Ajax - javascript intro

Andrea Ferracani
andreaferracani@gmail.com

Javascript è uno dei tre elementi fondamentali del web design, è un linguaggio di programmazione con oggetti, funzioni e proprietà:

- XHTML: struttura semantica del documento
- CSS: positioning e style
- DOM scripting: enhance not provides comportamento ed interazione

Enhancement or degradation sono essenziali nell'uso di javascript, perchè ajax risiede e dipende da una tecnologia.
L'accessibilità dell'informazione è la cosa primaria

Javascript è un linguaggio interpretato da un web browser e dunque risiede su software e tecnologie che possono essere molto diverse.

Dunque i requisiti fondamentali sono:

- Standard compliant
- Maintainable: facile da riusare, ad oggetti
- Accessibile, anche per chi non ha javascript enabled
- Usabile: efficace, efficiente e soddisfacente

Dunque la parte di scripting deve essere separata dala struttura del documento. Si fa come per i CSS. La sintassi giusta è questa:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>External File JavaScript</title>
<script type="text/javascript" src="source.js"></script>
</head>
<body>
<h1>External File Example</h1>
</body></html>
```

Uno script importato in questo modo serve ad interagire conil DOM.

DOM: Document object model.
Non significa jvascript, è uno web standard che rappresenta il documento.

Specifiche: http://www.w3.org/DOM - Specificano gli oggetti, le funzioni e le proprietà che linguaggi come javascript devono implementare per interagire con la struttura

Esistono 3 livelli di specifiche del DOM, la prima del 1998, il core; la seconda del 2000 e una piuttosto recente divise a loro volta in numerose sottosezioni con specifiche per task. La difficoltà è che non tutti i browsers seguono o implementano le specifiche stesse.

DOM

- Non è javascript
- Non è DHTML

DHTML è proprietario, contiene molti oggetti non

standard come document.all (getElementsByTagName('*');

Le specifiche del WRC revedono una serie di interfaces che degli object devono implementare per interagire con il DOM.

Non solo javascript, ma qualunque linguaggio in grado di parsare un xml può implementare queste interfaces.

Javscript le implementa automaticamente. Non potete lavorare direttamente su classi e interfacce

Ci sono diversi livelli del DOM:

- level 0: si riferisce al DHTML
- level 1: (http://www.w3.org/DOM/DOMTR#dom1) - 1998 - divido in due parti DOM core: il funzionamento dell'albero e DOM HTML: oggetti, proprietà e funzioni degli elements es. Document, Node, Attr, Element, Text,
  HTMLDocument, HTMLElement, and HTMLCollection

Ci sono diversi livelli del DOM:

- level 0: si riferisce al DHTML
- level 1: (http://www.w3.org/DOM/DOMTR#dom1) - 1998 - divido in due parti DOM core: il funzionamento dell'albero e DOM HTML: oggetti, proprietà e funzioni degli elements es. Document, Node, Attr, Element, Text,
  HTMLDocument, HTMLElement, and HTMLCollection
- Level 2 DOM (http://www.w3.org/DOM/DOMTR#dom2), 2000

Diviso in: DOM core, DOM html con aggiornamenti ai precedenti e DOM events:
ha introdotto i mouse-related events, no keyboard, DOM style, DOM traversal and range, DOM views:
con la possibilità di modificare una vista

- Level 3 DOM (http://www.w3.org/DOM/DOMTR#dom3): DOM core, DOM load and save (xml), DOM validation: validare un documento in base al doctype

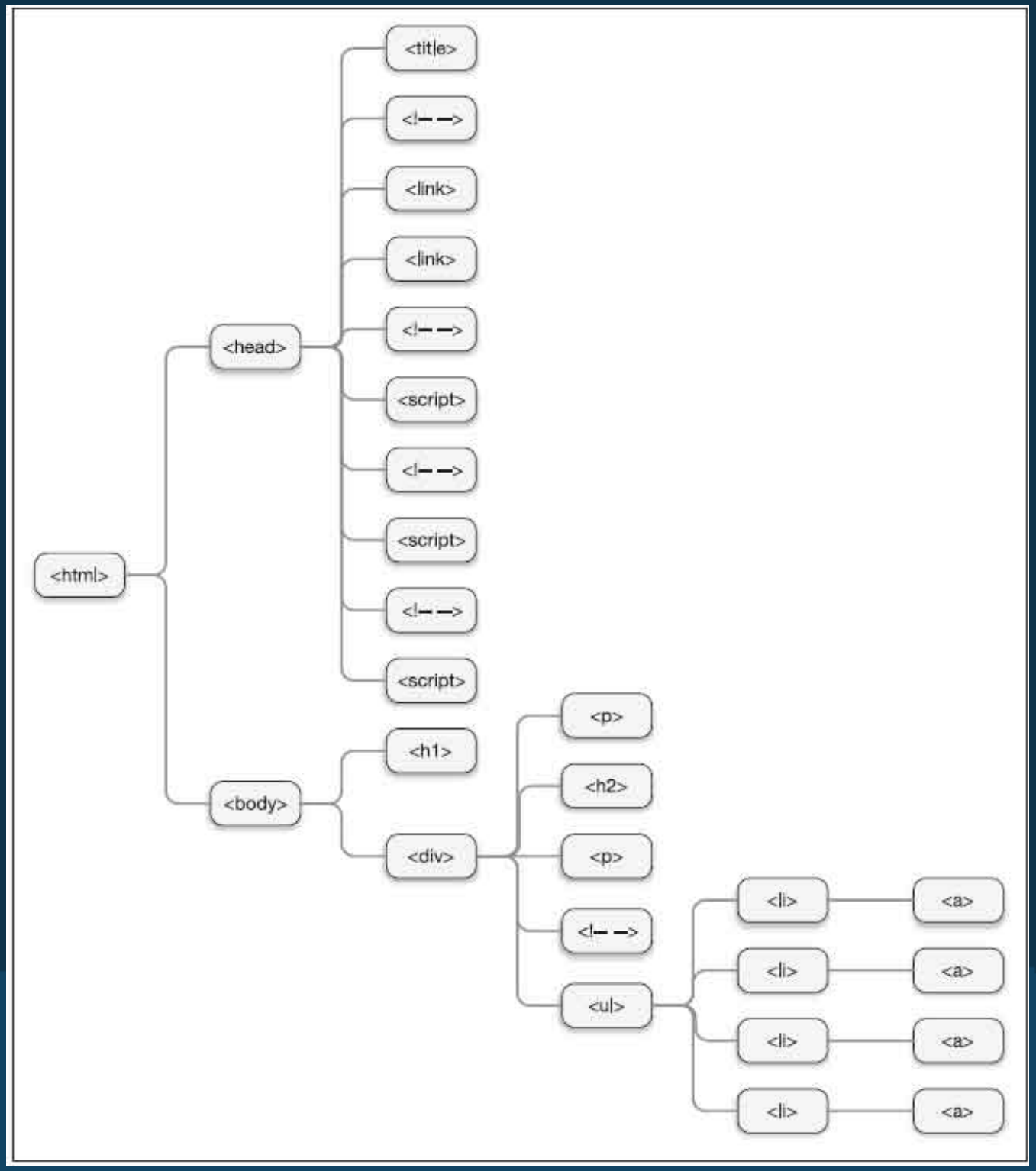
Per verificare se un browser supporta un preciso livello del DOM si può usare un codice tipo:

```
if(document.implementation) {
if(document.implementation.
hasFeature('Core','2.0')) {
alert('DOM2 Core Supported');
} else {
alert('DOM2 Core Not Supported')
}
} else {
alert('No DOMImplementation Support');
}
```

oppure:
http://www.w3.org/2003/02/06-dom-support.html

Cmq quasi tutti i browser adesso supportano il DOM 2

Vedi dom testing samples


<html>
<head>
<title>
<!-- -->
<link>
<link>
<!-- -->
<script>
<!-- -->
<script>
<!-- -->
<script>
<body>
<h1>

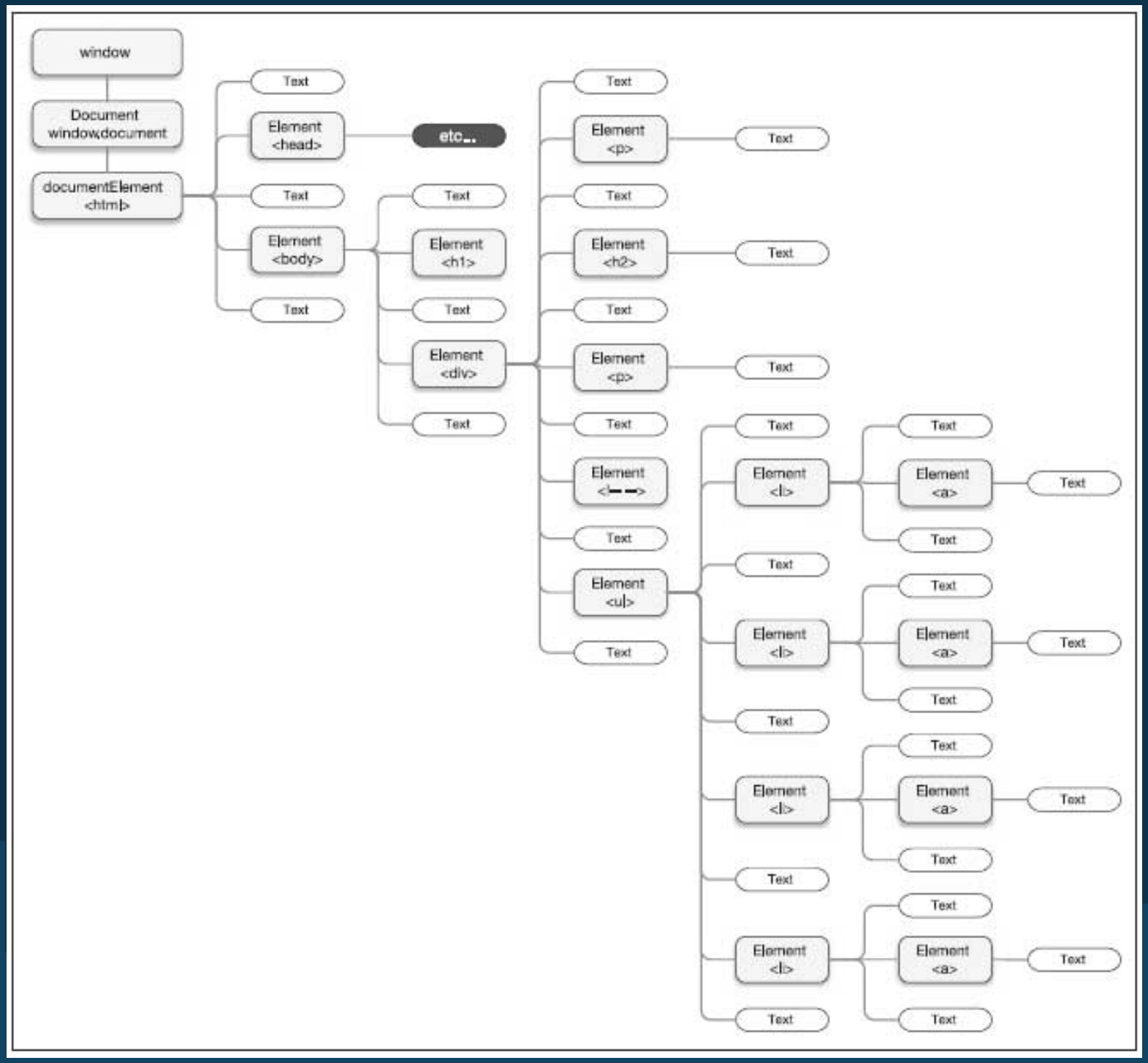
<p>
<h2>
<p>
<!-- -->

<a>

<a>

<a>

<a>


window
Document
window.document
documentElement
<html>
Text
Element
<head>
etc...
Text
Element
<body>
Text
Text
Element
<h1>
Text
Element

Text
Text
Element
<p>
Text
Element
<h2>
Text
Element
<p>
Text
Element
<!-- -->
Text
Element

Text
Text
Element

Text
Element
<a>
Text
Text
Text
Element

Text
Element
<a>
Text
Text
Text
Element

Text
Element
<a>
Text
Text
Text
Element

Text
Element
<a>
Text
Text

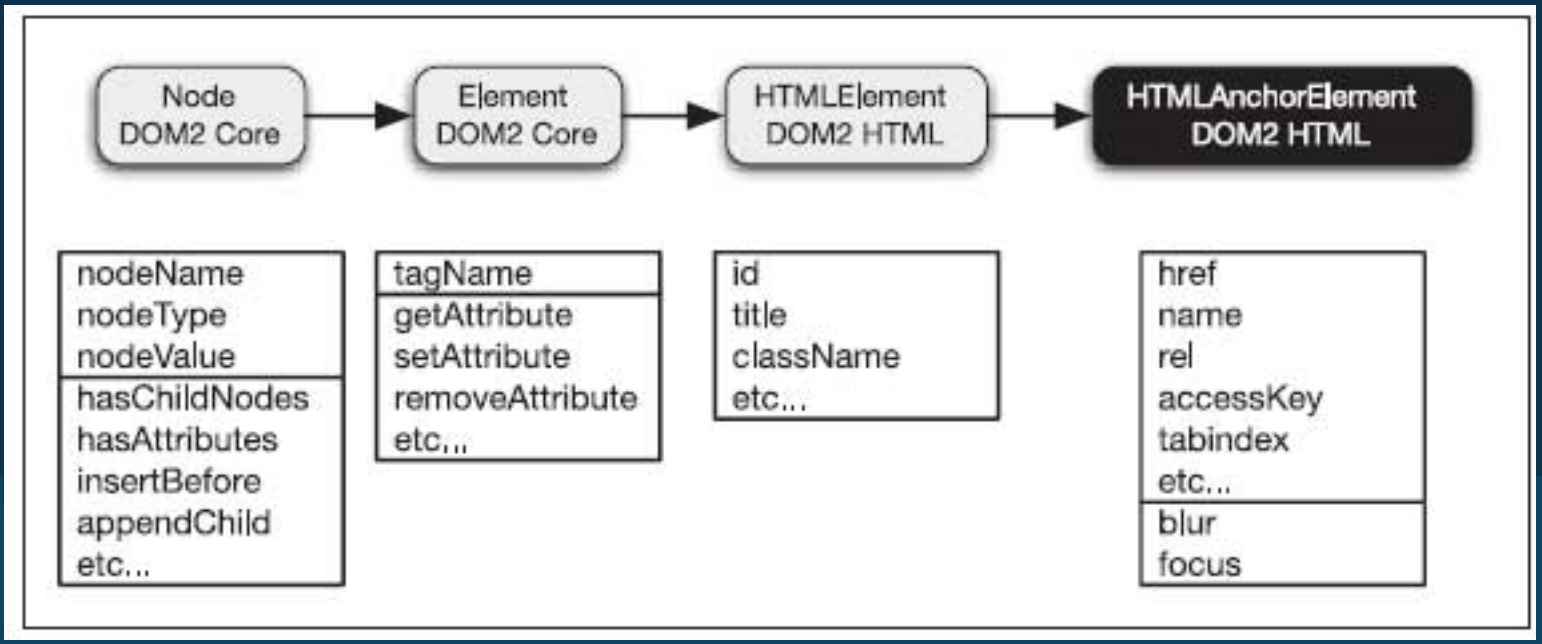
Quando il browser legge il nostro albero in sample.html cerca di convertire gli elementi in elementi del DOM secondo le sue specifiche

per es.
document = tutto il documento
documentElement = <html>

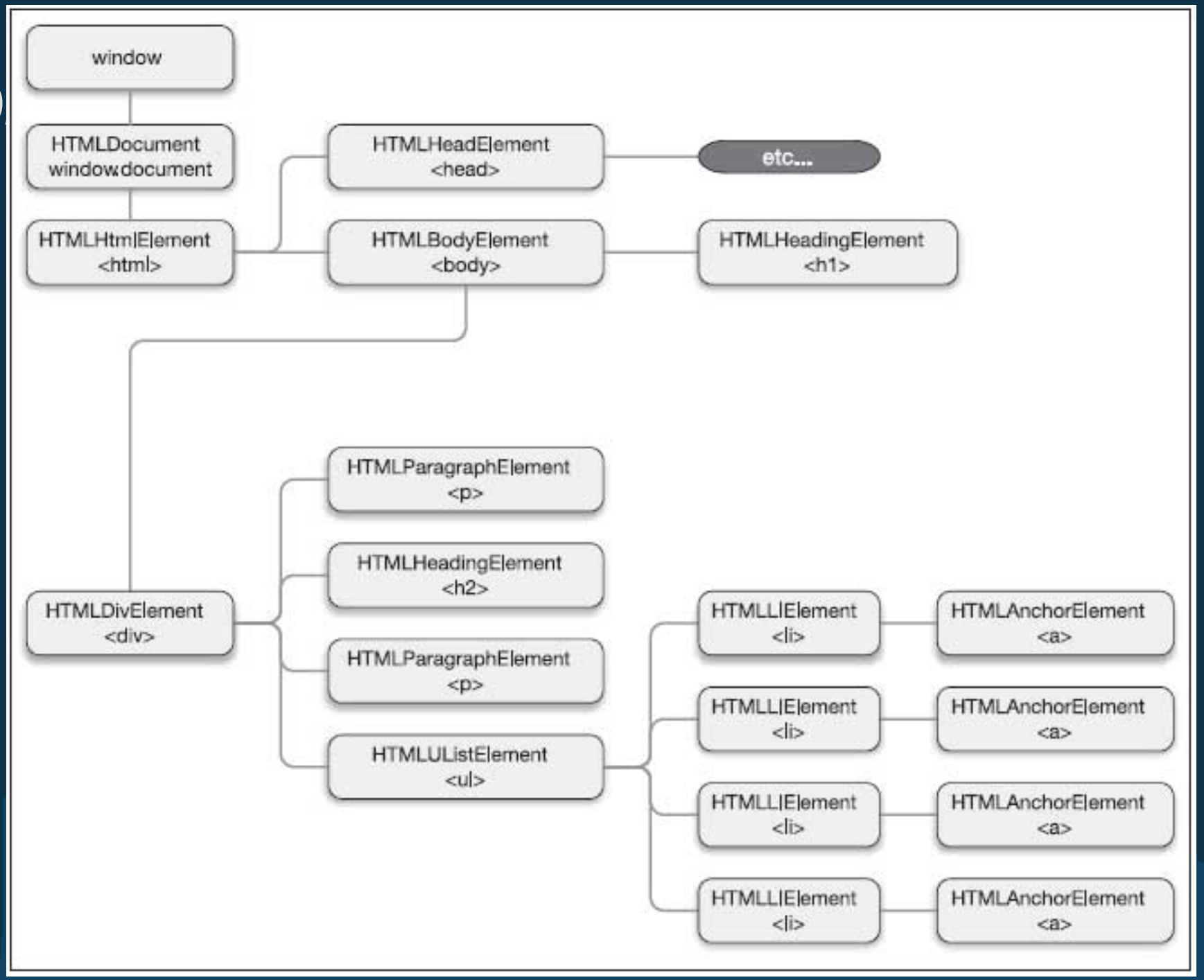
Attenzione! come si vede dalla figura firefox converte tutti gli spazi bianchi in text node, mentre explorer no... bisogna stare attenti quando per es, si itera sui figli di un certo tag

Tutti gli elementi ereditano da element object quindi ne hanno metodi e proprietà, però hanno anche metodi e propietà in base a che elemnti sono:

HTMLParagraphElement è diverso da HTMLAnchorElement



DO


window
HTMLDocument
window.document
HTMLHeadElement
<head>
etc...
HTMLHtmlElement
<html>
HTMLBodyElement
<body>
HTMLHeadingElement
<h1>
HTMLParagraphElement
<p>
HTMLHeadingElement
<h2>
HTMLDivElement

HTMLParagraphElement
<p>
HTMLULListElement

HTMLLIElement

HTMLAnchorElement
<a>
HTMLLIElement

HTMLAnchorElement
<a>
HTMLLIElement

HTMLAnchorElement
<a>
HTMLLIElement

HTMLAnchorElement
<a>

Ogni element estende in Node object per il quale i metodi principali e proprietà disponibili sono:

- nodeName
- nodeValue
- nodeType
- parentNode
- childNodes
- firstChild
- lastChild
- previousSibling
- nextSibling
- attributes

- ownerDocument

Questi metodi non sono disponibili per tutti i tag, bisogna fare attenzione da chi ereditano e leggere le specifiche. Per es nodeValue non funziona su un <a>

Type of nodes
principlali:

1 Node.ELEMENT_NODE
2 Node.ATTRIBUTE_NODE
3 Node.TEXT_NODE

Extensions

Per lavorare con javascript sono necessarie alcune estensioni:

Firefox

- Firebug: http://getfirebug.com
- Web Developer toolbar: http://chrispederick.com/work/webdeveloper

IE7

- http://www.my-debugbar.com/wiki/CompanionJS/HomePage

- Internet explorer developer toolbar

http://www.microsoft.com/downloads/details.aspx?FamilyID=e59c3964-672d-4511-bb3e-2d5e1db91038&displaylang=en

Javascript syntax

- Funzioni e variabili sono case sensitive, function myFunction() { } è diverso da function MyFunction() { }
- non c'è differenza fra virgolette semplici 'string' o doppie "string", ma si preferisce usare le semplici per distinguere dalle specifiche dell'xhtml che prevede virgolette doppi per i valori degli attributi dei tag e per evitare caratteri di escape

es.

```
var html = '<h2 class="a">A list!</h2>'
+ '<ol>'
+ '<li class="a">Foo</li>'
+ '<li class="a">Bar</li>'
+ '</ol>';
```

meglio di

```
var html = "<h2 class=\"a\">A list!</h2>"
+ "<ol>"
+ "<li class=\"a\">Foo</li>"
+ "<li class=\"a\">Bar</li>"
+ "</ol>";
```

anche se l'escape è sempre valido per i single quote

```
var html = '<p class="a">Don\'t forget to escape
single quotes!</p>';
```

BREAKING LINES

```
var html = '<h2 class="a">A list!</h2>
<ol>
<li class="a">Foo</li>
<li class="a">Bar</li>
</ol>';
```

causa un errore perchè il browser interpreta gli a capo come semicolon, si possono usare gli escape.

```
var html = '<h2 class="a">A list!</h2>\
<ol>\
<li class="a">Foo</li>\
<li class="a">Bar</li>\
</ol>';
```

ma è preferibile usare l'operatore di concatenamento +

```
var html = '<h2 class="a">A list!</h2>'
+ '<ol>'
+ '<li class="a">Foo</li>'
+ '<li class="a">Bar</li>'
+ '</ol>';
```

- i semicolon non sono necessari ma è buona pratica usarli
- javascript non supporta l'overloading delle funzioni, cioè non le distingue in base agli arguments:

```
function myFunction(a,b) { alert(a+b); }
function myFunction(a) { alert(a); }
```

javascript rimpiazza semplicemente la prima con la seconda

- si può sovrascrivere qualsiasi oggetto javascript, con le proprie funzioni:

```
function alert(message) {
Mine.$('messageBox').appendChild
(document.createTextNode(message));
}
```

- funzioni anonime, cioè senza nome:

```
function clicked() {
alert('Linked to: ' + this.href);
}
var anchor = Mine.$('someId');
Mine.addEvent(anchor, 'click', clicked);

var anchor = Mine.$('someId');
addEvent(anchor, 'click', function () {
alert('Linked to: 'this.href);
});
```

- Scope: è lo spazio di codice che ha accesso a una certa proprietà o funzione:

```
function myFunction() {
var myVariable = 'inside';
}
```

lo scope della variabile è ristretto a myFunction
questo quindi non funziona:

```
function myFunction() {
var myVariable = 'inside';
}
```

```
myFunction();
alert(myVariable);
```

questo è alla base di molti errori:

```
Mine.addEvent(window, 'load', function(W3CEvent)
{
for (var i=1 ; i<=3 ; i++ ) {
var anchor = document.getElementById('anchor' +
i);
Mine.addEvent(anchor,'click',function() {
alert('My id is anchor' + i);
});}}); dà sempre 4
```

vedi scopechain example1

```
function registerListener(anchor, i) {
Mine.addEvent(anchor, 'click', function() {
alert('My id is anchor' + i);
});}
function initAnchors(W3CEvent) {
for ( i=1 ; i<=5 ; i++ ) {
var anchor = document.getElementById('anchor'+i);
registerListener(anchor,i);
}}
Mine.addEvent(window, 'load', initAnchors);
/*In questo modo registriamo il listener ogni volta e
passiamo il parametro giusto che resta in quello
scope*/ vedi example2 ed example3
```

- Iterating:

```
var list = [1,2,3,4];
for( i=0 ; i<list.length ; i++ ) {
alert(list[i]);
}

var list = [1,2,3,4];
for( i in list ) {
alert(list[i]);
}
```

```
var all = document.body.getElementsByTagName
('*');
for( i=0 ; i<all.length ; i++ ) {
// Do something with all[i] element
}
```

i due approcci sono indifferenti se si cicla su un oggetto di tipo Array, ma attenzione se si cicla in un oggetto, potrebbe avere altre proprietà

- reference and call

var foo = exampleFunction(); - si esegue la funzione

var foo = exampleFunction; - si fa una reference

```
function loadPage() {
// Load script
}
// No parentheses
Mine.addEvent(window, 'load', loadPage);
```

PROGRAMMAZIONE AD OGGETTI

Cos'è un oggetto? Un oggetto è un istanza di un set di variabili o proprietà e metodi. Si intende programmazione ad oggetto quella in cui le diverse funzionalità sono eseguite attraverso l'interazione di vari oggetti

javascript è un prototype style OOP in quanto non ci sono classi ma ogni cosa è copia di un oggetto. Tutto, da una funzione a una stringa è un oggetto

In javascript ci sono due tipi principali di oggetti:

- Function object like alert('argument');
- Object object like var obj = new Object(); obj('argument'); //will error as obj is not a Function

Inoltre ci sono una serie di oggetti built-in:

- Object di base
- Function che è un oggetto con argomenti
- Array
- String, Number e Boolean

- Math, Date, RegExp

Tutti questi oggetti si istanziano con la parola chiave new

```
var myDate = new Date();
var myObject = {}
var my Array = []
```

EREDITARIETA'

Non c'è in javascript se non in questa forma:

```
// Create a person object instance
var person = {};
person.getName = function() { ... };
person.getAge = function() { ... };
// Create an employee object instance
var employee = {};
employee.getTitle = function() { ... };
employee.getSalary = function() { ... };
```

```
// Inherit methods from person
employee.getName = person.getName;
employee.getAge = person.getAge;
```

In pratica si copiano i metodi

ISTANZIARE UN OGGETTO PROPRIO

```
function myConstructor(message) {
    alert(message);
    this.myMessage = message;
}
```

var myObject = new myConstructor('Instantiating myObject!'); e no possiamo anche accedere la proprietà myMessage

Vedi Object/instantiation

Quando scriviamo una funzione generica in questa forma

myFunction('Without window object'); è un metodo dell'oggetto globale window ed equivale a window.myFunction('With window object');

FUNZIONI

```
// Create an Object object instance
var myObject = new Object(); //or var example =
{};
// Add a property
myObject.name = 'Jeff';
// Add a method
myObject.alertName = function() {
alert(this.name);
}
// Execute the method
myObject.alertName();
```

una funzione dichiarata così si dice statica ed è valida olo per quell'istanza. Per aggiungere metodi pubblici si usa la proprieta prototype di un oggetto.

E' una proprietà speciale che descrive la struttura interna dell'oggetto stesso. E' lo stampo su cui è formato l'oggetto, l'istanza.

es.

```
// Create the constructor
function myConstructor(message) {
alert(message);
this.myMessage = message;
}
// Add a public method
myConstructor.prototype.clearMessage = function
(string) {
this.myMessage += ' ' + string;
}
```

vedi instantiating/public_methods

uso:

```
var myObject = new myConstructor('Hello World!');
myObject.clearMessage();
```

Per ora la proprietà myMessage è pubblica, ci posso accedere dall'esterno; e se volessi una proprietà privata?

```
function myConstructor(message) {
this.myMessage = message;
// Private properties
```

```
var separator = ' -';
var myOwner = this;
// Private method
function alertMessage() {
alert(myOwner.message);
}
// Show the message on instantiation
alertMessage();
}
```

una proprietà o funzione privata è definita all'interno di un'altra funzione, la proprietà è preceduta da var.

Vedi instantiating/private_members.html

Per avere accesso ad un metodo private si deve creare una funzione detta privileged che si definisce nel costruttore usando this e rendendola dunque pubblica.

A che serve tutto ciò?

Serve soprattutto a creare una distinzione fra metodi pubblici e il funzionamento interno degli oggetti... si usa soprattutto se si deve sviluppare un progetto in più developers

SINTASSi

object literal:

```
var myObject = {
propertyA:'value',
propertyB:'value',
methodA:function() { },
methodB:function() { },
methodC:function() { },
methodD:function() { }
}
```

equivale a scrivere:

```
var myObject = new Object();
myObject.propertyA = 'value';
myObject.propertyB = 'value';
myObject.methodA = function() { };
myObject.methodB = function() { };
myObject.methodC = function() { };
myObject.methodD = function() { };
```

così stiamo effettivamente applicandole a una sola istanza però

per scrivere una 'classe':

```
function myConstructor() {
// Private and privileged members
};
// Public members
myConstructor.prototype = {
propertyA:'value',
propertyB:'value',
methodA:function() { },
methodB:function() { },
methodC:function() { },
methodD:function() { }}
```

Ancora scope:

se scrivete in un file .js una funzione e provate a chiamarla il suo context di esecuzione è la window

vedi cartella window/this e with without

ma se noi vogliamo chiamarla su un particolare elemento?

es.

```
var sound = 'mySound!';
function mySound() {
this.style.color='green';
alert(sound);
}
function initPage() {
var example = Mine.$('example');
// Using the event attribute method
example.onclick = mySound;
// or using the Mine.addEvent method
Mine.addEvent(example,'mouseover', mySound);
}
Mine.addEvent(window,'load',initPage);
```

si possono usare call and apply per definire l'ambito di esecuzione di una funzione:

```
function doubleCheck() {
this.message = 'Are you sure you want to leave?';
}
doubleCheck.prototype.sayGoodbye = function() {
return confirm(this.message);
}
initPage() {
var clickedLink = new doubleCheck();
var links = document.getElementsByTagName('a');
```

```
for (var i=0 ; i<links.length ; i++) {
Mine.addEvent(links[i], 'click', clickedLink.
sayGoodbye);
}
}
Mine.addEvent(window,'load',initPage);
```

non funziona non c'è nessun oggetto clickedlink nell'ambito di esecuzione di a.

```
functionReference.call(object, argument1,
argument2, ...)
functionReference.apply(object, arguments);
```

Try e catch (debugging)

```
var sound = 'mySound!';
function mySound() {
this.style.color='green';
alert(sound);
}
try {
mySound();
} catch(theException) {
alert('Oops, we caught an exception! Name: '
+ theException.name + ', message: '
+ theException.message);
}
```

JAVASCRIPT INCLUDE

Bisogna separare il behaviour dal markup.

Ci sono diversi modi per inserire javascript in una pagina web ma non tutti sono giusti.

Inline javascript:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Inline JavaScript</title>
</head>
```

```
<body>
<h1>Inline Example</h1>
<script type="text/javascript">
//JavaScript Code
</script>
</body>
</html>
```

ma stiamo mischiando comportamento e markup, così anche nel caso che lo mettessimo nell'head (solo a un livello minore)

es.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Head JavaScript</title>
<script type="text/javascript">
//JavaScript Code
</script>
</head>
<body>
<h1>Head Example</h1>
</body>
</html>
```

Correct method: external source file

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>External File JavaScript</title>
<script type="text/javascript" src="source.js"
></script>
</head>
<body>
<h1>External File Example</h1>
</body>
</html>
```

JAVASCRIPT prefix:

Google

L'attributo target non è xhtml script, dunque uiamo js.

<a href="javascript:window.open
('http://www.google.com');">
Google

ma il prefisso può gestire solo una funzione e inoltre

se restituisce qualcosa la pagina iniziale viene sovrascritta. Possiamo usare una funzione che non ritorna niente

Vedi sample 1_start/samples/include.

Altrimenti possiamo usare il gestore onclick.

```
<a href="#" onclick="window.open('http://www.
google.com');">
google</a>
```

ma nel caso js sia disabilitato non funziona il link

Soluzioni migliori:

<a href="http://advanceddomscripting.com"
onclick="this.href=
'javascript:popup
(\'http://ww.google.com\');"">
Google

best: <a href="http://www.google.com" onclick="
popup(this.href);return false;">google

return false; previene l'azione di default che è
seguire il link

Ma cmq i nostri script sono sempre mischiati al markup, dunque dimenticare tutto ciò.

```
// Add a load event to alter the page
Mine.addEvent(window,'load',function(W3CEvent) {
// Locate all the anchor tags with the popup class
in the document
var popups = Mine.getElementsByClassName
('popup', 'a');
for(var i=0 ; i<popups.length ; i++ ) {
// Add a click event listener to each anchor
Mine.addEvent(popups[i],'click',function(W3CEvent)
```

```
{
// Open the window using the href attribute
window.open(this.href);
// Cancel the default event
Mine.preventDefault(W3CEvent);
});
}
})
```

Un buon esempio di separazione script - markup

http://www.huddletogether.com/projects/lightbox2/

VERSION CHECK

è concettualmente sbagliato il browser sniffing:

```
if ( pitcher.name == 'Addison' ) {
pitcher.screwball.throw();
} elseif ( pitcher.name == 'Stephanie' ) {
pitcher.fastball.throw();
} else {
alert( 'Sorry, you can't throw the ball.' );
}
```

se restringono le funzionalità solo per alcuni

CAPABILITY DETECTION

```
if ( pitcher.screwball ) {
pitcher.screwball.throw();
} elseif ( pitcher.fastball ) {
pitcher.fastball.throw();
} else {
alert( 'Sorry, you can't throw the ball.' );
}

quindi if (document.body) {
// code that relies on document.body
} (netscape 4 ed explorer 3 non lo supportano)
```

Il controllo sulle funzioni dipende da quello che ciascuno deve fare, non ha senso controllarle tutte.

per esempio per le funzioni del DOM

```
var W3CDOM = document.createElement &&
document.getElementsByTagName;
function
exampleFunctionThatRequiresW3CMethods() {
if(!W3CDOM) return;
// Code using document.createElement()
// and document.getElementsById()
}
```

La funzione principe del DOM

```
function $(id) {
    return document.getElementById(id);
}
```

In questo caso il proplema principale è quello del namespace. La funzione $ che ho definito potrebbe entrare in conflitto con altre. Prototype usa la stessa notazione.

Trick: una funzione anonima che si autoesegue:

```
(function(){
//your code
})();
```

Le prime due parentesi restituiscono la funzione allo script e le successive parentesi vuote la eseguono. Così ogni funzione è nello scope della anonymous function e non possono essere chiamate dall'esterno del namespace

```
(function(arg){
    alert(arg);
})('This will show in the alert box');
```

quindi:

```
(function(){
function $(id) {
return document.getElementById(id);
}
```

```
function alertNodeName(id) {
    alert($(id).nodeName);
}
window['myNamespace'] = {};
window['myNamespace']['showNodeName']
=       alertNodeName;
})();
```

tutto avviene all'interno della funzione, ne suo scope, dunque se si chiama il metodo $, si sta chiamando il metodo definito in tale funzione

window['myNamespace']['showNodeName']

così invece dò la possibilità di eseguire la funzione anche all'esterno, da altri script, ma sempre nell'ambito del mio namespace

il segreto di javascript e di ogni linguaggio di programmazione ad oggetti, è di poter definire appunto degli oggetti che racchiudono delle funzionalità senza dover ogni volta riscrivere il codice

```
(function(){
//The Mine namespace
if(!window.Mine) { window['Mine'] = {} }

function isCompatible(other) { };
window['Mine']['isCompatible'] = isCompatible;

function $() { };
window['Mine']['$'] = $;

function addEvent(node, type, listener) { };
window['ADS']['addEvent'] = addEvent;
```

```
function removeEvent(node, type, listener) { };
window['Mine']['removeEvent'] = removeEvent;

function getElementsByClassName(className, tag,
parent){ };

window['Mine']['getElementsByClassName'] =
getElementsByClassName;

function toggleDisplay(node,value) { };
window['Mine']['toggleDisplay'] = toggleDisplay;
```

```
function insertAfter(node, referenceNode) { };
window['Mine']['insertAfter'] = insertAfter;

function removeChildren(parent) { };
window['Mine']['removeChildren'] = removeChildren;

function prependChild(parent, newChild) { };
window['Mine']['prependChild'] = prependChild;
})();
```

Vedi 1_start/html

```
function isCompatible(other) {
// Use capability detection to check requirements
if( other===false
|| !Array.prototype.push
|| !Object.hasOwnProperty
|| !document.createElement
|| !document.getElementsByTagName
) {
return false;
}
return true;
}
window['Mine']['isCompatible'] = isCompatible;
```

```
if(Mine.isCompatible()) {
// Codice che usa Mine library
}



function $() {
var elements = new Array();
// Find all the elements supplied as arguments
for (var i = 0; i < arguments.length; i++) {
var element = arguments[i];
// If the argument is a string assume it's an id
```

```
if (typeof element == 'string') {
element = document.getElementById(element);
}
// If only one argument was supplied, return the
// element immediately
if (arguments.length == 1) {
return element;
}
// Otherwise add it to the array
elements.push(element);
}
// Return the array of multiple requested elements
```

```
return elements;
};
window['Mine']['$'] = $;



function addEvent( node, type, listener ) {
// Check compatibility using the earlier method
// to ensure graceful degradation
if(!isCompatible()) { return false }
if(!(node = $(node))) { return false; }
if (node.addEventListener) {
```

```
// W3C method
node.addEventListener( type, listener, false );
return true;
} else if(node.attachEvent) {
// MSIE method
node['e'+type+listener] = listener;
node[type+listener] = function(){
node['e'+type+listener](window.event);
}
node.attachEvent( 'on'+type, node[type+listener] );
return true;
}
```

```
// Didn't have either so return false
return false;
};
window['Mine']['addEvent'] = addEvent;



function removeEvent(node, type, listener ) {
if(!(node = $(node))) { return false; }
if (node.removeEventListener) {
node.removeEventListener( type, listener, false );
return true;
```

```
} else if (node.detachEvent) {
// MSIE method
node.detachEvent('on'+type, node[type+listener]);
node[type+listener] = null;
return true;
}
// Didn't have either so return false
return false;
};
window['Mine']['removeEvent'] = removeEvent;
```

http://shark.emileaben.com/cgi-bin/regex.pl

```
function getElementsByClassName(className, tag,
parent){
parent = parent || document;
if(!(parent = $(parent))) { return false; }
// Locate all the matching tags
var allTags = (tag == "*" && parent.all) ? parent.all :
parent.getElementsByTagName(tag);
var matchingElements = new Array();
// Create a regular expression to
// determine if the className is correct
```

```
className = className.replace(/\-/g, "\\-");
var regex = new RegExp("(^|\\s)" + className +
"(\\s|$)");
var element;
// Check each element
for(var i=0; i<allTags.length; i++){
element = allTags[i];
if(regex.test(element.className)){
matchingElements.push(element);
}
}
// Return any matching elements
return matchingElements;
};
```

```
}
}
// Return any matching elements
return matchingElements;
};
window['Mine']['getElementsByClassName'] =
getElementsByClassName;


function toggleDisplay(node, value) {
if(!(node = $(node))) { return false; }
if ( node.style.display != 'none' ) {
node.style.display = 'none';
} else {
```

```
node.style.display = value || '';
}
return true;
}
window['Mine']['toggleDisplay'] = toggleDisplay;


function insertAfter(node, referenceNode) {
if(!(node = $(node))) return false;
if(!(referenceNode = $(referenceNode))) return
false;
return referenceNode.parentNode.insertBefore(
node, referenceNode.nextSibling
);
```

```
};
window['Mine']['insertAfter'] = insertAfter;


function removeChildren(parent) {
if(!(parent = $(parent))) { return false; }
// While there is a child remove it
while (parent.firstChild) {
parent.firstChild.parentNode.removeChild(parent.
firstChild);
}
// Return the parent again so you can chain the
methods
return parent;
};
```

```
window['Mine']['removeChildren'] = removeChildren;


function prependChild(parent,newChild) {
if(!(parent = $(parent))) { return false; }
if(!(newChild = $(newChild))) { return false; }
if(parent.firstChild) {
// There is already a child so insert before the first
one
parent.insertBefore(newChild,parent.firstChild);
} else {
```

```
// No children so just append
parent.appendChild(newChild);
}
// Return the parent again so you can chain the
methods
return parent;
}
window['Mine']['prependChild'] = prependChild;
```

VEDI DOM TESTING