Malmö högskola
Centrum för teknikstudier

Webbprogrammering, DA123A, Studiematerial 4

DOM – bakgrund

I webbens ungdom fanns endast möjlighet att visa statisk information i en webbläsare. När Netscape Communications släppte version 2.0 av Netscape Navigator introducerades även JavaScript. En ny värld av möjligheter öppnades då man nu kunde "programmera" webbläsaren till att dynamiskt påverka elementen i ett HTML-dokument. Detta gav upphov till det som skulle bli *DOM - Document Objekt Model*. Dessa första steg till DOM har kommit att kallas DOM Level 0 men är ingen formell standard utan bara ett samlingsnamn för de steg i denna riktning som togs av utvecklarna av webbläsare. W3C gav sig i kast med att samla dessa funktioner till en standard och DOM Level 1 skapades och publicerades 1998. Idag finns DOM Level 3 publicerat som en rekommendation men stödet för denna är inte så väl utvecklat som för Level 1 och 2 hos alla webbläsare.

Från början var det bara formulärelementen som kunde påverkas, men det utvecklades snabbt till att även inkludera de övriga elementen. Åren gick och mer och mer funktionalitet introducerades. Olyckligtvis hade nu Microsoft och Netscape valt olika vägar (som vanligt...). Sättet att "programmera" var inte längre likadant. Det gick till och med så långt att Microsoft introducerade sin egen variant av JavaScript, kallad JScript. Än i dag finns det markanta skillnader i hur de olika webbläsarna är uppbyggda och fungerar. Så som skapare av dynamiska webbsidor så bidrar ni till att öka möjligheterna till mer likriktade webbläsare genom att själva följa standarderna från W3C.

För att se hur de olika utvecklarna har implementerat DOM så hänvisas till följande webbsidor:
http://msdn.microsoft.com/en-us/library/ms533050(VS.85).aspx
http://developer.mozilla.org/en/docs/Gecko_DOM_Reference

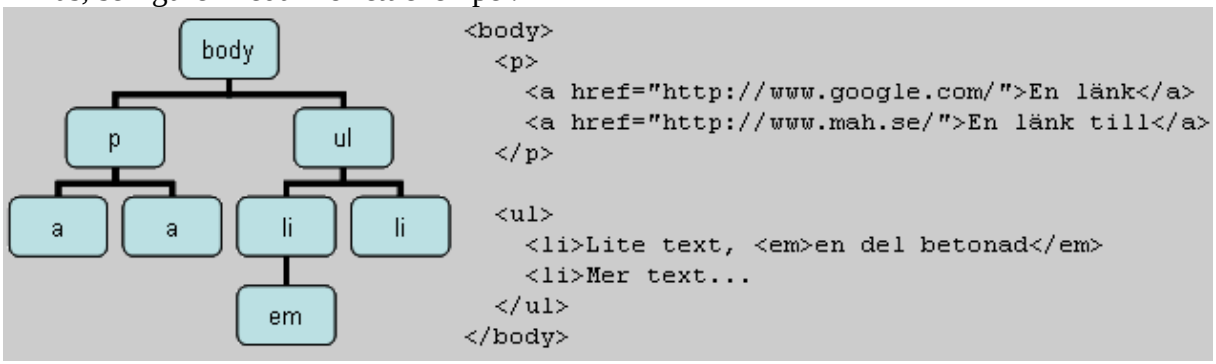
W3Cs standard hittar du här:
http://www.w3.org/DOM/DOMTR

Du bör ha dessa sidor i åtanke och ta för vana att konsultera dem när du är osäker på något som faller under DOM eller när du stöter på problem.

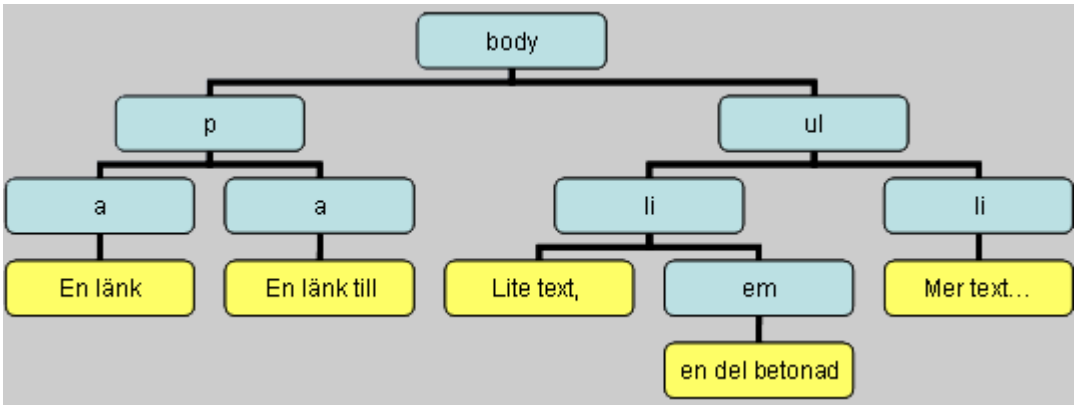
DOM – introduktion

DOM är en objektstruktur från vilken man kan få åtkomst till alla element i ett HTML-dokument (även XML och XHTML). DOM är till strukturen ett träd av noder, där varje nod representerar ett element. Alla noder har en *förälder* (utom högsta nivån) men kan ha flera *barn*.

Exempel: Ett stycke (<p>) ligger i dokumentet (<body>) och i stycket kan flera länkar (<a>) finnas, se figuren nedan för ett exempel.



Bilden ovan är inte riktigt korrekt, utöver noderna för de olika elementen (element-noder) så lagras även all text i så kallade *textnoder*. En mer korrekt bild av samma "kodsnutt" ser ut enligt nästa figur.

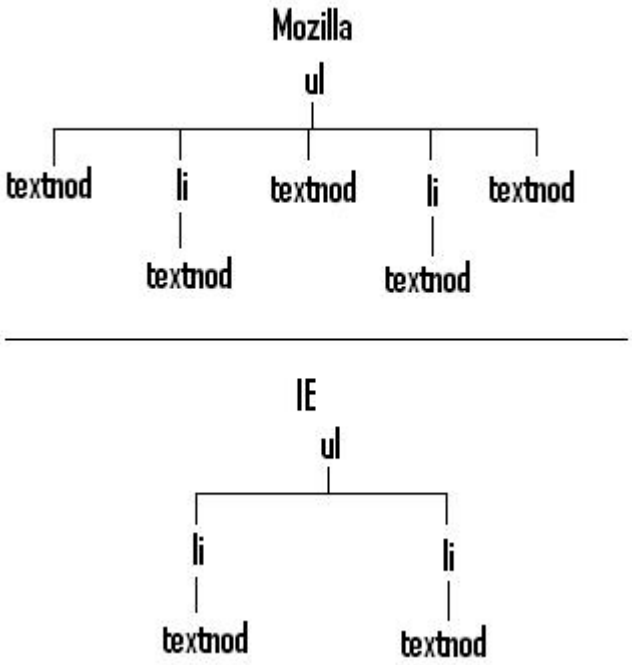


Webbläsaren håller på detta sätt reda på alla element i dokumentet. Som du förstår så kan det bli väldigt många element att hålla reda på. Det blir nu kanske lite lättare att förstå varför det är så viktigt att skriva korrekt HTML.

Bilden ovan är kanske inte heller riktigt korrekt då olika webbläsare har implementerat vad som räknas som en textnod olika. Enligt definitionen från W3C så räknas varje whitespace som en textnod. Detta innebär att det blir en extra textnod när man indenterar sin kod ordentligt. Mozillas implementation använder denna modell medan IE döljer textnoder med endast whitespace. Så om vi har ett kodstycke med följande utseende:

```
<ul>
  <li> listelement 1</li>
  <li> listelement 2</li>
</ul>
```

Så får vi följande träd:



Det kan med andra ord vara ganska vanskligt att förlita sig på att röra sig i trädet för att hitta en viss nod. Om man gör detta så måste man kontrollera nodernas typ så man hamnar rätt om det finns en risk att olika webbläsare skapar olika träd av kodstycket.

Även attribut hos elementen kan hanteras som noder.

DHTML - Dynamic HTML

Det är viktigt att påpeka att DHTML inte är ytterligare en specifikation från W3C. DHTML är ett populistiskt namn på hur man kan kombinera JavaScript, HTML och CSS för att skapa dynamiska webbsidor.

Vi skall nu se hur dynamiska webbsidor byggs upp genom att använda JavaScript och DOM så som det rekommenderas från W3C.

I koden nedan visas ett utdrag ur ett HTML-dokument. Detta kodstycke kommer från och med nu att användas för alla exempel i detta studiematerial.

```
<div id="dyn_content">
  <p id="dyn_paragraph">
    Stycketext.
  </p>
  <ul id="dyn_list">
    <li><a href="http://www.mah.se" id="link1">Webbplats för MAH</a></li>
    <li><a href="http://www.google.com" id="link2">Sökmotorn
Google</a></li>
  </ul>
</div>
```

Med bara en objektstruktur hade DOM inte betytt så mycket för oss som webbutvecklare, men eftersom DOM även har en mängd funktioner för att dynamiskt påverka alla element så har vi genast stor nytta av den.

De två viktigaste funktionerna i DOM är *getElementById()* och *getElementsByTagName()*. Med dessa kan man få åtkomst till alla element i ett dokument, för att sedan göra de dynamiska ändringar man vill. De två funktionerna finns definierade för dokumentobjektet i varje HTML-dokument, window.document eller bara document (window-objektet ingår inte i DOM men mer om detta senare). Som synes så är den sistnämnda funktionen i plural (getElements...) vilket antyder resultatet av den är flera element. Så är också fallet, getElementById("identitetsnamn") ger en referens till ett enskilt element med identiteten *identitetsnamn*, medan getElementsByTagName("tagg") ger en array med referenser till alla element i dokumentet av typen *tagg*. Observera att för att använda getElementById så gäller att du följer reglerna för elements id som säger att ett id skall vara unikt. Det vill säga du kan inte ha två element med samma id.

Koden nedan visar exempel på funktionerna getElementById och getElementsByTagName.

```
// En referens till ett enskilt element med en identitet
var link_to_mah = document.getElementById("link1");

// En array med referenser till alla länkar i dokumentet
var link_array = document.getElementsByTagName("a");

// Enskilda referenser kan sedan hämtas ur arrayen
var another_link_to_mah = link_array[0];
var link_to_google = link_array[1];
```

Nu har vi ett sätt att referera till alla element i ett dokument, och med dessa referenser kan elementen påverkas dynamiskt.

Innan vi börjar ändra på saker och ting behöver vi dock en sak till. Eftersom innehållet i elementen, till exempel texten i ett stycke, sparades i speciella *textnoder* måste vi även kunna referera till dessa. Detta gör vi med arrayen childNodes som är definierad för alla noder, men är tom om elementet i fråga inte har något *barn*. En nod utan *barn* kan till exempel vara ett tomt stycke. Alla *textnoder* är även barnlösa, då de alltid ligger längst ut i en gren i objektstrukturen.

```
// skapa en refererens till texnoden hos MAH-länken om det finns en
if ( link_to_mah.childNodes.length != 0 )
{
   var textnode = link_to_mah.childNodes[0];
}
```

Funktioner i DOM

DOM tillhandahåller funktioner för att bland annat göra följande saker:

- Ändra attributen hos elementen
- Ändra innehåll i elementen
- Ändra formatmallen för varje element
- Skapa nya element
- Ta bort element
- Flytta ett element till annan plats i dokumentet

Ändra attributen hos ett element

Alla attribut som kan andvändas i HTML kan ändras dynamiskt via DHTML. För att göras detta skapar man först en referens enligt tidigare till elementet i fråga. Men denna referens kan sedan alla attribut ändras med punktnotation, se exempel nedan.

```
// Ändra attributet href i länken
link_to_mah.href = "http://www.google.com";
```

Att ändra attributen hos elementen är lätt efter att man erhållit en referens till dem.

Observera att olika element har olika uppsättning attribut, och det är elementets typ som avgör vilka attribut som finns tillgängliga via DOM. Attributen heter samma sak i DOM som i vanlig HTML, med ett undantag. Eftersom class är ett reserverat ord i JavaScript, heter detta attribut istället className. Ovanstående exempel ändrar alltså länken till att peka på adressen http://www.google.com, inte vad som står i länken. För att ändra texten (innehållet) i länkens måste vi ändra i länkens *textnod*.

Ändra innehållet hos element

För att ändra innehållet hos ett element behöver vi ändra i elementets *textnod*. Som vi tidigare såg kunde vi komma åt ett elements *barn* men arrayen childNodes. Det är med hjälp av denna array som vi sedan kan komma åt "värdet" av *textnoden* hos ett element. Genom att först skaffa sig en referens till *textnoden*, kan vi sedan ändra dess värde med egenskapen nodeValue.

```
// Ändra innehållet i länken
var textnode = link_to_mah.childNodes[0];
textnode.nodeValue = "Länk till Google";
```

En annan egenskap som också finns definierad för varje element som har *barn* är firstChild. firstChild refererar alltid till det första *barnet* hos ett element. Denna egenskap kan användas för att ändra koden i ett barn utan att ha en direkt referens till barnet. I exemplet nedan sker samma sak som i det tidigare exemplet men med något kortare kod.

```
// Ändra innehållet i länken
link_to_mah.firstChild.nodeValue = "Länk till Google";
```

Tänk dock på att om du inte vet att det alltid finns ett barn så bör du först kontrollera så referensen har ett värde.

Kontrollera nodens typ

Noderna som man kommer åt med funktioner som exempelvis firstChild kan vara av olika typ. Detta beror på om det är en textnod, ett element eller en kommentar som objektet man har tag i representerar. Det finns ett flertal varianter av noder. Du kan hitta en lista över typerna på: http://www.w3.org/TR/DOM-Level-2-Core/core.html#ID-1950641247

Beroende på nodens typ så finns olika möjliga attribut att använda. Om det finns en osäkerhet om vilken typ av nod det kan vara så kan man testa detta genom att använda sig av attributet nodeType. Detta värde är et tal som representerar en viss typ av nod.

Tyvärr så implementerar inte IE konstanterna för nodtyper så ska man använda detta i IE så måste man skriva in talen direkt istället för konstanterna.

Som visat tidigare så skiljer det sig också mellan webbläsare hur de räknar antalet noder. Så fort det finns ett blanktecken i HTML-filen så räknar Mozillas implementation detta som en textnod. I IE så är implementationen sådan att endast blanka tecken inte räknas som en textnod. Så man bör alltid lägga in kontroll av att en nod är det man förväntar sig.

Ändra formatmallen för ett element

Det som förmodligen har gjort DHTML mest populärt är möjligheten att ändra formatmallen hos elementen. Vi kan då skapa webbsidor som dynamiskt ändrar färg och form. Den vanligaste användningen av detta är dynamiska länkar och menyer som många webbsidor har. På samma sätt som för attributen så finns alla formategenskaper i CSS definierade för elementen i DOM. Alla formategenskaperna finns tillgängliga via varje nods egenskap style Det som skiljer DOM och CSS åt är att de CSS-egenskaper som har bindestreck i namnet, till exempel font-family, istället skrivs som fontFamily i DOM. Alla bindestreck försvinner alltså och i stället dras namnet ihop och efterföljande orddel startar med en versal.

```
// Ändra formatmallen för ett element
link_to_mah.style.fontFamily = "Impact";
link_to_mah.style.fontWeight = "bold";
```

Skapa nya element

Det som gör DHTML med DOM mest kraftfullt är att vi kan skapa helt nya element, och lägga till dessa var vi vill i dokumentet. En sak som vi måste tänka på är att det inte räcker med att skapa till exempel en ny länk. Det är ju fortfarande så att innehållet i en länk ligger i en egen *textnod*, alltså måste även en textnod skapas. Till vår hjälp har vi två funktioner som båda finns definierade för dokumentobjektet, *document.createElement("elementtyp")* och *document.createTextNode("Texten i textnoden")*. För att lägga till det nyskapade elementet till documentet behöver vi sedan funktionen *appendChild(nodreferens)*. Denna funktion finns definierad för varje nod och det nyskapade elementet läggs till sist i denna. Men innan vi lägger till elementet i dokumentet lägger vi till *textnoden* till elementet, även detta med appendChild().

```
// Skapa ett nytt element och en tillhörande textnod
var link_to_webzone = document.createElement("a");
link_to_webzone.href = "http://webzone.ts.mah.se";
var link_to_webzone_textnode = document.createTextNode("Länk till
Webzone");

// Lägg till textnoden till elementet
link_to_webzone.appendChild(link_to_webzone_textnode);

//Lägg till elementet sist i stycket
document.getElementById("dyn_paragraph").appendChild(link_to_webzone);
```

Nu är det kanske på sin plats att ta en titt på hur vårt utdrag ur HTML-dokumentet ser ut efter skapande av vårt nya element.

```
<div id="dyn_content">
  <p id="dyn_paragraph">
    Stycketext.<a href="http://webzone.ts.mah.se">Länk tilll Webzone</a>
  </p>
  <ul id="dyn_list">
    <li><a href="http://www.mah.se" id="link1">Webbplats för MAH</a></li>
    <li><a href="http://www.google.com" id="link2">Sökmotorn
Google</a></li>
  </ul>
</div>
```

För att titta på ett dokuments DOM så kan du använda verktyget DOM Inspector i Firefox som du hittar under Tools-menyn.

Ta bort ett element
Element kan tas bort genom att använda funktionen removeChild(nodreferens) hos *föräldern* till den nod som skall tas bort. I exemplet nedan använder vi dessutom en egenskap som alla noder har (utom "toppnoden") och som ger en referens till dess *förälder*, parentNode.

```
// Ta bort ett element
link_to_webzone.parentNode.removeChild(link_to_webzone);
```

Efter borttagning av elementet enligt exemplet ovan kommer vår "kodsnutt" åter att se ut som den gjorde tidigare.

Flytta ett element till annan plats i dokumentet
Vi såg tidigare hur vi kunde använda funktionen appendChild() för att skapa ett nytt element. Samma funktion kan vi använda för att flytta omkring element. Om vi använder funktionen för ett element som redan finns i dokumentet så kommer det att flyttas från där det fanns innan funktionen anropades, till sist i den nod som vi nu anropade funktionen hos.

```
// Flytta ett element
document.getElementById("dyn_paragraph").appendChild(link_to_mah);
```

Efter flytten kommer vår kodsnutt att se ut enligt nedan.

```
<div id="dyn_content">
  <p id="dyn_paragraph">
    Stycketext.<a href="http://www.mah.se" id="link1">Webbplats för
MAH</a>
  </p>
  <ul id="dyn_list">
    <li></li>
    <li><a href="http://www.google.com" id="link2">Sökmotorn
Google</a></li>
  </ul>
</div>
```

Detta var en introduktion till DHTML med JavaScript och DOM. Detta är på detta sätt som W3C rekommenderar en att använda DHTML. De webbläsare som idag stödjer DOM bäst är Gecko-baserade webbläsare. Internet Explorer stödjer även de flesta funktioner, men det är som alltid viktigt att testa i flera olika webbläsare för att vara säker på att allt fungerar eller ser ut som man tänkt sig.

Skilj mellan DOM och JavaScript
Kom ihåg att DOM i sig själv inte är ett språk utan ett interface för hur man skall hantera HTML-koden som du kan använda då detta interface finns implementerat i exempelvis JavaScript.

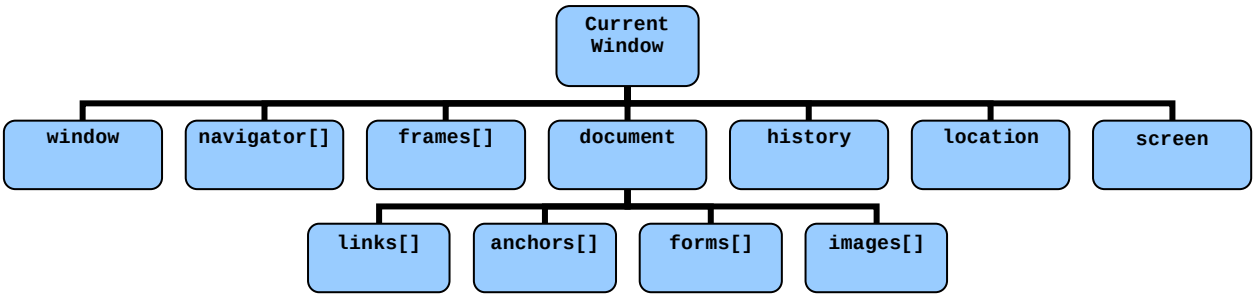
I W3Cs DOM så ingår inte exempelvis window-objektet utan detta är en del av JavaScripts modell för webbläsaren. Det spelar sällan någon direkt roll var implementeringen av DOM börjar i JavaScript men det är bra att ha i åtanke att alla funktioner och objekt i JavaScript inte kommer från DOM.

Ett par länkar som du bör läsa för att förstå mer av skillnaderna mellan DOM och JavaScript:
http://developer.mozilla.org/en/docs/The_DOM_and_JavaScript
https://developer.mozilla.org/en/Gecko_DOM_Reference/Introduction

Observera att DOM ibland kan betyda ett visst språks DOM som man implementerat och ibland avses DOM enligt W3C. I den andra länken ovan så skiljer man inte alltid tydligt på vad som är JavaScript DOM och vad som är W3C DOM. Level 0 DOM är inte W3C även om man ibland har tagit hänsyn till dessa första metoder och införlivat dem i DOM Level 1.

JavaScripts modell av webbläsaren

Då den har stor betydelse för JavaScript så kommer vi här att gå in på de delar av JavaScript som ofta tas för en del av DOM men som faktiskt inte är det. Dessa objekt används för att representera webbläsaren i JavaScript. Nedan syns en bild över de vanligaste av dessa objekt. De olika rutorna representerar JavaScripts olika fördefinierade objekt i webbläsaren.



De olika rutorna representerar JavaScripts olika fördefinierade objekt i webbläsaren. Via dessa objekt kan olika information hämtas och olika funktioner åstadkommas:

- **window** ger möjlighet till att skapa nya fönster och tillgång till funktioner i nuvarande fönster
- **navigator** ger information om användares webbläsare och miljö
- **frames** ger tillgång till olika ramar (om de finns)
- **document** ger tillgång till HTML-elementen i webbläsaren
  (DOM är implementerat här)
- **history** ger information om tidigare besökta webbsidor
- **location** ger information om nuvarande webbplats och möjlighet till att skicka användaren vidare till annan plats
- **screen** ger information om användarens bildskärmsegenskaper
  (upplösning, färgdjup etc.)

Window-objektet

Window-objektet refererar till ditt fönster i webbläsaren. Detta är ett underförstått objekt vilket gör att du inte behöver skriva window.document.funktionsnamn utan det räcker med att du anger document.funktionsnamn.

Några metoder för detta objekt:

- focus: fönstret får fokus (aktiveras)
- blur: tappar fokus
- scroll(x, y): flyttar rullningslisten i x- och y-led (pixlar)
- open(URL, namn, egenskapssträng)

Observera att ramar hanteras som egna fönster. Detta kommer dock inte att bli något problem för denna kurs då ramar inte är tillåtna i HTML 4.01 Strict.

För att öppna ett nytt fönster skriver man

myWindow = open(URL, namn, **egenskapssträng**);

Exempel på egenskaper som kan sättas i **egenskapssträngen:**

- height höjd
- widht bredd (anges m.h.a antal pixlar)
- toolbar verktygsraden
- location URL-fönstret
- status statusraden
- menubar menyraden
- scrollbars scrollningslister
- resizable tillåt storleksändring

De egenskaper man önskar anges genom egenskapsnamn=yes. För exempel gå till länken: https://developer.mozilla.org/en/DOM/window.open
Här får du även en bra lisa över vilka egenskaper som är kompatibla med vilken webbläsare.

För att stänga ett fönster du öppnat med open så använder du close. En eventuell referens till fönstret bör sättas till null för att nollställa denna och ta bort fönstret från minnet efter att fönstret stängts.

```
//Global var to store a reference to the opened window
var openedWindow;
function openWindow()
{
  openedWindow = window.open('moreinfo.htm');
}
function closeOpenedWindow()
{
  openedWindow.close();
   openedWindow = null;
}
```

För att stänga det aktuella fönstret så använder du referensen window direkt:

```
function closeCurrentWindow()
{
  window.close();
}
```

Null-värdet är INTE samma som noll. Ett null-värde talar om att variabelns värde är odefinierat (detta går att testa på).

Navigator-objektet
Detta objekt refererar till webbläsaren och dess egenskaper. Du hittar dokumentationen under window.navigator men objektet går precis som document att använda utan window-referensen. Navigator-objektet innehåller egenskaper så som exempelvis namn och version av webbläsaren som använts för att öppna webbsidan. Exemplet nedan visar på användning av detta.

```
alert("Ni har besökt denna hemsida med:
\n"+navigator.appName+"\n"+navigator.appVersion);
```

Man kan använda de angivna egenskaperna för att ta reda på vilken webbläsare surfaren ifråga använder. Till exempel om man vill använda icke standardiserade specialtaggar för respektive webbläsare.

History-objektet
History-objektet kan du tänka på som motsvarande pilarna framåt och bakåt i webbläsaren. Detta är en array med besökta URL. Metoderna är back() och forward() är självförklarande. Metoden go() kan som parameter ta emot en relativ position i besökshistoriken genom att ange -1, 0 och 1 vilket i princip motsvarar back och forward. Metoden go() kan också ta emot en sträng som anger URLen som skall besökas vilket är mer användbart.

Location-objektet
Location-objektet liknar metoden go() i history-objektet men är mer kraftfull. Du har här ett antal egenskaper (exempel utgår från URLen:
http://www.google.com:80/search?q=devmo#test):

- hash – det i URLen som följer efter #: #test
- host - host namn och portnummer: www.google.com:80
- hostname - host namn utan portnummer: www.google.com
- href - komplett URL: http://www.google.com:80/search?q=devmo#test
- pathname - sökväg relativt hosten: /search
- port – URLens portnummer: 80
- protocol – URLens protokol: http:
- search – den del av URLen som följer frågetecknet: ?q=devmo

window.location sätts på följande sätt:
window.location.href = ”http://www.ts.mah.se”;

Använd alltid location-objektets href-attribut för att ladda en ny sida. Jämför med Document-objektets Document.URL Document.URL kan läsas, men den kan inte sättas.

Screen-objektet
Screenobjektet returnerar information om skärmen dimensioner och färgdjup. Exempel på egenskaper för screen-ojektet:

- screen.availHeight - returnerar skärmhöjd i pixels (se anteckningar)
- screen.availWidth - returnerar skärmbredd i pixels (se anteckningar)
- screen.colorDepth - om en palett används returneras bitdjupet annars ges screen.pixelDepth
- screen.height - returnerar höjden av en skärmen i pixels
- screen.pixelDepth - returnerar höjden av skärmen i pixels
- screen.width - returnerar bredden av skärmen i pixels

Exempel:

```
<script type="text/javascript">
    function myFullScreen() {
      window.open('http://www.mah.se','myName',
        'width='+screen.width+',height='+screen.height+',top=0,left=0');
    }
</script>
```

Fler objekt
JavaScripts DOM innehåller så mycket mer än det som är rimligt att ta upp här. Du bör därför gå till dokumentationen hos Mozilla och MSDN och surfa runt i denna för att lära dig hitta där samt för att skaffa dig en större överblick över de möjligheter du har genom att använda JavaScript. När du surfar runt i dokumentationen hos Mozilla så lägg märke till de noteringar som finns längst ner för olika metoder eller egenskaper under Specification. Här anges varifrån något kommer exempelvis DOM Level 0 eller DOM Level 1.

Debugging
Den kod man skrivit innehåller ibland fel som kan vara mer eller mindre svåra att hitta. Man kan dela in de fel som uppstår i tre grupper:

- Syntaxfel
- Körfel
- Logiska fel

Det finns flera verktyg at använda för debugging av JavaScript, exempelvis Venkman eller Firebug. Det enklaste verktyget är nog Error Console i Firefox. Denna ger inte så utförlig information men går snabbt att använda. Om man inte hittar felet utifrån informationen som Error Console ger så kan man ta till något av de mer kraftfulla verktygen.

Ibland har man kvarglömda fel i sin kod som inte direkt påverkar funktionaliteten. Ta därför som vana att alltid använda Error Console för att försäkra dig om att inte fel uppstod även om koden verkade köra korrekt.

Syntaxfel
Syntaxfel är fel som uppstår på grund av att glömt något i koden som måste skrivas ut, exempelvis att det saknas en avslutande högerparentes eller en glömd punkt i en punktnotation, eller att det finns något tecken för mycket som inte borde finnas där.

Syntaxfel är lätta att fixa när man väl hittar dem. Ibland går det snabbt att hitta problemet och man behöver bara lägga till eller ta bort ett tecken på exakt den angivna raden. Ibland så blir man dock "hemmablind" när det gäller kod man skrivit själv. Man kan sitta och stirra på ett stycke kod och kan inte alls se vad det är som är fel. Om man ber någon annan titta på koden så ser de genast vad som är fel.

Genom att använda en editor med bra highlighter och färgkodning av matchande parenteser så kan man undvika en del syntaxfel. Om man har ett koncist sätt att skriva sin kod, det vill säga att man alltid gör indrag, utformar loopar och funktioner, med mera på samma sätt, så är det lättare att se när något avviker från denna mall och det blir lättare att hitta syntaxfel.

Nackdelen med syntaxfel är att om man har många av dem i ett stycke kod så kan det ta tid att hitta dem. När webbläsaren stöter på ett syntaxfel så vet den helt enkelt inte hur den ska tolka koden och avbryter exekveringen och ger ett felmeddelande om på vilken rad felet uppstod. Så webbläsaren kan bara hitta ett syntaxfel åt gången. Om det finns ytterligare ett syntaxfel precis efter det första så får du inte reda på detta på en gång utan först när det första syntaxfelet är åtgärdat.

Ibland blir felmeddelandena inte riktigt bra. Dessa anger nämligen var ett fel uppstod i exekveringen. Det egentliga felet kan vara något annat. Ofta finns då felet någon rad längre upp i koden, ofta precis innan det rapporterade felmeddelandet. Så syntaxfelet uppstår inte där

felet är i sig utan där webbläsaren förväntade sig något annat än vad som står. Felmeddelandet talar då om vad webbläsaren förväntade sig - inte vad som egentligen är problemet.

Körfel
Körfel är fel som uppstår på grund av att koden försöker göra något som inte fungerar eller som inte är tillåtet. Ett vanligt fel är att man försöker anropa en funktion som inte finns på grund av att man stavat fel i funktionens namn eller glömt lägga till den fil där funktionen finns.

Ett annat vanligt körfel är när man försöker använda ett objekt som inte är instansierat. Dessa fel uppstår exempelvis om man använder funktionen getElementById och har stavat fel i det id man angivit som argument till funktionen. Webbläsaren letar efter det felstavade id.et och hittar inget och returnerar därför null. Om man sedan, utan att kontrollera om man fick tillbaka en referens till ett objekt, försöker använda detta objekt, exempelvis genom att ändra något attribut, så får man ett felmeddelandet. Dessa felmeddelanden är ofta av typen "has no properites".

Logiska fel
Logiska fel är de som är svårast av alla att hitta. Dessa fel är nämligen inte fel som gör att skriptet stannar och ger ett felmeddelande utan är fel som innebär att koden inte gör det vi avsåg att den skulle göra. Koden är alltså helt korrekt ur ett syntax-perspektiv och försöker inte heller göra något som är otillåtet. Vi har alltså tänkt fel när vi gjort vår kod och den gör något annat än vad som var avsett. Det är detta som är de riktiga buggarna i en kod.

Ett annat problem med logiska fel är att de inte alltid uppstår. Ett exempel kan vara talområden. Så länge man anger tal inom ett tillåtet talområde så kör koden som den ska men om man anger ett tal som är så stort att det inte ryms i variabeln så får man ett fel. Ett annat vanligt fel är att man tillåter tryck på knappar som först kräver att man gjort något annat.

Så för att hitta logiska fel så behöver man testa sin kod. Här är några saker man alltid bör prova:

- Ange bokstäver där man förväntar sig att skriva in tal och tvärt om
- Ange negativa tal där man förväntar sig positiva tal
- Tryck på knappar i en ologisk ordning
- Avbryt ett flöde av händelser genom att trycka på en "avbryt/cancel"-knapp

Det är framförallt med logiska fel som man har nytta av en mer avancerad debugger än Error Console. Ett annat vanligt sätt att debugga kod är att skriva ut de värden som man misstänker har fel värden, exempelvis genom att visa en variabels värde i en alert-ruta innan felet uppstår.

Då kurslitteraturen har en introduktion till Firebug och de flesta debuggverktyg har sina egna tutorials och exempel så går vi inte igenom något specifikt verktyg i kursmaterialet.

Länkar

Länkarna nedan är en sammanfattning av de som finns i texten ovan.

Du läser knappast igenom allt med en gång men du bör bekanta dig med dessa sidor:
För att se hur de olika utvecklarna har implementerat DOM så hänvisas till följande webbsidor:
http://msdn.microsoft.com/en-us/library/ms533050(VS.85).aspx
http://developer.mozilla.org/en/docs/Gecko_DOM_Reference

W3Cs standard hittar du här:
http://www.w3.org/DOM/DOMTR

Titta på vilka nodtyper det finns:
http://www.w3.org/TR/DOM-Level-2-Core/core.html#ID-1950641247

Ska läsas: Om skillnaderna mellan DOM och JavaScript:
http://developer.mozilla.org/en/docs/The_DOM_and_JavaScript
http://developer.mozilla.org/en/docs/Gecko_DOM_Reference:Introduction