous désirez annuler ou poursuivre") ;
13.        if (!continuer) break ;
14.    }
15.</script>
16.<!-- h0501_exception.html -->
```

* Il n'y a pas de **throw** dans le script. Dans ce cas, le paramètre erreur est un objet **Error** ou similaire (**SyntaxError**). Dans ce cas, son introduction dans une chaîne conduit à un transtypage en chaîne, qui donne le genre de l'exception et quelques précisions. L'objet **Error** possède plusieurs propriétés qui permettent une orientation vers un traitement spécifique. Ce n'est pas ce qui nous préoccupe ici.

05 : variables

avertissement liminaire :

Le fonctionnement des variables est **très spécifique dans JavaScript** : il faut mieux oublier la façon dont sont gérées les variables dans les autres langages. Les similitudes apparentes peuvent induire des présupposés qui ne font que rendre incompréhensible le comportement de JavaScript dès que des problèmes de déclaration ou de *scope* des variables apparaissent.

1. variables dites «globales».

1.1. déclaration et variables globales.

On nomme en général «variable globale» une variable qui est reconnue partout : dans la partie globale du script ou dans le corps des fonctions où elles ne sont par redéclarées (mot réservé **`var`**).

On dit aussi qu'en Javascript, toute variable non déclarée par **var** est globale, quel que soit son lieu d'initialisation . De plus l'initialisation d'une variable équivaut à en faire la déclaration implicite. Mais on recommande par ailleurs de toujours déclarer les variables. Qu'en est-il **précisément ?**

1.2. JavaScript ne connaît pas les variables globales.

Quand l'interpréteur JavaScript est lancé, il crée un objet global. C'est un objet, avec ses propriétés propres. Dans les deux principales implémentations de JavaScript, le web et flash (ActionScript), cet **objet global est nommé**, il s'appelle **`window`**.

Or cet objet a une double particularité :

- c'est **un objet d'appel** : un objet d'appel est un objet créé en même temps qu'une fonction. Il a comme propriétés les paramètres et les variables déclarées dans le corps de la fonction. L'objet **`window`** est un objet d'appel particulier, puisqu'il est créé non dans une fonction, mais par le lancement de l'interpréteur. **Mais son usage est celui de tous les objets d'appel**. Quand l'interpréteur rencontre une déclaration de variable (**`var`**) il place la variable (initialisée ou non) comme propriété dans l'objet d'appel actuel. Si la déclaration est dans la partie globale du script, c'est l'objet global qui est choisi, donc **`window`**.

- **`window`** est **l'objet par défaut** : partout où un **identificateur d'objet** "propriétaire" est attendu, on peut l'omettre si c'est **`window`**. En pratique, quand l'interpréteur JavaScript n'arrive pas à identifier l'objet auquel appartient une propriété, (à "résoudre" un identificateur non qualifié**),** il utilise l'identificateur **`window`**. Et ceci , que ce soit dans l'espace global ou dans un corps de fonction.

Autrement dit écrire **`monIdentificateur`** et **`window.monIdentificateur`** sont rigoureusement équivalents lorsque **`monIdentificateur`** n'est pas déjà reconnu comme une propriété (en gros : paramètre ou variable locale d'une fonction).

Attention :

Si la propriété existe déjà, la redéclaration avec **`var`** est sans effet puisque la propriété existe déjà. Il ne faut donc pas espérer remettre une propriété à la valeur **`undefined`** en la déclarant à nouveau.

En résumé :

* déclarer une variable avec **`var`** dans l'espace global revient à l'ajouter comme propriété de type **`undefined`** dans **`window`**.

* affecter un identificateur qui n'existe pas encore dans l'espace global, est équivalent à enrichir **`window`** d'une propriété nouvelle . Écrire **`monNouvelIdentificateur = maValeur`** est en fait équivalent à :

```
window.monNouvelIdentificateur = maValeur
```

On retrouve ainsi justifiée la règle d'usage. Mais il faut bien voir qu'il y a en fait deux démarches différentes mises en jeu dans la création d'une variable globale.

1.3. Un cas particulier : les fonctions déclarées sur le mode traditionnel.

On a vu que la déclaration **`function maFonction (...) {...}`** est la manière connue dans les autres langages (*l'ancienne manière !*) et elle est équivalente à :

```
var maFonction = function(...) {...}
```

Cela est exact, mais incomplet : lors de la création de l'objet , l'interpréteur crée avant toutes choses, les propriétés déclarées explicitement par **`var`** dans la séquence, avant même de commencer l'interprétation. Par contre, les initialisations restent faites dans l'ordre du flux d'entrée. Il ne faut donc pas utiliser une variable qui n'a pas été initialisée auparavant. Ceci ne s'applique pas aux variables déclarées ai sein d'instructions composées.

Pour les fonctions définies *à l'ancienne manière*, il n'en est pas ainsi. Non seulement leur identificateur est créé comme propriété avant l'exécution, **mais elles sont aussi initialisées**. Ce qui fait que, si on ne peut utiliser une propriété non initialisée explicitement, il n'en est pas de même des fonctions à *l'ancienne manière* qu'on peut utiliser **avant leur déclaration**.

Une règle de bonne pratique veut que l'on déclare en premier lieu toutes les variables, en tête du script et qu'on les initialise le plus vite possible. Ceci rend obsolète cette priorité d'initialisation un peu déroutante, inutile et pouvant induire une mauvaise interprétation de la part du programmeur.

1.4. Mise en évidence des variables globales comme propriétés de window.

Le script :

```
1.<script type="text/javascript">
2.      __unePropriete = "message" ;
3.      var __fonctionTest = function () {return "fonction test"} ;
4.      function __FonctionModeC () {return "comme en C"} ;
5.      var __fonctionWindow = function () {
6.                                  __chn = "";
7.                                  for (var prop in window) {
8.                                      __chn += prop + "\n";
9.                                  }
10.                                 alert (__chn) ;
11.                             }
12.     __fonctionWindow();
13.     var __variableTest = 3.14 ;
14.     </script>
15.<!-- h0600_globales.html *** -->
```

* On a écrit tous les identificateurs avec un double souligné pour pouvoir les repérer facilement.

* ligne 6 : une variable globale ;

* noter à la ligne 13 une déclaration tardive ; elle apparaît cependant comme propriété ;

* noter dans les propriétés de **`window`** , les fonctions **`alert(), confirm(), prompt()`** etc.

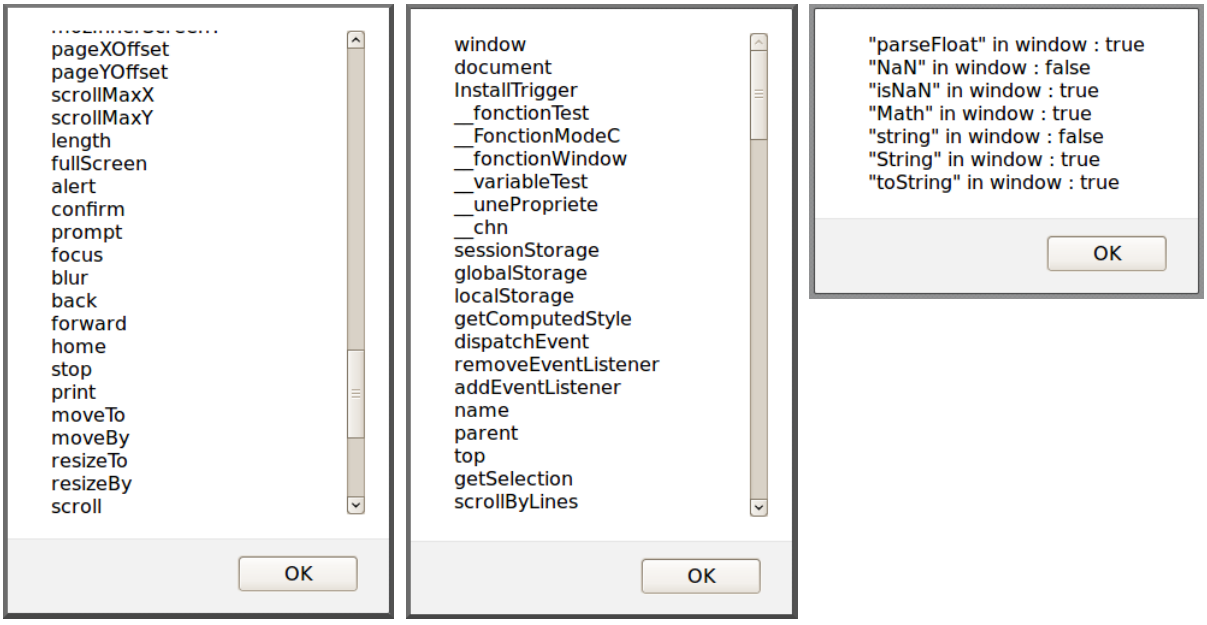
Important :

Certaines propriétés dites «prédéfinies» et qui appartiennent au cœur de JavaScript n'apparaissent pas dans les boucles **`for/in`**. Elles sont supposées connues car faisant partie de la définition même du langage. On peut citer pour mémoire : **`parseInt, parseFloat, isNaN, Function, Math`** etc**.**

Alors, pour tester plus profondément les propriétés de **`window`**, il faut faire le travail à la main ! Ainsi, une instruction du type : **`alert( "parseInt" in window)`** va afficher **`true`**. Ne pas oublier qu'en interne, les identifiants sont des chaînes de caractère, et donc de bien mettre les quotes.

```
1.<script type="text/javascript">
2.      var chn = "" ;
3.      chn += '"parseFloat" in window : '+ ("parseFloat" in window)+"\n" ;
4.      chn += '"NaN" in window : '       + ("Nan"        in window)+"\n" ;
5.      chn += '"isNaN" in window : '     + ("isNaN"      in window)+"\n" ;
```

```
6.      chn += '"Math" in window : '      + ("Math"       in window)+"\n" ;
7.      chn += '"string" in window : '    + ("string"     in window)+"\n" ;
8.      chn += '"String" in window : '    + ("String"     in window)+"\n" ;
9.      chn += '"toString" in window : '  + ("toString"   in window)+"\n" ;
10.      alert (chn);
11.</script>
12.<!-- h0607_predefini.html -->
```



1.5. Les valeurs.

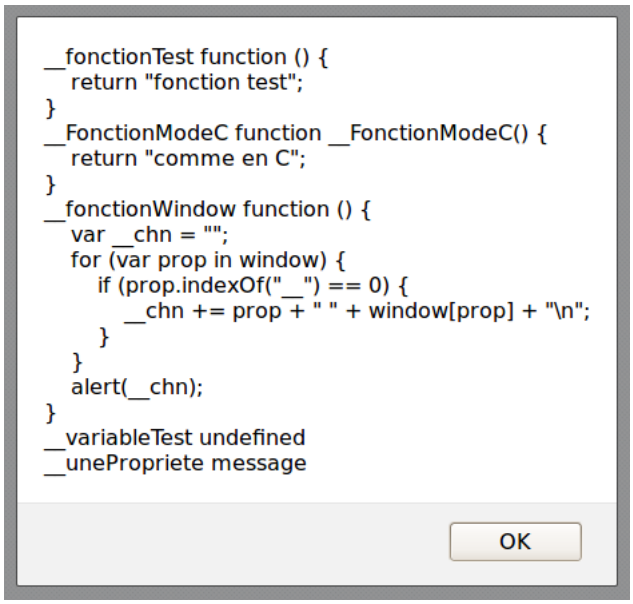
On reprend le script **h0600_globales**, mais cette fois en n'affichant que les variables globales définies dans le script et avec leur valeur d'initialisation. La variable **__chn** a été rendue locale à la fonction d'analyse, et n'apparaît pas.

Le script :

```
1.<script type="text/javascript">
2.    __unePropriete = "message" ;
3.    var __fonctionTest = function () {return "fonction test"} ;
4.    var __fonctionWindow = function () {
5.                               var __chn = "";
6.                               for (var prop in window) {
7.                                   if (prop.indexOf("__") == 0)
8.                                   __chn += prop+" "+window[prop]+"\n";
9.                               }
10.                              alert (__chn) ;
11.                          }
12.
13.    __fonctionWindow();
14.    var __variableTest = 3.14 ;
15.    function __FonctionModeC () {return "comme en C"} ;
16.    </script>
17.     <!-- h0601_globales.html *** -->
```

* ligne 7 : on teste si l'identificateur de la propriété commence par __

* ligne 8 : **prop** est connue comme chaîne de caractère ; on utilise donc la syntaxe correspondante, celle du tableau associatif avec des crochets **[]**.



* Noter que lors de l'appel de la fonction (ligne 14), la variable **`__variableTest`** existe, mais est **`undefined`**.

* rien ne distingue la fonction à l'ancienne des fonction définies comme variable.

* par contre, on voit que la fonction à l'ancienne, déclarée ligne 15, après l'appel de la fonction d'affichage, est déjà initialisée, contrairement à la variable explicitement déclarée avant de l'utiliser.

2. fonction et variable locale.

2.1. la notion d'objet d'appel.

Lors de l'appel de la fonction, préalablement à l'exécution de celle-ci, un objet, dit «**objet d'appel**», est créé. Cet objet est anonyme, contrairement à **`window`** qui est aussi un objet d'appel, **mais qui lui est nommé**. Il n'y a pas de moyen simple de mettre en évidence l'objet d'appel d'une fonction. Le but de cette création est de stocker les paramètres et variables «locales» (c'est à dire celles déclarées explicitement dans le corps de la fonction). Nous avons vu que les variables initialisées et non déclarées étaient *ipso facto* «globale». **Que comporte cet objet d'appel et selon quel mécanisme est-il constitué ?** On sait que comme pour les variables globales, les propriétés sont utilisées sans référence à l'objet d'appel, qui devient **objet par défaut** pour le code de la fonction.

2.2. la propriété `arguments`.

`arguments` est une propriété de l'objet d'appel ; il contient tous les **paramètres réels de la fonction appelée** c'est à dire la liste des valeurs allouées **lors de l'appel de la fonctions**. Leur nombre peut être différent de celui des paramètres formels cités dans la définition de la fonction. Attention, il ne s'agit pas des variables déclarées, dont rien n'impose qu'elles soient présentes à l'appel.

Le mécanisme est le suivant : une suite de propriétés, d'identifiants nommés "0", "1", "2" etc est créée. Il y en a autant que de **paramètres réels** qui sont respectivement les valeurs liées à ces identifiants. Attention, **`arguments`** n'est pas un tableau ! Cette propriété spéciale, **`arguments,`** possède une propriété **`length`**, qui donne le nombre de propriétés de **`arguments`**. Il possède aussi une propriété **`callee`** qui désigne la fonction actuelle. Attention, ceci n'est pas trivial : si la fonction a un identificateur, alors **`callee`** n'est pas nécessaire. L'identificateur désigne alors l'objet fonction que l'on peut manipuler avec ce nom. Mais si la fonction est une fonction anonyme, il n'y a pas d'autre moyen que **`callee`** pour travailler avec elle ; par exemple pour l'enrichir de propriétés nouvelles.

On peut donc gérer dans le corps de fonction **des paramètres réels en surnombre** ; et disposer des fonctions à **nombre indéterminé de paramètres**. Cela n'est pas sans rappeler, pour les connaisseurs, des pratiques de Python et de Ruby.

Exemple :

La fonction **`alertListe()`** a un nombre quelconque de paramètres ; elle affiche une liste de paramètres réels comme une suite de lignes.


```
1. <script type="text/javascript">
2.     var alertListe = function () {
3.         var chn = "" ;
4.         for (var i = 0 ; i< arguments.length ; i++) {
5.             chn += arguments[i]+"\n" ; // transtypage automatique
6.         }
7.         alert (chn) ;
8.     }
9.     alertListe("toto", 3.14, function() {return "message"})
10.</script>
11.<!-- h0602_arguments.html *** -->
```

2.3. Les paramètres formels comme alias de valeurs de `arguments`.

Supposons qu'une fonction ait deux paramètres formels :

```
maFonction = function (par1, par2) {...}
```

`par1` et **`par2`** sont-ils des variables à part entière ? **La réponse est non** : ce ne sont que des **alias** de **`arguments["0"]`** et de **`arguments["1"]`**. Tout changement de **`par1`** se répercute sur la valeur de **`arguments["0"]`** et réciproquement. Idem pour **`pars2`** et **`arguments["1"]`**, etc. Les paramètres formels qui apparaissent dans le corps de fonction sont donc résolus comme des propriétés dont les valeurs sont également dans **`arguments`**.

note : la particularité ci-dessus est unique dans JavaScript. En effet, en Javascript il n'y a pas à proprement parler d'alias : on peut avoir deux identificateurs se référant au même objet ; mais si on affecte un des identificateurs, l'autre reste pointé sur l'objet initial.

Exemple :


```
1.<script type="text/javascript">
2.   var alertListe = function (par1, par2) {
3.      var chn = "" ;
4.      par1 = "bibi" ;
5.      arguments[1] = 2.72 ;
6.      alert (par1+" // " + arguments[0] + "\n"+
7.             par2+" // " + arguments[1]) ;
8.   }
9.   alertListe("toto", 3.14)
10.</script>
11.<!-- h0603_param.html *** -->
```

2.4. Les variables locales.

Les variables locales sont les identificateurs déclarés par **`var`** ou qui sont des noms de fonctions *ancienne manière*. Attention leur traitement est différent :

1- les identificateurs déclarées par **`var` dans la séquence** du corps de fonction sont **tous** des propriétés de l'objet d'appel. Ces identificateurs même non initialisés, sont recensés lors de la création de l'objet d'appel. Par contre, l'initialisation se fait en séquence, c'est-à-dire dans l'ordre où l'on trouve les instructions dans le corps de fonction. Les variables déclarées dans les instructions composées (penser à la boucle : **`for(var i=0 ; i<7 ; i++)...`**) ont rigoureusement le même statut, sauf qu'elles ont une déclaration retardée. Il est d'usage commun de déclarer une variable dans une instruction composée et de la tester en dehors (tester la variable de la boucle **`for`** en sortie de boucle).

Attention :

Une propriété (globale) peut donc très bien être initialisée avant son **`var`** de création : cela n'a aucune importance ; même si ce n'est pas une bonne pratique, il n'y a pas de confusion avec une globale. Ainsi, les deux lignes suivantes sont équivalentes :

```
maNouvelleVariable = 3.1416 ; var MaNouvelleVariable ;
var MaNouvelleVariable ; maNouvelleVariable = 3.1416 ;
```

Elles créent toutes deux une variable locale !

2- les identificateurs qui proviennent d'une déclaration de fonction *ancienne manière* sont créés au même moment, mais sont initialisés, **où qu'ils se trouvent en séquence**. On a le même comportement que dans le script global. Répétons ici la règle de bon usage : déclarer toutes les variables en tête du corps de fonction, et les fonctions comme des fonctions anonymes, valeurs de variables déclarées.

3- on a évoqué la question **des déclarations `var`, internes à des instructions composées** (boucles, conditionnelles) : ces déclarations internes sont transformées en propriétés de l'objet d'appel de façon dynamique lors de l'exécution de ces instructions.

Insistons : ce sont bien des propriétés valides dans le corps de la fonction, pas des variables de bloc comme dans certains langages. En pratique **il vaut mieux** déclarer ces variables en tête du corps de la fonction : cela peut éviter bien des glissement dans l'interprétation de ces identificateurs puisque leur signification est **absolument différente de ce que l'on a dans les langages qui connaissent les variables de bloc** (le C par exemple).

Mais le choix ultime d'un bon usage revient au programmeur : par exemple, ce n'est par une mauvaise manière que d'écrire : **`for( var i = 0 ; i< arguments.length ; i++) {…`** Le risque est d'oublier de déclarer la variable de boucle, car elle est alors une variable globale. La déclarer dans la boucle est peut-être moins risqué que d'avoir une globale inattendue perturbatrice en écrivant :

~~`for (i = 0 ; i< arguments.length ; i++) {…`~~

note : une redéclaration d'une variable existante est sans effet. En cas de multiples initialisations, c'est la dernière qui est prise en compte (ce qui n'est pas trivial pour les *fonctions à l'ancienne*).

Exemple 1 :

```
1. <script type="text/javascript">
2.      var fnEtude = function () {
3.              alert (xxx) ; // undefined
4.              var xxx = 1937 ;
5.              alert (xxx) ; // 1937
6.              alert (fnAncienne ()) ; // à l'ancienne
7.              function fnAncienne () {
8.                          return "à l'ancienne";
9.                      }
10.             // alert (fnNouvelle()) ; ERREUR
11.             alert (fnNouvelle) ; // undefined
12.             var fnNouvelle = function () {
13.                     return "nouvelle mode" ;
14.                 }
15.             alert (fnNouvelle()) ; // nouvelle mode
16.    }
17.    fnEtude() ;
18. </script>
19. <!-- h0604_varloc.html *** -->
```

Exemple 2 : le piège.

A gauche : le script, avec ses déclaration et une redéclaration de fonction, selon les deux modes.

A droite, l'ordre effectif de déroulement, si l'on suit scrupuleusement ce qui vient d'être dit.

```
var fn = function () {
  uneGlobale = 1234 ;
  uneLocale = 1 ;
  var laSomme = 1 ;
  fnPiege = function () {
    alert("fonction affectée "+laSomme)
  };
  for (var i = 1 ; i < 10 ; i++) {
    var uneLocale ;
    laSomme += uneLocale;
    }
  function fnPiege () {
    alert("fonction à l'ancienne "+ laSomme)
  };
  fnPiege () ;
  alert ( ("fnPiege" in window) +" "+
          ("uneGlobale" in window)) ;
}
fn() ;
```

```
var fn = function () {
/* les déclarations de propriétés */
  var window.uneGlobale ;
  var laSomme ;
  var i ;
  var uneLocale ;
  var fnPiege ;
/* initialisation de fonction à l'ancienne */
  fnPiege = function () {
    alert("fonction à l'ancienne "+ laSomme)
  };
/* exécution du code séquentiel */
  window.uneGlobale = 1234 ;
  laSomme = 1 ;
  fnPiege = function () {
    alert("fonction affectée "+laSomme)
  };
  for (i = 1 ; i < 10 ; i++) {
    laSomme += uneLocale;
  }
  fnPiege () ;
  alert ( ("fnPiege" in window) +" "+
          ("uneGlobale" in window)) ;
}
fn() ;
```

le résultats :

fonction affectée 10 false true	C'est bien la fonction "affectée" qui a été retenue, car effectivement créée en second ! **`fnPiège,`** quoique déclarée sans **`var`** est effectivement locale et non dans **`window`**.

3. problèmes de scope.

3.1. résolution des identificateurs.

Un identificateur étant utilisé dans une fonction, comment est-il «résolu» ? C'est-à-dire, comment fait l'interpréteur pour trouver la bonne variable (ou paramètre) auquel il correspond ? Le problème est particulièrement important quand des fonctions sont imbriquées. Heureusement, la solution est assez intuitive même si le mécanisme est beaucoup plus sophistiqué ; il sera étudié au chapitre qui suit :

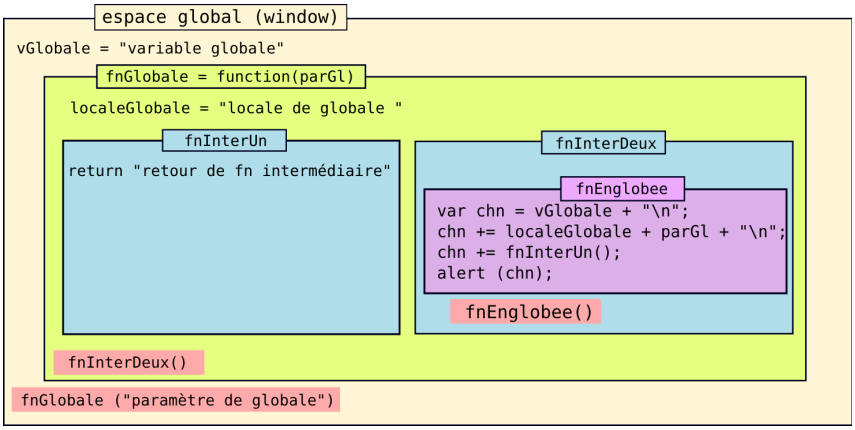
- **première règle** : un identificateur dans une fonction est recherché en priorité dans l'objet d'appel de la fonction. Rappelons que les **paramètres formels sont des alias** des propriétés de l'objet **`arguments`**.

- **seconde règle** : les objets d'appels de **fonctions imbriquées sont chaînés** dans l'ordre inverse de l'imbrication, en remontant jusque l'objet d'appel global (**`window`**). Un identificateur de variable est donc recherché dans la suite des objets d'appel, en remontant éventuellement jusque l'objet global, donc jusqu'à window : la recherche s'arrête sur la première trouvaille.

On dit souvent que depuis une fonction imbriquée, on «voit» les variables des fonctions imbriquantes, une variable occultant les variables de même nom placées au dessus d'elle.

3.2. Une illustration.

principe d'imbrication :



le script :

```
1. <script type="text/javascript">
2.     var vGlobale = "variable globale";
3.     var fnGlobale = function(parGl) {
4.         var localeGlobale = "locale de globale ";
5.         var fnInterUn = function() {
6.              return "retour de fn intermédiaire"
7.         } // fnInterUn
8.         var fnInterDeux = function() {
9.              var fnEnglobee = function() {
10.                 var chn = vGlobale + "\n" ;
11.                 chn += localeGlobale + parGl + "\n";
12.                 chn += fnInterUn();
13.                 alert (chn);
14.             } // fnEnglobee
15.             fnEnglobee();
16.         } // fnInterDeux
17.         fnInterDeux()
18.     } // fnGlobale
19.     fnGlobale ("paramètre de globale");
20.</script>
21.<!-- h0605_scope.html -->
```

l'affichage :

```
variable globale
locale de globale  paramètre de globale
retour de fn intermédiaire
```

06 : fermetures

1. Nécessité d'approfondir la notion de fonction.

1.1. Le problème.

La description de la gestion des variables dans les fonctions qui a été décrite dans le chapitre précédent est **incomplète.** Elle ne rend pas compte d'une situation autorisée par le fait que **toute fonction est un objet.** Et donc comme n'importe quel objet, une fonction peut être affectée à une variable dont le *scope* n'est pas celui de la fonction telle qu'elle a été définie.

Le cas caractéristique est celui d'une fonction englobée, retournée ou affectée par sa fonction englobante dans une variable globale. Quel est alors le *scope* de la nouvelle fonction ? Que deviennent par exemple les références aux variables locales ou aux paramètres de la fonction englobante ?

1.2. un exemple.

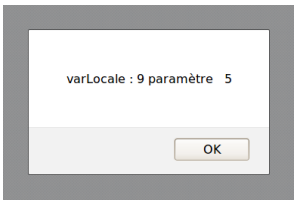
Le script.

```
1.<script type="text/javascript">
2.    var fnAncetre = function(param) {
3.        var fnUtile = function(x) {
4.            return x*x ;
5.        }
6.        var fnFille = function(y) {
7.            var varLocale = fnUtile(y) ;
8.            return "varLocale : "+varLocale+" paramètre "+ param ;
9.        }
10.       return fnFille ;
11.   }
12.   var nouveau = fnAncetre(5) ;
13.   alert (nouveau (3)) ;
14.</script>
```

la question :

La résolution des variables est simple pour la fonction **`fnFille()`**. Beaucoup moins pour la fonction **`nouveau()`** qui est une variable globale qui partage le même code, mais apparemment aussi le même *scope*. Le résultat suivant n'en est que plus étonnant ! Une fois encore, ce qui se pratique dans d'autres langages objets est insuffisant pour décrire le comportement de JavaScript.

résultat :



2. Fonctionnement de la chaîne de portée.

2.1. chaîne de portée.

On a vu que dans le cas de fonction imbriquées, la résolution d'une variable se fait en consultant d'abord **l'objet d'appel** ; puis l'objet d'appel de l'appelant. Si c'est nécessaire, on remonte les appels jusqu'à l'objet global dont on voit qu'il est bien un objet d'appel. **On pourrait décrire ceci en disant qu'il existe un enchaînement des objets d'appel, et que cet enchaînement appartient à la fonction appelée**. Ce n'est pas seulement une facilité d'exposition : il existe bien une composante des fonctions qui formalise ce chaînage : **la chaîne de portée**.

La chaîne de portée fait partie de la fonction, et elle est définie lors de la création de la fonction. Une fonction a donc, dans sa représentation en mémoire, trois parties : **son code, ses propriétés et sa chaîne de portée**. Pour comprendre cette notion, le mieux est d'étudier un exemple simple où l'enchaînement des objets d'appel est aisé à comprendre.

2.2. Le script type.

```
1. <script type="text/javascript">
2.     var fnCercle = function (rayon) {
3.         var localePi = 3.14 ;
4.         var getSurface = function () {
5.             var msg = "surface ";
6.             return msg + rayon*rayon*localePi
7.         }
8.         return getSurface()
9.      }
10.     var diametre = prompt ('le diamètre ? ') ;
11.     alert (fnCercle(diametre/2));
12.</script>
13.<!-- h0701_schema.html -->
```

2.3. Chargement et exécution du script.

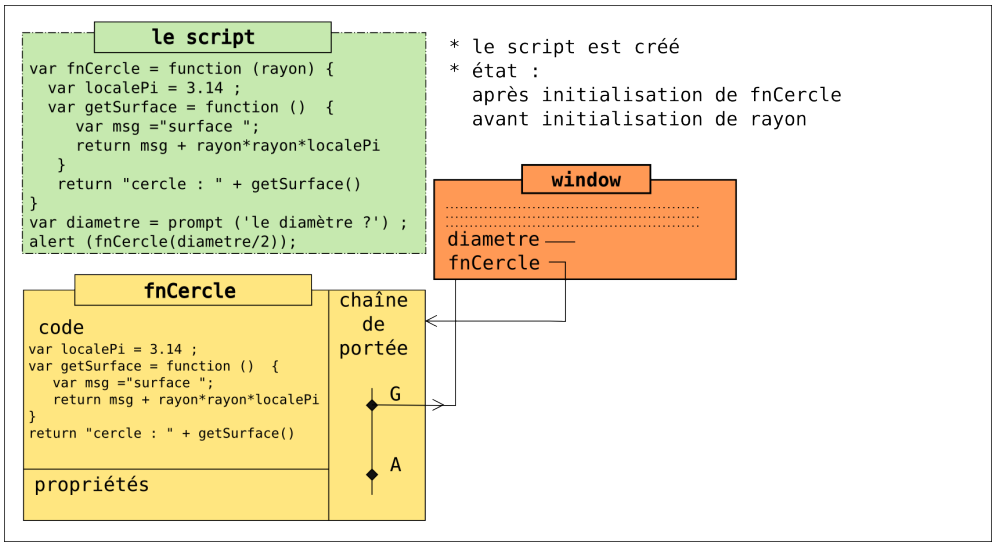
étape 1 : les variables sont créées.

Lorsque le script est créé (lecture dans le navigateur), les variables sont créées selon le processus décrit pour les variables locales des fonctions. A la différence près que l'objet d'appel est préexistant :

Les variables **`fnCercle`** et **`diametre`** sont des propriétés de l'objet **`window`**.

étape 2 :La première instruction du script est exécutée.

Il s'agit de l'initialisation d'une variable avec une fonction anonyme : **la fonction est créée.** Elle est ensuite liée à l'identificateur **`fnCercle`** (lien fléché). La création consiste comme pour tout objet à donner une représentation en mémoire de la fonction ; si elle comporte bien la table de hachage (propriétés), elle comporte aussi un segment pour le code et un segment pour la chaîne de portée.

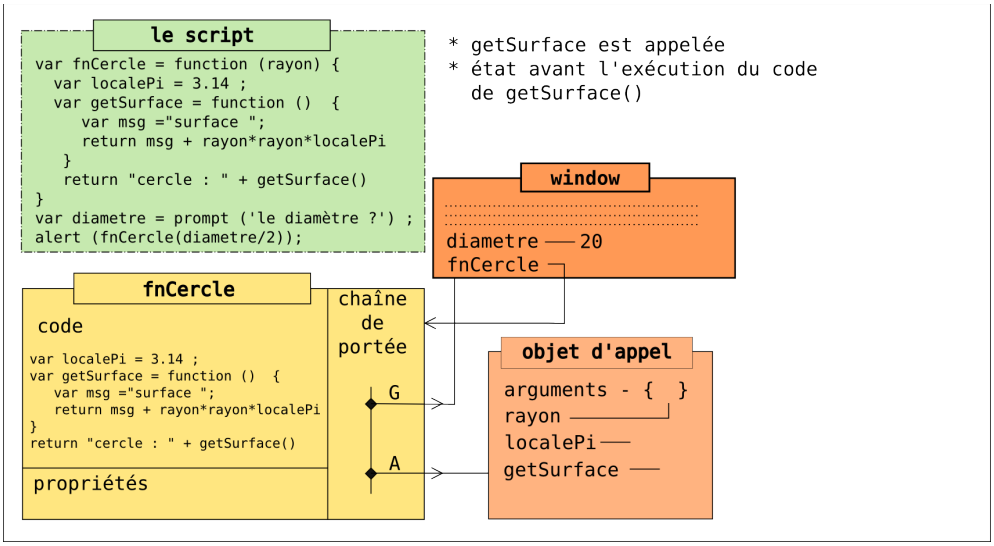


On a figuré la fonction **`fnCercle`** telle qu'elle est en mémoire : composée des trois parties, son code, ses propriétés (on n'en a explicité aucune pour simplifier), et la chaîne de portée. Elle comporte deux nœuds : le nœud A en attente d'une liaison sur l'objet d'appel, et le noeud G, qui comporte effectivement un lien vers l'objet global **`window`**. La chaîne de portée se lit de bas en haut. La résolution des variables de ce code se fera d'abord dans l'objet d'appel puis dans l'objet global.

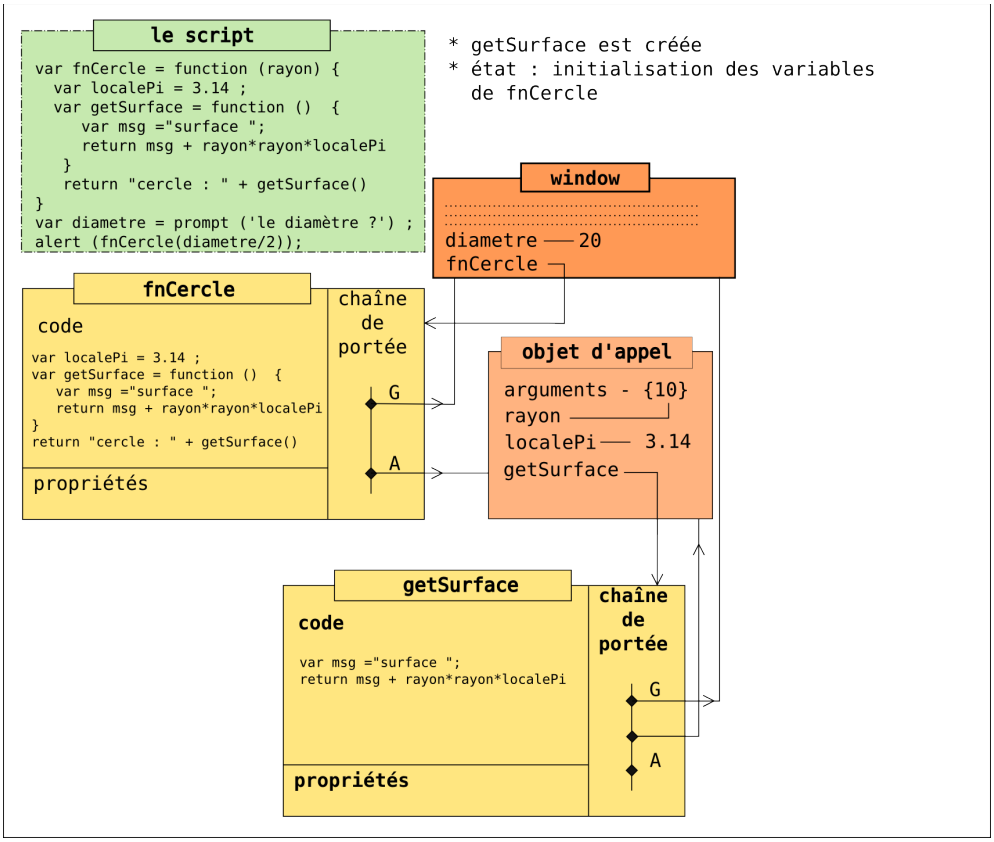
La valeur de la variable globale **`fnCercle`** est la fonction qui vient d'être créée, ce qui se symbolise par le lien fléché. La variable globale**, `diametre`,** n'a pas été encore initialisée.

étape 3 : on initialise la variable globale **`diametre`** (pour l'illustration, la valeur : 20).

étape 4 : la fonction **`fnCercle`** est appelée ; son **objet d'appel** est créé. Il se retrouve avec **le lien issu du premier nœud (en bas) de la chaîne de portée**.



étape 5 : initialisation de l'objet d'appel ; d'abord le paramètre **`rayon`**, puis **`localePi`** puis **`getSurface`**. Pour initialiser **`getSurface`**, il faut créer la fonction anonyme associée et la lier à l'identificateur **`getSurface`** de l'objet d'appel. La fonction a exactement le même schéma que la fonction fnCercle ; à la différence près qu'il y a un troisième nœud, intermédiaire entre G et A : c'est le lien qui pointe un objet d'appel déjà existant, celui de la fonction englobante **`fnCercle`**.



étape 6 : **`getSurface`** est appelée ; l'objet d'appel de **`getSurface`** est créé.

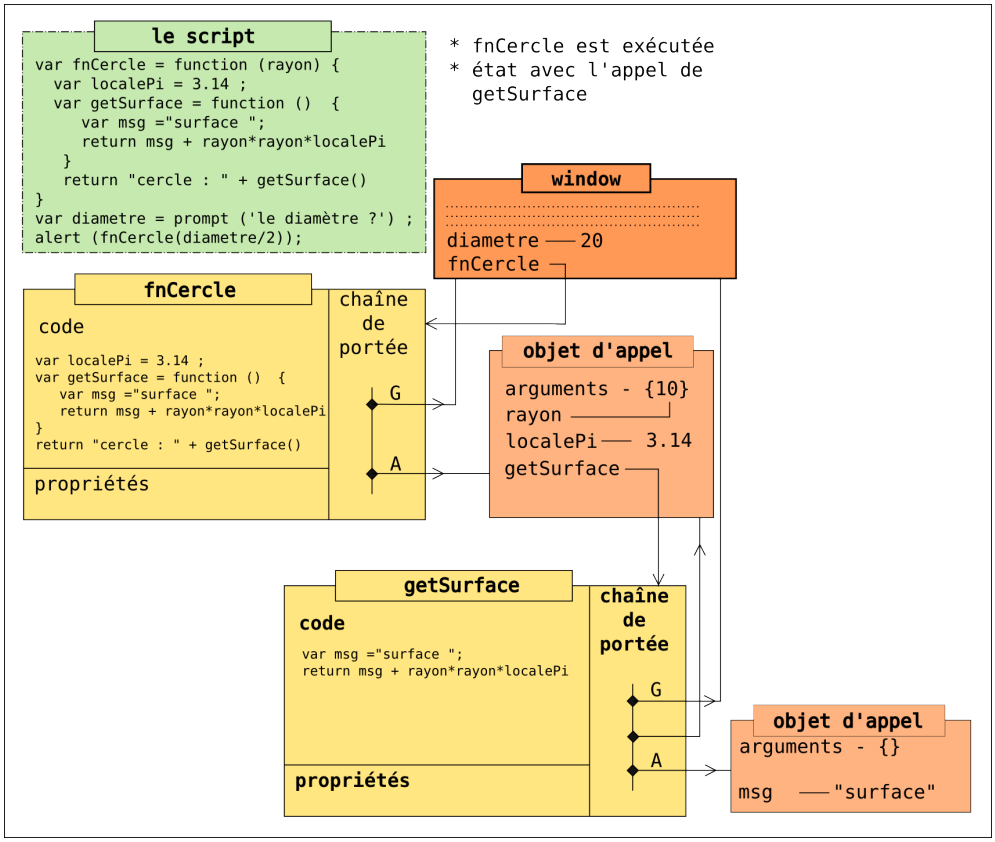
étape 7 : **`getSurface`** est exécuté ; ses variables sont initialisées mais la fonction n'est pas encore terminée par le **`return`**.

étape 8 : la fonction **`getSurface`** retourne (se termine).

étape 9 : la fonction **`fnCercle`** retourne.

Il faut ici faire une digression sur la notion de fin (ou retour) d'une fonction. On dit usuellement que tout ce qui est local non rémanent disparaît. C'est vrai en C, en Pascal, en Java). **C'est faux en ce qui concerne JavaScript**, quoiqu'en dise une littérature mal avisée. Il faut bien faire attention à ce que signifie la terminaison d'une fonction : **C'est uniquement la disparition du lien qui unit la chaîne de portée et l'objet d'appel**.

JavaScript est un langage qui gère la mémoire avec un ramasse-miette. A chaque fois qu'une zone mémoire est allouée, la réservation est notée. Même chose pour les liens (références) qui réunissent des zones mémoires allouées. Plus ou moins régulièrement, un algorithme se déclenche en tâche de fond et, à partir des liens qui restent et des zones effectivement allouée, **détermine les zone orphelines**, c'est-à dire des zones que l'on ne peut plus atteindre par le jeu des liens à partir du programme principal (c'est le cas pour une fonction anonyme dans le programme principal). Les zones mémoires orphelines sont alors récupérées et remises à la disposition du programme.

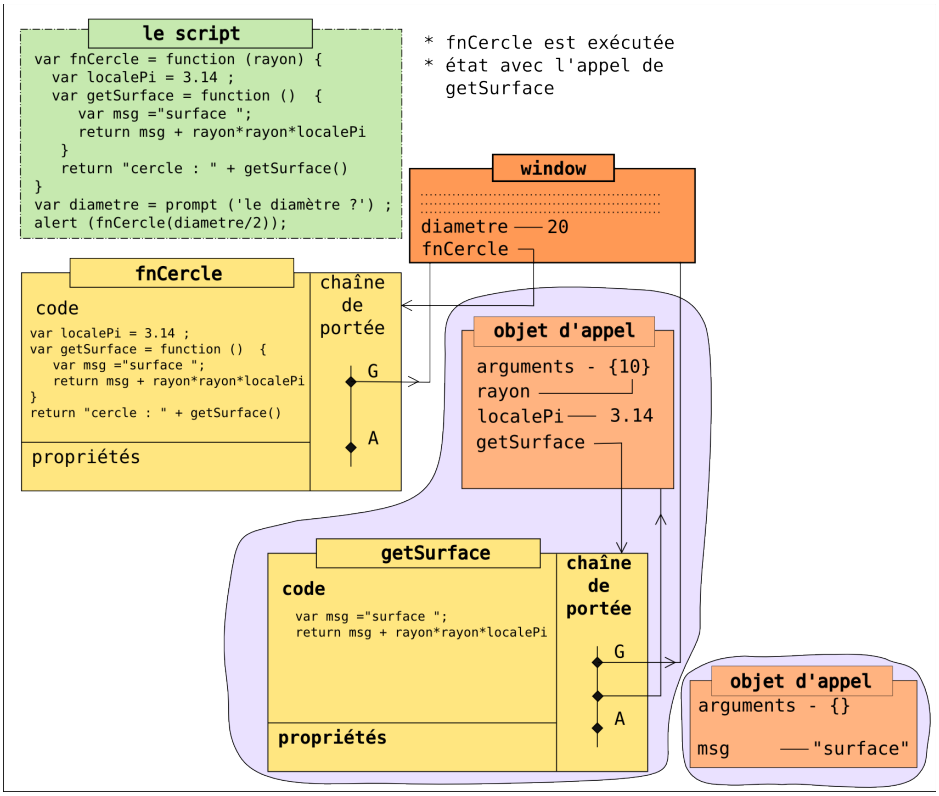


On voit l'importance de la création ou de la suppression des liens ! Dans des langages «avec pointeurs» comme le C ou Pascal, il faut faire ce travail de récupération à la main (ce qui constitue la difficulté majeure des ces langages et leur plus importante source d'erreurs). Dans les langages objets avec ramasse-miettes comme Java ou Python, création et suppression des liens sont automatiques, ainsi que la gestion mémoire ; ceci évidemment au prix d'une perte en performance pure (vitesse d'exécution, mémoire disponible).

Voici l'état de la mémoire quand les deux retours sont faits, en supposant que le ramasse -miettes n'est pas encore passé. Dans le cas présent, il y a deux zones orphelines que l'on peut considérer comme perdues.

Dans cet exemple, la définition d'une zone orpheline n'est pas tout à fait évidente : il faut commencer par identifier les «cycles» (A lié avec B et B lié avec A, voire plus complexes : A lié avec B, B lié avec C, C lié avec A) qui doivent être "enlevés" lorsque l'on recherche quelles sont les zones orphelines. C'est d'ailleurs là une grande difficulté pour programmer un ramasse-miettes ; la question du ramasse-miette reste, aujourd'hui encore en 2012, un problème ouvert.

Mais comme on le verra ultérieurement, ce n'est pas automatiquement le cas lorsqu'il y a des fonctions exportées. Plusieurs liens vers un même objet d'appel peuvent exister, et la disparition de l'un d'eux ne signifie pas la disparition de l'objet.



3. Chaîne de portée et fonction génératrice de fonction.

3.1. Fonction génératrice de fonction.

Une fonction génératrice de fonction est une fonction qui retourne une fonction ; la fonction retournée doit être une fonction **créée au sein de la fonction génératrice** :

- soit une fonction anonyme **`return function() {. . .}`**
- soit une fonction définie dans la fonction génératrice :

```
...
var monRetour = function() {. . .} ;
return monRetour ;
```

Si on appelle deux fois de suite une fonction génératrice, les deux objets fonctions retournées sont donc des objets distincts, différents.

3.2. Exemple type.

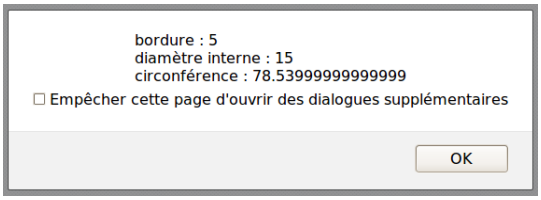
Le script :

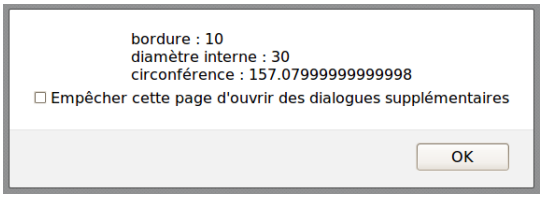
```
1. <script type="text/javascript">
2.      var fnGeneratrice = function (bordure) {
3.           var cstPi = 3.1416 ;
4.           var fnFille = function () {
5.                var diametre = parseInt(prompt("bordure : " + bordure +
6.                                          "\n Donnez le diamètre interne")) ;
7.                var fnCalcul = function () {
8.                       var diametreExterne = diametre + 2*bordure ;
9.                       return cstPi * diametreExterne ;
10.                  }
```

```
11.              alert ("bordure : "+ bordure +
12.                     "\ndiamètre interne : " + diametre +
13.                     "\ncirconférence : "    + fnCalcul())
14.              }
15.              return fnFille ;
16.         }
17./* génération des fonctions et test */
18.   var fnBordureCinq = fnGeneratrice (5) ;
19.// var fnBordureDix  = fnGeneratrice(10) ;
20.   fnBordureCinq() ;
21.// fnBordureDix();
22.</script>
23.<!-- h702_generateur.html -->
```

* la fonction génératrice engendre une fonction fille qui affiche la circonférence d'un cercle borduré. La largeur de la bordure est fixée à la génération, et ne peut être changée dans la fonction fille. La fonction fille saisit le diamètre interne du cercle (attention au **`parseInt()`** qui suppose une valeurs entière ; un **`parseFloat`**() aurait été possible), lui ajoute la bordure et affiche les résultats.

* voici les résultats des deux tests. Dans le premier test, la fonction est créée avec une bordure d'épaisseur 5 ; dans le second, la bordure a une épaisseur de 10.



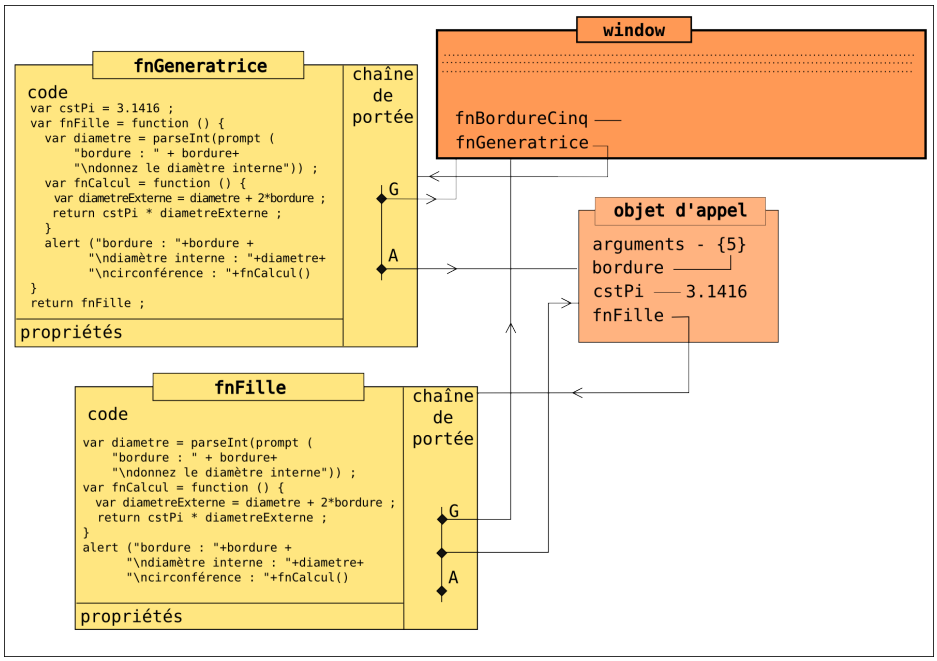


3.3. Exécution du script.

On propose de reprendre la symbolique de la section 2 et de l'appliquer à cet exemple. On ne reprend pas les premières étapes et on suppose que l'on est à la ligne 15, avant le **`return`**.

* **`fnGeneratrice`** a été appelée avec le paramètre égal à 5.

* La fonction **`fnFille`** est **créée, mais pas appelée** : elle a donc une chaîne de portée, mais pas d'objet d'appel.



fin de la ligne 18 : la fonction **fnGeneratrice** a effectué son retour :

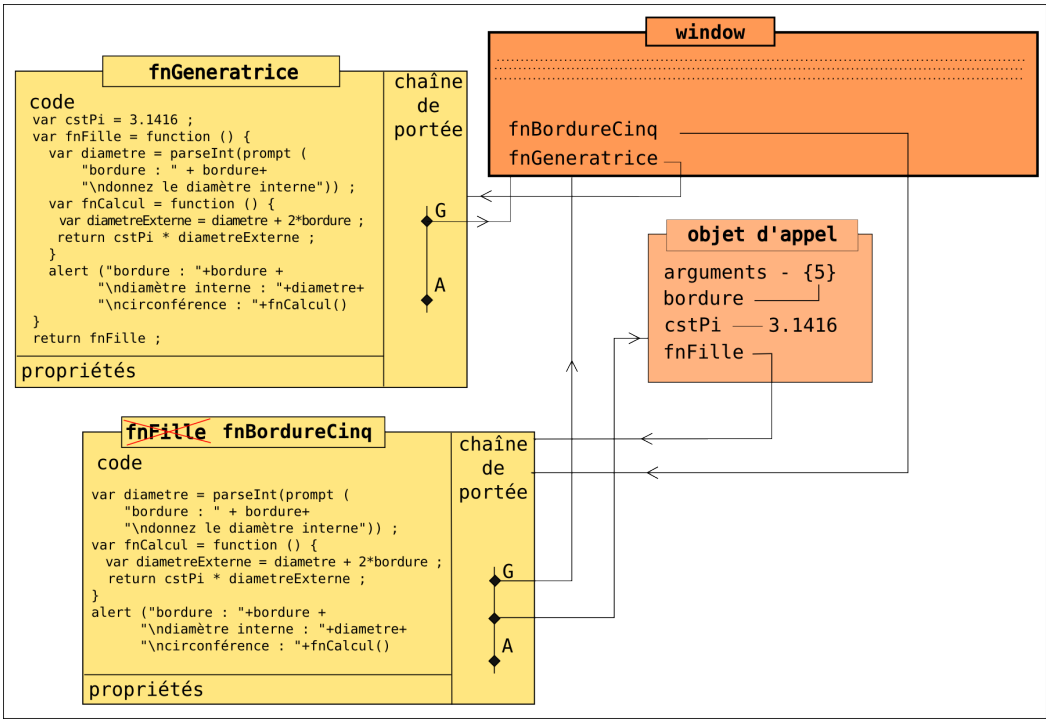
* le lien entre **fnGénératrice** et son objet d'appel a disparu (retour de fonction).

* un lien nouveau a été établi entre **fnBordureCinq** (dans **window**) et **fnFille** qui désormais peut être appelée depuis **window**, sous le nom de **fnBordureCinq**.

* il reste un lien sur l'objet **fnFille** (alias de **fnBordureCinq**) issu de **window** et un chemin pour arriver à l'objet d'appel de **fnGeneratrice**. Il n'y a donc aucune zone orpheline ! La fonction **fnGeneratrice** s'est terminée, mais ses variables persistent !

Désormais, la fonction **fnBordureCinq** a une composante supplémentaire : l'objet d'appel de sa fonction génératrice. Sa chaîne de portée a deux éléments permanents.

Une telle fonction qui, dans sa chaîne de portée, conserve toutes les références de la chaîne de portée de sa fonction génératrice, s'appelle une fermeture.



note : si la fonction **fnFille** avait été récursive, il n'y aurait eu aucun problème avec le changement de nom, puisque **fnBordureCinq** est un alias de **fnFille**.

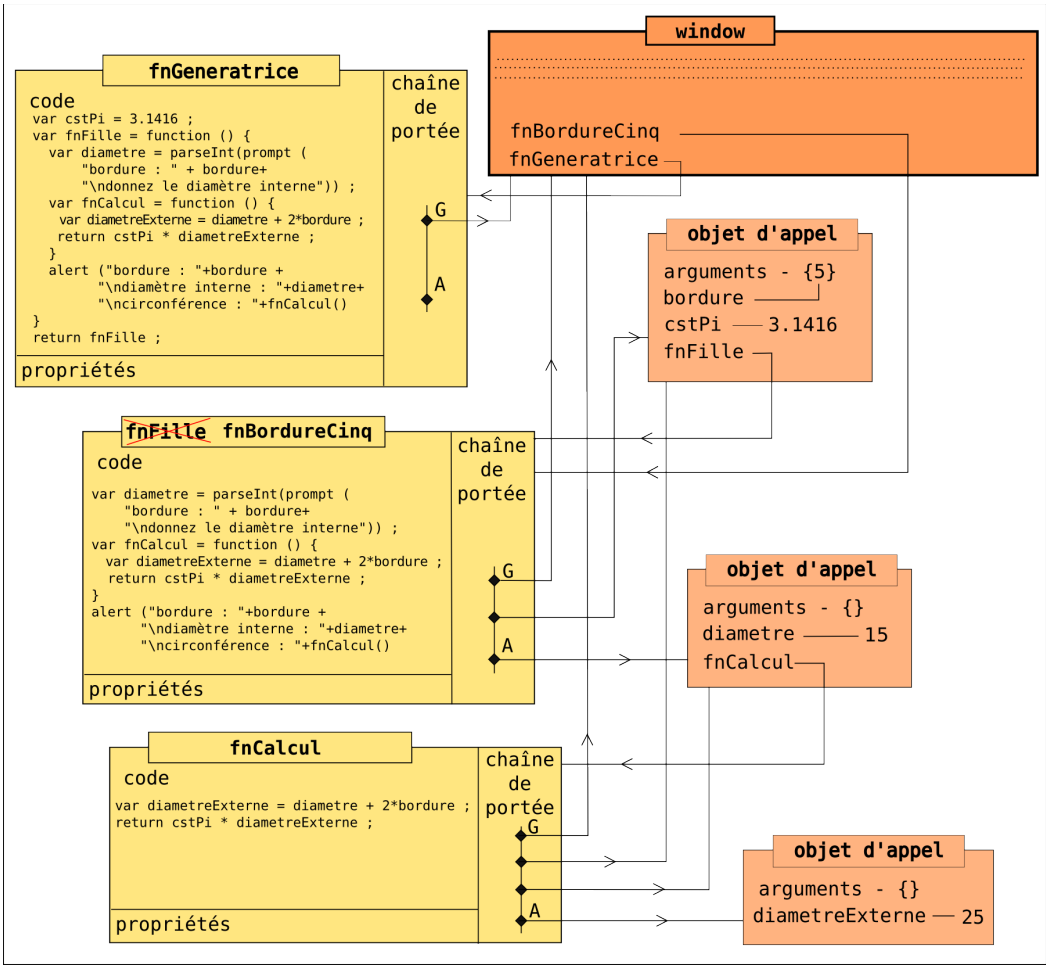
Il reste à tester la nouvelle fonction.

* pour la suite, on suppose qu'on a rentré la valeur 15 par la fonction **prompt()** ;

* le schéma suivant est relatif à la fin de la ligne 9 : la fonction englobée **fnCalcul** n'a pas effectué son retour.

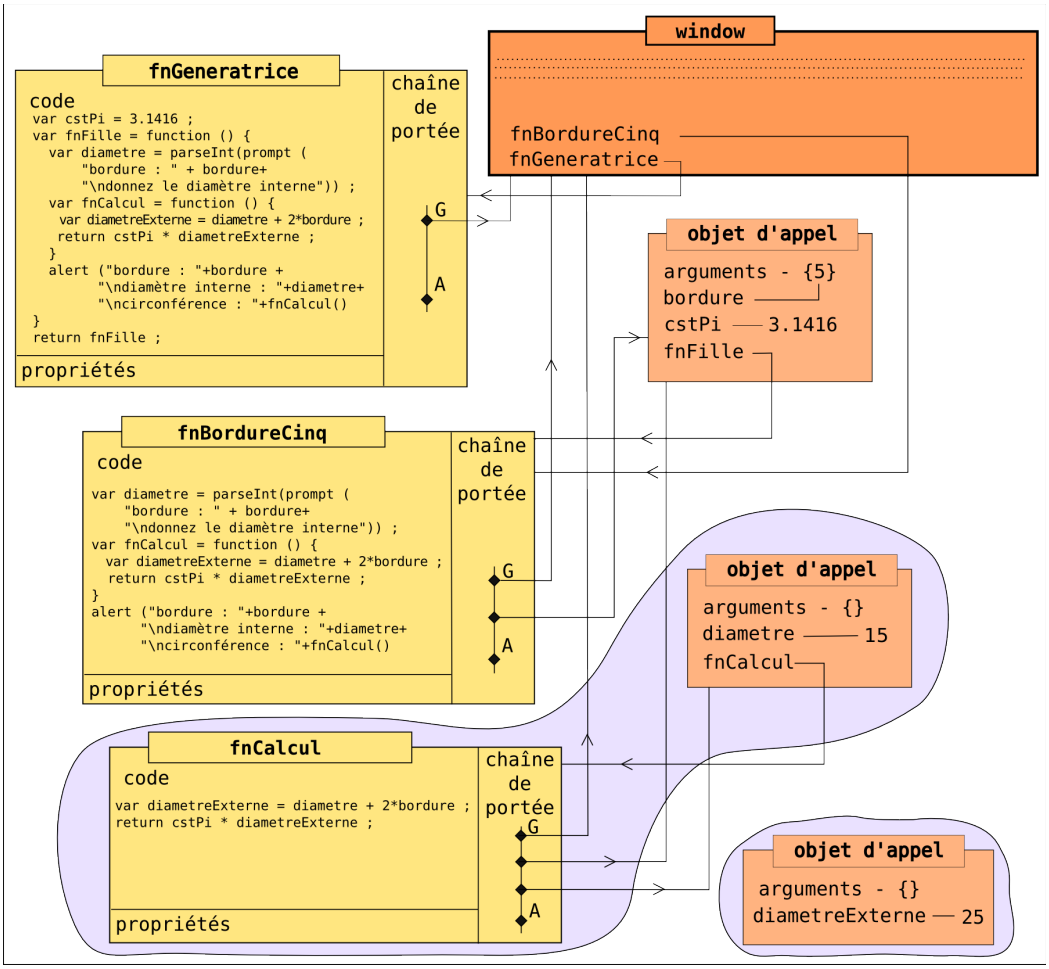
* la fonction **fnCalcul** a une chaîne de portée avec 4 objets d'appel...

Si une telle fonction avait été une fermeture, elle aurait comporté une chaîne de trois objets d'appels permanents.



Il reste à voir l'état des fonctions en fin de script

* les fonctions **`fnCalcul`** et **`fnBordureCinq`** ont effectués leur retour : il y a deux liens qui ont disparus sur les objets d'appels. Les zones orphelines apparaissent, que le ramasse-miette peut récupérer.



4.Variable privée, getter et setter.

4.1. Le problème.

En programmation objet une variable privée est une variable qui ne peut être ni changée, ni même consultée **directement** depuis un programme. Les seuls accès possibles se font par l'intermédiaire de méthodes :

- en lecture, par une méthode appelée un **getter**,
- en écriture par une méthode appelée un **setter**.

Les variables privées ont les gros avantages suivants :

* sécurisation des données : le **setter** peut contrôler les données à mémoriser, avant de le faire effectivement. On garantit ainsi l'intégrité des données car on passe nécessairement par le **setter**. Le **setter** peut aussi activer une exception si la donnée lue est incorrecte sans avoir perturbé la donnée réelle enregistrée.

Un **getter** assure de toujours restituer une donné au format désiré. Par exemple, un **getter** peut être donné comme ne fournissant que des entiers alors que la donnée interne peut être d'un autre format.

* du point de vue ingénierie, la nature des données effectivement stockées n'est pas connue du client, voire du programmeur (utilisation des librairies). On peut changer la structure de donnée sans avoir à reformuler tout le programme : seuls le **setter** et le **getter** doivent éventuellement être adaptés.

Rien de particulier n'est prévu en JavaScript, contrairement à Java ou Pascal. Mais rien n'empêche un tel traitement. **Par une fermeture** : il suffit de stocker les données effectives dans un objet d'appel, qui restera toujours invisible de l'extérieur de la fonction fermeture.

4.2. Un script pour le principe.

Le script :

On dispose d'un objet, **monObjet**, dont on veut qu'il mémorise une donnée **`valeurPrivee`**, mais qu'on ne veut pas en faire une propriété (elle serait affectable). Deux propriétés de **`monObjet`** vont servir de **`getter`** et de **`setter`**. Pour montrer les étapes, on a un peu «dilué» le code.

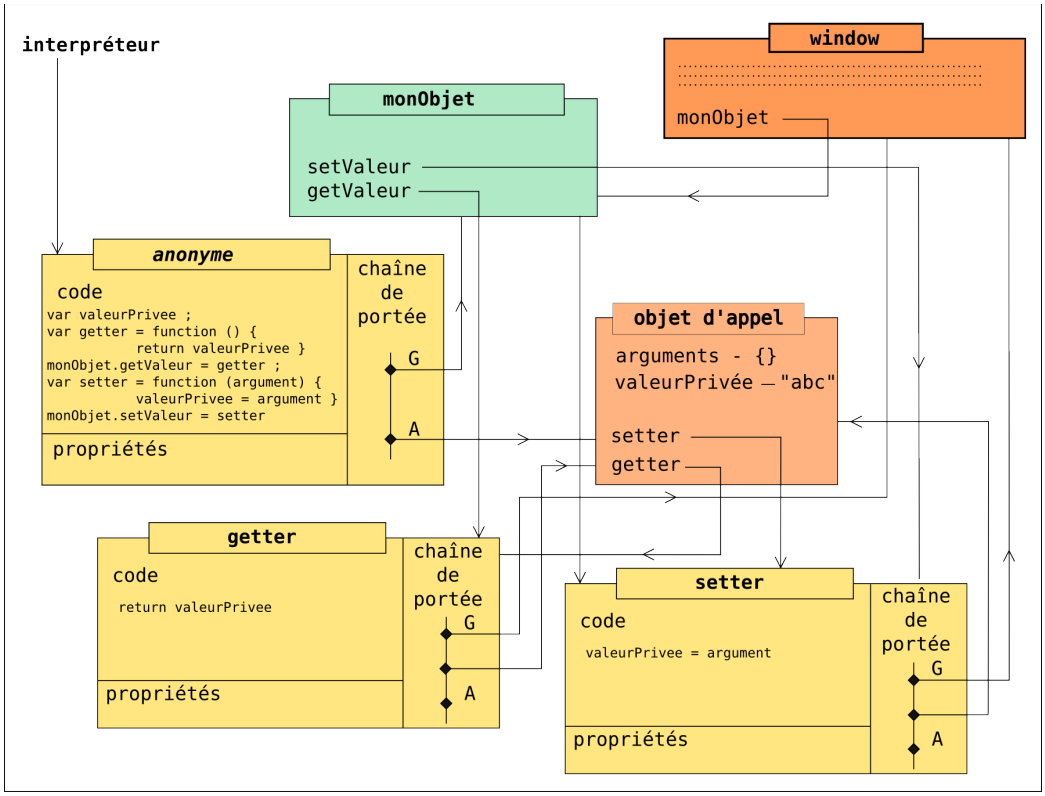
```
1. <script type="text/javascript">
2.     var monObjet = { "getValeur":null, "setValeur":null } ;
3.     (function () {
4.         var valeurPrivee ;
5.         var getter = function () { return valeurPrivee }
6.         monObjet.getValeur = getter ;
7.         var setter = function (argument) { valeurPrivee = argument }
8.         monObjet.setValeur = setter
9.      })();
10./* test */
11.    monObjet.setValeur (prompt("valeur à stoker ")) ;
12.    alert (monObjet.getValeur()) ;
13.</script>
14.<!-- h0703_get_set.html →
```

* On utilise une fonction anonyme dont deux variables sont des fonctions englobées.

Une fonction anonyme ne vit que le temps durant lequel l'interpréteur lui est liée. **Cela n'empêche pas la chaîne d'appel de lui survivre.**

Le schéma :

Le déroulement du traitement ne pose pas de problème particulier. Nous avons représenté ci-dessous **le dernier état de la fonction anonyme et le premier état après sa fin**, avant les tests, selon les conventions des sections qui précèdent.



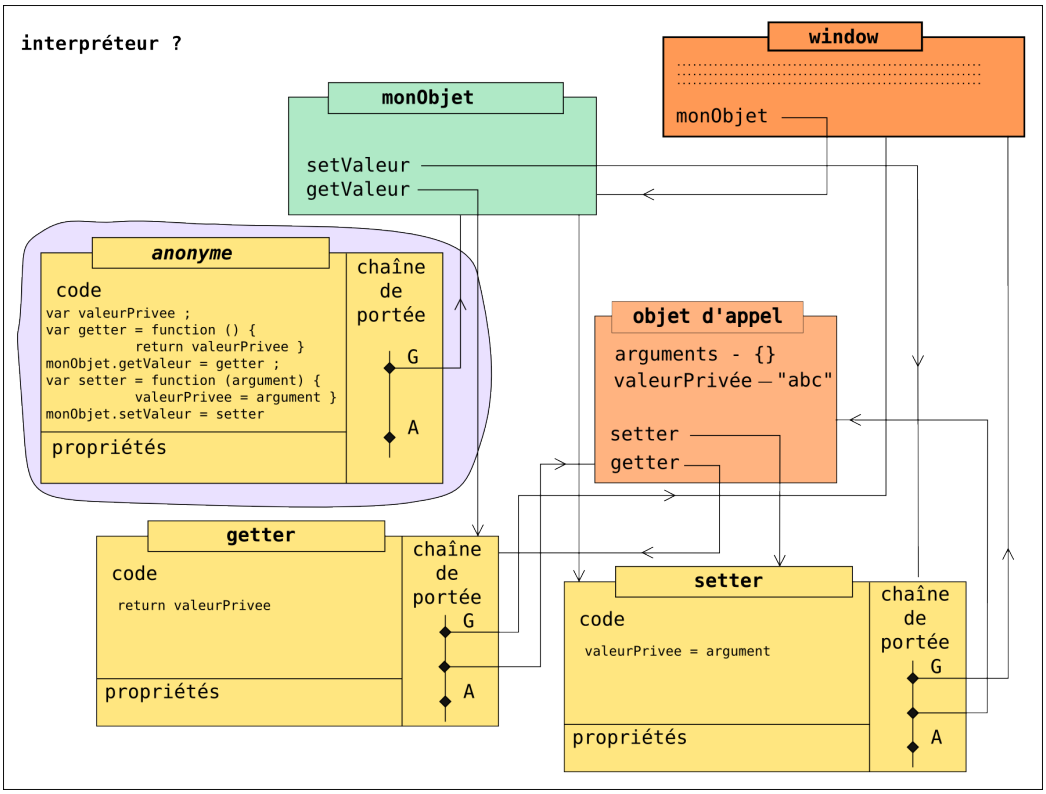
* la complexité du schéma est surtout liée au fait qu'il y a deux fonctions englobées.

* comme la fonction est anonyme on a figuré l'interpréteur, ce qui n'a pas été fait dans les sections précédentes.

On peut remarquer qu'un certain nombre d'éléments sont inutiles :

- l'objet **monObjet** aurait pu être vide, et simplement enrichi dans la fonction anonyme.
- les variables **setter** et **getter** sont inutiles : on aurait pu condenser le script en affectant directement les fonctions **setter** et **getter** en enrichissement de l'objet global.

Voici le second état où on a figuré la disparition de la fonction anonyme :



4.3. Un exemple plus réaliste.

Dans le script de principe, on n'a effectué aucun contrôle sur la valeur privée dans le **setter** ; ce n'est évidemment pas une pratique réaliste.

On propose d'écrire un script qui rajoute un **setter** et un **getter** à un objet quelconque, avec un contrôle sur le paramètre du **setter**. Le nom affecté à la donnée est également passé en paramètre.

Le contrôle consiste à vérifier que le paramètre réel du **setter** soit de type chaîne de caractères.

```
1.  <script type="text/javascript">
2.  var creerGetterSetter = function  (obj, nom, fnCondition){
3.      var valeurPrivee;   /* valeur stockée dans l'objet d'appel */
4.      obj["get"+nom] = function () { return valeurPrivee }
5.      obj["set"+nom] = function (valeur) {
6.          if (fnCondition && ! fnCondition(valeur)) {
7.               throw "valeur «"+valeur+"» invalide pour set"+nom;
8.          }
9.          else valeurPrivee = valeur
10.     }
11. }
12. /* test */
13. var monObjet = {} ;
14. creerGetterSetter ( monObjet,
```

```
                    "LaChaine",
                    function (x) {
                       return typeof x == "string" ; } );
monObjet.setLaChaine ("une chaîne sans ambigüité") ;
alert (monObjet.getLaChaine());
try {
    monObjet.setLaChaine (true);
}
catch (erreur) {
     alert (erreur)
}
</script>
<!-- h0704_gettersetter.html -->
```

07 : constructeurs

1. Le mot clef `this` dans une fonction.

1.1.Toute fonction est une propriété.

On a vu dans ce qui précède (chapitre **t06 variables**) qu'une fonction est un objet, et qu'elle est -presque- toujours aussi, la propriété d'un autre objet. Les fonctions globales, sont en réalité des propriétés de l'objet global **`window`**. Les fonctions englobées sont des propriété de la variable d'appel de leur fonction englobante ; leur utilisation ne se fait pas par qualification, puisque les objets d'appels, sauf **`window`**, sont anonymes.

Dans les cas rencontrés, soit elles sont **des propriétés d'un objet nommé**, et dans ce cas on emploie aussi le mot **méthode** (une fonction globale est une méthode de **`window`**), soit elles sont des propriétés de **l'objet d'appel** de la fonction où elles sont englobées.

Il existe une exception : les fonctions anonymes non liées à un identificateur : ces dernières ne sont ni des valeurs affectées à un identificateur, ni des paramètres réels d'une fonction. De telles fonctions sont "obligatoirement" **parenthésées** lorsqu'elles ne sont pas affectée à une variable, et sont immédiatement **appelées** ; elles sont donc exécutées *in situ*, là où elles sont définies. Sinon elles ne peuvent servir puisqu'on n'a aucun moyen de les évoquer !

exemple : **`monIdentificateur = (function (){ return "pageDeGarde" })() ;`**

Attention : ne pas utiliser **`this`** pour les fonctions imbriquées. Dans le cas des fonctions imbriquées, **`this`** est l'objet global ! L'erreur classique consiste à croire que le this d'une fonction imbriquée est celui de sa fonction englobante ; c'est évidemment faux ; si on veut utiliser le **`this`** de la fonction englobante, il faut définir un alias, variable locale de la fonction englobante (**`var aliasDeThis = this`**).

1.2. Les méthodes.

Ce qui suit ne concerne que les méthodes. La forme d'appel est donc
`monObjet.maMethode(`*`les paramètres`*`) ;`

Quand on définit une fonction, **on ne sait pas nécessairement quelle(s) méthode(s) elle deviendra**. Comment prendre en compte l'objet (propriétaire) dont elles ne seront qu'une des propriétés **quand on programme le bloc** d'une fonction ? Pour cela, on utilise le mot clef **`this`** : il désigne l'objet d'où la fonction pourra être appelée en tant que méthode. L'objet désigné par **this** peut être encore inexistant au moment ou l'on écrit ce mot clé dans la fonction. **Ce n'est qu'à l'exécution que sera connu le propriétaire de la méthode.**

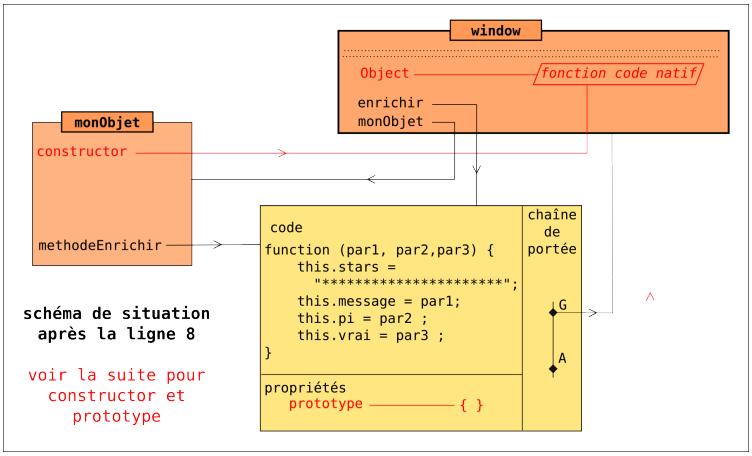
Voici un tel exemple :

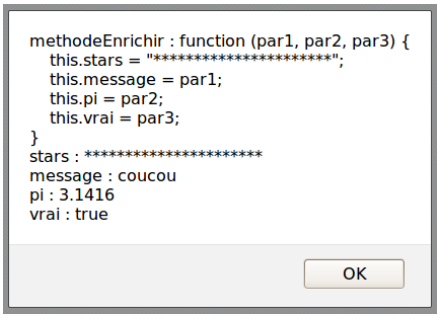
```
1.<script type="text/javascript">
2.     var enrichir = function (par1, par2,par3) {
3.         this.stars = "**********************";
4.         this.message = par1;
5.         this.pi = par2 ;
6.         this.vrai = par3 ;
7.     }
8.     var monObjet = {methodeEnrichir:enrichir} ;
9.     monObjet.methodeEnrichir("coucou",3.1416,true) ;
10.    chn = "";
11.    for (x in monObjet) {
12.        chn += x+" : "+monObjet[x]+"\n";
13.    }
14.    alert (chn) ;
15.</script>
16.<!-- h0800_this.html -->
```

this, dans **enrichir()** désigne l'objet courant, qui est ici, **window. Le code à partir de monObjet, sous le nom de methodeEnrichir,** il désigne **monObjet**. C'est toujours la même fonction, physiquement **le même objet. Et donc le même code** qui sera exécuté (il n'y a pas de duplication du code) après être devenue une méthode de **monObjet.**

On notera la profonde différence entre l'objet **this,** la chaîne de portée et les variables qui y sont définies. **this** n'est reconnu qu'à l'exécution de la méthode, alors que la chaîne de portée est définie à la création de l'objet (et donc indépendamment de l'utilisation qui sera faite de la fonction).

note : L'identificateur **this** est un mot clef, alors que **arguments**, propriété de l'objet d'appel, n'a pas ce statut et pourrait être redéfini (dangereusement !)





2. l'opérateur new.

2.1. Syntaxe pour l'opérateur new.

La syntaxe pour l'opérateur **new** est :

new *<expression appel de fonction>*

L'opérateur **new** suivi de son argument est une **expression**.

exemples : Voici deux expressions valides (**enrichir** : référence à l'exemple *h0800_this.html*)

```
new enrichir("coucou",3.1416,true)
new (function(){alert(param)})("coucou")
```

JavaScript admet une notation sans l'opérateur d'appel (les parenthèse) pour les fonctions invoquées **sans paramètre ; exemple** : **new enrichir** ;

2.2. l'expression new *argument* est un objet (object).

exemple de test :

```
1.<script type="text/javascript">
2.    alert (typeof new (function(){})());
3.    /* ******************************/
4.    var fn = function(param){alert(param)}
5.    var nouveau = new fn("coucou");
6.    alert (typeof nouveau);
7.</script>
8.<!-- h0801_new.html -->
```

Ce test fait apparaître le type **object** du résultat

2.3. La propriété constructor.

Appliquer l'opérateur **new** produit un objet, et ensuite exécute l'appel de fonction en argument. Cet

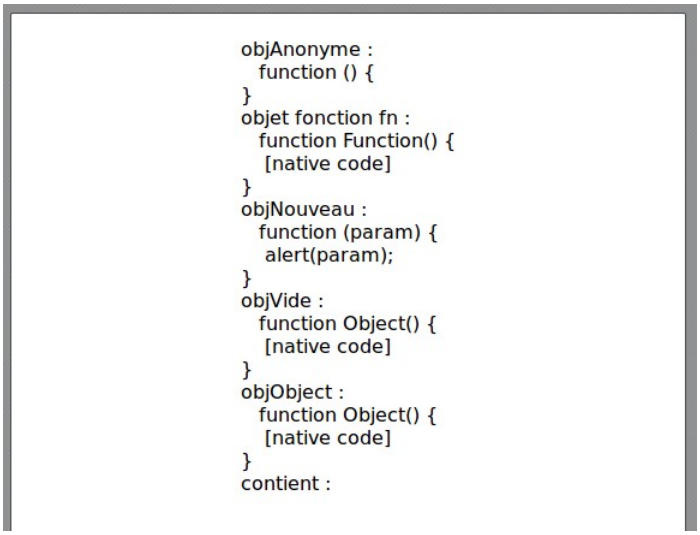
objet n'est pas vide ; comme tous les objets, il comporte une propriété prédéfinie, **`constructor.`** Cette propriété n'est pas énumérable. On peut donc l'utiliser mais elle n'apparaît pas dans une boucle **`for/in`**.

exemple test :

```
1. <script type="text/javascript">
2.     var objAnonyme = new (function(){})();
3.     /* ***************************** */
4.     var fn = function(param){alert(param)}
5.     var objNouveau = new fn("coucou");
6.     /* ***************************** */
7.     var objVide = {} ;
8.     /* ***************************** */
9.     var objObject = new Object() ;
10.    /* ***************************** */
11.    var chn = "" ;
12.    chn += "objAnonyme :\n   "      + objAnonyme.constructor + "\n";
13.    chn += "objet fonction fn :\n " + fn.constructor         + "\n";
14.    chn += "objNouveau :\n   "      + objNouveau.constructor + "\n";
15.    chn += "objVide :\n   "         + objVide.constructor    + "\n";
16.    chn += "objObject :\n   "       + objObject.constructor + "\ncontient : ";
17.    for (var x in objObject) chn += x +" \n\n";
18.    alert (chn);
19.</script>
20.<!-- h0802_constructor.html -->
```

On a ici cinq exemples de création d'objet. Trois avec **`new`** (ligne 2, ligne 5 et ligne 9) et deux par littéral (ligne 4 et ligne 7). Dans tous les cas, il y a bien un objet **`constructor`** qui, ici, est soit la fonction argument de **`new`**, soit la fonction prédéfinie **`Object`** dans le troisième cas. Noter que **`new Object()`** fournit également un objet vide.

Attention avec les objets "vides" : il y a autant d'objets différents que d'objets créés sans propriété explicitée. Si on teste **`({}=={})`** on obtient false !



2.4. L'appel de la fonction constructeur.

La fonction évoquée dans le paramètre de l'argument de **new** est appelé **une fonction constructeur**. Une telle fonction n'a pas de **`return`** car l'expression **`new argument`** prend alors comme valeur la valeur retournée. Si on a un **`return,`** l'objet créé par **`new`** est perdu (il n'y a donc pas d'erreur à le faire, mais cela n'a aucun intérêt).

Les fonctions wrapper : **`Object(), String(), Boolean()`** et **`Number()`** ont un fonctionnement différent des autres constructeurs et peuvent dans certaines configurations retourner une valeur ou être utilisées avec **new**. Cette spécificité sera exploitée dans les études particulières qui concernent les wrappers.

Lorsqu'une fonction constructeur est appelée ; s'il y a un objet **`this`** dans le corps de la fonction, le **`this`** est relatif au nouvel objet créé. On peut voir la conséquence de cette affirmation : le nouvel objet créé par **new** est le propriétaire provisoire de la fonction exécutée ensuite ; si dans le corps de fonction on ajoute une propriété à **`this`**, c'est en fait au nouvel objet que cette propriété est ajoutée. On là le moyen de fabriquer des objets complexes en une seule instruction.

Dans le cas ci-dessous, la propriété **`constructor`** est le **constructeur**. Mais, comme on le verra, **ce n'est pas la règle** lorsqu'on définit un héritage.

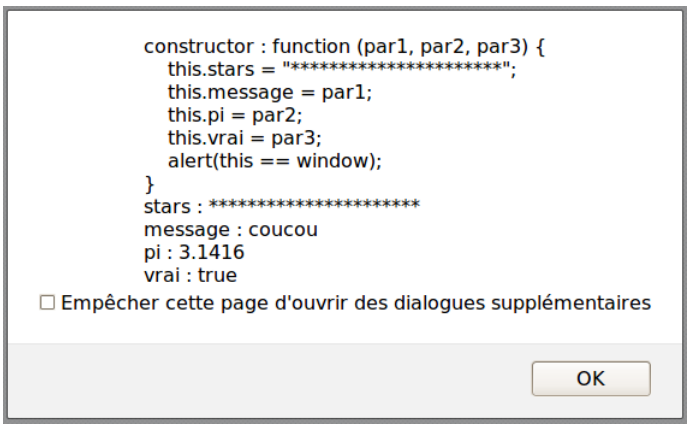
exemple de création d'un objet avec ses propriétés :

```
1. <script type="text/javascript">
2.     var enrichir = function (par1, par2,par3) {
3.         this.stars   = "**********************";
4.         this.message = par1;
5.         this.pi      = par2 ;
6.         this.vrai    = par3 ;
7.         alert (this == window + "  "+ monObject) ;
8.     }
9.     var monObjet = new enrichir("coucou",3.1416,true) ;
10.    chn = "constructor : " + monObjet.constructor + "\n";
11.    for (x in monObjet) {
12.        chn += x+" : "+monObjet[x]+"\n";
13.    }
14.    alert (chn) ;
15.</script>
16.<!-- h0803_this.html -->
```

* la ligne 7 : l'affichage est : **`false undefined`**.

Cela permet d'abord de tester que l'objet appelant n'est pas **`window`**. quand on exécute le code de la fonction avec l'opérateur **new**. Le test sur **this** permet cette vérification :

- l'objet est **d'abord** créé par **new** et il est **anonyme** ;
- **ensuite** le constructeur devient propriété de cet objet anonyme est il est appelé**.** **`this`** désigne alors un objet, mais comme il est anonyme, il n'est pas question de faire de test !
- **enfin** c'est cet objet qui est affecté à l'identificateur **`monObjet`**. Cet identificateur reste **`undefined`** tant que l'affectation n'a pas eu lieu, c'est à dire l'objet n'est pas terminé !



2.5. Questions de vocabulaire : constructeur, instance, classe.

La fonction qui sert à créer un objet s'appelle son constructeur.

Il y a ici une question de vocabulaire : le mot **constructeur** existe dans la programmation objet, mais recouvre un concept différent de celui du JavaScript. La difficulté conceptuelle pour bien faire la différence avec Java ou Python est d'autant plus grande que la syntaxe est très ressemblante. Mais il n'y a pas de classe en JavaScript !

Dans les langages objet avec des *classes*, un objet créé à partir d'une classe est appelé une *instance* de cette classe. Il est fréquent que les *constructeurs* soient appelés des *classes* et par dérivation, que les objets créés soient appelées des *instances de ces classes*. Ceci crée évidemment une situation ambiguë, puisque **les classes n'existent pas en JavaScript**. Nous n'utiliserons pas le mot *classe* pour désigner les constructeurs. Par contre, le mot *instance* évitera les circonlocutions : *instance du constructeur* … plutôt que *objet créé par le constructeur*... Évidemment parler de classe, de superclasse etc peut apparaître pratique ; mais le plus souvent, cela égare l'esprit en entraînant la pensée vers un modèle de classe qui n'est pas le bon modèle pour JavaScript.

3. Le prototypage.

3.1. La propriété `prototype`.

Toute fonction créée a une propriété **prédéfinie : `prototype`.** La propriété **`prototype`** n'est pas énumérable. Il faut faire attention au fait que **`prototype`** n'est pas un mot clef : on peut donc créer par accident**,** une propriété ayant comme nom **: `prototype`**, qui serait dénuée de la signification que l'on accorde en JavaScript à cet identificateur.

exemple :

```
1.<script type="text/javascript">
2.    var fnUn = function() {} ;
3.    alert (fnUn.prototype);
4.    /* et même !!!! */
5.    alert (function(){}.prototype) ;
6.</script>
7.<!-- h0804_prototype.html -->
```

Dans les deux cas, la fonction **`alert()`** affiche : [**`object Object]`**, car, lorsque **`alert()`** a comme argument un objet, la fonction **`alert`** affiche en premier lieu le type, en second son constructeur.

La propriété **`prototype`** est de type **`object`** dont le constructeur est **`Object()`**.

3.2. affecter un objet à la propriété `prototype` d'un constructeur.

exemple 1 :

```
1. <script type="text/javascript">
2.     /* un objet littéral */
3.     var monObjetProto = {
4.           prop1 : "une chaîne",
5.           prop2 : 3.1416,
6.           prop3 : false
7.     }
8.     /* une fonction au prototype «enrichi»  */
9.     var fnUn = function() {} ;
10.    fnUn.prototype = monObjetProto ;
11.    var monNouvelObjet = new fnUn() ;
12.    /* analyse */
13.    chn = "constructor : "+monNouvelObjet.constructor+"\n";
14.    for (var x in monNouvelObjet)
15.        chn += x+" : "+monNouvelObjet[x]+"\n" ;
16.    alert (chn );
17.</script>
```

```
18.<!-- h0805_prototype.html -->
```

* la fonction **fnUn** en tant que constructeur construit normalement un objet vide. On voit ici que les propriétés de **prototype** deviennent des propriétés de l'objet construit par **fnUn**.

* La fonction **constructor** et le **constructeur** ne sont plus les mêmes !

* On peut légitimement se poser les questions suivantes :

- l'objet construit est-il un alias de prototype ?

- quel lien unit le nouvel objet et son prototype ? C'est à dire **monNouvelObjet** et **fnUn.prototype**.

Pour le savoir, on va tester une modification du prototype et une modification de l'objet créé.

On rappelle que les deux appellations suivantes, **fnUn.prototype** et **monObjetProto** sont deux noms pour le même objet (alias).

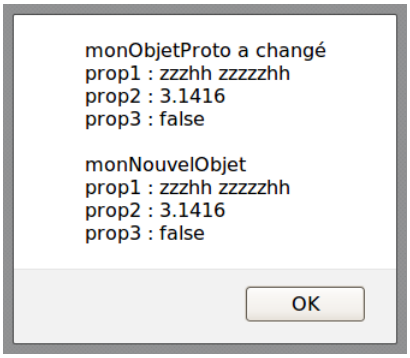
exemple 2 :

```
1. <script type="text/javascript">
2.     /* un objet littéral */
3.     var monObjetProto = {
4.         prop1 : "une chaîne",
5.         prop2 : 3.1416,
6.         prop3 : false
7.     }
8.     /* une fonction au prototype «enrichi»  */
9.     var fnUn = function() {} ;
10.    fnUn.prototype = monObjetProto ;
11.    var monNouvelObjet = new fnUn() ;
12.    /* analyse 1 on change le prototype */
13.    monObjetProto.prop1 = "zzzhh zzzzzhh" ;
14.    chn = "monObjetProto a changé\n";
15.    for (var x in monObjetProto)
16.        chn += x+" : "+monObjetProto[x]+"\n" ;
17.    chn += "\nmonNouvelObjet\n";
18.    for (var x in monNouvelObjet)
19.        chn += x+" : "+monNouvelObjet[x]+"\n" ;
20.    alert (chn );
21.</script>
22.<!-- h0806_prototype.html -->
```

* **on a changé le prototype** : la propriété prop1 a changé de valeur : ce changement se répercute sur l'objet créé.

La valeur initiale prise par une propriété est celle de son prototype.

* si on avait ajouté une propriété ou supprimé une propriété au prototype, la répercussion sur le nouvel objet aurait été immédiate.



exemple 3 :

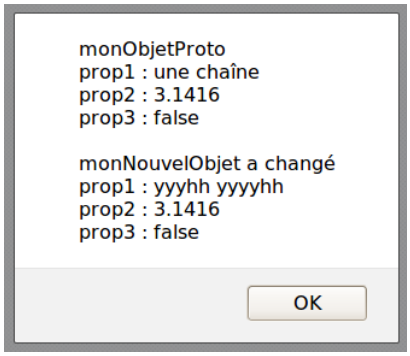
```
1. <script type="text/javascript">
2.     /* un objet littéral */
3.     var monObjetProto = {
4.         prop1 : "une chaîne",
5.         prop2 : 3.1416,
6.         prop3 : false
7.     }
8.     /* une fonction au prototype «enrichi»  */
9.     var fnUn = function() {} ;
10.    fnUn.prototype = monObjetProto ;
11.    var monNouvelObjet = new fnUn() ;

12.    /* analyse 2 on change le nouvel objet */
13.    monNouvelObjet.prop1 = "yyyhh yyyyhh" ;
14.    chn = "monObjetProto n";
15.    for (var x in monObjetProto)
16.        chn += x+" : "+monObjetProto[x]+"\n" ;
17.    chn += "\monNouvelObjet a changé\\n";
18.    for (var x in monNouvelObjet)
19.        chn += x+" : "+monNouvelObjet[x]+"\n" ;
20.    alert (chn );
21.</script>
22.<!-- h0807_prototype.html -->
```

* la propriété prop1 de l'objet créé a changé : cela n'a pas affecté la propriété du prototype.

Dit d'une autre manière : quand on croit modifier une propriété héritée, on crée en fait une nouvelle propriété qui occulte la propriété du prototype : il y a surcharge.

* que se serait-il passé si on avait, dans la foulée changé le prototype ?

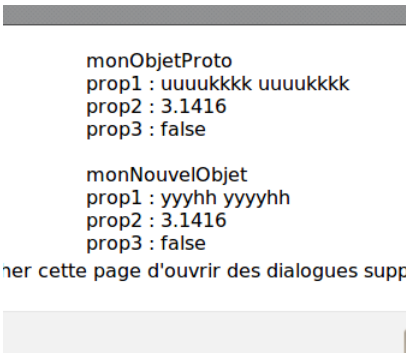


exemple 4 :

Il s'agit de l'exemple 3 où on a ajouté un nouveau changement de **`prop1`** dans le prototype ?

```
21.    monObjetProto.prop1 = "uuuukkkk uuuukkkk" ;
22.    chn = "monObjetProto \n";
23.    for (var x in monObjetProto)
24.        chn += x+" : "+monObjetProto[x]+"\n" ;
25.    chn += "\nmonNouvelObjet\n";
26.    for (var x in monNouvelObjet)
27.        chn += x+" : "+monNouvelObjet[x]+"\n" ;
28.    alert (chn );
29.</script>
30.<!-- h0808_prototype.html -->
```

* Le résultat peut paraître en contradiction avec celui de l'exemple 2 : la propriété **`prop1`** du prototype a changé et cela ne se retrouve pas sur **`monNouvelObjet`**. Question d'occultation de propriété !



3.3. la propriété `prototype` : propriété propres

Le processus selon lequel lors de sa création, un objet possède, **en lecture,** les propriétés de son prototype s'appelle l'**héritage**. Mais les propriétés héritées n'ont pas le même statut que les propriétés obtenues par enrichissement : les propriétés définies par enrichissement sont dites **propres (`own property`). Le** sens est à prendre comme dans "posséder en propre", posséder exclusivement, sans autre possibilité de revendication. Le fait d'être propre ou non ne change pas le fait d'être énumérable, c'est-à-dire détectées dans une boucle **`for/in`**.

Si une propriété héritée, qui n'est donc pas "propre", change de valeur dans une affectation (ou équivalent comme l'opérateur **`++`**) alors une propriété propriétaire est créée, de même nom et qui **occulte** la propriété du prototype, sans la changer.

On peut détecter le fait ou non d'être une propriété propre par la méthode prédéfinie **`hasOwnProperty`** qui retourne un booléen :

`monNouvelObjet.hasOwnProperty("prop1")` testé à la fin du dernier exemple retourne **`true`**.

3.4. héritage et règles d'usage.

L'héritage est un concept de la programmation objet ; dans les langages classiques comme C++, Java ou Python, l'héritage se fait à travers une hiérarchies de classes ; les classes n'existent pas en JavaScript. Mais **il existe une hiérarchie** de fait avec les prototypes : en effet les prototypes sont des objets qui ont eux aussi leur constructeur... Les prototypes sont chaînés par le système lui même.

Il est donc possible de définir **une hiérarchie des objets**, et chaque propriété définies dans un prototype est consultable dans sa descendance, sauf si elle est redéfinie (réaffectées) auquel cas, cette propriété redéfinie deviendra celle de sa descendance. On peut très bien créer une propriété "virtuelle" pour un objet (une fonction qui ne fait rien par exemple), et la redéfinir en l'adaptant aux caractéristiques de ses descendances. On dit que l'on **surcharge** la méthode. L'exemple le plus classique est celui de la méthode **`toString()`** de **`object`**, qui existe donc pour tout objet, et qui s'adapte pour donner une forme affichable plus conforme à la nature des objets étudiés. On reviendra sur ce concept en étudiant les constructeurs prédéfinis de JavaScript.

En tout état de cause, on peut voir, comme dans les autres langages objets, que le principal intérêt du processus est **l'héritage des méthodes** (propriété fonction). L'avantage est double : d'une part, on n'a pas à redéfinir à chaque création d'objets toutes les méthodes ; d'autre part, il y a un avantage au niveau des performances : en effet, il n'y a pas duplication du code hérité, ce qui allège d'autant le travail de la mémoire : une seule référence, celle de son constructeur suffit à se réserver l'accès à toute une bibliothèque !

Les spécialistes de la programmation **recommandent** donc de constituer les prototypes comme recueils de méthodes, et de ne pas y insérer de données autres que des méthodes, ou quelques constantes avérées comme les constantes mathématiques pi ou e... Si on définit les constructeurs comme des fermetures, on peut placer des données classiques (nombres, chaînes, booléens, objets sans code exécutables) non «visibles» autrement que par des setters ou des getters.

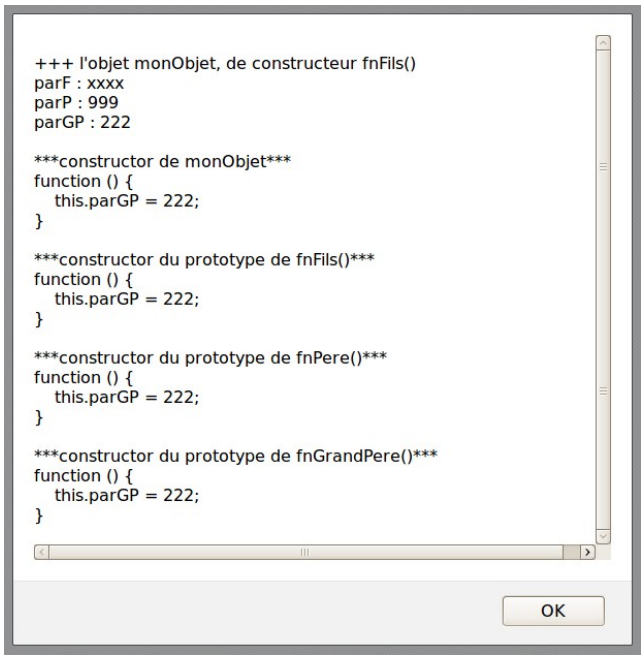
3.5. Retour sur la propriété `constructor`.

S'il y a une chaîne de prototypes explicites, leur propriété **`constructor`** est celle du **`protoype`** non explicité (en bout de chaîne). Celui-ci admet comme **`constructor`** le constructeur dont il est la

propriété. Par contre son constructeur est **`Object()`** ; cela implique que tous les objets héritent de l'objet engendré par **`Objects()`**. En voici une illustration :

```
1.<script type="text/javascript">
2.      var fnGrandPere = function () {
3.          this.parGP = 222 ;
4.      }
5.      var fnPere  = function () {
6.          this.parP = 999;
7.      }
8.      fnPere.prototype = new fnGrandPere() ;
9.      var fnFils = function () {
10.         this.parF ="xxxx" ;
11.     }
12.     fnFils.prototype = new fnPere () ;
13.     monObjet = new fnFils() ;
14.     chn = "+++ l'objet monObjet, de constructeur fnFils()\n";
15.     for (var x in monObjet) {
16.         chn += x + " : " + monObjet[x]+"\n" ;
17.     }
18.     chn  += "\n***constructor de monObjet***\n"+
19.      monObjet.constructor+"\n\n" ;
20.     chn  +=  "***constructor du prototype de fnFils()*** \n"+
21.     fnFils.prototype.constructor+"\n\n" ;
22.     chn  +=  "***constructor du prototype de fnPere()*** \n"+
23.     fnPere.prototype.constructor+"\n\n" ;
24.     chn  += "***constructor du prototype de fnGrandPere()*** \n"+
25.     fnGrandPere.prototype.constructor+"\n\n" ;
26.     alert (chn) ;
27.</script>
28.<!-- h0809_prototype_constructor.html -->
```

La propriété **`constructor`**, à la différence de **`prototype`**, n'influe pas sur le système. On peut donc l'utiliser librement, par exemple en lui donnant, dès la définition de l'objet la valeur du constructeur de l'objet (**`this.constructor = fnConstructeur`**) !



3.5. évocation de méthodes surchargées.

Il reste un problème classique lié à la surcharge. En général, on ne redéfinit pas complètement une méthode du prototype, mais on définit une méthode qui l'englobe : le problème est donc d'appeler dans la méthode redéfinie la méthode de même nom de son prototype, et qu'elle occulte.

```
1.<script>
2.     var fnPere  = function () {
3.         this.parP = 999;
4.         this.toString = function() { return "fnPere()\n" } ;
5.     }
6.     var fnFils = function () {
7.         this.parF ="xxxx" ;
8.         this.toString = function() {
9.              return fnFils.prototype.toString()+ "fnFils()"
10.        } ;
11.    }
12.    fnFils.prototype = new fnPere () ;
13.    var monObjet = new fnFils() ;
14.    alert (monObjet) ;
15.</script>
16.<!-- h0810__prototype.html -->
```

* la méthode **`toString()`** est une méthode qui existe pour tous les objets ; elle est définie dans **`Objects()`** qui est toujours le constructeur du dernier prototype de la chaîne des prototypes. Lorsqu'un objet **`xxx`** est pris comme argument de **`alert()`**, c'est en fait **`xxx.toString()`** qui est invoqué (même chose dans une concaténation : **`toString()`** transtype l'objet en chaîne). L'exemple actuel redéfinit deux fois cette méthode ; en plus, dans l'objet "fils", elle réutilise la méthode de son objet père.

08 : objets natifs, Object()

1. Objets prédéfinis.

1.1. Le noyau JavaScript.

On a vu que le langage JavaScript connaît deux grands types de données : les données primaires et les objets. Les données primaires permettent un fonctionnement très basique du langage et ne disposent pas des extensions minimales de tout langage évolué comme les tableaux.

Depuis les années 1990, tous les langages nouveaux ont utilisé la puissance modélisatrices des objets. Mais sans quelques moyens évolués inscrits en son cœur, aucun langage objet ne peut se développer ; en général, c'est à travers des bibliothèques (librairies, modules...) que ces langages se sont développés. JavaScript s'en distingue cependant :

- d'une part, il possède dans ses références d'un certain nombre d'objets, qui sont en quelle quelque sorte ses modules basiques, obligatoires, répondant à des normes (ECMAScript)

- d'autre part, JavaScript dispose toujours d'un objet global, dont toutes les propriétés ne sont pas inscrites dans les dites normes, et qui constitue la spécificité d'une implémentation : pour l'intégration dans les pages web, l'objet s'appelle **`window`**, et l'implémentation, **Javascript côté client**. Les tentatives de normalisation n'ont pas empêché l'existence de deux implémentations plus ou moins parallèles, celle de Microsoft et celle de Netscape.

1.2. Les objets de référence.

On peut classer les objets prédéfinis, correspondant donc à un **code natif**, en quatre catégories :

-les objets simples (**sans code exécutable**) :

* **`Math`** : ses propriétés permettent l'accès aux fonctions et données des mathématiques usuelles.

* **`arguments`** : joue un rôle particulier dans les objets d'appel des fonctions.

- les objets fonctions (**constructeurs**) à usage général :

* **`Function()`** : c'est le constructeur des objets «fonction».

* **`Object()`** : c'est le constructeur de base de tous les objets ; les propriétés qu'il définit appartiennent à tous les objets .

* **`Array()`** : implémente des objets qui ressemblent aux tableaux indexés des autres langages, avec une panoplie de propriétés spécifiques.

* **`Date()`** : permet de gérer finement les dates, données nécessaires dans les implémentations classiques du web.

* **`RegEx()`** : c'est un module qui donne accès aux expressions régulières, avec une approche voisine de celle du Perl, de Ruby et Python.

- les objets fonctions enveloppants (**wrapper**) :

* **`Boolean()`** : permet de créer des objets qui enveloppent des données primaires booléennes, et leur apporte des méthodes.

* **`Number()`** : même chose pour les nombres.

* **`String()`** ; même chose pour les chaînes de caractère.

* **`Object()`** a un fonctionnement wrapper, sans être réellement un wrapper.

- les objets fonctions pour la gestion d'**erreurs** :

* **`Error ()`** : il s'agit du constructeur générique des objets erreurs.

* **`TypeError()`**, **`EvalError()`**, **`RangeError()`**, **`ReferenceError()`**, **`URIError()`** sont des constructeurs qui génèrent des objets *erreurs* héritiers de ceux engendrés par **`Error()`**.

1.3. frameworks.

JavaScript ne connaît pas la notion de module ou de fichier inclus. La seule façon d'étendre le langage est de lui ajouter des objets nouveaux. Un framework peut se réduire à un seul objet nouveau, comme **JQuery**, voire simplement une fonction anonyme, qui ne peut donc être exécutée qu'une fois, lors de son adjonction à la création du script.

1.4. Objets de référence et prototypes.

Les objets de référence «fonctions» sont des **objets fonction** comme les autres : ils ont un **`prototype`** et les objets qu'ils engendrent ont une propriété **`constructor`**.

Il est déconseillé de modifier l'objet prototype d'une fonction constructeur de référence (§ 1.2.), car cela revient à modifier le cœur même de JavaScript ; sauf pour des implémentations auxquelles il manque manifestement une propriété importante il vaut mieux éviter de le faire.

La propriété **`constructor`** des objets créés les constructeurs de référence a pour valeur le constructeur ; les instances créés avec ces constructeurs**,** peuvent être détectés par leur propriété **`constructor`**. C'est l'une des principales utilisation de la propriété **`constructor`**.

Exemple.

```
1.<script>
2.    var maFonction = new Function("toto", "alert('')");
3.    var uneDate = new Date() ;
4.    var monObjet = {prop : "une chaîne"} ;
5.    alert (maFonction.constructor==Function) ; // true
6.    alert (monObjet.constructor==Object) ; // true
7.    alert (uneDate.constructor) ; // function Date() {[native code]}
8.</script>
9.<!-- h0900_constructor.html -->
```

2.Le constructeur Object().

2.1. Le constructeur.

Il y a deux syntaxes à la création d'objet :

`new Object()`

`Object(`*`arg`*`)` où *arg* est un **primaire** nombre, chaîne ou booléen

La première forme créée une instance vide, la seconde crée une instance wrapper qui sera étudiée ultérieurement.

rappel : le type des objets est **`object`**.

C'est le constructeur par défaut des objets simples littéraux ; c'est le constructeur ultime du **prototype** en bout d'une chaîne de prototypes.

2.2. les propriétés des instances de Object().

On rappelle que les propriétés des instances de Object() sont des propriétés de tous les objets.

*** hasOwnProperty() :** retourne un booléen ;

syntaxe : ***`objet`*`.hasOwnProperty(`*`chaîne de l'identifiant de propriété`*`)`**

`monObjet.hasOwnProperty("message")`

*** isPrototypeOf()** : retourne un booléen ;

syntaxe : ***`objet1`*`.isPrototypeOf(`*`objet2`*`)`**

*** propertyIsEnumerable()** : retourne un booléen ;

`monObjet.propertyIsEnumerable("toString")` retourne **`false`**.

On rappelle que l'objet global est **`window`**. Les objets de référence sont donc des propriétés de **`window`** ; mais elles ne sont pas énumérables ; par contre ce sont des propriétés dont **`window`** est propriétaire. Tester :

`window.hasOwnProperty("Function")` → `true` ;
`window.propertyIsEnumerable("Function")` → `false`

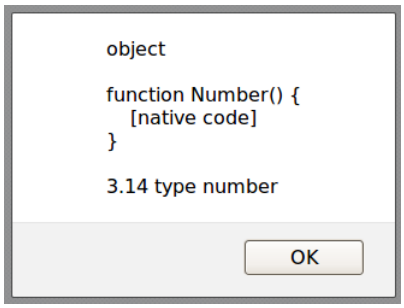
*** toString() :** retourne une représentation littérale de l'objet. Cette méthode est presque toujours redéfinie pour apporter quelque chose d'utile. C'est la fonction qui et appelées **automatiquement** lorsque une chaîne est attendue (**`alert()`**, concaténation).

*** toLocaleString() :** identique à **`toString()`** sauf si elle est surchargée, en fonction des constantes locales (temps, monnaie, écriture des nombres).

*** valueOf() :** retourne l'objet lui-même sauf si l'objet (wrapper) a une valeur primaire associée. Cette fonction est rarement appelée, sauf par le système lui-même.

C'est un peu l'inverse de **`Object(arg)`**.

```
1.<script type="text/javascript">
2.    var numObj = Object(3.14) ;
3.    var numPrim = numObj.valueOf() ;
4.    alert ((typeof numObj) + "\n\n" +
5.        numObj.constructor + "\n\n" +
6.        numPrim + " type " + typeof numPrim) ;
7.</script>
8.<!-- h0901_valueof.html -->
```



3. le constructeur Function().

3.1. Le constructeur et le littéral.

Utiliser le constructeur **`Function()`** est rigoureusement équivalent à définir une fonction anonyme.

La syntaxe seule diffère :

fonction anonyme : `function (liste des paramètres formels) { code }`

constructeur : `new Function ( liste des paramètres formels, code)`

La liste des paramètres formel et le code sont **des chaînes de caractère** ; le séparateur est la virgule. Les paramètres formels sont facultatifs.

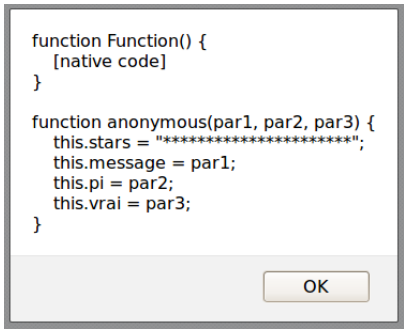
Si les arguments ne sont pas corrects, l'analyseur lève une erreur **`SyntaxError`**.

exemple :

```
1.<script type="text/javascript">
2.    var leCode = "this.stars = '**********';"+
3.       "this.message = par1; this.pi = par2 ; this.vrai = par3"
4.    var enrichir = new Function ("par1", "par2","par3",leCode) ;
5.    var monObjet = new enrichir ("une chaîne", 3.1416, true) ;
6.    alert (enrichir.constructor +"\n\n" + enrichir) ;
7.    alert (enrichir.length) ;
8.</script>
9.<!-- h0901_function.html -->
```

* cet exemple reprend le thème de la création d'un constructeur. On remarque cependant que les paramètres formels sont des chaînes de caractère ainsi que le code ; il vaut mieux ne pas oublier les point-virgules de fin d'instruction.

* La structure interne ne dépend pas de la méthode choisie, comme on peut s'en rendre compte sur l'affichage de constructor (ne pas oublier cependant qu'il y a un **`toString()`** pour réaliser l'affichage final.



3.2. les propriétés

*** length :** nombre d'arguments formels (ne pas confondre avec la propriété du même nom de la propriété **`arguments`** de l'objet d'appel, qui donne le nombre de paramètres réels).

Voir la ligne 7 de l'exemple précédent, qui affiche la valeur 3.

*** prototype** : déjà étudié.

*** call() : `fonction.call(objet, par1, par2...)`** équivaut au code suivant :

```
objet.provisoire = fonction ;
objet.provisoire (par1, par2...) ;
delete objet.provisoire ;
```

Cette méthode permet d'appeler une méthode sur un objet qui n'a pas été prévu. Il s'agit d'un changement d'objet appelant. L'objet peut être **`null`**, auquel cas la fonction est globale (méthode de **`window`**) ; ce peut très bien être l'objet **`this`**.

Lorsque l'appel est illicite, **`call()`** provoque une exception **`TypeError`**.

*** apply() :** même chose, la liste des paramètre étant remplacée par un tableau.

*** toString() :** la propriété est surchargée ; affiche tout le code sous la forme fonctionnelle littérale. Cf. l'exemple précédent.

09 : tableaux

1. les tableaux en interne.

1.1. Les indices.

On a l'habitude des tableaux dans les langages comme Basic, C, Pascal, Java. Ce sont des données structurées, dont les éléments sont ordonnés et peuvent être atteints par leur indice, qui numérote la position de l'élément dans le tableau. Le tableau a donc un nombre prédéfini de constituants, la longueur du tableau, où les constituants sont (souvent) de même type. La numérotation commence par 0. Affecter ou tenter de lire un élément hors tableau conduit à une erreur (**`RangeError`**). Tous les éléments du tableau sont sensés être initialisés à une valeur idoine.

Il vaut mieux éviter de penser à ce modèle si on veut comprendre les objets JavaScript de constructeur **`Array()`**, appelés **tableaux**. Le tableau est un type d'objet assez ordinaire, mais disposant d'un ensemble de méthodes prédéfinies. Les identifiants du tableau sont des chaînes qui représentent des entiers décimaux positifs, et donc du genre : "0", "1", "2" etc.

L'entier représenté ne peut être négatif (ex : "-2" n'est pas une propriété de tableau) ni trop grand (la limite est "4294967295" soit 2^{32}-1). Une erreur de type **`RangeError`** est produite si on essaie de donner une taille non convenable.

On sait qu'un nombre ne peut être un identificateur ; en toute rigueur, il faudrait désigner la valeur d'un élément de tableau sous la forme : **`monTableau["45"]`** ou **`monTableau['45']`**, soit **`monTableau[`*`expression chaîne de chiffres décimaux`*`]`**. Mais si on écrit un nombre à la place, l'analyseur syntaxique très accommodant de JavaScript effectue par le transtypage. On écrit donc **`monTableau[45]`** et alors, on retombe sur la syntaxe (trompeuse) de tous les langages habituels. Le décimal **`45`** est appelé l'**index** ou l'**indice** du tableau. Le tableau est une structure **indexée**. Il s'agit d'un **vocabulaire abusif,** mais sans grand danger et bien pratique.

1.2. Longueur d'un tableau.

La longueur d'un tableau donnée par le système est égale à sont **indice le plus grand plus un**. Elle est connue comme la propriété **`length`** du tableau. Un tableau vide a la longueur 0. La longueur égale au nombre de ses éléments que si le tableau n'est pas **creux**, c'est-à-dire qu'**il utilise tous les indices** entre 0 et la longueur moins un.

La longueur d'un tableau est une donnée valable à un instant donné, pas une caractéristique de l'objet : il suffit de définir une propriété d'indice supérieur et celle-ci devient la base de calcul de la longueur.

1.3. Le constructeur Array().

Les tableaux sont des objets de constructeur **`Array()`**. On peut très bien imiter la structure de tableau sans que l'on obtienne un tableau : on verra qu'un objet **JQuery** a tous les aspects d'un tableau sans en être un ; on a déjà rencontré l'objet **`arguments`** qui se traite comme un tableau, mais n'en a pas les méthodes.

Il y a quatre manières de définir une objet de constructeur **`Array()`** (qui est aussi le nom du **`constructor`** de ces objets).

`new Array()` : crée un tableau vide ;

`new Array(`*`taille`*`)` : taille est la valeur qui sera retournée comme longueur provisoire du tableau ;

Aucun élément n'est défini. Noter que l'on peut réinitialiser **`length`** sans rien changer au tableau.

`new (`*`suite d'éléments`*`)` : crée un tableau continu, initialisé avec la suite des éléments séparés par des virgules, donnée en argument. Attention : **il faut au moins deux éléments**, sinon JavaScript essaie de transcoder l'argument unique en nombre pour appliquer la syntaxe **`new Array(`*`taille`*`).`**

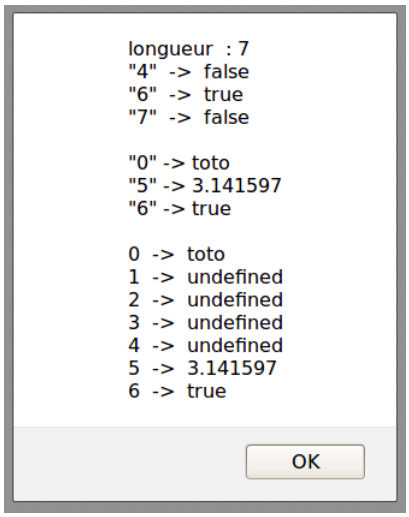
La quatrième méthode est littérale et utilise les crochets (notation Json) :

`[suite d'éléments]`

note : une méthode d'instance **`split()`** de **`String`** permet de transformer une chaîne de caractères dotée d'un séparateur en tableau : **`"a,b,c,d".split(',')`** retourne **`["a","b","c","d"]`**

exemple 1 :

```
1. <script>
2.     var monTableau = new Array() ;
3.     monTableau[0] = "toto" ;
4.     monTableau[5] = 3.1415926 ;
5.     monTableau[6] = true ;
6.     chn = "longueur  : " + monTableau.length + "\n";
7.     chn += '"4"  ->  ' +("4" in monTableau) + "\n";
8.     chn += '"6"  ->  ' +("6" in monTableau) + "\n";
9.     chn += '"7"  ->  ' +("7" in monTableau) + "\n\n";
10.    for (var x in monTableau) {
11.            chn+='"'+ x +'"'+' -> '+ monTableau[x]+"\n" ;
12.    }
13.    chn += "\n";
14.    for (var i = 0 ; i<7 ; i++) {
15.         chn+= i +'  ->  '+ monTableau[i]+"\n" ;
16.    }
17.    alert (chn) ;
18.</script>
19.<!-- h1000.html -->
```



exemple 2 :

```
1. <script>
2.     var monTableau = new Array(4) ;
3.     alert("0" in monTableau) ;
4.     monTableau [7] = "n'importe quoi" ;
5.     alert (monTableau.length) ;
6. </script>
7. <!-- h1001_array.html -->
```

* ligne 3 : montre qu'aucune propriété n'est créée.
* ligne 4 : on crée un élément hors intervalle [0 ; 3] : la longueur devient 8 !

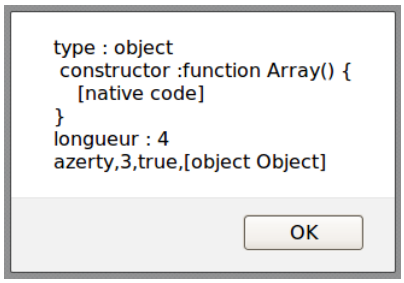
exemple 3 :

```
1.<script>
2.    var monTableau = ["azerty", 3, true, {x:1, y:2}] ;
3.    chn = "type : "+(typeof monTableau)+"\n" ;
```

```
4.    chn += " constructor :" + monTableau.constructor +"\n";
5.    chn += "longueur : "+ monTableau.length +"\n";
6.    alert (chn) ;
7.</script>
8.<!-- h1002_array.html -->
```

* lignes 3 et 4 : **type** et **constructor** du tableau littéral.

* ligne 5 : la méthode **toString()** surchargée des tableaux.



2. Les méthodes des instances de Array().

2.1. concat() :

syntaxe : ***tableau*.concat (*élément1, élément2 …*)** ;

ajoute au bout du tableau des propriétés de valeurs ***élément1, élément 2*** etc . Si l'un des élément est un tableau, ses éléments constitutifs deviennent autant de propriété du tableau.

exemple :

```
1. <script>
2.     var monTableau =[].concat ("azert",1,true,[9,8,[7,6]]) ;
3.     chn ="";
4.     for (var x in monTableau) {
5.          var y = monTableau[x] ;
6.          var t = typeof y ;
7.          if (t == "object") t += ' '+y.constructor;
8.          chn += x + "  -> "+ y + " " + t + "\n" ;
9.     }
10.    alert (chn) ;
11.</script>
12.<!-- h1003_concat.html -->
```

```
0 -> azert string
1 -> 1 number
2 -> true boolean
3 -> 9 number
4 -> 8 number
5 -> 7,6 object function Array() { [native code] }
```

Autrement dit : **["azert",1,true,9,8,[7,6]]]**

2.2. join() :

syntaxe : ***tableau*.join(*chaîne séparatrice*)**

retourne une chaîne, les éléments étant séparés par la chaîne séparatrice ; par défaut c'est la virgule si l'argument est omis.

exemple :

```
1.<script>
2.    var monTableau =["azert",1,true,[45,89,123]] ;
```

```
3.    alert (monTableau.join(" / ")) ;
4.</script>
5.<!-- h1004_join.html -->
```

affiche : **"azert / 1 / true / 45,89,123"**

2.3. pop() :

syntaxe : ***tableau*.pop()**

Supprime le dernier élément du tableau ; la longueur diminue de une unité.
Retourne l'élément supprimé. Fonctionne comme un dépilage.

exemple :

```
1.<script>
2.    var monTableau =["azert",1,4589123,3.1416] ;
3.    alert ("retour : " +monTableau.pop()+
4.           "\ntableau : "+monTableau+
5.           "\nlongueur : "+monTableau.length) ;
6.</script>
7.<!-- h1005_pop.html -->
```

```
retour : 3.1416
tableau : azert,1,4589123
longueur : 3
```

2.4. push() :

syntaxe : ***tableau*.push(*valeur1, valeur2, …*)**

Ajoute les valeurs au tableau (empilage) ; retourne la nouvelle longueur.

```
1.<script>
2.    var monTableau =["azert", 1, 4589123] ;
3.    var l = monTableau.push("a","b","c") ;
4.    alert ("tableau : " +monTableau + "\nretour : "+l) ;
5.</script>
6.<!-- h1006_push.html -->
```

```
tableau : azert,1,4589123,a,b,c
retour : 6
```

2.5. reverse() :

syntaxe : ***tableau*.reverse()**

Retourne le tableau **sur place** ; s'il y a plusieurs références au tableau, l'inversion est évidemment générale. Retourne le tableau inversé.

exemple :

```
1.<script>
2.    var monTableau = [1 ,2, 3, 4, 5] ;
3.    var alias = monTableau ;
4.    var ret = alias.reverse() ;
5.    alert (ret +"\n"+monTableau) ;
6.<!-- h1007_reverse.html -->
```

```
5,4,3,2,1
5,4,3,2,1
```

2.6. shift() :

syntaxe : ***tableau*.shift()**

Retourne le premier élément ; décale les éléments en les avançant de un rang ; ajuste la longueur.

exemple :

```
1.<script>
2.     alert = console.log ;
3.     var monTableau = [1 ,2, 3, 4, 5] ;
4.     var ret = monTableau.shift() ;
5.     alert ("retour : " + ret +"\n"+
6.            "tableau : [" + monTableau+"]\n"+
7.            "longueur : "+monTableau.length) ;
8.</script>
9.<!-- h1008_shift.html -->
```

```
retour : 1
tableau : [2,3,4,5]
longueur : 4
```

2.7.slice() :

syntaxe : ***tableau*.slice(*début, fin*)**

Retourne une tranche du tableau, début et fin désignent l'offset du début (compris) et de la fin **(non comprise)**. S'ils sont positifs, c'est l'indice (0, 1, 2..) ; négatifs, ils sont comptés. à partir de la fin (-1, -2, -3...). La tranche définie va dans le sens des indices croissants. **Le tableau est invariant**.

fin est optionnel ; en cas d'absence, c'est la fin du tableau qui est prise.

exemple :

```
1.<script>
2.     var monTableau = [0, 1 ,2, 3, 4, 5, 6, 7, 8 ,9] ;
3.     var positifs = monTableau.slice (3,  7) ;
4.     var negatifs = monTableau.slice(-7, -3) ;
5.     var lesDeux  = monTableau.slice( 3, -3) ;
6.     alert ("tableau : ["+monTableau+"]\n" +
7.            "slice ( 3,  7) : [" + positifs +"]\n"+
8.            "slice (-7, -3) : [" + negatifs +"]\n"+
9.            "slice ( 3, -3) : [" + lesDeux +"]\n"+
10.           "tableau : [" + monTableau+"]\n" ) ;
11.</script>
12.<!-- h1009_shift.html -->
```

```
tableau : [0,1,2,3,4,5,6,7,8,9]
slice (3, 7) : [3,4,5,6]
slice (-7, -3) : [3,4,5,6]
slice (3, -3) : [3,4,5,6]
tableau : [0,1,2,3,4,5,6,7,8,9]
```

2.8. sort() :

syntaxe : ***tableau*.sort()**
***tableau*.sort(*fonction de tri*)**

Trie le tableau **sur place**. Retourne le tableau trié.

La fonction de tri est de la forme **fn(a,b)** ;elle retourne 0 si a et b sont équivalents, un nombre négatif si a doit apparaître avant b, un nombre positif dans le cas contraire.

exemple 1 :

```
1.<script>
2.    var monTableau = [0, 3, 5, 7, 4, 1, 6, 2, 9 ,8] ;
3.    var alias = monTableau ;
4.    var ret = monTableau.sort() ;
5.    alert ("tableau : ["+ monTableau + "]\n" +
6.           "retour : [" + ret        + "]\n" +
7.           "alias : ["  + alias      + "]\n") ;
8.</script>
9.<!-- h1010_sort.html -->
```

```
tableau : [0,1,2,3,4,5,6,7,8,9]
retour : [0,1,2,3,4,5,6,7,8,9]
alias : [0,1,2,3,4,5,6,7,8,9]
```

exemple 2 :

Il s'agit de classer les valeurs décimales selon leur nombre de chiffres

```
1. <script>
2.     var monTableau = [26, 489, 147, 7777, 45, 1, 66, 2222, 1119] ;
3.     var fn = function (a,b) {
4.            var las = (a.toString()).length ;
5.            var lbs = (b.toString()).length ;
6.            if (las==lbs) return 0 ;
7.            else if(las < lbs) return -1 ;
8.            else return 1 ;
9.        }
10.    alert ("avant : ["  + monTableau          +"]\n"+
11.           "retour : [" + monTableau.sort(fn) +"]\n"+
12.           "après : ["  + monTableau          +"]") ;
13.</script>
14.<!-- h1011_sort.html -->
```

```
avant : [26,489,147,7777,45,1,66,2222,1119]
retour : [1,26,45,66,489,147,7777,2222,1119]
après : [1,26,45,66,489,147,7777,2222,1119]
```

2.9. splice() :

syntaxe : ***tableau*.splice (*début, nombre, valeur1, valeur2...*)**

Insère, supprime ou remplace des éléments d'un tableau. **Travaille directement sur le tableau**.

début désigne le numéro du début de l'opération ;

nombre désigne le nombre d'éléments à supprimer ; optionnel, et si non spécifié, la suppression va jusque la **fin** du tableau. Zéro s'il n'y a aucun élément à supprimer.

valeur1, valeur2 … valeurs à insérer à partir de **début**.

Retourne un tableau avec les éléments supprimés ou insérés.

```
1.<script>
2.    var monTableau = [26, 489, 147, 7777, 45, 1, 66, 2222, 1119] ;
3.    var copie =  monTableau.join(",").split(",") ;
```

```
4.    var ret = monTableau.splice (2,5,"***","+++") ;
5.    alert ("avant : [" + copie +"]\n" +
6.           "retour : [" + ret + "]\n"+
7.           "après : [" + monTableau+"]") ;
8.</script>
9.<!-- h1013_splice.html -->
```

```
avant : [26,489,147,7777,45,1,66,2222,1119]
retour : [147,7777,45,1,66]
après : [26,489,***,+++,2222,1119]
```

2.10. toString, toLocaleString() :

Retournent la conversion d'un tableau en chaîne de caractère.
Chaque élément du tableau est converti en chaîne puis retourné en une liste dont la virgule est le séparateur ; c'est la même chose que **`join()`** sans argument. **`toString()`** est utilisé dans tous les transtypages automatiques (ex : **`alert (monTableau)`**).

`toLocaleString()` : si des objets du tableau on une représentation "locale", comme les dates, cette fonction est préférable.

2.11. unshift() :

attention : tout en minuscules !

syntaxe : ***`tableau`*`.unshift (`*`valeur1, valeur2, …`*`)`**

Insère les valeurs en début de tableau, alors que **`push()`** le fait en fin de tableau. **`unshift()`** et **`shift()`** fonctionnent comme **`push()`** et **`pop()`** ; les premiers travaillent en début de tableau et autres en fin de tableau.

Les valeurs ont les indices 0, 1, 2 … dans l'ordre la liste. **Le tableau est modifié directement**.

Retourne le **nombre d'éléments insérés**.

10 : utilitaires

1. L'objet `Math`.

1.1. utilisation de l'objet `Math`.

L'objet **`Math`** est un conteneur, pas un constructeur. Il s'utilise donc pour les fonctions et constantes utiles auxquelles il permet d'accéder, pas pour créer de nouveaux objets **`: Math.PI, Math.log()`**

1.2. Les constantes.

Les constantes sont de type **`number`**.

`* Math.E 2.`	**`718281828459045`**	e
`* Math.LN10`	**`2.302585092994046`**	logarithme népérien de 10
`* Math.Ln2`	**`0.6931471805599453`**	logarithme népérien de 2
`* Math.LOG10E`	**`0.4342944819032518`**	logarithme décimal de e
`* Math.LOG2E`	**`1.4426950408889634`**	logarithme base 2 de 10
`* Math.SQRT1_2`	**`0.7071067811865476`**	**`inverse de racine de 2`**
`* Math.SQRT2`	**`1.4142135623730951`**	racine de 2

1.3. les méthode de Math.

Les arguments et les valeurs retournées sont de type **`number`**.

Les fonctions trigonométriques fonctionnent en radians.

*** abs(arg)**	Retourne la valeur absolue de l'argument
*** acos(arg)**	Retourne l'arc cosinus de l'argument .
*** asin(arg)**	Retourne l'arc sinus de l'argument.
*** atan(arg)**	Retourne l'arc tangente de l'argument.
*** ceil(arg)**	Retourne l'entier le plus proche, supérieur ou égal à l'argument +1.6 => 2 +1.2 => 2 -1.2 => -1 -1.6 => -1 arrondi par excès
*** cos(arg)**	Retourne le cosinus de l'argument .
*** exp(arg)**	Retourne l'exponentielle du réel donné en paramètre
*** floor(arg)**	Retourne l'entier inférieur ou égal le plus proche de l'argument.
*** log(arg)**	Retourne le logarithme népérien (ln) de l'argument.
*** max(arg1, arg2...)**	Retourne le maximum parmi les arguments.
*** min(arg1, arg2...)**	Retourne le minimum parmi les arguments.
*** pow(argx, argy)**	Retourne argx à la puissance argy. ; exemple : racine cubique : **`Math.pow(arg, 1/3)`**
*** random()**	Retourne un nombre aléatoire entre 0 et 1.
*** round(arg)**	Retourne l'arrondi (entier) de l'argument
*** sin(arg)**	Retourne le sinus de l'argument.
*** sqrt(arg)**	Retourne la racine carrée de l'argument .
*** tan(arg)**	Retourne la tangente de l'argument.

Pour convertir des degrés en radians multiplier par **`Math.PI/180`**

2. l'objet `Date`

2.1. le constructeur `Date`.

syntaxes :

```
new Date()
```

Crée un objet **`Date`** dont les méthodes sont prédéfinies, représentant **la date et l'heure courante** que l'on peut afficher (**`toString()`**). C'est à cet objet que l'on appliquera les méthodes utilitaires :

```
Date {Wed Apr 11 2012 12:31:20 GMT+0200 (CEST)}
```

Dans les cas suivants, l'argument décrit la durée depuis **le premier janvier 1970 (UTC),** et retourne un objet **`Date`**.

```
new Date (millisecondes)
new date(chaîne date acceptée par Date.parse())
new Date (année, mois, jour, heures, minutes, secondes, ms)
```

2.2. Méthodes de `Date`.

*** Date.parse()** : retourne la durées en millisecondes, entre la date et l'heure donnée en argument et la référence UTC. La date est donnée suivant un schéma précis :

```
Date.parse("Tue, 1 Jan 2000 00:00:00 GMT");
```

*** Date.UTC()** : même chose, avec un format spécifique, en millisecondes.

```
Date.UTC(1998,2,11,19,26,00)
```

2.3. Méthodes des objets Date.

Les méthodes, nombreuses, ont comme appelant un objet rendu par le constructeur **`Date()`**.

*** getDate()**	rechercher le jour du mois
*** getDay()**	rechercher le jour de la semaine
*** getFullYear()**	rechercher l'année complète)
*** getHours()**	rechercher la partie heures de l'heure
*** getMilliseconds()**	rechercher les millièmes de secondes
*** getMinutes()**	rechercher la partie minutes de l'heure
*** getMonth()**	rechercher le mois
*** getSeconds()**	rechercher la partie secondes de l'heure
*** getTime()**	rechercher l'heure
*** getTimezoneOffset()**	rechercher le décalage horaire de l'heure locale
*** getUTCDate()**	rechercher le jour du mois de l'heure UTC
*** getUTCDay()**	rechercher le jour de lla semaine de l'heure UTC
*** getUTCFullYear()**	rechercher l'année complète de l'heure UTC
*** getUTCHours()**	rechercher la partie heures de l'heure UTC
*** getUTCMilliseconds()**	rechercher les millièmes de secondes de l'heure UTC
*** getUTCMinutes()**	rechercher la partie minutes de l'heure UTC
*** getUTCMonth()**	rechercher le mois de l'heure UTC
*** getUTCSeconds()**	rechercher la partie secondes de l'heure UTC
*** getYear()**	rechercher l'année
*** setDate()**	fixer le jour du mois
*** setFullYear()**	fixer l'année complète
*** setHours()**	fixer la partie heures de l'heure
*** setMilliseconds()**	fixer la partie millièmes de seconde de l'heure
*** setMinutes()**	fixer la partie minutes de l'heure
*** setMonth()**	fixer la partie mois de la date
*** setSeconds()**	fixer la partie secondes de l'heure

*** setTime()**	fixer la date et l'heure
*** setUTCDate()**	fixer le jour du mois de l'heure UTC
*** setUTCDay()**	fixer le jour de la semaine de l'heure UTC
*** setUTCFullYear()**	fixer l'année complète de l'heure UTC
*** setUTCHours()**	fixer la partie heures de l'heure UTC
*** setUTCMilliseconds()**	fixer la partie millièmes de seconde de l'heure UTC
*** setUTCMinutes()**	fixer la partie minutes de l'heure UTC
*** setUTCMonth()**	fixer le mois de l'heure UTC
*** setUTCSeconds()**	fixer la partie secondes de l'heure UTC
*** setYear()**	fixer la date et l'heure
*** toGMTString()**	convertir la date et l'heure au format GMT
*** toLocaleString()**	convertir la date et l'heure au format local

Les setters changent le champ précisé de l'objet date et retourne la nouvelle date (en millisecondes ...)

exemple :

```
1.<script type="text/javascript">
2.     var mois = new Array (" janvier ", " février "," mars ", " avril ",
3.               " mai ", " juin ", " juillet ", " août ", " septembre ",
4.               " octobre ", " novembre ", " décembre ") ;
5.     var maintenant = new Date() ;
6.     var chn ="" ;
7.     var ageDate = function () {
8.            var a = maintenant.getFullYear() ;
9.            var m = maintenant.getMonth() ;
10.           var j = maintenant.getDate() ;
11.           chn += ("aujourd'hui : " + j+mois[m]+a + "\n") ;
12.           chn += ("mon âge : " + (a - 1937) + " ans\n" ) ;
13.       }
14.    var bonWeekEnd = function () {
15.           var j = maintenant.getDay()
16.           if (j==0 || j==6)
17.               chn += "Nous vous souhaitons un bon week-end\n" ;
18.           else
19.               chn += "nous espérons que vous avez bien travaillé\n" ;
20.        }
21.    ageDate () ;
22.    bonWeekEnd() ;
23.    chn+= "nouvelle date : " +maintenant.setDate(15)+ " "+maintenant+"\n" ;
24.    bonWeekEnd() ;
25.    alert (chn);
26.</script>
27.<!-- h1101_date.html →
```

résultat :

```
aujourd'hui : 11 avril 2012
mon âge : 75 ans
nous espérons que vous avez bien travaillé
nouvelle date : 1334493956853 Sun Apr 15 2012 14:45:56 GMT+0200 (CEST)
Nous vous souhaitons un bon week-end
```

11 : les wrappers

1. notion de constructeur «d'enveloppement» (wrapper).

1.1. Les données primaires ne sont pas des objets.

Ce truisme est cependant lourd de conséquence en programmation objets. Les objets disposent de propriétés, de données et surtout de méthodes qui permettent de les utiliser pour réaliser les actions essentielles d'un programme ; ce qui vient d'être exposé sur les objets **`Array`** ou **`Math`** en témoigne.

Rien de tel pour les données primaires : un de leurs gros avantages est la simplicité de leur représentation en mémoire, gage d'économie et de rapidité, qui étaient importants tant que la mémoire était chère et les processeurs lents. Mais les données primaires ne possèdent aucun des paradigmes qui fondent l'ingénierie objet. On doit se contenter de quelques fonctions globales et opérateurs communs comme **`parseInt(), +, -, *, / % &,&&, |, !`** et l'arsenal est restreint.

Aussi, les logiciels ayant conservé l'usage de données primaires (Java par exemple) ont aussi donné la possibilité de leur adjoindre des objets qui les «enveloppent» et qui eux**,** disposent de tout l'arsenal logiciel des objets. Ces objets sont les **wrappers**.

1.2. fonctionnement.

syntaxe de construction : **`new wrapper (<donnée primaire>)`**

La donnée primaire peut être : **`boolean, string, number`** et les wrappers correspondant **`Boolean()`**, **`String()`** et **`Number()`**. L'opérateur **`new`** retourne un objet.

syntaxe de "déconstruction" : **`wrapper (valeur)`**

Le problème est que dans certaines circonstances, ce n'est pas de l'objet dont on a besoin, mais de la valeur primaire : c'est le cas avec les opérateurs classiques, qui ne fonctionnent pas sur des objets !

Les fonctions wrapper sont donc également des fonctions de conversion qui retournent une valeur primaire. La valeur est en principe un objet, mais pas nécessairement.

note importante : lorsque le **wrapper** est utilisé comme fonction, son appelant est **`window`**. Ce n'est évidemment pas le cas dans le mode constructeur : on rappelle le mécanisme selon lequel **`new`** crée un objet "générique" qui est l'appelant du constructeur, et que celui-ci est en mesure d'enrichir (usage de **`this`**).

2. le wrapper Number().

2.1. Syntaxes.

constructeur : **`new Number(valeur)`**

valeur est un nombre (**`number`**)

conversion : **`Number (valeur)`**

`valeur` est une donnée à convertir en nombre : un objet, une chaîne, un booléen… Interprète bien les chaînes avec des décimaux (considérées comme décimales, pas octale) ou écrits avec la base 16.

`false` se transforme en 0 et **`true`** en 1.

Si la conversion échoue, retourne **`NaN`**.

2.2. Les propriétés de `Number`.

`MIN_VALUE, MAX_VALUE, NaN, NEGATIVE_INFINITY, POSITIVE_INFINITY`

Ces propriétés de **`Number`** retournent un élément de type **`number`**.

exemple : **`alert (Number.MIN_VALUE) ;`**

2.3. Les propriétés des instances.

Les méthodes retournent une erreur **`RangeError`** ou **`TypeError`** le cas échéant.

*** toExponential (*nombre de chiffres*)**

retourne une chaîne, écrivant le nombe wrappé avec ***`nombre de chiffres`*** *après la virgule.*

```
var unNumber123 = new Number(123) ;
   alert ( Number.MAX_VALUE) ;
   alert (unNumber123.toExponential(3));
```

donne :

```
1.7976931348623157e+308
1.230e+2
```

*** toFixed (*nombre de chiffres*)**

même chose, mais en notation à virgule fixe.

*** toPrecision(*precision*)**

`precision` indique le nombre de chiffres significatifs (entre 1 et 21 chiffres)

*** toString(), toLocaleString()**

conversion en chaînes. Surcharge des méthodes de **`Object()`**.

*** valueOf()**

retourne la valeur primaire enveloppée ; il est rare de devoir faire appel à cette méthode qui est utilisée en interne.

3. le wrapper String().

3.1. Syntaxes.

constructeur : **`new Sring(`*`valeur`*`)`**

valeur est une chaîne (**`string)`**

conversion : **`String (`*`valeur`*`)`**

`valeur` est une donnée à convertir en chaîne primaire : un objet, une chaîne, un nombre, un booléen…

3.2. Les propriétés de String.

Le transcodage avec les chaînes primaires est automatique.

*** fromCharCode(*liste de valeurs*)**

Les valeurs sont les codes numériques des caractères, séparés par des virgules. Utiles pour les **conversions clavier** dans un environnement web.

```
var s = String.fromCharCode(83,97,105,110,116,32,201,108,111,105) ;
alert ((typeof s)+"   "+s);
```

affiche : string Saint Éloi

*** concat (*liste de valeurs*)**

Les valeurs séparées par des virgules sont transformés en chaînes. Retourne une chaîne primaire, concaténation des valeurs de la liste. On préfère souvent utiliser l'opérateur **+** .

`String.concat("la valeur de 3*5 est ", 3*5)`

équivaut à : **`"la valeur de 3*5 est " + 3*5`**

3.3. Les propriétés des instances de String.

*** length**

nombre de caractères de la chaîne. **Le transtypage est automatique** sur les chaînes primaires.

```
var maChaine = "ceci est une chaîne primaire" ;
alert ((typeof maChaine)+"  "+maChaine.length) ;
```

affiche : `string 28`

*** charAt(*position*)**

retourne une chaîne primaire avec un seul caractère, situé au rang ***position*** (attention : on commence à 0).

*** charCodeAt(*position*)**

retourne un nombre (**number**) qui est le code du caractère à la position indiquée.

*** concat(*liste de valeurs*)**

Les valeurs séparées par des virgules sont transformés en chaînes. La chaîne propriétaire n'est pas changée. Retourne une chaîne primaire, concaténation des valeurs de la liste.

*** indexOf(*sous chaîne*)** et **indexOf(*sous chaîne, début*)**

retourne l'indice de la *sous chaîne* dans l'objet chaîne propriétaire. Dans le second cas, commence la recherche à *début*.

*** localeCompare(*chaîne cible*)**

compare l'ordre lexicographique local de la *chaîne propriétaire* et de la *chaîne cible* et retourne une valeur booléenne.

*** match(*expression régulière*)**

voir le chapitre expressions régulières.

*** replace (*expression régulière, chaîne de* remplacement)**

voir le chapitre expressions régulières.

*** search(*expression régulière*)**

voir le chapitre expressions régulières.

*** slice(*début, fin*)**

extrait une sous chaîne entre *début* et *fin*. *début* est pris, *fin* exclus ; la longueur de la chaîne extraite est (fin - début). Les valeurs peuvent être négatives (comme pour les tableaux). Ne touche pas à la chaîne propriétaire et retourne la chaîne extraite.

*** split(*chaîne délimitrice, limite*)**

retourne un tableau de chaînes obtenue par sectionnement de la chaîne propriétaire par la *chaîne délimitrice*. Le délimiteur n'apparaît pas dans le tableau. *limite* donne la dimension maximale du tableau. Au dela de *limite* éléments, tout est perdu, il n'y a pas de concaténation du reste de la ligne dans l'élément d'indice *limite*.

Si la chaîne propriétaire commence par le *délimiteur*, le premier item du tableau est une chaîne vide.

Si la *chaîne délimitrice* est vide, on obtient un tableau de caractères.

La chaîne *délimitrice* peut être une expression régulière (voir le chapitre expressions régulières)

La méthode peut être vue comme l'inverse de **join()** dans **Array**.

```
var s = "il était une fois à l'est d'Éden" ;
var tableau = s.split(" ");
```

0	"il"

```
1       "était"
2       "une"
3       "fois"
4       "à"
5       "l'est"
6       "d'Éden"
```

*** substr(début, longueur)**

retourne une sous chaîne commençant à *début* et de taille *longueur* de la chaîne propriétaire ; ne touche pas à celle-ci. L'absence de *longueur* donne toute la fin de chaîne. Un indice négatif est permis mais mal supporté.

*** substring(*début, jusque*)**

retourne une sous chaîne commençant à *début* et se terminant avant *jusque*. Le second argument est facultatif (on va en fin de chaîne dans ce cas).

*** toLowerCase, toLocaleLowerCase, toUpperCase, toLocaleUpperCase**

Change la casse. Ne touche pas à la chaîne propriétaire et retourne la chaîne résultante.

```
var s = "il était une fois à l'est d'Éden" ;
var chn ="";
chn += "chaîne avant : " + s + "\n" ;
chn +=  "upper        : " + s.toUpperCase() + "\n";
chn += "lower        : " + s.toLowerCase() + "\n";
chn += "chaîne après : " + s  ;
alert (chn) ;
```

```
chaîne avant : il était une fois à l'est d'Éden
upper : IL ÉTAIT UNE FOIS À L'EST D'ÉDEN
lower : il était une fois à l'est d'éden
chaîne après : il était une fois à l'est d'Éden
```

*** toString(*chaîne*)**

retourne la chaîne.

*** valueOf()**

retourne la chaîne !

4. le wrapper Boolean().

4.1. Syntaxes.

constructeur : **`new Boolean(`*`valeur`*`)`**

valeur est un **`booléen`**, où une *valeur* transformée en booléen.

conversion : **`Boolean (`*`valeur`*`)`**

`valeur` est une donnée à convertir en booléen : retourne un primaire booléen.

Les valeurs **`0, NaN, null, `*`chaîne vide`*`, undefined`** sont converties à **`false`**.

4.2. Les propriétés des instances de Boolean.

*** toString()**

retourne **`"true"`** ou **`"false"`**.

*** valueOf()**

retourne un booléen primaire.

5. Object() comme fonction.

5.1. Objet() est une fonction globale.

Lorsque **`Object()`** est appelé avec un argument primaire **`boolean, string, number,`** il retourne la même chose que **`new Boolean(), new String(), new Number()`**. Si l'argument est un objet, il le retourne simplement.

5.2. une simulation.

On trouvera ci-dessous une simulation de la fonction **`Object()`**, avec un autre nom évidemment, **`fn`**. On peut voir comment avec une évaluation de l'objet propriétaire, un constructeur peut, dans certaines circonstances, retourner une donnée. (ce qui nuance l'assertion selon laquelle **un constructeur ne retourne rien**). Dans le cas actuel, **`new fn()`** se comporte comme **`new Object().`**

```
1.<script type="text/javascript">
2.     var fn = function (param) {
3.         if (param & this==window){
4.              switch (typeof param) {
5.                  case "number"   : return new Number(param) ;
6.                  case "string"   : return new String(param);
7.                  case "boolean"  : return new Boolean (param) ;
8.                  case "object"   : return param ;
9.                  case "function" : return param ;
10.             }
11.       }
12.    }
13.    var xxx =  new fn
14. </script>
15. <!-- h1100_constructeur.html -->
```

12 : objet prédéfini Error

1. Exceptions et erreurs.

1.1. Argument d'exception.

On a vu au chapitre t05 que **`throw`** et **`catch`** ont un argument dont l'utilité consiste à transmettre tous les renseignements concernant une exception : celle provoquée par **`throw`**, ou celle qui doit être traitée das le bloc de **`catch()`**. Toute donnée peut être donnée comme valeur d'argument, tout primaire ou tout objet. Cependant, l'usage a fait que cet argument est soit un entier (code d'erreur), soit une chaîne de caractère (message d'erreur). JavaScript a prévu un objet spécialisé qui caractérise les exceptions, de constructeur appelé **`Error`**.

1.2. L'exception générique Error.

syntaxes :

```
new Error()
new Error(message)
```

`message` est une chaîne de caractères fournissant des détails sur l'exception

l'opérateur new retourne un nouvel objet erreur qui peut être argument de **`throw`**.

1.3. Les propriétés.

*** `message` :** contient la chaîne éventuellement passée en paramètre du constructeur.

* **`name`** : chaîne qui donne le genre d'exception (pour les decendants de **`Error`**, c'est le nom du constructeur).

* **`toString()`** : retourne une chaîne ; cette chaîne est définie par l'implémentation de **`Error.`**

2. Constructeurs implémentés par JavaScript.

2.1. les constructeurs implémentés.

Le constructeur Error n'est pas utilisé directement ; ce sont ses descendants qui le sont : **`EvalError, RangeError, ReferenceError, SyntaxError, TypeError...`**

2.2. Utilisation de ces constructeur.

`* EvalError,`

Lors d'une erreur dans l'utilisation de la fonction **`eval()`**, une exeption **`EvalError`** est générée.

`* RangeError,`

Erreur générée lorsqu'un nombre est hors de la portée légale des valeurs numériques acceptées : par exemple, valeur négative pour un indice de tableau..

`* ReferenceError,`

Erreur générée quand on utilise (en lecture) une variable qui n'existe pas.

`* SyntaxError,`

Erreur générée par une erreur de syntaxe (en particulier avec **`eval(), Function(), RegEx()`**).

`* TypeError,`

Erreur lorsqu'une valeur utilisée n'est pas du type attendu : utilisation d'une méthode qui n'appartient pas à l'objet, utilisation de **`new`** sur une fonction qui n'est pas un constructeur, utilisation d'une variable **`undefined`** ou **`null`** où ce n'est pas autorisé, ou quand il y a trop d'arguments dans un appel de méthode ....

13 : RegExp

1. Expressions Régulières.

1.1. principe.

Les expressions régulières constituent **une méthode de description** de chaînes de caractère. **Une expression régulière est un objet** qui constitue une telle description. La chaîne de caractère qui sert à exprimer la description s'appelle **un motif (*pattern*)**. Les principales utilisations des expressions régulières sont la recherche de motifs dans une chaîne donnée, et la recherche/remplacement de sous chaînes d'une chaîne qui correspondent au motif.

exemple :

- le motif à décrire : suite d'espaces en début de ligne ;
- action : supprimer dans un texte les suites d'espace en début de ligne.

note : par abus, on utilise le mot **motif** pour désigner le texte de la description, la chaîne descriptive, ou l'objet JavaScript créé à partir de la chaîne descriptive (ou chaîne de motif, techniquement appelée **`source`** de l'objet **`RegExp`**). Il y a un abus également dans l'utilisation du terme «expression régulière», qui peut désigner aussi bien la chaîne de motif que l'objet **`RegExp`**. Le vocabulaire est issu des usages dans d'autres langages et dans la théorie des automates, en particulier du langage **Pearl** dont le module a été adapté pour **JavaScript**.

1.2. Les objets RegExp.

`RegExp()` est un constructeur ; il produit des **objets RegExp**.

syntaxe : **`new RegExp (`*`chaîne de motif`*`)`**

L'argument est une chaîne de caractères avec une syntaxe propre, pour pouvoir être transformée en objet **`RegExp`**.

littéral : il existe un littéral **`RegExp`**, qui évite d'appeler explicitement le constructeur :

`/`*`chaîne de motif`*`/``options`**

Pour des raisons pratiques, nous utiliseront presque toujours la forme littérale dans les exemples.

1.3. retour sur les objets String.

Les objets **`RegExp`** sont inséparables des objets **`String`**. En effet, les méthodes les plus immédiates sont celles qui consistent à identifier (et éventuellement à remplacer) dans un objet **`String`** les sous-chaînes qui sont décrites par le motif. On reviendra sur les détails concernant chacune des méthodes.

Pour l'instant, voici quelques indications pour comprendre les exemples :

*** match(*expression régulière*)**

Recherche la ou les sous-chaînes décrites par l'expression régulière (si celle-ci est remplacée par une chaîne, le transtypage est automatique). Retourne un tableau

Retourne **`null`** si aucune sous-chaîne n'a pu être identifiée. Un tableau concernant la ou les identifications dans le cas contraire.

*** replace (*expression régulière, chaîne de remplacement*)**

Remplace la ou les sous-chaînes identifiée par la chaîne de remplacement.

*** search(*expression régulière*)**

Retourne la position de la première sous-chaînes identifiées ; **`-1`** en cas d'échec.

2. Définition de la chaîne de motif.

2.1. caractères littéraux.

Tous les caractères ont vocation à se représenter eux mêmes.

exemple : **`/papa est en voyage/`** est un objet **`RegExp`**, formé de 18 caractères «ordinaires». Une action classique dans les traitements de texte consiste à rechercher (ou à chercher/remplacer) la

première occurrence d'une chaîne dans un texte (où l'occurrence à partir d'un certain endroit du texte). La chaîne à recherchée peut être une chaîne de motif de **RegExp**.

caractères non graphiques : un certain nombre de caractères n'ont pas de glyphe. Par exemple la tabulation, la fin de ligne etc. Dans ces cas on doit utiliser un caractère d'échappement pour les introduire dans une chaîne de motif. Le caractère d'échappement en javaScript est **l'antislash **

Les caractères sont alors représentés par deux caractères graphique, le premier étant ****

	quel caractère ?	remarques	unicode
\0	caractère NUL		**\u0000**
\t	caractère de tabulation	**TAB**	**\u0009**
\n	caractère de retour à la ligne	**LF**	**\u000A**
\v	caractère tabulation verticale		**\u000B**
\f	caractère saut de page	**FF**	**\u000C**
\r	retour chariot	**LF**	**\u000D**
\x*nn*	caractère latin de code hexadécimal *nn*	les caractères hex **a-f** en majuscules ou minuscules	
\uxxxx	caractère unicode de code hexadécimal *xxxx*		
\cX	caractère de contrôle		

caractères interdits :

Certains caractère **ne se représentent pas eux mêmes**, car ils vont avoir une signification particulière. Ce sont des caractères de ponctuation ou des caractères ayant un usage spécifique :

^ $. * + ? = ! : | \ / ( ) [ ] { }

Noter que dans ces caractères, il n'y a pa le moins **-**, les quotes simples ou doubles **" '** , l'espace, la virgule ou le point-virgule **;** le dièse, **#**, l''esperluette **&**, l'arrobase **@**, l'underscore **_**

Si on introduit un caractère interdit, il faut l'échapper : **\(** pour la parenthèse, **\.** pour le point et bien évidemment **** pour l'antislash. Ainsi le littéral **RegExp** pour l'antislash est **/\\/**

Si on utilise la syntaxe littérale, **il n'y a pas lieu d'échapper les quotes**. Un antislash abusif est en général sans conséquence sauf dans les cas où l'échappement donne un sens particulier (**\i** , avec **i** entier par exemple).

2.2. les classes de caractères.

On a souvent besoin d'exprimer un caractère appartenant à une classe de caractères : un chiffre, tout sauf un chiffre, un caractère valide dans un identificateur, un ASCII majuscule...

le caractère	correspondance	équivalent	exemple
[caractères]	un des caractères entre crochets		[abc] soit a, soit b, soit c
[^caractères]	tout sauf un caractère entre crochets		[^abc] tout sauf a, b, c
[x-y]	tout caractère entre x et y		[A-Z] majuscule ASCII
[x-yz-t]	tout caractère entre x et y ou entre z et t		[A-Za-z_0-9]
.	le point : tout caractère sauf un caractère de fin de ligne.		
\w	tout ASCII pour identificateur	[A-Za-z_0-9]	
\W	tout sauf ASCII pour identificateur	[^A-Za-z_0-9]	
\s	tout caractère d'espacement		
\S	tout caractère non espacement		
\d	tout chiffre	[0-9]	
\D	tout sauf un chiffre	[^0-9]	

* les caractères entre crochet peuvent être échappés : **`/[\s\d]/`** désigne tout caractère qui est soit un espacement (espace, tab etc) soir un chiffre.

* il existe un échappement spécial : **`\b`** ; il désigne le back-space **si on le trouve entre crochets** :

`/[\b]/` est valide pour le back-space.

* quelques exemples :

* **`/\d\d/`** signifie une suite de deux chiffres et **`/\d\d\d\d/`** une suite de quatre chiffres.

* **`/ /`** signifie un espace, ce qui est souvent source d'erreur, surtout si on a tendance à éclaircir les sources pour les rendre plus lisibles : les espaces appartiennent au motif !

* /**`[0-9a-fA-F]`**/désigne un chiffre hexadécimal ; faire attention qu'un espace ne se glisse pas entre les crochets !

exemple : test sur \w

```
1.<script type="text/javascript">
2.    var chnEntree ="une chaîne-123&leçon";
3.    var motif = /\W/g;
4.    var chnSortie = chnEntree.replace (motif, "^^^");
5.    alert (chnSortie) ;
6.</script>
7.<!-- h1401_re.html -->
```

```
une chaîne-123&leçon
une^^^cha^^^ne^^^123^^^le^^^on
```

exemple : test sur **`\s`** (espaces)

```
1.<script type="text/javascript">
2.    var chnEntree ="une chaîne       (2 tab)\n-123\xA0&\u00a0leçon";
3.    var motif = /\s/g;
4.    var chnSortie = chnEntree.replace (motif, "^^^");
5.    alert (chnEntree+"\n"+chnSortie) ;
6.</script>
7.<!-- h1402_re.html →
```

```
une chaîne       (2 tab)
-123 & leçon
une^^^chaîne^^^^^^(2^^^tab)^^^-123^^^&^^^leçon
```

2.3. les répétiteurs.

Un répétiteur est un moyen de décrire la répétition d'un caractère (ou d'un groupe de caractère)

exemples : autant de fin de lignes successives que l'on veut ;
deux espaces au moins de suite ;
entre 3 et 5 chiffres...

répétiteur	signification	exemple
*	zéro, un ou plusieurs	**`a*`** : 1, n... ou 0 caractère(s) **a** en suivant
+	au moins un	**`\d+`** au moins un chiffre
?	facultatif	**`a? a peut être présent ou non, pris en considération ou non (3 cas posibles)`**
{n}	exactement n fois	**`[0-9a-fA-F]{4}`** un hexadécimal à 4 chiffres

{n,} {n,m}	au moins n fois au moins n fois, pas plus de m fois	

exemples :

* **/\d{2,4}/** une suite de 2 à 4 chiffres : 00, 595, 9874, 0001

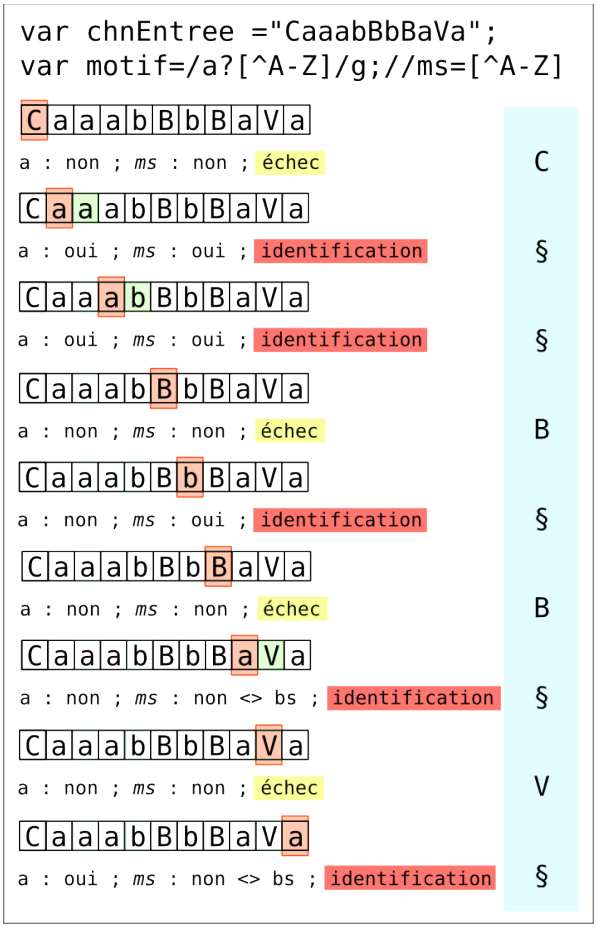
* **/\s+java\s+/** le mot java encadré obligatoirement par des espaces ou assimilés.

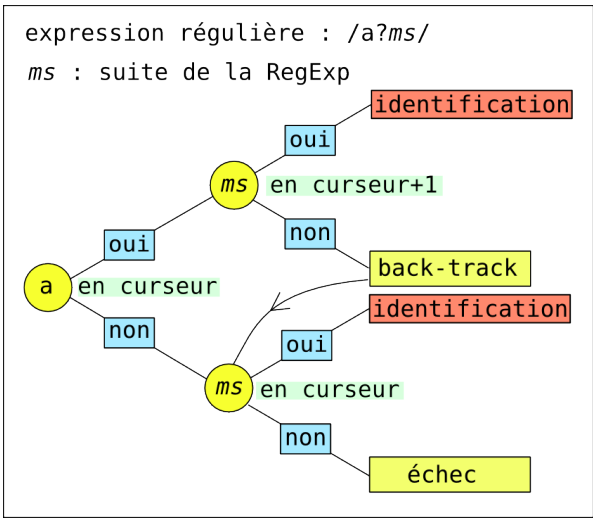
* **/"[^"]*"/** une quote, éventuellement des caractères, sauf une quote, puis une quote comme dans **"abcd efgh"**, ou **""**

attention : les répétiteurs **?** et ***** peuvent se révéler peu compréhensibles et doivent être utilisés avec précaution.

```
1.<script type="text/javascript">
2.    var chnEntree ="CaaabBbBaVa"; ;
3.    var motif = /a?[^A-Z]/g;
4.    var chnSortie = chnEntree.replace (motif, "§");
5.    alert (chnEntree+"\n"+chnSortie) ;
6.</script>
7.<!-- h1403_re.html -->
```

le motif **/a?[^A-Z]/** décrit : 0 ou 1 caractère **a** non suivi par une majuscule. On examine d'abord si un **a** convient ; en cas d'échec, on prend l'absence de **a** en considération.





CaaabBbBaVa
C§§B§B§V§

2.4. gourmandise.

L'utilisation des répétiteurs peut poser problème : Par exemple, on peut demander si le motif **/ba+/**

décrit une sous chaîne de **baaaaaaaaaab** et quelle est cette sous chaîne. La solution courte se serait **ba** , la solution longue **baaaaaaaaaa** et tous les intermédiaires sont possibles. En fait on ne retient que les deux extrêmes. Mais il reste à la discriminer : **par défaut, les répétiteurs sont gourmands** (avides), c'est-à-dire qu'il décrivent la solution la plus longue. On peut **les forcer** à décrire la solution la plus courte ; dans ce cas on leur adjoint un point d'interrogation : **+? *? {n,m}?**

Attention : supposons que l'on ait **/a*?b/** Comment se fait la description ? Il faut rechercher **a.** Si on le trouve, on continue avec la recherche de **b**, mais en égrenant les **a**. Donc le motif décrit bien la sous-chaîne **aaab** par exemple dans la chaîne **cccaaabbb**. Par contre dans la chaîne **cccbeee**, seule la sous-chaîne **b** est décrite.

```
1.<script type="text/javascript">
2.    var chnEntree ="CaaabBBBaVaaaabDbbEbF";
3.    var motif = /a*?b/g;
4.    var chnSortie = chnEntree.replace (motif, "§");
5.    alert (chnEntree+"\n"+chnSortie) ;
6.</script>
7.<!-- h1403_re.html →
```

```
CaaabBBBaVaaaabDbbEbF
C§BBBaV§D§§E§F
```

2.5. Les attributs ou drapeaux.

Les attributs modulent la façon dont la recherche de sous-chaîne décrite par le motif doit être faite.

* attribut **i** : (ignoreCase) ne pas distinguer majuscules et minuscules ;

* attribut **g** : (global) rechercher dans tout le texte et ne pas s'arrêter à la première identification ;

* attribut **m** : recherche multiligne ; permet d'identifier les débuts et fins de lignes (voir Ancrage)

Les attributs (appelés aussi drapeaux) se placent après le slash final des littéraux, ou en second argument du constructeur.

```
/chaîne de motif/img
new RegExp (chaîne de motif, attributs)
```

Les attributs sont des propriétés des objets **RegExp** : **global, multiline, ignoreCase.** Attention, ces propriétés sont **en lecture seule** ! Ces attributs ne sont pas modifiables : ils appartiennent à la **RegExp** ; et si dans un script on veut utiliser le même motif plusieurs fois avec des attributs différents, il faut créer autant de **RegExp** différentes.

exemple d'utilisation :

```
motif = new RegExp (chn, "g")
      alert (motif.global+" "+motif.multiline+" "+motif.ignoreCase);
```

3. Ancrage des RegExp.

3.1. La syntaxe.

Les textes sont en général considérés comme ayant une double structure : en mots et en lignes (ce que les traitements de textes nomment paragraphes !).

Les ancrages servent à préciser où, dans un texte, il faut chercher les sous-chaînes décrites par un motif.

ancrages	explications
^	le motif doit décrire une sous-chaîne en début de chaîne ou de ligne
$	le motif doit décrire une sous-chaîne en fin de chaîne ou de ligne
\b	limite d'un mot. Encadrement par **\b** : le motif doit décrire un mot
\B	la position n'est pas limite d'un mot
(*?=chaîne de motif*)	les caractères qui suivent recherchent le motif, mais ne l'incluent pas dans

(?!*chaîne de motif*)	la sous-chaîne identifiée. les caractères suivants excluent le motif.

3.2. Début et fin de la chaîne de recherche.

Les ancres **^**et **$** fonctionnent sur la chaîne à analyser si le mode est monoligne et sur les lignes lorsque l'attribut m est posé (mode multiligne).

exemple : supprimer les blancs (espaces, tabulations) en début de chaîne (ligne) ou fin de chaîne (ligne).

```
1.<script type="text/javascript">
2.    var chnEntree ="    papa est en voyage          ";
3.    var motif1 = /^\s*/ , motif2 = /\s*$/ ;
4.    var resultat = (chnEntree.replace(motif1,"")).replace(motif2,"") ;
5.    alert ("***"+chnEntree+"***\n"+"***"+resultat+"***") ;
6.</script>
7.<!-- h1405_re.html →
```

```
***         papa est en voyage        ***
***papa est en voyage***
```

3.3. notion de mots.

la forme d'échappement **\b** présente quelques difficultés : dans la chaîne de motif, elle peut représenter aussi la tabulation ; on a vu qu'il faut la crocheter ! Elle se révèle incommode dès que l'on travaille sur des textes : elle considère comme début ou fin de mot tout ce qui appartient à **\W** et donc les caractères accentués, la ponctuation (sauf l'underscore) etc. Il convient donc souvent de définir «à la main» ce que l'on considère comme un mot.

exemple 1 :

```
1.<script type="text/javascript">
2.  var chnEntree ="Java apparaît en 1993, JavaScript un an plus tard.\n"+
3.    "Il s'appelle LiveScript, et tente de profiter de la notoriété de Java,\n"+
4.    "non pas java, le «petit noir», ni «lajavanaise» de Gainsbourg \n"+
5.    "ni l'île de Java, mais le nouveau langage dont on commence à parler.";
6.  var motif = /\bjava\b/gmi;
7.  var resultat = chnEntree.replace(motif, "§§§§") ;
8.  var alert (chnEntree+"\n\n"+resultat) ;
9.</script>
10.<!-- h1406_re.html -->
```

```
Java apparaît en 1993, JavaScript un an plus tard.
Il s'appelle LiveScript, et tente de profiter de la notoriété de Java,
non pas java, le «petit noir», ni «lajavanaise» de Gainsbourg
ni l'île de Java, mais le nouveau langage dont on commence à parler.

§§§§ apparaît en 1993, JavaScript un an plus tard.
Il s'appelle LiveScript, et tente de profiter de la notoriété de §§§§,
non pas §§§§, le «petit noir», ni «lajavanaise» de Gainsbourg
ni l'île de §§§§, mais le nouveau langage dont on commence à parler.
```

exemple 2 :

Voici un exemple qui peut être affiné (il y a quelques caractères parasites et il manque les ligatures) .

On recherche des mots en langues ouest européennes :

```
1.<script type="text/javascript">
2.    var chnEntree ="aaa Ami avecéàaze qsdf%sdf,vcx";
3.    var motif = /[a-zA-Z\u00C0-\u00FF]+/g;
4.    var resultat = chnEntree.match(motif) ;
5.    alert (resultat) ;
6.</script>
7.<!-- h1407_re.html -->
```

```
0       "aaa"
1       "Ami"
2       "avecéàaze"
3       "qsdf"
4       "sdf"
5       "vcx"
```

3.4. assertion vers l'avant.

```
1.<script type="text/javascript">
2.  var chnEntree ="Java apparaît en 1993, JavaScript un an plus tard.\n"+
3.    "Il s'appelle LiveScript, et tente de profiter de la notoriété de Java,\n"+
4.    "non pas java, le «petit noir», ni «lajavanaise» de Gainsbourg \n"+
5.    "ni l'île de Java, mais le nouveau langage dont on commence à parler.";
6.  var motif = /java(?!script)/gmi;
7.  var resultat = chnEntree.replace(motif, "§§§§") ;
8.  alert (chnEntree+"\n\n"+resultat) ;
9.</script>
10.<!-- h1408_re.html -->
```

`/java(?!script)/gmi;` le mot **`java`** est recherché, mais non suivi de **`script`** ; la recherche est globale (tout est recherché), multi-ligne, et indifférente à la casse.

```
Java apparaît en 1993, JavaScript un an plus tard.
Il s'appelle LiveScript, et tente de profiter de la notoriété de Java,
non pas java, le «petit noir», ni «lajavanaise» de Gainsbourg
ni l'île de Java, mais le nouveau langage dont on commence à parler.

§§§§ apparaît en 1993, JavaScript un an plus tard.
Il s'appelle LiveScript, et tente de profiter de la notoriété de §§§§,
non pas §§§§, le «petit noir», ni «la§§§§naise» de Gainsbourg
ni de l'île de §§§§, mais le nouveau langage dont on commence à parler.
```

4. Expressions parenthésées (ou groupements).

4.1. Une chaîne de motif peut être parenthésée.

Le système de parenthèses (évidemment bien balancé) peut fonctionner comme en mathématiques : il isole un segment du motif, un groupement, sur lequel par exemple appliquer un répétiteur (ou ne rien faire). Les parenthèses peuvent être imbriquées.

Une chaîne de motif qui comporte des parenthèses définit des sous-motifs. Il est utile de **numéroter les parenthèses ouvrantes, de gauche à droite, en commençant à 1 et pas à zéro !** Les sous-chaîne reconnues sont numérotées de la même façon, que celles-ci soient imbriquées ou non. La

chaîne entière est numérotée 0. Si on veut qu'une parenthèse de groupement ne soit pas numérotée, on utilise **(?:...)**. Si un groupe est numéroté avec la valeur **i**, alors **\i** permet de rechercher les mêmes caractères que ceux trouvés lorsque le numéro de groupe **i** a été identifié.

Lors d'une recherche ou d'un remplacement, les sous-chaînes identifiées sont désignées par **$** suivi du numéro afférent au sous-motif correspondant (pour la chaîne complète, **$&**, attention : **$&** et non **$0**) On peut donc réutiliser une sous-chaîne identifiée dans une chaîne de remplacement. On numéroter jusque 99 parenthèses ouvrantes.

exemple 1 :

On cherche à vérifier si une adresse mail usuelle est correcte ou non. Pour des raisons liées à l'exposé, on a utilisé la méthode **match()** et on a surparenthésé ce qui permet de décomposer la reconnaissance.

On rappelle qu'on considère une adresse mail valide par le fait qu'elle est constituée des «segments d'identification» commençant par une lettre ASCII et que les chiffres, l'undescore, le tiret sont ensuite acceptés. L'adresse se termine par un indicatif de pays (**.fr**) ou de genre (**.com**), avec au moins deux caractères et constitué de lettres.

```
1.<script type="text/javascript">
2.    var chnEntree ="jean-fr.mercier@orange.fr" ;
3.    var chnOblg = "([a-z_][\\w-]*)";
4.    chnFac = "((\\.([a-z_][\\w-]*))*)";
5.    chnPays = "(\\.[a-z]{2,})";
6.    chn = "^"+chnOblg+chnFac+"@"+chnOblg+chnFac+chnPays+"$";
7.    var motif= new RegExp (chn);
8.    alert (motif.source);
9.    alert (chnEntree.match(motif));
10./script>
11.<!-- h14015_re.html -->
```

```
^([a-z_][\w\-]*)((\.([a-z_][\w\-]*))*)@([a-z_][\w\-]*)((\.([a-z_][\w\-]*))*)(\.[a-z]{2,})$
```

```
0     "jean-fr.mercier@orange.fr"
1     "jean-fr"
2     ".mercier"
3     ".mercier"
4     "mercier"
5     "orange"
6     ""
7     undefined
8     undefined
9     ".fr"
index 0
input "jean-fr.mercier@orange.fr"
```

* noter le double slash pour un premier échappement de l'antislash dans la chaîne argument. Par contre, le signe tiret **-** n'a pas à être échappé.

exemple 2 : mode de résolution de **\i**

```
1.<script type="text/javascript">
2.    var chnEntree =
3."un \"fleuriste\" qui n'aime par la \"rose\" ni"   l'\"églantine\" sauvage";
4.    var motif= /(['"])(.*?)\1/g;
```

```
5.    alert (chnEntree.replace(motif,"«$2»"));
6.</script>
7.<!-- h14016_re.html -->
```

```
un «fleuriste» qui n«aime par la "rose" ni l»«églantine» sauvage
```

* ligne 4 : noter en premier lieu la recherche non gourmande. Sinon, la résolution de **\1** se serait faite sur la quote double après **églantine**. La quote simple de **n'aime** est résolue avec la quote simple de **l'églantine**, et la chaîne intermédiaire est reprise à l'identique.

4.2. la méthode de chaîne `replace` avec utilisation des sous-chaînes trouvées.

Le second argument de **replace** peut comporter des sous-chaînes reconnues, mais en plus peut être une fonction qui calcule la chaîne de remplacement.

exemple :

Remplacer dans un texte les guillemets droits (quote double) par les guillemets français.

```
1.<script type="text/javascript">
2.    var chnEntree ='"Java" apparaît en 1993, "JavaScript" un an plus tard.\n'+
3.'Il s\'appelle "LiveScript", et tente de profiter de la notoriété de "Java",\n'+
4.'non pas "java", le «petit noir», ni «lajavanaise» de Gainsbourg \n'+
5.'ni l\'île de "Java", mais le nouveau langage dont on commence à parler.';
6.    var motif = /"([^"]*)"/gmi;
7.    var resultat = chnEntree.replace(motif, "«$1»") ;
8.    alert (chnEntree+"\n\n"+resultat) ;
9.</script>
10.<!-- h1409_re.html →
```

```
"Java" apparaît en 1993, "JavaScript" un an plus tard.
Il s'appelle "LiveScript", et tente de profiter de la notoriété de "Java",
non pas "java", le «petit noir», ni «lajavanaise» de Gainsbourg
ni l'île de "Java", mais le nouveau langage dont on commence à parler.

«Java» apparaît en 1993, «JavaScript» un an plus tard.
Il s'appelle «LiveScript», et tente de profiter de la notoriété de «Java»,
non pas «java», le «petit noir», ni «lajavanaise» de Gainsbourg
ni l'île de «Java», mais le nouveau langage dont on commence à parler.
```

4.3. utilisation du dollar dans les chaînes `replace`.

$1, $2 … $99	sous -chaînes d'ordre 1, 2, … 99 ; **$** est supposé suivi de un ou deux chiffres.
$&	sous-chaîne correspondant à la RegExp
$`	texte à gauche de la sous-chaîne
$'	texte à droite
$$	dollar littéral

5. Les méthodes.

5.1. La méthode `match()` des instances de `String` sans l'attribut `g`

Recherche la première correspondance avec la RegExp.

Retourne un objet contenant les résultats de la correspondance ; **null** si rien n'est trouvé

Les propriétés **"0", "1", "2"** et correspondent à **$&, $1, $1** etc.

La propriété **index** donne la position de la sous-chaîne trouvée.

La propriété **input** donne la chaîne de recherche.

Il y a la propriété de tableau prédéfinie **length**.

exemple :

```
1.<script type="text/javascript">
2.    var chnEntree = "azertaaabbbccdddddeeeqsdfg" ;
3.    var motif = /a+(b+(c+)(d+))e+/ ;
4.    var resultat = chnEntree.match(motif) ;
5.    for(x in resultat)
6.        alert (x+"  "+resultat[x]);
7.</script>
8.<!-- h14010_re.html -->
```

```
0           aaabbbccdddddeee
1           bbbccddddd
2           cc
3           ddddd
index       5
input       azertaaabbbccdddddeeeqsdfg
```

5.2. La méthode `match()` des instances de `String` avec l'attribut `g`

Recherche les correspondances successives dans toute la chaîne.

Retourne un tableau des sous-chaînes trouvées, mais **pas les détails** correspondant aux parties parenthésées. Il n'y a pas de propriété **index** ou **input**. Il y a toujours la propriété prédéfinie **length**.

exemple :

```
1.<script type="text/javascript">
2.    var chnEntree ="CaaabBbBaVa";
3.    var motif = /a?[^A-Z]/g;
4.    chnSortie = chnEntree.match(motif);
5.    alert (chnSortie) ;
6.</script>
7.<!-- h1403_re.html →
```

```
0     "aa"
1     "ab"
2     "b"
3     "a"
4     "a"
```

5.3. La méthode `search()` des instances de `String`.

Retourne la position de la première sous-chaîne trouvée ; **-1** si rien n'est trouvé.

La méthode ignore l'attribut **g**.

5.4. La méthode `split()` des instances de `String`.

La méthode accepte comme premier argument une chaîne, mais aussi une expression régulière.

Attention à la particularité suivante : les sous-motifs parenthésés se retrouvent dans le tableau :

exemple :

```
1.<script type="text/javascript">
2.    var chnEntree ="bonjour <b>le monde</b>, Bonjour <i>JoJo</i>";
3.    var motif = /(<[^>]+>)/;
4.    var chnSortie = chnEntree.split(motif);
5.    alert (chnSortie) ;
6.</script>
7.<!-- h1411_re.html →
```

```
0      "bonjour "
1      "<B>"
2      "le monde"
3      "</B>"
4      ", Bonjour "
5      "<i>"
6      "JoJo"
7      "</i>"
8      ""
```

5.5. La méthode `replace()` des instances de `String`.

Le second argument peut être une fonction, son paramètre étant la sous-chaîne trouvée.

exemple :

Mettre en majuscules tous les mots d'un texte (ouest européen).

Les attributs sont **`g`** (recherche dans tout le texte) et **`i`** (ne pas tenir compte de la casse).

```
1.<script type="text/javascript">
2.    var chnEntree ='"java" apparaît en 1993, "JavaScript" un an plus tard.\n'+
3.'Il s\'énonce "LiveScript", et tente de profiter de la notoriété de "java",\n'+
4.'non pas "java", le «petit noir», ni «lajavanaise» de Gainsbourg \n'+
5.'ni l\'île de "Java", mais le nouveau langage dont on commence à parler.';
6.    var motif = /[A-Z\u00C0-\u00FF]+/gi;
7.    var fnRpl = function(chn){
8.       return chn.substring(0,1).toUpperCase()+chn.substring(1)
9.    };
10.   var resultat = chnEntree.replace(motif, fnRpl) ;
11.   alert (resultat) ;
12.</script>
13.<!-- h1412_re.html -->
```

```
"Java" Apparaît En 1993, "JavaScript" Un An Plus Tard.
Il S'Énonce "LiveScript", Et Tente De Profiter De La Notoriété De "Java",
Non Pas "Java", Le «Petit Noir», Ni «Lajavanaise» De Gainsbourg
Ni L'Île De "Java", Mais Le Nouveau Langage Dont On Commence À Parler.
```

5.6. La méthode `test()` des instances de `RegExp`.

syntaxe : **`expRegExp.test ( chaîne).`**

Retourne **`true`** si l'expression régulière peut être identifiée dans chaîne, **`false`** sinon.

5.7. La méthode `exec()` des instances de `RegExp`.

syntaxe : **`expRegExp.exec ( chaîne).`**

`exec()` est la méthode à tout faire des Expressions Régulières, mais elle est plus complexe à utiliser que les méthodes de chaîne. En effet, elle est destinée à être appelée de façon itérée, avec un drapeau **`g`** posé ; à chaque itération, un paramètre, **`lastIndex`**, se met à jour qui indique où peut commencer la recherche suivante.

* **si l'attribut g n'est pas posé** :

une seule recherche est effectuée ; si aucune correspondance n'est trouvée la fonction retourne **`null`**. Sinon, elle retourne un objet **`Array`** qui est similaire à celui de la méthode **`match()`** avec une recherche non globale.

exemple :

```
1.<script type="text/javascript">
2.    var chnEntree = "000abbbccdddddeee222aaabbcddeee6666";
3.    var motif = /a+(b+(c+)(d+))e+/;
4.    var resultat = motif.exec(chnEntree);
5.    for(x in resultat)
6.    alert (x+"  "+resultat[x]);
7.</script>
8.<!-- h14013_re.html -->
```

```
0           abbbccdddddeee
1           bbbccddddd
2            cc
3           ddddd
index       3
input       000abbbccdddddeee222aaabbcddeee6666
```

*** si l'attribut g est posé :**

exemple :

```
1.<script type="text/javascript">
2.    var chnEntree = "000abbbccdddddeee222aaabbcddeee6666";
3.    var motif = new RegExp ("a+(b+(c+)(d+))e+","g");
4.    while (true) {
5.        var resultat = motif.exec (chnEntree) ;
6.        if (resultat == null) break ;
7.        var chn ="lastIndex"+" \t"+motif["lastIndex"]+"\n" ;
8.        for(var x in resultat)
9.            chn+=x+" \t\t"+resultat[x]+"\n";
10.       alert (chn) ;
11.</script>
12.<!-- h14014_re.html -->
```

```
lastIndex   17
0           abbbccdddddeee
1           bbbccddddd
2           cc
3           ddddd
index       3
input       000abbbccdddddeee222aaabbcddeee6666
```

```
lastIndex   31
0           aaabbcddeee
1           bbcdd
2           c
3           dd
index       20
```

```
input        000abbbccdddddeee222aaabbcddeee6666
```

* Les propriétés énumérables sont identiques au cas précédent ;

* la différence essentielle est la propriété prédéfinie **`lastIndex`** de l'objet **`RegExp.`** Dans le cas où l'attribut **`g`** est posé, **cette propriété est gérée** :

- à chacun des appels de **`exec()`**, elle indique sur quel caractère commencer la recherche ;

- à la fin d'une recherche, **`lastIndex`** indique une position après le dernier caractère reconnu. C'est le caractère 0 en cas d'échec.

- la propriété appartient à l'objet **`RegExp`** : par défaut, lors de la création de l'objet, il est mis à 0. Mais, si on utilise le même objet **`RegExp`** appelant **`exec()`**, la mise à jour est automatique. Il ne faut donc pas oublier, le cas échéant, **d'initialiser cette propriété** (si on veut commencer l'examen ailleurs qu'au début de chaîne, ou si une analyse précédente n'a pas été menée au terme).

6. L'alternative.

6.1. l'un ou l'autre.

Deux sous-motifs peuvent être choisis en alternative : si le premier ne peut être identifié à une sous-chaîne dans une chaîne, on essaie alors le second. Le signe de l'alternative est **`|`**

6.2. exemple.

```
1. <script type="text/javascript">
2.     alert = console.log ;
3.     var chnEntree ='Java apparaît en 1993, JavaScript un an plus tard.\n'+
4. 'Il s\'appelle LiveScript, et tente de profiter de la notoriété de Java,\n'+
5. 'non pas java, le \"petit noir\", ni lajavanaise de Gainsbourg \n'+
6. 'ni l\'île de Java, mais du nouveau langage dont on commence à parler.';
7.     var motif= /\bjava\w*?|\w*?Script\b/ig;
8.     var chnSortie = chnEntree.replace(motif,"«$&»")
9.     alert (chnEntree+"\n\n"+chnSortie);
10.</script>
11.<!-- h14017_re.html -->
```

```
Java apparaît en 1993, JavaScript un an plus tard.
Il s'appelle LiveScript, et tente de profiter de la notoriété de Java,
non pas java, le "petit noir", ni lajavanaise de Gainsbourg
ni l'île de Java, mais du nouveau langage dont on commence à parler.

«Java» apparaît en 1993, «Java»«Script» un an plus tard.
Il s'appelle «LiveScript», et tente de profiter de la notoriété de «Java»,
non pas «java», le "petit noir", ni lajavanaise de Gainsbourg
ni l'île de «Java», mais du nouveau langage dont on commence à parler.
```