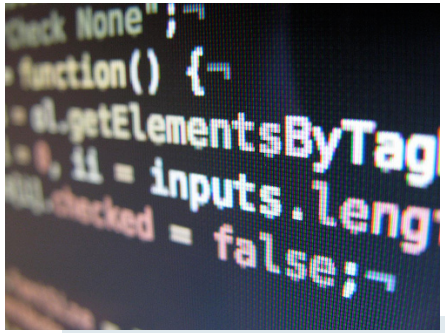



```
<script type="application/javascript>
    document.write("JS programming");
</script>
```

Reti di Calcolatori

Client-side programming & javascript

Client side programming

- Possibilità di eseguire codice sul client (browser)
  - Introdotto in origine come meccanismo per rendere più efficiente la validazione dei dati inseriti in un form
    - Il controllo di validità viene effettuato sul browser prima di inviare i dati allo script server-side
    - Si riducono la latenza nella notifica degli errori all'utente, il carico del server e il traffico di rete
  - E' diventato un supporto indispensabile per lo sviluppo di applicazioni Web 2.0
    - Gli script permettono di introdurre interattività nelle pagine HTML
    - Il linguaggio di script fornisce il supporto per accedere agli elementi della pagina HTML e a caratteristiche del browser
    - Il browser diviene un ambiente di esecuzione di applicazioni che utilizzano la pagina come output grafico e possono interagire con server Web per aggiornare parti di essa in modo dinamico

javascript come standard

- javascript è di fatto diventato uno standard per un linguaggio di script Web lato client
  - Supportato a partire dalla versione 2.0 del browser Netscape Navigator (1995)
  - Il nome javascript fu scelto vista la popolarità crescente del linguaggio Java senza che ci sia una vera relazione fra i due linguaggi
  - Per creare uno standard javascript 1.1 fu sottomesso alla European Computer Manufacturers Association (ECMA) nel 1997
    - Il Technical Committee #39 di ECMA propose uno standard per la sintassi e la semantica di un "general purpose, cross-platform, vendor-neutral scripting language" (ECMAscript)
    - Nel 1998 ECMAscript fu adottato come standard ISO/IEC-16262 (International Electrotechnical Commission) ed è diventato il riferimento per l'implementazione del javascript sui browser

Struttura di javascript

- Le implementazioni di javascript prevedono tre parti
  - ECMAscript
    - specifiche base indipendentemente dall'ambiente di esecuzione (host environment – il browser per la programmazione client-side)
    - Definisce la sintassi, i tipi dato, le istruzioni, le parole chiave del linguaggio, le parole riservate (non usabili per identificatori), gli operatori, gli oggetti predefiniti
    - supporta la codifica Unicode dei caratteri
  - Document Object Model (DOM)
    - supporto per la libreria DOM per la rappresentazione e elaborazione di documenti HTML e XML (supporto XML completo da level 3 DOM)
    - da DOM level 2 è stato introdotto il supporto agli eventi (DOM Events)
  - Browser Object Model (BOM)
    - Supporto per l'accesso alla finestra del browser (non sempre standard)
    - Aprire finestre pop-up, spostare e redimensionare le finestre del browser, gestire la location (indirizzo della pagina), accesso ai cookie

javascript e HTML

- Il codice javascript è incluso in un file HTML col tag **<script>...</script>**
  - fra i tag si può specificare il codice dello script
  - l'attributo **type** specifica il linguaggio di scripting (default javascript)
  - l'attributo **src** permette di specificare un file esterno che contiene il codice javascript

```
<html>
<head>
  <script type="application/javascript"
       src="00-jscode.js" ></script>
</head>
<body>
<script type="application/javascript">
  generatePage();
</script>
</body>
</html>
```


```
function generatePage() {
  document.write("<h2>Pagina generata da javascript</h2>");
}
```

00-jscode.js

Dove mettere il codice javascript..

- In genere è preferibile mettere il codice principale in un file esterno e non inline
  - C'è maggiore sicurezza perché il codice non è direttamente visibile all'interno della pagina
  - E' più semplice la manutenzione perché il codice non è disperso nei file HTML
  - Permette di sfruttare il caching dei browser
    - se due pagine usano lo stesso codice, questo viene caricato solo una volta
- Il codice viene caricato ed eseguito in base alla sua posizione nella pagina HTML
  - Alcuni elementi della pagina possono non essere ancora definiti
  - Di solito si preferisce associare l'esecuzione ad eventi

ECMAScript: sintassi

- Tutto è case-sensitive
- Le variabili non hanno un tipo specifico
  - Come in PHP le variabili assumono il tipo del valore assegnato
  - Possono essere definite con l'operatore var (non necessario)
  - I nomi di variabile possono iniziare con `_` o `$` (non necessario)

```
var nome = "Giovanni", anni = 5
colore = "red"
```

- I commenti sono definiti da `/*.. */` (multilinea) o `//` (singola linea)
- Il `;` alla fine delle istruzioni è opzionale (ma consigliato)
- Le parentesi graffe `{...}` permettono di definire blocchi di istruzioni

Variabili javascript: esempio

```
<h2>Variabili stampate da javascript</h2>
<em>nome</em> è una variabile javascript con valore
<em id="nome"></em> e tipo <em id="tiponome"> </em><br />
<em>anni</em> è una variabile javascript con valore
<em id="anni"> </em> e tipo <em id="tipoanni"> </em><br />
<em>colore</em> è una variabile javascript con valore
<em id="colore"> </em> e tipo <em id="tipocolore"> </em><br />

<script type="application/javascript">
  var nome = "Giovanni", anni = 5;
  colore = "red";

document.getElementById("nome").innerHTML = nome;
document.getElementById("anni").innerHTML = anni;
document.getElementById("colore").innerHTML = colore;
document.getElementById("tiponome").innerHTML = typeof nome;
document.getElementById("tipoanni").innerHTML = typeof anni;
document.getElementById("tipocolore").innerHTML = typeof colore;
</script>
```

Variabili stampate da javascript

nome è una variabile javascript con valore *Giovanni* e tipo *string*
anni è una variabile javascript con valore *5* e tipo *number*
colore è una variabile javascript con valore *red* e tipo *string*

Tipi di variabile

- Le variabili possono essere associate ai seguenti tipi
  - tipi primitivi
    - `undefined`, `null`, `boolean`, `number`, `string`
    - l'operatore `typeof` permette di controllare il tipo di una variabile
    - il tipo `undefined` è associato ad una variabile definita con l'operatore var ma non inizializzata con un valore
    - il tipo `number` permette di rappresentare sia gli interi a 32 bit che i float a 64 bit (può essere anche `Infinity` o `NaN` se si tenta di convertire un non-numero)
    - il tipo `string` è una sequenza di caratteri Unicode
      - le stringhe costanti si possono definire sia con "" che '' con lo stesso significato
  - riferimenti
    - Sono utilizzati per accedere a oggetti

Conversioni di tipo

- I tipi primitivi sono di fatto pseudo-oggetti visto che hanno proprietà e metodi
  - le variabili string hanno la proprietà **`length`**

    `var color = 'red'; - color.length` vale 3
  - tutti i tipi hanno il metodo di conversione in stringa **`toString()`**

    `var anni = 5; - anni.toString(2)` vale `101` (in base 2)
  - Le conversioni da string a number si possono fare con le funzioni **`parseInt(string)`** e **`parseFloat(string)`**
    - Le coversioni vengono fatte analizzando la stringa da sinistra a destra
    - Se non ci sono cifre all'inizio della stringa le funzioni produce `NaN`
    - `parseInt` può avere un secondo parametro per indicare la base (2,8,16)
  - Le conversioni si possono fare anche col casting
    - **`Boolean(v), Number(v), String(v)`**

Riferimenti

- I riferimenti sono usati per accedere ad oggetti
  - ECMAScript non ha meccanismi per la definizione diretta di classi (non ha la keyword class)
  - Un oggetto è un'istanza di una classe
  - Un oggetto di crea con l'operatore **`new`**
    - `var o = new Object();`
    - Se il costruttore non ha parametri le parentesi si possono omettere
  - **`Object`** è la classe base di tutti gli oggetti
    - Tutti i metodi e proprietà presenti in questa classe sono ereditati da tutte le altre
      - proprietà: `constructor`, `prototype`
      - metodi: `hasOwnProperty('name')`, `isPrototypeOf(obj)`, `propertyIsEnumerable('name')`, `toString()`, `valueOf()`

```
var obj = new Object();
objId.innerHTML= obj.toString();
```

obj è un riferimento ad un oggetto *[object Object]*

La classe String

- La classe **`String`** è la rappresentazione a oggetti del tipo primitivo string

```
var objString = new String("Javascript language");
```

  - la proprietà **`length`** fornisce la lunghezza in caratteri (Unicode)
  - ha numerosi metodi utili per la manipolazione di stringhe
    - `charAt(pos)`, `charCodeAt(pos)` – accesso ai caratteri
    - `indexOf(subs)`, `lastIndexOf(subs)` – ricerca di sottostringhe
    - `substring(start,end)` – estrae una sottostringa
    - `toUpperCase()`, `toLowerCase()` – convertono la Stringa in maiuscolo o minuscolo
  - L'operatore + permette di concatenare stringhe (per il tipo primitivo string)
    - `var s1 = new String('java'), s2 = new String('javascript');`
    - `s3 = s1+" e "+s2+ " sono parenti alla lontana!"`

Operatori 1

- Sono in larga parte gli stessi degli altri linguaggi
  - Aritmetici
    - binari: **+**,**–**,*****,**/**,**%**
      - Se uno degli operandi non è `numeric` viene convertito (a parte per **+**)
      - `var v = 10*"5";` - `v` è il numero `50`
    - **+** su stringhe – concatenazione
      - conversione automatica se necessario di variabili non stringa in stringa
      - `var v = 10+"5";` - `v` è la stringa `"105"`
    - Autoincremento/decremento pre o postfisso (**++**,**––**)
  - Operatori bit a bit
    - not (~), and (**&**), or (**|**), xor (^), left shift (**<<**), signed right shift (**>>**), unsigned right shift (**>>>**)
  - Operatori booleani
    - NOT (**!**), AND (**&&**), OR (**||**)

Operatori 2

- Operatori relazionali
  - confronto (<,<=,>,>=)
    - Si applicano sia a numeric che string con eventuale conversione di tipo
  - uguaglianza/disuguaglianza (==,===,!=,!==)
- Operatore condizionale
  - `(cond)? valIfTrue : valIfFalse`
- delete
  - permette di cancellare (rendere undefined) il riferimento ad una proprietà o metodo di un oggetto definita in precedenza
    - `var o = new Object(); o.language = "Java"; delete o.language;`
- void
  - ritorna `undefined` per ogni valore
    - è usato per evitare l'output di un valore
    - `<a href="javascript:void window.open('www.google.com');>google</a>`

Controllo di flusso

- Esecuzione condizionale
  - `if` (*cond*) *true-code* [`else` *false-code*]
  - `switch`(*exp*) { `case` ... `default:` ..}
- Cicli
  - `do` { *loop-code* } `while` (*cond*)
  - `while`(*cond*) { *loop-code* }
  - `for`(*init*; *cond*; *post-loop*) { *loop-code* }
  - `for`(*property* `in` *expression*) { *loop-code* }
    - E' utilizzato per enumerare le proprietà di un oggetto
    - Non tutte le proprietà sono enumerabili
      - si può verificare col metodo propertyIsEnumerable(nome)
  - `break; continue;`

Funzioni

- Le funzioni sono definite con la keyword function

```
function nomeF(arg0, arg1,.....,argN){
  istruzioni;
}
```

  - La funzione può fornire un valore con return val; o nessun valore (non si specifica return o si usa return;)
  - Se si definiscono più funzioni usando lo stesso nome
    - non si generano errori
    - la definizione in un certo punto del codice è l'ultima incontrata
  - L'oggetto arguments permette di accedere agli argomenti passati alla funzione anche se questi non sono elencati
    - arguments[0], arguments[1]...
    - il numero di argomenti è arguments.length
    - Non c'è nessun controllo sul numero di argomenti effettivamente passato

Funzioni come classi

- Le funzioni in ECMAScript sono oggetti

```
var f = new Function(arg1,arg2,...,argN,function-body)
```

  - Il nome della funzione è il riferimento
  - La proprietà `length` indica il numero di parametri
  - I metodi `toString()`/`valueOf()` producono il codice
  - E' però sconsigliato usare questa tecnica per creare funzioni

```
var istr1 = "st=\"\\\"color: yellow; background-color: green\\\"\";";
var istr2 = "e.innerHTML=\"<span style=\"+st+\">Java</span>\";";
var printJava = new Function("e",istr1+istr2);

printJava(s3Id);
```

chiamata alla funzione dopo averla ridefinita come oggetto Function - Java

Oggetti

- Oggetti senza classi...?
  - Un oggetto è una collezione non ordinata di proprietà ciascuna delle quali contiene un valore primitivo, un riferimento ad un oggetto o ad una funzione (metodo)
  - Un oggetto ricorda un array associativo in cui le proprietà sono le chiavi
  - ECMAScript non ha la definizione formale di classe ma introduce il concetto di *object definition* come template con cui creare le istanze di oggetti
    - L'object definition è implementata nella funzione costruttore che aggiunge all'oggetto le proprietà previste dall'interfaccia
    - Sono comunque supportate le funzionalità di un linguaggio ad oggetti
      - Encapsulation (l'oggetto racchiude proprietà e metodi)
      - Aggregation (un oggetto può contenerne un altro)
      - Inheritance (una classe eredita proprietà e metodi da un'altra)
      - Polymorphism (lo stesso metodo può gestire oggetti di classi diverse)

Array

- Un oggetto Array è una collezione ordinata di elementi
  - Si crea col costruttore che può prevedere
    - nessun argomento - `var arr = new Array()`
    - il numero iniziale di elementi – `var arr = new Array(10)`
    - la lista di elementi – `var arr = new Array("a","b","c")`
  - Gli elementi sono accessibili con l'indice numerico `0,...,length-1` con la notazione `v[i]`
    - la proprietà `length` contiene la dimensione attuale del vettore
    - se si assegna un valore ad un elemento oltre la dimensione attuale, l'array viene redimensionato di conseguenza
  - Sono definiti numerosi metodi
    - `toString()`, `join()`, `split()`, `slice()`, `concat()`, `pop()` – preleva dalla fine dell'array, `push()` – inserisce in coda all'array, `shift()` - estrae dalla testa dell'array, `unshift()` - inserisce in testa all'array, `sort()`,...

Factory di oggetti

- Si possono aggiungere dinamicamente proprietà o metodi a un oggetto
  - si assegna un valore alla proprietà o metodo
  - si può usare il riferimento con **`obj.prop`** o come indice di array associativo **`obj["prop"]`**
  - la keyword **`this`** permette di riferirsi alla proprietà dell'oggetto specifico in un metodo
- Si può creare una classe di oggetti con il meccanismo della Factory
  - Si scrive una funzione che genera gli oggetti di una data classe a partire da una classe base
  - Non è però un metodo "pulito"
    - il codice dei metodi è replicato per ogni oggetto

```
function textBoxFactory(txt,clr,bgclr,size) {
  var o = new Object(); // oggetto di classe base

  // proprietà aggiunte
  o.color = clr;
  o.bgcolor = bgclr;
  o.size = size;
  o.text = txt;
  var st = "color: "+clr+";";
  st += "background-color: "+bgclr+";";
  st += "font-size: "+size+";";
  o.style = st;

  // metodi
  o.createBox = function() {
     return "<span style=\""+this.style+"\">"
           +this.text+"</span>";
   }
  return o;
}
```

Costruttori di oggetti...

- Un costruttore è una funzione con il nome della classe
  - Per convenzione si usano nomi con iniziale maiuscola
  - Nel corpo della funzione non si creano oggetti ma si istanziano le proprietà usando `this` come riferimento
  - Il costruttore viene chiamato usando l'operatore `new` per generare un oggetto con le proprietà istanziate
  - Il costruttore può avere zero o più argomenti per inizializzare le proprietà

```
function TextBox(txt,clr,bgclr,size) {
  // definizione delle proprietà
  this.color = clr;
  this.bgcolor = bgclr;
  this.size = size;
  this.text = txt;

  var st = "color: "+clr+";";
  st += "background-color: "+bgclr+";";
  st += "font-size: "+size+";";
  this.style = st;
}
```

```
o1 = new TextBox("Tinky Winky","purple","lightgray",14);
test = o1 instanceof TextBox
```

....e i metodi?

.. la proprietà prototype

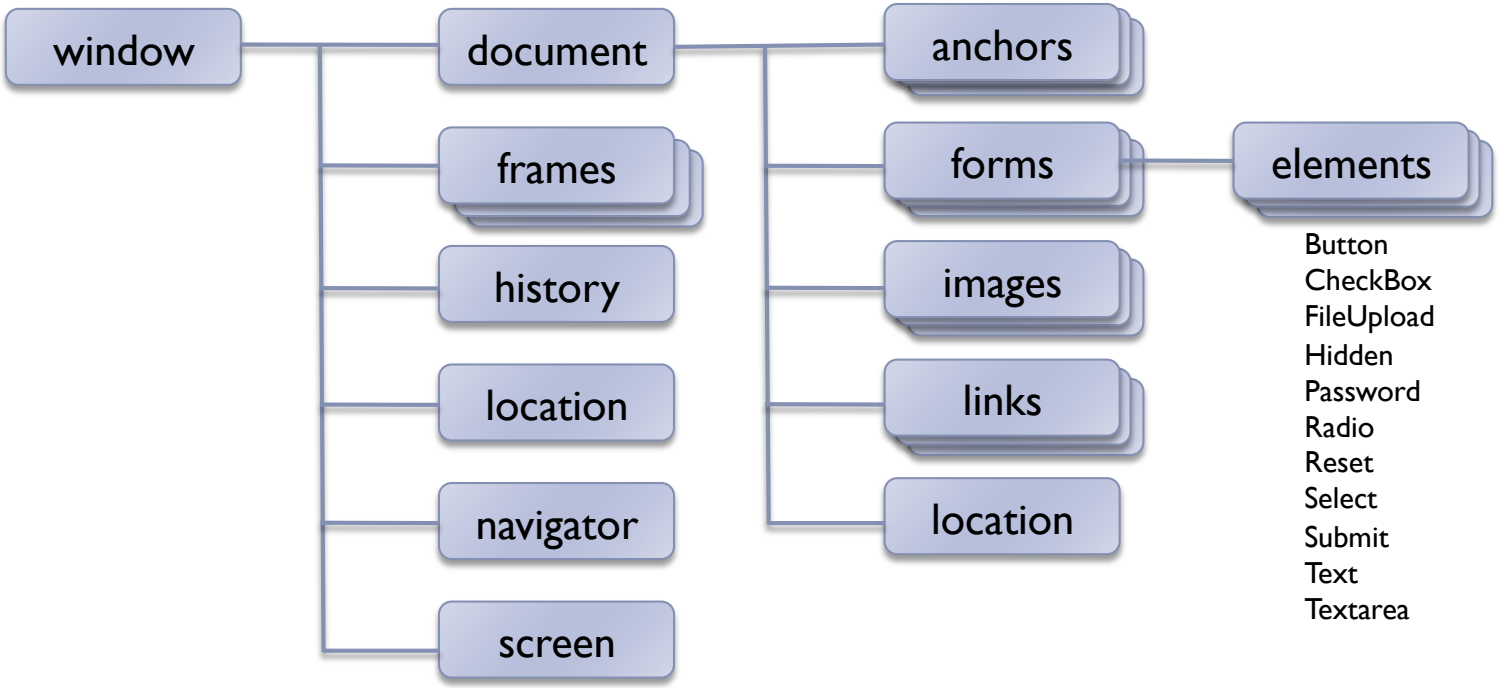
- La proprietà **`prototype`** di un oggetto definisce il template con cui creare nuovi oggetti con `new`
  - E' una lista di proprietà e metodi con il relativo valore
  - Gli elementi di prototype vengono aggiunti agli oggetti creati con `new` inizializzandoli con il valore predefinito
  - Se il valore è un riferimento tutti gli oggetti condividono lo stesso oggetto
  - E' utile per definire i metodi in modo che tutti gli oggetti condividano lo stesso codice

```
TextBox.prototype.createBox = function() {
  return "<span style=\""+this.style+"\">"+this.text+"</span>";
}
```

```
o1 = new TextBox("Tinky Winky","purple","lightgray",14);
o1Id.innerHTML = o1.createBox();
```

Il Browser Object Model

- IL BOM definisce gli oggetti per interagire con la finestra del browser indipendentemente dal contenuto
  - Il BOM contiene una serie di oggetti organizzati gerarchicamente
  - La radice è l'oggetto `window`



window popups & system dialogs

- E' possibile aprire nuove finestre

```
newwin = window.open("url","winName","settings")
```

  - In genere il browser blocca l'apertura di finestre popup chiedendo all'utente un'abilitazione per sito
  - Le settings permettono di definire le caratteristiche della finestra (`height`, `width`, `top`, `left`, `resizable`, `scrollable`, `toolbar`, `status`, `location`)
  - Le finestre popup hanno metodi per la loro manipolazione (`close`, `moveBy`, `resizeBy`, `moveTo`,..)

- Si possono aprire finestre di dialogo
  - `alert("messaggio)`
  - `confirm("conferma")`
  - `prompt("richiesta")`

Intervals & timeout

- Esecuzione temporizzata di codice
  - Timeouts - `ToutId = setTimeout(funct,msec)`
    - Esecuzione di una funzione dopo un numero prefissato di millisecondi
    - Si può annullare prima che scada con `clearTimeout(ToutId)`
  - Interval – `IntId = setInterval(funct,msec)`
    - Esecuzione ripetuta di una funzione ad intervalli di un numero prefissato di millisecondi
    - Si può sospendere con `clearInterval(IntId)`

```
var TIntId;
function startAdRotation() {
if(popupwin) TIntId = setInterval(changeAds,1500);
}

function stopAdRotation() {
  if(TIntId) clearInterval(TIntId);
}
```

```
var ads = new Array("Eat popUP!!",
      "Drink PoPUP!!","Read POPup!!");
var ad;
function changeAds() {
  if(!popupwin.document) return;
  el = popupwin.document.getElementById("adId");
  el.innerHTML=ads[ad++];
  if(ad>ads.length) ad=0;
}
```

Supporto al DOM

- javascript ha il supporto alle API standard DOM
  - Forniscono il modo consigliato per manipolare il documento HTML visualizzato nel browser e accessibile come `window.document`
  - La base è l'oggetto **`Node`** la cui interfaccia prevede (fra gli altri)
    - **`nodeName`** – `String` il nome del nodo
    - **`nodeValue`** – `String` il valore del nodo
    - **`nodeType`** – `Number` il tipo del nodo (es. `Node.ELEMENT_NODE`)
    - **`firstChild`** – `Node` il riferimento al primo figlio
    - **`childNodes`** – `NodeList` la lista di tutti i figli
    - **`hasChildNodes()`** – `Boolean` è `true` se il nodo ha figli
    - **`attributes`** – `NamedNodeMap` è la lista degli attributi
    - **`appendChild(`*`node`*`)`** – `Node` aggiunge *`node`* ai figli in ultima posizione
    - **`removeChild(`*`node`*`)`** – `Node` rimuove il nodo dalla lista dei figli
    - **`insertBefore(`*`node`*`,`*`refnode`*`)`** – `Node` inserisce il nodo nei figli nella posizione precedente il nodo *`refnode`*

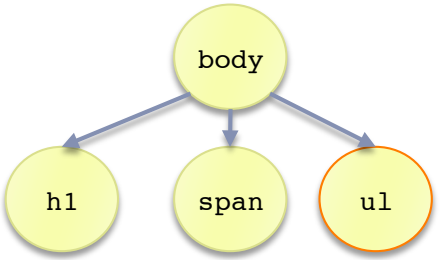
Accesso a nodi specifici

- La libreria prevede metodi per ottenere il riferimento a nodi specifici
  - permettono di ottenere nodi di interesse senza dover effettuare una visita dell'albero DOM
  - `getElementsByTagName("tag")` – `NodeList` fornisce tutti gli elementi (`Element`) col tag specificato
  - `getElementsByName("name")` – `NodeList` fornisce la lista di tutti gli elementi con il valore indicato per l'attributo `name`
  - `getElementById("id")` – `NodeList` fornisce l'elemento con il valore indicato per l'attributo `id`

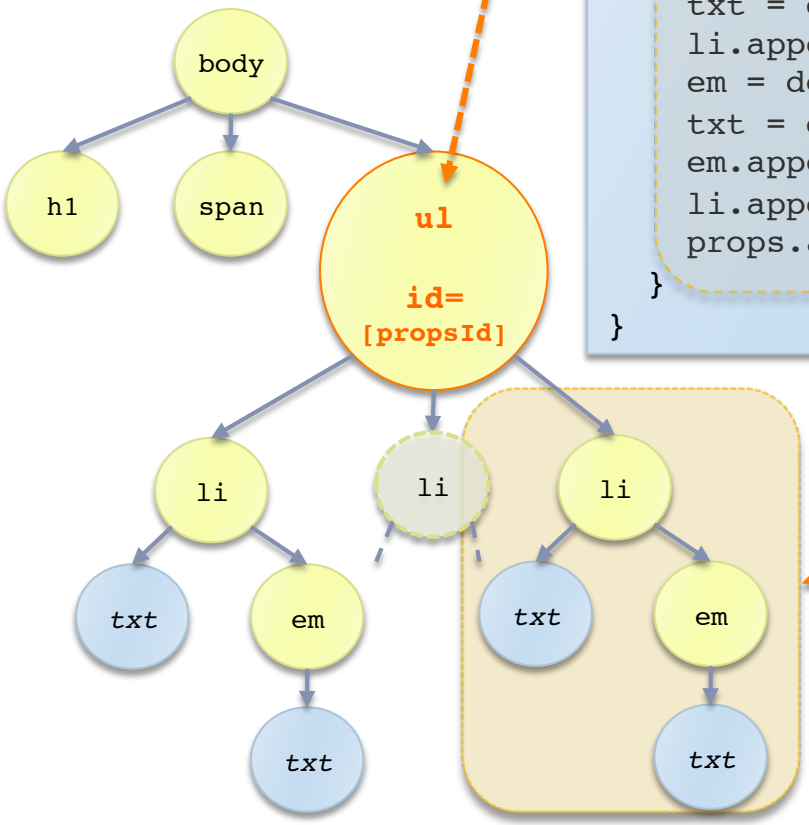
```
props = document.getElementById("propsId");
```

HTML

```
<ul id="propsId"></ul>
```



Esempio di generazione di DOM




```
function propertiesToList(obj) {
  props = document.getElementById("propsId");
  for(p in obj) {
    li = document.createElement("li");
    txt = document.createTextNode(p+": ");
    li.appendChild(txt);
    em = document.createElement("em");
    txt = document.createTextNode(obj[p].toString());
    em.appendChild(txt);
    li.appendChild(em);
    props.appendChild(li);
  }
}
```

Creazione dei nodi
- createElement("tag")
- createTextNode("text content")

HTML DOM

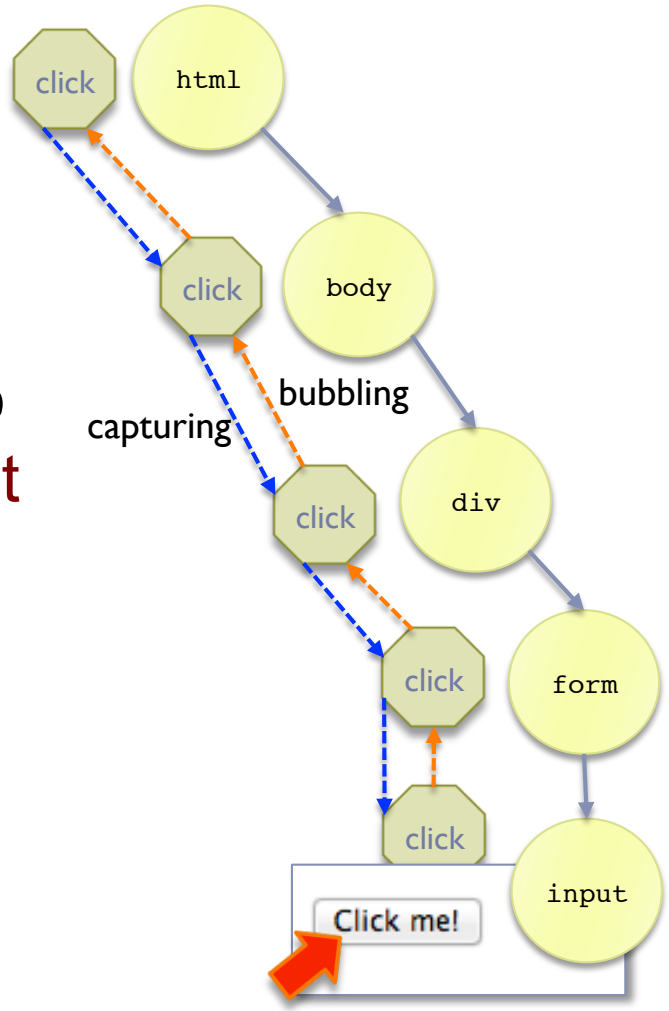
- Il supporto specifico DOM per HTML prevede la semplificazione di alcune operazioni
  - Accesso agli attributi con i metodi del Core DOM
    - `node.getAttribute("style")`
    - `node.setAttribute("style","color: red")`
    - `node.removeAttribute("style")`
  - Accesso agli attributi con HTML DOM – gli attributi sono direttamente proprietà dell'oggetto
    - `st = node.style;`
    - `node.style = "color: red";`
    - `delete node.style;`
  - Gestione degli `Element <table>` con proprietà e metodi
    - `tab.rows`, `tab.deleteRow(pos)`, `tab.insertRow(pos)`,... `tr.cells`,...

Eventi

- In genere l'interazione del codice col documento HTML avviene sulla base del verificarsi di **eventi**
  - Gli eventi sono generati dall'utente o dal browser
    - caricamento della pagina
    - click su un elemento
    - passaggio del cursore del mouse su un elemento
    - fuoco dell'input su un elemento
  - E' possibile associare l'esecuzione di codice al verificarsi di un particolare evento
  - Gli eventi sono associati ad un **event flow**
    - più di un elemento sulla stessa pagina può rispondere allo stesso evento
      - quando si clicca un bottone l'evento è per il bottone, il contenitore in cui si trova (es. una tabella o un form), fino ad arrivare alla pagina stessa

Propagazione e cattura degli eventi

- L'event si propaga nel DOM dall'elemento più in basso in cui è stato generato verso la radice (event bubbling)
- L'event si propaga nel DOM dall'elemento più in alto verso l'elemento più specifico (event capturing)
  - Il DOM supporta entrambe le modalità
  - L'event capturing viene effettuato prima



Event handlers/Listeners

- Una funzione chiamata in risposta ad un evento è detta event handler or event listener
  - una funzione che risponde all'evento click è un *onclick handler*
  - per assegnare un event handler ad un elemento della pagina HTML occorre inserire il nome della funzione nella proprietà corrispondente dell'elemento

```
<div id= "d1" onclick="myClickHandler()">..</div>
```

attributo del tag HTML

```
div1 = document.getElementById("d1");
div1.onclick = myClickHandler;
```

proprietà dell'elemento DOM corrispondente all'evento

```
div1 = document.getElementById("d1");
div1.addEventListener("onclick",myClickHandler,true);
```

metodo DOM per la registrazione di gestori di evento (se ne possono registrare anche più di 1)

Rimozione di un event handler

- Per rimuovere un event handler
  - Si può assegnare null alla proprietà corrispondente
    - `div1.onclick = null;`
  - Si può usare il metodo DOM di rimozione
    - `div1.removeEventListener("onclick",myClickHandler,true);`
    - Nel caso si siano assegnati più event handler allo stesso evento vanno rimossi uno alla volta
  - Se si vuole cambiare l'event handler è sufficiente assegnare alla proprietà corrispondente all'event handler un nuovo valore
    - `div1.onclick = newEventHandler;`

L'oggetto evento

- Le caratteristiche dell'evento sono utili per la gestione
  - L'Event Object contiene le informazioni sull'evento che si è verificato
    - L'oggetto che ha causato l'evento
      - `target` – elemento che ha generato l'evento
    - Informazioni riguardanti il puntatore (mouse) al momento dell'evento
      - `button` – stato dei bottoni del mouse
      - `clientX`, `clientY` – le coordinate del mouse nella finestra
    - Informazioni sulla tastiera al momento dell'evento
      - `charCode` – codice del carattere battuto sulla tastiera
      - `ctrlKey` – indica se il tasto CTRL è premuto
    - Informazioni generali sull'evento
      - `timeStamp` – istante di tempo in cui si è verificato
      - `type` – tipo dell'evento ("click", "mouseover",...)
      - `eventPhase` – (0 capturing phase, 1 - at target, 2 – bubbling phase)
  - L'oggetto è disponibile come parametro dell'event handle

Tipi di evento

- **Mouse events**
  - generati da azioni effettuate dall'utente col mouse
    - click, dblclick, mouseover, mousemove, mouseout, mouseup, mousedown
- **Keyboard events**
  - generati dalla pressione di tasti sulla tastiera
    - keydown, keypress, keyup
- **HTML events**
  - generati quando avvengono cambiamenti nella finestra del browser o si verifica interazione fra client e server
    - load, change (textbox), submit (form), focus,....
- **Mutation events**
  - generati quando ci sono cambiamenti nella struttura del DOM
    - DOMSubtreeModified, DOMNodeInserted,...