Université Hassan 1er

Faculté des Sciences et Techniques de Settat

COURS JAVASCRIPT

Professeur Laachfoubi Nabil

Département des Mathématiques et Informatique

Sommaire

A. Les bases du JavaScript

A.1. Petit historique du langage

En 1995, Brendan Eich travaillait chez Netscape Communication Corporation, la société qui a édité le navigateur Netscape Navigator, principal concurrent d'Internet Explorer à l'époque. Il a développé le LiveScript, un langage de scripts qui s'inspire du langage Java, et qui est destiné à être installé sur les serveurs développés par Netscape. La société se met à développer une version client du LiveScript, qui sera renommée JavaScript en hommage au langage Java créé par la société Sun Microsystems. En effet, à cette époque, le langage Java était de plus en plus populaire, et appelé le LiveScript. «JavaScript» était une manière de faire de la publicité à la fois au Java et au JavaScript lui-même. Mais attention, au final, ces deux langages sont radicalement différents. Il ne faut pas confondre le Java et le JavaScript car ils n'ont pas le même fonctionnement.

Le JavaScript est sorti en décembre 1995, il était embarqué dans le navigateur Netscape 2. Il a rencontré un succès, si bien que Microsoft a développé une version semblable, appelée JScript, qu'il a été intégré dans Internet Explorer 3, en 1996.

Netscape a décidé d'envoyer sa version de JavaScript à l'ECMA International (*European Computer Manufacturers Association* à l'époque, aujourd'hui *European Association for Standardizing Information and Communication Systems*) pour que le langage soit standardisé, c'est-à-dire pour qu'une référence du langage soit créée et qu'il puisse ainsi être utilisé par d'autres personnes et embarqué dans d'autres logiciels. L'ECMA International standardise le langage sous le nom d'ECMAScript (*http : //fr.wikipedia.org/wiki/ECMAScript*).

Depuis, les versions de l'ECMAScript ont évolué. La version la plus connue et mondialement utilisée est la version ECMAScript 5, parue en décembre 2009.

A.2. Les versions du JavaScript

Les versions du JavaScript sont basées sur celles de l'ECMAScript (ES). Ainsi, il existe :

- ES 1 et ES 2, qui sont les prémices du langage JavaScript,
- ES 3 (sorti en décembre 1999),
- ES 4, qui a été abandonné en raison de modifications trop importantes qui ne furent pas appréciées,
- ES 5 (sorti en décembre 2009), la version la plus répandue et utilisée à ce jour,
- ES 6 finalisé en juin 2015 est la dernière version du langage mais elle n'est pas encore complètement supportée par tous les navigateurs.

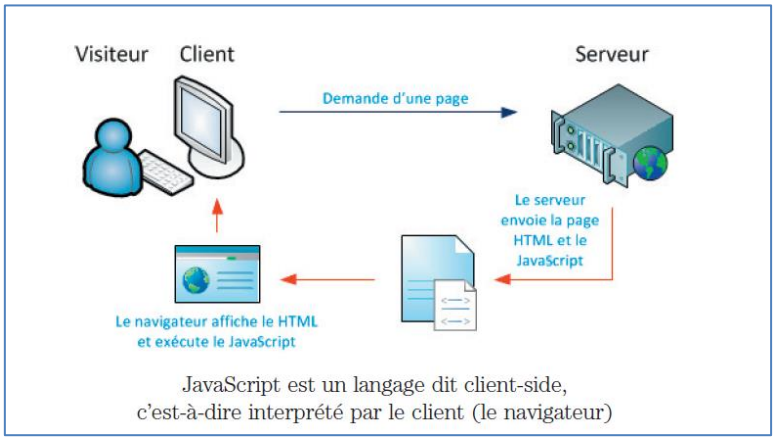
A.3. JavaScript et le texte HTML

JavaScript est un langage de script du côté client, il s'exécute au sein d'un navigateur web. Pour appeler un script, on place le code entre les balises HTML <script> et </script>. Par exemple, pour afficher "Bonjour tout le monde" on peut écrire le code suivant :

```
<!doctype html>
<html>
   <head>
      <title>Bonjour tout le monde</title>
   </head>
   <body>
      <script type="text/javascript">
         document.write("Bonjour tout le monde ")
      </script>
      <noscript>
         Votre navigateur ne prend pas en charge ou a désactivé JavaScript
      </noscript>
   </body>
</html>
```

Dans cet exemple, la ligne de code JavaScript affiche la chaine fournie dans le document courant.

La balise HTML <noscript>…</noscript>, permet d'afficher un texte alternatif au cas où le navigateur de l'utilisateur ne prend pas en charge le JavaScript ou si on l'a désactivé.



JavaScript est un langage dit client-side, c'est-à-dire interprété par le client (le navigateur)

En plus du corps d'un document, on peut placer un script dans la section d'entête <head>. Dans ce cas, le script s'exécute au moment du chargement de la page. Le placement du code et des fonctions essentielles à cet endroit, permet un usage immédiat de celles-ci par les autres sections de script du document. Cela a aussi un autre avantage, car il donne la main à JavaScript pour écrire des choses comme des balises méta dans la section <head>.

Inclure des fichiers JavaScript

Au lieu d'écrire du code JavaScript directement dans des documents HTML, on peut inclure des fichiers de code JavaScript à partir de notre site ou de n'importe où sur internet. La syntaxe est la suivante :

```
<script type="text/javascript" src="monscript.js">
    document.write("Variable " + err);
</script>
<script type="text/javascript" src="https : //ssl.google-analytics.com/ga.js"></script>
```

On peut omettre Le paramètre *type="text/javascript"* puisque tous les navigateurs modernes supposent que ces scripts contiennent du JavaScript.

Déboguer les erreurs en JavaScript

Souvent, lorsque le code JavaScript contient des erreurs, rien ne se produira. Il n'y a aucun message d'erreur, et on n'obtient aucune indication où rechercher des erreurs.

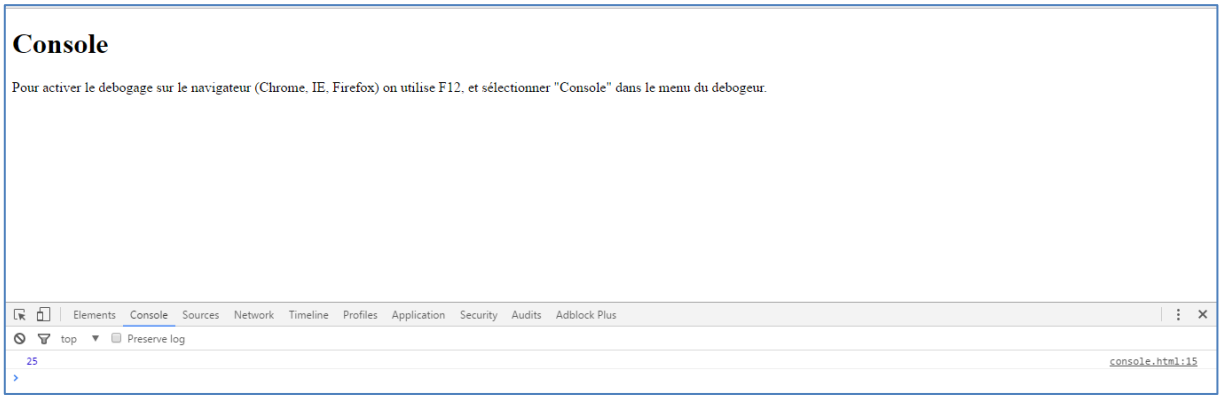
Le débogage des erreurs est le fait de rechercher les erreurs dans le code.

Le débogage n'est pas facile. Mais heureusement, tous les navigateurs modernes disposent d'un débogueur JavaScript intégré. Avec un débogueur, on peut définir des points d'arrêt (endroits où l'exécution du code peut être arrêtée) et examiner les variables pendant l'exécution du code.

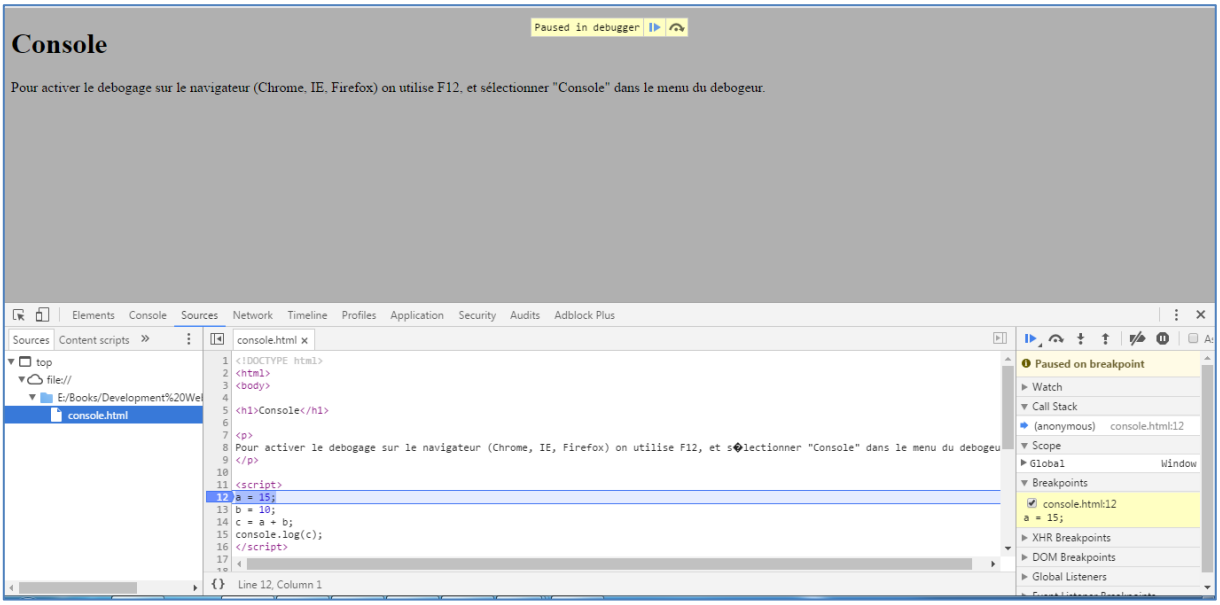
Pour activer le débogage dans le navigateur (Chrome, Firefox et IE), on appuie sur la touche F12 et on sélectionne "Console" dans le menu du débogueur.

Si le navigateur prend en charge le débogage, on peut utiliser *console.log ()* pour afficher les valeurs JavaScript dans la fenêtre du débogueur :

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Console</h1>
    <p>
      Pour activer le débogage sur le navigateur (Chrome, IE, Firefox) on utilise F12, et sélectionner "Console"
      dans le menu du débogueur.
    </p>
    <script>
      a = 15;
      b = 10;
      c = a + b;
      console.log(c);
    </script>
  </body>
</html>
```



On peut également définir des points d'arrêt (Breakpoints en anglais) dans le code JavaScript via la fenêtre du débogueur. À chaque point d'arrêt, JavaScript cesse d'exécuter et nous permet d'examiner les valeurs JavaScript. Après avoir examiné les valeurs, on peut reprendre l'exécution du code (généralement avec un bouton de lecture).



Commentaires

En JavaScript, les commentaires peuvent être sur une seule ou plusieurs lignes, comme suit :

```
// Voici un commentaire sur toute la ligne
/* Voici un bloc
De commentaire
Sur plusieurs lignes */
```

Points-virgules

Généralement, le JavaScript ne nécessite pas de points-virgules si les instructions figurent seules par ligne. Donc on peut écrire :

```
x+=1
```

En revanche, lorsqu'on place plusieurs instructions sur une même ligne, on doit les séparer par des points-virgules, comme suit :

```
x+=1 ; y-=1 ; z=0
```

A.4. Les variables

Aucun caractère particulier n'identifie spécialement une variable en JavaScript. Les variables suivent les règles de nommage suivantes :

1. Le nom d'une variable ne peut contenir que les lettres a-z, A-Z, 0-9, le symbole $ et le soulignement (_),
2. Aucun autre caractère, espace ou de ponctuation n'est autorisé dans un nom de variable,
3. Le premier caractère d'un nom de variable ne peut être qu'une lettre a-z, A-Z, le symbole $ ou le soulignement (_), mais pas un chiffre,
4. Les noms de variables sont sensibles à la casse, donc y et Y désignent des variables différentes,
5. Aucune limite n'est définie au niveau de la longueur des noms de variables,
6. Les mots réservés JavaScript ne peuvent pas être utilisés comme des noms de variables.

Variables de chaines

En JavaScript, les variables de chaines de caractères sont entourées de guillemets ou d'apostrophes :

```
salutation = "Bonjour"
avertissement = 'Attention'
```

L'apostrophe est permise dans une chaine entourée de guillemets et le guillemet est permis dans une chaine entourée d'apostrophes. En revanche, on doit placer une séquence d'échappement (\) devant un guillemet ou une apostrophe si la chaine est entourée de caractères du même type, comme suit :

```
salutation = "\"Bonjour \" est une salutation"
avertissement = '\'Attention\' est un avertissement '
L'affectation d'une variable de chaine à une autre prend l'allure suivante :
nouvellechaine = anciennechaine
```

Variables numériques

Pour créer une variable numérique, on lui affecte une valeur de type nombre, comme suit :

```
compteur = 1
prix = 300
```

A l'instar des variables de chaines, les variables numériques peuvent intervenir dans des expressions, des fonctions et peuvent en recevoir les résultats.

A.5. Les tableaux

Les tableaux en JavaScript se comportent de manière similaire à ceux de PHP par exemple, car ils peuvent contenir des chaines, des données numériques et d'autres tableaux. Pour attribuer des valeurs à un tableau, on utilise la syntaxe suivante (ici pour créer un tableau de chaines) :

```
voitures = ['BMW', 'Ford', 'Volvo', 'Totoya']
```

Pour créer u tableau à plusieurs dimensions, on imbrique de plus petits tableaux dans un plus grand. Ainsi, pour constituer un tableau à deux dimensions on peut écrire :

```
matrice =[ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
```

Ensuite, pour accéder à l'élément situé à la deuxième ligne et à la troisième colonne de cette matrice, il suffit d'écrire (les indices des éléments des tableaux commencent à 0) :

```
alert(matrice[1][2])
```

A.6. Les opérateurs

Les opérateurs en JavaScript portent sur les mathématiques, la conversion en chaine, ainsi que les opérations de comparaison et logiques.

Opérateurs arithmétiques

Les opérateurs arithmétiques permettent d'effectuer des opérations algébriques qui portent sur les quatre opérations principales (addition, soustraction, multiplication et division), plus le modulo (ou reste de la division entière), ainsi que l'incrémentation et la décrémentation.

Opérateur	Description	Exemple
+	Addition	i+ 5
-	Soustraction	i - 10
*	Multiplication	i * 1.5
/	Division	i / 3.33
%	Modulo	i % 2
++	Incrémentation	++i
--	Décrémentation	-- i

Opérateurs d'affectation

Les opérateurs d'affectation servent à attribuer des valeurs à des variables.

Opérateur	Exemple	Equivaut à
=	j = 12	j = 12
+=	j += 2	j = j + 2
+=	j += 'chaine'	j = j + 'chaine'
-=	j -= 6	j = j - 6
*=	j *= 10	j = j * 10
/=	j /= 5	j = j / 5
%=	j %= 3	j = j % 3

Opérateurs de comparaison

Les opérateurs de comparaison servent généralement dan ne construction telle qu'une instruction if où il faut comparer deux éléments.

Opérateur	Description	Exemple
==	est égale à	x == 105
!=	est différent de	x != 8
>	est plus grand que	x > 0
<	est plus petit que	x < 0
>=	est plus grand ou égale à	x >= 33
<=	est plus petit ou égale à	x <= 11
===	est égal à (et de même type que)	x === 78
!==	est différent à (et de type différent que)	x !== '1'

Opérateurs logiques

Les opérateurs logiques sont utilisés pour déterminer la logique entre les variables ou les valeurs. JavaScript possède trois opérateurs logiques : *&&*, *||* et *!*.

Opérateur	Description	Exemple
&&	Et	x < 10 && y > 1
||	Ou	x == 5 || y == 5
!	Non(ou pas)	!(x == y)

Concaténation des chaines

Pour la concaténation de chaines, JavaScript utilise le signe + (plus) comme suit :

```
messages = 4
alert ("Vous avez " + messages + " messages.")
```

Le résultat de ce code est le suivant :

```
Vous avez 4 messages.
```

De même, on peut aussi utiliser opérateur += pour concaténer deux chaines :

```
Boxeur = "Mike "
Boxeur += "Tyson"
```

Séquences d'échappement

Les séquences d'échappement permettent des caractères spéciaux tels que les tabulations, les nouvelles lignes et les retours à la ligne.

Séquence	Ségnification
\b	Retour arrière
\f	Saut de page
\n	Nouvelle ligne
\r	Retour à la ligne
\t	Tabulation
\'	Apostrophe verticale
\"	Guillemet vertical
\\	Barre oblique inverse

A.7. Le typage des variables

Comme PHP ou Perl, JavaScript est un langage à typage faible. Le type d'une variable est défini seulement lorsqu'elle reçoit une valeur et peut évoluer si la variable apparait dans des contextes différents. Il n'est pas généralement pas nécessaire de se préoccuper du type, car JavaScript devine ce qu'on veut faire et le fait, tout simplement.

Soit par exemple une variable x. Si on utilise l'opérateur *typeof* pour déterminer le type, ca va afficher "undefined". Si on affecte à x la valeur 5 et on utilise typeof ca va afficher "number" et si on affecte au même variable la valeur "Mazda", utilisation de typeof affichera "string".

Les fonctions

Les fonctions en JavaScript permettent d'isoler des portions de code qui exécutent des taches spécifiques, puis de les appeler selon les nécessités. Pour créer une fonction, on peut la déclarer comme suit :

```
<script>
   function addition(a, b)
   {
      return a+b
   }
</script>
```

Cette fonction demande deux paramètres et les additionne pour en renvoyer le total.

Les variables globales

Les variables globales sont celles définies en dehors de toute fonction (ou au sein de fonctions mais définies sans le mot clé var). Elles peuvent être définies de la manière suivante :

```
a = 123 // Portée globale
var b = 456 // Portée globale
if (a == 123) var c = 789 // Portée globale
```

Que vous mentionniez ou non le mot clé var, tant que la définition d'une variable ait lieu en dehors d'une fonction, sa portée est globale. Cela signifie que la variable est accessible de partout dans un script.

Les variables locales

Les paramètres transmis à une fonction ont d'emblée une portée locale, c'est-à-dire qu'il n'est possible d'y faire référence qu'au sein de cette fonction. Subsiste toutefois une exception au niveau des tableaux. Les tableaux sont passés par référence, donc si vous modifiez le moindre élément d'un tableau en paramètre d'une fonction, alors l'élément correspondant du tableau original est également modifié.

Pour définir une variable locale dont la portée se limite à la fonction courante qui ne soit pas passée en paramètre de celle-ci, on utilise le mot clé var pour la définir.

```
<script>
   function test()
   {
      a = 123 // Portée globale
      var b = 456 // Portée locale
      if (a == 123) var c = 789 // Portée locale
   }
</script>
```

A.8. Le modèle DOM (Document Object Model)

Le modèle objet de DOM (Document Object Model) décompose les portions d'un document HTML en objets discrets, qui possèdent chacun leurs propres propriétés et méthodes, dont le contrôle devient ainsi à la portée de JavaScript. Le DOM HTML est un standard W3C définit comme :

« Une plate-forme et une interface neutre du langage qui permet aux programmes et aux scripts d'accéder et de mettre à jour dynamiquement le contenu, la structure et le style d'un document ».

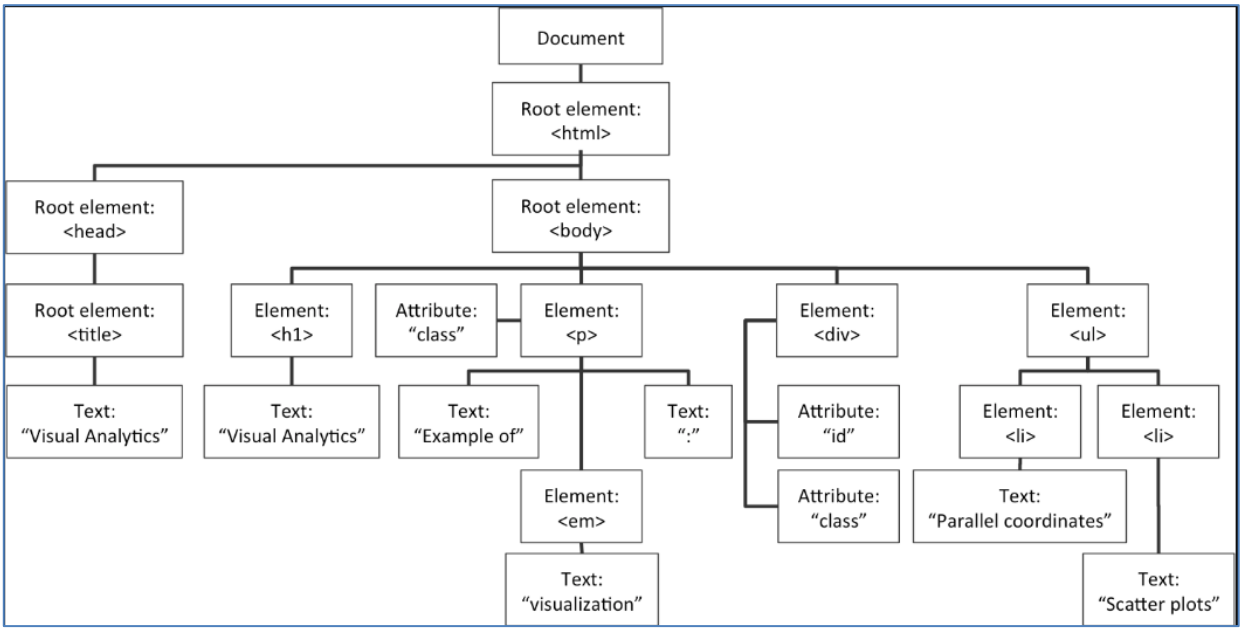
Le standard DOM permet de définir :

- Les éléments HTML en tant qu'***objets***,
- Les ***propriétés*** de tous les éléments HTML,
- Les ***méthodes*** d'accès à tous les éléments HTML,
- Les ***événements*** de tous les éléments HTML.

En d'autres termes : Le DOM HTML est un standard pour savoir comment obtenir, modifier, ajouter ou supprimer des éléments HTML.

Avec le modèle d'objet, JavaScript obtient toute la puissance dont il a besoin pour créer du HTML dynamique :

- JavaScript peut changer tous les éléments HTML de la page,
- JavaScript peut changer tous les attributs HTML dans la page,
- JavaScript peut changer tous les styles CSS dans la page,
- JavaScript peut supprimer des éléments et des attributs HTML existants,
- JavaScript peut ajouter de nouveaux éléments et attributs HTML,
- JavaScript peut réagir à tous les événements HTML existants dans la page,
- JavaScript peut créer de nouveaux événements HTML dans la page.



B. Expressions et contrôle du flux d'exécution en JavaScript

B.1. Les expressions

Une expression est une combinaison de valeurs, de variables, d'opérateurs et de fonctions qui renvoie une valeur ; ce résultat peut être un nombre, une chaine ou une valeur booléenne (*true* ou *false*).

Voici un exemple qui illustre quelques expressions simples :

```
<script>
   document.write("a : " + (11 > 3) + "<br>")
   document.write("b : " + (85 < 4) + "<br>")
   document.write("c : " + (7 == 5) + "<br>")
   document.write("d : " + (1 < 16) + "<br>")
</script>
```

Les résultats de ce script sont les suivants :

```
a : true
b : false
c : false
d : true
```

En JavaScript, *true* et *false* s'écrivent obligatoirement en lettres minuscules.

B.2. Les valeurs littérales et les variables

La forme la plus simple d'expression est le littéral, ou valeur littérale, qui désigne quelque chose qui s'évalue à lui-même, comme le nombre 11 ou la chaine 'Soyez prudent'. Une expression peut aussi être une variable, dont l'évaluation donne la valeur qui lui a été affectée. Ce sont là deux types d'expressions parce qu'elles renvoient une valeur.

Voici un exemple qui illustre les valeurs de retours de différentes valeurs littérales et deux variables :

```
<script>
   monnom = "Chakib"
   monage = 22
   document.write("a : " + 62 + "<br>") //Littéral numérique
   document.write("b : " + "Bonjour" + "<br>") // Littéral chaine
   document.write("c : " + true + "<br>") // Littéral constant
   document.write("d : " + monnom + "<br>") // Variable chaine
   document.write("e : " + monage + "<br>") // Variable numérique
</script>
```

Ce script affichera Les résultats suivants :

```
a : 62
b : Bonjour
c : true
d : Chakib
e : 22
```

Les opérateurs permettent de créer des expressions plus complexes, qui s'évaluent à des résultats plus utiles. Lorsqu'on combine l'affectation ou des constructions de contrôle de flux de programme avec des expressions, on obtient une *instruction*.

L'exemple suivant montre deux instructions simples en JavaScript :

```
<script>
   jours_avant_nouvel_an = 366 - numero_du_jour ;
   if (jours_avant_nouvel_an < 30) document.write("Le Nouvel An est pour bientôt !")
</script>
```

B.3. Utiliser try…catch

L'instruction *try...catch* regroupe des instructions à exécuter et définit une réponse si l'une de ces instructions provoque une exception (une erreur).

Sa syntaxe est :

```
try
{
    // Bloc d'instructions à exécuter
}
catch(err)
{
    // Bloc de code pour gérer les erreurs
}
finally
{
    // Bloc de code à exécuter quel que soit le résultat de l'instruction try / catch
}
```

L'instruction try est composée d'un bloc try qui contient une ou plusieurs instructions et au moins d'une clause catch ou d'une clause finally ou des deux. On peut donc avoir les trois formes suivantes pour cette instruction :

- try...catch
- try...finally
- try...catch...finally

Une clause catch contient les instructions à exécuter si une exception est levée par une instruction du bloc try.

La clause finally s'exécute après le bloc try et après le bloc catch (si celui-ci a été déclenché) mais avant les instructions qui suivent. Les instructions de cette clause sont toujours exécutées, qu'il y ait eu ou non une exception de déclenchée et/ou d'interceptée.

Par défaut, en cas d'erreur, JavaScript s'arrête et génère un message d'erreur. Si en veut générer une exception personnalisée, on doit utiliser l'instruction throw.

L'exception peut-être une chaîne, un nombre, un booléen ou un objet.

En combinant throw avec try /catch, on peut contrôler le flux du programme et générer des messages d'erreur personnalisés.

Voici un programme complet qui utilise les instructions try…catch :

```
<script>
    x = "" ;
    try
    {
        if(x == "") throw "est vide";
        if(isNaN(x)) throw "n'est pas un nombre";
        if(x > 10) throw "trop grand";
        if(x < 5) throw "trop petit";
    }
    catch(err)
    {
        document.write("Variable "+err);
    }
    Finally
    {
        alert("Ceci est une instruction finally") ;
    }
</script>
```

B.4. Conditions

Les instructions conditionnelles modifient le flux d'exécution d'un programme.

Instruction if

Instruction if exécute un bloc d'instructions si l'expression donnée s'évalue en true. Sa syntaxe est :

```
if (condition) {
    Bloc de code à exécuter si la condition est correcte
}
```

Par exemple :

```
if (a > 100)
{
    document.write("a est supérieur à 100")
}
```

Instruction else

Quand une condition n'est pas remplie, on peut exécuter une voie alternative à l'aide d'une instruction else, comme suit :

```
if (a > 100)
{
    document.write("a est supérieur à 100")
}
else
{
    document.write("a est plus petit que 100")
}
```

Si on veut spécifier une nouvelle condition si la première n'est la remplie, on peut utiliser instruction else if. Sa syntaxe est :

```
if (condition_1)
{
    Bloc de code à exécuter si la condition_1 est correcte
}
else if (condition_2)
{
    Bloc de code à exécuter si la condition1 n'est pas correcte et que  la condition_2 est correcte
}
…
else if (condition_k)
{
    Bloc de code à exécuter si la condition_k-1 n'est pas correcte et que la condition_k est correcte
}
else
{
    Bloc de code à exécuter si toutes les conditions sont fausses
}
```

Instruction switch

L'instruction switch s'avère la plus utile lorsqu'une variable ou le résultat d'une expression peut avoir plusieurs valeurs possibles, et qu'on veut exécuter une fonction différente pour chaque valeur. Sa syntaxe est :

```
switch(expression)
{
    case a :
       bloc de code
       break;
    case b :
       bloc de code
       break;
   ...
   case n :
       bloc de code
       break;
    default :
       bloc de code
}
```

La commande *break* permet de sortir de l'instruction *switch* lorsqu'une condition est satisfaite. On doit toujours inclure le *break* à moins qu'on souhaite poursuivre l'exécution aux instructions du *case* suivant.

Lorsqu'aucune condition n'est satisfaite, on peut définir une action par défaut à l'aide du mot clé *default*.

Voici un exemple complet de l'instruction switch :

```
switch (aujourdhui)
{
    case 0 :
       jour = "Dimanche";
       break;
    case 1 :
       jour = "Lundi";
       break;
    case 2 :
       jour = "Mardi";
       break;
    case 3 :
       jour = "Mercredi";
       break;
    case 4 :
       jour = "Jeudi";
       break;
    case 5 :
       jour = "Vendredi";
       break;
    case 6 :
       jour = "Samedi";
}
```

L'opérateur ternaire ?

En combinaison avec le caractère **:** l'opérateur ternaire **?** fournit un moyen rapide de réaliser des tests if...else. Sa syntaxe est :

```
nomvariable = (condition) ? valeur1 : valeur2
```

Voici un exemple d'utilisation :

```
Frais_livraison = (total > 200) ? "Gratuit" : "25 MAD";
```

B.5. Boucles

La boucle while

La boucle while (tant que) en JavaScript vérifie d'abord la valeur d'une expression et commence l'exécution des instructions dans la boucle seulement si cette expression est vraie. Si la condition est fausse, l'exécution passe à l'instruction JavaScript suivante (s'il y en a). L'exemple suivant illustre une telle boucle :

```
<script>
   compteur=0
   while (compteur < 5)
   {
      document.write("compteur : " + compteur + "<br>")
      ++ compteur
   }
</script>
```

La boucle do…while

Lorsqu'une boucle doit itérer au moins une fois avant d'effectuer le premier test, on utilise plutôt la boucle do...while, semblable à la boucle while, à l'exception du fait que l'expression de test n'est évaluée qu'à la fin des itérations de la boucle. L'exemple suivant illustre une telle boucle :

```
<script>
   compteur = 1
   do
   {
      document.write(compteur + " fois 7 est " + compteur * 7 + "<br>")
   } while (++compteur <= 7)
</script>
```

Boucles for

La boucle for combine le meilleur de tous les mondes dans une construction de boucle simple qui prend trois paramètres dans son instruction :

1. Une expression d'initialisation,
2. Une expression de condition,
3. Une expression de modification.

Un exemple simple d'utilisation est :

```
for (i = 0; i < 5; i++)
{
    texte += "Le numéro est " + i + "<br>";
}
```

On peut aussi utiliser la boucle for pour itérer sur les propriétés d'un objet avec l'instruction for / in comme dans cet exemple :

```
var personne = {prenom : "Ahmed", nom : "Tazi", age : 25};
var texte = "";
var x;
for (x in personne)
 {
    texte += personne[x];
}
```

B.6. Conversion de type (cast) explicite

JavaScript ne possède pas de conversion de type explicite. Lorsqu'on a besoin d'une valeur d'un type déterminé, on utilise plutôt une des fonctions intégrées de JavaScript regroupées dans le tableau suivant :

Conversion en type	Fonction à utiliser
Int, Integer	parseInt()
Bool, Boolean	Boolean()
Float, Double, Real	parseFloat()
String	String()
Array	split()

C. Fonctions, objets et tableaux en JavaScript

Comme plusieurs langages, JavaScript offre la possibilité de manipuler des fonctions et des objets. En fait, JavaScript est fondamentalement basé sur des objets car, comme on l'a vu, il doit accéder au DOM qui permet de disposer de chaque élément d'un document HTML en tant qu'objet.

C.1. Les fonctions en JavaScript

En plus de pouvoir accéder à des dizaines de fonctions (et méthodes) intégrées, telles que write, on peut créer nos propres fonctions. Chaque fois qu'on a une portion de code complexe susceptible d'être réutilisée, on a un candidat à la définition d'une fonction.

Définir une fonction

Pour définir une fonction, on utilise la syntaxe générale suivante :

```
function nom_fonction ([parametre [,...]])
{
   instructions
}
```

Les noms de fonctions sont sensibles à la casse.

La première ligne de cette syntaxe indique que :

- Une définition débute par le mot clé *function*,
- Un nom vient ensuite, qui doit commencer par une lettre ou un soulignement, suivi de n'importe quel nombre de lettres, chiffres, symboles dollar ou soulignements,
- Les parenthèses sont obligatoires,
- Un ou plusieurs paramètres, séparés par des virgules, viennent s'insérer de manière facultative entre les parenthèses (les crochets ne font pas partie de la syntaxe mais indiquent que ces paramètres sont facultatifs).

Le tableau arguments

Le tableau *arguments* est membre de toute fonction. Il permet de déterminer le nombre de variables passées à une fonction et ce qu'elles sont.

Si une fonction est appelée avec les arguments manquants (moins que ceux déclarés), les valeurs manquantes auront la valeur undefined.

Si une fonction est appelée avec trop d'arguments (plus que ceux déclarés), ces arguments peuvent être atteints en utilisant le tableau *arguments*.

Soit par exemple la fonction suivante :

```
<script>
   afficherElements("Chien", "Chat", " Perroquet", "Hamster", "Tortue")
   function afficherElements(v1, v2, v3, v4, v5)
   {
      document.write(v1 + "<br>")
      document.write(v2 + "<br>")
      document.write(v3 + "<br>")
      var personne = {prenom : "Ahmed", nom : "Tazi", age : 25};
      var texte = "";
      var x;
      for (x in personne)
      {
          texte += personne[x];
      }
      document.write(v4 + "<br>")
      document.write(v5 + "<br>")
   }
</script>
```

Que se passe-t-il si on décide de passer plus que cinq arguments à la fonction ? La forme actuelle d'afficherElements ne permet pas de tenir compte d'un nombre de paramètres supérieur à cinq. Heureusement, le tableau arguments apporte la souplesse nécessaire pour gérer un nombre variable d'arguments. La fonction "afficherElements" peut être réécrite comme suite :

```
<script>
   function afficherElements ()
   {
      for (j = 0 ; j < afficherElements.arguments.length ; ++j)
         document.write(afficherElements.arguments[j] + "<br>")
   }
```

```
</script>
```

Grace à cette technique, on dispose d'une fonction capable de recevoir autant (et aussi peu) d'arguments qu'on le souhaite et d'agir sur chacun de ces arguments à volonté.

Renvoyer une valeur

Les fonctions ne sont pas utiles qu'à l'affichage. Elles servent essentiellement à effectuer des calculs ou à manipuler des données et renvoyer un résultat. Une fonction JavaScript renvoie un résultat avec le mot clé return.

Voici un exemple de fonction renvoyant une valeur :

```
function multiplication(a, b) {
    return a * b
}
```

C.2. Objets en JavaScript

En JavaScript, un objet représente une étape supérieure par rapport à une variable, qui peut ne contenir qu'une valeur au même moment, en ceci que les objets peuvent contenir plusieurs valeurs et même des fonctions. Un objet regroupe des données et les fonctions qui servent à les manipuler.

Déclarer une classe

Lors de la création d'un script destiné à gérer des objets, on doit concevoir une combinaison de données et de code appelée classe. Chaque nouvel objet basé sur cette classe est appelé une instance (ou une occurrence) de cette classe. Les données associées à un objet s'appellent ses propriétés, tandis que les fonctions qu'il utilise sont appelées ses méthodes.

Voyons comment déclarer une classe dénommée Utilisateur qui, par l'intermédiaire des objets qui en découleront, contiendra des informations à propos de l'utilisateur courant. Pour créer la classe, on écrit simplement une fonction du même nom que la classe. Cette fonction peut attendre des arguments. Nous allons également créer des propriétés et des méthodes dans cette classe. Cette fonction s'appelle le *constructeur* de la classe.

Voici le code de création de la classe «Utilisateur»

```
<script>
   function Utilisateur(prenom, nomutilisateur, motdepasse)
   {
      this.prenom = prenom
      this.nomutilisateur = nomutilisateur
      this.motdepasse = motdepasse
      this.afficherUtilisateur = function()
      {
         document.write("Prenom : " + this.prenom + "<br>")
         document.write("Nomutilisateu : " + this.nomutilisateu + "<br>")
         document.write("Motdepasse : " + this.motdepasse + "<br>")
      }
   }
</script>
```

Cette fonction diffère des autres fonctions vues précédemment à deux niveaux :

- Elle fait référence à un objet nommé *this*(ceci). Lorsqu'un programme crée une instance d'Utilisateur par l'exécution de cette fonction, *this* se réfère à l'instance en cours de création. On peut appeler la fonction plusieurs fois avec des arguments différents et elle crée chaque fois un Utilisateur avec des valeurs différentes pour la propriété *prenom* et les suivantes,
- Une nouvelle fonction *afficherUtilisateur* est créée au sein de la fonction. Il s'agit d'une syntaxe nouvelle et assez complexe, mais son rôle est de lier *afficherUtilisateur* en tant que méthode de la classe Utilisateur.

Il est aussi possible de faire référence à des fonctions définies en dehors du constructeur, comme suit :

```
<script>
   function Utilisateur(prenom, nomutilisateur, motdepasse)
   {
      this.prenom = prenom
      this.nomutilisateur = nomutilisateur
      this.motdepasse = motdepasse
      this.afficherUtilisateur = afficherUtilisateur
   }
   function afficherUtilisateur()
   {
      document.write("Prenom : " + this.prenom + "<br>")
      document.write("Nomutilisateur : " + this.nomutilisateu + "<br>")
      document.write("Motdepasse : " + this.motdepasse + "<br>")
   }
</script>
```

Créer un objet

Pour créer une instance de la classe «Utilisateur», on utilise une instruction du style de la suivante :

```
details = new Utilisateur ("Sara", "Naji", "Test123")
```

Mais on peut très bien créer un objet vide, comme suit :

```
details = new Utilisateur ()
```

Et le remplir par la suite, comme ceci :

```
details.prenom = "Sara"
details.nomutilisateur = " Naji"
details.motdepasse = "Test123"
```

On peut aussi ajouter de nouvelles propriétés à un objet existant, comme suit :

```
details.salutation = "Bonjour"
```

Accéder aux objets

Pour accéder à un objet, faites référence à ces propriétés, comme dans les deux instructions suivantes, sans aucun rapport entre elles :

```
nom = details.prenom
if (details. nomutilisateur == "Admin") connecterCommeAdmin()
```

Ainsi pour accéder à la methode «afficherUtilisateur» d'un objet de la classe «Utilisateur», on utilise la syntaxe suivante, où l'objet «details» est supposé déjà créé et rempli de données :

```
details.afficherUtilisateur()
```

Le mot clé Prototype

Le mot clé prototype permet parfois d'économiser beaucoup de mémoire. Dans la classe Utilisateur, chaque instance contient les trois propriétés et la méthode. Par conséquent, si on gère 1000 exemplaires de cet objet en mémoire, la méthode affficherUtilisateur est également répliquée 1000 fois. Or, comme cette méthode est identique dans chaque instance, on peut indiquer à chaque nouvel objet qu'il doit faire référence à une seule instance de la méthode au lieu d'en créer une copie. Pour cela, au lieu d'utiliser ce qui suit dans un constructeur de class :

```
this.afficherUtilisateur = function()
```

On le remplace par ceci :

```
Utilisateur.prototype.afficherUtilisateur = function()
```

Et donc la définition de la classe utilisateur deviendra :

```
<script>
   function Utilisateur(prenom, nomutilisateur, motdepasse)
   {
      this.prenom = prenom
      this.nomutilisateur = nomutilisateur
      this.motdepasse = motdepasse
      Utilisateur.prototype.afficherUtilisateur = function()
      {
         document.write("Prenom : " + this.prenom + "<br>")
         document.write("Nomutilisateur : " + this.nomutilisateu + "<br>")
         document.write("Motdepasse : " + this.motdepasse + "<br>")
      }
   }
</script>
```

Ceci fonctionne parce que toutes les fonctions possèdent la propriété prototype, conçue pour conserver des propriétés et des méthodes à ne répliquer dans aucun des objets créés à partir d'une classe.

Cela signifie que on peut ajouter une propriété ou une méthode prototype à tout moment et tous les objets (même ceux déjà créés) en héritent, comme l'illustrent l'instruction :

```
Utilisateur.prototype.salutation = "Bonjour"
```

C.3. Tableaux en JavaScript

Les tableaux numériques

Pour créer un tableau, on utilise la syntaxe suivante :

```
nomtableau = new Array()
```

Une forme abrégée existe aussi :

```
nomtableau = []
```

Affecter des valeurs aux éléments d'un tableau

En JavaScript, pour ajouter un nouvel élément à un tableau, on fait appel à la méthode push, comme ceci :

```
Nomtableau.push("élément 1") ;
Nomtableau.push("élément 2") ;
```

Ceci permet d'ajouter des éléments à un tableau sans devoir tenir compte du nombre d'éléments. Lorsqu'on souhaite connaitre le nombre d'éléments présents dans un tableau, on fait appel à sa propriété *length*, comme suit :

```
document.write(montableau.length)
```

Si on souhaite plutôt suivre les emplacements des éléments et les placer dans des emplacements déterminés, on utilise la syntaxe suivante :

```
Nomtableau[0] = élément 1" ;
Nomtableau[1] = "élément 2" ;
```

Affectation à l'aide du mot clé Array

Une autre technique permet de créer rapidement un tableau défini avec quelques éléments initiaux, à l'aide du mot clé Array, comme suit :

```
nombres = new Array("Un", "Deux", "Trois");
```

L'écriture abrégée suivante demeure la plus utilisée :

```
nombres = ["Un", "Deux", "Trois"];
```

Les tableaux multidimensionnels

La création d'un tableau multidimensionnel se résume à insérer des tableaux dans un autre tableau. Ainsi, le code suivant crée le tableau nécessaire pour contenir un damier bidimensionnel avec ses pions (de 10*10 case) :

```
<script>
   damier = Array(
                        Array(' ', 'o', ' ', 'o', ' ', 'o', ' ', 'o'),
                        Array('o', ' ', 'o', ' ', 'o', ' ', 'o', ' '),
                        Array(' ', 'o', ' ', 'o', ' ', 'o', ' ', 'o'),
                        Array(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '),
                        Array(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '),
                        Array('O', ' ', 'O', ' ', 'O', ' ', 'O', ' '),
                        Array(' ', 'O', ' ', 'O', ' ', 'O', ' ', 'O'),
                        Array('O', ' ', 'O', ' ', 'O', ' ', 'O', ' ')
                    )
   document.write("<pre>")
   for (j = 0 ; j < 8 ; ++j)
   {
      for (k = 0 ; k < 8 ; ++k)
      document.write(damier[j][k] + " ")
      document.write("<br>")
   }
   document.write("</pre>")
</script>
```

Utiliser les méthodes des tableaux

Etant donné la puissance des tableaux, JavaScript fournit d'emblée un certain nombre de méthodes pour les manipuler ainsi que leurs données. Voici une sélection des plus utiles.

Utiliser concat

La méthode concat concatène deux tableau ou séries de valeurs au sein d'un tableau.

```
fruit = ["Banane", "Raisin"]
legume = ["Carotte", "Chou"]
document.write(fruit.concat(legume))
```

Une autre manière d'utiliser concat permet de concaténer à un tableau des valeurs discrètes.

```
animaux = ["Chat", "Chien", "Poisson"]
plus_animaux = animaux.concat("Lapin", "Hamster")
document.write(plus_animaux)
```

Utiliser forEach

La méthode forEach en JavaScript offre une autre manière d'assurer une fonctionnalité comparable à la méthode foreach en PHP. Voici un exemple d'utilisation :

```
<script>
   animaux = ["Chat", "Chien", "Lapin", "Hamster"]
   animaux.forEach(afficher)
   function afficher(element, indice, tableau)
   {
      document.write("L'élément d'indice " + indice + " à la valeur " + element + "<br>")
   }
</script>
```

Utiliser Join

La méthode join permet de convertir toutes les valeurs contenues dans un tableau en chaine, puis de les concaténer en une grande chaine, en plaçant un éventuel séparateur facultatif entre ces éléments.

```
<script>
   animaux = ["Chat", "Chien", "Lapin", "Hamster"]
   document.write(animaux.join() + "<br>")
   document.write(animaux.join(' ') + "<br>")
   document.write(animaux.join(' : ') + "<br>")
</script>
```

Sans aucun paramètre, join utilise une virgule pour séparer les éléments ; sinon la chaine de séparation passée à join vient s'insérer entre chacun des arguments.

Utiliser push et pop

On sait déjà que la méthode push permet d'insérer une valeur dans un tableau. La méthode inverse s'appelle pop. Elle supprime le dernier élément dans un tableau et le renvoie. Voici un exemple qui illustre l'utilisation :

```
<script>
   sports = ["Football", "Tennis", "Baseball"]
   document.write("Début = " + sports + "<br>")
   sports.push("Hockey")
   document.write("Après Push = " + sports + "<br>")
   supprime = sports.pop()
   document.write("Après Pop = " + sports + "<br>")
   document.write("supprimé = " + supprime + "<br>")
</script>
```

Utiliser reverse

La méthode reverse inverse l'ordre de tous les éléments d'un tableau.

```
<script>
   sports = ["Football", "Tennis", "Baseball", "Hockey"]
   sports.reverse()
   document.write(sports)
</script>
```

Utiliser sort

La méthode sort permet de placer tous les éléments d'un tableau dans l'ordre alphabétique ou un autre, selon les paramètres passés. L'exemple suivant illustre quatre types de classements :

```
<script>
   // Tri en ordre alphabétique croissant
   sports = ["Football", "Tennis", "Baseball", "Hockey"]
   sports.sort()
   document.write(sports + "<br>")
   // Tri en ordre alphabétique décroissant
   sports = ["Football", "Tennis", "Baseball", "Hockey"]
   sports.sort().reverse()
   document.write(sports + "<br>")
   // Tri en ordre numérique croissant
   numbers = [7, 23, 6, 74]
```

```
    numbers.sort(function(a, b){return a - b})
    document.write(numbers + "<br>")
    // Tri en ordre numérique décroissant
    numbers = [7, 23, 6, 74]
    numbers.sort(function(a, b){return b - a})
    document.write(numbers + "<br>")
</script>
```

Par défaut, la fonction sort () trie les valeurs sous forme de chaînes, donc il ne peut pas faire un tri numérique. On peut corriger cela en fournissant une fonction de comparaison définit comme :

```
function(a, b){return a-b} // Pour un tri croissant
function(a, b){return b-a} // Pour un tri décroissant
```

D. JavaScript et le DOM

D.1. Accéder à des éléments HTML

L'objet Document représente votre page Web. Si on souhaite accéder à n'importe quel élément d'une page HTML, on commence toujours par accéder à l'objet Document.

L'objet Document propose plusieurs façons différentes d'accéder à un ou plusieurs éléments HTML :

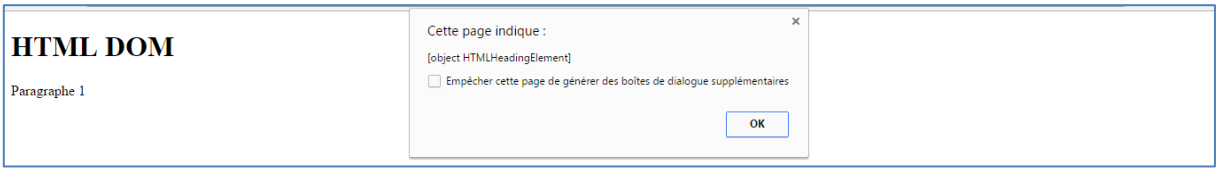
- La méthode getElementById(),
- La méthode getElementsByTagName(),
- La méthode getElementsByClassName(),
- La méthode querySelector(),
- La méthode querySelectorAll().

Accéder à des éléments HTML par identifiant

La manière la plus simple de trouver un élément HTML dans le DOM, c'est d'utiliser l'élément id. cela peut se faire via La méthode ***getElementById()*** de l'objet document. Si l'élément est trouvé, getElementById() va renvoyer l'élément en tant qu'objet. Si aucun élément n'est trouvé, la méthode renverra la valeur null.

Voici un exemple d'utilisation :

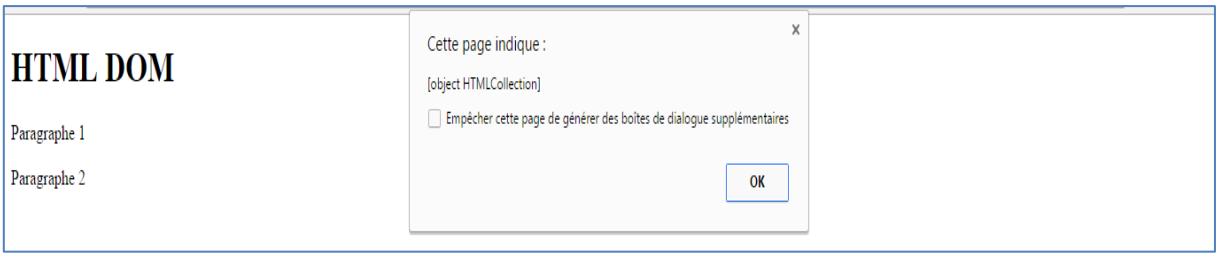
```
<!DOCTYPE html>
<html>
   <head><title> Javascript et DOM</title></head>
   <body>
      <h1 id="titre1">HTML DOM</h1>
      <p class="para">
         Paragraphe 1
      </p>
      <script>
         var titre = document.getElementById("titre1");
         alert(titre);
      </script>
   </body>
</html>
```



Accéder à des éléments HTML par nom de balise

Si on veut accéder à des éléments HTML du même type (tous les éléments p par exemple), on doit utiliser la méthode getElementsByTagName(). Cette méthode prend en argument le nom du type d'élément à récupérer et retourne des informations relatives à tous les éléments HTML ayant la même balise dans un tableau.

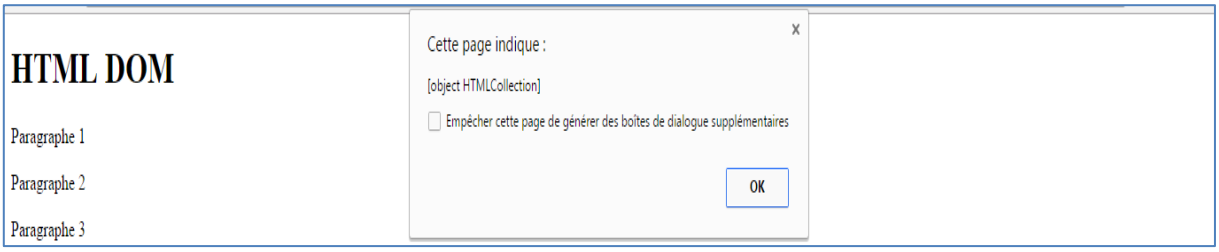
```
<!DOCTYPE html>
<html>
   <head>
      <title> Javascript et DOM</title>
   </head>
   <body>
      <h1 id="titre1">HTML DOM</h1>
      <p class="para"> Paragraphe 1</p>
      <p class="para">Paragraphe 2</p>
      <script>
         var paras = document.getElementsByTagName("p");
         alert(paras)
      </script>
   </body>
</html>
```



Accéder à des éléments HTML par nom de classe

Si on veut accéder à des éléments HTML disposant d'un attribut *class* particulier, on doit utiliser la méthode getElementsByClassName(). Cette méthode prend en argument la valeur d'un attribut *class* et retourne des informations relatives à tous les éléments HTML ayant le même attribut *class* dans un tableau.

```
<!DOCTYPE html>
   <html>
   <head><title> Javascript et DOM</title></head>
   <body>
      <h1 id="titre1">HTML DOM</h1>
      <p class="para">Paragraphe 1</p>
      <p class="para">Paragraphe 2</p>
      <p> Paragraphe 3</p>
      <script>
         var paras = document.getElementsByClassName("para");
         alert(paras)
      </script>
   </body>
</html>
```



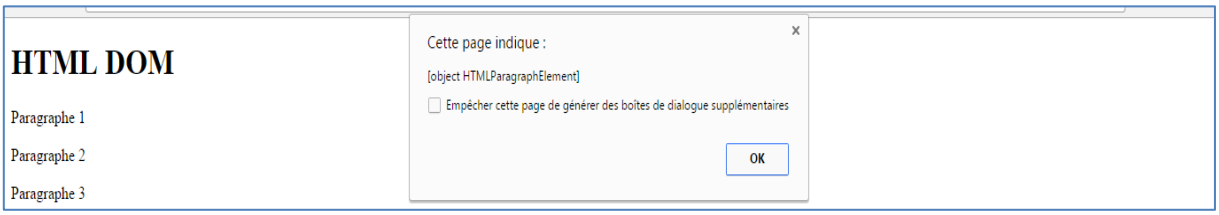
Accéder à des éléments HTML par les sélecteurs CSS

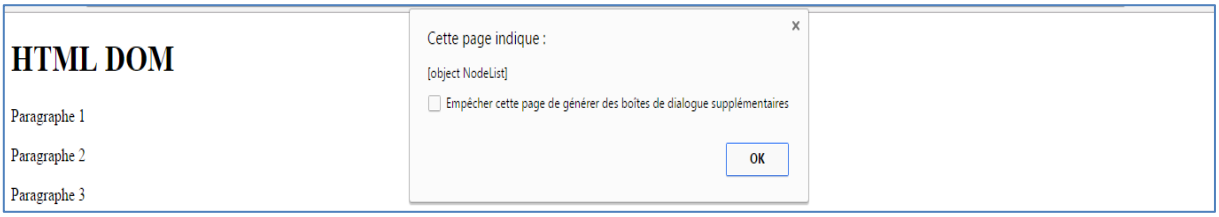
Si on veut accéder à tous les éléments HTML qui correspondent à un sélecteur CSS spécifié (id, class, type d'élément, un attribut, une valeur d'attribut, etc.), deux méthodes sont disponibles : *querySelector()* et *querySelectorAll()*.

Ces deux méthodes prennent un sélecteur CSS en argument.

querySelector() renvoie des informations relatives au premier élément trouvé correspondant au sélecteur CSS sélectionné, tandis que querySelectorAll() renvoie des informations sur tous les éléments correspondants dans un tableau.

```
<!DOCTYPE html>
<html>
   <head>
      <title> Javascript et DOM</title>
   </head>
   <body>
      <h1 id="titre1">HTML DOM</h1>
      <div>
         <p class="para">Paragraphe 1</p>
         <p class="para">Paragraphe 2</p>
         <p> Paragraphe 3</p>
      </div>
      <script>
         var paras1 = document.querySelector("p.para");
         var paras2 = document. querySelectorAll("div > p");
         alert(paras1)
         alert(paras2)
      </script>
   </body>
</html>
```





Accéder à des éléments HTML par collections d'objets HTML

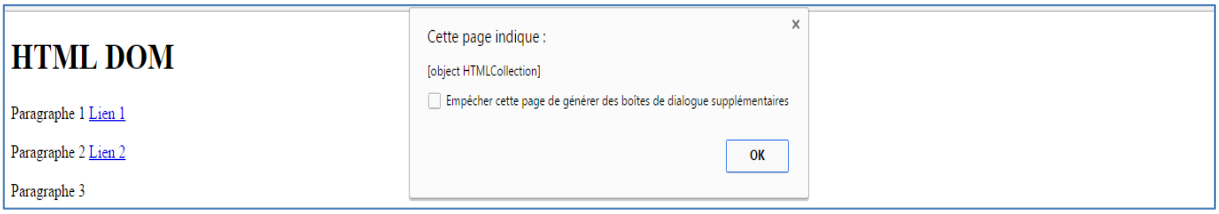
Le HTML DOM définit plusieurs objets et collections d'objets HTML standard qu'on peut y accéder via JavaScript.

Le tableau suivant énumère les objets et collections d'objets HTML les plus utilisés :

Propriété	Description
document.anchors	Renvoie tous les éléments <a> qui ont un attribut name
document.body	Renvoie l'élément <body>
document.documentElement	Renvoie l'élément <html>
document.embeds	Renvoie tous les éléments <embed>
document.forms	Renvoie tous les éléments <form>
document.head	Renvoie l'élément <head>
document.images	Renvoie tous les éléments <img>
document.links	Renvoie tous les éléments <area> et <a> qui ont un attribut href
document.scripts	Renvoie tous les éléments <script>
document.title	Renvoie l'élément <title>

Si on veut accéder à tous les éléments <area> et <a> qui ont un attribut href, on peut utiliser la collection d'objets **document.links :**

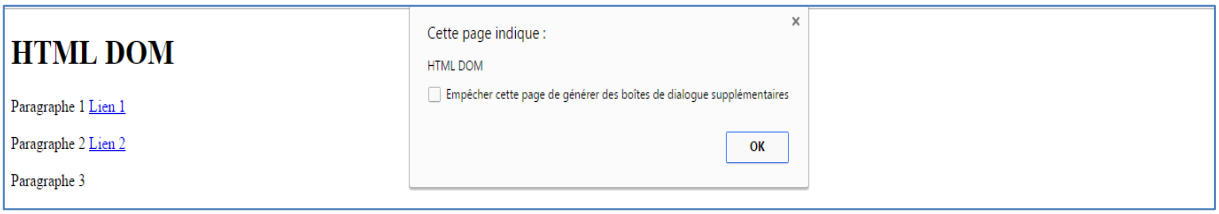
```
<!DOCTYPE html>
<html>
   <head>
      <title> Javascript et DOM</title>
   </head>
   <body>
      <h1 id="titre1">HTML DOM</h1>
      <div>
         <p class="para">Paragraphe 1 <a href="page1.html">Lien 1</a></p>
         <p class="para">Paragraphe 2 <a href="page2.html">Lien 2</a></p>
         <p>Paragraphe 3</p>
      </div>
      <script>
         var liens = document.links;
         alert(liens)
      </script>
   </body>
</html>
```



Accéder au contenu des éléments HTML et au texte

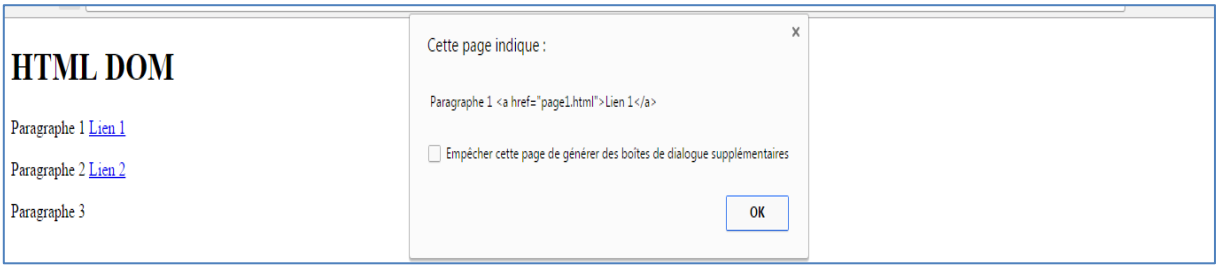
La manière la plus simple d'obtenir le contenu d'un élément HTML est d'utiliser la propriété ***innerHTML***.

```
<!DOCTYPE html>
<html>
  <head>
    <title> Javascript et DOM</title>
  </head>
  <body>
    <h1 id="titre1">HTML DOM</h1>
    <div>
      <p class="para"> Paragraphe 1 <a href="page1.html">Lien 1</a></p>
      <p class="para"> Paragraphe 2 <a href="page2.html">Lien 2</a></p>
      <p> Paragraphe 3</p>
    </div>
    <script>
      var titre = document.getElementById("titre1").innerHTML;
      alert(titre);
    </script>
  </body>
</html>
```



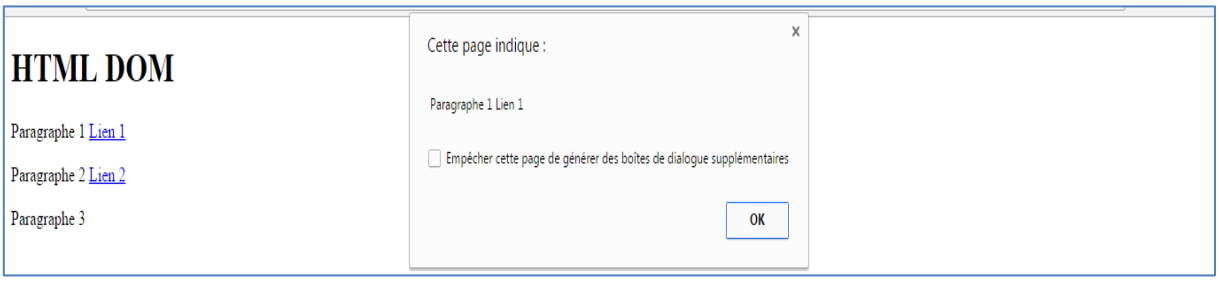
Remarquons qu'innerHTML récupère tout le contenu d'un élément HTML et pas seulement le texte contenu dans celui-ci. Donc si on applique la propriété innerHTML sur le premier paragraphe, ça va retourner le texte et le code HTML du lien, comme suit :

```
<!DOCTYPE html>
<html>
  <head>
    <title> Javascript et DOM</title>
  </head>
  <body>
    <h1 id="titre1">HTML DOM</h1>
    <div>
      <p class="para"> Paragraphe 1 <a href="page1.html">Lien 1</a></p>
      <p class="para"> Paragraphe 2 <a href="page2.html">Lien 2</a></p>
      <p> Paragraphe 3</p>
    </div>
    <script>
      var texte = document.querySelector(".para").innerHTML;alert(texte);
    </script>
  </body>
</html>
```



Si l'on souhaite ne récupérer que le contenu textuel présent à l'intérieur d'un élément, on utilisera alors la propriété **textContent**.

```
var texte = document.querySelector(".para").textContent;
alert(texte);
```



D.2. Modifier du contenu HTML

Modifier le contenu d'un élément HTML

La manière la plus simple de modifier le contenu d'un élément HTML est d'utiliser la propriété innerHTML. Pour modifier le contenu d'un élément HTML, on utilise cette syntaxe :

```
document.getElementById(id).innerHTML = nouveau HTML
```

Si on veut changer par exemple le contenu d'un élément h1, on peut écrire :

```
<html>
<body>
   <h1 id="titre1">Titre 1</1>
   <script>
      document.getElementById("titre1").innerHTML = "Nouveau Titre!";
   </script>
</body>
</html>
```

Notez que l'on peut également utiliser d'autres opérateurs d'affectation pour modifier le contenu d'un élément HTML. Par exemple, en utilisant +=, on peut rajouter du contenu.

Soit par exemple :

```
<html>
   <body>
      <h1 id="titre1">Titre 1</1>
      <script>
         document.getElementById("titre1").innerHTML += "Nouveau";
      </script>
   </body>
</html>
```

Modifier la valeur d'un attribut HTML

Pour modifier la valeur d'un attribut HTML, on utilise cette syntaxe :

```
document.getElementById(id).attribute = nouvelle valeur
```

Si on veut changer par exemple l'attribut href d'un élément <a>, on peut écrire :

```
<!DOCTYPE html>
<html>
   <body>
      <p class="para">Paragraphe 1 <a href="page1.html">Lien 1</a></p>
      <script>
         document.querySelector('a').href="https : //www.maroc.ma/";
      </script>
   </body>
</html>
```

Attention néanmoins à l'attribut class. Si on souhaite modifier sa valeur, il faudra utiliser la propriété className comme ceci :

```
<!DOCTYPE html>
<html>
   <body>
      <p class="para"> Paragraphe 1 <a href="page1.html">Lien 1</a></p>
      <p class="para"> Paragraphe 2 <a href="page2.html">Lien 2</a></p>
      <script>
         document.querySelector('.para').className="para1";
      </script>
   </body>
</html>
```

Changer Le style HTML

Pour changer le style d'un élément HTML, on utilise cette syntaxe :

```
document.getElementById(id).style.property = nouveau style
```

Si on veut changer par exemple la couleur d'un élément <h1>, on peut écrire :

```
<!DOCTYPE html>
<html>
   <body>
      <h1 id="titre1">HTML DOM</h1>
      <p class="para"> Paragraphe 1 <a href="page1.html">Lien 1</a></p>
      <p class="para"> Paragraphe 2 <a href="page2.html">Lien 2</a></p>
      <script>
         document. getElementById ('titre1').style.color="red";
      </script>
   </body>
</html>
```

D.3. Ajouter et insérer des éléments HTML

Créer un nouvel élément HTML

Pour ajouter un nouvel élément au DOM HTML, on doit d'abord créer l'élément (noeud d'élément), puis l'ajouter à un élément existant.

Pour créer un nouveau l'élément HTML, on utilise la méthode createElement() :

```
<!DOCTYPE html>
<html>
   <body>
      <h1 id="titre1">HTML DOM</h1>
      <p class="para"> Paragraphe 1 <a href="page1.html">Lien 1</a></p>
      <p class="para"> Paragraphe 2 <a href="page2.html">Lien 2</a></p>
      <script>
         document. createElement ('p');
      </script>
   </body>
</html>
```

Ajouter un attribut et du texte à un élément HTML

Pour ajouter des attributs à un nouvel élément HTML, on utilise la propriété attribute.

Pour ajouter du texte, on utilise utiliser la méthode createTextNode() :

```
<!DOCTYPE html>
<html>
   <body>
      <h1 id="titre1">HTML DOM</h1>
      <p class="para"> Paragraphe 1 <a href="page1.html">Lien 1</a></p>
      <p class="para"> Paragraphe 2 <a href="page2.html">Lien 2</a></p>
      <script>
         var nouveau_p = document.createElement('p');
         nouveau_p.id = 'para0' ;
         var texte = document.createTextNode("Nouvelle paragraphe") ;
      </script>
   </body>
</html>
```

Insérer du texte et un élément dans une page HTML

Pour insérer notre texte dans notre élément et notre élément dans le flux de notre page, on utilise la méthode appendChild().

La méthode appendChild() va insérer un objet en tant que dernier enfant d'un autre objet.

```
<!DOCTYPE html>
<html>
   <body>
      <h1 id="titre1">HTML DOM</h1>
      <p class="para"> Paragraphe 1 <a href="page1.html">Lien 1</a></p>
      <p class="para"> Paragraphe 2 <a href="page2.html">Lien 2</a></p>
      <script>
         var nouveau_p = document. createElement ('p');
         nouveau_p.id = 'para0' ;
         var texte = document.createTextNode("Nouvelle paragraphe");
         nouveau_p.appendChild(texte);
         document.body.appendChild(nouveau_p);
      </script>
   </body>
</html>
```

Insérer un élément HTML à un endroit précis

La méthode appendChild() va toujours insérer un objet en tant que dernier enfant de l'objet de type element indiqué.

Si on voudra insérer un élément dans un endroit précis d'une page HTML, on va pourra utiliser la méthode insertBefore(), qui va insérer un objet juste avant un élément comme son nom l'indique.

```
<!DOCTYPE html>
<html>
   <body>
      <h1 id="titre1">HTML DOM</h1>
      <p class="para"> Paragraphe 1 <a href="page1.html">Lien 1</a></p>
      <p class="para"> Paragraphe 2 <a href="page2.html">Lien 2</a></p>
      <script>
         var nouveau_p = document. createElement ('p');
         nouveau_p.id = 'para0' ;
         var texte = document.createTextNode("Nouvelle paragraphe");
         nouveau_p.appendChild(texte);
         var para1 = document.querySelector('.para');
         document.body.insertBefore(nouveau_p, para1) ;
      </script>
   </body>
</html>
```

D.4. Modifier ou supprimer des éléments HTML

Supprimer un élément HTML

Pour supprimer un élément HTML d'une page, on utilise la méthode removeChild().

Cette méthode supprime un élément HTML enfant ciblé relativement à son parent, donc il est primordial de connaitre le parent de l'élément pour pouvoir le supprimer.

```
<!DOCTYPE html>
<html>
   <body>
      <h1 id="titre1">HTML DOM</h1>
      <p class="para"> Paragraphe 1 <a href="page1.html">Lien 1</a></p>
      <p class="para"> Paragraphe 2 <a href="page2.html">Lien 2</a></p>
      <script>
         var titre = document.getElementById('titre1');
         var parent = document.body ;
         parent.removeChild(titre) ;
      </script>
   </body>
</html>
```

Modifier/remplacer des éléments HTML

Pour modifier ou remplacer des éléments HTML par d'autres, on utilise la méthode replaceChild().

Cette méthode prend deux arguments : la valeur de remplacement et l'élément qui doit être remplacé.

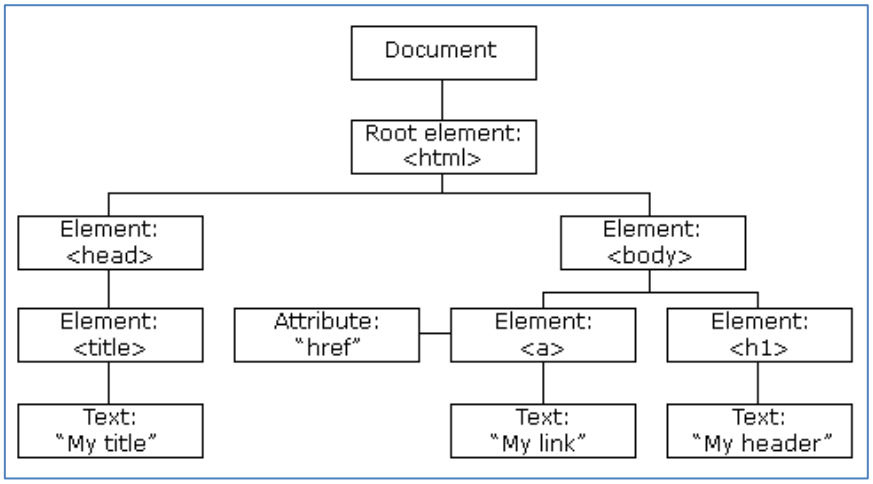
```
<!DOCTYPE html>
<html>
   <body>
      <h1 id="titre1">HTML DOM</h1>
      <p class="para"> Paragraphe 1 <a href="page1.html">Lien 1</a></p>
      <p class="para"> Paragraphe 2 <a href="page2.html">Lien 2</a></p>
      <script>
         var titre = document.getElementById('titre1');
         var parent = document.body ;
         var nouveau_titre = document.createElement('h2');
         nouveau_titre.id = 'titre2' ;
         nouveau_titre.innerHTML= 'Titre 2' ;
         parent.replaceChild(nouveau_titre, titre) ;
      </script>
   </body>
</html>
```

D.5. Naviguer dans le DOM HTML

Les nœuds du DOM

Selon la norme W3C HTML DOM, tout dans un document HTML est un nœud :

- Le document entier est un nœud de document,
- Chaque élément HTML est un nœud d'élément,
- Le texte à l'intérieur des éléments HTML sont des nœuds de texte,
- Chaque attribut HTML est un nœud d'attribut,
- Tous les commentaires sont des nœuds de commentaires.



Avec le DOM HTML, tous les nœuds de l'arbre des nœuds peuvent être accédés par JavaScript.

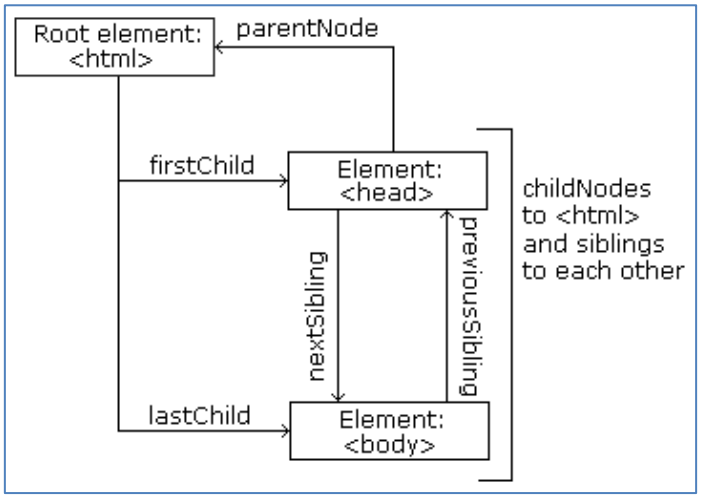
De nouveaux nœuds peuvent être créés, et tous les nœuds peuvent être modifiés ou supprimés.

Les relations entre les nœuds du DOM

Les nœuds de l'arborescence des nœuds ont une relation hiérarchique entre eux.

Les termes parent, enfant et frère ou sœur sont utilisés pour décrire les relations.

- Dans un arbre de nœud, le nœud supérieur est appelé racine (ou nœud racine),
- Chaque nœud a exactement un parent, sauf la racine (qui n'a pas de parent),
- Un nœud peut avoir un certain nombre d'enfants,
- Les frères et sœurs (Siblings) sont des nœuds avec le même parent.



La navigation entre les nœuds

On peut utiliser les propriétés de nœud suivantes pour naviguer entre les nœuds avec JavaScript :

- parentNode,
- childNodes[numéroNœud],
- firstChild,
- lastChild,
- nextSibling,
- previousSibling.

La propriété parentNode

La propriété parentNode permet d'accéder au nœud parent d'un certain nœud, et donc de se déplacer dans le DOM d'une page HTML.

```
<!DOCTYPE html>
<html>
   <body>
      <h1 id="titre1">HTML DOM</h1>
      <div>
         <p class="para"> Paragraphe 1 <a href="page1.html">Lien 1</a></p>
         <p class="para"> Paragraphe 2 <a href="page2.html">Lien 2</a></p>
      </div>
      <script>
         var p = document.querySelector('.para');
         var parent = document.body ;
         var div = p.parentNode ;
         div.style.color = 'green' ;
      </script>
   </body>
</html>
```

Les propriétés childNodes et nodeValue

ChildNodes, à l'inverse de parentNode, permet d'accéder aux nœuds enfants d'un certain nœud HTML.

Cette propriété va renvoyer un tableau contenant les enfants d'un certain nœud. On va donc pouvoir soit récupérer tous les enfants d'un coup avec une boucle, soit accéder à un élément en particulier en passant une clef à childNodes.

```
<!DOCTYPE html>
<html>
   <body>
      <h1 id="titre1">HTML DOM</h1>
      <div>
         <p class="para"> Paragraphe 1 <a href="page1.html">Lien 1</a></p>
         <p class="para"> Paragraphe 2 <a href="page2.html">Lien 2</a></p>
         <p class="para"> Paragraphe 3 </p>
      </div>
      <script>
         var div = document.querySelector('div');
         var para1 = div. childNodes[1];
         var text = div. childNodes[1].childNodes[0].nodeValue;
         alert(text)
      </script>
   </body>
</html>
```

Les propriétés firstChild et lastChild

D'autres possibilités de navigation dans le DOM sont offertes avec les propriétés firstChild et lastChild.

Ces deux propriétés vont nous permettre d'accéder respectivement au premier et au dernier enfant d'un nœud.

```
<!DOCTYPE html>
<html>
   <body>
      <h1 id="titre1">HTML DOM</h1>
      <div>
         <p class="para"> Paragraphe 1 <a href="page1.html">Lien 1</a></p>
         <p class="para"> Paragraphe 2 <a href="page2.html">Lien 2</a></p>
         <p class="para"> Paragraphe 3 </p>
      </div>
      <script>
         var p2 = document. querySelectorAll('.para')[1];
         var premier = p2.firstChild ;
         var dernier = p2.lastChild ;
         var text1 = premier.nodeValue ;
         var text2 = dernier.innerHTML ;
         alert('Premier Noeud : ' + text1 + '\nDernier Noeud : ' + text2);
      </script>
   </body>
</html>
```

Les propriétés nextSibling et previousSibling

Les propriétés nextSibling et previousSibling permet d'accéder respectivement au nœud « frère » (c'est-à-dire de même niveau) suivant le nœud ciblé ou à celui précédent le nœud ciblé.

```
<!DOCTYPE html>
<html>
   <body>
      <h1 id="titre1">HTML DOM</h1>
      <div>
         <p class="para"> Paragraphe 1 <a href="page1.html">Lien 1</a></p>
         <p class="para"> Paragraphe 2 <a href="page2.html">Lien 2</a></p>
         <p class="para"> Paragraphe 3 </p>
      </div>
      <h2 id="titre2">Titre 2</h2>
      <script>
         var div = document. querySelector('div');
         var noeud1 = div.nextSibling ;
         var noeud2 = div.previousSibling ;
         var text1 = noeud1. innerHTML ;
         var text2 = noeud2.innerHTML ;
         alert('Premier Noeud : ' + text1 + '\nDernier Noeud : ' + text2);
      </script>
   </body>
</html>
```

D.6. Les événements en JavaScript

Définition et découverte des événements

Les évènements correspondent à des actions effectuées soit par un utilisateur, soit par le navigateur lui-même.

Par exemple, lorsqu'un utilisateur clique sur un bouton HTML ou lorsque le navigateur va finir de charger une page web, on va parler d'événement.

Parfois, on va vouloir « attacher » une action spécifique à un évènement, comme par exemple modifier la taille des textes sur une page lorsque l'utilisateur clique sur un bouton.

Pour faire cela, nous allons utiliser des attributs HTML de « type » évènement, et ensuite en JavaScript créer le code correspondant à l'action que l'on souhaite attacher à notre évènement.

Les attributs HTML de type évènements sont nombreux. Parmi ceux-ci, nous avons notamment :

- L'attribut ***onclick*** : se déclenche lorsque l'utilisateur clique sur un élément,
- L'attribut ***onmouseover*** : se déclenche lorsque l'utilisateur passe le curseur de sa souris sur un élément,
- L'attribut ***onmouseout*** : se déclenche lorsque l'utilisateur sort son curseur d'un élément,
- L'attribut ***onkeydown*** : se déclenche lorsque l'utilisateur appuie sur une touche de son clavier sans la relâcher avec son curseur sur l'élément.

```
<!DOCTYPE html>
<html>
   <body>
      <h1 onclick="this.innerHTML = 'Merci!'">Cliquer ce texte!</h1>
   </body>
</html>
```

Attribuer des événements à l'aide du DOM HTML

Le DOM HTML permet d'assigner des événements à des éléments HTML à l'aide de JavaScript :

```
<script>
   document.getElementById("monBtn").onclick = afficherFacture;
</script>
```

Les événements onload et onunload

Les événements onload et onunload sont déclenchés lorsque l'utilisateur entre ou sort de la page.

L'événement onload peut être utilisé pour vérifier le type de navigateur du visiteur et la version du navigateur et charger la version appropriée de la page Web en fonction des informations.

Les événements onload et onunload peuvent être utilisés pour traiter les cookies.

```
<body onload="verifierCookies()">
```

L'événement onchange

L'événement onchange est souvent utilisé en combinaison avec la validation des champs d'entrée.

```
<input type="text" id="nom" onchange="upperCase()">
```

Les événements onmouseover et onmouseout

Les événements onmouseover et onmouseout peuvent être utilisés pour déclencher une fonction lorsque l'utilisateur effectue une souris sur ou hors d'un élément HTML.

Les événements onmousedown, onmouseup et onclick

Les événements onmousedown, onmouseup et onclick sont tous des éléments d'un clic de souris. Tout d'abord quand on clique sur un bouton de souris, l'événement onmousedown est déclenché, puis, quand le bouton de

la souris est relâché, l'événement onmouseup est déclenché, enfin, lorsque le clic de souris est terminé, l'événement onclick est déclenché.

La méthode addEventListener()

La méthode addEventListener() attache un gestionnaire d'événements à un élément sans écraser les gestionnaires d'événements existants.

Vous pouvez ajouter de nombreux gestionnaires d'événements à un seul élément.

La méthode addEventListener () facilite le contrôle de la réaction de l'événement aux bulles.

Lorsque vous utilisez la méthode addEventListener (), le JavaScript est séparé du balisage HTML pour une meilleure lisibilité et vous permet d'ajouter des auditeurs d'événements même si vous ne contrôlez pas le balisage HTML.

Vous pouvez facilement supprimer un écouteur d'événements en utilisant la méthode removeEventListener ().

La syntaxe de cette méthode est :

```
element.addEventListener(event, function, useCapture);
```

Le premier paramètre est le type de l'événement (comme «click» ou «mousedown»).

Le second paramètre est la fonction que nous voulons appeler lorsque l'événement se produit.

Le troisième paramètre est une valeur booléenne indiquant s'il faut utiliser le bullage d'événements ou la capture d'événements. Ce paramètre est facultatif.

Ajouter un gestionnaire d'événements à un élément

```
element.addEventListener("click", function(){ alert("Bonjour!"); });
```

Ou l'ajouter encore de cette manière :

```
element.addEventListener("click", maFonction);
function maFonction()
{
    alert ("Bonjour!");
}
```

Ajouter plusieurs gestionnaires d'événements au même élément

La méthode addEventListener () vous permet d'ajouter de nombreux événements au même élément, sans écraser les événements existants :

```
element.addEventListener("click", maFonction1);
element.addEventListener("click", maFonction2);
```

On peut aussi ajouter des événements de différents types au même élément :

```
element.addEventListener("mouseover", maFonction1);
element.addEventListener("click", maFonction2);
element.addEventListener("mouseout", maFonction3);
```

Ajouter un gestionnaire d'événements à l'objet Window

La méthode addEventListener () vous permet d'ajouter des écouteurs d'événement sur n'importe quel objet DOM HTML, tels que des éléments HTML, le document HTML, l'objet de fenêtre ou d'autres objets qui prennent en charge des événements, comme l'objet xmlHttpRequest.

On peut ajouter un événement qui se déclenche quand on redimensionne la fenêtre du navigateur :

```
window.addEventListener("resize", function()
{
    document.getElementById("demo").innerHTML = untexte;
});
```

Passage des paramètres avec addEventListener()

Lorsque vous passez des valeurs de paramètres, utilisez une "fonction anonyme" qui appelle la fonction spécifiée avec les paramètres :

```
element.addEventListener("click", function(){ maFonction(p1, p2); });
```

La propagation des événements : bouillonnement et capture

Il existe deux façons de propagation d'événements dans le DOM HTML, le bouillonnement et la capture. La propagation d'événement est un moyen de définir l'ordre des éléments lorsqu'un événement se produit. Si vous avez un élément <p> dans un élément , et que l'utilisateur clique sur l'élément <p>, quel événement de clic doit-on traiter ? En bouillonnant, l'événement de l'élément le plus interne est traité en premier, puis l'événement de clic externe : l'événement de clic de l'élément <p> est traité en premier, puis l'événement de clic de l'élément . Lors de la capture, l'événement de l'élément le plus externe est géré en premier, puis l'interne : l'événement de clic de l'élément sera traité en premier, puis l'événement de clic de l'élément <p>. Avec la méthode addEventListener (), vous pouvez spécifier le type de propagation en utilisant le paramètre "useCapture" :

```
addEventListener(event, function, useCapture);
```

La valeur par défaut est false, qui utilise la propagation de bulles, lorsque la valeur est définie sur true, l'événement utilise la propagation de capture.

```
document.getElementById("monP").addEventListener("click", maFonction, true);
document.getElementById("monDiv").addEventListener("click", maFonction, true);
```

La méthode removeEventListener()

La méthode removeEventListener () supprime les gestionnaires d'événements qui ont été attachés avec la méthode addEventListener () :

```
element.removeEventListener("mousemove", maFonction);
```

Propriétés et méthode de l'objet Event

L'objet Event possède des propriétés et des méthodes nous informant sur le contexte de l'évènement déclenché ou qui vont impacter l'environnement.

Notez que cet objet n'est accessible que durant le déclenchement d'un évènement. Il faut donc y accéder au sein de la fonction servant à exécuter une action lors du déclenchement de l'évènement.

Event possède une dizaine de propriétés. Nous allons voir les propriétés target, currentTarget et type.

Les propriétés target et currentTarget de l'objet Event

La propriété target va retourner le type de l'élément qui a déclenché l'évènement.

La propriété currentTarget va elle retourner le type de l'élément portant le gestionnaire de l'évènement déclenché.

Ces deux propriétés vont être très utiles pour connaître précisément quel élément possède le gestionnaire d'évènement et dans quelle phase (capture ou bouillonnement) celui-ci s'est déclenché.

Connaître l'élément déclencheur de l'évènement et celui porteur du gestionnaire d'évènement va s'avérer surtout utile dans le cas de scripts assez complexes, dans lesquels on ne peut pas accéder directement à ces éléments, ou alors, on ne peut pas les connaît pas à priori.

La propriété type de l'objet Event

La propriété type de l'objet Event va tout simplement retourner le type d'évènement qui a été déclenché.

Empêcher la propagation d'un évènement

Il est possible d'empêcher la propagation d'un événement. Pour cela, on utilise généralement la méthode stopPropagation(), qui est une méthode de l'objet Event.

Cette méthode va faire qu'un évènement va arrêter de se propager après avoir rencontré l'élément le portant.

Empêcher l'exécution d'un gestionnaire d'évènements

Pour certains évènements, on va également pouvoir empêcher tout simplement l'évènement en soi.

Pour cela, on va utiliser une autre méthode de l'objet Event : la méthode preventDefault().

Attention cependant : cette méthode ne va fonctionner qu'avec les évènements dont on peut stopper l'action.

Notez également que preventDefault() ne va en revanche pas stopper la propagation d'un évènement.

Par exemple, cela peut être utile lorsque :

- En cliquant sur un bouton "Soumettre", vous pouvez l'empêcher de soumettre un formulaire
- Cliquer sur un lien, empêcher le lien de suivre l'URL

```
document.getElementById("monancre").addEventListener("click", function(event){event.preventDefault()});
```

E. Ajax

E.1. Introduction

Ajax (Asynchronous JAvascript and Xml) est une architecture informatique qui permet de construire des applications Web et des sites web dynamiques interactifs.

Ajax combine le Javascript, les CSS, JSON, XML, le DOM et le XMLHttpRequest

Afin d'améliorer maniabilité et confort d'utilisation des applications web.

Grâce à Ajax on peut :

- Mettre à jour une page Web sans recharger la page,
- Demander des données à partir d'un serveur - une fois la page chargée,
- Recevoir des données d'un serveur - une fois la page chargée,
- Envoyer des données à un serveur - en arrière-plan.

E.2. L'objet XMLHttpRequest

Tous les navigateurs modernes prennent en charge l'objet XMLHttpRequest. L'objet XMLHttpRequest est utilisé pour échanger des données avec un serveur en coulisse. Cela signifie qu'il est possible de mettre à jour des parties d'une page Web, sans recharger la page entière.

La création d'un objet XMLHttpRequest

Tous les navigateurs modernes (Chrome, IE7 +, Firefox, Safari et Opera) ont un objet XMLHttpRequest intégré.

La syntaxe pour créer un objet XMLHttpRequest est :

```
variable = new XMLHttpRequest();
```

Les Anciennes versions d'Internet Explorer (IE5 et IE6) utilisent un objet ActiveX :

```
variable = new ActiveXObject("Microsoft.XMLHTTP");
```

Pour gérer tous les navigateurs, y compris IE5 et IE6, vérifiez si le navigateur prend en charge l'objet XMLHttpRequest. Si c'est le cas, créez un objet XMLHttpRequest, sinon, créez un ActiveXObject :

```
var xhttp;
if (window.XMLHttpRequest)
{
    xhttp = new XMLHttpRequest();
 }
else
{
    // code for IE6, IE5
    xhttp = new ActiveXObject("Microsoft.XMLHTTP");
}
```

Envoyer une requête à un serveur

Pour envoyer une requête à un serveur, nous utilisons les méthodes open () et send () de l'objet XMLHttpRequest :

```
xhttp.open("GET", "ajax_info.txt", true);
xhttp.send();
```

Méthode	Description
open(méthode, url, async)	Spécifie le type de requête méthode : le type de requête : GET ou POST url : url du fichier au serveur async : true (asynchronous) ou false (synchronous)
send()	Envoie une requête au serveur (utilisé avec GET)
send(chaine)	Envoie une requête au serveur (utilisé avec POST)

GET ou POST?

GET est plus simple et plus rapide que POST, et peut être utilisé dans la plupart des cas.

Toutefois, utilisez toujours les requêtes POST lorsque :

- Un fichier mis en cache n'est pas une option (mettre à jour un fichier ou une base de données sur le serveur),
- Envoi d'une grande quantité de données au serveur (POST n'a pas de limitations de taille),
- Envoi de l'entrée utilisateur (qui peut contenir des caractères inconnus), POST est plus robuste et sécurisé que GET.

Les requêtes GET

On peut créer une requête GET simple avec le code suivant :

```
xhttp.open("GET", "demo_get.asp", true);
xhttp.send();
```

Dans l'exemple ci-dessus, vous pouvez obtenir un résultat mis en cache. Pour éviter cela, ajoutez un ID unique à l'URL :

```
xhttp.open("GET", "demo_get.asp?t=" + Math.random(), true);
xhttp.send();
```

Si vous souhaitez envoyer des informations avec la méthode GET, ajoutez les informations à l'URL :

```
xhttp.open("GET", "demo_get2.asp?prenom=Yassir&nom=Ilhami", true);
xhttp.send();
```

Les requêtes POST

On peut créer une requête POST simple avec le code suivant :

```
xhttp.open("POST", "demo_post.asp", true);
xhttp.send();
```

Pour les données POST comme un formulaire HTML, ajoutez un en-tête HTTP avec setRequestHeader(). Spécifiez les données à envoyer dans la méthode send() :

```
xhttp.open("POST", "ajax_test.asp", true);
xhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
xhttp.send("prenom=Yassir&nom=Ilhami");
```

Le paramètre url

Le paramètre url de la méthode open () est une adresse vers un fichier sur un serveur :

```
xhttp.open("GET", "ajax_test.asp", true);
```

Le fichier peut être n'importe quel type de fichier, comme.txt et.xml, ou des fichiers de script comme «.asp» et «.php» (qui peuvent effectuer des actions sur le serveur avant d'envoyer la réponse).

Asynchronous - True ou False?

Pour envoyer la demande de manière asynchrone, le paramètre async de la méthode open() doit être défini sur true :

```
xhttp.open("GET", "ajax_test.asp", true);
```

L'envoi de requêtes asynchrones est une énorme amélioration pour les développeurs web. Beaucoup de tâches exécutées sur le serveur prennent beaucoup de temps. Avant AJAX, cette opération peut provoquer l'application pour suspendre ou arrêter.

En envoyant asynchrone, le JavaScript n'a pas à attendre la réponse du serveur, mais peut à la place :

- Exécuter d'autres scripts en attendant la réponse du serveur,
- Traiter la réponse lorsque la réponse est prête.

Async = true

Lorsque vous utilisez async = true, spécifiez une fonction à exécuter lorsque la réponse est prête dans l'événement onreadystatechange :

```
xhttp.onreadystatechange = function()
{
    if (this.readyState == 4 && this.status == 200)
   {
        document.getElementById("reponse").innerHTML = this.responseText;
   }
};
xhttp.open("GET", "ajax_info.txt", true);
xhttp.send();
```

Async = false

Pour utiliser async = false, changez le troisième paramètre de la méthode open() sur false :

```
xhttp.open("GET", "ajax_info.txt", false);
```

L'utilisation de async = false n'est pas recommandée, mais pour quelques petites requêtes cela peut être correct.

N'oubliez pas que JavaScript ne continuera PAS à s'exécuter jusqu'à ce que la réponse du serveur soit terminée. Si le serveur est occupé ou lent, l'application se bloque ou s'arrête.

Remarque : Lorsque vous utilisez async = false, n'écrivez PAS une fonction onreadystatechange, il suffit de mettre le code après l'instruction send () :

```
xhttp.open("GET", "ajax_info.txt", false);
xhttp.send();
document.getElementById("reponse").innerHTML = xhttp.responseText;
```

La propriété onreadystatechange

La propriété readyState contient l'état du XMLHttpRequest.
La propriété onreadystatechange définit une fonction à exécuter lorsque le readyState change.
La propriété status et la propriété statusText contiennent l'état de l'objet XMLHttpRequest.

Propriété	Description
onreadystatechange	Définit une fonction à appeler lorsque la propriété readyState change
readystate	Détient l'état du XMLHttpRequest. 0 : demande non initialisée 1 : connexion serveur établie 2 : demande reçue 3 : demande de traitement 4 : demande terminée et réponse est prête
status	200 : "OK" 403 : "Forbidden" 404 : "Page not found" …
statusText	Renvoie le texte d'état (par exemple "OK" ou "Non trouvé")

La fonction onreadystatechange est appelée chaque fois que le readyState change. Lorsque readyState est 4 et l'état est 200, la réponse est prête :

```
function loadDoc()
{
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function()
   {
        if (this.readyState == 4 && this.status == 200)
       {
            document.getElementById("reponse").innerHTML =  this.responseText;
        }
    };
    xhttp.open("GET", "ajax_info.txt", true);
    xhttp.send();
}
```

Remarque : l'événement onreadystatechange est déclenché cinq fois (0-4), une fois pour chaque changement dans readyState.

L'utilisation de la fonction callback

Une fonction de rappel (callback) est une fonction passée comme paramètre à une autre fonction. Si vous avez plus d'une tâche AJAX dans un site Web, vous devez créer une fonction pour exécuter l'objet XMLHttpRequest et une fonction de rappel pour chaque tâche AJAX. L'appel de fonction doit contenir l'URL et la fonction à appeler lorsque la réponse est prête.

```
loadDoc("url-1", maFonction1);
loadDoc("url-2", maFonction2);
function loadDoc(url, cFunction)
{
    var xhttp;
    xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function()
    {
        if (this.readyState == 4 && this.status == 200)
        {
            cFonction(this);
        }
    };
    xhttp.open("GET", url, true);
    xhttp.send();
}
function maFonction1(xhttp)
{
    // Les actions à exécuter
}
function maFonction2(xhttp)
{
    // Les actions à exécuter
}
```

Propriétés de réponse du serveur

Propriété	Description
responseText	Obtenir les données de réponse sous forme de chaîne
responseXML	Obtenir les données de réponse sous forme de données XML

Méthodes de réponse du serveur

Propriété	Description
getResponseHeader()	Renvoie les informations d'en-tête spécifiques de la ressource serveur
getAllResponseHeaders()	Renvoie toutes les informations d'en-tête de la ressource serveur

La propriété responseText

La propriété responseText renvoie la réponse du serveur sous la forme d'une chaîne JavaScript et vous pouvez l'utiliser en conséquence :

```
document.getElementById("reponse").innerHTML = xhttp.responseText;
```

La propriété responseXML

L'objet HttpRequest XML possède un analyseur XML intégré. La propriété responseXML renvoie la réponse du serveur en tant qu'objet XML DOM. En utilisant cette propriété, vous pouvez analyser la réponse en tant qu'objet XML DOM :

```
xmlDoc = xhttp.responseXML;
txt = "";
x = xmlDoc.getElementsByTagName("ARTIST");
for (i = 0; i < x.length; i++)
{
    txt += x[i].childNodes[0].nodeValue + "<br>";
 }
document.getElementById("demo").innerHTML = txt;
xhttp.open("GET", "cd_catalog.xml", true);
xhttp.send();
```

La méthode getAllResponseHeaders()

La méthode getAllResponseHeaders () renvoie toutes les informations d'en-tête de la réponse du serveur.

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function()
{
    if (this.readyState == 4 && this.status == 200)
   {
        document.getElementById("reponse").innerHTML = this.getAllResponseHeaders();
    }
};
```

La méthode getResponseHeader()

La méthode getResponseHeader () retourne des informations d'en-tête spécifiques de la réponse du serveur.

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function()
{
    if (this.readyState == 4 && this.status == 200)
   {
        document.getElementById("reponse").innerHTML = this.getResponseHeader("Last-Modified");
    }
};
xhttp.open("GET", "ajax_info.txt", true);
xhttp.send();
```