Malmö högskola
Centrum för teknikstudier

Webbprogrammering, DA123A, Studiematerial 3

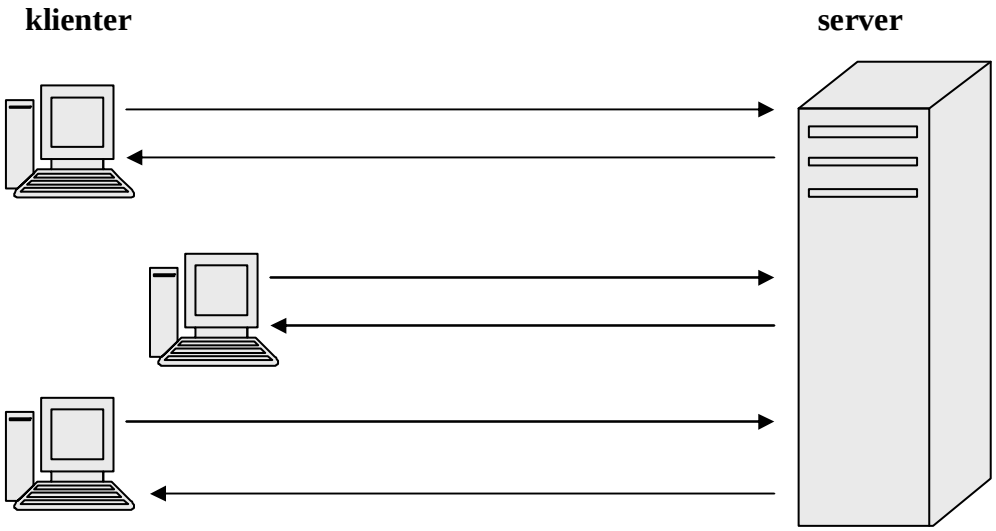
Detta studiematerial handlar till största delen om grunderna i JavaScript och är att se som en repetition med viss utökning för den som läst webbutveckling – fortsättningskurs eller en snabb introduktion till ett nytt språk för den som redan läst 7,5 hp programmering.

Vi utgår från att du redan har en grundläggande kunskap om och har använt JavaScript eller att du är familjär med programmering i stort. För den som behöver repetera eller den som behöver utöka sin kunskap om JavaScript så förutsätts du använda dig av de länkar som finns på kursplatsen och i studiematerialet. Du bör leta upp, om du inte redan har det, bra referenssidor på nätet över JavaScript så du kan repetera eller sätta dig in i den mängd av inbyggda funktioner som finns samt om du är intresserad av detaljer i språket. Du kommer att behöva referenssidor inför inlämningsuppgifterna. Under Länkar sist i studiematerialet finns några referenser. Dessa är dock kanske inte de mest lättnavigerade men de är definitivt pålitliga.

Klient/Server-modellen

För att lyckas med webbprogrammering så är det viktigt att förstå klient/server-modellen för att förstå var olika typer av kod exekveras och följderna av detta. Några begrepp:

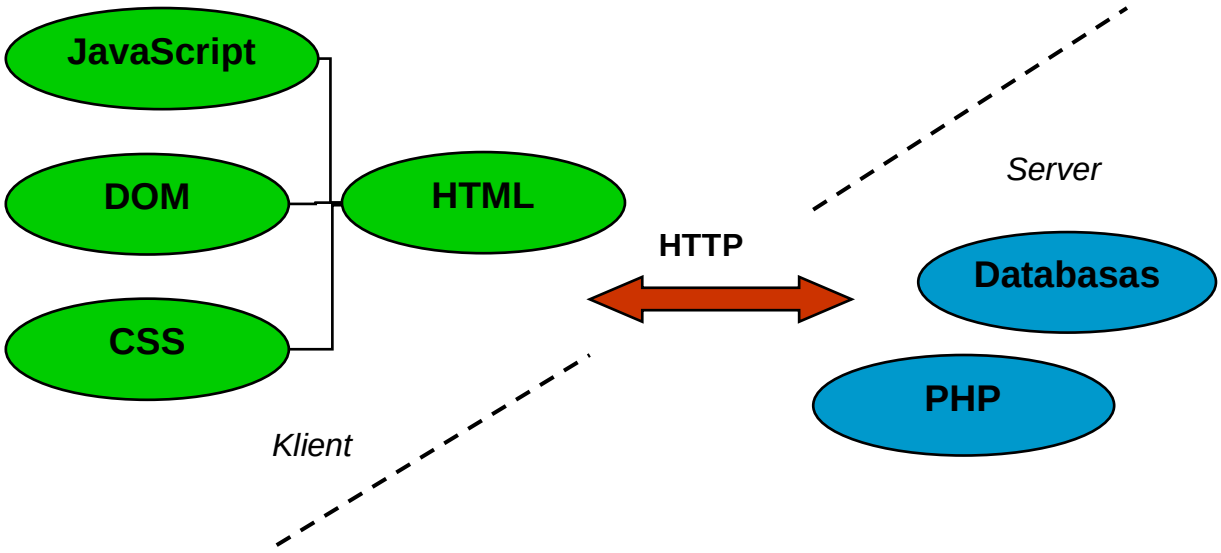
- Server – en server tillhandahåller tjänster för klienter. Detta kan vara ett enda program för en viss specifik tjänst eller en dator med serverprogramvara som tillhandahåller flera olika tjänster.
- Klient – en klient anropar en server för att begära en tjänst. Det är klienten som initierar kommunikationen mellan klient och server.



De flesta tjänster på Internet fungerar enligt server/klient-modellen. Den vanligaste tjänsten är att www-klienter, det vill säga webbläsare som begär webbsidor från en server.

De två språk vi kommer att använda på kursen, JavaScript och PHP, skiljer sig åt på en väsentlig punkt när det gäller hur de fungerar i server/klient modellen. JavaScript exekveras hos klienten efter att denna fått webbsidan medan PHP exekverar på servern innan klienten får webbsidan.

En webbplats är alltså uppbyggd av olika delar på både klient och server-sidan. Kommunikationen mellan dessa sker oftast med http, HyperText Transfer Protocol. I modellen nedan syns vilka teknologier som utförs på vilken sida i server/klient modellen.



HTML (HyperText Markup Language) är grundstommen i webbplatsen. Med **HTML** struktureras informationen på ett logiskt sätt. Informationen delas in i stycken, rubriker, listor, tabeller etc.

Med **CSS** (Cascading Style Sheets) läggs färg, typografi och layout till den logiska struktur som byggts upp med **HTML**

JavaScript ger möjlighet till interaktivitet och dynamik i det annars statiska gränssnittet som utgörs av **HTML** och **CSS**

DOM (Document Object Model) är modellen efter vilken en webbsida är uppbyggd. Alla objekt/element i webbsidan finns hierarkiskt ordnade i **DOM**.

PHP används för att generera webbsidor utifrån givna premisser på serversidan innan den genererade sidan sedan skickas till klienten. PHP används ofta för att hämta och strukturera information från en databas.

Det finns andra teknologier än de vi använder i denna kurs. Till exempel finns ASP (Active Server Pages) som fungerar på ungefär samma sätt som PHP, och som är bättre lämpat för vissa miljöer/plattformar. Valet av databas görs även baserat på den miljö/plattform man arbetar i. Ett annat exempel, som binder samman de två teknologierna är AJAX (Asynchronous JavaScript And XML) som använder sig av http för att hämta information från en server och sedan infoga detta med JavaScript i en sida som redan hämtats till viss del.

JavaScript – introduktion

Med HTML och CSS kan bara statisk information visas men med hjälp av JavaScript så kan man skapa dynamisk information och interaktion med användaren. Då JavaScript exekveras på klientsidan så kan man även flytta en del beräkningskraft från servern till klienten. Nästan alla webbläsare stödjer JavaScript men där är ibland skillnader i hur JavaScript är implementerat och vilka funktioner som finns tillgängliga. Vi kommer i kursen att hålla oss till W3C-standard för DOM(Document Object Model) vilket gör att vi inte avser använda vissa ofta vanliga funktioner som exempelvis document.write, annat än i exempel där dessa kan kortas ner om en enklare funktion används.

I vissa fall så kommer vi att frångå W3Cs DOM-modell för att sidor ska kunna fungera i den populära webbläsaren IE (Internet Explorer från Microsoft). Denna följer inte alltid W3C-standard men har så stora marknadsandelar att vi inte kan ignorera att skriva sidor som fungerar i IE.

Som nämnt i tidigare material så har JavaScript sin början hos Netscape 1995. Det gick under flera namn till en början och kallades bland annat LiveScript och skapades för att vara ett enklare och mer lättillgängligt alternativ till Java-applets. Påtryckningar från marknadsföringen gjorde att namnet ändrades till JavaScript i slutänden vilket inte har gagnat språket. I början så uppstod ofta förvirring då Java och JavaScript misstogs för varandra och hos nybörjare på programmering så sker detta ibland än idag. JavaScripts stora tillgänglighet har dock gjort det till ett av de mesta använda språken på Internet idag. Den stora styrkan ligger i att det inte behöver någon kompilator och att kod kan kopieras från ett ställe till ett annat med få ändringar eller inga.

Internet Explorer fick snart sitt eget språk JScript som är mycket lik JavaScript. Men då Netscape var den dominerande webbläsaren när dessa språk introducerades så kom JavaScript att vinna striden som det mest använda språket. JavaScripts stora spridning gjorde att man överlät den vidare utvecklingen till ECMA som en standard 1996. Det officiella namnet är alltså ECMAScript eller ECMA-262[1] men det är sällan dessa namn används utan namnet JavaScript har stått sig på gott och ont. Tyvärr har inte försöket att standardisera JavaScript lyckats till så sett att olika webbläsare utökar språket med sina egna bibliotek och något som fungerar i en typ av webbläsare fungerar kanske inte i en annan. Netscapes JavaScript 1.3 och 1.5 skall vara kompatibla med ECMA-262 men utökar denna. Formellt sett så handlar det fortfarande om JScript i IE även om detta ofta kallas för JavaScript där också. Microsoft är tyvärr inte helt kompatibla med ECMA-262. För att se vad som ingår i grunden för JavaScript så använd fotnot 1 nedan och titta på dokumentationen för ECMA-262.

JavaScripts styrka är alltså lättheten med vilket det infogas i HTML och att det inte behövs någon kompilator. Svagheter hos JavaScript ligger i bristen på bra utvecklingsverktyg och den stora flora av extra bibliotek som är knutna till en viss webbläsare som gör att det svårare att skriva skript som fungerar i alla typer av webbläsare. Det har dock kommit flera utvecklingsverktyg som kan hantera JavaScript på senare år men det är fortfarande få och begränsade om man jämför med traditionella programmeringsspråk. Även de bibliotek av JavaScript-funktioner som kommit de senare åren är bättre än tidigare. Så utvecklingen är positiv för tillfället. Det finns idag versioner av JavaScript som exekveras på serversidan men för detta ändamål kommer vi på kursen att använda det vanligare PHP.

[1] http://www.ecma-international.org/publications/standards/Ecma-262.htm

Det som oftast lättsamt benämns JavaScript är egentligen tre delar: ECMA-262 som är själva grunden i språket. Till detta kommer modeller för hur en webbsidas HTML-element är uppbyggda, DOM (document objekt model) samt funktioner för webbläsaren BOM (browser objekt model). DOM regleras som standard av W3C medan BOM ligger helt utanför någon standardisering. Detta gör att de största skillnaderna mellan olika webbläsare oftast hittas i BOM. Vi kommer att titta närmare på DOM och BOM i kommande studiematerial.

Java och JavaScript
På grund av likheten i namnen och att bägge språken går att kombinera med webbsidor så händer det att JavaScript förväxlas med programmeringsspråket Java. Det är dock stora skillnader mellan dessa.

- Java - fristående programmeringsspråk
  - applets på webbsidor
  - oberoende applikationer
  - kompilerande (bytekod)
  - plattformsoberoende (JVM)

- JavaScript
  - måste bäddas in i HTML-kod, eller refereras till från HTML-kod.
  - Interpreterande (tolkande)
  - Skriptspråk som utvidgar HTML/CGI
  - flyttar jobb till klienten (minskar kommunikationsbehov)
  - begränsad funktionalitet
  - körs i webbläsaren

Med plattformsoberoende menas att till exempel ett Java-program utvecklat på en Macintoch kan köras på en Linux-dator, under förutsättning att en jämförbar JVM (Java Virtual Machine) finns installerad på den senare. Detta är möjligt tack vare den JVM som tolkar koden som generas av Java-kompilatorn.

JavaScript måste bäddas in i HTML-koden eller refereras till från koden om man väljer att skapa en särskild script-fil där man lägger funktioner med mera (mer om det längre fram).

För den som efter denna kurs vill lära sig mer om programmering med kompilerande språk så finns grundläggande programmeringskurser där du bland annat kan välja en kurs som använder Java.

JavaScript – egenskaper
JavaScript är objektbaserat och använder en objektorienterad modell. Du kan skapa egna objekt i JavaScript men JavaScript använder sig inte utav klassiska klass-baserade objekt utan av så kallade prototyp-baserade objekt. I klass-baserade objekt så skapas en klass som bestämmer vilka egenskaper och metoder en klass har och därmed alla instanser av denna klass. I prototypbaserade objekt så används en mall, en prototyp, för att skapa ett nytt objekt men detta objekts egenskaper kan sedan ändras oberoende av andra objekt som skapats av samma prototyp. I JavaScript finns det alltså inte instanser av en viss prototyp eller klass som ger en garanti om vissa egenskaper eller metoder utan bara objekt med olika, ibland samma, egenskaper och metoder. Vi kommer att återkomma till objekt i JavaScript senare.

I JS finns det ett stort antal inbyggda (fördefinierade) objekt som medger enkel åtkomst av text och resurser i webbläsaren.

Till skillnad från Java (som är kompilerat), så är JavaScript ett tolkat (interpreterat) språk. Tolkningen görs alltså när programmet (kallas ibland skript) körs. Detta medför att det är svårt att felsöka JavaScript. Att JavaScript dessutom fungerar lite olika i olika webbläsare gör inte heller saken lättare. Ett av de enklare verktygen för att felsöka JavaScript-kodär den inbyggda Error Console som du hittar i standardinstallationen av webbläsaren Mozilla Firefox 2+ i Tools-menyn (JavaScript Console i FF 1.5). Om du inte redan har Firefox så rekommenderas du att ladda hem denna och använda dig av Error Console när du testar JavaScript. Här kan du se eventuella fel som uppstod när skriptet kördes. Detta kan vara väldigt användbart då JavaScript upphör att exekvera när ett fel uppstår och det är väldigt bra ett veta var detta fel uppstod så man slipper leta efter något fel i hela skriptet.

Om du har en nyare version av Firefox så har du också programmet Venkman[2] inbyggt som du kan använda för att debugga JavaScript. För att hitta Venkman så väljer du Tools -> JavaScript Debugger. En annan vanlig debugger är Firebug.

JavaScript har även begränsad funktionalitet (saknar t.ex. filhantering) om man jämför med Java och andra traditionella programmeringsspråk. Regler om datatyper samt hur man får hantera dessa och vad man inte får göra är mycket lösare i JavaScript än i de flesta traditionella programmeringsspråk. Detta innebär att du kan använda variabler utan att deklarera dem och utan att ange deras typ. En variabel kan till och med byta typ under programmets gång. JavaScript är dock case sensitive, vilket innebär att det görs skillnad mellan små och stora bokstäver. Variablerna summa och Summa är alltså två skilda variabler.

Script-taggen och inkapsling av kod

JavaScript kan infogas på två sätt i HTML-koden via script-taggen: Antingen genom att man använder sig av src-attributet och anger en separat fil som innehåller JavaScript-koden eller genom att man skriver JavaScript-koden mellan start och sluttaggen för script-taggen. Utöver de två sätten via script-taggen så kan JavaScript infogas dirket i HTML-koden på vissa ställen i form av värden till attribut.

För mindre skript så kan den första varianten med separat fil kännas besvärlig MEN det är bara att vänja sig vid detta. Med införande av XHTML så blir det svårare att dölja JavaScript-koden för webbläsaren så denna inte visas som text eller försöks tolkas som XHTML.

Script-taggen är den tag som anger att det som följer är ett skript som skall tolkas av webbläsaren. För att webbläsaren skall veta hur den skall tolka skriptet så måste man ange vilken typ av script det är i attributet type.

```
<script type="text/javascript">
```

De flesta webbläsare klarar av denna tag idag och att tolka scriptet eller att ignorera scriptet. När JavaScript började dyka upp så kunde inte alla webbläsare tolka det. Skulle det vara så att webbläsaren inte kunde tolka skriptet så dolde man skriptet genom att göra en bortkommentering. Det ser ut så här:

```
<script type="text/javascript">
       <!--
         Här skriver du koden
       // -->
</script>
```

Observera JavaScript-kommentaren framför den avslutande html-taggen för kommentar.

[2] http://www.mozilla.org/projects/venkman/

Om du bara skriver:

```
<script type="text/javascript">
       <!--
         Här skriver du koden
       -->
</script>
```

så kommer du att få ett **fel i ditt JavaScript**.

Denna typ av bortkommentering används i många fall fortfarande av ren slentrian och ni bör känna till den så ni känner igen den om ni stöter på den. Det är dock inte många webbläsare i dag som inte klarar av att hantera innehållet i en script-tagg korrekt så denna typ av bortkommentering är idag överflödig.

På sikt kan den gamla bortkommenteringen av skriptet dessutom komma att ställa till problem när fler webbläsare klarar av att tolka XHTML korrekt och allt fler webbsidor är skrivna med XHTML. Med en sträng XHTML tolkning så ska allt mellan taggarna för kommentaren ignoreras av webbläsaren och detta kan få till följd att skriptet inte läses in till interpretatorn och helt enkelt aldrig körs.

Behovet av bortkommentering har dock inte försvunnit. Som nämndes i studiematerial 2 så kan kombinationen XHTML och JavaScript orsaka en del problem. Om du använder vanlig HTML så tolkas texten mellan starttaggen för scriptet och sluttaggen som CDATA (charachter data) och ignoreras av webbläsaren. **Så, så länge du använder vanlig HTML så är det inga problem.**

Men om du använder XHTML så tolkas texten mellan starttaggen och sluttaggen för skriptet som PCDATA (parsed character data) och webbläsaren försöker därför tolka innehållet vilket kan leda till fel. Så om du vill skriva korrekt XHTML så måste du bortkommentera ditt script för detta. Vi löser detta genom att tala om för webbläsaren att de som står i skriptet skall tolkas som CDATA:

```
<script type="text/javascript">
  <![CDATA[
  här skriver du koden
  ]]>
</script>
```

Detta ger ju dock problemet att JavaScriptet nu inte är korrekt. Detta kan vi rätta till genom att tecknen för en kommentar (presenteras lite längre ner) i JavaScript inte orsakar fel i validatorn. Så vi kapslar in instruktionerna till validatorn i varsin blockkommentar:

```
<script type="text/javascript">
  // <![CDATA[
  här skriver du koden
  // ]]>
</script>
```

Om vi dessutom skall ha kvar den gamla bortkommenteringen av skriptet blir det:

```
<script type="text/javascript">
  <!--//--> <![CDATA[//><!--
  här skriver du koden
  //--><!]]>
</script>
```

Den absolut enklaste lösningen är att använda ett externt skript. Du skriver då din kod i en separat fil och talar om vad den fil som skriptet skall hämtas ifrån heter. För att infoga JavaScript-kod som finns i en separat fil så använder du attributet src.

```
<script type="text/javascript" src="mitt_javascript.js"></script>
```

Det är ofta smidigt att inkludera JavaScript-koden från en fil även om man använder vanlig HTML då HTML-koden blir mer lättläst om man slipper bryta den med en massa skript-kod. Eftersom det inte finns någon kod mellan script-taggarna så finns det inget validatorn kan hänga upp sig på om du använder XHTML. Att använda externa skript stöder dessutom filosofin om 3-lagersprincipen.

För webbläsare som inte kan tolka scriptet så kan du använda noscript-taggen för att visa att där är något som inte blir som det är tänkt. Detta kan även vara bra att göra då man kan stänga av stödet för JavaScript i de flesta moderna webbläsare och man då kan påpeka detta för användaren.

```
<noscript><p>Inget stöd för JavaScript finns.</p></noscript>
```

I kursen så kommer vi senare att titta på hur du även undersöker specifik funktionalitet hos webbläsarens implementering av JavaScript för att kunna anpassa sidor after detta.

Vi kommer i nästa studiematerial att ta upp DOM – Document Objekt Model där vi i stort kommer att hålla oss till W3C-standard. Detta får dock effekter redan nu när vi ska använda ett första exempel. De vanliga första exemplen på JavaScript brukar använda document.write för att skriva ut något på en HTML-sida. Write-metoden är dock inte en del av W3Cs standard men implementeras som en del av DOM både hos IE och hos Mozilla. Så därför så kommer här ett lite annorlunda exempel som använder DOM:

HTML-filen:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dts">
<html>
  <head>
    <title>Exempel 1</title>
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-
1">
  </head>
  <body>
    <h1>Webbprogrammering exempel 1 med extern fil med JavaScript</h1>
    <div id="main_text">
      Lite text som redan står här.<br>
    </div>
    <script type="text/javascript" src="ex1.js"></script>
    <noscript><p>Inget stöd för JavaScript finns.</p></noscript>
  </body>
</html>
```

JavaScript-filen:

```
var new_text = document.createTextNode('Text infogad med JavaScript och
DOM');
document.getElementById('main_text').appendChild(new_text);
```

Koden ovan finns i filen exempel_1.html och ex_1.js. Du behöver inte förstå den exakt nu utan det är ett exempel på hur du bör strukturera din skriptkod i förhållande till HTML-koden. Pröva att stänga av stödet för JavaScript i din webbläsare och pröva exemplet utan detta stöd.

Kommentarer och indentering

Det är viktigt att du använder dig av indentering och kommentarer i din kod för att göra den mer lättläsligt för andra och dig själv.

Kommentarer kan skrivas på två sätt:

- radslutskommentarer som inleds med // och sträcker sig till slutet av raden
- blockkommentarer som inleds med /* och avslutas med */

```
//radkommentar

/* blockkommentar som kan stäcka
   sig över flera rader */
```

Indentering innebär att du strukturerar din kod med indrag så indelningen blir lätt att följa med blicken. Exempelvis så bör du indentera koden i en funktion ett steg i förhållande till funktionshuvudet och kod i loopar ett steg i förhållande till loopens första rad. Exempel på indentering kan du se i de olika exemplen till studiematerialet.

Variabler

Datatyper

Som nämnt tidigare så är JavaScript löst typat. Typning innebär att en variabel måste tillhöra en speciel datatyp exempelvis ett heltal, ett flyttal eller text. I hårt typade språk så måste en variabel deklareras som en viss datatyp och måste innehålla just denna datatyp. I JavaScript så är det inte så. En variabel behöver inte deklareras och är inte knuten till en viss datatyp. Variabelns datatyp beror på vad du tilldelat den. I JavaScript kan du till exempel göra följande:

```
lion = 5;
lion = "Leo Lejonsson";
lionFood = "Igår åt Leo" + 3 + "tigrar.";
//lionFood innehåller nu texten "Igår åt Leo 3 tigrar."
```

Om du vill så kan du deklarera en variabel genom att skriva var framför

```
var lion = 7;
```

Av tydlighetsskäl kan det vara bra att deklarera en variabel första gången man använder den så man uppmärksammar att det är en ny variabel.

Variabelnamnet är inte variabeln i sig utan en referens till den plats i minnet där variabelns data lagras.

JavaScript har följande primitiva (inbyggda) datatyper:

- boolean – kan anta ett av två värden sant eller falskt representerade av true eller false
- tal – i form av heltal eller flyttal
- undefined – en variabel utan ett tilldelat värde och därför odefinerad typ
- strängar – text
- arrayer eller vektorer – en samling av flera element av samma eller olika datatyp

I detta sammanhang så bör också null-värdet nämnas. Null innebär ett värde som inte innehåller någonting. Dock inte att förväxla med värdet 0 för siffror.

Variabelnamn

Variabelnamn får inte inledas med en siffra utan måste inledas med en bokstav eller underscore (_). Variabler är dessutom begränsade till det engelska alfabetet och kan därmed inte innehålla å, ä och ö. Variabelnamn får inte heller innehålla mellanslag eller kommatecken (,) eller punkt (.). Observera återigen att JavaScript är case sensitive.

```
Lion = 4;
lion = 6;
/*Lion och lion är två skilda variabler som innehåller talet 4 respektive
6 */
```

Som regel så brukar man namnge variabler med gemener och använda versaler till att skilja ord inom variabelnamnet åt eller använda underscore mellan olika ord.

```
nbrOfLionTails = 1;
lionMane = ”gul”;
```

eller

```
nbr_of_lion_tails = 1;
lion_mane = ”gul”;
```

Välj vad du föredrar och var konsekvent i den stil du använder för att namnge dina variabler.

I standarden för JavaScript så finns inga konstanter. Det är dock vanligt att man använder variabler som man inte avser ändra som konstanter. För att markera att dessa inte skall ändras så brukar de namnges med versaler.

```
LIONTAILS = 1;
nbrOfTailsForLeo = LIONTAILS;
```

Reserverade ord och escapade tecken

Reserverade ord har en speciell betydelse i ett språk. Reserverade ord kan inte användas som variabelnamn då dessa är just reserverade för en viss betydelse. Exempel på reserverade ord är var, function och if. En komplett lista för reserverade ord enligt Ecma-262-standarden[3] hittar du hos Ecma. Sedan kan varje webbläsares implementation av denna standard ha ytterligare reserverade ord så det kan vara bra att ha detta i åtanke när man felsöker sin kod.

Även vissa tecken har också speciell betydelse och kan inte användas direkt. Exempel på dessa är backslash (\) och citationstecken("). Om man vill skriva dessa tecken i en text-sträng så säger man att man escapar tecknet genom att skriva ett backslash-tecken framför.

Exempelvis:

```
lionMenu = "tiger\\puma\\huskatt"
/* ger textsträngen tiger\puma\huskatt vid utskrift*/
leoLikes = "\"huskatt är gott men lite lite mat\""
/* ger textsträngen "huskatt är gott men lite lite mat" vid utskrift*/
```

[3] http://www.ecma-international.org/publications/standards/Ecma-262.htm

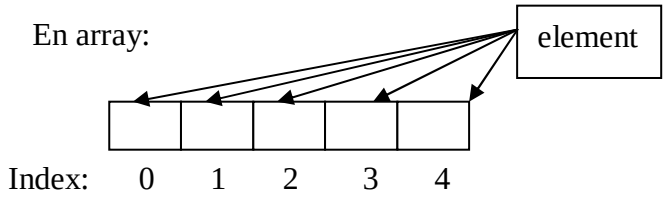
Nedan följer en liten lista över tecken som har en escapesekvens och som kan vara bra att känna till.

escapesekvens	unicodetecken	betydelse
\b	**\u0008**	backspace
\t	**\u0009**	horizontal tab
\n	**\u000A**	line feed (new line)
\v	**\u000B**	vertical tab
\f	**\u000C**	form feed
\r	**\u000D**	carriage return
\"	**\u0022**	double quote "
\'	**\u0027**	single quote '
****	**\u005C**	backslash \

Arrayer

Kärt barn har många namn; array, vektor, fält, fältvariabel, matris är alla namn för samma sak. Egentligen borde studiematerialet kanske använd ett svenskt uttryck, exempelvis fält eller vektor. Men vektor används huvudsakligen för endimensionella arrayer/fält och uppgifter kommer att arbeta med fler dimensioner än så under kursens gång. Nackdelen med ordet fält är att den direkta engelska översättningen field är något helt annat. Så därför så kommer studiematerialet att använda ordet array direkt från engelskan för det blir mindre risk för missförstånd då.

En array är en samling av variabler som har ett namn. Man säger att en array har flera element, där varje element är en variabel i sig. Varje element är förknippat med ett nummer och man hittar ett element genom att hänvisa till arrayens namn och sedan elementets nummer. Detta nummer kallas för index.



Arrayen ovan har 5 element med index från 0 till 4. Indexeringen börjar på 0 för första elementet och följaktligen så får det sista elementet index (antal element -1). Värdet i ett element nås genom notationen variabelnamn[index].

```
var myGames = new Array(4); //ny array med 4 element numrerade 0-3
//elementen har nu inga värden utan vi måste fulla dem med något
myGames[0] = ”Spacehulk”;
myGames[1] = ”Red Alert”;
//element 0 och 1 har nu värden medan 2 och 3 inte är definierade
var arrayLength = myGames.length;
//variabeln arrayLength har nu värdet 4
```

Den inbyggda funktionen length för arrayer är mycket nyttig.

I JavaScript så kan en arrays element vara skilda typer. Du kan alltså blanda boolean med strängar med tal eller till och med andra arrayer. Vi kan alltså göra följande:

```
var myGames = new Array(4); //ny array med 4 element numrerade 0-3
//elementen har nu inga värden utan vi måste fulla dem med något
myGames[0] = ”Spacehulk”;
myGames[1] = 2000; //året jag köpte spelet
myGames[2] = ”Red Alert”;
myGames[3] = 2003;
//vi har nu blandat tal och strängar i arrayen
```

Operatorer

I JavaScript finns ett antal operatorer.

Aritmetiska operatorer

Aritmetiska operatorer:
+ addition Används även som strängoperator för att konkatenera två strängar
- subtraktion
* multiplikation
/ division
% modulus Representerar resten vid heltalsdivision 7%5=2, 10%2=0
-- decremental Antag x=5; x--; x har nu värdet 4.
++ incremental Antag x=5; x++; x har nu värdet 6.

Tilldelningsoperatorn är ett likhetstecken (=) och det gäller att inte förväxla detta med en jämförelse vars operator består av två eller tre upprepade likhetstecken.

Det går att kombinera tilldelningsoperatorn med de aritmetiska operatorerna och få följande tilldelningsoperatorer:

operator	**exempel**	**motsvarar**
+=	x+=y	x=x+y
-=	x-=y	x=x-y
*=	x*=y	x=x*y
/=	x/=y	x=x/y
%=	x%=y	x=x%y

Jämförelseoperatorer

Jämförelseoperatorer jämför två uttryck eller variabler:
== likhet (två likhetstecken) 5==8 returnerar false

=== likhet för både värde och typ(tre upprepade likhetstecken) Antag x=5 och y="5" x==y returnerar true medan x===y returnerar false

!= skiljt från 5!=8 returnerar true

> större än 5>8 returnerar false, 8>5 returnerar true

< mindre än 5<8 returnerar true

>= större eller lika med 5>=8 returnerar false

<= mindre eller lika med 5<=8 returnerar true

Logiska operatorer
För att de logiska operatorerna skall fungera så måste värdena på bägge sidorna vara logiska uttryck, det vill säga uttryck som kan utvärderas som sant eller falskt.

! negering (!true) får värdet false, utläset som icke eller not på engelska.

|| eller ((3==5) || (4==4)) ger true, ((3==5) || (4==7)) ger false

&& och ((3==5) && (4==4)) ger false, ((3==3) && (4==4)) ger true

Bitvisa operatorer
Bitvisa operatorer opererar på de enstaka bitarna i ett värde.

& bitvis och 0xF & 0xA = 0xA

| bitvis eller

^ bitvis XOR (exklusivt eller)

bitvis NOT (inverterar alla bitar)

I det första exemplet för bitvist och så använde hexadecimal notation. Binärt blir detta:
```
 1111
 1010
 ------
 1010
```

På samma sätt för övriga de övriga bitvisa operatorerna. Exklusivt eller innebär att de två operanderna ska vara olika för att uttrycket ska vara sant.

Exempel
Man vill till exempel kontrollera om en bok innehåller minst 1000 sidor *och* kosta högst 500 kr.

Bok innehåller minst 1000 sidor, variabeln sidor innehåller antalet sidor: *sidor >= 1000*
Bok kostar högst 500 kr, variabeln pris innehåller bokens pris: *pris <= 500*
Uttrycket för att kontrollera det hela är: *(sidor >= 1000) && (pris <= 500)*

Man vill kontrollera om en bok innehåller kostar högst 300 kr *eller* om den finns i minst 3 ex på biblioteket. (Mer pengar än 300 kr kan man inte betala och om boken finns i färre än 3 exemplar så brukar den vara utlånad. Frågan är om boken blir läst inom den närmsta framtiden)

Bok kostar högst 300 kr, variabeln pris innehåller bokens pris: *pris <= 300*
Antalet böcker med rätt titel finns i variabeln antal: *antal>=3*
Uttrycket för att kontrollera om boken blir läst är: *(pris <= 300)* || *(antal>=3)*

Uttryck med logiska operatorer beräknas från vänster till höger. Det får vissa följder:

Om *(sidor>=1000)* är *false* i uttrycket *(sidor >= 1000) && (pris <= 500)* så beräknas aldrig *(pris <= 500)*. Det behövs ju inte, uttrycket blir ju i vilket falls om helst *false* .

Om *(pris <= 300)* är *true* i uttrycket *(pris <= 300)* || *(antal>=3)* så beräknas aldrig *(antal>=3)*. Det behövs ju inte, uttrycket blir ju i vilket falls om helst *true* .
Exempel på villkor:

villkor	*kodning*
tal större än 67	tal > 67
tal är 3, 6 eller 7	(tal == 3) || (tal == 6) || (tal == 7)
tal är i intervallet 25-50	(tal >= 25) && (tal <= 50)
tal är delbart med 3	(tal % 3) == 0
tal är negativet eller större än 10	(tal < 0) || (tal > 10)
tal är ej delbart med 4	(tal % 4) > 0
	(tal % 4) != 0
	!((tal % 4) == 0)
entalssiffran i tal är 7	(tal % 10) == 7
tal tillhör något av intervallen 1-4 eller 7-9	((tal >=1) && (tal <= 4)) || ((tal >=7) && (tal <= 9))
hundratalssiffran i tal är 7	((tal / 100) % 10) == 7

Algoritm

En algoritm beskriver steg för steg hur ett problem eller en uppgift kan lösas. Varje gång man ska göra ett program så måste man börja med att arbeta med algoritmer som löser problemet eller uppgiften i programmet. Man kan beskriva sin algoritm på olika sätt, till exempel genom pseudokod eller aktivitetsdiagram.

Antag att vi skall göra ett program där man matade in hur mycket man skulle spara varje vecka och hur många veckor man skulle spara och sedan räknar ut kapitalet. Om man beskriver detta exempels algoritm så skulle den kunna se ut ungefär så här:

pseudokod

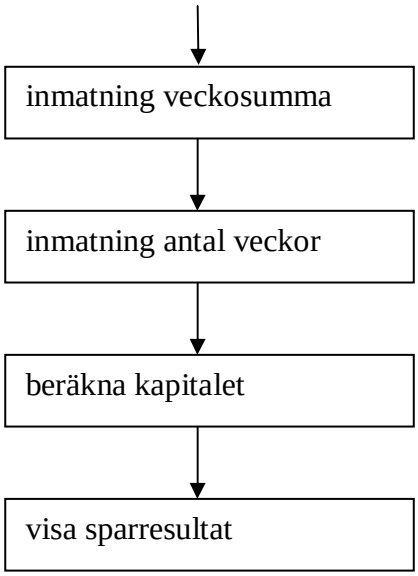
```
deklarera variabler(sparande,veckor,kapital)
inmatning av veckosumma från användaren

inmatning av antal veckor från användaren

beräkna kapital

visa sparresultat
```

aktivitetsdiagram



När man är klar med en algoritm och har försäkrat sig om att den verkar korrekt så implementerar man den i något språk. Stegen i algoritmen kan vara av olika upplösning. Ibland motsvarar ett steg en instruktion, ibland flera instruktioner och ibland en hel metod. Upplösningen väljs beroende på hur komplicerat och välkänt problemet är. Ju mer komplext och mindre känt problem desto högre upplösning krävs och motsatt att om problemet är välkänt och inte så komplext så räcker det med en lägre upplösning.

De klassiska grundstenarna när man utformar en algoritm är

- **Sekvens** - en följd av instruktioner. (exemplet ovan)
- **Selektion** - en sekvens som utförs under vissa betingelser. Man formulerar villkor för att sekvensen ska utföras.
- **Iteration** - en upprepning av en sekvens. Villkor i konstruktionen bestämmer hur många gånger sekvensen ska upprepas.

När man tycker att implementeringen av en algoritm är färdig är det dags att testa den. Det är inte ovanligt att testet visar att det finns brister i algoritmen som måste åtgärdas. Man är nästan tillbaka vid ruta ett det vill säga att skapa en ny (modifierad) algoritm, vilken ska implementeras och testas.

Det är viktigt att man testar varje implementering (kodning) av en algoritm innan implementeringen (kodningen) används i programmet. Genom att varje algoritm testas först och när den väl är felfri infogas i programmet så har man som programkonstruktör kontroll över programutvecklingen.

Exempel: gå och fylla hinken med sand

```
1 Gå med hinken till sandhögen
  2 Ta sand i spaden
  3 Om det är för mycket sand i spaden så
    3a Häll ut lite sand
  4 Annars om det är för lite sand i spaden så
    4a Fyll på lite sand
  5 Häll sanden i hinken
6 Om hinken ej är full så fortsätt med punkt 2.
7 Gå tillbaka med hinken
```

Sekvens: Raderna 1-7 beskriver en sekvens.

Selektion: På raderna 3 och 4 finns villkor för vad vi skall göra för att det skall bli lagom med sand på spaden. Vi kontrollerar ett villkor för mängden sand på spaden. Först kontrollerar vi om det är för lite sand, om det var för lite sand så tar vi mer sand och häller sedan sanden i hinken. Om det inte var för lite sand på spaden så kontrollerar vi om det var för mycket sand på spaden. Om det var för mycket sand så häller vi av lite och tömmer sedan spaden i hinken. Om det inte var för lite eller för mycket sand på spaden så tömmer vi direkt spaden i hinken.

Iteration: På rad 6 kontrollerar vi om hinken är full. Om den inte är det så börjar vi från rad 2 igen. Villkoret på rad 6 avgör om vi skall upprepa rad 2-6 igen eller om vi skall gå vidare till rad 7.

Villkorshantering

I JavaScript, som med all annan programmering, är det viktigt att kunna göra olika saker beroende på utgången av villkor. Man gör en selektion.

If-satsen

if-satsen fungerar så här

```
if(uttryck)//uttrycket måste gå att utvärdera som true eller false
   sats eller block av satser som utförs(sekvens)
```

I text skulle det kunna uttryckas som:
Om villkor är sant utför sekvens

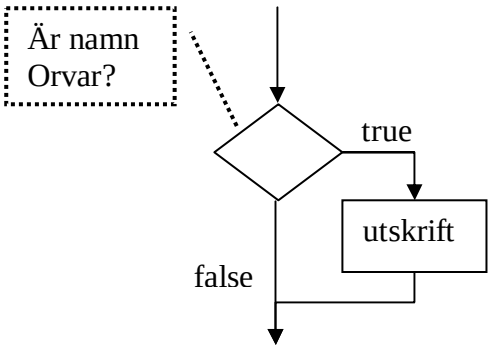
Uttrycket som utgör villkoret skall alltid kunna utvärderas till **true** eller **false**. Med hjälp av jämförelseoperatorerna och de logiska operatorerna bygger man denna typ av uttryck.

Exempel if 1, gratulation:
Vi har ett program som jämför två strängar och om de är lika så skrivs en text ut. Om strängarna inte är lika så sker ingen utskrift.

pseudokod

```
användaren matar in sitt namn
OM inmatat namn är samma som namnsdagsnamnet
   skriv ut en gratulation
```

aktivitetsdiagram



Kod

```
var namn;
var meddelande;
namn = prompt("Ange ditt namn");
if(namn=="Orvar"){
  meddelande = "Grattis du har namnsdag idag!";
}
```

If-satsen utför en eller flera satser om villkoret är sant annars kan andra satser utföras genom användningen av else-satsen:

```
var namn;
var meddelande;
namn = prompt("Ange ditt namn");
if(namn=="Orvar"){
  meddelande = "Grattis du har namnsdag idag!";
}
else{
   meddelande = "Du har inte namnsdag idag.";
}
```

Man kan utelämna blockparenteserna,{ }, om villkoret endast innehåller en rad men du bör ändå använda dem då detta minskar risken för fel och gör din kod tydligare.

Genom att använda kombinationen else if kan man kontrollera, och skilja på, flera olika villkor.

```
if (villkor1){
  var new_text = document.createTextNode('Villkor 1 är sant');
}
else if (villkor2){
 var new_text = document.createTextNode('Villkor 2 är sant');
}
else if (villkor3){
 var new_text = document.createTextNode('Villkor 3 är sant');
}
else if (villkor4){
 var new_text = document.createTextNode('Villkor 4 är sant');
}
```

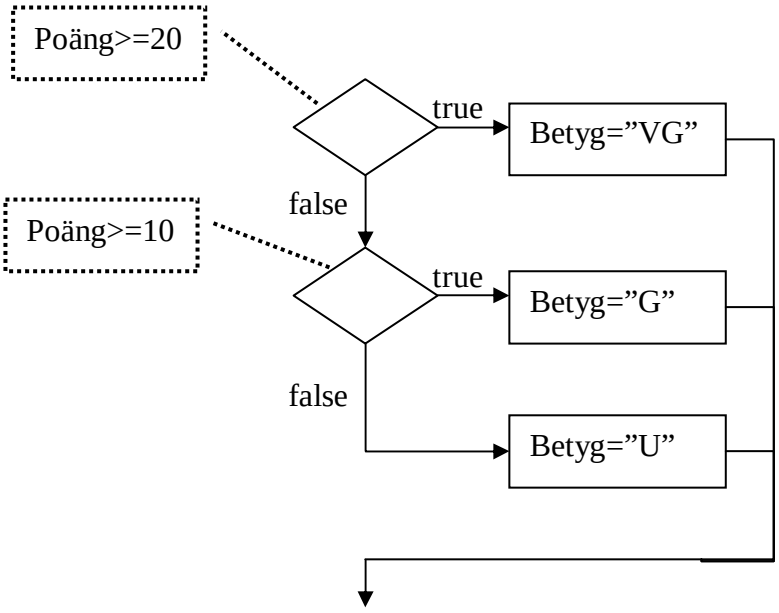
Om man vill kontrollera om flera villkor är sanna samtidigt kan man nästla dem:

```
if (villkor1){
  if(villkor2){
    if(villkor3){
      var new_text = document.createTextNode('Villkor 1-3 är sant');
     }
     else {
       var new_text = document.createTextNode('Villkor 1-2 är sanna men
inte villkor 3');
     }
   }
}
```

Exempel if 2, betyg visar på hur man kan utvärdera om en av fler alternativ än 2 gäller.

Pseudokod och aktivitetsdiagram

```
Eleven anger uppnådd poäng på skrivningen
Om poängen>=20 så
   Betyget är VG
Annars om poängen>=10 så
   Betyget är G
Annars
   Betyget är IG
Meddela eleven hennes betyg
```



Kod

```
resultat = prompt("Ange resultat");
if(resultat>=20){
  meddelande = resultat + " poäng ger betyget VG";
}
else if(resultat>=10){
  meddelande = resultat + " poäng ger betyget G";
}
else{
  meddelande = resultat + " poäng ger betyget IG";
}
```

Switch-satsen

Switch-satsen är ett sätt att välja satser baserat på en variabels värde.

```
betyg = prompt("Ange betyg");
switch (betyg)  {
  case "3":
    meddelande = 'Bra';
    break;
  case "4":
    meddelande = 'Jättebra';
    break;
  case "5":
    meddelande = 'Superbra';
    break;
  default:
    meddelande = 'Omtentamen går till Jul';
}
```

Finns det en begränsad uppsättning av förvalda värden som ska testas så kan switch-satsen vara ett bra alternativ. Som synes hanterar switchsatsen flera villkor i samma variabel.

Observera att man måste ange break efter varje alternativ. Om man inte anger break så fortsätter interpretatorn och utför nästa alternativ.
default: innebär att satsen utförs om inga av de andra villkoren är uppfyllda.

En switch-sats ersätts enkelt med en upprepad if-else. Jämför de två delarna nedan:

```
switch(val)
    case 1:  sekvens1;
             break;
    case 2:  sekvens2;
             break;
    case 3:  sekvens3;
             break;
    case 4:  case 8: case 9:
             sekvens4;
             break;
    default: sekvens5;
}
```

```
if(val==1) {
  sekvens 1
}
else if(val==2) {
  sekvens 2
}
else if(val==3) {
  sekvens 3
}
else if((val==4)||(val==8)||(val==9)){
  sekvens 4
}
else{
    sekvens 5
}
```

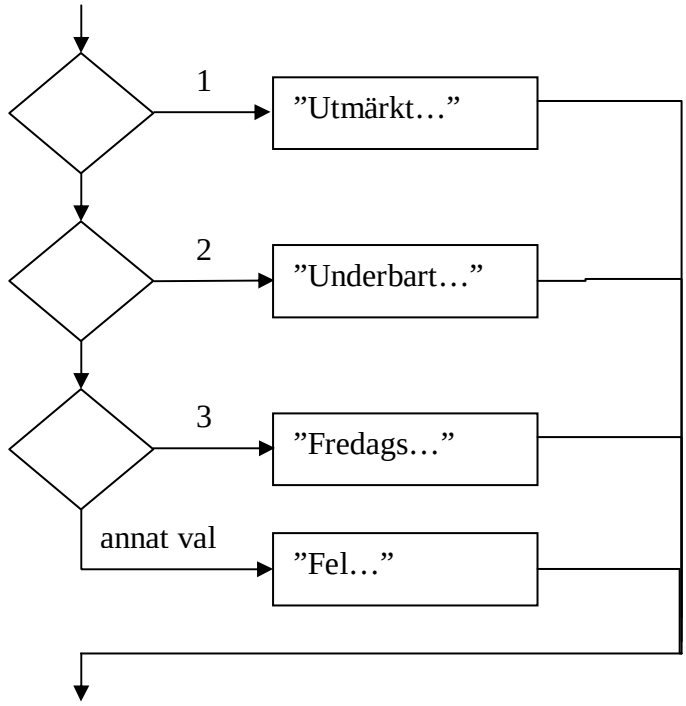
Exempel 2 swith-sats
Vi utgår från en students val klockan 15 en fredag. Möjliga val är:

1. Gå på föreläsning
2. Gå hem och läsa kurslitteratur
3. Gå till datorsal och jobba

Pseudokod

```
Ange ditt alternativ:
Om du valde 1 så
  Meddela ”Ett utmärkt val!”
Om du valde 2 så
  Meddela ”Underbart att läsa!”
Om du valde 3 så
  Meddela ”Fredagsfavoriten!”
Annars
  Meddela ”Du råkade trycka på en felaktig tangent!”
```

Aktivitetsdiagram



Kod

```
val = prompt('Ange vad du vill göra');
switch (val)  {
  case '1':
    meddelande = 'Ett utmärkt val!';
    break;
  case '2':
    meddelande = 'Underbart att läsa!';
    break;
  case '3':
    meddelande = 'Fredagsfavoriten!';
    break;
  default:
    meddelande = 'Du råkade trycka på en felaktig tangent!';
}
```

Iterationer

Iterationer (kallas även loopar) användas när man upprepar samma sats(er) flera gånger.
Det finns 4 sätt att göra detta på;

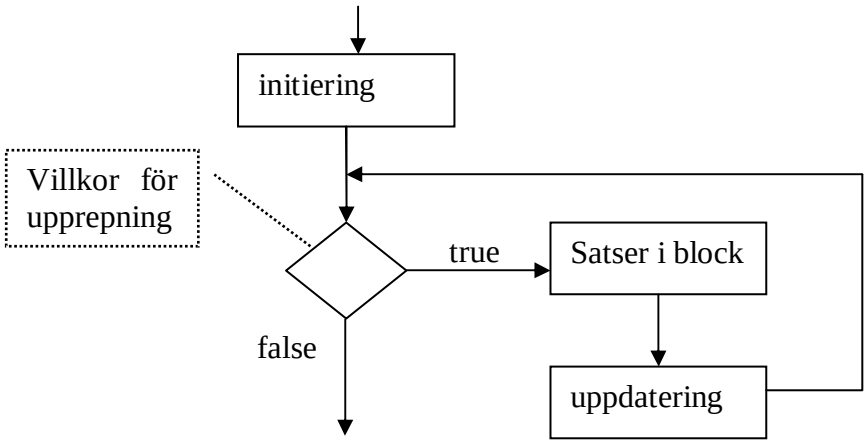
- for-loopen: Används när man vet exakt hur många gånger loopen ska köras.
- while-loopen : Används när man inte vet hur många gånger loopen ska köras.
- do-while-loopen: används när man vet om att en while-loop ska köras minst en gång.
- rekursiva funktioner

For-loop

En for-loop itererar från ett visst startvärde och ökar detta värde med ett visst antal steg i varje loop tills värdet når et slutvärde. Strukturen har följande utseende:

```
for(initiering; villkor för upprepning; uppdatering){
  block med kod som upprepas om villkor är sant
}
```

Detta kan illustreras med ett aktivitetsdiagram:



Initiering utförs från början. Detta sker endast en gång. Sedan beräknas villkoret. Blir villkoret true så utförs blocket av kod och därefter uppdateringen och villkoret beräknas igen. Skulle villkoret bli false så fortsätter programmet efter loopen.

Generellt så kan man säga att for-loopen används när något skall upprepas ett känt antal gånger och start- och slut-värdet är känt.

```
Exakt 20 upprepningar       Känt startvärde och slutvärde
for(i=0; i<20; i++)          for(i=startValue; i<=endValue; i++)
for(i=1; i<=20; i++)
```

Loop-variabeln eller räknaren, i fallen ovan i, förändras varje iteration. Den kan räknas uppåt eller neråt, och med olika antal steg. Vanliga loop-variabler är i, j, k och l. Men naturligtvis kan man ge loopvariabeln ett mer beskrivande namn också.

```
for(i = 0; i<15; i++){
  myArray[i]= i; //ger varje element samma värde som sitt index
}
for(count=10; count>0; count--) {
  meddelande1 += count + " "; //skriver 10 till 1
}
```

For/in–loopen
I JavaScript finn även en variant på for-loop som används för att iterera över element i ett objekt, exempelvis en array. Denna har strukturen:

```
for(variabel in objekt){
  block med kod som upprepas om villkor är sant
}
```

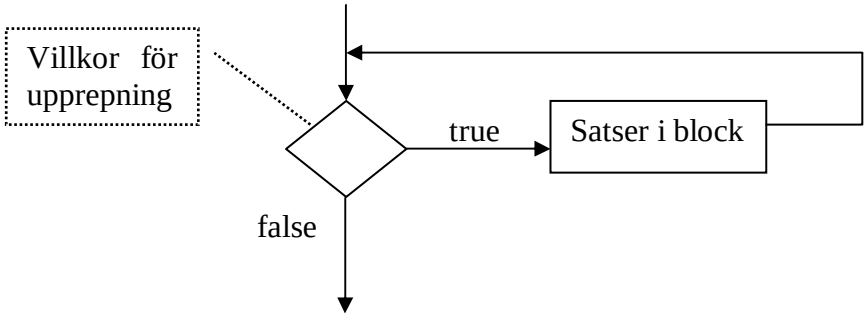
Detta innebär att man inte på förhand känner till antalet iterationer som görs.

```
for (i in myArray){ //för alla element i myArray oavsett antal
  // skriver ut varje elements värde
  meddelande2 += myArray[i] + " ";
}
```

While-loopen
While-loopen används när något skall upprepas men det är inte känt hur många gånger detta skall ske. While-loopen har den generella konstruktionen:

```
while(villkor för upprepning){
  block med kod som skall upprepas om villkor är sant
}
```

Aktivitetsdiagram:



Villkoret beräknas. Om villkoret blir true så utförssatserna i blocket och annars så fortsätter exekveringen efter loopen.

Upprepningen i en while-lop fortsätter alltså tills en viss händelse inträffar.

```
while(sparkapital<10000) Upprepa så länge sparkapitalet inte uppgår till
                         10000 kr.
while(krig==true)        Upprepa medan krig är sant. Kan användas av en
                         general,eller en förhandlare.
while(krig)              Samma som ovan, men kortare skrivet
while(true)              En loop som pågår för evigt
```

Koden i en while-loop behöver inte exekveras. Villkoret kan ju vara falskt redan vid första beräkningen. I exemplet med sparkapital ovan inträffar detta om sparkapital är t.ex.12000 kr från början.

En for-loop kan alltid ersättas med en while-loop och vice versa, även om det inte alltid är lämpligt.

```
                                    var i=0;
for(i=0; i<20; i+=2) {              while(i<20) {
  // kod                              // kod
}                                     i += 2;
                                    }

                                    var i=20;
for(i=20; i>0; i-=8) {              while(i>0) {
  // kod                              // kod
}                                     i -= 8;
                                    }

for(;sparkapital<10000;) {          while(sparkapital<10000) {
  // kod i vilken                     // kod i vilken
  //sparkapital förändras            //sparkapital förändras
}                                   }
```

Continue och break

```
while villkor1 {
  if (villkor2) continue;          // nytt varv
  satser;               // bör påverka villkor1
  if (villkor3) break;           // gå ur loopen
}
```

villkor1 måste påverkas av satserna inuti { } - uttrycket, annars hamnar programmet i en evig loop. Om vilkor1 är uppfyllt redan när loppen skall starta kommer inget att hända.

Uttrycket continue betyder att man påbörjar ett nytt varv utan att gå längre i de satser loopen innehåller.

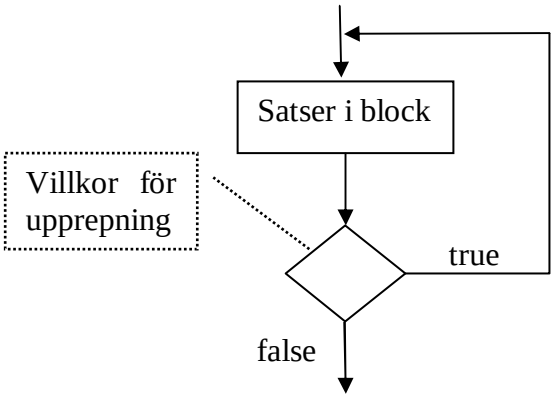
Uttrycket break betyder att man helt avbryter exekveringen av loopen och förflyttar sig utanför loopen och går vidare.

Do-loopen

Do-loopen är en variant av while-loopen och kallas ofta också do-while-loop. Do-loopen används som while-loopen men med skillnaden att satserna i loppen alltid skall utföras minst en gång när loopen startar. Konstruktionen har mönstret:

```
do{
  block med satser som
  upprepas om villkoret är sant
  samt alltid en gång
  i första iterationen
}while(villkor för upprepning)
```

Aktivitetsdiagram



Antag att användaren skall mata in ett tal. Om talet är positivt så skall ett nytt tal anges men om talet är negativt så skall inga fler tal anges. Vi vet inte på förhand hur många tal som skall matas in så en while-loop eller en do-loop är lämplig:

```
//while-loop
inputText = prompt("Ange ett heltal: ");
tal = parseInt(inputText);
while(tal>=0) {
   inputText = prompt("Ange ett heltal: ");
   tal = parseInt(inputText);
}

//do-loop
do{
  inputText = prompt("Ange ett heltal: ");
   tal = parseInt(inputText);
}while(tal>=0);
```

I do-loopen slipper vi två extra rader för att läsa in det första talet och därför så lämpar sig den bättre än while-loopen för detta exempel. I do-while-loopen så utförs alltid satserna i loopens kropp minst en gång då villkoret kontrolleras efter den första iterationen. Även här kan continue och break användas.

Rekursiva funktioner

En rekursiv funktion anropar sig själv och ger därför upphov till en loop. Det måste alltid finnas ett villkor inbäddat i funktionen som någon gång förhindrar anropet till sig själv så man inte hamnar i en oändlig loop. Ett exempel på en rekursiv funktion skulle kunna vara att räkna ut fakulteten av ett tal.

```
function fakultet(tal) //7-fakultet innebär matematiskt 1*2*3*4*5*6*7
{
   if (tal==0) return 1;
   else return tal*fakultet(tal-1); //här sker anropet till funktionen igen
}
```

Rekursiva funktioner kan ibland kräva lite tankegymnastik med ger ofta upphov till färre rader kod och exekverar snabbare än en while-loop eller for-loop. Skillnaden i hastighet blir dock inte märkbar annat än för mycket stora instanser.

Fakuletet skrivet med en while-loop:

```
function fakultet(tal)
{
  summa=1;
  while(tal!=0)
  {
   summa = summa*tal;
    tal--;
  }
  return summa;
}
```

Funktioner används för att strukturera kod och för att samla kod som kommer att utföras flera gånger så denna kod inte behövs skrivas om och om igen. En funktion kan anropas ett obegränsat antal gånger. Det finns fördefinierade och egendefinierade funktioner. Fördefinierade funktioner är funktioner som finns inbyggda i JavaScript eller i DOM.
Exempelvis så är funktionen alert en inbyggd funktion i DOM som du kan använda via JavaScript.

Egendefinierade funktioner

Egendefinierade funktioner skriver du, som det låter, själv. Den generella strukturen är:

```
function funktionsnamn(parametrar)
{
  kod som utför uppgifter eller dylikt.
}
```

Funktioner kan returnera ett värde genom att man använder en return-sats.

```
//här har vi vår egendefinierade funktion
function talIKubik(tal){
  return tal*tal*tal;
}

//här använder vi funktionen
var kubik = talIKubik(5);
```

Talet 5 sänds i som ett argument till parametern i funktionen talIKubik. Funktionen gör en beräkning och returnerar ett resultat som lagras i variabeln kubik. Resultatet är beroende på vad vi gav för argument.

Fördefinierade funktioner

Nedan följer några fördefinierade funktioner som kan vara till nytta.

De i JavaScript inbyggda funktionerna parseInt() och parseFloat omvandlar en sträng till ett heltal respektive flyttal.

```
var heltal = "10";
var decimaltal = "6.77";
var hexadecimaltal = "1A";
var dectal = parseFloat(decimaltal) //ger  värdet: 6.77
dectal = parseFloat(hexadecimaltal) //värdet: NaN   (Not a Number)
tal = parseInt(decimaltal) //ger värdet: 6
tal parseInt(hexadecimaltal,16) //ger värdet 26
```

När man skall hantera text kan de i JavaScript inbyggda funktionerna substring(), charAt() och split() komma till nytta. Antag att vi har:

```
var agentnamnet = ”Bond James”
```

Hitta de två första tecknen:

```
first = agentnamnet.substring(0, 2) // Bo
```

Hitta det andra tecknet:

```
secChar = agentnamnet.charAt(1) // o
```

Dela upp strängen:

```
parts = agentnamnet.split(” ”)
```

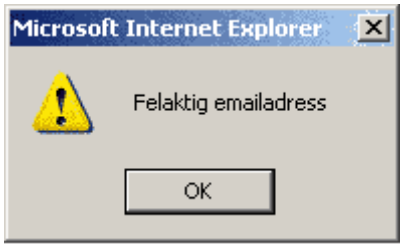
Ger parts[0] värdet Bond och parts[1] värdet James.

Första tecknet i strängen har alltid index 0 (noll). De tre första tecknen finns alltså i elementen 0-2. Det innebär också att sista tecknet i en sträng med 6 tecken har index 5. Tänk på detta, då det är ett mycket vanligt fel att glömma att antalet tal i en array inte är det samma som sista index.

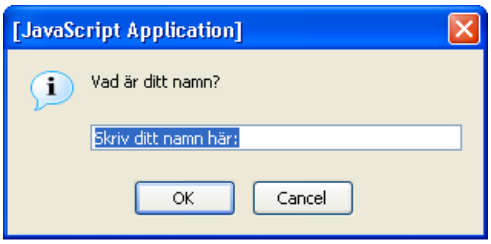
Observera den kraftfulla split-funktionen och dess resultat. Argumentet är alltså det tecken vi ska dela strängen på (i detta fall ett mellanslag). Om man delar upp en sträng med split-funktionen så returneras en vektor av strängar.

För att kommunicera med användaren så kan följande funktioner som finns inbyggda i DOM vara till nytta.

Alert-meddelande, en meddelande-ruta som skrivs ut med ett utropstecken. Användaren måste konfirmera genom att klicka på OK. Meddelandet stannar JavaScript-exekveringen till det att användaren kvitterat att denna läst meddelandet genom att trycka OK. Detta är ett sätt att varna användaren och skall inte utnyttjas i onödan.



Prompt-meddelande, en dialogruta som visas i webbläsaren där användare kan skriva in text. Texten kan sedan tas emot som en variabel. Om användaren avbryter så returneras null istället för en text.



Confirm-meddelande, en meddelande ruta framträder som skrivs ut med ett frågetecken, användaren måste trycka på OK eller Avbryt. Funktionen returnerar ett booleskt värde så om användaren trycker OK så returneras true och om användaren avbröt så returneras false.

Objekt

I JavaScript finns flera fördefinierade objekt som du kommer att stifta bekantskap med underkursens gång men du kan även själv skapa egna objekt. Objekt är ett sätt att samla ihop olika funktioner och egenskaper till sammanhängande enheter och är användbart när man behöver flera sådana enheter. En enhet skall givetvis vara logiskt sammanhängande på något sätt.

JavaScript kan alltså använda objekt och är då att betrakta som ett så kallat objektbaserat eller prototypbaserat språk till skillnad från objektorienterade språk. I objektorienterade språk har man en klass som utgör en mall för hur objekt ska se ut. Man skapar objekt genom att skapa instanser av klassen. Jämför med att använda en ritning till att bygga flera likadana hus. För att kunna skapa objekt behöver man först en klass. I JavaScript kallas klasser för objekt, och objekt kallas för objektinstanser men rent praktiskt är det samma saker. Därefter upphör dock likheterna.

Den stora skillnaden är att objektinstanser kan modifieras i betydligt högre grad i JavaScript än i mer strikta objektorienterade språk. Du kan exempelvis lägga till eller ta bort attribut och metoder hos objektinstanser. Detta får till följd att du aldrig är garanterad ett visst utseende hos en objektinstans av ett visst objekt. I exempelvis Java så är du garanterad att en instans av

en klass har ett visst beteende och egenskaper men med objektinstanser i JavaScript så finns inte detta krav.

Egenskaper hos objekt kallas ofta även attribut och representerar data som vi lagrar med objektinstansen. Objektets beteende bestäms av olika funktioner som vi kopplar till objektet. Dessa benämns då som ett objekts metoder.

Det finns flera sätt att skapa och hantera objekt på i JavaScript. Här presenteras några av de vanligare.

Objekt genom konstruktorn Object()
Om vi vill skapa **en** objektinstans som har vissa attribut och metoder så kan vi använda den allmänna konstruktorn Object() för att skapa en objektinstans som vi sedan kan ge attribut och metoder. En funktion som skapar ett objekt kallas för konstruktor. Konstruktorn Object() används för alla objekt i JavaScript men oftast osynligt. Senare i materialet så kommer vi att titta på hur man kan skapa egna konstruktorer.

Så för att skapa en tom objektinstans med namnet Cat så skriver vi:

```
var Cat = new Object();
```

Den gängse standarden är att man namnger objekt men stor inledande bokstav.

Vi vill nu lägga en egenskap till objektinstansen:

```
Cat.name = "Knas";
```

Objektinstansen med namnet Cat har nu ett attribut nemn namnet name och detta har fått ett värde i form av en textsträng.

För att ge objektinstansen en metod så gör vi följande:

```
Cat.meow = function()
{
  Alert("Mjau");
}
```

För att anropa metoden så skriver vi :

```
Cat.meow();
```

Om vi vill använda attributen i objektinstansen i en metod som kopplas till denna så använder vi det reserverade ordet *this*. This är en sorts inbyggd variabel som refererar till den aktuella objektinstansen.

Så en metod som använder egenskapen name hos objektinstansen Cat skulle kunna vara:

```
Cat.meow = function()
{
  alert("Mjau, jag heter "+this.name+".");
}
```

Så denna metod är bra om vi har orsak att behöva bygga upp en objektinstans över ett större segment kod efterhand. Om vi redan vet en del om vad objektinstansen ska innehålla så kan vi göra detta snabbare.

Literala objekt
Om vi på förhand vet en del om det som skall ingå den objektinstans vi vill skapa så kan vi skapa denna som ett literalt objekt. Med en literal avses ett fast värde - vi skapas alltså objektinstansen med vissa bestämda attribut och metoder direkt istället för att lägga till dem efter att själva objektinstansen skapats. Så det literala objektet för det tidigare exemplet skulle bli:

```
var Cat =
{
  name: "Knas",

  meow: function()
  {
    alert("Mjau, jag heter "+this.name+".");
  }
};
```

Det är denna syntax som studielitteraturen avseende JavaScript huvudsakligen använder. Nackdelen med denna syntax är att vi så att säga måste "börja om från början" om vi vill ha flera objektinstanser efter samma modell.

Konstruktorfunktioner
Om vi vill ha flera objektinstanser efter samma mall och inte vill upprepa kod så kan vi använda konstruktorfunktioner. Genom att en gång definiera en konstruktorfunktion så kan denna sedan användas flera gånger.

En konstruktorfunktion motsvarande de tidigare exemplen:

```
function Cat(name){
  this.name=name;
  this.meow = function(){
    alert("Mjau, jag heter "+this.name+".");
  }
}
```

Denna konstruktorfunktion har en parameter men detta är inte nödvändigt.

För att sedan använda den egenskapade konstruktorfunktionen så gör vi på motsvarande sätt som med den allmänna konstruktorfunktionen Object().

```
var Knas = new Cat("Knas");
var  Tass = new Cat("Tass");
Knas.meow();
Tass.meow();
```

Vi kan alltså skapa flera objektinstanser av samma mall men ändå ha olika värden på attributen.

Även de funktioner som utgör metoder kan ha parametrar:

```
function Cat(name){
  this.name=name;
  this.meow = function(){
    alert("Mjau, jag heter "+this.name+".");
  }
  this.changeName = function(newName){
    this.name = newName;
  }
}
var Knas = new Cat("Knas");
Knas.changeName("Kapten Knas");
```

Denna kurs kommer inte att gå in mer utförligt om objekt. För den som är intresserad av objektorienterad programmering i stort så finns en mängd andra kurser att läsa. Om du vill läsa mer nu om objekt i JavaScript så finns ett par länkar nedan att starta med.

Länkar

Standarder och referenser som det är nyttigt att lära sig hitta i:
http://developer.mozilla.org/en/docs/JavaScript
http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Guide
http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference
http://www.ecma-international.org/publications/standards/Ecma-262.htm

Information om JavaScript debuggern Venkman:
http://www.mozilla.org/projects/venkman/

Objekt i JavaScript
http://www.sitepoint.com/article/oriented-programming-1
http://www.sitepoint.com/article/oriented-programming-2

Lite mer lättsamma saker av intresse som du dock bör läsa:
http://javascript.about.com/od/reference/a/history.htm
http://javascript.about.com/od/reference/a/nqjavascript.htm
http://javascript.about.com/od/reference/a/cannot.htm
http://javascript.about.com/library/blxhtml.htm

Det finns en mängd guider och referenser till JavaScript på nätet. De som finns ovan är grunderna. Hittar du någon bra referens eller guide som ger ett pålitligt intryck så lägg gärna upp den i forumet för studiematerial.

Tillhörande kod-filer

I filen sm3.zip finns flera filer med exempel från studiematerialet som du bör titta på och köra. Modifiera gärna dessa filer efter egna idéer. Här finns även fler filer som är av intresse och som du bör studera.