

Webbprogrammering, DA123A, Studiemateriel 6

Reguljära uttryck

Reguljära uttryck, regular expressions på engelska eller kortare regexp är ett formellt språk för att söka efter strängar i strängar. Ett reguljärt uttryck i JavaScript skapas antingen direkt eller genom objektet RegExp.

```
//direkt  
var pattern = /s$/;  
//eller genom objektet RegExp  
pattern = new RegExp("s$");
```

Som du ser ovan så definieras ett reguljärt uttryck av / istället för " som en vanlig sträng när man skapar ett reguljärt uttryck direkt. Om man däremot använder RegExp-objektet så sätter du vanliga fnuttar runt det reguljära uttrycket. Det reguljära uttrycket ovan matchar alla strängar som avslutas med ett s.

Reguljära uttryck är inte bundet till JavaScript som sådant utan ett fristående språk som finns implementerat i JavaScript, Perl, PHP med flera språk. Olika språk kan dock implementera reguljära uttryck lite olika.

Reguljära uttryck består av en serie med tecken. De flesta tecken beskriver de tecken de bokstavligt motsvarar. Sedan tillkommer ett antal metatecken och element som har en specifik betydelse utöver sin bokstavliga betydelse. I uttrycket ovan så motsvarar s sin bokstavliga betydelse medan tecknet \$ är ett metatecken som motsvarar slutet på en sträng.

Grammatik

Det som nu följer är ett stycke med grammatiken som bygger upp reguljära uttryck. Så som du känner igen från tidigare så stöds ett antal escapade tecken för att motsvara vissa saker. Nedan följer en lista över de vanligaste av dessa element.

\0	NUL-tecknet (\u0000)
\t	tab (\u0009)
\n	ny rad (\u000A)
\v	vertikal tab (\u000B)
\f	form feed (\u000C)
\r	vagnretur (\u000D)
\xnn	latinskt tecken specificerat av det hexadecimala nn, \x0A är samma sak som \n
\uxxxx	unicode-tecken bestämt av det hexadecimala xxxx

Tecken som utgör metatecken är:

`^ $. * + ? = ! : | \ / () [] { }`

Betydelsen av dessa kommer att framgå senare. Om något av dessa tecken skall ingå i sin bokstavliga betydelse i ett uttryck så måste de föregås av backslash \.

Genom att innesluta ett antal tecken inom [] så skapar du en teckenklass. En teckenklass matchar något av de tecken den innehåller. Det reguljära uttrycket /[abc]/ matchar något av tecknen a, b och c medan uttrycket /abc/ bara matchar den exakta strängen "abc". För att enkelt beskriva längre sekvenser av tecken i en teckenklass så kan du använda bindestreck i

förhållande till alfabetisk ordning. Om du vill ha ett uttryck som matchar en liten bokstav a intervallet a-z så skriver du `/[a-z]/`. Observera att detta inte fungerar med ett intervall a-ö.

Du kan negera en teckenklass genom att använda tak (^). Uttrycket `/[^abc]/` matchar alla andra tecken än just a, b och c.

Vissa tecken har i JavaScript fått speciella förkortade uttryck då de är mycket vanliga. Exempelvis så motsvarar `\s` ett mellanslag eller något annat "white space"-tecken i Unicode så som exempelvis tab. Dessa tecken kan också negeras genom att man anger dem med stora bokstäver. `\S` matchar alla andra tecken än just white space.

.	vilket tecken som utom ny rad(newline) eller någon annan Unicode radslutstecken (line terminator)
<code>\w</code>	ASCII-tecken som kan förekomma i ord. Motsvarar <code>[a-zA-Z0-9_]</code>
<code>\W</code>	negationen av <code>\w</code>
<code>\s</code>	någon Unicode whitespace
<code>\S</code>	negationen av <code>\s</code>
<code>\d</code>	vilken ASCII-siffra 0-9 som helst
<code>\D</code>	negationen av <code>\d</code>

För att få en teckenklass som innehåller alla svenska tecken så behöver du lägga till å, ä, ö och Å, Ä, Ö: `[\wääöÅÄÖ]`

Ibland så vill man beskriva en upprepning av tecken exempelvis en följd av fyra siffror. Vi skulle kunna skriva detta som `\d\d\d\d` men det blir i längden besvärligt och klumpigt. Därför så finns det ett antal sätt att beskriva en upprepning.

<code>{n,m}</code>	matchar det föregående elementet minst n gånger men inte fler än m gånger
<code>{n,}</code>	matchar det föregående elementet minst n gånger
<code>{n}</code>	matchar det föregående elementet precis n gånger
<code>?</code>	matchar det föregående elementet ingen eller en gång, motsvarar <code>{0,1}</code>
<code>+</code>	matchar det föregående elementet en eller flera gånger, motsvarar <code>{1,}</code>
<code>*</code>	matchar det föregående elementet inga eller flera gånger, motsvarar <code>{0,}</code>

Några exempel:

`\d{2,4}/` matchar mellan två och fyra siffror
`\w{3}\d?/` matchar tre ord-tecken och en eller ingen siffra
`\s+java\s+/` matchar "java" med en eller flera mellanslag före och efter
`/[^\"]*/` matchar inga eller flera tecken som inte är "

För att välja att antingen så skall uttryck x matchas eller så skall uttryck y matchas så använder man tecknet |. Exempelvis så matchar uttrycket `/ab|cd|ef/` strängen "ab" eller "cd" eller "ef". Alternativen matchas från vänster till höger till något matchas eller man kommit till slutet av alternativen. Så om vi har uttrycket `/a|ab/` och strängen "ab" så matchas bara a då detta hittas först.

Parenteser använd på flera sätt för reguljära uttryck. Ett är att gruppera tecken så att de alla påverkas av metatecken så som | eller *. Uttrycket `/java(script)?/` matchar "java" följt av ett valfritt "script".

Pparenteser används också för att definiera deluttryck. Dessa deluttryck kan man senare vilja referera till. När en sträng matchas så vill vi kunna hitta delar av resultatet. Antag att vi letar efter ett mönster där en eller flera gemener följs av en eller flera siffror. Vi kan då använda mönstret `/[a-zAÖ]+d+/. Men det vi egentligen behöver är siffrorna, tecknen är bara en del i att hitta dessa. Vi kan då använda parenteser för att skilja ut siffrorna. /[a-zAÖ]+(d+)/ tillåter oss senare att få fram bara siffrorna. Hur vi gör detta förklaras i samband med string-metoderna.`

Pparenteser kan genom att de används för att definiera deluttryck också användas när en del av mönstret upprepas. Genom att låta ett `\` följas av en siffra så refererar man till ett tidigare deluttryck av mönstret. `\1` hänvisar till den första delen, `\2` till den andra och så vidare. Då deluttryck kan vara nästlade så är det den vänstra parentesen som räknas. I följande uttryck

```
/([J]ava([S]cript?))\sis\(fun\w*)/
```

så refereras deluttrycket `([S]cript)` till som `\2`. En referens till ett deluttryck refererar inte till uttrycket som sådant utan till det matchade mönstret. På detta sätt kan deluttryck användas för att säkerställa att delar av mönstret innehåller exakt samma tecken. Antag följande uttryck

```
/['"]^[^']*['"]/
```

Som hittar en sträng som inleds med `'` eller `"` och därefter består av inga eller valfritt antal tecken som inte är `'` eller `"` och sedan följs av `'` eller `"`. Det vi är ute efter är "någon text" eller 'annan text' men inte "olika fnuttar". För att bara hitta de strängar som inleds med matchande fnuttar så kan vi skriva

```
/(['"])[^']*\\1/
```

`\1` referensen gör nu att vi matchar de inledande fnuttarna så dessa två strängar måste vara lika. Observera att du inte kan använda referenser till deluttryck i en teckenklass så det går inte att skriva `/(['"])[^\\1]*\\1/`.

Förutom de element som matchar ett specifikt tecken i en sträng så finns det element som matchas positioner snarare än direkts tecken. `\b` matchar exempelvis gränsen(boundary) mellan ett ASCII-ordtecken `\w` och ett icke ASCII-ordtecken `\W` eller gränsen mellan `\w` och slutet eller början på en sträng. `\b` kan negeras med `\B`. Dessa typer av element kallas ankare då de ankrar ett mönster till en viss position i söksträngen. Om `^` används utanför en teckenklass(där det negerar teckenklassen) så ankrar denna till början av en sträng. `$`, som vi hade i första exemplet, ankrar till slutet av en sträng. Om vi exempelvis vill matcha ordet "JavaScript" på en ensam rad så skulle vi kunna använda uttrycket `^JavaScript$`.

Flaggor används utanför det egentliga uttrycket för att specificera hur sökningen skall ske. Tecknet `i` som flagga anger att sökningen inte skall ta hänsyn till gemener och versaler (case-insensitive). Tecknet `g` som flagga anger att sökningen skall vara global, det vill säga att alla matchningar som kan göras i söksträngen skall ske och inte bara den första. Om vi söker alla ställen där "java" förekommer ensamt i en text oavsett om det är stavat "Java" eller "JAVA" eller med någon annan kombination av gemener och versaler så skulle vi kunna använda `\bjava\b`.

I JavaScript 1.5 finns även flaggan `m` som anger "multiline mode". Detta innebär att om söksträngen innehåller newline-tecken så matchar `^` och `$` börjar och slutet på en rad så väl som början och slutet på en sträng

String-metoder för mönstermatchning

I JavaScript så finns metoderna `search()`, `replace()`, `match()` och `split()` till alla strängar. Dessa kan ta reguljära uttryck som argument för att matcha en teckenföljd i en sträng.

Metoden `search()` tar emot ett reguljärt uttryck och returnerar antingen den position där den matchade strängen börjar om en sådan finns eller `-1` om ingen matchning kunde göras. Exempelvis så returnerar följande 4.

```
result = "JavaScript".search(/script/i);
```

Om argumentet till `search()` inte är ett reguljärt uttryck så konverteras det till ett sådant genom användning av `RegExp` objektet. Observera att `search()` inte fungerar med flaggan `g`. Om denna är satt så ignoreras den bara.

Metoden `replace()` utför en sök och ersätt funktion. Det första argumentet är ett reguljärt uttryck som anger vad som skall matchas och det andra argumentet anger vad en eventuell matchning skall ersättas med. Om flaggan `g` är satt så kommer alla matchningar i strängen att ersättas annars så ersätts bara den första matchningen. Till skillnad från `match()` så fungerar `replace()` så att om det första argumentet inte är ett reguljärt uttryck så söker metoden efter den angivna strängen exakt. Så för att se till att man alltid skrivit JavaScript istället för exempelvis Javascript i en text så skulle man kunna göra följande:

```
text.replace(/javascript/gi, "JavaScript");
```

I `replace()` får vi nytta av att definiera deluttryck. Om `$` följt av en siffra förekommer i den sträng som vi vill ersätta matchningar med så tolkas detta som det deluttryck som siffran motsvarar. Så om vi till exempel vill ersätta " med ' i en sträng så skulle vi kunna göra så här:

```
var quotePattern = /"([^\"]*)"/g;  
text.replace(quotePattern, "'$1'");
```

Andra nyttiga egenskaper hos `replace()` är att det andra argumentet kan vara en funktion som beräknar ersättningssträngen. För mer information hänvisas till referenserna i Länk-avsnittet.

Metoden `match()` har precis som `search()` bara ett argument som är ett reguljärt uttryck eller omvandlas till ett sådant. `Match` returnerar dock en array som motsvarar resultatet av matchningen. Exempelvis

```
"1 plus 2 blir 3".match(/\d/g);
```

Returnerar arrayen `["1", "2", "3"]`.

Om flaggan `g` inte används så returneras en array där det första elementet innehåller matchningen i sin helhet och efterföljande element innehåller definierade deluttryck.

Ett exempel:

```
var urlPattern = /(\\w+):\\/\\/([\\w.]+)\\/\\(\\S*)\\/;  
var text = "Besök hemsidan med adress http://homeweb.mah.se/~tf123456";  
var result = text.match(urlPattern);  
if (result != null){  
    var fullurl = result[0]; //innehåller "http://homeweb.mah.se/~tf123456"  
    var protocol = result[1]; //innehåller "http"  
    var host = result[2]; //innehåller "homeweb.mah.se"  
    var path = result[3]; //innehåller "~tf123456"  
}
```

Uttrycket ovan innebär att första deluttrycket skall vara en eller flera ASCII-ordtecken. Detta skall följas av ett kolon och två slash (://). Det andra deluttrycket skall utgöras av en ordklass som består av ASCII-ordtecken och alla andra tecken utom Unicode radslut. Deluttrycket kan innehålla ett eller flera tecken ut ordklassen. Efter denna skall det komma ett slash (/) som följa av det tredje deluttrycket som kan vara inga eller flera tecken så länge det inte är ett "white space".

Men om vi nu har en längre text och vill hitta alla url:ar som finns och dess delar? Om vi anger flaggan g så får vi ju bara matchningarna i sin helhet och utan g så får vi bara den första. Om man anger ett uttryck utan g så anger match även egenskaperna index och input. Index anger det första tecknet i matchningens position och input är en kopia av strängen match() anropades för. I exemplet ovan så blir index 26 och input samma som variabeln text. Om vi antar att vi har det reguljära uttrycket r som inte har flaggan g satt och anropar s.match(r) så får vi samma resultat som om vi anropar r.exec(s) vilket vi kommer till i delen om RegExp-objektet nedan.

Metoden split() delar upp en sträng i delsträngar i en array utifrån en given separator.

```
"123,456,789".split(","); //ger ["123","456","789"]
```

Genom att använda reguljära uttryck kan vi få mer flexibla separatorer. Om vi vill ha en separator som tillåter ett valfritt antal mellanslag så kan vi skriva:

```
"1,2, ,3 , 4 ,5".split(/\\s*,\\s*/); //ger ["1","2","3","4","5"]
```

RegExp-objektet

Konstruktorn till RegExp() tar en eller två strängar som argument och skapar ett nytt RegExp-objekt. Det första argumentet skall vara en sträng som innehåller det reguljära uttrycket. Då denna behandlas som en sträng så måste du escapa alla \ som förekommer framför element. Det andra argumentet är valfritt och motsvarar de flaggor som kan sättas för reguljära uttryck.

```
var zipcodePattern = RegExp("\\d{5}", g)  
//hittar alla (post)nummer utan mellanslag. Observera \\ i elementet \\d
```

RegExp-objektet behövs för att kunna skapa dynamiska reguljära uttryck som inte kan förutsägas vid kodningen, exempelvis om en användare skall kunna söka efter något.

För att göra matchningar av uttrycket mot en text så används metoden exec(). Denna liknar metoden match() för strängar men utförs på uttrycket med en sträng som argument istället. Om ingen matchning hittas så returnerar exec() null. Om en matchning hittas så returnerar exec() en array motsvarande den som beskrivits för match(). Skillnaden mot match() är att exec() returnerar samma array oavsett om flaggan g är satt eller inte. Om du sedan anropar

exec() igen för samma uttryck så börjar sökningen från positionen direkt efter den matchade strängen. Om ingen matchning gjordes denna gång så sätts egenskapen lastIndex till 0. På detta sätt så kan du loopa igenom en text och alltid få fram alla matchningar med alla deluttryck. Exempelvis:

```
var pattern = /Java/g;
var text = "JavaScript är inte samma sak som Java!";
var result;
while((result = pattern.exec(text)) != null){
    alert("Matchade '"+result[0]+' ' på position " +
        result.index + "; nästa sökning startar på "+ pattern.lastIndex);
}
```

Om du vill att sökningen skall starta från början igen så sätter du lastindex till 0 själv. Det gäller att tänka på detta om du avbryter en sökning i en sträng för att sedan använda samma uttryck till att söka i en annan sträng. Om du då inte satt lastindex till 0 så kommer sökningen i den nya strängen att börja på lastIndex som i värsta fall är utanför den nya strängen och ett fel uppstår. I bästa fall så missar du "bara" någon matchning. Om uttryck utan flaggan g används så ignoreras lastIndex.

Metoden test() talar om ifall en matchning finns. Metoden beter sig på samma sätt som exec() i förhållande till egenskapen lastIndex så man kan loopa sig igenom en text och exempelvis räkna antalet matchningar om man inte är intresserad av matchningarna i sig.

Formulär och validering

Formulär

Du bör redan vara bekant med formulär och grunderna i hur dessa fungerar men här kommer en snabb repetition.

Formulär är en del av HTML som används för att ge interaktivitet i en webbsida, användaren kan skriva in information och göra olika val i formuläret. Formulär kan infogas i ett HTML-dokument tillsammans med "vanliga" HTML-taggar med <form>-taggen. Datan i formuläret skickas till ett program som bearbetar det. Vart och hur datan skickas specificeras i formuläret.

Vi börjar ett formulär med taggen <form> och avslutar det med </form> Mellan dessa taggar kan en eller flera byggstenar läggas till. Alla övriga HTML-taggar kan också skrivas mellan taggarna. Meningen med formulär är att man skall kunna skicka data från användaren till ett program som bearbetar det. Vart data skall skickas specificeras i attributet *action="URL"*. Som värde i *action* skriver man adressen till det program som data skall skickas till. Det finns två olika metoder att skicka formulärdata med: POST och GET.

Med GET skickas all data i URL:en. Om ni har använder till exempel Hotmail så kolla på URL:en nästa gång ni loggar in, så får ni se en lång konstig URL. Det beror på att en massa data från det underliggande formuläret har skickats men GET.

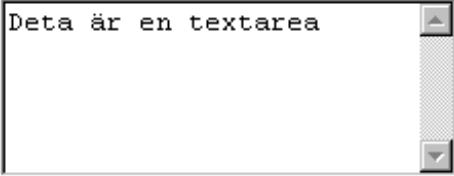


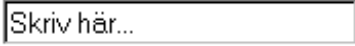


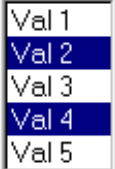
Med POST skickas all data som innehåll i HTTP-protokollet. Hur detta görs behöver vi inte bry oss om, det viktiga här är att data inte syns för användaren. POST använder vi också om vi vill att data skall kunna formateras läsligt. Det vill säga om vi vill skicka data via e-post. Att använda mailto är dock en ganska dålig form av action-värde då alla användare kanske inte alltid skriver in POP3 server och kontouppgifter i sin webbläsare. En modernare metod är

då att använda sig av serverbaserade tekniker och skicka e-post-meddelandet från servern. Vi kommer att se detta i den andra hälften av kursen.

Sammanfattningsvis:

- `<form method = "get | post">`
 - anger *hur* datat ska skickas
 - *get* sänder data med URL:en (*synligt i QUERY_STRING*)
ex: `http://www.studweb.ts.mah.se/cgi-bin/registry.cgi?Namn=&Land=Sverige`
 - *post* sänder via STDIN (vanligast)
- `<form action = "URL">`
 - anger mottagande script (alt: mailto) dvs vem (obs dock att användaren måste ha anggett sina uppgifter och mailserver etc. för att mailto ska fungera.)

Det finns en mängd olika byggstenar att bygga upp ett formulär med. Här är några av de vanligaste

<p>Textarea</p> 	<p>Återställning</p> 	<p>Skicka</p> 
<p>Flervalsruta – variant 1</p>		
<p>Textfält</p> 		
<p>Dolt textfält</p> 		
<p>Radioknappar</p> <p><input checked="" type="radio"/> Val 1 <input type="radio"/> Val 2 <input type="radio"/> Val 3</p>	<p>Kryssrutor</p> <p><input checked="" type="checkbox"/> Val 1 <input type="checkbox"/> Val 2 <input type="checkbox"/> Val 3</p>	<p>Flervalsruta – variant 2</p> 

Beroende på vad det är för typ av information man vill ha från användaren så använder man en passande byggsten.

Textfält används för enradiga inmatningar, t.ex. ett namn

Dolt textfält används för t.ex. lösenord som inte får visas på skärmen.

Kryssrutor används för att t.ex. välja svar på en fråga (flera kan väljas)

Radioknappar används för att t.ex. välja ett svar på en fråga (endast en kan väljas)

Textarea (eller flerradigt textfält) används för att hämta godtycklig inmatning.

Flervalsruta – variant 1, används ungefär som radioknappar (tar mindre plats vid många val). Viks ut vid klick.

Flervalsruta – variant 2, som ovan fast med alla val synliga och möjlighet till att välja flera val.

Återställningsknapp, rensar inmatningsfält och återställer övriga valbara saker

Knapp för att skicka, skicka all information i formuläret till ett mottagande program på en specificerad adress.

Det finns två huvudtyper av sätt att hämta data från användaren:

Inmatning från användaren - `<input>`

Fördefinierade val - `<select></select>`, `<option>`

Vid inmatning använder man taggen `<input>`. De viktigaste attributen för `<input>` är:

name, *value* och *type*. *Name* namnger inmatningen så att vi kan känna igen var användaren har skrivit vad. Till exempel namnger vi ett textfält med `name="Namn"`. Det som skickas till servern blir då `Namn=Linus`, om nu användaren skrivit in "Linus". I *value* skriver vi eventuell text som skall stå där när sidan laddas. Till exempel `value="Skriv ditt namn här..."`. Detta står då ifyllt när användaren skriver i något. Beroende på vad man sätter attributet *type* till så får inmatningen olika utseende. De vanligaste typerna och de som jag visat här är:

- Textfält – enradig inmatning, t.ex. ett namn
- Dolt textfält – dolt enradig inmatning, t.ex. ett lösenord
- Återställningsknapp – Rensar allt inskrivet i formuläret och ställer tillbaka alla val.
- Skickaknapp – Skickar iväg all data till den i formuläret specificerade adressen.

Typerna `type="reset"` och `type="submit"` blir automatiskt knappar och vi kan skriva texten på dem med `value="Skicka"` och `value="Återställ"`.

Det finns två andra typer av `<input>` som är av intresse då de behandlas lite annorlunda. Den ena är radioknappar och den är andra kryssrutor. Dessa väljer man om man vill att användaren skall svara ja eller nej på en fråga. Dessa kan också användas om man vill låta användaren välja mellan några få olika val. Om vi sätter attributet *type* till `type="radio"` så får vi en radioknapp. Om vi sätter attributet *type* till `type="checkbox"` så får vi en kryssruta. Meningen med radioknappar är att man endast skall kunna välja ett av ett antal val. Med kryssrutor däremot kan man välja flera.

OBS! Viktigt är att man sätter *name* attributet i alla radioknapparna i samma grupp till samma värde. Annars kan inte webbläsaren känna av att de tillhör samma grupp. Det är bra att klippa och klistra in men kom ihåg att inte ha samma namn på olika grupper, detta kan ge svårlokaliserade (logiska) fel, speciellt i JavaScript. Attributet *value* innehåller det värde som skickas till servern. Med attributet *checked* som är ett attribut utan värde sätter man den förvalda radioknappen. Man kan även välja att ha en kryssruta ikryssad från början med detta attribut.

Skillnaden mellan radio- och kryssrutorna är att i det första fallet kan endast ett alternativ väljas - det är alltså av uteslutande typ. För kryssrutor kan inget, ett eller flera alternativ väljas. För radiatorerna måste *name*-attributet vara samma för alla element som hör till samma grupp, det vill säga de alternativ som finns att välja på. *Checked* har värdet `true` eller `false` beroende om alternativet är valt eller inte (förväxla inte detta med HTML-attributet *checked* som inte har något värde).

Man kan få tag värdena i gruppen dator på följande sätt:

```
for (var i in dator) {  
    if dator[i].checked {  
        found = true;  
        break;  
    }  
}  
choice = dator[i];
```

Utöver den data som användaren matar in i ett formulär så vill man ibland skicka mer. Det kan till exempel röra sig om priset på en produkt i en webbshop. Det vill man ju inte att användaren vare sig skall mata in eller ändra på. Då kan man använda sig av så kallade gömda fält. Gömda fält fungerar på samma sätt som övriga `<input>`-objekt förutom att användaren inte kan påverka värdet. Man anger alltså både attributet `name` och `value` redan från början och användaren kommer aldrig att se det.

```
<form action="http://www.studweb.ts.mah.se/cgi-bin/registry.cgi"  
metod="get">  
    <input type="hidden" name="Pris" value="520">  
    <input type="submit" value="Skicka">  
</form>
```

I mer avancerade tillämpningar ska man hålla sig till metoden POST när man skickar data och kontrollera data innan det kommer till webbservern eftersom användaren annars mycket väl kan skicka kod i indata i försök till olaga intrång på servern.

Ibland vill man begränsa användaren till att ange vissa svarsalternativ. I dessa fall gör användaren ingen egen inmatning utan väljer ibland en mängd fördefinierade val. Framförallt pratar vi nu om fall då man har många alternativ att välja mellan. Man märker att det tar för mycket plats att använda radioknappar eller kryssrutor. Vi använder då oss av taggarna `<select>` och `<option>`. Man startar den valbara listan med taggen `<select>`. Man anger attributet `name` i `<select>` precis på samma sätt som i `<input>`, för att ge namn åt data. Sedan anger man för varje valbar sak taggen `<option>` och efter `option` skriver man det som skall stå i den valbara listan.

Om man inte skriver några attribut i taggen `<option>` så skickas det som står i listan som värde. Vill man istället att något annat skall skickas så anger man attributet `value="värde"` i taggen `<option>`. Till exempel så kanske man vill lista alla världens länder men istället för att skicka hela namnet på landet så vill man bara skicka landskoden.

```
<option value="SE">Sverige</option>
```

Det finns två andra viktiga attribut till `<select>`, *multiple* och *size*. Om man inte anger *size* sätts det som standard till "1" och vi får den klassiska rullgardinslistan (variant 1) som visar sig när man klickar på pilen. Anger man till exempel `size="5"` visas en lista med 5 synliga alternativ. Finns det fler alternativ så kommer en scrollist fram automatiskt.

Multiple är ett argument utan värde. Man anger alltså bara *multiple*. Om man gör det så kan användaren välja flera alternativ i listan genom att hålla ner CTRL eller SHIFT och klicka.

Ett fullständigt exempel kan alltså se ut så här:

```
<select name="smak">
  <option value="vanilj">Vanilj</option>
  <option value="jordgubb">Jordgubb</option>
  <option value="RomRussin">RomRussin</option>
  <option selected>Aprikos och Apelsin</option>
</select>
```

Ett flerradigt inmatningsfält specificeras med hjälp av antalet rader *rows* och kolumner *cols*. `<textarea>` använder vi om vi vill ha flerradig inmatning från användaren. Med attributen *rows* och *cols* anger vi storleken på vår textarea. Skulle texten överstiga den storleken får vi automatiskt rullister. Vi namnger på samma sätt som innan data med attributet *name*. Vi har här däremot inget *value*-attribut. Vill vi att något skall stå skrivet i textarean från början så skall det skrivas mellan taggarna `<textarea>Står från början...</textarea>`

Validering

Validering bör ske redan på klientsidan av ett formulär för att undvika att man tar upp onödig bandbredd genom att skicka data som ändå är felaktig. Validering är egentligen bara en form av händelse. Det viktiga är dock att man stoppar defaulthändelsen, skickandet av formulärdata, om formuläret skulle innehålla ogiltig data. Detta beskrevs i det föregående studiematerialet. För att validera formuläret innan det skickas så används händelsen `submit` för `form`-elementet.

Man behöver även bestämma sig för när man behöver validera ett formulär. Detta kan ske efterhand som användaren skriver in data eller innan formulärets data skickas. Ofta gör man både och men som minst så måste formuläret valideras innan det skickas. För att validera en del av ett formulär när användaren angivit data är det lämpligt att använda händelsen `blur`. Denna inträffar när fokus lämnar ett element.

När fel har upptäckts i formulärets data så måste detta meddelas till användaren. Dessa meddelanden skall vara genomtänkta och inte lämna användaren frågandes vad det är som egentligen är fel. Om man använder `alert`-metoden så skall man vara noggrann med att ange vilket element det är som felet finns i samt vad som är fel eller reglerna för giltig data. Ett bättre sätt är att se till att lägga utrymme för felmeddelanden i anslutning till de olika elementen. Genom att använda `span`-taggen och låta innehållet vara tomt från början så kan du vid fel fylla denna med ett meddelande om felet. Fördelen med detta är att felmeddelandet kan visas vid det element som är felaktigt och detta ökar användbarheten.

Underskatta inte heller effekten av färg när det gäller att visa felmeddelanden. Att färga texten i ett felmeddelande med rött eller någon avvikande färg om rött används redan, gör det lättare för användaren att uppmärksamma felet. Det gäller dock att tänka på att somliga är färgblinda och att felmeddelanden syns tydligt även om man har problem med detta.

Man bör också förebygga fel genom att vara tydlig med vilken data som väntas och vilka fält som exempelvis är obligatoriska.

Cookies

Vad cookies är och innebär

På vanliga webbsidor styr vi helt och hållet hur vår "kund" kan surfa genom att lägga ut länkar med mera. Vi för alltså i strikt betydelse en monolog med användaren som endast kan välja bland de alternativ vi ger. Vår sida är alltså statisk. Med hjälp av formulär kan vi få en dialog med användaren, det vill säga hämta data och presentera en ny sida med ledning av vilken data kunden skickat. Vi får alltså en dialog med kunden, det vill säga interaktivitet. Vi spelar alltså delvis över "bollen" till kunden som själv kan bestämma mer.

Syftet med cookies är att kunna lagra information på klientens dator och därmed kringgå http-protokollets svaghet att inte kunna lagra data. Exempel på data som ofta lagras är val som användaren gjort tidigare och olika inställningar av en webbsida. När användaren så besöker sidan igen så kan man hämta denna information och anpassa sidan efter detta.

Cookies är en textfil som lagras på klientens dator och är enda sättet för en server att lagra information på din dator utan att begära nedladdning. Cookies kan sparas en godtycklig tidsperiod och hanteras enklast i JavaScript genom det inbyggda objektet `document.cookie`

Enligt den lagen om elektronisk kommunikation, som trädde i kraft den 25 juli 2003, ska besökaren på en webbplats informeras om att webbplatsen använder cookies. Användaren ska också ges möjlighet att hindra cookies. Om du inte accepterar användandet av cookies kan du ändra inställningar i din webbläsare så att den inte tillåter cookies.

Cookies fungerar exempelvis så att första gången besökaren surfar till sidan skrivs namnet in (t ex "Lena Olin") Nästa gång hon, eller någon vid samma dator i varje fall, besöker sidan kan denna visa: "Välkommen till xyz igen Lena Olin! Trevligt att du besöker sajten igen." Det är naturligtvis inte så cookies används i banksystem etc., men principen är densamma. Information lagras hos klienten för att användas igen.

Ett annat exempel skulle kunna vara att första gången kunden surfar till din sida frågar du efter vilket språk som föredras och lagrar det valda språket i en cookie. Vid nästa besök läser vi besökarens cookie och han/hon hamnar på en sida med det valda språket (d.v.s. vi behöver inte fråga igen). Vi kan alltså spara (komma ihåg) de val som kunden gjort och utgå från vilka val kunden tidigare gjort.

En annan vanlig funktion som cookies används till är att första gången kunden surfar till din sida frågar du efter vilket lösenord han/hon vill ha och du sparar det i en cookie. Vid nästa besök hämtas lösenordet från cookien. Sådana här lösningar skall man dock vara försiktig med. Kunden kan själv kasta bort cookies som ligger som textfiler på dennes hårddisk eller kunden kan också ställa in så att cookies inte accepteras av webbläsaren. Så nästa gång så finns inte lösenordet sparad längre.

Cookiens uppbyggnad

Varje cookie består av en eller flera crumbs (smulor eller delar). Hela cookien (med alla smulor) hanteras med hjälp av `document.cookie` som nämnt tidigare. Varje smula kan styras av egenskaperna:

- namn (name)
- värde (smulans innehåll)
- bäst-före-datum (expires)
- sökväg (path)
- domän (domain)
- säker (secure)

Endast en cookie kan sättas per html-sida. Namnet cookies är använt fel i många böcker på grund av att författarna ofta refererar till smulorna (crumbs) som cookies. Detta beror delvis på att när vi sedan använder funktionerna `setCookie` och `getCookie` så gör vi samma sak, det vill säga vi sätter en del av en cookie, en crumb men kallar funktionen `setCookie`. Funktionen borde egentligen heta `setCrumb`.

En cookie är en textsträng och smulorna är delar av denna textsträng. I smulorna lagrar vi data. Cookien kan antingen vara varaktig (tidsstyrd) eller en session-cookie.

Varaktig (tidsstyrd) –cookie:

- kan läsas av webbservern och JavaScript intill dess att värdet i `expires` har passerats
- kan bara läsas av server och sidor när de laddas från samma server och samma path eller tillhörande underkatalog
- raderas genom att sätta egenskapen `expires` till en tidpunkt som redan varit

Session-cookie

- kan läsas av webbservern och JavaScript (samma server och path)
- raderas om du "surfar" till annan webbplats eller stänger webbläsaren
- sätts genom att utelämna "expires" egenskapen.

Varje smula har ett namn. Det är smulans namn som gör att vi kan hitta den igen (adressera den). Jämför med variabelnamn. Varje meningsfull smula har utöver namnet alltid ett värde. Detta värde är alltid en textsträng.

Följande kod sätter smulan med namnet "namn" till värdet "Adam":

```
document.cookie = "namn=Adam";
```

Cookien består nu alltså enbart av smulan "namn". Texten som finns sparad i smulan är nu "namn=Adam". Denna sessioncookien är mycket enkel och vi använder endast två av de tidigare nämnda egenskaperna.

Följande kod sätter två smulor

```
document.cookie = "fornamn=Adam";  
document.cookie = "efternamn=Svensson";
```

Cookien består nu av smulorna `fornamn` och `efternamn`. Texten som är sparad i cookien är "fornamn=Adam; efternamn=Svensson". Vi har alltså den sträng av så kallade smulor (delar av strängar eller sub-strängar) som utgör själva cookien. Det är alltså ; och ett mellanslag som

separerar smulorna. Semikolon ; separerar crumbs från varandra och likhetstecken = separerar namn från värde.

Ofta innehåller en sidas cookie mer än bara en smula. JavaScript innehåller ingen färdig teknik för att hämta specifik smula - därför måste följande tankesätt användas:

- 1) Hämta hela kakan
- 2) Dela kakan vid alla ; så att du får en array som innehåller alla smulor
- 3) Loopa genom denna array
- 4) Dela smulan vid "=" så du får namn och värde i en ny array
- 5) Stämmer namnet så har du hittat din smula. Hämta värdet och avbryt loopen. Leta eventuellt efter fler smulor ... (goto 3)

Alternativ teknik

- 1) hämta hela kakan
- 2) finn position A för den sökta smulans namn i strängen följt av ett "="
- 3) sök position B för efterföljande ";"

Det som erhålles mellan A och B är kakans värde. När vi skall separera data i en cookie så har man alltså nytta av reguljära uttryck.

Vid kodning kan det vara bra att skriva ut alla smulor. Exempelvis så tilldelas smulor:

```
document.cookie="fordon=bil";  
document.cookie="maxfart=100km/tim";  
document.cookie="farg=rod";  
document.cookie="passagerare=3";
```

Texten i cookien är nu "fordon=bil; maxfart=100km/tim; farg=rod; passagerare=3". Följande skript loopar genom alla smulor och skapar formaterade rader:

```
var crumbs = document.cookie.split('; ');  
for (var i in crumbs) {  
    var cname = crumbs[i].split('=');  
    var cvalue = unescape(crumbs[i].split('=')[1]);  
    document.write('smula nr ' + i +  
        ' heter ' + cname +  
        ' och har värdet ' + cvalue + '<BR>');  
}
```

ger:

smula nr 0 heter fordon och har värdet bil
smula nr 1 heter maxfart och har värdet 100km/h
smula nr 2 heter farg och har värdet röd
smula nr 0 heter passagerare och har värdet 4

Attributet expires motsvarar bäst-före-datum och består av en sträng som erhålles av metoden toGMTString(). Vi vill sätta smulan "namn" med värdet "Adam" och hållbarheten två timmar:

```
var expdate = new Date();  
expdate.setTime(expdate.getTime() + 2 * 60 * 60 * 1000);  
document.cookie = "namn=Adam; " + "expires=" + expdate.toGMTString();
```

Expires kan liknas med ett bäst före datum på kakan. Efter det har passerats kommer kakan att sluta existera.

En cookie består av alla de smulor som ligger i samma path som dokumentet själv eller i undermappar. Exempel: tre webbsidor på samma server sparar varsin smula, URL till sidorna är:

<http://x.com/1.htm>

<http://x.com/~nisse/2.htm>

<http://x.com/~nisse/mapp/3.htm>

1 sparar smulan x, 2 smulan y och 3 smulan z. När en kund besökt sidorna 1, 2 och 3 gäller från sidan 1 smula x, från sidan 2 smulorna x och y från sidan 3 smulorna x, y och z

Attributet path kan vara exakt samma mapp som dokumentet finns i, eller en eller flera mappar under denna. Exempel på <http://x.com/~nisse/sida.html>

```
var expdate = new Date();
expdate.setTime(expdate.getTime() + 2*60*60*1000);
document.cookie = "namn=Adam; " +
    "expires=" + expdate.toGMTstring() + "; " +
    "path=/~nisse/katalog"; *
```

Detta medför att smulan namn syns i både kakor från <http://x.com/~nisse> och <http://x.com/~nisse/katalog>

Om domän sätts så förhindrar detta att cookien kan läsas från "felaktiga" servrar, t ex om någon "lånar" ditt skript och lägger det på en egen server. Detta kan dock enkelt ändras av kopieraren om denne vet vad han/hon gör. Sätts på samma sätt som path, t ex:

```
var expdate = new Date();
expdate.setTime(expdate.getTime() + 2*60*60*1000);
document.cookie = "namn=Adam; " +
    "expires=" + expdate.toGMTstring() + "; " +
    "path=/~nisse; " + "domain=x.com; ";
```

Cookien kan med detta attribut fås att bara läsas av den server som skrev den. För att detta attribut ska fungera så måste den läsas med ett annat script-språk än JavaScript, till exempel PHP eller ASP för att vi skall kunna sätta path och domän innan sidan når klienten.

Sista attributet för en smula är "secure". Genom att sätta denna begränsas åtkomligheten ytterligare. Cookien blir *enbart* läsbar från server och script *om* anslutningen mot servern är säker (<https://x.com>). Exempel:

```
var expdate = new Date();
expdate.setTime(expdate.getTime() + 2*60*60*1000);
document.cookie = "namn=Adam; " +
    "expires=" + expdate.toGMTstring() + "; " +
    "path=/~nisse; " + "domain=x.com; " +
    "secure";
```

Kodningen underlättas av att skriva funktioner som läser och sätter cookies. Dessa funktioner måste till skillnad från till exempel PHP (som vi ska behandla senare i kursen) skrivas för hand. De är alltså inte inbyggda funktioner i språket JavaScript. Exempel på hur man kan skriva sådana funktioner hittar du i de medföljande exemplen.

Länkar

Referenser, tutorials med mera:

<http://www.regular-expressions.info/>

Hos Mozillas JavaScript

http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Guide:Regular_Expressions

http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference:Global_Objects:RegExp

http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference:Global_Objects:String:match

http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference:Global_Objects:String:replace

http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference:Global_Objects:String:search

http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference:Global_Objects:String:split

Kort om cookies

På svenska: <http://www.javascript.nu/kakor.shtml#omkakor>

På engelska: <http://www.echoecho.com/jscookies.htm>

http://www.w3schools.com/js/js_cookies.asp

Länk till lagtext som medför att man måste upplysa om cookies via konsumentverket

<http://www.internetit.konsumentverket.se/mallar/sv/artikel.asp?lngCategoryId=874&lngArticleId=3213>