Le langage

Javascript

Partie 1

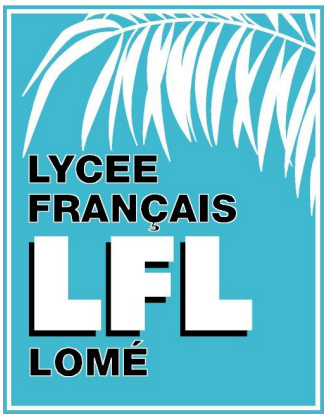

Table des matières

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt

I. Présentation

Le langage **JavaScript**a été inventé par Netscape en 1995. Pour contre carré son adversaire, Microsoft a mis au point un langage similaire appelé *JScript*. Comme pour le HTML, la gueurre des formats vit le jour : chaque constructeur apporta sa petite innovation, empêchant ainsi une standardisation dès son origine.

Une première standardisation du langage vit le jour en 1996 posant les bases communes aux deux langages mais les différences persistent dans l'interprétation du code par les différents navigateurs, obligeant le programmateur à adapter son code aux différents navigateurs

Voici quelques caractéristiques de ce langage :

- Le code d'un script **JavaScript** est directement inclu dans la page Web et exécuté par la machine du client : on parle alors de langage "*côté client*".
- **JavaScript** est un langage peu typé : le contenu d'une variable peut successivement changer de type de données *(numérique, chaîne de caractère...)*. Une variable contenant inialement un nombre peut par la suite contenir une chaîne de caractère.
- **JavaScript** est sensible à la casse : le nom des variables et des fonctions est sensible au changement de minuscules/majuscules dans un nom :

  `compteur` ; `COMPTEUR` ; `comPteur`

  représentent trois noms de variables distincts.
- **JavaScript** est un langage événementiel : en réaction à certaines actions du client *(action des boutons de la souris, enfoncement des touches du clavier,...)*, du code **JavaScript** peut être exécuté répondant ainsi à l'action solicitée.
- **JavaScript** est un langage orienté objet : la programmation en **JavaScript** consiste dans la manipulation d'objet. La fenêtre du navigateur contenant plusieurs objets
  - La barre des titres,
  - La barre d'état,
  - Le contenu de la page qui elle-même contient d'autres objets tels que les images, les paragraphes, la description des changements de fontes...

  Ainsi, la manipulation d'une page Web via un script **JavaScript** consiste en la manipulation d'objets et de sous-objets.

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt

II. L'insertion du c

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt*A. Le principe :*

Il existe deux manières d'insérer du code **JavaScript** dans une page Web :

- le code est tapé directement au milieu du code HTML de la page,
- le code est inséré dans un fichier extérieur et une référence à ce fichier est placé à l'intérieur de la page Web voulant exécuté ce code.

Dans les deux cas, on utilise l'élément HTML script pour indiquer l'insertion du code **JavaScript** :

```
Pour insérer le code directement dans la page :
  <script language="javascript">
    ...Mettre le code ici
  </script>

Pour appeler un fichier externe de script :
  <script type="text/javascript" src="urlFichier">
  </script>
```

Le second exemple montre comment appeler un fichier externe : on utilise l'attribut *src* pour indiquer l'emplacement du fichier cible contenant le code à exécuter. Le contenu de la balise script est vide : son contenu n'est alors pas exécuté.

Cette balise peut être placé à deux endroits dans une page Web :

- Au sein de l'élément head : le code est chargé et exécuté avant l'affichage de la page. A ce moment l'élément body n'existe pas et aucun élément n'est affiché, on place de préférance à cet endroit les fonctions utilisées dans le reste du document.
- Au sein de l'élément body : le navigateur recevant au fur et à mesure les éléments de la page Web, le code **JavaScript** sera exécuté au fur et à mesure de son chargement dans le navigateur.

Contrairement à l'impression donnée dans ce paragraphe, le code **JavaScript** ne s'exécute pas uniquement au moment de son chargement : **JavaScript** est un langage évenementiel.
Le code s'exécutera également en réaction aux actions du client *(mouvement de souris, action des boutons de la souris ou des touches du clavier...)*.

III. Notions de base du noy

A. Introduction :

Depuis 1996, **JavaScript** connait un début de standardisation : elle concerne uniquement le fondement du langage *(le noyau)* : règle syntaxique, définition d'une variable, définition des types de variable de base...

Cet standardisation est assurée par l'ECMA *(European Computer Manufacturers Association)* sous la référence ECMA-262.

Comme il a été dit précédemment, cette standardisation ne touche que le noyau du langage et de nombreuses différences dans l'interprêtation d'un code existe entre les navigateurs même si ces différences tendent à s'effacer avec les nouveaux navigateurs.

Le langage **JavaScript** tel que décrit dans la spécification ECMA-262 est utilisée dans deux autres langages :

- L'actionScript utilisé comme langage de programmation dans les animations Flash.
- Le javascript utilisé dans les fichiers pdf.

Ces trois langages partagent donc les même définitions de base. Seul les objets manipulés changent d'un langage à d'autres :

- Avec l'actionScript, on manipule des animations,
- Avec **JavaScript** dans un navigateur, on manipule des objets HTML *(paragraphes, images. . .)*

Dans ce paragraphe, nous allons voir les bases communes à tous ces langages **JavaScript** :

- Les règles syntaxiques et grammaticale.
- Les boucles de contrôles.
- Les objets définis dans le noyau.

f-javascript/03-noyauJavascript/

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt*B. Les règles syntaxiques :*

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt1. *Un langage sensible à la casse :*

Il a déjà été précisé que ce langage est sensible à la différence entre minuscules et majuscules dans les identifiants *(les noms)* des variables et des fonctions. On dit alors que ce langage est **sensible à la casse**.

Les trois noms suivants représentent des variables distinctes :

```
this ; This ; THIS
```

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt2. *Les mots réservés :*

Certains mots sont déjà utilisés par **JavaScript**. Ainsi, vous ne pourrez pas vous servir des mots suivants pour nommer vos fonctions et variables :

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

Voici les mots dont **JavaScript** se réserve dans un futur proche l'utilisation :

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

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt3. *Les identifiants :*

Une variable ou une fonction est identifiée à l'aide d'un nom. Celui-ci peut contenir des lettres *(non accentuées)*, les signes $ ou _ mais également des chiffres *(ces derniers ne peuvent pas se trouver en début de nom)*

Il est bon, ici, de rappeler que le langage **JavaScript** est sensible à la casse.

Bien que peu utile, il est à noter qu'un identifiant accepte des caractères unicode. Ceux-ci s'écrivent sous la forme :

`\uxxxx`

où `xxxx` représente un entier en notation de héxadécimal codé sur 2 octets *(quatre chiffres héxadécimal)*.

Identifiants valides	*Identifiants incorrects*
nombrePas , i3 , phrase_2 , thom\u0061s	3pas , phrase.vraie , phrasé

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt4. *Les nombres :*

Dans la spécification, ECMQ-262, **JavaScript** utilise pour représenter les nombres de la norme "*Double-precision 64 bit de IEE-754*".

Soyons un peu plus clair :

Un nombre est codé sur 64 bits :

- Le bit de poid fort est utilisé pour le signe du nombre,
- les 11 bits suivants pour l'exposant du nombre représentant une valeur de -126 à +127,
- les 52 bits de poids faible restant sont utilisés pour la mantisse du nombre : représentant l'écriture d'un nombre décimal de l'intervalle $[0\,;\,1[$ en écriture binaire.

Cette représentation offre les caractéristiques suivantes :

- les nombres strictement positifs sont compris dans l'intervalle $\left[4{,}9\times10^{-324}\,;\,1{,}8\times10^{308}\right]$,
- dans le système décimal, on obtient une précision d'au moins 15 chiffres.

Ce codage permet aussi de coder les "*nombres*" $+\infty$ et $-\infty$ ayant pour représentation respectivement `Infinity` et `-Infinity`.
Lorsque le résultat obtenu n'est pas un nombre, une valeur particulière est retournée, représentée par `NaN` *(Not A Number)* : par exemple, pour $\log(-1)$.

Voici les différentes écritures et interprêtation que fait **JavaScript** sur les nombres :

- Une suite de chiffres représente un nombre exprimé en notation décimale avec le point comme séparateur de la partie entière et décimale. Les écritures suivantes représentent des nombres décimaux :

  `3.513` ; `.56` ; `345`
- On exprimera un nombre à l'aidee d'une puissance entière de 10 en utilisant la lettre "E" pour séparer la mantisse de l'exposant. Le nombre 5×10^6 peut indifféremment se coder par :
  - `5e6`
  - `5E6`
  - `5 000 000`
- Pour écrire un nombre en base héxadécimale, on utilise la séquence `0x...` suivit de l'expression du nombre dans cette base. Par exemple, le nombre décimal 687 s'écrit $2AF$ en base héxadécimale et se code en **JavaScript** indifféremment par :
  - `0x2AF`
  - `687`

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt5. *Les chaînes de caractères :*

On définit une chaîne de caractères en encadrant les caractères formant cette chaîne par deux apostrophes ' ou alors par deux guillemets ".

Tout langage de programmation impose des restrictions quant à la manière d'écrire une chaîne de caractères : à cause de certains caractères ayant un status spécial dans le langage ou à cause des règles syntaxiques du langage.

Voici quelques contraintes dans l'écriture d'une chaîne de caractères en **JavaScript** :

- En **JavaScript**, une instruction doit être écrite sur une seule ligne.
  Ainsi, on ne peut inclure directement un saut de ligne dans une chaîne de caractères.

- La barre anti-oblique est utilisée pour un usage interne : plus précisemment, elle sera utilisée pour écrire les séquences d'échappement *(voir plus loin)*.

- On ne peut utiliser une apostrophe dans une chaîne de caractères dont l'apostrophe est également utilisé comme délimiteur. La séquence suivante provoquera une erreur de syntaxe de la part de **JavaScript** :

```
'Il l'a pris'
```

Pour comprendre la suite, il faut déjà comprendre que pour un ordinateur, le saut de ligne est un caractère comme un autre, qui, à l'écran se notera par un retour à la ligne !
Un saut de ligne est le dixième caractère de la table d'encodage ASCII. La barre contre-oblique permet d'utiliser les tables de codage des caractères :

- Pour la table ASCII :

  - En héxadécimal, le nombre dix est représenté par A, on accédera au saut de ligne par le code :

```
\x0A
```

  - En notation octale, le nombre dix est représenté par 12, on accédera au saut de ligne par le code :

```
\012
```

- Pour l'unicode UTF-16, on représente le caractère de saut de ligne par la séquence suivante :

```
\u000A
```

Ainsi, la barre contre-oblique est un caractère spécial permettant d'introduire n'importe quel caractère de la table ASCII courante ou de la table unicode UTF-6.

Voici quelques raccourcis des caractères d'échappement le plus souvent utilisé que nous offre **JavaScript**. Remarquez l'utilisation du caractère spécial \ initiant chaque séquence d'échappement :

Caractère d'échappement	Code Unicode	Désignation	Abbrévation
\b	\0008	backspace	$< BS >$
\t	\0009	Tabulation horizontale	$< HT >$
\n	\u000A	Nouvelle ligne	$< LF >$
\v	\u000B	Tabulation verticale	$< VT >$
\f	\u000C	Form feed	$< FF >$
\r	\u000D	Retour chariot	$< CR >$
\"	\u0022	Guillemet double	"
\'	\u0027	Guillement simple	'
\\	\u005C	barre oblique inversé *(backslash)*	\

Sous Window, pour utiliser un retour à la ligne, il faut combiner les deux caractères "*Retour chariot*" et "*Nouvelle Ligne*". Ainsi, pour introduire un saut de ligne dans une chaîne de caractères, il faut insérer une des deux séquences équivalentes suivantes :

- \r\n
- \u00D\u00A

Voici quelques chaînes de caractères comportant des erreurs de syntaxes et leurs équivalents bien codés :

Expression fausse	Expression correcte
'Salut l'ami'	'Salut l\'ami' *ou alors* "Salut l'ami".
"Salut Ca va ?"	"Salut\r\n Ça va ?"
"C :\MonDossier\"	"C :\\MonDossier\\"

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt6. *Délimiter les intructions :*

Pour signaler la fin d'une instruction, on utilise le point virgule " ;".

En **JavaScript**, une instruction doit être écrie sur une même ligne : **JavaScript** interprête tout saut de ligne comme la fin d'une instruction et le point-virgule n'est plus obligatoire.

La manière dont **JavaScript** rajoute les points virgules est un peu compliqué. Les deux règles citées précédemment sont largement suffisantes pour écrire correctement du code.

Les deux parties du code ci-dessous sont totalement équivalentes :

```
var a=1; c=a+b;

var a=1
c=a+b
```

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt7. *Les blocs d'intructions :*

Un bloc d'instructions est utilisé pour délimiter un jeu d'instructions utilisé pour :

- déclarer le corps d'une fonction,
- délimiter les instructions se répétant plusieurs fois pour les boucles `for`, `while`, `do`,
- les différents blocs d'exécution dans les branchements conditionnels `if...else...`, `switch`

Le code se trouvant à l'intérieur s'exécutera séquentiellement : les instructions seront exécutés l'une après l'autre dans l'ordre de leur apparation.
Seul les mots :

- `return` pour les fonctions,
- `break` pour le branchement conditionnel `switch(){...}`,
- `continue` pour les boucles, permettent d'arrêter l'exécution séquentielle d'un bloc *(mais n'arrête pas l'exécution de la boucle)*.

```
<script type="text/javascript">
var texte="monsieur";
var Texte="homme";
function affiche(){
  var texte="madame";
  document.write("1-"+texte+"<br>");
  Texte="femme";
  document.write("2-"+texte+"<br>");
}
affiche();
document.write("3-"+texte+"<br>");
document.write("4-"+Texte+"<br>");
for(i=0;i<1;i++){
  var texte="mademoiselle";
  document.write("5-"+texte+"<br>");
}
document.write("6-"+texte);
</script>
```

L'exemple ci-dessus présente deux groupes d'instructions : celui de la déclaration de la fonction `affiche` et celui de la boucle `for`.

L'étude de cet exemple met en évidence un phénomène particulier quant à l'influence des variables définies dans l'intérieur du corps d'une fonction sur les variables du reste du programme. Après l'appel à la fonction `affiche` :

- la variable `texte` n'a pas changé de valeur ;
- la variable `Texte` a subit l'affectation `Texte="femme"` du corps de la fonction.

Ce phénomène s'appelle la "*portée des variables*" et distingue les variables locales et variables globales définies dans le corps d'une fonction.

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt8. *Les commentaires :*

Lorsqu'on tape un code assez long, il est habituel de laisser des commentaires dans celui-ci pour qu'au cours d'une prochaine maintenance du code, la structure du code soit plus facilement compréhensible.

Ces commentaires sont faits pour être lu par le programmateur et ne pas être exécutés par **JavaScript**.

Il existe deux manières d'introduire des commentaires dans du code :

- Pour insérer des commentaires sur une ligne ou en bout de ligne, on précède ceux-ci d'une double barre contre-oblique.
- Pour insérer des commentaires sur plusieurs lignes, on entoure les commentaires de la séquence ouvrante /* et fermante */.

f-javascript/03-noyauJavascript/

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt*C. Les variables. :*

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt1. *Généralité :*

Une variable en informatique est une partie de l'espace mémoire représentant un objet *(nombre, tableau, chaîne de caractères...)* reliée par un identifiant *(son nom)*.

En **JavaScript**, l'identifiant suit les règles suivantes *(déjà citées)* :

- l'identifiant d'une variable peut contenir tous les caractères alphabétiques non-accentués, les signes $ et _, ainsi que tout caractère au format Unicode ; les chiffres sont admis mais pas en début d'identifiant.
- l'identifiant ne doit pas être un des mots réservés à l'usage de **JavaScript**.

Les trois temps de vie d'une variable sont :

- la *déclaration)* : le mot-clé `var`, bien que facultatif, permet de déclarer une nouvelle variable dans le code. **JavaScript** étant un langage faiblement typé, on ne définit pas nécessairement la nature de son contenu lors de la déclaration.
- l'*affectation* : à l'aide de l'opérateur "=", on peut affecter une valeur à la variable.
- la *destruction* : lors de la fin du script, la variable sera détruite et la mémoire sera ainsi libérée.

Il est fréquent que la déclaration et l'affectation soient effectuées en même temps en **JavaScript**. Voici quelques exemples de déclaration/affectation en **JavaScript** :

```
1  //Déclaration typée sans affectation
2  var a=new String();
3
4  //Déclaration typée puis affectation
5  var a=new String();
6  a="Bonjour";
7
8  //Déclaration typée avec affectation
9  var a=new String("Bonjour");
10
11 //Déclaration non-typée puis affectation
12 var a;
13 a="Bonjour";
14
15 //Déclaration non-typée et affectation
16 var a="Bonjour";
17
18 //Déclaration non-typée et affectation
19 a="Bonjour";
```

Une variable déclaré sans avoir reçu d'affectation aura sa valeur égale à `undefined`

Dans le corps du script, l'utilisation du mot-clé `var` pour la déclaration d'une variable est facultative.
Dans le corps d'une fonction, l'abscence ou la présence du mot-clé `var` aura une signification particulière : elle permettra de définir la variable localement au corps de la fonction ou globale à l'ensemble du script : voir le chapitre sur les fonctions pour voir cette différence subtile sur la durée de vie *(sa portée)* d'une variable dans le corps d'une fonction.

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt2. *Les valeurs particulières des vari*

- `undefined` : une variable aura cette valeur notamment lorsqu'elle aura été déclarée mais

pas encore affectée d'une valeur.

- `null` : une variable ayant cette valeur a été affecté mais par quelques choses sans valeur, sans référence.
  Tant que l'élément body n'a pas été déclaré dans la page, l'objet `document.body` aura la valeur `null` : il existe pour **JavaScript** mais n'a pas d'existence formelle dans la page.
- `NaN` *(Not-a-Number)* : cette valeur de type nombre indique que cette valeur n'est pas un nombre ! Par exemple la valeur retournée par `sqrt(-1)` sera `NaN` : cette valeur est important pour la gestion d'erreur lors de calcul à l'intérieur du script.
- Lors d'un calcul, lorsque **JavaScript** dépasse sa capacité *(par exemple* $2^{10\,000}$*)*, une des deux valeurs suivantes est retournée : `-Infinity` et `+Infinity`).
  `+Infinity` et `Infinity` sont des synonymes.

Remarquons que **JavaScript** fait également la différence entre $0+$ et $0-$.

f-javascript/03-noyauJavascript/

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt*D. Les objets :*

Comme la plupart des langages de programmation, une grande partie de la programmation **JavaScript** est séquentielle et procedural :

- les intructions sont effectuées les unes après les autres dans le sens de lecture.
- certaines parties du programme sont placés dans des procédures *(fonctions)* afin d'être utilisées à plusieurs endroits du programme.

JavaScript est également un langage de programmation orienté objet *(POO)*. Il faut s'imaginer que la fenêtre de votre navigateur est un objet pour **JavaScript** :

- que cette objet contient des *propriétés* qui peuvent être :
  - des valeurs telles que sa position actuelle sur l'écran ou sa hauteur...
  - des sous-objets telles que le contenu Web de cette fenêtre, l'historique de navigation du client...
- des *méthodes (c'est à dire des fonctions)* sont attachés à cet objet permettant de manipuler cet objet et son contenu.

La standardisation ECMA-262 définit les objets du noyau de **JavaScript**. Pour les objets propres aux navigateurs, un consensus existe.
Les objets du noyau tels que les objets de type chaîne de caractère, de type nombre, de type tableau ... seront présentés dans la seconde partie de cette formation.
Les objets propres aux navigateurs seront présentés dans la troisième partie de la formation.

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt1. *Accéder aux propriétés et aux mé*

En **JavaScript**, tout doit être imaginé en termes d'objet. La fenêtre du navigateur, représenté par l'identifiant `window`, est un objet. `window` contient des sous-objets créés automatiquement dès l'ouverture du navigateur :

- `document` : le contenu de la page ;
- `location` : information sur l'adresse actuelle du navigateur ;
- `history` : l'historique de navigation du client.

On accédera à ces sous-objets en les faisant précéder de l'objet parent `window` et en les séparant d'un point :

```
window.document ;  window.location ;  window.history
```

L'objet `document` contient également des objets *(ses sous-objets)* tels que :

- `forms` : représente l'ensemble des formulaires présents dans la page ;
- `images` : représente l'ensemble des images présentes sur la page.

On accédera ainsi à ces objets en écrivant :

```
window.document.forms ;  window.document.images
```

L'objet `window` est considéré comme l'objet "*Global*", il est la référence de toute chose, ainsi le nommer est facultatif. On aurait pû également appeler les anciens objets en écrivant :

```
document.forms
```

Les méthodes sont des fonctions propres à un objet et permettant de le manipuler. Par exemple, l'objet `window` contient les méthodes `moveBy(...)` et `alert()` permettant respectivement de déplacer la fenêtre du navigateur et d'afficher une avertissement dans une boîte à dialogue.
On appelle ces méthodes en les faisant précéder également de leur objet et en les séparant d'un point :

```
window.moveBy(6,4) ;  window.alert("Fin du chargement")
```

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt2. *Le constructeur et le préfixe* `new`

Nous rentrons ici un peu plus dans le fonctionnement orientée objet de **JavaScript**.

Le noyau de **JavaScript** contient un certain nombre d'objet "*prototype*" : parmi `Number`, `String`, `Array`, `Date` dont la signification se passe de commentaire. On dit prototype car chacun d'eux contient la structure de base *(propriétés et méthodes)* de chaque objet de ce type.

Tout objet de type *string* possèdera des propriétés *(telle que sa longueur)* et des méthodes *(tel que* `substr` *permettant l'extraction d'une sous-chaîne)* propre à sa nature.

L'objet `String` représente le modèle de chaque variable de type *string*.
Lors de la déclaration/affectation d'une variable de type chaîne de caractère lors d'une instruction du type :

```
var chaine="Bonjour";
```

JavaScript va effectuer une copie de l'objet `String` et placé `Bonjour` dans son contenu ; la variable `chaine` possédera toutes les propriétés et méthodes inhérantes à son type *string*.

JavaScript a crée une **instance** de l'objet `String` et l'a appelé `chaine`

Pour déclarer explicitement une instance de l'objet `String`, on utilise la séquence :

```
var chaine=new String();
```

Puis, on effectue séparemment l'affectation :

```
chaine="Bonjour";
```

On peut également déclarer et affecter une nouvelle instance de cette objet :

```
var chaine=new String("Bonjour");
```

La méthode `String(...)` est le constructeur de l'objet, on la fait précéder du mot-clef `new` pour créer une instance *(une copie)* du modèle `String`.

Nous avons vu que **JavaScript** nous permet d'éviter de telle déclaration , réduisant ainsi dans un premier temps le niveau de difficulté.

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt3. *Les propriétés :*

Les propriétés d'un objet sont les variables rattachées à cet objet.
Elle peuvent être de n'importe quel type. Au moment de sa création *(de sa déclaration typée ou de son affectation)*, un objet hérite des propriétés définies pour son type par défaut.

Par exemple l'objet `document` possède la propriété `title` représentant le titre de la fenêtre du navigateur. On accède à sa valeur en séparant l'objet et sa propriété par un point :

```
document.title
```

On modifie, alors, le titre de la page à l'aide de l'opérateur "=" :

```
document.title="Mon titre";
```

Les propriétés des objets du noyau *(pas celle créer par l'utilisateur)* possèdent des attributs régissant certains de leurs comportements :

- ReadOnly : la valeur de ces propriétés ne peut être modifiée par l'utilisateur. Le signe d'affection "=" n'aura aucun effet sur elles : `var s=new String("Bonjour"); s.length=2;`
  La dernière instruction n'aura aucun effet car la propriété `length` ne peut être modifiée directement par l'utilisateur.
- DontEnum : la propriété ne sera référencée par la commande **for-in**.
- DontDelete : la propriété ne peut être effacée de l'objet.
  L'opérateur `delete` permet d'effacer des propriétés : pas celle affectée de cet attribut.

- Internal : la propriété ne peut être accédée directement par l'utilisateur.

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt4. *Les méthodes :*

Sous **JavaScript**, les objets intégrent des fonctions permettant de le manipuler. Celles-ci s'appellent des méthodes.

Créer une fonction dans le corps du script revient à ajouter une méthode à l'objet `window`.

Un prochain chapitre étudiera plus en détail les fonctions et leur déclaration.

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt5. *L'existence d'un objet :*

Dans le **JavaScript** pour les navigateurs, l'objet `window` représente l'objet Global. Ainsi, tout objet peut vu comme une propriété ou la propriété d'un de ses sous-objets.

Une propriété non-définie *(ni déclarée, ni affectée)* renverra la valeur `undefined`.

```
1  <script type="text/javascript">
2  if(window.a)
3  document.write("oui");
4  else
5  document.write("non");
6
7  var a;
8  if(window.a)
9  document.write("oui");
10 else
11 document.write("non");
12
13 a=2;
14 if(window.a)
15 document.write("oui");
16 else
17 document.write("non");
18 </script>
```

Ce code renverra `nonnonoui`.

Lorsque viendra le chapitre de la conversion des variables, on verra que les valeurs `undefined` et `null` ont pour correspondance pour les booléens la valeur `false`

On peut ainsi vérifier l'existence ou non d'une propriété/d'un objet.
On se sert de ce raisonnement pour déterminer la nature du navigateur utilisé par le client : en sachant que la propriété `document.all` n'existe que pour Internet Explorer, on peut facilement

voir si le navigateur utilisé est IE :

```
<script type="text/javascript">
  if(document.all)
    document.write("Internet Explorer");
  else
    document.write("pas Internet Explorer");
</script>
</script>
```

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt6. *L'objet this :*

Au cours de l'exécution du programme, le mot-clef `this` fait référence à l'objet faisant appel à l'instruction contenant ce mot-clef.
Le corps du script est exécuté par l'objet Global *(pour les navigateurs* `window`*)* ainsi dans le corps du script, `this` sera synonyme de `this`; un autre objet peut lancer l'exécution d'une partie du script notamment lors de l'invocation d'événement.

Le script pourra vous éclaircir sur ce point :

```
<body>
  Le corps de l'affichage
  <script type="text/javascript">
  function a(){
    if(this==window)
      document.title+="oui-";
    else
      document.title+="non-";

    if(this==document.body)
      document.title+="oui-";
    else
      document.title+="non-";
  }

  if(this==window)
    document.title+="yes-";

  a();
  document.body.onclick=a
  </script>
</body>
```

Après avoir cliqué sur les quelques mots contenus dans body, le titre de la page deviendra :

```
yes-oui-non-non-oui-
```

A vous d'interprêter, à chaque fois la valeur de l'objet `this`.

Lors du lancement d'un événement par un objet *(dans le prochain cas un élément HTML)*, `this` permettra de faire référence plus rapidement et facilement à cette élément. En voici un exemple :

```
1 <body bgcolor="#AAAA33">
2  <div onmouseover="javascript:this.style.↩
     backgroundColor='#555555'"
3       onmouseout="javascript:this.style.backgroundColor↩
        ='transparent'">
4    Salut
5  </div>
6 </body>
```

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt7. *Conversions de type :*

Le langage **JavaScript** est un langage peu typé : les variables peuvent changer de types de données facilement. **JavaScript** peut également, au moment de test, comparer deux variables de type différent ! Pour cela, **JavaScript** procèdera à des conversions de type totalement transparent pour le progammeur, mais il est préférable de connaitre ces mécanismes internes.

Il est fréquent en **JavaScript** de :

- d'ajouter un nombre à une chaîne de caractère : implicitement le nombre sera transformé en une chaîne de caractères.
- Une chaîne de caractère peut être passer comme argument à une fonction mathématiques : **JavaScript** essayera de transformer la chaîne en un nombre.

Ces instructions constitueraient une faute grave dans des langages de programmation fort typé comme le C++, java... entraînant l'arrêt de l'exécution d'un programme *(voir son plantage)*. **JavaScript** se charge des changements de type nécessaire pour effectuer les opérations demandées ; ce qui apparait comme un avantage pour la simplification de programmation peut s'avérer dans certains cas comme un handicap pour débugger des programmes en cas de comportement inattendu.

Voici la liste des conversions qu'opère **JavaScript** :

Conversion en booléen :

Type d'entrée	Vers les booléens
`undefined`	`false`
`null`	`false`
Booléen	garde sa valeur d'origine
Nombre	Si l'argument valait +0, -0, NaN alors le résultat sera `false`. Dans les autres cas, la conversion retournera `true`.
Chaîne de caractères	Si la chaîne est vide, la conversion renverra `false` ; sinon elle renverra `true`

- **Conversion en nombres :**

  Cette conversion peut occurer lorsqu'on effectue la multiplication de deux variables. Par exemple, l'instruction suivante est valide pour **JavaScript** :

```
var a=2;var b="5"; document.write(a*b);
```

Type d'entrée	Vers les nombres
`undefined`	`NaN`
`null`	`+0`
Booléen	Si l'argument est `true`, la conversion donnera 1, sinon elle donnera `+0`.
Nombre	Aucune modification

Chaîne de caractères	`"1.45kg"` ⇝ `1.45` `"77.3"` ⇝ `77.3`
	`"077"` ⇝ `77` `"0x77"` ⇝ `119`
	`"0x77.3"` ⇝ `NaN` `".3"` ⇝ `0.3`
	`"0.1e6"` ⇝ `100000`

Voici quelques remarques quant à la conversion d'une valeur en nombre :

- L'instruction :

```
document.title=077;
```

  affichera 63 dans la barre des titre car un nombre commençant par un zéro représente un nombre en base octale.

  L'instruction :

  document.title="077"*1 ; affichera 77 dans la barre des titres ; A l'approche de la multiplication, "077" sera converti en 77 par **JavaScript**.

- Les fonctions globales `parseInt()` et `parseFloat()` permettent de transformer n'importe quel objet respectivement en un entier ou en un nombre décimal. Mais elles liront caractères

après caractères pour en sortir le premier nombre qu'elles voient. Voici quelques conversions surprenantes :

`parseFloat(0x77)`⇝`0` ; `parseInt(0.2e-6)`⇝`2`

`parseInt(010)`⇝`8` ; `parseFloat(010)`⇝`10`

- **Conversion en chaînes de caractères :**

Type d'entrée	Vers les chaînes de caractères
`undefined`	`"undefined"`
`null`	`"null"`
Booléen	`"true"` ou `"false"`
Nombre	Si l'argument est `NaN`, le résultat retourné sera `"NaN"` Pour `+0` et `-0`, retournera `"0"`. Pour une valeur infinie, le résultat de la conversion sera `"Infinity"` Sinon elle retournera le nombre sous son écriture en base 10 dans une chaîne
Chaîne de caractères	retourne la chaîne sans changement

La conversion peut également se faire manuellemenet à l'aide de la méthode `toString()` que possède tout objet **JavaScript**.

f-javascript/03-noyauJavascript/

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt*E. Les fonctions :*

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt1. *Définitions des fonctions :*

Une fonction *(également appelée procédure)* en informatique est le lien créé entre un bloc d'instructions et un identifiant. Ce bloc d'instructions pourra être exécutée à différentes endroits du programme en invoquant simplement son identifiant.

L'utilisation des fonctions procure les avantages suivants :

- éviter de recopier plusieurs fois le même bout de code dans un script ce qui facilite également la maintenance du code.
- simplifier la structure du programme et réduire sa taille.

Une fonction peut recevoir des valeurs *(appelées arguments)* lors de chaque appel, ceci permet de faire varier le comportement de la fonction à chacun de ces appels. Ces arguments pourront être rappelés dans le corps de la fonction à l'aide d'identifiants spécifiés lors de la déclaration de la fonction.

On déclare une fonction à l'aide du mot-clef `function` de la manière suivante :

```
function nomFonction(arg1, arg2,...){
    ...corps de la fonction...
}
```

`arg1`, `arg2`... sont les identifiants des valeurs passés en arguments lors d'un appel à la fonction : `arg1`, `arg2`... sont des variables existant uniquement dans le corps de la fonction. On parlera de variables locales.

Le bloc d'instructions d'une fonction n'est pas exécutées lors de sa déclaration, mais seulement lors de leurs appels. On exécute une fonction par l'instruction suivante :

```
nomFonction(arg1, arg2,...);
```

où `arg1`, `arg2`... seront remplacés, dans le cas de l'appel, par les valeurs passées en argument.

Voici un code HTML définissant un formulaire plutôt simple :

```
<form action="" method="post">
Homme <select name="h">
<option value=0>0
<option value=1>1
<option value=2>2
</select>
<p>
Femme <select name="f">
<option value=0>0
<option value=1>1
<option value=2>2
</select>
</form>
```

On remarque que le code précédent est composé de deux éléments select très ressemblants. Le code suivant montre comment on peut simplifier le code avec l'utilisation des paramètres qui "*personnalisent*" chaque appel à la fonction :

```
<script>
function ecrit(x){
  document.write('<select name="'+x+'"><option value=0>0<option value=1>1<option value=2>2</select>');
<script>

<form action="" method="post">
```

```
Homme <script> ecrit("h"); </script>
<p>
Femme <script> ecrit("h"); </script>
</form>
```

On peut également utiliser l'objet-classe `Function` pour déclarer une fonction. Cette façon de faire est rarement utilisée sauf dans quelques cas précis. Cet objet sera vu dans la deuxième partie de cette formation.

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt2. *Les arguments :*

Lors d'un appel à une fonction, le nombre d'arguments fournis peut ne pas correspondre à celui indiquer lors de la déclaration de la fonction : **JavaScript** n'emmettra aucune erreur.

Normalement, le programmeur doit savoir combien une fonction attend d'arguments. Mais on peut également utiliser cette caractéristique de **JavaScript** afin de créer des fonctions admettant un nombre variable d'arguments.

Pour gérer ce phénomène, **JavaScript** crée à chaque appel d'une fonction, la variable `arguments` propre au corps de la fonction : cette variable est locale au corps de la fonction.

Le type de cette variable est proche d'un tableau. Voici son utilisation :

- `arguments.length` est un nombre représentant le nombre d'arguments passés à la fonction lors l'appel.
- `arguments[i]`, où i est un entier, est la valeur de l'argument de rang `i`, sachant que le premier argument a un rang de 0, le second argument un rang de 1....
  S'il n'existe pas d'éléments de rang `i`, la valeur retournée sera `undefined`.

```
<script>
 function multiplie(arg1,arg2){
   var chaine=arguments.length+" - ";
   chaine+=arguments[0]+" - ";
   chaine+=(arg1*arg2)+"<br>";
   document.write(chaine);
 }

multiplie(2);
multiplie(9,5);
</script>
```

Voici quelques remarques quant à l'exécution de ce script :

- L'exécution du script affichera le résultat suivant :

1 - 2 - NaN
2 - 9 - 45

- On obtient la valeur `NaN` dans le premier appel, car aucun second argument n'a pas été passé.
- Dans le corps de la fonction, `arg1` et `arguments[0]` sont des synonymes.

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt3. *L'instruction "return" :*

L'exemple précédent de la fonction `multiplie` affiche directement le résultat à l'écran. Parfois, une fonction doit retourner une valeur au script afin que celle-ci soit de nouveau utilisable dans le script : on utilise l'instruction `return` suivi de la valeur a renvoyée.

Voici un exemple mettant en avant l'utilisation de l'instruction `return`.

```
1  <script>
2  function alea(){
3    a=Math.random();
4    a=Math.floor(a*5);
5    return a;
6  }
7
8  function affiche(){
9    var a=alea();
10   document.write("Valeur au premier tirage : "+a+"<br>"↩
       );
11   a+=alea();
12   document.write("Valeur après second tirage : "+a+"<br↩
       >");
13   return;
14   document.write("sa valeur");
15 }
16
17 affiche();
18 </script>
```

Voici quelques commentaires sur ce script :

- On déclare deux fonctions :
  - `alea()` qui retourne un entier aléatoire entre 0 et 4 compris.
  - `affiche()` qui simule deux tirages aléatoire à l'aide de la fonction `alea()` et affiche au fur et à mesure la somme des résultats.

On termine le script par un appel à la fonction `affiche()`

- L'instruction `return` arrête l'exécution d'une fonction, ainsi "*sa valeur*" ne sera pas affichée à l'écran

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt4. *Les variables locales et globales :*

Commençons par étudier l'exemple suivant :

```
1  <script>
2  var a=0;
3  var b=0;
4
5  function f(){
6    var a=2;
7    a=3;
8    b=3;
9  }
10
11 document.write(a+" - "+b+"<br>");
12 f();
13 document.write(a+" - "+b);
14 </script>
```

L'exécution de ce script affichera :

0 - 0 0 - 3

On remarque que la déclaration de la fonction `f()` n'a pas changé les valeurs de `a` et de `b` *(normal puisqu'à la déclaration le code n'est pas exécuté)*, mais que son appel n'a modifié que celle de `b`.

Lors de l'écriture d'une fonction, le programmeur peut avoir envie :

- d'utiliser des variables n'interagissant pas avec les autres variables du script ;
- ou tout au contre, il peut vouloir récupérer des informations du reste du programme via des variables.

On parlera alors pour le bloc d'instructions définissant le corps d'une fonction de **variables locales** dans le premier cas et de **variables globales** dans l'autre cas.

Voici comment distinguer chacune d'elles :

- Une variable sera locale si lors de sa déclaration ou de sa première déclaration/affectation dans le corps de la fonction, elle sera précédée du mot-clef `var` : c'est le cas dans l'exemple ci-dessus de la variable `a`.
- Une variable dont l'utilisation a aucun moment ne sera précédé par `var` *(comme la variable*

`b`*)* sera globale et représentera la variable, de même identifiant, existant dans l'ensemble du script.

Lors de l'étude ou du débogage d'un script, il est important d'identifier la nature de chaque variable afin de connaître la durée de vie *(on parle également de porté)* de chaque variable utilisée dans le corps des fonctions.

f-javascript/03-noyauJavascript/

F. Les opérateurs :

1. L'opérateur d'affectations "="

Le signe "=" utilisé en informatique n'est pas là, comme en mathématique, pour indiquer une égalité. Mais l'affectation d'une valeur *(opérande gauche)* à une variable *(opérande droite)*.

On remarquera dans les exemples suivants la présence en préfixe de signes opératoires modifiant le comportement de l'affectation.

`var a=5`	La variable est définie et affecté de la valeur 5. `a` sera de type Number.
`a+=5`	La valeur de la variable `a` sera augmentée de 5.
`a-=5`	La valeur de la variable `a` sera retranchée de 5.
`a*=5`	La valeur de la variable `a` sera multipliée par 5.
`a/=5`	La valeur de la variable `a` sera divisé par 5.
`var b="chaine"`	La variable `b` est définie et affecté de la chaîne de caractère `"chaine"`.
`b+="rajout"`	La variable `b` se vera rajouté en son bout la chaîne `rajout`.

Les opérateurs travaillant sur les nombres peuvent attributà la variable `a` la valeur `NaN` dans le cas où la valeur de `a` ne lui permet pas d'être convertie en un nombre :

```
var a="Bonjour";     a*=2;
```

2. Les opérateurs d'incrémentation

Ce sont les opérateurs classiques d'incémentation ++ ou de décrémentation -- utilisés sur des variables numériques permettant respectivement d'augmenter ou de réduire la valeur de 1.

Ces opérateurs ont un effet différent suivant qu'ils soient placés en préfixe ou en suffixe. Ainsi, si `a` est une variable numérique :

- L'instruction `a++;` renverra la valeur de `a` puis incrémentera la valeur de `a` : après exécution

de l'instruction, `a` aura pour valeur `a`+1.

- L'instruction `++a;` incrémentera la valeur de `a` et renverra la nouvelle valeur : l'exécution de cette instruction renvoit la valeur `a`+1.

Le positionnement d'un tel opérateur influence donc la valeur retournée par l'instruction : la valeur de la variable `a` après l'exécution de l'instruction sera toujours `a+1`.

Voici une illustration de l'utilisation de ces opérateurs :

```
<script>
var i=2;
var j=2;

var chaine="";
chaine += i++;
chaine += " - ";
chaine += i+"<br>";

chaine += ++j;
chaine += " - ";
chaine += j+"<br>";

document.write(chaine);
</script>
```

Le script ci-dessus renverra comme valeur :

```
2 - 3
3 - 3
```

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt3. *L'opérateur % :*

Cet opérateur permet de connaître le reste de la division euclidienne de deux entiers. L'instruction :

```
a% b
```

renvoit le reste de la division euclidienne de `a` par `b`.

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt4. *Les opérateurs de comparaison :*

Les opérateurs de comparaison permettent de comparer deux valeurs et renvoit leur résultat de comparaison sous forme d'un booléen : soit `false`, soit `true`. Ces opérateurs sont utilisés dans les structures conditionnels pour contrôler l'exécution d'un programme.

`a==b`	Renvoit **true** dans le cas où `a` et `b` ont la même valeur *(après une possible conversion de type)*.
`a! =b`	Renvoit **true** dans le cas où `a` et `b` ont une valeur différente *(après une possible conversion de type)*.
`a===b`	Renvoit **true** dans le cas où `a` et `b` ont la même valeur ainsi que le même type.
`a! ==b`	Renvoit **true** dans le cas où `a` et `b` sont de types ou alors de valeurs distinctes.
`a>b`	Renvoit **true** dans le cas où `a` est strictement plus grand que `b`.
`a>=b`	Renvoit **true** dans le cas où `a` est plus grand ou égal que `b`.
`a<b`	Renvoit **true** dans le cas où `a` est strictement plus petit que `b`.
`a<=b`	Renvoit **true** dans le cas où `a` est plus petit ou égal que`b`.
`!a`	La négation de `a` : renverra `true` si après conversion de type, la valeur de `a` vaut `false`.

Des conversions de type sont implicitement effectués par **JavaScript** dans le cas où `a` et `b` ne sont pas de même type *(sauf pour les opérateurs "===" et "!==")*.
Il est inutile de dire que ces opérateurs renvoient `false` dans le cas contrairement explicité.

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt5. *Les opérateurs logiques :*

Les opérateurs logiques permettent de manipuler les booléens : on dispose des deux opérateurs `&&` et `||` correspondant aux opérateur logiques AND et OR.

- L'opérateur logique AND permet de savoir si deux booléens ont la valeur `true` ;
- L'opérateur logique OR permet de savoir si parmi deux booléens au moins un a la valeur `true`.

Voici leurs tableaux de valeurs :

`a&&b`

a \ b	`true`	`false`
`true`	`true`	`false`
`false`	`false`	`false`

`a||b`

a \ b	`true`	`false`
`true`	`true`	`true`
`false`	`true`	`false`

Ainsi, on a :

- `true && false` retourne `false`
- `true || false` retourne `true`

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt6. *Les opérateurs sur les bits :*

JavaScript permet de manipuler les bits *(sur 32 bits)* composant la valeur d'une variable. On ne développera pas trop ce sujet : le niveau d'utilisationde ces fonctions est trop élevé pour cette formation.

- `a& b` : compare bit à bit les valeurs de `a` et de `b` avec l'opérateur AND.
- `a|b` : compare bit à bit les valeurs de `a` et de `b` avec l'opérateur OR.
- `a^b` : compare bit à bit les valeurs de `a` et de `b` avec l'opérateur XOR.
- `~a` : renvoie la négation, bit à bit, de `a`.
- les opérateurs `a << 2` et `a <<< 2` permettent le décalage à gauche avec ou non extensions à l'aide de zéros.
- les opérateurs >> et >>> effectuent un décalage à droite avec ou non extensions à l'aide de zéros.

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt7. *Les opérateurs de concaténation*

L'opérateur binaire "+" sert à concaténer les chaînes de caractères : c'est à dire mettre bouts à bouts deux chaînes de caractères.
Il faut qu'au moins une de ses opérandes soit une chaîne de caractère, si l'autre a un autre type, **JavaScript** procédera implicitement à une conversion.

```
1 <script>
2 var a="Bonjour, ";
3 var b="monsieur ";
4
5 document.write(a+b+"Jacques");
6 </script>
```

Ce script affichera dans la page Web :

Bonjour, monsieur Jacques

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt8. *Quelques opérateurs supplémenta*

- `delete` : cet opérateur permet l'effacement d'une variable : attention, il ne marche pas avec les variables déclarées explicitement à l'aide du mot-clef `var`.

```
1 <script>
2  var a=2;
3  delete a;
```

```
 if (window.a)
   chaine="a existe";
 else
   chaine="a n'existe pas";
 chaine+="<br>";
 b=2;
 delete b;
 if (window.b)
   chaine+="b existe";
 else
   chaine+="b n'existe pas";
 document.write(chaine);
</script>
```

Le script précédent renverra :

a existe

b n'existe pas

Voici quelques remarques pour comprendre le script précédent :

- On utilise `window.a` pour tester si la variable `a` existe.
  La variable `a` est globale, elle est donc une propriété de l'objet globlal `window`.
- L'instruction :
  `if (a) ...1...else ...2...` exécutera :
  - `...1...` si `a` existe,
  - `...2...` si `a` n'existe pas.

- `in` est un opérateur permettant de savoir si un objet possède une propriété : elle renvoie un booléen. Pour savoir si `a` est une propriété de `b`, on utilise l'instruction :
  `"a" in b;`

```
<script>
chaine="title" in window.document;
chaine+=" ; ";

var a=5;
chaine+=a in window; //Renvoie true
chaine+=" ; ";
chaine+="a" in window; //Renvoie false
chaine+=" ; ";

var liste=new Array(5,"banane");
```

```
liste["prenom"]="Jacques";
chaine+="nom" in liste; //Renvoie false
chaine+=" ; ";
chaine+=2 in liste; //Renvoie false
chaine+=" ; ";
chaine+=5 in liste; //Renvoie false
chaine+=" ; ";
chaine+="Jacques" in liste; //Renvoie false
chaine+=" ; ";
chaine+="prenom" in liste; //Renvoie true
chaine+=" ; ";
chaine+="length" in liste; //Renvoie true
chaine+=" ; ";
chaine+="toString" in liste; //Renvoie true

document.write(chaine);
</script>
```

Ce script donnera pour résultat :

true ; false ; true ; false ; false ; false ; false ; true ; true ; true

Voici quelques remarques pour comprendre cet exemple :

- 5 n'est pas une propriété de `liste`, c'est la valeur de l'élément de rang 0. Il sera accessible via l'instruction `liste[0]`.
- En écrivant `liste["prenom"]="Jacques";`, on ajoute un élément, dit associatif au tableau. On rajoute en fait la propriété `prenom` au tableau. Sa valeur peut également être rappelée par :

  `liste.prenom`
- `toString` est une méthode héritée de la classe `Object` *(voir chapitre sur la programmation orientée objet)*. Elle appartient donc au prototype de l'objet `liste`. La méthode `hasOwnProperty` ne cherche pas la présence d'une propriété dans le prototype de l'objet.

  `liste.hasOwnProperty(toString)`

  renverra la valeur `false`.

- `Instanceof` : cet opérateur permet de savoir si deux objets sont liées par l'héritage de classe. Cet opérateur est utile dans la programmation orientée objet ; pas dans cet formation.
- L'opérateur `new` permet d'invoquer le constructeur d'un objet afin de déclarer un nouvel objet. Cet opérateur est utile dans la programmation orientée objet ; pas dans cet formation.

- L'opérateur `typeof` permet de connaître le type de la variable passée en argument :

  ```
  typeof arg
  ```

  renvoit une chaîne de caractère notamment parmi : `"boolean"`, `"string"`, `"number"`, `"function"`, `"object"`, `"undefined"`

- Beaucoup d'instructions renvoient, sous **JavaScript**, une valeur. L'instruction :

  ```
  a=5
  ```

  renvoie la valeur de l'affectation.

  L'opérateur `void` oblige l'instruction qui la suit à renvoyer la valeur `undefined`. Ainsi contre tout attente, l'instruction suivante ne renverra aucune valeur :

  ```
  void(num=num+2)
  ```

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt9. *Priorités des opérateurs :*

Voici la liste des opérateurs disponibles classés dans l'ordre décroissant de prioriétés.

- () ; []
- −− ; ++ ; ! ; ~ ; −
- * ; / ; + ; −
- < ; <= ; >= ; >
- == ; !=
- ~
- |
- && ; ||
- ? ; :
- = ; += ; −= ; *= ; /= ; %= ; <<= ; >>= >>>= ; &= ; ~= ; |=
- ,

Pour lever un doute, lors de la succession de plusieurs opérateurs, il est toujours préférable d'utiliser les parenthèses.

f-javascript/03-noyauJavascript/

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt*G. Les structures de contrôles :*

Les structures de contrôles permettent de contrôler l'ordre d'exécution des instructions du script.

- Les alternatives : le programme exécute une partie du script ou une autre suivant la valeur d'un test.
  - `if() ...`
  - `if() ...else ...`
  - `if() ...if else() else ...`

- `switch(){...}`
- opérateur ternaire : `...?... : ...;`

- Les boucles : le programme exécutent plusieurs fois la même partie d'un code. Le nombre de répétition peut être fixé à l'avance ou dépendre de la validité d'une condition.
  - `while(){...}`
  - `do{...}while()`
  - `for(...){...}`

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt1. *Les structures conditionnelles :*

Les structures conditionnelles permettent à une fonction d'exécuter une partie ou une autre de son code en fonction de test que la fonction effectuera‘dépendant la plupart du temps de la valeur des arguments passés à la fonction ou de la valeur des variables globales.

- `if(`*Condition*`){`*InstructionsVrai*`}` `else {`*InstructionsFaux*`}`

  Cette structure évalue la valeur de "*Condition*" *(soit* `true`*, soit* `false`*)* :

  - Si celle-ci est vrai, elle exécutera le bloc d'instructions "*InstructionsVrai*".
  - Si celle-ci est fausse, le blocs d'instructions "*InstructionsFaux*" sera exécuté.

  La partie `else` (*InstructionsFaux*) est facultative : dans ce cas si la condition a pour valeur `false`, aucune instruction de la structure ne sera exécuté.

  Si l'un des deux blocs "*InstructionsVraies*" ou "*InstructionsFausses*" ne contiennent qu'une seule instruction, il est possible d'ommettre les accolades.

  Le script suivant montre l'utilisation d'une structure conditionnelle demande au client de donner son âge :

```
1  <script>
2  function verif(age){
3    if(age>18)
4      document.write("Bienvenu sur ce site");
5    else{
6      alert("Vous ne pouvez accéder à ce site");
7      return;
8    }
9  }
10
11 verif(prompt("Rentrez votre âge :"));
12 </script>
```

- ··· *Condition* ··· **?** ··· *InstructionVrai* ··· **:** ··· *InstructionFaux* ···

  Dans le cas où le test est simple et les blocs à exécuter sont composés d'une seule instruction, il existe une forme allégée pour la structure conditionnelle précédete.

  Voici une simplification du script précédent :

```
<script>
function verif(age){
  document.write(age>=18?"Bienvenu":"Au revoir");
}

verif(prompt("Rentrez votre âge :"));
</script>
```

- **switch(** ··· *Valeur* ··· **) {** ··· *Instructions* ··· **}**

  Cette structure s'appelle le branchement conditionnel : elle permet à partir d'une valeur de tester son égalité avec plusieurs valeurs et d'effectuer les instructions adéquates.

  Cette structure permet d'éviter d'imbriquer les structures `if()...else...` les unes dans les autres et simplifie le test de plusieurs valeurs. Voici un exemple de script testant plusieurs fois la valeur de la variable `age` :

```
<script>
function verif(){
  age=prompt("Rentrez votre âge :")
  var chaine=new String();
  switch(age){
   case "15":
     chaine="quinze";
     break;
   case "16":
     chaine="seize";
     break;
   case "17":
     chaine="dix-sept";
     break
   default:
     chaine="autre";
  }
  return chaine+" ans";
}

document.write(verif());
```

```
</script>
```

Voici quelques règles de syntaxe à respecter et quelques explications :

- L'instruction `switch` prend la valeur `val` puis effectuera les tests dans le bloc d'instructions le suivant.
- Ce bloc d'instructions contient les différents clause ayant la forme suivante :

```
case test1:
  instruction1
break;
```

  Dans le cas où `val` vérifie l'égalité stricte *(avec l'opérateur ===)* avec la valeur `test1`, le bloc d'instructions `instruction1` sera exécuté.
- On peut rajouter en fin du bloc d'instruction, les instructions :

```
case default:
  instructionDefaut
```

  Les instructions `instructionDefaut` seront évaluées dans le cas où tous les tests de comparaisons ont échoués.

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt2. *Les structures répétitives :*

Ces instructions permettent de répéter plusieurs fois une même séquence d'instructions : elle se répètent un nombre de fois déterminé à l'avance *(pour l'instruction* `for`*)* ou alors durant la validité d'un test *(pour les instructions* `do` *et* `while`*)* : attention alors aux boucles infinies.

- `while(···` *Condition* `···) { ···` *Instructions* `··· }`

  L'instruction `while` exécutera les instructions tant que la "*Condition*" aura pour valeur `true`.

  Voici un exemple du calcul de PGCD :

```
<script>
function pgcd(a,b){
  if(a<b){// Il me faut a plus grand que b.
    pgcd(b,a);
    return;
  }

  document.write("Le PGCD de "+a+" et "+b+" est : ");
  while(b != 0){
    c = b;
    b = a % b; //donne le reste de la division avec ↩
       quotient entier
```

```
12      a = c;
13    }
14    document.write(a);
15  }
16
17  pgcd(3212,72);
18  document.write("<p>");
19  pgcd(72,3212);
20  </script>
```

- `do { ···` *Instructions* `··· } while( ···` *Condition* `··· );`

  Cette boucle exécute les *Instructions* tant que *Condition* a la valeur `true`.

```
1  <script>
2  var i=5;
3  do{
4    document.write("Yep!<br>");
5  }while(i--);
6  </script>
```

  Essayez d'anticiper le nombre de fois que le mot "*Yep*" apparaîtra sur la page. Et si la condition avait été remplacée par `--i` ?

⚠ Les boucles `while(){...}` et `do{...}while()` sont assez semblables. La seule différence est :

- La première boucle vérifie la condition avant d'exécuter la première fois les instructions ;
- La seconde boucle exécutera une première fois les instructions, puis vérifie la condition pour lancer une seconde fois le bloc d'instructions.

- `for(`*AffectationCompteur* `;` *ConditionCompteur* `;` *ModificationCompteur*`)`
  `{ ···` *Instructions* `··· }`

  La boucle `for(){...}` permet d'exécuter une séquence d'instructions un certain nombre de fois. Son fonctionnement est simple :

  - On utilise un compteur qu'on initialise une première fois : *AffectationCompteur)*
  - Si "*ConditionCompteur*" a pour valeur `true` les "*Instructions*" seront exécutés une nouvelle fois, sinon on sort de la boucle pour revenir à l'exécution normale du code.
  - A la fin de chaque exécution d'*Instructions*, "*ModificationCompteur*" sera exécuté. Généralement, on incrémente le compteur de cette boucle pour maîtriser le nombre d'exécution d'*Instructions*.

- `for( var prop in objet ) { ···` *Instructions* `··· }`

  Cette boucle va exécuter autant de fois "*Instructions*" que `objet` possède de propriété.
  A chaque exécution de "*Instructions*", **JavaScript** introduira la variable `prop` contenant l'identifiant de la propriété d'`objet` ciblée dans ce passage.
  Cette boucle ne peut lister les propriétés ayant l'attribut `dontEnum`.

  On pourra connaître la valeur de cette propriété en utilisant la syntaxe :

  ```
  objet[prop]
  ```

  puisque `propriete` est une chaîne de caractères représentant l'identifiant de celle-ci.

Voici un exemple, vous permettant d'explorer quelqu'uns des objets les plus rencontrés en **JavaScript** :

```
<html>
<head>
 <style type="text/css">
   td{background-color:AAAAAA;cursor:pointer;text-align:center}
 </style>
 <script>
  var tableau = new Array("Ma","voiture","est","rouge");

  function affichePropriete(a){
   var chaine="";
   a=eval(a); //evite problème avec this
   for(var prop in a){
     chaine += prop+" : ";
     try{
       chaine += a[prop]+"<br>";
     }
     catch(erreur){
     chaine += "ne supporte pas [[call]] : interdit d'appeler<br>";
     }
   }
   return "<br>"+chaine;
  }

 function affiche(a){
  document.getElementById("texte").innerHTML =
```

```
      afficheProprietes(a);
 }

 </script>
</head>
<body>
 <table width="100%"><tr><td onclick="affiche('this')">this</td><td onclick="affiche('window')">Window</td><td onclick="affiche('document')">Document</td><td onclick="affiche('navigator')">navigator</td><td onclick="affiche('tableau')">tableau</td></tr></table>

 <div id=texte>Pour afficher les propriétés d'un objet</div>
</body>
</html>
```

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt3. *Les contrôles* break *et* continue

Il existe deux instructions permettant de sortir de l'exécution de boucles :

- A l'intérieur même d'un bloc d'instructions d'une structure `while`, `do` ou `switch`, l'instruction `break` interrompt l'exécution de la boucle et retourne dans le déroulement normal du programme.
- Lors de l'exécution d'une structure répétitive `do`, `while`, `for`, on peut interrompre l'itération courante pour demander à **JavaScript** d'évaluer une nouvelle fois la condition et d'exécuter le cas échéant une nouvelle fois le bloc d'instructions : pour cela, on utilise l'instruction `continue`

Voici une illustration de ces deux commandes :

```
<script>
 for(i=0;i<10;i++){
  if(i == 7)
    break;
  if(i == 3)
    continue;
  document.write(i+" - ");
 }
</script>
```

Ces deux instructions fonctionnent également avec des labels permettant de rediriger, l'exécution du script à un endroit précis. Cette façon de programmer, connû dans le jeune âge du basic, a tendance à disparaître.

14bb 2520.0pta 0.0ptbc 25.2ptd0.0pt4. *La structure with :*

Le mot-clef **with** permet de simplifier l'écriture du script lorsque plusieurs propriétés et méthodes d'un même objet sont successivement utilisés :

Le code suivant montre comment modifier plusieurs propriétés de style d'un même objet, sans avoir à répéter la présence de l'objet :

```
<div id=titre>Bonjour, </div>
<script>
with(document.getElementById("titre").style){
  backgroundColor="#AAAAAA";
  margin="10px 30px";
  padding="5px 15px";
  textAlign="center";
}
</script>
```