Plan du cours

Introduction

Chapitre 1 : c'est quoi ajax ?

- Comparaison entre web 1.0 et web 2.0
- Rich internet application
- Les parties d'ajax
- Les technologies d'ajax.

Chapitre 2 : JavaScript

- Les types de bases
- Les événements
- Quelques exemples

Chapitre 3 : Cascade Style Sheet

- Style des composants graphiques
- Style des tableaux
- Style des formulaires

Chapitre 4 : Document Object Model

- L'arbre des composants
- La navigation dans un document
- La mise a jour de la structure d'un document

Chapitre 5 : eXtensible Markup Language (XML)

- Comparaison entre HTML et XML
- La syntaxe de XML
- Le domaine d'application de XML

Introduction

Les applications web de future visent à fournir des réponses rapides aux actions des utilisateurs, et fournissent la possibilité de collaboration, création, et partage de contenu de site.

Ce type d'application est généralement appelé WEB2.0 , on peut citer faceBook , google+ , linkden ,Google map comme exemple.

facebook :

Offre une interface hyperinteractive , ou tu peux consulter les publications et les vidéos adjacentes immédiatement, en plus le contenu de facebook pourra être collecter a partir d'autre page ou autre site, des vidéos des serveurs vidéos comme youtube ainsi de suite.

1.1 Comparaison entre web 1.0 et web 2.0

Dans le WEB1.0, l'information est descendante, c'est-à-dire que le webmaster diffuse de temps en temps de l'information sur un site Web, et l'internaute ou le visiteur consulte cette information sans possibilité d'interagir avec le site -> l'information est à sens unique. Le webmaster et éventuellement quelques collaborateurs sont les seuls contributeurs. Ce WEB là est toujours présent, beaucoup de sites ne fonctionnent que de cette manière.

Le WEB2.0 est un ensemble d'usages et d'outils qui vont placer l'internaute au cœur de l'information. Dans le WEB2.0, l'internaute est contributeur. C'est pour cette raison que WEB2.0 est synonyme de WEB social, WEB collaboratif, WEB communautaire …

Le WEB1.0 se contente d'une seule plateforme (le WEB) pour diffuser de l'information.

Le WEB2.0 englobe une multitude d'outils : blog, réseaux sociaux (Facebook, Twitter, Viadéo), partage multimédias (flickr, picasa, youtube) pour partager de l'information.

Ce qu'il faut retenir également, c'est que le WEB 2.0 change notre relation avec l'information. Traditionnellement, pour avoir une information, nous la cherchons, et une fois l'information trouvée, nous l'archivons (éventuellement). C'était le cas bien avant Internet et le WEB et ce principe est resté très présent durant les 1ères années du WEB (1995-2005) -> on cherche l'info à l'aide d'un moteur, une fois la page trouvée, on l'enregistre dans nos 'favoris' (symptomatique du web1.0).

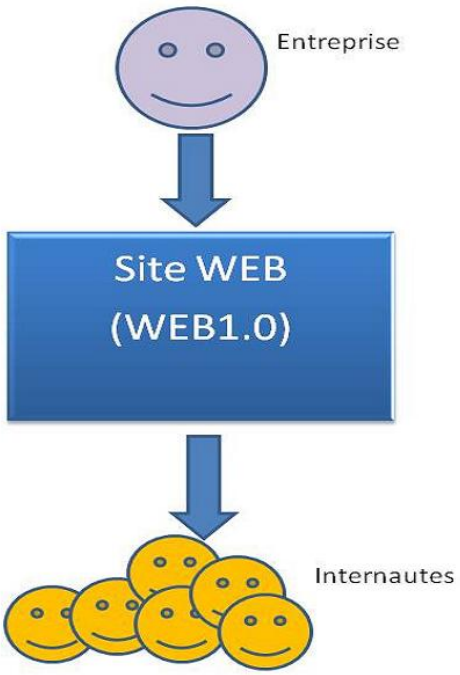
Or le principe du web2.0 et des réseaux sociaux est différent. L'idée est de faire en sorte que l'information nous parvienne plutôt que d'aller la dénicher. Et les outils ainsi que les usages vont dans ce sens :

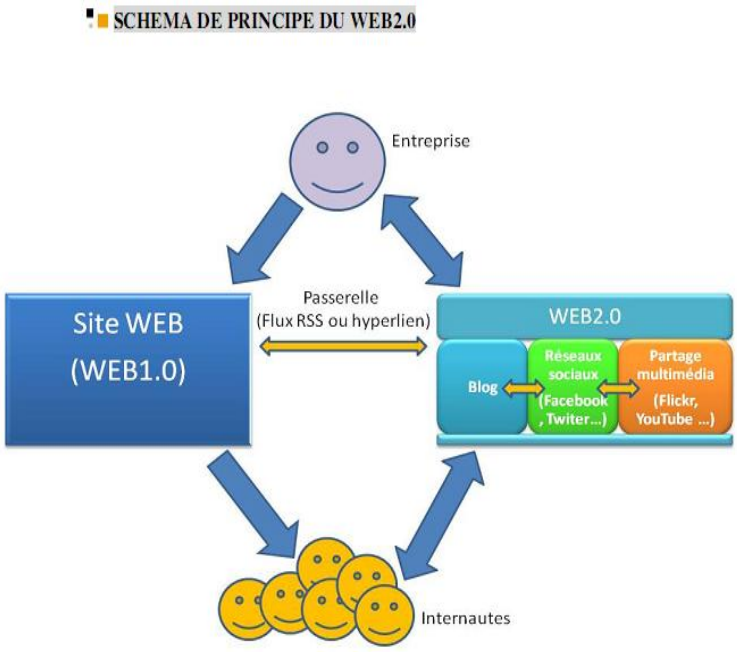
- S'abonner au flux RSS d'un blog pour recevoir automatiquement une notification signalant la présence d'un nouvel article.

- Renseigner son profil dans les réseaux sociaux pour être identifié.

- S'inscrire à un hub (à une thématique) sur Viadeo, pour recevoir toutes les publications des contributeurs de ce hub.

Sont des exemples qui montrent que nous avons tendance désormais à faire venir l'information à nous.





WEB 1.0	WEB 2.0
Sites d'Entreprises	Communautés virtuelles
Lire	Ecrire
Individuel	collectif
Compétition	Compétition collaborative
email, chat, forum	inMail, AIM, wiki, tweet, blog, social bookmark…
Inscription = Qui je suis	Profil = qui et quoi (ce que je cherche)
Utilisateur d'un site	Membre d'un réseau social
Nombre d'inscrits	Nombre de liens entre membres
Communication unilatérale avec un serveur	Communication multilatérale
Client-server	Peer to peer, peer to machine, machine / machine
HTML	XML
home page	blogs, twitter page, facebook apps,
Lectures	Conversation
Publicité	Bouche à oreille
Services vendus sur le web	Web Services pour vendre
Médias	Médias sociaux
e-business	s-business (social business)
e-commerce	s-commerce

Une riche Internet application (RIA) :

Une application Internet riche, est une application Web qui offre des caractéristiques similaires aux logiciels traditionnels installés sur un ordinateur. La dimension interactive et la vitesse d'exécution sont particulièrement soignées dans ces applications Web.

Une RIA peut être :

- exécutée sur un navigateur Internet, aucune installation n'est requise ;
- exécutée localement dans un environnement sécurisé appelé *sandbox* (bac à sable).

Le terme *Rich Internet Application* a été introduit dans une publication de Macromedia en mars 2002.

Les Pages web a des interfaces riches :

Les applications Web traditionnelles s'articulent souvent sur une architecture utilisant des clients légers : les traitements étant réalisés sur le serveur (distant), le client (local) ne faisant qu'en réaliser une présentation (exemple : HTML). Le client envoie ses données au serveur, celui-ci effectue le/les traitement(s) puis une page de réponse est renvoyée au client. Le serveur est donc sollicité à chaque interaction, hormis quelques cas spécifiques comme la saisie dans un formulaire.

Les RIA s'efforcent de rapatrier chez le client (local) une partie des traitements normalement dévolus au serveur. Le langage Javascript en particulier a été conçu dans cette optique. Il permet par exemple d'indiquer au fur et à mesure de la frappe, le nombre de caractères qu'il est encore possible de saisir, dans un champ de texte de taille limitée, plutôt que de tout accepter et de renvoyer ensuite seulement un message d'erreur du serveur avec perte d'une partie du contenu frappé.

Les standards Internet ont évolué lentement et continuellement à travers le temps pour s'accommoder de ces techniques, aussi il est difficile de définir clairement ce qui constitue une RIA et ce qui n'en constitue pas une. Généralement, ce qui peut être effectué au moyen d'une RIA est limité par les capacités du système client.

Parce que les RIA utilisent les ressources du système client, elles offrent aux applications Web des possibilités d'interfaces utilisateur en plus réactives, ce qui serait impossible avec des balises HTML standards.

On peut déporter sur le client de nombreuses fonctionnalités, comprenant le glisser-déposer, l'utilisation de barres d'outils pour modifier les données, des calculs (par exemple : taux d'intérêt pour un prêt), données n'ayant pas nécessairement besoin d'être renvoyées au serveur

Avantages et inconvénients :

Bien que le développement d'applications s'exécutant dans un navigateur Web en limite la portée, bien que ce soit une tâche difficile à mettre en œuvre, et bien que l'on ajoute un degré de complexité supplémentaire pour développer des applications équivalentes aux applications bureautiques classiques, ces efforts sont souvent récompensés parce que :

- Aucune installation n'est nécessaire—la mise à jour et la distribution de l'application est un processus instantané et transparent pour l'utilisateur ;
- Les utilisateurs peuvent utiliser l'application depuis n'importe quel ordinateur équipé d'une connexion Internet et d'un navigateur récent ;
- Étant donné que l'utilisation du Web s'accroît, les utilisateurs d'ordinateur sont de moins en moins enclins à installer de nouveaux logiciels lorsqu'une alternative se basant sur le navigateur (qu'il ne faudra pas installer) est disponible.

Ce dernier point est généralement vrai, même si cette alternative est lente ou qu'elle est dépouillée. Un bon exemple de ce phénomène est l'utilisation du webmail.

Que sont les applications riches (RIA) ? - Partie 1 - Définition et usages.

Le web est un milieu qui évolue rapidement on le sait. L'écosystème qu'il constitue change en fonctions des usages que l'on en fait ou des technologies utilisées. L'une des évolutions majeures qu'il va connaître sera sans aucun doute les applications dites "riches", RIA, RDA ou encore "Internet Riche". Bref beaucoup de dénominations derrière un concept trop souvent jugé complexe dans sa compréhension tout comme dans sa définition. Il est donc temps pour moi de commencer à vous en parler un peu plus souvent.

Définition générale de l'Internet Riche

L'*Internet riche* est le terme souvent employé pour qualifier toutes les **nouveaux types d'interface**, d'**ergonomie** ou d'**usages** que l'on trouve sur Internet. D'un scope très large, ce terme a du mal à trouver une définition exacte car il ne s'agit pas d'un concept nouveau, apparu à une date précise mais plutôt, et comme bien souvent, d'évolutions faites petit à petit sur le Web. Chaque acteur créant une nouvelle application web regarde toujours ce que la concurrence a fait précédemment et c'est en reprenant de bonnes idées et en les améliorant que les grandes évolutions voient le jour. C'est un peu de cette manière que l'Internet riche s'est construit mois après mois. Si des ergonomies nouvelles et par conséquent des usages nouveaux d'Internet ont été les fondations de ce que l'on nomme Internet riche, ce concept englobe aujourd'hui beaucoup de choses et de notions différentes. Que cela soit en fonction de la plateforme sur lesquels les services sont développés : navigateur Internet pour RIA, système d'exploitation directement pour RDA ou que cela soit pour ce qui est des technos qui ont vu le jour autour de ce concept ou encore pour les designs qui ont suivi cette tendance de très près (les services dit *riches* sont souvent faits de manières à faciliter l'ajout de composant personnalisables), l'Internet riche regroupe aujourd'hui une multitude de concepts divers et variés trouvant un axe commun dans la **volonté de changer les usages et de rendre la navigation toujours plus simple et intuitive**.
Le terme (imaginé par Macromedia dans un White Paper en **mars 2002** et disponible ici (PDF) pour ceux que cela intéresse) est donné également en comparaison des interfaces que l'on trouvait sur le Web il y a 5 ans. Celles-ci étaient conçu à l'époque de manière très carrée, sans originalité entre les différents sites web et avec le désir de s'occuper plus du fond que de la forme du site. Aujourd'hui, sans négliger le fond bien évidemment, la forme d'un site et son utilisation (re)deviennent deux éléments essentiels dans la conception d'application web. Les RIA permettent d'apporter alors un lot impressionnant de possibilités supplémentaires pour cela. Par ailleurs, on peut être sûr que, d'ici 3-4 ans, ces concepts seront devenus monnaie courante et que l'on se souviendra de l'Internet riche comme une évolution des usages plutôt que comme une révolution d'Internet à part entière.
Enfin et par abus de langage, il est commun d'appeler RIA tout ce qui englobe l'Internet riche même si ce n'est en réalité qu'une partie de toutes les (r)évolutions de l'Internet riche.

Qu'est ce que RIA ?

Les RIA pour *Rich Internet Application* offrent de **nouvelles possibilités aux utilisateurs en les ouvrants au plus grand nombre**. Au delà d'un aspect "rich media", c'est à dire permettant par exemple l'utilisation de vidéos, ou de musique facilement, il faut ajouter une nouvelle perception de la navigation : le modèle de page en page n'existe plus. Les codes ont changés : un bouton de formulaire ne va pas forcément recharger toute la page, il peut par exemple avoir une influence sur une partie de la page ou charger une image. Ce sont des concepts très bien repris par Gmail par exemple, en simulant des comportements d'un vrai client mail, tout en offrant les avantages du net : accès à un même point, de n'importe où. Cela permet donc de décentraliser les données en ligne. Les technologies utilisées ont permis une adaptation simple des utilisateurs car les comportements que l'on retrouve sont plutôt mimés par rapport à une application classique.

Qu'est ce que RDA ?

Les RDA pour *Rich Desktop Application* apportent ce que l'on trouve sur le web sur le bureau pour une**meilleure expérience utilisateur**. Ainsi avec cette déportation d'Internet, les possibilités sont plus grandes : l'intégration avec le système d'exploitation est meilleure, de nouvelles fonctionnalités sont disponibles en s'affranchissant du navigateur, ou encore l'application peut fonctionner même déconnectée.
Internet est basé sur le langage de description HTML. Des acteurs ont poussés cette vision (Mozilla, Macromedia) pour définir des interfaces avec plus de composants, plus de possibilités donc. Ces technologies sont devenues de véritables technologies d'interface, non pas simplement destinées à l'Internet mais également à pouvoir décrire toute interface souhaitée. En prenant cette philosophie et en l'appliquant à un domaine où les interfaces sont plutôt lourdes à créer (sur le bureau), les acteurs offrent de nouveaux outils bénéficiant de plusieurs avantages, venus de plusieurs mondes :

- Le fait d'être connect&eacute
  ;, pour
  garder ses informations décentralisées sur Internet.
- La possibilité d'être autonome en s'affranchissant du navigateur : une plus grande liberté pour beaucoup de possibilités, comme le stockage en local de données ou le fonctionnement en mode déconnecté.
- L'utilisation d'outils web (très porté sur la description d'interface) pour faciliter la création d'applications plus lourdes, c'est à dire les applications de bureau.

Il faut voir les RDA comme une ouverture des technologies de RIA, un domaine où plus de possibilités sont présentes pour voir le véritable potentiel exploitable des technologies. Le résultat ? Des applications aux interfaces plus faciles à créer, avec une utilisation plus aisée.

Quels usages des RIA peut-on imaginer ?

L'Internet riche apporte et va apporter une quantité de nouveaux usages qui n'ont de limite que l'imagination des webmasters et développeurs Web. En effet, ces nouvelles interfaces et les technologies qui en découlent permettent d'envisager de nombreuses nouvelles utilisations du web comme par exemple de la retouche photo, de la vidéo ou toute manipulation de contenu "riche" (élément 3D, carte enrichie, etc.) directement à travers un site Internet. Et même si toutes ces nouveautés commencent à peine à arriver, les interfaces web d'hier évoluent déjà petit à petit pour profiter des avantages des RIA comme par exemple la fin du rafraichissement des pages web, le glisser-déposer, la présentation d'informations de type graphiques ou documents avec une réelle interactivité possible, le parcours client simplifié et plus intuitif dans un site de vente en ligne, etc.

Vous l'aurez compris les usages des RIA sont infinis et sont en passe de révolutionner le web d'hier pour le rendre plus accessible et plus intuitif à utiliser. En augmentant l'interactivité avec le visiteur, les créateurs de sites Internet rendent le web de demain encore plus attractif et simple d'emploi.

Conclusion

On l'a dit, l'Internet riche propose une magnifique évolution des usages et des interfaces du web d'hier. En pleine croissance et bien que lancé dès 2002, ce concept n'a pas fini de faire parler de lui. Les plus grands acteurs du web l'ont très vite compris (Google en tête) en développant les outils et services nécessaires à l'envol de ce concept sur le web d'aujourd'hui. De plus, les enjeux qui s'en dégagent semblent énormes pour l'industrie du web à en croire les différentes études que Interfaces-Riches.com annonce :

- D'après une étude de Scene7, **90% des marketers** déclarent **vouloir intégrer** les technologies riches sur leur site pour améliorer l'expérience utilisateur.
- D'ici 2008, d'après Gartner Group, **48% des banques** auront déployé des **clients riches**.
- D'après Forrester, **70% des utilisateurs** plébiscitent l'**ergonomie** de ces nouvelles interfaces.

J'espère que ce premier article vous aura permis de clarifier les choses sur ce sujet assez complexe. Il m'a permis moi de synthétiser les idées que j'en avais et je vous invite à revenir sur le blog pour les deux prochaines parties de cet article à savoir une présentation des technologies RIA à travers des exemples concrets de RIA et enfin quelques réponses quant au future des RIA et de l'Internet riche.

Chapitre 1 : C'est quoi AJAX ?

C'est un ensemble de technologies qui permettent à un site d'être réactif , ces technologies se basent sur une mise à jour asynchrone et partielle du contenu de la page web.

AJAX ; est une abréviation de " Asynchronous javascript and xml", et un type de "Rich Internet Application".

La mise a jour partielle "rafraichissement partiel".

C'est par exemple quand un utilisateur entre des informations dans un formulaire et clique sur submit, le serveur traite les informations reçus et retourne une réponse limitée concernant, les informations reçus pratiquement il n'envoie pas la page entière.

L'ancien type était appelé "click, wait, and refresh", qui est orienté page.

Et AJAX(web 2.0) orienté environnement , ou donnée , elles ne sont pas limitées aux données envoyées par formulaire, au contraire elles sont réactive.

Le déplacement de la souris sur la page, la saisie partielle.

3-les technologies qui composent AJAX:

Cascading Style Sheets (CSS) : est un langage de balisage definissent le style de presentation d'une page "fonts , et couleur".

Javascript , est un langage de scriptage exemple l'objet "XMLHTTPRequest" objet utilisée pour le transfert de donnée entre le web client et le web serveur.

Document object Model (DOM) : qui donne une vue logique de la page comme un arbre de contenu.

XML , c'est le format de données envoyés entre client et serveur.

4- les parties d'ajax :

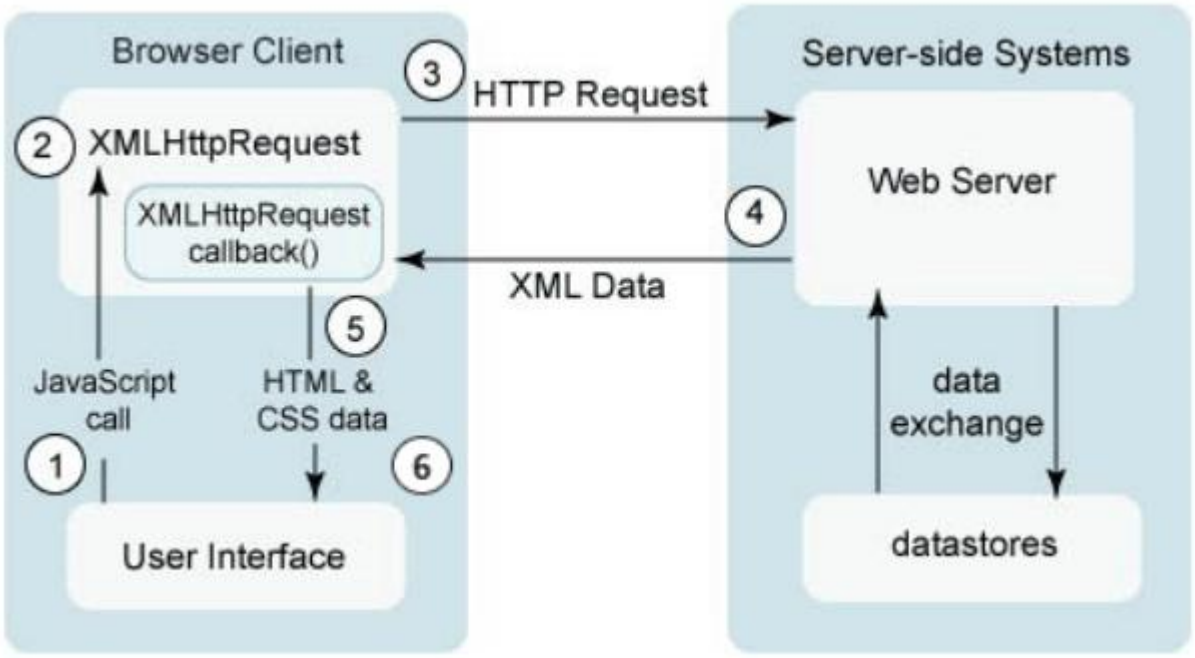
RIA est composé de deux parties client/serveur.

La partie client : utilise HTML ou XHTML pour présenter le contenu de la page web. Et le javascript comme langage de programmation.

La partie serveur : basé sur "JSP" javaserver page , qui utilise "javaEE" qui support la validation de données , et l'amenagement de l'identité d'utilisateur.

Le XML une langage intermédiaire de communication entre le client et le serveur.

5-Le fonctionnement d'RIA:



L'utilisateur genère un envenement sur la partie cliente, se qui provoque un appel de type javascript.

Une fonction de javascript créer et configure un objet "XMLHttprequest" et spécifie une fonction de retoure.

L'objet "XMLHttpRequest " fait un appel asynchrone au sereveur.

Le serveur web traite la requette et retourne le resultat dans fichier XML.

Affiche le resultat de l'objet XMLhttpRequest.

Le client mise a jour la dom representant la page avec les nouvelle données.

Les stratégies des développement

Faites tout le travail tout seul :

Vous faites le codage a partir du zero pour le client et le serveur

Avantage

Donne un contrôle bas niveau

Inconvénient

Nécessite beaucoup de codage

Nécessite la connaissance de beaucoup de langage javascript , css, dom, jsp.

La connaissance des incompatibilités de multiple navigateur

Utilisation de bibliothèque de javascript:

On peut utiliser des bibliothèques de javascript pour le coté client , telque : dojo tool kit, prototype, sript.aculo.us,rico.

7-JavaScript & page (X)HTML

Le Javascript a une histoire très intéressante. Il est originairement né en **1995** d'une guerre entre **Microsoft** et **Netscape**. **Brendan Eich** (futur CTO de Mozilla) a donc dû créer ce langage **en seulement 10 jours** !

L'objectif ? Avoir un langage moins orienté "entreprise" et permettre à tout développeur naissant de pouvoir interagir avec le navigateur sans avoir à se lancer dans l'apprentissage du langage Java.

La tâche n'était pas simple, car il fallait que Brendan Eich invente un langage assez simple et léger pour que tout un chacun puisse l'utiliser, mais qu'il soit tout de même assez sophistiqué pour que des développeurs confirmés puissent l'utiliser pour créer des applications puissantes.

Le prototye s'appelais **Mocha**, puis le nom officiel de lancement devint **LiveScript**. Etant donné que la consigne de base était que ce nouveau langage ressemble à Java, rapidement LiveScript devient Javascript. Les développeurs avaient donc Java côté serveur, et Javascript côté navigateur.

Malgré son utilisation très répandue, Javascript a beaucoup souffert d'une mauvaise image. Au début il était utilisé mais personne n'en enseignaient les bonnes pratiques. Il était donc considéré comme un langage "**pour amateur**". Ce n'est que dans les années 2000 que les développeurs ont compris le potentiel de ce langage et comment l'utiliser pour améliorer l'expérience utilisateur.

Deux événements majeurs ont changés ce point de vue :

- **L'explosion des frameworks MVC** (Modèle-Vue-Controlleur) Javascript, qui apporte des bonnes pratiques et une structure solide pour réaliser des applications;
- **L'apparition de bibliothèques telles que JQuery ou encore Mootols**, disponibles à partir de 2005/2006, permettent une utilisation massive de Javascript sans pour autant comprendre les concepts utilisés du langage. Elles ont permis de faciliter grandement l'écriture de petits scripts à l'écriture d'application complètes;
- **La possibilité d'utiliser le Javascript également côté serveur grâce à Node.** Un seul langage pour les coder tous, le développement s'en trouve grandement faciliter !

7.1 Code a inclure dans une page web

<script langage="JavaScript">
... code ...
</script>

Code a inclure dans un fichier séparé
<script langage="JavaScript"
type="text/javascript"
src="URL">
</script>

L'url est le chemin vers le fichier
Le code JavaScript est chargé à l'URL indiquée.

8- les types de base

- Nombres
- Chaînes de caractères
- Booléens
- Fonctions
- Objets
- Array
- Null, undefined

8.1.Nombres

Tous les nombres sont des flottants (pas d'entiers), en 64 bits (IEEE 754).
Opérateurs habituels : +, -, /, *, %
Math :
Math.PI, Math.E, ...
Math.abs(x), Math.cos(x), Math.sqrt()
Math.ceil(x), Math.floor(x), Math.round(x)
Math.random()
Conversions : parseInt("123"), parseFloat("3.14")
NaN, Infinity
1/0 == Infinity
NaN + 1 == NaN (fonction: isNaN(x))

8.2 Séquences de caractères.

"Bonjour"
Les chaînes sont des objets :
n = "Bonjour".length

Les caractères sont des chaînes de longueur 1 :
« b » = "bruit".charAt(0)
Nombreuses méthodes utiles (voir doc)
"Feu rouge".replace("rouge", "vert")
"Bobo".toUpperCase()

8.3 Null, undefined et booléens

Deux types spéciaux ayant une seule valeur :

null : variable sans valeur (comme en SQL)

undefined : variable non initialisée (ou non existante)

Booléens : true, false

Opérateurs logiques &&, ||, !

8.4 Variables

Déclaration d'une variable :

Utiliser var :

var x;

var pi = 22/7; // approximatif

Toujours utiliser var, sinon la variable est globale, et c'est le début de la

fin...

Si la variable n'est pas initialisée, elle est égale à undefined.

8.5 Opérateurs

+, -, *, /, %

+=, -=, *=, /=

a++, b--, --a, ++a

"Salut " + "Toto" == "Salut Toto"

"3" + 45 == 48

8.6 les structures de contrôles

Comme en C ou Java...

(if, while, do { ... } while (); switch)

Boucle for : for (var i =0; i < 10; i++) { ... }

Exceptions : try/catch/finally

8.7 les Objets

Deux notations équivalentes :

var obj = new Object();

var obj = {}

Accès aux membres (“propriétés”) :

obj.nom = "Toto";

obj["name"] = "Toto";

(notez que "name" peut être une variable)

Notation :

```
var obj = {
        nom : "Toto";
        prenom : "Marcel";
         age : 22;
}
```

8.8 Tableaux (array)

Les tableaux sont juste une classe spéciale d’objets. Les clés sont des

nombres (indices).

Les tableaux sont donc hétérogènes :

```
var A = new Array();
A[0] = "Fromage";
A[1] = 3.12;
A[2] = new Object();
// ici A.length == 3
```

Autre notation :

var A = ["Fromage", 3.12, new Object()]

Ajout en fin :

A[A.length] = 12;

8.9 Fonctions

Très simple, arguments non typés statiquement :

function ajouter(x, y) {

var somme = x + y;

return somme;

}

Si pas de valeur retournée, la fonction retourne undefined ;

si un argument manque, il est aussi undefined.

Université de Skikda Skikda, le 02/10/2013

3 eme LMD Pro Réseaux et Systèmes Distribués l'enseignant : BOUTINE Rachid

Module : AJAX

Série de TD N°01

Exercices

1 Ecrire une page web qui lors de son chargement affiche un message

"Bienvenue" (fenêtre popup) lors de son chargement ;.

2 Ecrire une page web qui lors de son déchargement affiche un message

"au revoir" (fenêtre popup) lors que l'utilisateur quitte la page.

3 Modifier le script pour qu'il affiche successivement les phrases

contenues dans un tableau. Par exemple si var Messages = ["Bienvenue", "sur ce site", "Merci?"];

la page affichera successivement 3 fenêtres popups.

4 Ecrire un script qui affiche au démarrage une question sur l'heure (matin ou apres midi)?

5 Ecrire un script qui affiche un formulaire qui récupère des informations personnelles, qui s'assure de la validité de votre âge?

Université de Skikda Skikda, le 14/10/2018

3 eme LMD (ISIL) l'enseignant : BOUTINE Rachid

Module : Programmation web avancée

Série de TP N°01

1 Ecrire une page web qui affiche deux champs nom et temps ;.

```
<html>
<body>
<form name="myForm">
Name: <input type="text" name="username" />
Time: <input type="text" name="time" />
</form>
</body>
</html>
```

2 initialiser un objet XMLHttpRequest?

```
<script type="text/javascript">
function ajaxFunction() { var xmlhttp;
if (window.XMLHttpRequest)
  {   // code for IE7+, Firefox, Chrome, Opera, Safari
  xmlhttp=new XMLHttpRequest();   }
else if (window.ActiveXObject)   {  // code for IE6, IE5
  xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");  }
else   {  alert("Your browser does not support XMLHTTP!");  } }
</script>
```

3 utiliser la methode open et send de cet objet pour envoyer des donnes vers le serveur?

```
xmlhttp.open("GET","http:\\localhost :8080\ HelloWorld",true);
xmlhttp.send(null);
```

4 spécifier la fonction retour?

```
xmlhttp.onreadystatechange=function()
{ if(xmlhttp.readyState==4)   {
  document.myForm.time.value=xmlhttp.responseText;   } }
```

5 Ecrire un script d'une servlet pour le serveur , dont le role est de retourner au client l'heure courante

```
package myservlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.Date;

public class HelloWorld extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        // set header field first
        res.setContentType("text/html");
        // then get the writer and write the response data
        PrintWriter out = res.getWriter();
   Date df = new Date();
        out.println( df.getDay() + "/"+df.getMonth()+ "/"+df.getYear() );
      out.close();    }
    public String getServletInfo() {
        return "A simple servlet";
    }}
```

Série de TP N°01

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

public class actionTP1 extends JFrame implements ActionListener {
	public actionTP1() {
	super(); setSize(300, 200);
	JButton jButton = new JButton("OK");
	JButton jButton2 = new JButton("envoyer");

	jButton2.addActionListener(new ActionListener() {
		public void actionPerformed(ActionEvent e) {
			System.out.println("Envoyer");
		}

	});

	jButton.addActionListener(this);
	JPanel p1=new JPanel() ;
	p1.add(jButton);

	this.getContentPane().add(p1);
	}
	@Override
	public void actionPerformed(ActionEvent e) {
		// TODO Auto-generated method stub
		System.out.println("bonjour");
	}
	public static void main(String[] args){
		SwingUtilities.invokeLater(new Runnable() {public void run() {actionTP1 thisClass
= new actionTP1();thisClass.setVisible(true);}});


 }
}
```

Série de TP N°02

```
import javax.swing.SwingUtilities;
import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.awt.event.WindowFocusListener;
import java.awt.event.WindowStateListener;
import javax.swing.*;
public class TP1 extends JFrame{
public TP1(){
super();
 setSize(300, 200);
JButton jButton = new JButton("OK");
jButton.addActionListener(new java.awt.event.ActionListener(){public void
actionPerformed(java.awt.event.ActionEvent e) {System.exit(0);}});
JPanel p1=new JPanel() ;
p1.add(jButton);
this.getContentPane().add(p1);
addWindowListener(new WindowAdapter(){
       public void windowClosing(WindowEvent e){
       System.out.println("exit");  System.exit(0);}
       public void windowClosed(WindowEvent e){ System.out.println("windowClosed");}
       public void windowOpened(WindowEvent e){System.out.println("windowOpened");}
       public void windowActivated(WindowEvent e){ System.out.println("Activated");}
       public void windowDeactivated(WindowEvent e){ System.out.println("DeActivated");}
       public void windowDeiconified(WindowEvent e){ System.out.println("Deiconified");}
       public void windowIconified(WindowEvent e){System.out.println("Iconified");}});

// focus Listener
addWindowFocusListener(new WindowAdapter(){
public void windowGainedFocus(WindowEvent e){ System.out.println("windowGainedFocus");}
public void windowLostFocus(WindowEvent e){
       System.out.println("WindowFocusListener method called: windowLostFocus.");} });

//state changed
addWindowStateListener(new WindowAdapter(){
       public void windowStateChanged(WindowEvent e) {
System.out.println( "WindowStateListener method called: windowStateChanged."+ e);}});

setLocationRelativeTo(null);
//setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
public static void main(String[] args){
SwingUtilities.invokeLater(new Runnable() {public void run() {TP1 thisClass = new
TP1();thisClass.setVisible(true);}}); }

}
```

Série de TD N°02

Exercices

1 Ecrire un script HTML qui affiche une calculatrice dans une page web?

(trois zones de texte pour opérande 1 , opérande 2 , et résultat , quatre boutons +,-,*,/)

2 Ecrire un script en JavaScript dans un fichier séparé qui gère le fonctionnement de la calculatrice ?

3 Ecrire un script CSS dans un fichier séparé qui donne une présentation conviviale à la calculatrice ?

4 Ecrire un script html d'un relevé de note de trois matières (intitulé matière, note td, note control, moyenne module, moyenne globale ?

5 réutiliser les scripts (js, css) développés auparavant ?

6 Ecrire un script css spécial pour l'impression du relevé de note sur une imprimante noire et blanc

Université de Skikda Skikda, le 01/02/2010

3 eme LMD Pro Réseaux et Systèmes Distribués l'enseignant : BOUTINE Rachid

Module : AJAX

Série de TD N°02

Exercices

1 Ecrire une page web JSP qui affiche la date courante ?

2 Ecrire une page web JSP qui affiche un tableau de 20 lignes et 2 colonnes contenants les valeurs entre 40 et 60 ?

3 Ecrire une classe pile dans java , ensuite utiliser cette classe avec une page web JSP ?

4 Ecrire un script qui vous affiche un message d'accueil selon votre sexe ?

5 Ecrire un script qui affiche une calculatrice normal (+,-,/,*) , avec une couleur de l'arrière plan de la zone résultat qui se change selon l'opération réalisée ?

Série de TP N°01

1 Ecrire une page web qui affiche deux champs nom et temps ;.

```
<html>
<body>
<form name="myForm">
Name: <input type="text" name="username" />
Time: <input type="text" name="time" />
</form>
</body>
</html>
```

2 initialiser un objet XMLHttpRequest?

```
<script type="text/javascript">
function ajaxFunction()
{ var xmlhttp;
if (window.XMLHttpRequest)
  {   // code for IE7+, Firefox, Chrome, Opera, Safari
  xmlhttp=new XMLHttpRequest();   }
else if (window.ActiveXObject)
  {  // code for IE6, IE5
  xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");  }
else
  {  alert("Your browser does not support XMLHTTP!");  } }
</script>
```

3 utiliser la methode open et send de cet objet pour envoyer des donnes vers le serveur?

```
xmlhttp.open("GET","time.asp",true);
xmlhttp.send(null);
```

4 spécifier la fonction retour?

```
xmlhttp.onreadystatechange=function()
{ if(xmlhttp.readyState==4)   {
  document.myForm.time.value=xmlhttp.responseText;   } }
```

5 Ecrire un script en php pour le serveur , dont le role est de retourner au client l'heure courante

```
<?php
// Date in the past
header("Expires: Mon, 26 Jul 1997 05:00:00 GMT");
header("Cache-Control: no-cache");
echo(date("G:i:s",time()));
?>
```

6 développer votre code pour qu'il donne des suggestions à l'utilisateur sur les noms sauvegarder dans le serveur?

9-Les événements

Généralités

Avec les événements et surtout leur gestion, nous abordons le côté "**magique**" de Javascript.
En Html classique, il y a un événement que vous connaissez bien. C'est le clic de la souris sur un lien pour vous transporter sur une autre page Web. Hélas, c'est à peu près le seul. Heureusement, Javascript va en ajouter une bonne dizaine, pour votre plus grand plaisir.

Les événements Javascript, associés aux fonctions, aux méthodes et aux formulaires, ouvrent grand la porte pour une réelle **interactivité** de vos pages.

Les événements

Passons en revue différents événements implémentés en Javascript.

Description	Evénement
Lorsque l'utilisateur clique sur un bouton, un lien ou tout autre élément.	**Click**
Lorsque la page est chargée par le browser ou le navigateur.	**Load**
Lorsque l'utilisateur quitte la page.	**Unload**
Lorsque l'utilisateur place le pointeur de la souris sur un lien ou tout autre élément.	**MouseOver**
Lorsque le pointeur de la souris quitte un lien ou tout autre élément. Attention : Javascript 1.1 (donc pas sous MSIE 3.0 et Netscape 2)	**MouseOut**
Lorsqu'un élément du formulaire a le focus, c-à-d devient la zone d'entrée active.	**Focus**
Lorsqu'un élément de formulaire perd le focus, c-a-d que l'utilisateur clique hors du champ et que la zone d'entrée n'est plus active.	**Blur**
Lorsque la valeur d'un champ de formulaire est modifiée.	**Change**
Lorsque l'utilisateur sélectionne un champ dans un élément de formulaire.	**Select**
Lorsque l'utilisateur clique sur le bouton *Submit* pour envoyer un formulaire	**Submit**

Les gestionnaires d'événements

Pour être efficace, il faut qu'à ces événements soient associées les actions prévues par vous. C'est le rôle des gestionnaires d'événements. La syntaxe est

onévénement="fonction()"

Par exemple, **onClick="alert('Vous avez cliqué sur cet élément')"**.
De façon littéraire, au clic de l'utilisateur, ouvrir une boite d'alerte avec le message indiqué.

Onclick

Evénement classique en informatique, le clic de la souris.

Le code de ceci est :
<FORM>
<INPUT TYPE="button" VALUE="Cliquez ici" **onClick="alert('Vous avez bien cliqué ici')"**>
</FORM>
Nous reviendrons en détail sur les formulaires dans le chapitre suivant.

onLoad et onUnload

L'événement Load survient lorsque la page a fini de se charger. A l'inverse, Unload survient lorsque l'utilisateur quitte la page.

Les événements onLoad et onUnload sont utilisés sous forme d'attributs de la balise <BODY>ou <FRAMESET>. On peut ainsi écrire un script pour souhaiter la bienvenue à l'ouverture d'une page et un petit mot d'au revoir au moment de quitter celle-ci.

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE='Javascript'>
function bienvenue() {
alert("Bienvenue à cette page");
}
function au_revoir() {
alert("Au revoir");
}
</SCRIPT>
</HEAD>
<BODY onLoad='bienvenue()' onUnload='au_revoir()'>
Html normal
</BODY>
</HTML>
```

onmouseOver et onmouseOut

L'événement onmouseOver se produit lorsque le pointeur de la souris passe au dessus (sans cliquer) d'un lien ou d'une image. Cet événement est fort pratique pour, par exemple, afficher des explications soit dans la barre de statut soit avec une petite fenêtre genre infobulle.

onFocus

L'événement onFocus survient lorsqu'un champ de saisie a le focus c.-à-d. quand son emplacement est prêt à recevoir ce que l'utilisateur à l'intention de taper au clavier. C'est souvent la conséquence d'un clic de souris ou de l'usage de la touche "Tab".

Si vous cliquez dans la zone de texte, vous effectuez un focus

onBlur

L'événement onBlur a lieu lorsqu'un champ de formulaire perd le focus. Cela se produit quand l'utilisateur ayant terminé la saisie qu'il effectuait dans une case, clique en dehors du champ ou utilise la touche "Tab" pour passer à un champ. Cet événement sera souvent utilisé pour vérifier la saisie d'un formulaire.

Si après avoir cliqué et/ou écrit dans la zone de texte, vous cliquez ailleurs dans le document, vous produisez un événement Blur.

Le code est :

```
<FORM>
<INPUT TYPE=text onBlur="alert('Ceci est un Blur')">
</FORM>
```

Onchange

Cet événement s'apparente à l'événement onBlur mais avec une petite différence. Non seulement la case du formulaire doit avoir perdu le focus mais aussi son contenu doit avoir été modifié par l'utilisateur.

Onselect

Cet événement se produit lorsque l'utilisateur a sélectionné (mis en surbrillance ou en vidéo inverse) tout ou partie d'une zone de texte dans une zone de type text ou textarea.

Gestionnaires d'événement disponibles en Javascript

Il nous semble utile dans cette partie "avancée" de présenter la liste des objets auxquels correspondent des gestionnaires d'événement bien déterminé.

Objets	Gestionnaires d'événement disponibles
Fenêtre	onLoad, onUnload
Lien hypertexte	onClick, onmouseOver, on mouseOut
Elément de texte	onBlur, onChange, onFocus
Elément de zone de texte	onBlur, onChange, onFocus
Elément bouton	onClick
Case à cocher	onClick
Bouton Radio	onClick
Liste de sélection	onBlur, onChange, onFocus, onSelect
Bouton Submit	onClick
Bouton Reset	onClick

La syntaxe de onmouseOver

Le code du gestionnaire d'événement onmouseOver s'ajoute aux balises de lien :

<A HREF="" onmouseOver="action()">lien</A>

Ainsi, lorsque l'utilisateur passe avec sa souris sur le lien, la fonction action() est appelée. L'attribut HREF est indispensable. Il peut contenir l'adresse d'une page Web si vous souhaitez que le lien soit actif ou simplement des guillemets si aucun lien actif n'est prévu. Nous reviendrons ci-après sur certains désagréments du codage HREF="".

Voici un exemple. Par le survol du lien "message important", une fenêtre d'alerte s'ouvre.

Le code est :

```
<BODY>
...
<A HREF="" onmouseOver="alert('Coucou')">message important</A>
...
<BODY>
```

ou si vous préférez utiliser les balises <HEAD>

```
<HTML>
<HEAD>
<SCRIPT language="Javascript">
function message(){
alert("Coucou")
}
</SCRIPT>
</HEAD>
<BODY>
```

```
<A HREF="" onmouseOver="message()">message important</A>
</BODY>
</HTML>
```

La syntaxe de onmouseOut

Tout à fait similaire à onmouseOver, sauf que l'événement se produit lorsque le pointeur de la souris quitte le lien ou la zone sensible.

Au risque de nous répéter, si onmouseOver est du Javascript 1.0 et sera donc reconnu par tous les browsers, onmouseOut est du Javascript 1.1 et ne sera reconnu que par Netscape 3.0 et plus et Explorer 4.0 et plus (et pas par Netscape 2.0 et Explorer 3.0)

On peut imaginer le code suivant :

```
<A HREF="" onmouseOver="alert('Coucou')" onmouseOut="alert('Au revoir')">message important</A>
```

Les puristes devront donc prévoir une version différente selon les versions Javascript.

Problème! Et si on clique quand même...

Vous avez codé votre instruction onmouseOver avec le lien fictif <A HREF=""... >, vous avez même prévu un petit texte, demandant gentiment à l'utilisateur de ne pas cliquer sur le lien et comme de bien entendu celui-ci clique quand même.
Horreur, le browser affiche alors l'entièreté des répertoires de sa machine ou de votre site). Ce qui est un résultat non désiré et pour le moins imprévu.

Pour éviter cela, prenez l'habitude de mettre l'adresse de la page encours ou plus simplement le signe # (pour un ancrage) entre les guillemets de HREF. Ainsi, si le lecteur clique quand même sur le lien, au pire, la page encours sera simplement rechargée et sans perte de temps car elle est déjà dans le cache du navigateur.
Prenez donc l'habitude de mettre le code suivant <A HREF="#" onmouseOver="action()"> lien </A>.

Changement d'images

Avec le gestionnaire d'événement onmouseOver, on peut prévoir qu'après le survol d’une image par l'utilisateur, une autre image apparaisse (pour autant qu'elle soit de la même taille).
le code est relativement simple.

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="Javascript1.1">
function lightUp() {
document.images["homeButton"].src="button_hot.gif"
}
```

```
function dimDown() {
document.images["homeButton"].src="button_dim.gif"
}
</SCRIPT>
</HEAD>
<BODY>
<A HREF="#" onmouseOver="lightUp();" onmouseOut="dimDown();">
<IMG SRC="button_dim.gif" name="homeButton" width=100 height=50 border=0> </A>
</BODY>
</HTML>
```

Compléter toujours en Javascript les attributs width=x height=y de vos images.

Il n'y a pas d'exemple ici pour la compatibilité avec les lecteurs utilisant explorer 3.0 en effet, non seulement onmouseOut mais aussi image[] est du Javascript 1.1.

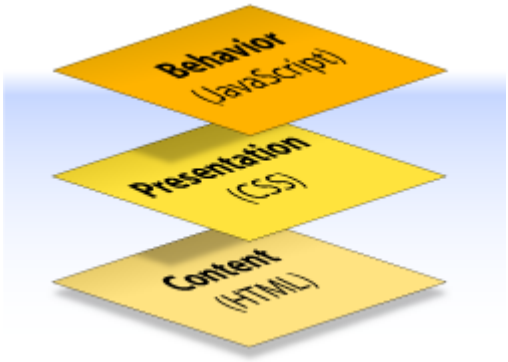
L'image invisible

Ce changement d'image ne vous donne-t-il pas des idées?... Petit futé! Et oui, on peut prévoir une image invisible de la même couleur que l'arrière plan (même transparente). On la place avec malice sur le chemin de la souris de l'utilisateur et son survol peut ,à l'insu de l'utilisateur, déclencher un feu d'artifice d'actions de votre choix. Magique le Javascript ?

4 CSS :

une page web peut être vue comme construite de trois couches :

1- Couche contenue : c'est l'ensemble d'information que l'auteur voudrait afficher , contenu dans les balises html , la plupart de ce contenu est de texte, ou d'image , ou de video.
2- Couche présentation : détermine la façon d'affichage de contenu.
3- Couche comportement : concerne l'interaction entre l'utilisateur et la page assurée par le java script.



Il est possible de fusionner les trios couches ou de les laisser séparées, ce qui est le meilleur, parce qu'on peut changer une couche ou la modifier sans toucher les autres. Par exemple on peut définir la présentation de 10000 pages web par un seul fichier css.

Avantage :

-cela simplifier la maintenance de l'affichage d'un seul fichier au lieu de 10000 pages.

- la vitesse de chargement sera améliorée parce que le fichier de css est charger une seule fois a la mémoire.

- le contenu peut être utilisé pour d'autre taches.

-

What Is CSS?

Have you ever thought about what a web page is? I mean, what it *really* is? Some people think of a web page as a visual medium—an aesthetically pleasing experience which may or may not contain information that's of interest to the viewer. Other people think of a web page as a document that may be presented to readers in an aesthetically pleasing way. From a technical point of view, the document interpretation is more appropriate.

When we examine the elements of its construction, a web document can consist of up to three layers—content, presentation, and behavior—as illustrated in Figure 1.

The **content layer** is always present. It comprises the information the author wishes to convey to his or her audience, and is embedded within HTML or XHTML markup that defines its structure and semantics. Most of the content on the Web today is text, but content can also be provided through images, animations, sound, video, and whatever else an author wants to publish.

The **presentation layer** defines how the content will appear to a human being who accesses the document in one way or another. The conventional way to view a web page is with a regular web browser, of course, but that's only one of many possible access methods. For

example, content can also be converted to synthetic speech for users who have impaired vision or reading difficulties.

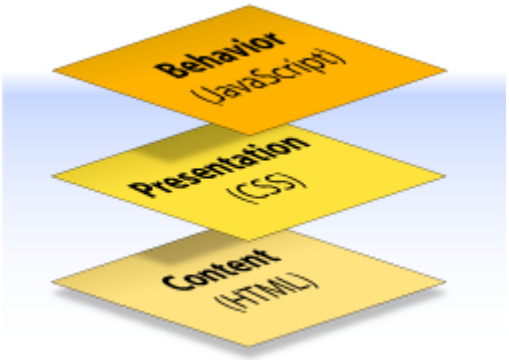


Figure 1. The three layers of a web document

The **behavior layer** involves real-time user interaction with the document. This task is normally handled by JavaScript. The interaction can be anything from a trivial validation that ensures a required field is filled in before an order form can be submitted, to sophisticated web applications that work much like ordinary desktop programs.

It's possible to embed all three layers within the same document, but keeping them separate gives us one valuable advantage: we can modify or replace any of the layers without having to change the others.

Certain versions of HTML and XHTML also contain **presentational element types**—that is, elements that specify the appearance of the content, rather than structure or semantics. For example, <b> and <i> can be used to control the presentation of text, and <hr> will insert a visible rule element. However, as these types of elements embed presentation-layer information within the content layer, they negate any advantage we may have gained by keeping the layers separate.

Cascading Style Sheets, or **CSS**, is the recommended way to control the presentation layer in a web document. The main advantage of CSS over presentational HTML markup is that the styling can be kept entirely separate from the content. For example, it's possible to store all the presentational styles for a 10,000-page web site in a single CSS file. CSS also provides far better control over presentation than do presentational element types in HTML.

By externalizing the presentation layer, CSS offers a number of significant benefits:

- All styling is kept in a limited number of style sheets. The positive impact this has on site maintenance can't be overestimated—editing one style sheet is obviously more efficient than editing 10,000 HTML files!
- The overall saving in bandwidth is measurable. Since the style sheet is cached after the first request and can be reused for every page on the site, it doesn't have to be downloaded with each web page. Removing all presentational markup from your web pages in favor of using CSS also reduces their size and bandwidth usage—by more than 50% in many documented

cases. This benefits the site owner, through lower bandwidth and storage costs, as well as the site's visitors, for whom the web pages load faster.

- The separation of content from presentation makes it easier for site owners to reuse the content for other purposes, such as RSS feeds or text-to-speech conversion.
- Separate styling rules can be used for different output media. We no longer need to create a special version of each page for printing—we can simply create a single style sheet that controls how every page on the site will be printed.

Although CSS is designed to be independent of the markup language of the docume

General Syntax and Nomenclature

In this section, we'll describe the building blocks of a CSS style sheet and the correct syntax for each part. We'll also define the unavoidable jargon we'll use throughout this reference. When everyone uses the same term for the same thing, communication is usually easier and less error prone.

CSS syntax is not rigid: whitespace can usually be added freely between tokens, and line breaks have no semantic value.

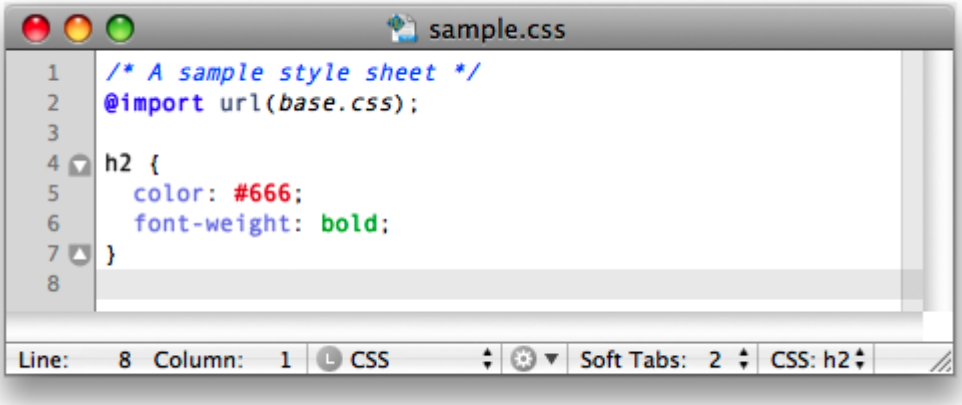
CSS is case insensitive in all matters under its control. However, some things lie outside the control of CSS and these may or may not be case sensitive, depending on external factors such as markup language and operating system.

Element type names, for instance, are case insensitive for HTML but case sensitive for XML (including XHTML served as XML). Font names, with the exception of the generic font family CSS keywords, may be case sensitive on some operating systems.

Disambiguating the Nomenclature

In order to name the various items that make up CSS syntax, let's consider the example in Figure 1.

Figure 1. Sample CSS syntax



The example begins with a comment:

```
/* A sample style sheet */
```

The comment is followed by two statements. The first statement is an at-rule:

```
@import url(base.css);
```

The second statement is a rule set:

```
h2 {
  color: #666;
  font-weight: bold;
}
```

The rule set consists of a selector (the text before the left curly brace, {) and a declaration block (delimited with the curly braces, { … }). The block contains two declarations separated by semicolons:

```
  color: #666;
  font-weight: bold;
```

Each declaration includes a property name and a value, separated by a colon.

The Cascade, Specificity, and Inheritance

Other than being the C in the acronym CSS, the fact that style sheets are described as "cascading" is an important, if complex, part of the way styles are applied to the elements in a document. It's called the CSS cascade, because style declarations cascade down to elements from many origins.

The cascade combines the importance, origin, specificity, and source order of the applicable style declarations to determine exactly—and without conflict—which declaration should be applied to any given element.

Inheritance is the means by which, in the absence of any specific declarations applied by the CSS cascade, a property value of an element is obtained from its parent element.

CSS Layout and Formatting

While CSS1 didn't have much to offer for the graphical layout of documents, CSS2 has introduced several new properties for layout, and CSS3 will probably add even more. Although CSS still doesn't provide total control over the page layout, it's far more powerful than the old-school technique of using layout tables and presentational markup.

A web browser typically reads and renders HTML documents. This happens in two phases: the **parsing phase** and the **rendering phase**.

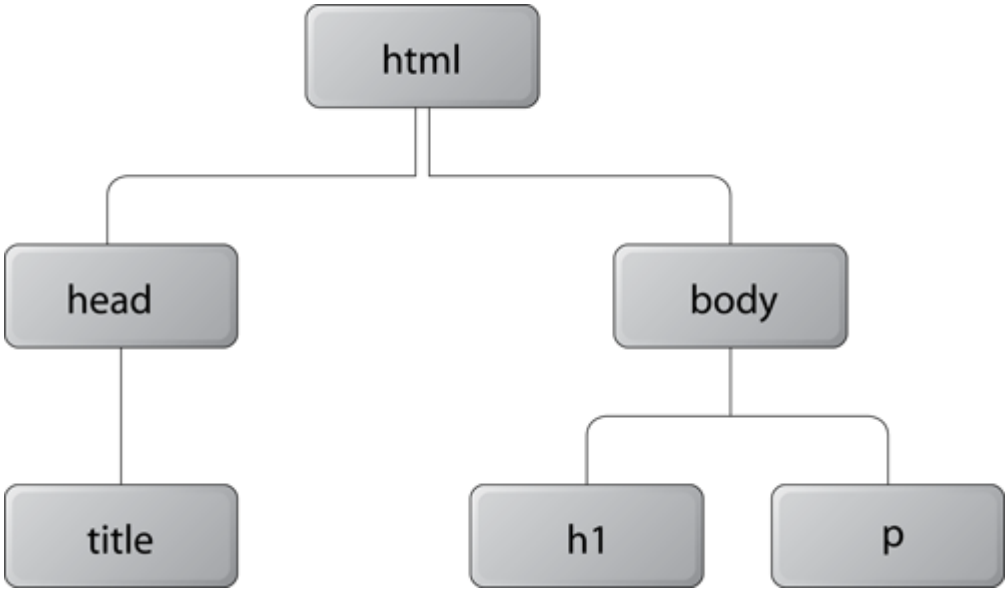
During the parsing phase, the browser reads the markup in the document, breaks it down into components, and builds a document object model (DOM) tree.[1]

Consider this example HTML document:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <title>Widgets</title>
  </head>
  <body>
    <h1>Widgets</h1>
    <p>Welcome to Widgets, the number one company
    in the world for selling widgets!</p>
  </body>
</html>
```

The above HTML document can be visualized as the DOM tree in Figure 1 (in which the text nodes have been omitted for clarity).

Figure 1. The DOM tree



Each object in the DOM tree is called a **node**. There are several types of nodes, including element nodes and text nodes. At the top of the tree is a document node, which contains an element node called the **root node**; this is always the htmlelement in HTML and XHTML documents, and it branches into two child element nodes—head and body—which then branch into other children.

A **child node** is structurally subordinate to its **parent node**. In HTML terms, this means that the child's tags are nested inside the tags of the parent. For example, we can see in Figure 1 that the h1 element is a child node of the body element and the body element is the parent node of the h1 element. A node can be called a**descendant node** if it's a child, grandchild, and so on, of another node. A node can be called an **ancestor node** if it's a parent, grandparent, and so on, of another node. For example, the h1 element is a descendant node of the html element, and the html element is an ancestor node of the h1 element. Nodes that have the same parent are called **siblings**. The h1 and p elements are sibling nodes.

When the DOM tree has been constructed, and any CSS style sheets have been loaded and parsed, the browser starts the rendering phase. Each node in the DOM tree will be rendered as zero or more boxes.

Just as there are block-level elements and inline elements in HTML, there are block boxes and inline boxes in CSS. In fact, there are several other box types, but they can be seen as subtypes of the block and inline boxes.

A CSS box is always rectangular.[2] It has four sides with a 90° angle between each of them.

From a somewhat simplified perspective, we can say that it's the user agent style sheet which specifies that block-level HTML elements generate block boxes, while inline-level HTML elements generate inline boxes. We can, of course, use thedisplay property to change the type of the box generated for any element.

CSS does not, however, affect the markup in any way. The separation into block-level and inline elements in HTML is specified in the HTML document type definition, and cannot be changed. For example, setting the display property to block for aspan element doesn't allow us to nest an h1 element inside it, because the HTML document type definition forbids it.

Workarounds, Filters, and Hacks

Unfortunately, as you deal with CSS you'll eventually discover differences in the way user agents apply and render CSS rules. These differences can be caused by the user agents' varying interpretations of, and levels of support for, the CSS standards, as well as rendering problems and bugs. But—luckily for us—they can be addressed using workarounds, filters, and hacks.

If you search the Web for "CSS hacks," you'll find numerous sites and articles from as far back as 2001 describing ways to tackle browser-related CSS problems. These problems were discovered once people started attempting to create completely CSS-based web design and layout. Happily, modern browser support for CSS is fairly good, so many of those old-school hacks are no longer needed. Older browsers have fallen into disuse and workarounds for problematic browsers that are still in use are well documented.

All software has bugs. Browsers are no exception to this rule, but some browsers are certainly buggier than others. In the past, some bugs related to browsers' CSS rendering caused web pages to become unreadable, and in some cases, they even crashed browsers. It's also true that browsers don't provide perfect support for CSS—a fact that's often the cause of much frustration. Of course, the situation was far worse in the past, when levels of support could differ wildly.

Workarounds, Filters, and Hacks Defined

Once CSS-based layout and design became popular, web designers and developers needed a way to supply different CSS rules to different browsers—a capability that's absent from CSS. A **hack** has typically been regarded as a temporary, inelegant, or unadvised solution to a problem. But in CSS terms, applying a hack generally means exploiting incorrect or buggy CSS features in order to target or exclude a browser, or group of browsers, so that alternative styling may be applied to them.

Other techniques—often called **workarounds** or **filters**—include targeting the proprietary features of a specific browser, or employing advanced CSS features to exclude older browsers that don't support the newer features. If all this jargon's getting a bit much for you, just remember that workarounds are CSS-oriented solutions to these problems, while filters and hacks are browser-oriented solutions.

The Problem with Workarounds, Filters, and Hacks

While it's often tempting to leap in and apply a complicated hack to force a particular browser to behave, a more careful approach is needed to address CSS problems efficiently. First, you need to make sure that the problem you're addressing is a real CSS problem—not just the result of incorrect CSS code or an incomplete understanding of CSS. If your web page looks as you intended in one browser but not another, you may be tempted to think that the browser that's not displaying your site properly has a CSS bug, but of course the exact opposite is equally likely.

Consider, for instance, the fact that different browsers apply varying default margin and padding values to HTML elements like headings and list items. You'll often see sites on which CSS hacks are used to apply particular rules to different browsers simply because the designers weren't aware of the variations in these values. The use of CSS hacks in these kinds of situations is redundant; simply spending a few minutes to gain an understanding of the margin and padding rules would negate the need to apply hacks.

If you're sure that you have a valid CSS rendering problem, and you're tempted to use a hack, first see whether a change of design could enable you to avoid the issue altogether. If you can design layouts that don't depend on problematic CSS features, in most cases you won't need hacks at all.

The Internet Explorer 5 box model problem is a famous example of the unnecessary use of hacks. Many complicated hacks were developed to solve this problem, but with a simple design change—the addition of padding to the parent of an element with a fixed width, instead of to the element itself—designers could have avoided the problem altogether.[1] This approach wasn't possible in every case, but the option was there.

Avoiding Implementation Pitfalls

If you find yourself in a position where you have no choice but to use a workaround, filter, or hack, be aware of the dangers involved. Your chosen hack may be unreliable—in the future, it may actually cause more problems than would have resulted had you not used it at all. As newer browser versions are produced, new features are implemented, and bugs are fixed, the hack mechanism you've been using may cease to work. Also consider the maintenance issues that can arise when many hacks are spread throughout a style sheet.

In reality, the only completely safe way to use a browser hack is to target **dead browsers**—those browsers that are no longer in development, like Internet Explorer 6—and target them in such a way that you can be sure the hacks you're using will continue to work in that browser.

Don't apply hacks to newer browsers, such as Firefox 2, and Opera 9—they're updated regularly, and new features and bug fixes are addressed relatively quickly. It's just not safe to use a hack for these newer browsers, and usually they don't need it anyway—even if they

do need adjustment, a change of design will often accommodate any deficiencies you find. Finally, whenever you use a hack, you face the difficulty of finding one that will work on just the browser you're targeting without affecting all the others. Let us tell you now: in the end, it's a fruitless pursuit. That's why the modern approach is to attempt to shun hacks altogether.

Using conditional comments is now the recommended way to target various versions of Internet Explorer; a number of workaround techniques that don't rely on ugly hacks are also available. Finally, we've included a list of popular CSS hacks here, not because they're recommended, but in case you come across them and need to understand what they attempt to achieve.

il y a de multiples avantages à séparer les feuilles de styles de contenu.
1 La réduction de la taille des pages : Les définitions de style ne sont faites qu'une seule fois, même si elles sont utilisées plusieurs fois ;
2 La réduction des temps de connexion : Les navigateurs garderont en mémoire (en cache) le contenu de la feuille de style CSS qui s'appliquera sur toutes les pages du site. Seuls les contenus des pages devront être chargés au cours de la navigation ;
3 Une mise à jour plus facile : Vous n'aurez besoin que de changer la feuille de style pour mettre à jour la présentation de l'ensemble de votre site ;
4 Scinder le travail de rédaction et le travail de présentation : Vous pouvez commencer à rédiger le contenu de vos pages sans vous soucier de leur présentation finale. Pensez simplement à placer correctement vos balises sémantiques (titre, sous-titres, listes, classes et ID, etc.). Vous pourrez travailler votre mise en page et votre design plus tard.

Les CSS représentent une nouvelle façon très efficace d'appliquer des styles aux éléments (X)HTML.
Elles vous permettent de définir n'importe quelle propriété de style comme la bordure, le type de caractère, la couleur de fond, l'espace entre les lettres, etc. (nous reviendrons plus tard sur la manière d'y parvenir).
Il y a trois façons principales d'appliquer des styles CSS :

1 Dans le corps du code (X)HTML ;
2 Dans l'en-tête de la page ;
3 Dans une feuille de style totalement séparée du code (X)HTML.

II-A - Les CSS dans le corps du code (X)HTML (à utiliser avec modération)

Vous pouvez définir des styles CSS directement dans la définition d'une balise (X)HTML. Dans l'exemple ,nous utilisons une balise qui permet de définir une "boîte" à l'intérieur d'un contenu :

Exemple de code

```
<div style="background-color:orange; border:1px  solid black; color:yellow; font-size:150%;  padding:1em;">

  Cette balise div a du style !

</div>
```

Cette approche est extrêmement proche de l'ancienne façon de définir des styles et présente les mêmes inconvénients.

Elle ne présente un intérêt que lorsque vous êtes certain que le style défini ne sera utilisé à aucun autre endroit ni sur aucune autre de vos pages. S'il y a la moindre chance pour que

vous ayez à nouveau besoin de ce style à un autre endroit, vous devriez absolument utiliser l'une des deux autres méthodes proposées plus bas, afin de faciliter la maintenance et l'évolution de votre site.

Les CSS dans l'en-tête de la page

Plutôt que par la méthode précédente, il est préférable de définir vos styles CSS une fois pour toute dans une section particulière de votre page Web (on utilise normalement la section <head>).

```
<head>
  <style type="text/css">
    Div
    {
      background-color:#339;
      color:#fff;
      padding:15px;
      border-bottom:5px solid red;
      margin-bottom:15px;
    }
  </style>
</head>
<body>
  <div>
    Cette phrase est présentée en fonction du style défini dans l'en-tête
  </div>
  <div>
    Cette phrase aussi, est pourtant le style n'a été défini qu'une fois !
  </div>
</body>
```

Grâce à cette nouvelle façon de procéder, vous n'avez besoin de définir votre style qu'une seule fois. Dans notre exemple, le style défini s'appliquera automatiquement à toutes les balises de la page.

Avec cette méthode, vous pouvez appliquer le même style plusieurs fois dans la même page, mais pas à plusieurs pages d'un coup. Pour aller plus loin dans la standardisation de vos pages, vous devrez utiliser la troisième méthode.

Les CSS dans une feuille de style totalement séparée du code (X)HTML

1-La façon idéale de définir les CSS consiste à les enregistrer dans un document indépendant de vos pages (X)HTML.

Grâce à cette méthode, toutes les pages qui font référence à cette feuille de style externe hériteront de toutes ses définitions.

Un autre intérêt de cette méthode est de pouvoir définir plusieurs feuilles de styles pour le même contenu et de basculer d'une feuille à l'autre en fonction du support sur lequel le contenu est affiché (écran, imprimante, etc.).

Document 'mes-styles.css'

```
body
{   background-color:#ccf;
  letter-spacing:.1em; }
p {
  font-style:italic;
  font-family:times,serif; }
```

Document 'ma-page.html'

```
<head>
  <link href="mes-styles.css" media="all" rel="stylesheet" type="text/css" />
</head>
```

```
<body>
  <p>Voici un exemple de paragraphe.</p>
  <p>Et voici un deuxième paragraphe.</p>
</body>
```

La méthode "<link href=..." permet également de mettre en place plusieurs feuilles de styles destinées aux différents medias (imprimante, navigateurs de PDA, etc.).

Déclaration	périphérique
<link href="general.css" rel="stylesheet" type="text/css" media="all">	Commune a toute les medias
<link href="ecran.css" rel="stylesheet" type="text/css" media="screen, projection">	ecran
<link href="mobile.css" rel="stylesheet" type="text/css" media="handheld">	PDA , mobile
<link href="impression.css" rel="stylesheet" type="text/css" media="print">	imprimante

2-Une deuxième déclaration :

Cette définition de style est à placer dans la section <head> de votre page.

```
<style type="text/css">
  @import url(styles.css) all;
</style>
```

Vous devrez remplacer "styles.css" par le nom que vous souhaitez donner à votre feuille de style. Vous pouvez également remplacer "all" par le type de média auquel se destine votre feuille de style.

Les Tableaux

Comment faire un tableau à bordures finess !? Sans les feuilles de style, impossible. Rien à faire à part des grosses

bordures saillantes et peu élégantes. Mais avec une petite dose de CSS, c'est fait en un rien de temps.

Code CSS :

```
table
{
  border-width:1px;
  border-style:solid;
  border-color:black;
  width:50%;
}
td
{
  border-width:1px;
  border-style:solid;
  border-color:red;
  width:50%;
}
```

Code HTML

```
<table>
  <tr>
    <td><p>Oh qu'il est beau !</p></td>
    <td><p>Ce jouli tableau à traits fins !!</p></td>
```

```
  </tr>
</table>
```

Le resultat est

Oh qu'il est beau !	Ce jouli tableau à bordures fines !!

exemple 2 de table :

Code CSS

```
.grise
{
  border-width:1px;
  border-style:solid;
  border-color:black;
  background-color:silver;
}
.blue
{
  border-width:1px;
  border-style:solid;
  border-color:blue;
}
.none
{
  border-style:none;
}
```

Code HTML

```
<table class="none">
  <tr>
    <td class="grise">Oh la belle cellule grise !</td>
    <td class="blue">Oh la belle bleue !!</td>
  </tr>
</table>
```

Le resultat est :

4- Les liens bouton :

Faire un bouton qui se change lors du passage de la souris.

Code HTML

```
<p class="bouton">
  <a href="cours3.php" class="bouton">Cliquez ici !</a>
</p>
```

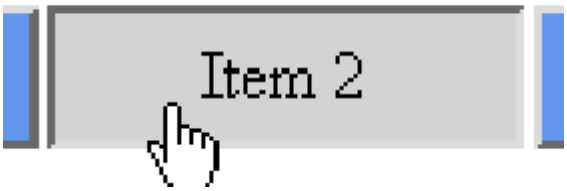
Code CSS

```
a.bouton:link
{
  width:150px;
  height:30px;
  text-decoration:none;
  color:white;
  text-align:center;
  font-weight:bold;
```

```
  background-color:#000080;
  padding:5px
}
a.bouton:visited
{
  width:150px;
  height:30px;
text-decoration:none;
  color:white;
  text-align:center;
  font-weight:bold;
  background-color:#000080;
  padding:5px
}
a.bouton:hover
{
  width:150px;
  height:30px;
  text-decoration:none;
  color:white;
  text-align:center;
  font-weight:bold;
  background-color:#0000FF;
  background-image:url(aqua.jpg);
  padding:5px
}
```

```
.bouton
{
  text-align:center;
  padding:5px;
}
```

Le resultat est :



5- Les listes avec images :

Code CSS

```
li
{
 font-family: Verdana, Arial, Helvetica, Geneva, sans-serif;
 font-size: 100%;
 color: black;
 display : list-item;
 list-style-image : url(puce.gif);
 list-style-position: outside;
}
```

Code HTML

```
<ul>
  <li>liste 1</li>
  <li>liste 2</li>
  <li>liste 3</li>
```


Le resultat :

7 – les formulaires :

Code CSS :

```
p
{
  font-family:"trebuchet ms",sans-serif;
  font-size:80%;
}
form
{
  background-color:#F5F5F5;
  padding:10px;
  width:350px;
}
label
{
  font-family:"trebuchet ms",sans-serif;
  font-weight:bold
}
input
{
  border:1px solid black;
```

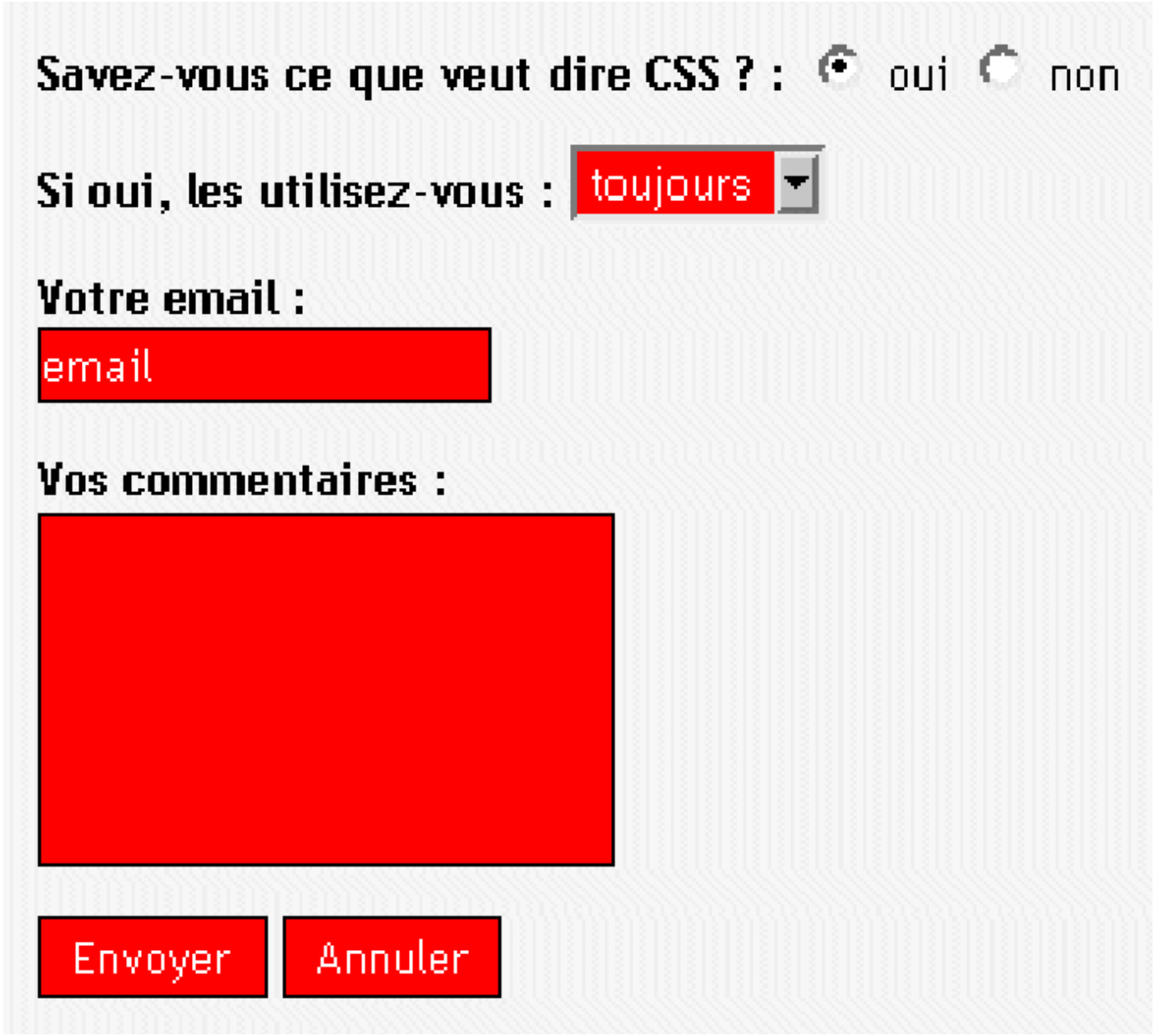
```
  background-color:red;
  font-family:"trebuchet ms",sans-serif;
  color:white;
}
select, option
{
  background-color:red;
color:white;
}
textarea
{
  border:1px solid black;
  background-color:red;
  font-family:"trebuchet ms",sans-serif;
  color:white;
}
```

Le Code HTML

```
<form action="toto.php" method="post" >
  <p>
<label>Savez-vous ce que veut dire CSS ? : </label>
    <input type="radio" name="CSS" value="oui" checked="checked" /> oui
    <input type="radio" name="CSS" value="non" /> non
  </p>
  <p>
    <label>Si oui, les utilisez-vous plutôt : </label>
    <select name="utilise">
```

```
      <option value="toujours"> toujours</option>
      <option value="parfois"> parfois</option>
      <option value="jamais"> jamais</option>
    </select>
  </p>
  <p>
    <label for="email">Votre email :</label><br />
    <input type="text" name="email" size="20" maxlength="40" value="email" id="email"
/>
  </p>
  <p>
    <label for="comments">Vos commentaires :</label><br />
    <textarea name="comments" id="comments" cols="20" rows="4"></textarea>
  </p>
  <p>
    <input type="submit" value="Envoyer" />
    <input type="reset" value="Annuler" />
  </p>
</form>
```

Le resultat est :


Savez-vous ce que veut dire CSS ? : oui non
Si oui, les utilisez-vous : toujours
Votre email :
email
Vos commentaires :
Envoyer
Annuler

Chapitre 4 : The Document Object Model

A *document object model* (DOM) is an application programming interface (API) for representing a document (such as an HTML document) and accessing and manipulating the various elements (such as HTML tags and strings of text) that make up that document. JavaScript-enabled web browsers have always defined a document object model; a web-browser DOM may specify, for example, that the forms in an HTML document are accessible through the `forms[]` array of the Document object.

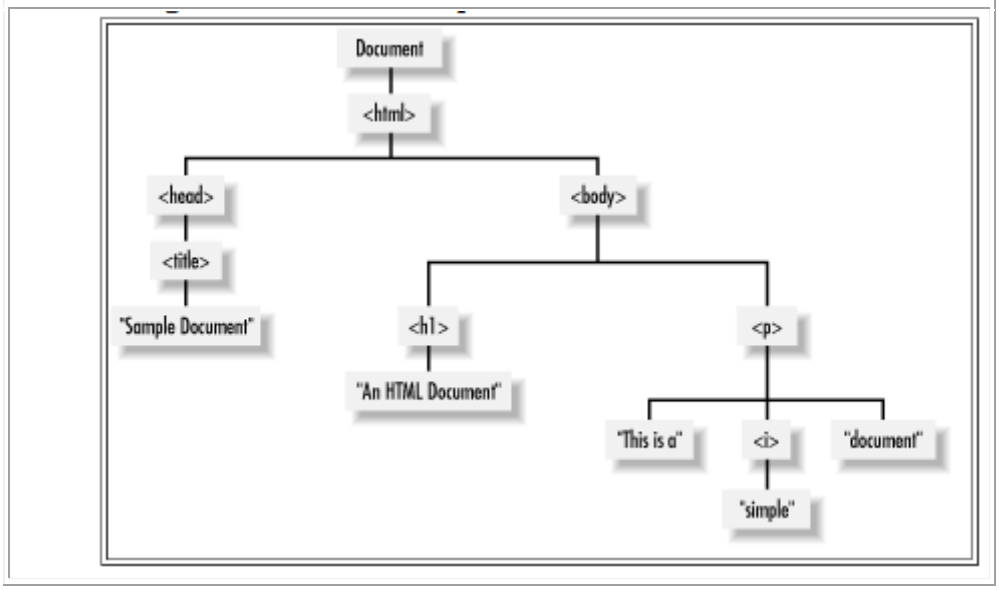
Representing Documents as Trees

HTML documents have a hierarchical structure that is represented in the DOM as a tree structure. The nodes of the tree represent the various types of content in a document. The tree representation of an HTML document primarily contains nodes representing elements or tags such as `<body>` and `<p>` and nodes representing strings of text. An HTML document may also contain nodes representing HTML comments.[2] Consider the following simple HTML document:

```
<html>
  <head>
    <title>Sample Document</title>
  </head>
  <body>
    <h1>An HTML Document</h2>
    <p>This is a <i>simple</i> document.
  </body>
</html>
```

The DOM representation of this document is the tree pictured in Figure 17-1.

Figure 17-1. The tree representation of an HTML document



If you are not already familiar with tree structures in computer programming, it is helpful to know that they borrow terminology from family trees. The node directly above a node is the *parent* of that node. The nodes one level directly below another node are the *children* of

that node. Nodes at the same level, and with the same parent, are *siblings*. The set of nodes any number of levels below another node are the *descendants* of that node. And the parent, grandparent, and all other nodes above a node are the *ancestors* of that node.

Nodes

The DOM tree structure illustrated in Figure 17-1 is represented as a tree of various types of Node objects. The Node interface[3] defines properties and methods for traversing and manipulating the tree. The `childNodes` property of a Node object returns a list of children of the node, and the `firstChild`, `lastChild`, `nextSibling`, `previousSibling`, and `parentNode` properties provide a way to traverse the tree of nodes. Methods such as `appendChild( )`, `removeChild( )`, `replaceChild( )`, and `insertBefore( )` enable you to add and remove nodes from the document tree. We'll see examples of the use of these properties and methods later in this chapter.

Types of nodes

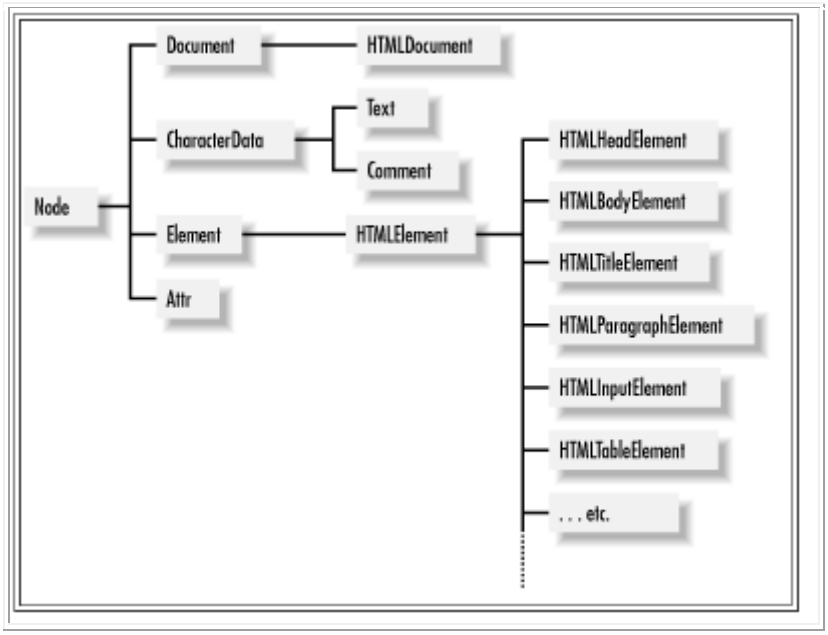
Different types of nodes in the document tree are represented by specific subinterfaces of Node. Every Node object has a `nodeType` property that specifies what kind of node it is. If the`nodeType` property of a node equals the constant `Node.ELEMENT_NODE`, for example, you know the Node object is also an Element object and you can use all the methods and properties defined by the Element interface with it. Table 17-1 lists the node types commonly encountered in HTML documents and the `nodeType` value for each one.

Table 17-1: Common node types

Interface	nodeType constant	nodeType value
Element	Node.ELEMENT_NODE	1
Text	Node.TEXT_NODE	3
Document	Node.DOCUMENT_NODE	9
Comment	Node.COMMENT_NODE	8
DocumentFragment	Node.DOCUMENT_FRAGMENT_NODE	11
Attr	Node.ATTRIBUTE_NODE	2

The Node at the root of the DOM tree is a Document object. The `documentElement` property of this object refers to an Element object that represents the root element of the document. For HTML documents, this is the `<html>` tag that is either explicit or implicit in the document. (The Document node may have other children, such as Comment nodes, in addition to the root element.) The bulk of a DOM tree consists of Element objects, which represent tags such as `<html>` and `<i>`, and Text objects, which represent strings of text. If the document parser preserves comments, those comments are represented in the DOM tree by Comment objects. Figure 17-2 shows a partial class hierarchy for these and other core DOM interfaces.

Figure 17-2. A partial class hierarchy of the core DOM API



Attributes

The attributes of an element (such as the `src` and `width` attributes of an `<img>` tag) may be queried, set, and deleted using the `getAttribute( )`, `setAttribute( )`, and`removeAttribute( )` methods of the Element interface.

Les propriétés de l'interface Node

`Node.childNodes`	
`Node.firstChild`	
`Node.lastChild`	
`Node.nextSibling`	
`Node.nodeType`	
`Node.nodeValue`	
`Node.parentNode`	
`Node.rootNode`	

Edit

Inherits properties from its parents `EventTarget`.[1]

`Node.baseURI` Read only

> Returns a `DOMString` representing the base URL. The concept of base URL changes from one language to another; in HTML, it corresponds to the protocol, the domain name and the directory structure, that is all until the last `'/'`.

`Node.baseURIObject`

(Not available to web content.) The read-only `nsIURI` object representing the base URI for the element.

`Node.childNodes` Read only

Returns a live `NodeList` containing all the children of this node. `NodeList` being live means that if the children of the `Node` change, the `NodeList` object is automatically updated.

`Node.firstChild` Read only

Returns a `Node` representing the first direct child node of the node, or `null` if the node has no child.

`Node.lastChild` Read only

Returns a `Node` representing the last direct child node of the node, or `null` if the node has no child.

`Node.nextSibling` Read only

```
for(var m = n.firstChild; m != null; m = m.nextSibling) {
```

Returns a `Node` representing the next node in the tree, or `null` if there isn't such node.

`Node.nodeName` Read only

Returns a `DOMString` containing the name of the `Node`. The structure of the name will differ with the node type. E.g. An `HTMLElement` will contain the name of the corresponding tag, like `'audio'` for an `HTMLAudioElement`, a `Text` node will have the `'#text'` string, or a `Document` node will have the `'#document'` string.

`Node.nodePrincipal`

A `nsIPrincipal` representing the node principal.

`Node.nodeType` Read only

Returns an `unsigned short` representing the type of the node. Possible values are:

Name	Value
ELEMENT_NODE	1
ATTRIBUTE_NODE	2

```
                    TEXT_NODE       3
                CDATA_SECTION_NODE       4
            ENTITY_REFERENCE_NODE       5
                    ENTITY_NODE       6
        PROCESSING_INSTRUCTION_NODE       7
                    COMMENT_NODE       8
                    DOCUMENT_NODE       9
                DOCUMENT_TYPE_NODE      10
            DOCUMENT_FRAGMENT_NODE      11
                    NOTATION_NODE      12
```

`Node.nodeValue`

Returns / Sets the value of the current node

`Node.ownerDocument` Read only

Returns the `Document` that this node belongs to. If the node is itself a document, returns `null`.

`Node.parentNode` Read only

Returns a `Node` that is the parent of this node. If there is no such node, like if this node is the top of the tree or if doesn't participate in a tree, this property returns `null`.

`Node.parentElement` Read only

Returns an `Element` that is the parent of this node. If the node has no parent, or if that parent is not an `Element`, this property returns `null`.

`Node.previousSibling` Read only

Returns a `Node` representing the previous node in the tree, or `null` if there isn't such node.

`Node.textContent`

Returns / Sets the textual content of an element and all its descendants.

Deprecated properties

`Node.rootNode` Read only

Returns a `Node` object representing the topmost node in the tree, or the current node if it's the topmost node in the tree. This has been replaced by `Node.getRootNode()`.

Obsolete properties

`Node.localName` Read only

Returns a `DOMString` representing the local part of the qualified name of an element.

Note: In Firefox 3.5 and earlier, the property upper-cases the local name for HTML elements (but not XHTML elements). In later versions, this does not happen, so the property is in lower case for both HTML and XHTML.

`Node.namespaceURI` Read only

The namespace URI of this node, or `null` if it is no namespace.

Note: In Firefox 3.5 and earlier, HTML elements are in no namespace. In later versions, HTML elements are in the http://www.w3.org/1999/xhtml/namespace in both HTML and XML trees.

`Node.prefix` Read only

Is a `DOMString` representing the namespace prefix of the node, or `null` if no prefix is specified.

Les propriétés de l'interface Node

`Node.appendChild()`	`parent.appendChild(child);`
`Node.cloneNode()`	`var cln = itm.cloneNode(true);`
`Node.getRootNode()`	`rootNode = node.getRootNode();`
`Node.hasChildNodes()`	`var boolean = document.getElementById("myList").hasChildNodes();`
`Node.insertBefore()`	`parent.insertBefore(newItem, brother);`
`Node.removeChild()`	`var oldChild = node.removeChild(child);`
`Node.replaceChild()`	`parent.replaceChild(newNode, n);`
`Node.hasAttributes()`	`var x = document.getElementById("myBtn").hasAttribute("onclick");`
`Node.setAttribute(),`	`document.getElementsByTagName("H1")[0].setAttribute("class", "democlass");`
`Node.removeAttribute()`	`document.getElementsByTagName("H1")[0].removeAttribute("class");`

MethodsEdit

Inherits methods from its parent, `EventTarget`.[1]

Node.appendChild()

Adds the specified childNode argument as the last child to the current node. If the argument referenced an existing node on the DOM tree, the node will be detached from its current position and attached at the new position.

```
        n.appendChild(c);
```

```
var node = document.createElement("LI");                // Create a <li> node
var textnode = document.createTextNode("Water");        // Create a text node
node.appendChild(textnode);                             // Append the text to
<li>
document.getElementById("myList").appendChild(node);    // Append <li> to <ul>
with id="myList"
```

Node.cloneNode()

Clone a `Node`, and optionally, all of its contents. By default, it clones the content of the node.

```
// Get the last <li> element ("Milk") of <ul> with id="myList2"
var itm = document.getElementById("myList2").lastChild;

// Copy the <li> element and its child nodes
var cln = itm.cloneNode(true);

// Append the cloned <li> element to <ul> with id="myList1"
document.getElementById("myList1").appendChild(cln);
```

Node.compareDocumentPosition()

Compares the position of the current node against another node in any other document.

Node.contains()

Returns a `Boolean` value indicating whether a node is a descendant of a given node or not.

Node.getRootNode()

Returns the context object's root which optionally includes the shadow root if it is available.

Node.hasChildNodes()

Returns a `Boolean` indicating if the element has any child nodes, or not.

Node.insertBefore()

Inserts a `Node` before the reference node as a child of the current node.

```
var newItem = document.createElement("LI");       // Create a <li> node
var textnode = document.createTextNode("Water");  // Create a text node
newItem.appendChild(textnode);                    // Append the text to <li>
```

```
var list = document.getElementById("myList");    // Get the <ul> element to insert a new node
list.insertBefore(newItem, list.childNodes[0]);  // Insert <li> before the first child of <ul>
```

`Node.isDefaultNamespace()`

Accepts a namespace URI as an argument and returns a `Boolean` with a value of `true` if the namespace is the default namespace on the given node or `false` if not.

`Node.isEqualNode()`

Returns a `Boolean` which indicates whether or not two nodes are of the same type and all their defining data points match.

`Node.isSameNode()`

Returns a `Boolean` value indicating whether or not the two nodes are the same (that is, they reference the same object).

`Node.lookupPrefix()`

Returns a `DOMString` containing the prefix for a given namespace URI, if present, and `null` if not. When multiple prefixes are possible, the result is implementation-dependent.

`Node.lookupNamespaceURI()`

Accepts a prefix and returns the namespace URI associated with it on the given node if found (and `null` if not). Supplying `null` for the prefix will return the default namespace.

`Node.normalize()`

Clean up all the text nodes under this element (merge adjacent, remove empty).

`Node.removeChild()`

Removes a child node from the current element, which must be a child of the current node.

```
var c = n.removeChild(kids[i]);    // Remove a child
```

```
var list = document.getElementById("myList");   // Get the <ul> element with id="myList"
list.removeChild(list.childNodes[0]);           // Remove <ul>'s first child node (index 0)
```

`Node.replaceChild()`

Replaces one child `Node` of the current one with the second one given in parameter.

```
parent.replaceChild(newNode, n);
```

```
// Create a new text node called "Water"
var textnode = document.createTextNode("Water");

// Get the first child node of an <ul> element
var item = document.getElementById("myList").childNodes[0];

// Replace the first child node of <ul> with the newly created text node
item.replaceChild(textnode, item.childNodes[0]);

// Note: This example replaces only the Text node "Coffee" with a Text node "Water"
```

Obsolete methods

`Node.getFeature()`

X

`Node.getUserData()`

Allows a user to get some `DOMUserData` from the node.

`Node.hasAttributes()`

Returns a `Boolean` indicating if the element has any attributes, or not.

`Node.isSupported()`

Returns a `Boolean` flag containing the result of a test whether the DOM implementation implements a specific feature and this feature is supported by the specific node.

`Node.setUserData()`

Allows a user to attach, or remove, `DOMUserData` to the node.

Example

Language-Independent DOM Interfaces

Although the DOM standard grew out of a desire to have a common API for dynamic HTML programming, the DOM is not of interest only to web scripters. In fact, the standard is currently most heavily used by server-side Java and C++ programs that parse and manipulate XML documents. Because of its many uses, the DOM standard is defined to be language-independent. This book describes only the JavaScript binding of the DOM API, but you should be aware of a few other points. First, note that object properties in the JavaScript binding are typically mapped to pairs of get/set methods in other language bindings. Thus, when a Java programmer asks you about the `getFirstChild( )` method of the Node interface, you need to understand that the JavaScript binding of the Node API doesn't define a `getFirstChild( )` method. Instead, it simply defines a `firstChild` property, and reading the value of this property in JavaScript is equal to calling `getFirstChild( )` in Java.

Another important feature of the JavaScript binding of the DOM API is that certain DOM objects behave like JavaScript arrays. If an interface defines a method named `item( )`, objects that implement that interface behave like read-only numerical arrays. For example, suppose you've obtained a NodeList object by reading the `childNodes` property of a node. You can obtain the individual Node objects in the list by passing the desired node number to the `item( )` method, or, more simply, you can simply treat the NodeList object as an array and index it directly. The following code illustrates these two options:

```
var n = document.documentElement;  // This is a Node object.
var children = n.childNodes;       // This is a NodeList object.
var head = children.item(0);       // Here is one way to use a NodeList.
var body = children[1];
```

Traversing a Document

As we've already discussed, the DOM represents an HTML document as a tree of Node objects. With any tree structure, one of the most common things to do is traverse the tree, examining each node of the tree in turn. Example 17-1 shows one way to do this. It is a JavaScript function that recursively examines a node and all its children, adding up the number of HTML tags (i.e., Element nodes) it encounters in the course of the traversal. Note the use of the `childNodes` property of a node. The value of this property is a NodeList object, which behaves (in JavaScript) like an array of Node objects. Thus, the function can enumerate all the children of a given node by looping through the elements of the `childNodes[]` array. By recursing, the function enumerates not just all children of a given node, but all nodes in the tree of nodes. Note that this function also demonstrates the use of the `nodeType` property to determine the type of each node.

Example 17-1: Traversing the nodes of a document

```
<head>
<script>
// This function is passed a DOM Node object and checks to see if that node
// represents an HTML tag; i.e., if the node is an Element object. It
// recursively calls itself on each of the children of the node, testing
// them in the same way. It returns the total number of Element objects
// it encounters. If you invoke this function by passing it the
// Document object, it traverses the entire DOM tree.
function countTags(n) {                          // n is a Node
    var numtags = 0;                             // Initialize the tag counter
    if (n.nodeType == 1 /*Node.ELEMENT_NODE*/)   // Check if n is an Element
        numtags++;                               // Increment the counter if so
    var children = n.childNodes;                 // Now get all children of n
    for(var i=0; i < children.length; i++) {     // Loop through the children
        numtags += countTags(children[i]);       // Recurse on each one
    }
    return numtags;                              // Return the total number of tags
}
</script>
</head>
<!-- Here's an example of how the countTags( ) function might be used -->
<body onload="alert('This document has ' + countTags(document) + ' tags')">
This is a <i>sample</i> document.
</body>
```

Another point to notice about Example 17-1 is that the `countTags( )` function it defines is invoked from the `onload` event handler, so that it is not called until the document is completely loaded. This is a general requirement when working with the DOM: you cannot traverse or manipulate the document tree until the document has been fully loaded.

In addition to the `childNodes` property, the Node interface defines a few other useful properties. `firstChild` and `lastChild` refer to the first and last children of a node, and `nextSibling`and `previousSibling` refer to adjacent siblings of a node. (Two nodes are siblings if they have the same parent node.) These properties provide another way to loop through the children of a node, demonstrated in Example 17-2. This example counts the number of characters in all the Text nodes within the `<body>` of the document. Notice the way the `countCharacters( )` function uses the `firstChild` and `nextSibling` properties to loop through the children of a node.

Example 17-2: Another way to traverse a document

```
<head>
<script>
// This function is passed a DOM Node object and checks to see if that node
// represents a string of text; i.e., if the node is a Text object. If
// so, it returns the length of the string. If not, it recursively calls
// itself on each of the children of the node and adds up the total length
// of the text it finds. Note that it enumerates the children of a node
// using the firstChild and nextSibling properties. Note also that the
// function does not recurse when it finds a Text node, because Text nodes
// never have children.
function countCharacters(n) {                // n is a Node
    if (n.nodeType == 3 /*Node.TEXT_NODE*/)  // Check if n is a Text object
        return n.length;                     // If so, return its length
    // Otherwise, n may have children whose characters we need to count
    var numchars = 0;  // Used to hold total characters of the children
    // Instead of using the childNodes property, this loop examines the
    // children of n using the firstChild and nextSibling properties.
    for(var m = n.firstChild; m != null; m = m.nextSibling) {
        numchars += countCharacters(m);  // Add up total characters found
    }
    return numchars;                     // Return total characters
}
</script>
</head>
<!--
  The onload event handler is an example of how the countCharacters(  )
  function might be used. Note that we want to count only the characters
  in the <body> of the document, so we pass document.body to the function.
-->
<body onload="alert('Document length: ' + countCharacters(document.body))">
This is a sample document.<p>How long is it?
</body>
```

Finding Specific Elements in a Document

The ability to traverse all nodes in a document tree gives us the power to find specific nodes. When programming with the DOM API, it is quite common to need a particular node within the document or a list of nodes of a specific type within the document. Fortunately, the DOM API provides functions that make this easy for us.

In Example 17-2, we referred to the <body> element of an HTML document with the JavaScript expression document.body. The body property of the Document object is a convenient special-case property and is the preferred way to refer to the <body> tag of an HTML document. If this convenience property did not exist, however, we could also refer to the <body> tag like this:

```
document.getElementsByTagName("body")[0]
```

This expression calls the Document object's getElementsByTagName( ) method and selects the first element of the returned array. The call to getElementsByTagName( ) returns an array of all <body> elements within the document. Since HTML documents can have only one <body>, we know that we're interested in the first element of the returned array.[6]

You can use getElementsByTagName( ) to obtain a list of any type of HTML element. For example, to find all the tables within a document, you'd do this:

```
var tables = document.getElementsByTagName("table");
alert("This document contains " + tables.length + " tables");
```

Note that since HTML tags are not case-sensitive, the strings passed to getElementsByTagName( ) are also not case-sensitive. That is, the previous code finds <table> tags even if they are coded as <TABLE>. getElementsByTagName( ) returns elements in the order in which they appear in the document. Finally, if you pass the special string "*" to getElementsByTagName( ), it returns a list of all the elements in the document, in the order in which they appear. (This special usage is not supported in IE 5 and IE 5.5. See instead the IE-specific Document.all[] array in the client-side reference section.)

Sometimes you don't want a list of elements but instead want to operate on a single specific element of a document. If you know a lot about the structure of the document, you may be able to usegetElementsByTagName( ). For example, if you want to do something to the fourth paragraph in a document, you might use this code:

```
var myParagraph = document.getElementsByTagName("p")[3];
```

This typically is not the best (nor the most efficient) technique, however, because it depends so heavily on the structure of the document; a new paragraph inserted at the beginning of the document would break the code. Instead, when you need to manipulate specific elements of a document, it is best to give those elements an id attribute that specifies a unique (within the document) name for the element. Then you can look up your desired element by its ID. For example, you might code the special fourth paragraph of your document with a tag like this:

```
<p id="specialParagraph">
```

You can then look up the node for that paragraph with JavaScript code like this:

```
var myParagraph = document.getElementById("specialParagraph");
```

Note that the getElementById( ) method does not return an array of elements like getElementsByTagName( ) does. Because the value of every id attribute is (or is supposed to be) unique, getElementById( ) returns only the single element with the

matching `id` attribute. `getElementById( )` is an important method, and its use is quite common in DOM programming.

`getElementById( )` and `getElementsByTagName( )` are both methods of the Document object. Element objects also define a `getElementsByTagName( )` method, however. This method of the Element object behaves just like the method of the Document object, except that it returns only elements that are descendants of the element on which it is invoked. Instead of searching the entire document for elements of a specific type, it searches only within the given element. This makes it possible, for example, to use `getElementById( )` to find a specific element and then to use `getElementsByTagName( )` to find all descendants of a given type within that specific tag. For example:

```
// Find a specific Table element within a document and count its rows
var tableOfContents = document.getElementById("TOC");
var rows = tableOfContents.getElementsByTagName("tr");
var numrows = rows.length;
```

Finally, note that for HTML documents, the HTMLDocument object also defines a `getElementsByName( )` method. This method is like `getElementById( )`, but it looks at the `name`attribute of elements rather than the `id` attribute. Also, because the `name` attribute is not expected to be unique within a document (for example, radio buttons within HTML forms usually have the same `name`), `getElementsByName( )` returns an array of elements rather than a single element. For example:

```
// Find <a name="top">
var link = document.getElementsByName("top")[0];
// Find all <input type="radio" name="shippingMethod"> elements
var choices = document.getElementsByName("shippingMethod");
```

Modifying a Document

Traversing the nodes of a document can be useful, but the real power of the core DOM API lies in the features that allow you to use JavaScript to dynamically modify documents. The following examples demonstrate the basic techniques of modifying documents and illustrate some of the possibilities.

Example 17-3 includes a JavaScript function named `reverse( )`, a sample document, and an HTML button that, when pressed, calls the `reverse( )` function, passing it the node that represents the `<body>` element of the document. (Note the use of `getElementsByTagName( )` within the button's event handler to find the `<body>` element.) The `reverse( )` function loops backward through the children of the supplied node and uses the `removeChild( )` and `appendChild( )` methods of the Node object to reverse the order of those children.

Example 17-3: Reversing the nodes of a document

```
<head><title>Reverse</title>
<script>
function reverse(n) {           // Reverse the order of the children of Node n
    var kids = n.childNodes;    // Get the list of children
    var numkids = kids.length;  // Figure out how many children there are
    for(var i = numkids-1; i >= 0; i--) {  // Loop backward through the children
        var c = n.removeChild(kids[i]);    // Remove a child
```

```
        n.appendChild(c);                   // Put it back at its new position
    }
}
</script>
</head>
<body>
<p>paragraph #1<p>paragraph #2<p>paragraph #3  <!-- A sample document -->
<p>                                    <!-- A button to call reverse(  )-->
<button onclick="reverse(document.body);"
>Click Me to Reverse</button>
</body>
```

shows one way to change the text displayed in a document: simply set the `data` field of the appropriate Text node. Example 17-5 shows another way. This example defines a function, `uppercase( )`, that recursively traverses the children of a given node. When it finds a Text node, the function replaces that node with a new Text node containing the text of the original node, converted to uppercase. Note the use of the `document.createTextNode( )` method to create the new Text node and the use of Node's `replaceChild( )` method to replace the original Text node with the newly created one. Note also that `replaceChild( )` is invoked on the parent of the node to be replaced, not on the node itself. The `uppercase( )` function uses Node's `parentNode` property to determine the parent of the Text node it replaces.

In addition to defining the `uppercase( )` function, Example 17-5 includes two HTML paragraphs and a button. When the user clicks the button, one of the paragraphs is converted to uppercase. Each paragraph is identified with a unique name, specified with the `id` attribute of the `<p>` tag. The event handler on the button uses the `getElementById( )` method to get the Element object that represents the desired paragraph.

Example 17-5: Replacing nodes with their uppercase equivalents

```
<script>
// This function recursively looks at Node n and its descendants,
// replacing all Text nodes with their uppercase equivalents.
function uppercase(n) {
    if (n.nodeType == 3 /*Node.TEXT_NODE*/) {
        // If the node is a Text node, create a new Text node that
        // holds the uppercase version of the node's text, and use the
        // replaceChild(  ) method of the parent node to replace the
        // original node with the new uppercase node.
        var newNode = document.createTextNode(n.data.toUpperCase(  ));
        var parent = n.parentNode;
        parent.replaceChild(newNode, n);
    }
    else {
        // If the node is not a Text node, loop through its children
        // and recursively call this function on each child.
        var kids = n.childNodes;
        for(var i = 0; i < kids.length; i++) uppercase(kids[i]);
    }
}
</script>

<!-- Here is some sample text. Note that the <p> tags have id attributes. -->
<p id="p1">This <i>is</i> paragraph 1.</p>
<p id="p2">This <i>is</i> paragraph 2.</p>

<!-- Here is a button that invokes the uppercase(  ) function defined above. -->
<!-- Note the call to document.getElementById(  ) to find the desired node. -->
```

```
<button onclick="uppercase(document.getElementById('p1'));">Click Me</button>
```

The previous two examples show how to modify document content by replacing the text contained within a Text node and by replacing one Text node with an entirely new Text node. It is also possible to append, insert, delete, or replace text within a Text node with the `appendData( )`, `insertData( )`, `deleteData( )`, and `replaceData( )` methods. These methods are not directly defined by the Text interface, but instead are inherited by Text from CharacterData. You can find more information about them under "CharacterData" in the DOM reference section.

In the node-reversal examples, we saw how we could use the `removeChild( )` and `appendChild( )` methods to reorder the children of a Node. Note, however, that we are not restricted to changing the order of nodes within their parent node; the DOM API allows nodes in the document tree to be moved freely within the tree (only within the same document, however). Example 17-6 demonstrates this by defining a function named `embolden( )` that replaces a specified node with a new element (created with the `createElement( )` method of the Document object) that represents an HTML `<b>` tag and "reparents" the original node as a child of the new `<b>` node. In an HTML document, this causes any text within the node or its descendants to be displayed in boldface.

Example 17-6: Reparenting a node to a <b> element

```
<script>
// This function takes a Node n, replaces it in the tree with an Element node
// that represents an HTML <b> tag, and then makes the original node the
// child of the new <b> element.
function embolden(node) {
    var bold = document.createElement("b");  // Create a new <b> element
    var parent = node.parentNode;            // Get the parent of the node
    parent.replaceChild(bold, node);         // Replace the node with the <b> tag
    bold.appendChild(node);                  // Make the node a child of the <b> tag
}
</script>

<!-- A couple of sample paragraphs -->
<p id="p1">This <i>is</i> paragraph #1.</p>
<p id="p2">This <i>is</i> paragraph #2.</p>

<!-- A button that invokes the embolden( ) function on the first paragraph -->
<button onclick="embolden(document.getElementById('p1'));">Embolden</button>
```

In addition to modifying documents by inserting, deleting, reparenting, and otherwise rearranging nodes, it is also possible to make substantial changes to a document simply by setting attribute values on document elements. One way to do this is with the `element.setAttribute( )` method. For example:

```
var headline = document.getElementById("headline");  // Find named element
headline.setAttribute("align", "center");            // Set align='center'
```

The DOM elements that represent HTML attributes define JavaScript properties that correspond to each of their standard attributes (even deprecated attributes such as `align`), so you can also achieve the same effect with this code:

```
var headline = document.getElementById("headline");
headline.align = "center";  // Set alignment attribute.
```

Adding Content to a Document

The previous two examples showed how the contents of a Text node can be changed to uppercase and how a node can be reparented to be a child of a newly created `<b>` node. The`embolden( )` function showed that it is possible to create new nodes and add them to a document. You can add arbitrary content to a document by creating the necessary Element nodes and Text nodes with `document.createElement( )` and `document.createTextNode( )` and by adding them appropriately to the document. This is demonstrated in Example 17-7, which defines a function named `debug( )`. This function provides a convenient way to insert debugging messages into a program, and it serves as a useful alternative to using the built-in `alert( )` function. A sample use of this `debug( )` function is illustrated in Figure 17-4.

Figure 17-4. Output of the debug( ) function

The first time `debug( )` is called, it uses the DOM API to create a `<div>` element and insert it at the end of the document. The debugging messages passed to `debug( )` on this first call and all subsequent calls are then inserted into this `<div>` element. Each debugging message is displayed by creating a Text node within a `<p>` element and inserting that `<p>` element at the end of the`<div>` element.

Example 17-7 also demonstrates a convenient but nonstandard way to add new content to a document. The `<div>` element that contains the debugging messages displays a large, centered title. This title could be created and added to the document in the way that other content is, but in this example we instead use the `innerHTML` property of the `<div>` element. Setting this property of any element to a string of HTML text causes that HTML to be parsed and inserted as the content of the element. Although this property is not part of the DOM API, it is a useful shortcut that is supported by Internet Explorer 4 and later and Netscape 6. Although it is not standard, it is in common use and is included in this example for completeness.[7]

Example 17-7: Adding debugging output to a document

```
/**
 * This debug function displays plain-text debugging messages in a
 * special box at the end of a document. It is a useful alternative
 * to using alert(  ) to display debugging messages.
 **/
function debug(msg) {
    // If we haven't already created a box within which to display
    // our debugging messages, then do so now. Note that to avoid
    // using another global variable, we store the box node as
    // a proprty of this function.
    if (!debug.box) {
        // Create a new <div> element
        debug.box = document.createElement("div");
        // Specify what it looks like using CSS style attributes
        debug.box.setAttribute("style",
                               "background-color: white; " +
```

```
                                "font-family: monospace; " +
                                "border: solid black 3px; " +
                                "padding: 10px;");

        // Append our new <div> element to the end of the document
        document.body.appendChild(debug.box);

        // Now add a title to our <div>. Note that the innerHTML property is
        // used to parse a fragment of HTML and insert it into the document.
        // innerHTML is not part of the W3C DOM standard, but it is supported
        // by Netscape 6 and Internet Explorer 4 and later. We can avoid
        // the use of innerHTML by explicitly creating the <h1> element,
        // setting its style attribute, adding a Text node to it, and
        // inserting it into the document, but this is a nice shortcut.
        debug.box.innerHTML = "<h1 style='text-align:center'>Debugging Output</h2>";
    }

    // When we get here, debug.box refers to a <div> element into which
    // we can insert our debugging message.
    // First create a <p> node to hold the message.
    var p = document.createElement("p");
    // Now create a text node containing the message, and add it to the <p>
    p.appendChild(document.createTextNode(msg));
    // And append the <p> node to the <div> that holds the debugging output
    debug.box.appendChild(p);
}
```

The `debug( )` method listed in Example 17-7 can be used in HTML documents like the following, which is the document that was used to generate Figure 17-4:

```
<script src="Debug.js"></script>  <!-- Include the debug(  ) function -->
<script>var numtimes=0;</script>  <!-- Define a global variable -->
<!-- Now use the debug(  ) function in an event handler -->
<button onclick="debug('clicked: ' + numtimes++);">press me</button>
```

Why use DOM instead of DHTML?

In this article, I look at the structure of a Web page from a Document Object Model point of view, examining children and parents and adding nodes to and editing nodes within an existing document. To illustrate this, I build a page where the user can create a series of notes to the page and edit their content. (The page won't get too fancy; I'll leave saving the content, moving them around, resizing them and such to you.) As this is something that could be accomplished without much trouble using DHTML techniques, why bother with DOM?

One of the driving forces behind the development of XML was the fragmentation of HTML, and DHTML follows that trend. Some things have become standard, but a significant divergence in how browsers handle various aspects still exists. DOM is a standard set of interfaces that can be implemented across the board, letting programmers write just one version of their pages.

Besides, Web pages are moving towards XML anyway. XHTML 1.0 is a reformulation of HTML 4.0.1 into XML, and XML is increasingly used to generate Web pages. Ultimately, DOM will be the main technique for accessing the elements and text within these pages. (You can also use it in XSLT style sheets.)

Back to top

The basic page

Let's start with the basic page, which includes a work area and form indicating how many notes are already created:

Listing 1. The basic page

```
<html>
  <head>
    <title>Getting Sticky</title>
    <style type="text/css">
      * {font-family: sans-serif}
      a {font-size: 6pt}
      .editButton {font-size:6pt}
    </style>

  </head>
  <body>
    <div id="mainDiv" style="height:95%; width:95%; border:3px solid red;
                                                          padding: 10px;">

       <h1>Getting Sticky</h1>

       <form id="noteForm">
           Current number of notes:
                       <input type="text" name="total" value="0" size="3"/>
           <input type="button" value="Add a new note"/>
       </form>

    </div>
  </body>
</html>
```

Lisitng 1 shows a basic page with a couple of styles (just to pretty it up a little). The body of the page includes a single `div`element, which includes a heading element (`h1`) and a `form` element. Even in pre-DOM browsers, accessing the information within the form wasn't a problem.

Figure 1. The basic page

Back to top

The DHTML way

Using DHTML, you can access the information in a form field, or even change it. For example, you can create a script that increments the current form value by 1, then tell the page to execute the script when the user presses the button:

Listing 2. Incrementing the current number

```
<html><head>
    <title>Getting Sticky</title>
    <style type="text/css">
      * {font-family: sans-serif}
      a {font-size: 6pt}
      .editButton {font-size:6pt}
    </style>

    <script type="text/javascript">
      function incrementCurrent() {
        current = parseInt(document.forms["noteForm"].total.value);
        document.forms["noteForm"].total.value = current + 1;
      }
    </script>

  </head><body>
    <div id="mainDiv" style="height:95%; width:95%; border:3px solid red;
                                                          padding: 10px;">

       <h1>Getting Sticky</h1>

       <form id="noteForm">
           Current number of notes:
                      <input type="text" name="total" value="0" size="3"/>
           <input type="button" value="Add a new note"
                                          onclick="incrementCurrent()"/>
       </form>

    </div>
</body></html>
```

In Listing 2, the `incrementCurrent()` function takes the document object, and then pulls the `noteForm` object out of the array of forms within the page. From the `noteForm` object, the function gets the form field named `total` and retrieves the value. It then updates this value within the page.
These kinds of changes are *live*. If you make the changes to the page, reload, and press the button repeatedly, you'll see that the text field is updated each time.
Similarly, you can retrieve the text of the `div` element using DHTML properties. Because it has an `id` of `mainDiv`, you could use the property `innerHTML`, as in:

```
theText = mainDiv.innerHTML;
```

In this case, you see two DHTML techniques: the `forms` array, and calling elements by names based on attribute values, rather than by element names or by overall structure. The problem here is that it doesn't lend itself well to generalization. Yes, you can use a variable for the name of the form, but what if an alternate presentation doesn't actually use a form?

Back to top

DOM and JavaScript

If one principle defined the Document Object Model, it would be that information is arranged as a parent-child hierarchy. For example, the following XML:

```
<parentElement><childElement>My Text Node</childElement></parentElement>
```

has three nodes: the *root* node is `parentElement`. It has one child, the `childElement`. The `childElement` element has as its parent the `parentElement`, and as its child the text node with a value of "My Text Node". The text node has `childElement` as its parent. Two nodes that share a parent are considered siblings. Notice that the text is its own node. Elements don't actually have a value, they simply have text node children.
You may be familiar with the idea of reading the structure of an XML document using Java technology or another language. When you do so, you use an API, with well-defined functions and object properties. For example, this document has an `html` element as its root element, which has two children, `head` and `body`. (I've removed the white space that would normally appear between these elements to simplify matters; browsers are not yet consistent on how they handle this white space.) To access the `body`element using Java technology, you could use several different expressions, assuming that you've named the `Document` object`document`.
The first approach is to get a list of all of the element's children, then choose a specific item from the list, as in:

```
bodyElement = document.getChildNodes().item(0).getChildNodes().item(1);
```

Here, you first get the `html` element, which is the first child of the document, then get its second child, the `body`. (`getChildNodes()` is zero-based.) An alternative approach is to access the `html` element as first child directly, then move to its first child (`head`), and then to that element's next sibling (the second child, `body`):

```
bodyElement = document.getFirstChild().getFirstChild().getNextSibling();
```

From there you can get the type of node:

```
typeInt = bodyElement.getNodeType()
```

Node types are returned as integers, and allow you to handle each node appropriately; an element (type `1`) has a name, but no value, whereas a text node (type `3`) has a value but no name.
Once you know what you have, you can retrieve the name of the element:

```
elementName = bodyElement.getNodeName();
```

or its text contents:

```
elementContent = bodyElement.getFirstChild().getNodeValue();
```

(Remember, the text of an element is a node unto itself, with the element as its parent.)
When you move these functions over to JavaScript, the same basic API is in place, but it's handled slightly differently. For example, properties are accessed directly, rather than through `get` and `set` methods, so the same statements in JavaScript would be represented as

```
bodyElement = document.childNodes.item(0).childNodes.item(1);
bodyElement = document.firstChild.firstChild.nextSibling;
```

From there you can get the type and name of the element, as well as its contents:

```
ElementType = bodyElement.nodeType;
elementName = bodyElement.nodeName;
elementContent = bodyElement.firstChild.nodeValue;
```

Be aware, however, that only properties undergo this name change. Functions that return objects remain constant. For example, you can retrieve a specific object based on its ID, as in:

```
formElement = document.getElementById("noteForm");
```

That's the basic idea. Let's see it in action.

Back to top

Adding a new note

The actual note itself is just a small box with standard text and a link that enables the user to edit the text later, as seen in Figure 2.

Figure 2. The new note

This is the equivalent of the following HTML:

Listing 3. The target HTML

```
<div id="note1" style="width: 100; height:100; border: 1px solid blue;
                       background-color: yellow; position: absolute;
                       top: 150; left: 135">
    <a href="javascript:editNote('note1')">edit</a>
    <br />
    New note
</div>
```

Adding the actual element uses standard DOM techniques, as seen in Listing 4:

Listing 4. Adding the new note element

```
<html>
  <head>
    <title>Getting Sticky</title>
    <style type="text/css">
      * {font-family: sans-serif}
      a {font-size: 6pt}
      .editButton {font-size:6pt}
    </style>

    <script type="text/javascript">
...
      function getCurrentNumber() {
        formElement = document.getElementById("noteForm");
        return formElement.childNodes.item(1).value;
      }

      function makeNewNote(){
        mainDivElement = document.getElementById("mainDiv");

        newNote = document.createElement("div");
        newNote.setAttribute("id", "note"+getCurrentNumber());
```

```
        mainDivElement.appendChild(newNote);

        incrementCurrent();

      }
    </script>
  </head>
  <body>
    <div id="mainDiv" style="height:85%; width:85%; border:3px solid red;
                                          padding: 10px; z-index: -100" >

      <h1>Getting Sticky</h1>

      <form id="noteForm">
          Current number of notes  <input type="text" name="total" value="0"
                                          size="3"/>
          <input type="button" value="Add a new note"
                                          onclick="makeNewNote()"/>
      </form>

    </div>
  </body>
</html>
```

In this case, you've changed the button so that it causes the execution of `makeNewNote()` rather than `incrementCurrent()`, though that function is still in use within `makeNewNote()`. First, you use `getElementById()` to get a reference to the main `div`element that ultimately will contain the note. You then use the `document` object to create a new element, just as you would do in any other language, with the name of `div`. To set the `id` attribute, you simply use the `setAttribute()` method on the new element.
Each note will have a unique `id` attribute, so you need to know what the current total is. To get this information, you start at the level of the form element itself. From there, you retrieve the list of children. The first (with an index of `0`) is the text, and the second (with an index of `1`) is the actual `input` element.
But what about `.value`? Isn't that a typo? Shouldn't it be `nodeValue`?
Actually, no. Remember, elements don't have values. At first glance, it might look as though I'm mixing DOM and DHTML properties, but in actuality, the element retrieved here is not just an implementation of `org.w3c.dom.Element`, it's an implementation of `org.w3c.dom.html.HTMLInputElement`, which also includes a value property that represents the value of the form field. In this way, DOM mimics some (though not all) of the properties that were available through DHTML.
Once the attribute is set, you simply append the new `div` element to the `mainDiv` element, where it will appear. Or at least, it would if it had any presentation properties or text.
To add the style information, you will actually use the DHTML style object:

Listing 5. Adding style information

```
...
      function makeNewNote(){
        mainDivElement = document.getElementById("mainDiv");

        newNote = document.createElement("div");
        newNote.setAttribute("id", "note"+getCurrentNumber());

        newNote.style.width="100";
        newNote.style.height="100";
        newNote.style.border="1px solid blue";
        newNote.style.backgroundColor="yellow";
        newNote.style.position="absolute";
        newNote.style.top=(150);
        newNote.style.left=(25 + 110*getCurrentNumber());

        mainDivElement.appendChild(newNote);

        incrementCurrent();

      }
...
```

The result is a small yellow box with a blue border, as shown in Figure 3.

Figure 3. The empty box

Notice that the `left` property of the note depends on the current number of notes, which increments after each note is added. This way, you can add a series of boxes, as shown in Figure 3.

Back to top

Adding the content

Adding the content of the `div` presents a bit of a problem. I could use the innerHTML property:

```
newNote.innerHTML = "<a href=\"javascript:editNote('note"
                     +getCurrentNumber()+")\">edit</a><br />New note";
```

but how could I do it using straight DOM methods? The first thought would be to simply set the value of the text node child of the`div` element:

```
noteText = document.createTextNode(
          "<a href=\"javascript:editNote('note"+getCurrentNumber()+")\">"+
          "edit</a><br />New note");
newNote.appendChild(noteText);
```

The text does indeed get added, but the results might not quite be what you expect, as shown in Figure 4.

Figure 4. The text in the box

The problem is that you're not really adding text, but rather mixed content, consisting of text and elements. The browser assumes that you mean this as CDATA, which is taken literally, and the elements are not created. Rather than simply adding all of the content in one block, you need to actually add each element:

Listing 6. Adding the content

```
...
  function makeNewNote(){
    mainDivElement = document.getElementById("mainDiv");

    newNote = document.createElement("div");
    newNote.setAttribute("id", "note"+getCurrentNumber());
...
      editLink = getEditLink("note"+getCurrentNumber());
      newNote.appendChild(editLink);
      newNote.appendChild(document.createElement("br"));

      noteText = document.createTextNode("New Form");
      newNote.appendChild(noteText);

    mainDivElement.appendChild(newNote);

    incrementCurrent();

  }

    function getEditLink(thisId){
      editLink = document.createElement("a");
      linkText = document.createTextNode("edit");

      editLink.setAttribute("href", "javascript:editNote('"+thisId+"')");

      editLink.appendChild(linkText);
      return editLink;
    }
...
```

First, you've created a new function, `getEditLink`, which returns an object. That object is the `a` element, which you created using standard DOM methods. Next, you add a standard break tag, `br`, and finally, the node that contains the actual note text. The result is the completed note, with elements intact and ready for use.

Back to top

Changing existing nodes

Now you have the content, but how can you change it? Because you added elements separately, you can just edit the single text node that represents the text of the note. You can do this three ways. First, you can remove the offending node and add a new one:

Listing 7. Removing the node and adding a replacement

```
...
      function editNote(editLink){
        theDiv = document.getElementById(editLink);
        newText = prompt("What should the note say?");

        oldNode = theDiv.firstChild.nextSibling.nextSibling;
        theDiv.removeChild(oldNode);

        newNode = document.createTextNode(newText);
        theDiv.appendChild(newNode);

      }
...
```

Another option is to simply replace the existing node:

Listing 8. Replacing a node

```
...
      function editNote(editLink){
        theDiv = document.getElementById(editLink);
        newText = prompt("What should the note say?");

        oldNode = theDiv.firstChild.nextSibling.nextSibling;
        newNode = document.createTextNode(newText);
        theDiv.replaceChild(newNode, oldNode);
      }
...
```

In this case, the `oldNode` is replaced with the `newNode`, and the document changes accordingly.
Finally, you could simply change the text of the existing node:

Listing 9. Changing the existing node

```
...
      function editNote(editLink){
        theDiv = document.getElementById(editLink);
        newText = prompt("What should the note say?");

        theDiv.firstChild.nextSibling.nextSibling.nodeValue=newText;
      }
...
```

Because `editNote` takes the `id` value of the appropriate `div`, the same function can be used for any note, as seen in Figure 5.

Figure 5. The final page

Back to top

Summary

In this article, you've taken a very basic look at the use of DOM in the JavaScript of a Web page. You can use the same principles anywhere JavaScript is in use, such as within an XSLT style sheet.
The Document Object Model represents elements, text, and other types of nodes within an XML document as a series of parent-child relationships. By manipulating these individual nodes, you can affect the page itself. In addition to DOM Core methods, an XHTML page can also expose properties and methods that are part of the DOM HTML module, which attempts to integrate many of the DHTML properties that programmers have been using for years.

DOM est une spécification W3C finale qui peut être implémentée sans avoir à craindre de modifications ultérieures.

Sommaire

Le but de DOM

Selon le W3C: "DOM permet aux programmes et scripts d'accéder et de modifier dynamiquement le contenu, la structure et le style de documents XML ou HTML".

De quoi est constitué DOM?

Le programmeur dispose d'objets, qui ont des propriétés, des, méthodes et des évènements qui interfacent le document XML ou HTML.
En résumé:
- un ensemble d'objets,
- un modèle pour la façon dont ces objets peuvent être combinés,
- et une interface pour y accéder les manipuler.

DOM XML et HTML

Le coeur de DOM fournit:
1) les interfaces fondamentaux pour représenter tout document structuré.
2) des interfaces étendus qui représentent les documents xml.
La version XML doit implémenter à la fois l'interface xml et l'interface fondamentale.
La version HTML fournit:
1) l'interface fondamentale comme ci-dessus,
2) l'interface étendue pour les documents html.

Comment utiliser DOM?

A travers les objets et leurs attributs et méthodes.

Document

Document est un objet DOM correspondant à la page en cours. Toutefois certaines balises comme <iframe>, <browser> et <tabbrowser> peuvent introduire de nouveaux documents.

Méthodes statiques essentielles et attributs

1. getElementById(x). Retourne la balise dont l'ID est x.
2. innerHTML. Attribut pour lire ou assigner le contenu d'un ID.
3. getElementsByName(x). Retourne la NodeList (liste de noeuds), dont les Nodes (noeuds) sont créés à partir des balises dont la classe (ou nom de balise pour XML) est x.
4. item(n). Retourne l'élément en position n dans une NodeList.
5. firstChild. Attribut désignant le premier élément enfant dans le Node, lequel est retourné par item(n).
6. nextSibling. L'élément suivant. S'utilise après firstChild.

Méthodes dynamiques essentielles

- createElement(type, nom). Crée un élément et retourne un objet Element (un type de Node).
- appendChild(Node). Ajoute un élément à l'instance, en tant que dernier enfant.
- insertBefore(Node, Node).
- removeChild(Node).
- setAttribute(nom, valeur). Ajoute un attribut à l'élément.

Exemple d'utilisation avec JavaScript

```
var anchorList = document.getElementsByTagName("a") ;
for (var i = 0; i < anchorList.length ; i++)
{
 alert("href: " + anchorList[i].href + "\n");
```

```
}
```

Cet exemple parse une page web pour trouver des liens et les afficher.
- *document* est un objet défini dans le coeur de dom pour représenter le document.
- *getElementsByTagName* est une méthode de l'objet qui construit un tableau avec chaque balise correspondant au paramètre, "a" en l'occurrence.
- *href* est une propriété du DOM HTML.
- *anchorList* est une variable déclarée.

Chapitre 5 : XML :

• eXtensible Markup Language

• Generic format for tagging data

• World Wide Web Consortium (W3C)

specification

Like HTML, the tags are symetrical . . .

<open> DATA</open>

Tags can be nested…

<verbs>

<open> DATA</open>

<close> DATA</close>

</verbs>

The design goals for XML are:

– XML shall be straightforwardly usable over the Internet.

– XML shall support a wide variety of applications.

– XML shall be compatible with SGML.

– It shall be easy to write programs which process XML documents.

– The number of optional features in XML is to be kept to the absolute minimum, ideally zero.

– XML documents should be human-legible and reasonably clear.

– The XML design should be prepared quickly.

– The design of XML shall be formal and concise.

– XML documents shall be easy to create.

– Terseness in XML markup is of minimal importance.

Why XML?

• XML is “built in”:

– Databases

– Web Servers

–Browsers

– Application development software

• Applications don’t have to handle:

– Parsing

– Validation

– Translation to presentation format

Why XML?

• It is the currency of web services and SOAP communication.

• XML-appliances are available to optimize security and performance.

• A growing infrastructure of tools (DOM, SAX, XPath, XML Query).

• XML is open, interoperable across vendor platforms

• XML has archival “readability”

Why not XML (yet)?

• XML is a text-formatting convention, and not optimized today for handling binary objects.

• WebServices is the principal mechanism

for carrying XML payloads.

– The SOAP envelope, security and guaranteed

delivery have yet to mature

The good news is that many of the limitations of HTML have been overcome in XML, the Extensible Markup Language. XML is easily comprehensible to anyone who understands HTML, but it is much more powerful. More than just a markup language, XML is a *metalanguage* -- a language used to define new markup languages. With XML, you can create a language crafted specifically for your application or domain.

XML will complement, rather than replace, HTML. Whereas HTML is used for formatting and displaying data, XML represents the contextual meaning of the data.

- **HTML isn't extensible**

An extensible markup language would allow application developers to define custom tags for application-specific situations. Unless you're a 600-pound gorilla (and maybe not even then) you can't require all

browser manufacturers to implement all the markup tags necessary for your application. So, you're stuck with what the big browser makers, or the W3C (World Wide Web Consortium) will let you have. What we need is a language that allows us to make up our own markup tags without having to call the browser manufacturer.

- **HTML is very display-centric**

HTML is a fine language for display purposes, unless you require a lot of precise formatting or transformation control (in which case it stinks). HTML represents a mixture of document logical structure (titles, paragraphs, and such) with presentation tags (bold, image alignment, and so on). Since almost all of the HTML tags have to do with how to display information in a browser, HTML is useless for other common network applications -- like data replication or application services. We need a way to unify these common functions with display, so the same server used to browse data can also, for example, perform enterprise business functions and interoperate with legacy systems.

- **HTML isn't usually directly reusable**

Creating documents in word-processors and then exporting them as HTML is somewhat automated but still requires, at the very least, some tweaking of the output in order to achieve acceptable results. If the data from which the document was produced change, the entire HTML translation needs to be redone. Web sites that show the current weather around the globe, around the clock, usually handle this automatic reformatting very well. The content and the presentation style of the document are separated, because the system designers understand that their content (the temperatures, forecasts, and so on) changes *constantly*. What we need is a way to specify data presentation in terms of structure, so that when data are updated, the formatting can be "reapplied" consistently and easily.

- **HTML only provides one 'view' of data**

It's difficult to write HTML that displays the same data in different ways based on user requests. Dynamic HTML is a start, but it requires an enormous amount of scripting and isn't a general solution to this problem. (Dynamic HTML is discussed in more detail below.) What we need is a way to get all the information we may want to browse at once, and look at it in various ways on the client.

- **HTML has little or no semantic structure**

Most Web applications would benefit from an ability to represent data by meaning rather than by layout. For example, it can be very difficult to find what you're looking for on the Internet, because there's no indication of the meaning of the data in HTML files (aside from META tags, which are usually misleading). Type *red* into a search engine, and you'll get links to Red Skelton, red herring, red snapper, the red scare, Red Letter Day, and probably a page or two of "Books I've Red." HTML has no way to specify what a particular page item means. A more useful markup language would represent information in terms of its meaning. What we need is a language that tells us not how to *display* information, but rather, what a given block of information *is* so we know what to do with it.

HTML example

Take a look at the little chunk of HTML in Listing 1:

```
<!-- The original html recipe -->
<HTML>
<HEAD>
<TITLE>Lime Jello Marshmallow Cottage Cheese Surprise</TITLE>
</HEAD>
<BODY>
<H3>Lime Jello Marshmallow Cottage Cheese Surprise</H3>
```

```
My grandma's favorite (may she rest in peace).
<H4>Ingredients</H4>
<TABLE BORDER="1">
<TR BGCOLOR="#308030"><TH>Qty</TH><TH>Units</TH><TH>Item</TH></TR>
<TR><TD>1</TD><TD>box</TD><TD>lime gelatin</TD></TR>
<TR><TD>500</TD><TD>g</TD><TD>multicolored tiny marshmallows</TD></TR>
<TR><TD>500</TD><TD>ml</TD><TD>cottage cheese</TD></TR>
<TR><TD></TD><TD>dash</TD><TD>Tabasco sauce (optional)</TD></TR>
</TABLE>
<P>
<H4>Instructions</H4>
<OL>
<LI>Prepare lime gelatin according to package instructions...</LI>
<!-- and so on -->
</BODY>
</HTML>
```

Listing 1. Some HTML

(A printable version of this listing can be found at example.html.)

Looking at the HTML code in Listing 1, it's probably clear to just about anyone that this is a recipe for something (something awful, but a recipe nonetheless). In a browser, our HTML produces something like this

Produit	Qte	Prix
Disque dur	10	5000
Lecteur CD	5	1500
Flash Disk	100	1000
Total		**157500**

Lime Jello Marshmallow Cottage Cheese Surprise
My grandma's favorite (may she rest in peace).

Ingredients

Qty	Units	Item
1	box	lime gelatin
500	g	multicolored tiny marshmallows
500	ml	Cottage cheese
	dash	Tabasco sauce (optional)

Instructions

.. Prepare lime gelatin according to package instructions...

```
<?xml version="1.0"?>
<Recipe>
   <Name>Lime Jello Marshmallow Cottage Cheese Surprise</Name>
   <Description>
     My grandma's favorite (may she rest in peace).
   </Description>
   <Ingredients>
      <Ingredient>
         <Qty unit="box">1</Qty>
         <Item>lime gelatin</Item>
      </Ingredient>
      <Ingredient>
         <Qty unit="g">500</Qty>
         <Item>multicolored tiny marshmallows</Item>
      </Ingredient>
      <Ingredient>
         <Qty unit="ml">500</Qty>
         <Item>Cottage cheese</Item>
      </Ingredient>
      <Ingredient>
         <Qty unit="dash"/>
         <Item optional="1">Tabasco sauce</Item>
      </Ingredient>
   </Ingredients>
   <Instructions>
      <Step>
Prepare lime gelatin according to package instructions
      </Step>
      <!-- And so on... -->
   </Instructions>
</Recipe>
```

It will come as little surprise to you, being the astute reader you are, that this recipe in its new format is actually an XML document. Maybe the fact that the file started with the odd header

```
<?xml version="1.0"?>
```

gave it away; in fact, every XML file should begin with this header. We've simply invented markup tags that have a particular meaning; for example, "An `<Ingredient>` is a `<Qty>` (quantity in specified units) of a single `<Item>`, which is possibly`optional`." Our XML document describes the information in the recipe in terms of *recipes,* instead of in terms of how to*display* the recipe (as in HTML). The semantics, or meaning of the information, is maintained in XML because that's what the tag set was designed to do.

Notes on notation

It's important to get some nomenclature straight. In Figure 1, you see a *start tag,* which begins an enclosed area of text, known as an `Item`, according to the *tag name.* As in HTML, XML tags may include a list of *attributes* (consisting of an*attribute name* and an *attribute value.*)

The `Item` defined by the tag ends with the *end tag.*

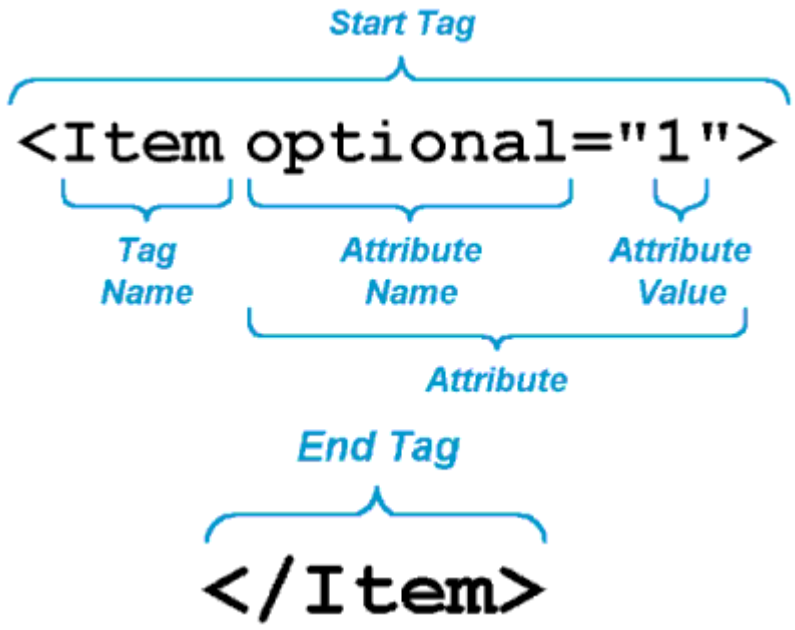


Figure 1. An XML start tag and its corresponding end tag

Not every tag encloses text. In HTML, the `<BR>` tag means "line break" and contains no text. In XML, such elements aren't allowed. Instead, XML has *empty tags,* denoted by a slash before the final right-angle bracket in the tag. Figure 2 shows an empty tag from our XML recipe. Note that empty tags may have attributes. This empty tag example is standard XML shorthand for `<Qty units="g"></Qty>`.

Figure 2. An empty tag

In addition to these notational differences from HTML, the structural rules of XML are more strict. Every XML document must be *well-formed.* What does that mean? Read on!

Ooh-la-la! Well-formed XML

The concept of well-formedness comes from mathematics: It's possible to write mathematical expressions that don't mean anything. For example, the expression

```
2 ( + + 5 (=) 9 > 7
```

looks (sort of) like math, but it isn't math because it doesn't follow the notational and structural rules for a mathematical expression (not on this planet, at least). In other words, the "expression" above isn't *well-formed.* Mathematical expressions must be well-formed before you can do anything useful with them, because expressions that aren't well-formed are meaningless.

A well-formed XML document is simply one that follows all of the notational and structural rules for XML. Programs that intend to process XML should reject any input XML that doesn't follow the rules for being well-formed. The most important of these rules are as follows:

- **No unclosed tags**
  You can get away with all kinds of wacko stuff in HTML. For example, in most HTML browsers, you can "open" a list item with `<LI>` and never "close" it with `</LI>`. The browser just figures out where the `</LI>` would be and automatically inserts it for you. XML doesn't allow this kind of sloppiness. Every start tag must have a corresponding end tag. This is because part of the information in an XML file has to do with how different elements of information relate to one another, and if the structure is ambiguous, so is the information. So, XML simply doesn't allow ambiguous structure. This nonambiguous structure also allows XML documents to be processed as data structures (trees), as I'll explain shortly in the discussion of the Document Object Model.

- **No overlapping tags**
  A tag that opens inside another tag must close before the containing tag closes. For example, the sequence

  ```
  <Tomato>
  Let's call <Potato>the whole thing off</Tomato>
  </Potato>
  ```

  isn't well-formed because `<Potato>` opens inside of `<Tomato>` but doesn't close inside of `<Tomato>`. The correct sequence must be

  ```
  <Tomato>
  Let's call <Potato>the whole thing off</Potato>
  </Tomato>
  ```

  In other words, the structure of the document must be strictly hierarchical.

- **Attribute values must be enclosed in quotes**
  Unlike HTML, XML doesn't allow "naked" attribute values (i.e., HTML tags like `<TABLE BORDER=1>`, where there are no quotes around the attribute value). Every attribute value must have quotes (`<TABLE BORDER="1">`).

- **The text characters (<), (>), and (") must always be represented by 'character entities'**
  To represent these three characters (left-angle bracket, right-angle bracket, and double quotes) in the text part of the XML (not in the markup), you must use the special character entities (`&lt;`), (`&gt;`), and (`&quot;`), respectively. These characters are special characters for XML. An XML file using, say, the double quote character in the text enclosed in tags in an XML file isn't well-formed, and correctly designed XML parsers will produce an error for such input.

XML and Java

Pour utiliser XML avec le langage JAVA il existe deux packages le DOM et le SAX , par la suite je vous donne quelques exemple de lecture , création et modification d'un document XML .

Malgré que le DOM qu'il le plus gourmand en terme de consommation mémoire , il est le plus facile package a utilisé:

1. ***Lecture d'un document XML.***

Voici un exemple.xml

```
<?xml version="1.0"?>
<company>
	<staff>
		<firstname>yong</firstname>
		<lastname>mook kim</lastname>
		<middlename>mkyong</middlename>
		<salary>100000</salary>
	</staff>
	<staff>
		<firstname>low</firstname>
		<lastname>yin fong</lastname>
		<middlename>fong fong</middlename>
		<salary>200000</salary>
	</staff>
</company>
```

Exemple java de lecture ReadXMLFile

```
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import org.w3c.dom.Element;
import java.io.File;

public class ReadXMLFile {
	public static void main(String argv[]) {
	  try {	File fXmlFile = new File("c:\\file.xml");
```

```
                    DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
                    DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
                    Document doc = dBuilder.parse(fXmlFile);
                    doc.getDocumentElement().normalize();
                    System.out.println("Root element :" + doc.getDocumentElement().getNodeName());
                    NodeList nList = doc.getElementsByTagName("staff");
                    System.out.println("----------------------");
                    for (int temp = 0; temp < nList.getLength(); temp++) {
                      Node nNode = nList.item(temp);
                      if (nNode.getNodeType() == Node.ELEMENT_NODE) {
                        Element eElement = (Element) nNode;
                        System.out.println("First Name : " + getTagValue("firstname", eElement));
                        System.out.println("Last Name : " + getTagValue("lastname", eElement));
                        System.out.println("middle Name : " + getTagValue("middlename", eElement));
                        System.out.println("Salary : " + getTagValue("salary", eElement));
                      }          }
            } catch (Exception e) {
                    e.printStackTrace();
            }   }
  private static String getTagValue(String sTag, Element eElement) {
          NodeList nlList = eElement.getElementsByTagName(sTag).item(0).getChildNodes();
      Node nValue = (Node) nlList.item(0);

          return nValue.getNodeValue(); }
```

le resultat sur l'ecran

```
Root element :company
----------------------
First Name : yong
Last Name : mook kim
middle Name : mkyong
Salary : 100000
First Name : low
Last Name : yin fong
middle Name : fong fong
Salary : 200000
```

2. ***Mettre a jour un document XML***

Dans cet exemple on montre que DOM peut être utilisé pour

1. Ajout d'un nouvel élément
2. Mise à jour d'attribut d'un élément
3. Mise à jour de la valeur d'un élément
4. Suppression d'un élément

Voici un fichier d'entrée

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<company>
   <staff id="1">
           <firstname>yong</firstname>
           <lastname>mook kim</lastname>
           <middlename>mkyong</middlename>
           <salary>100000</salary>
   </staff>
</company>
```

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<company>
   <staff id="2">
           <lastname>mook kim</lastname>
           <middlename>mkyong</middlename>
           <salary>2000000</salary>
         <age>28</age>
   </staff>
</company>
```

Voici le code à utiliser

```
import java.io.File;
import java.io.IOException;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
```

```
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

public class ModifyXMLFile {
        public static void main(String argv[]) {
          try {     String filepath = "c:\\file.xml";
                    DocumentBuilderFactory docFactory = DocumentBuilderFactory.newInstance();
                    DocumentBuilder docBuilder = docFactory.newDocumentBuilder();
                    Document doc = docBuilder.parse(filepath);

                    Node company = doc.getFirstChild();
                    Node staff = doc.getElementsByTagName("staff").item(0);
                    NamedNodeMap attr = staff.getAttributes();
                    Node nodeAttr = attr.getNamedItem("id");
                    nodeAttr.setTextContent("2");
                    Element age = doc.createElement("age");
                    age.appendChild(doc.createTextNode("28"));
                    staff.appendChild(age);
                    NodeList list = staff.getChildNodes();

                    for (int i = 0; i < list.getLength(); i++) {
                     Node node = list.item(i);
                       // get the salary element, and update the value
                       if ("salary".equals(node.getNodeName())) {
                                node.setTextContent("2000000");    }
                       if ("firstname".equals(node.getNodeName())) {
                                staff.removeChild(node);              }                        }
                    // write the content into xml file
                    TransformerFactory transformerFactory = TransformerFactory.newInstance();
                    Transformer transformer = transformerFactory.newTransformer();
                    DOMSource source = new DOMSource(doc);
                    StreamResult result = new StreamResult(new File(filepath));
                    transformer.transform(source, result);
                    System.out.println("Done");
          } catch (ParserConfigurationException pce) {
                    pce.printStackTrace();
          } catch (TransformerException tfe) {
                    tfe.printStackTrace();
          } catch (IOException ioe) {
                    ioe.printStackTrace();
          } catch (SAXException sae) {
                    sae.printStackTrace();
          }
        }
}
```

1- Creation d'un document XML

Voici le fichier de sortie

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<company>
          <staff id="1">
                    <firstname>yong</firstname>
                    <lastname>mook kim</lastname>
```

```
                <middlename>mkyong</middlename>
                <salary>100000</salary>
        </staff>
</company>
```

Le code est le suivant :

```
package com.mkyong.core;
 import java.io.File;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Attr;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class WriteXMLFile {
        public static void main(String argv[]) {
          try {
                DocumentBuilderFactory docFactory = DocumentBuilderFactory.newInstance();
                DocumentBuilder docBuilder = docFactory.newDocumentBuilder();
                // root elements
                Document doc = docBuilder.newDocument();
                Element rootElement = doc.createElement("company");
                doc.appendChild(rootElement);
                // staff elements
                Element staff = doc.createElement("Staff");
                rootElement.appendChild(staff);
                // set attribute to staff element
                Attr attr = doc.createAttribute("id");
                attr.setValue("1");
                staff.setAttributeNode(attr);
                Element firstname = doc.createElement("firstname");
                firstname.appendChild(doc.createTextNode("yong"));
                staff.appendChild(firstname);
                Element lastname = doc.createElement("lastname");
                lastname.appendChild(doc.createTextNode("mook kim"));
                staff.appendChild(lastname);
                Element middlename = doc.createElement("middlename");
                nickname.appendChild(doc.createTextNode("mkyong"));
                staff.appendChild(middlename);
                Element salary = doc.createElement("salary");
                salary.appendChild(doc.createTextNode("100000"));
                staff.appendChild(salary);
                TransformerFactory transformerFactory = TransformerFactory.newInstance();
                Transformer transformer = transformerFactory.newTransformer();
                DOMSource source = new DOMSource(doc);
                StreamResult result = new StreamResult(new File("C:\\file.xml"));
                transformer.transform(source, result);
                System.out.println("File saved!");
          } catch (ParserConfigurationException pce) {
                pce.printStackTrace();
          } catch (TransformerException tfe) {
                tfe.printStackTrace();
          }
        }
}
```

3- <u>Compter les balises d'un document :</u>

```
import java.io.IOException;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;

public class CountXMLElement {
  public static void main(String argv[]) {
        try {
                String filepath = "c:\\file.xml";
                DocumentBuilderFactory docFactory = DocumentBuilderFactory.newInstance();
                DocumentBuilder docBuilder = docFactory.newDocumentBuilder();
                Document doc = docBuilder.parse(filepath);
                NodeList list = doc.getElementsByTagName("staff");
                System.out.println("Total of elements : " + list.getLength());
        } catch (ParserConfigurationException pce) {
                pce.printStackTrace();
        } catch (IOException ioe) {
                ioe.printStackTrace();
        } catch (SAXException sae) {
                sae.printStackTrace();
        }
  }
}
```