Programmering 1

med JavaScript

För kurser i programmering 1

och Yrkeshögskolans preparandkurser (BFU)

Med övningar och
projektupppgifter

Förlag: Lieta AB

Titel: Programmering 1 med JavaScript

Författare: Taifun Alishenas
info@taifun.se

Tel: 073 - 757 70 69

Maj 2023

Kopieringsförbud!

Innehåll

Ämne	Sida	Program
Kapitel 1 **Introduktion till programmering**	**5**	
1.1 Om programmering	6	
- Algoritmiskt tänkande	7	
- Val av programmeringsspråk	8	
1.2 Programmeringens ABC	9	
- Interpretator vs. kompilator	9	
- Editorer	9	
- Om JavaScript	10	
- Att hantera filändelser	11	
1.3 Att komma igång med JavaScript	12	
- Programmet `Welcome`	12	`Welcome`
- Kommentar	13	
- Satser i JavaScript	13	
1.4 Konkatenering	15	`Concat`
- Överlagring	16	
1.5 Utskrift i flera rader	17	`Break`
- Radbrytning i utskriften med JavaScript	18	`Escape`
- Funktionen alert()	19	
- Escapesekvenser	19	
Frågor till kap 1	20	
Övningar till kap 1	21	
Kapitel 2 **Grundbegrepp i programmering**	**22**	
2.1 Variabler	23	`Variable`
- Vad är en variabel?	23	
- Tilldelningsoperatorn =	24	
2.2 Överskrivning eller kan x = x + 1 vara sant?	26	`Overwrite`
- Prioritet av operatorer	27	
- Tilldelning vs. likhet	27	
2.3 Inläsning av data	28	`Input`
- Funktionerna `prompt()` och `parseInt()`	29	
2.4 Arrays	30	`Arraydef`
- Arrayens initieringslista	33	`ArrayInit`
2.5 Hantering av slumptal	35	`Random`
- Slumptal inom ett intervall	36	
Övningar till kap 2	37	
Inlämningsuppgift	**38**	

Kapitel 3 **Kontrollstrukturer** **39**

3.1 Vad är kontrollstrukturer? 40
3.2 Enkel selektion: `if`-satsen 41 `SimpleIf`
- Villkor 42
- Jämförelseoperatorer 43
- Bestämning av max/min 44 `Max`
- Modularisering 45
- Funktionen max() 46 `MaxFct`
- Om funktioner 46

3.3 Tvåvägsval: `if`-else-satsen 47 `IfElse`
- Modulooperatorn 49
- Tillämpningar av modulo 49

3.4 Flervägsval 50
- `if-else`-stegen 51 `GissaTal`
- `switch`-satsen 50 `Switch`

3.5 Efter-testad repetition: `do`-satsen 55 `Collatz`
3.6 För-testad repetition: `while`-satsen 59 `Sum_while`
- Evighetsloop 60

3.7 Bestämd repetition: `for`-satsen 61 `Sum_for`
- `for`-satsens struktur 61
- En tillämpning av `for`-satsen 63 `Borr`

Övningar till kap 3 65

Kapitel 1

Introduktion till programmering

Ämne	Sida	Program
1.1 Om programmering	6	
- Algoritmiskt tänkande	7	
- Val av programmeringsspråk	8	
1.2 Programmeringens ABC	9	
- Interpretator vs. kompilator	9	
- Editorer	9	
- Om JavaScript	10	
1.3 Att komma igång med JavaScript	12	
- Programmet Welcome	12	`Welcome`
- Kommentar	13	
- Satser i JavaScript	13	
1.4 Konkatenering	15	`Concat`
- Överlagring	16	
1.5 Utskrift i flera rader	17	`Break`
- Radbrytning i utskriften med JavaScript	18	`Escape`
- Funktionen alert()	19	
- Escapesekvenser	19	
Frågor till kap 1	20	
Övningar till kap 1	21	

1.1 Om programmering

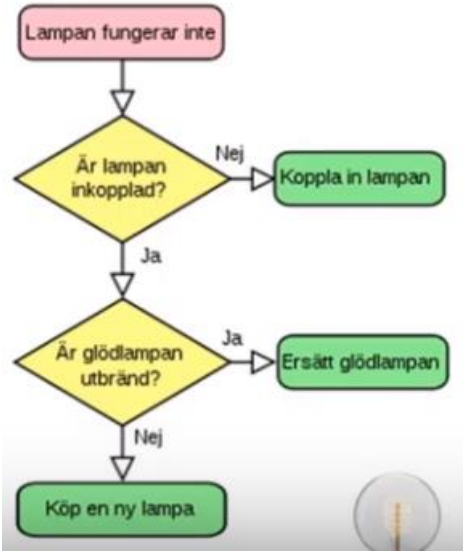
Världen vi lever i är full med prylar som är programmerade. De kallas för "intelligenta". Man pratar om *artificiell intelligens*. Men prylarna kan inte tänka själva. Någon har programmerat dem, närmare bestämt de elektroniska komponenterna i dem – små datorer. Det är de som styr all funktionalitet.

Programmering är ett av de mest spännande kapitlen i teknologihistorien. Inte bara därför att den har lagt grunden till den moderna IT-industrin och på gott och ont revolutionerat världen. Den har också bidragit till att förverkliga den urgamla mänskliga drömmen att förenkla mödosamma arbeten. Istället för att plåga sig instruerar man en maskin med idéer. Programmering realiserar önskemålet att låta datorn göra jobbet för att ha mer tid över för annat i livet.

När man tröttnat på att använda program som andra skrivit – maila, surfa eller lyssna på musik – är det dags att börja programmera själv. Det är roligare att köra en bil än att bara åka med. Det är kreativiteten och det fria skapandet som lockar. Med programmering kan du testa helt nya egna idéer.

*"Everyone in this country should learn how to program a computer. Because it teaches you how to **think**."*

Steve Jobs



Men hur programmerar man?

Egentligen gör vi det varje dag utan att vara medvetna om det. Är t.ex. en lampa trasig följer vi ungefär det som kan beskrivas med bilden till höger, ett s.k. *flödesschema*. I praktiken löser vi problemet att ersätta en trasig lampa genom att tänka och göra så utan att någonsin rita ett flödesschema. Flödesschemat illustrerar och dokumenterar dock *algoritmen*, dvs tillvägagångssättet för problemets lösning. När

den en gång är ritad skulle den kunna användas av vem som helst som vill byta en trasig lampa. Den blir en slags allmängiltig manual för just detta problem. Men ännu viktigare är att metodiken kan tas över till svårare problem.

Ett annat vardagligt exempel är matlagning. Vare sig vi använder ett recept ur en kokbok eller lagar efter känsla, följer vi en algoritm som dessutom – till skillnad från lampalgoritmen – även har en *input*, råvaror och en *output*, maträtten. Hårdvaran som hjälper oss är köket med alla sina instrument. Matreceptet är mjukvaran dvs programmet. Det är precis samma struktur när vi kör ett program på datorn, matar in indata och får ut utdata som resultat. Programmet vi använder är avgörande för resultatet, precis som matreceptet samt dess förverkligande är avgörande för om vi lyckas med maträtten.

Algoritmiskt tänkande

Båda exemplen visar: Det är algoritmer som medvetet eller omedvetet styr *hur* vi gör – ett sätt att tänka vars gemensamma drag kan generaliseras så här:

1. Att formulera problemet och definiera målet. Hur når vi målet – problemets lösning?

2. Genom att bryta ner problemet i mindre, överskådliga och enklare delar, s.k. *moduler*. Varje modul ska i princip kunna utföras av vem som helst. Detta kallas för *modularisering* som är en allmän princip inte bara i programmering utan i all problemlösning.

3. Genom att ge *instruktioner* som leder till problemets lösning. De måste formuleras på ett entydigt sätt så att de inte kan tolkas på olika sätt. För datorer gör exakt som vi säger. Det har visat sig att det vanliga språket inte lämpar sig för detta ändamål, för det är tolkbart. Skönlitteraturen är ett praktexempel för olika tolkningar av språket. Det vore synd om det inte vore så. Därför har man i programmering hittat på andra, speciella programmeringsspråk vars vokabulär och syntax följer strikta regler som är entydiga. Datorn kan tolka dessa regler endast mekaniskt.

4. I denna process uppstår situationer där vi måste träffa ett *val* – samma sak som att besvara en *fråga*. Den första frågan i algoritmen "Att byta lampa" är "Är lampan inkopplad?" (ovan). Valet mellan "Ja" och "Nej" avgör hur algoritmen fortsätter. Ytterligare val följer.

Det är avgörande att skilja mellan *instruktion* och *val*. En instruktion är ett *kommando* som måste *utföras* medan ett val är en *fråga* som måste *besvaras*. I flödesplanen till lampalgoritmen är *instruktion* (grön) och *val* (gul) markerade med olika färger. Deras distinktion blir avgörande när man går över från flödesplan till kod.

Algoritmers byggstenar

Man delar in algoritmers viktigaste ingredienser i tre kategorier och kallar dem för *kontrollstrukturer*, eftersom de är generella strukturer som styr och kontrollerar algoritmerna och ger dem den karakteristiska ordningen. Dessa grundläggande kontrollstrukturer är *sekvens*, *selektion* och *repetition* och kommer att tas upp i boken. De anses vara algoritmers byggstenar. Alla algoritmer är uppbyggda av dem.

Avgörande för en algoritms funktionalitet är ingrediensernas *inbördes ordning*. Tar man in i en kokande gryta potatisen först och köttet sedan – istället för tvärtom – blir det mos istället för maträtt. I detta sammanhang hör även algoritmens korrekta avslutning. Utan ett exakt formulerat *avslutningskriterium* som uppnås i ändlig tid uppstår evighetsloopar. När sådant inträffar brukar vi ofta säga att datorn "hängt sig". I själva verket är orsaken en algoritm med ett inkorrekt konstruerat avslutningskriterium. Allt detta kommer att behandlas utförligt i boken.

Ytterligare en ingrediens av algoritmer är *logik*. Datorer kan ingen logik. Människan måste föra över logiken in i datorn. Det är det som kallas för *artificiell intelligens*. Bl.a. formuleringen av korrekta avslutningskriterier i val och loopar, men även modularisering och strukturering kräver logiskt tänkande.

Att upptäcka *mönster* är också en förmåga som ofta behövs i konstruktion av algoritmer, vilket vi kommer att se i våra programexempel som följer i boken.

I valet av instruktioner som ska tas med i en algoritm är det en självklarhet att man sorterar bort allt som är mindre relevant och tar in endast det som är relevant. Dvs även att avgöra *relevansen* av saker och ting för att uppnå det definierade målet (punkt 1) hör till programmerarens uppgifter.

Val av programmeringsspråk

I denna bok har vi bestämt oss för programmeringsspråket JavaScript för att introducera till programmering. Men språket är bara ett medel av underordnad betydelse. Målet är att lära sig *tankesättet* och *tekniken* att programmera, oberoende av språk. Har man en gång förstått de grundläggande koncept som är gemensamma för alla språk, blir det närmast en teknikalitet att på egen hand lära sig ett nytt språk.

1.2 Programmeringens ABC

> Programmering är i allra högsta grad ett praktiskt ämne.

Man kan inte lära sig programmering genom att endast läsa böcker. För att lära sig programmering måste man *programmera*, precis som bilkörning. Och för att programmera behöver man en miljö, där man kan skriva och testa kod. Denna miljö är en programvara som i regel måste laddas ned och installeras innan man kan testa någon kod. Det finns en uppsjö av programmeringsmiljöer för de olika programmeringsspråken. Ofta kallas de för IDE.

IDE står för *Integrated Development Environment*, är alltså en integrerad programutvecklingsmiljö som inkluderar en editor, en *interpretator* resp. *kompilator* och andra verktyg för programutveckling i en och samma samlad miljö. *Visual Studio* är ett exempel på en professionell IDE som har utvecklingsverktyg för ett antal språk, bl.a. JavaScript. Det finns även andra, enklare miljöer. Men egentligen behöver JavaScript endast en texteditor där koden skrivs och sparas samt en webbläsare där koden exekveras. Vi kommer att använda oss av denna möjlighet som är oberoende av tredje parts verktyg för att slippa installera nya program. En editor och en webbläsare finns förinstallerade på alla datorer.

Interpretator vs. kompilator

En *interpretator* är ett program som *tolkar* källkod till maskinkod och skickar maskinkoden till datorns processor utan att mellanlagra den på hårddisken. Processorn exekverar maskinkoden. Källkod är kod som endast människan förstår, men inte datorn. Maskinkod är kod som endast datorn förstår, men inte människan.

Till skillnad från en interpretator är en *kompilator* ett program som *översätter* källkod till maskinkod och lagrar maskinkoden på hårddisken. Först när man exekverar skickas den kompilerade maskinkoden till datorns processor och utförs där. Vissa programmeringsspråk är kompilerande, andra är interpreterande. Det finns även hybrider. JavaScript är interpreterande och behöver ingen IDE.

Editorer

En *editor* är ett skrivverktyg på datorn, dvs ett program som kan hantera text. *Ordbehandlingsprogram* är en annan beteckning på editorer. På de flesta datorerna finns ofta minst en editor förinstallerad. För att skriva källkod och spara den i en fil behövs en editor. Men källkod får endast innehålla tecken som kan tolkas av interpretatorn resp. 'förstås' av kompilatorn. Därför måste editorn spara filen som *oformaterad textfil*, dvs utan styr- och formateringskoder. Arbetar man t.ex. i Windows kan Notepad (Anteckningar) eller Notepad++ vara lämpliga texteditorer, eftersom de sparar alla filer som rena textfiler av typ **`*.txt`** utan några formateringar. Även andra bra alternativ som t.ex. TextPad finns att gratis ladda ned från Internet. Ordbehandlare däremot av typ Word och andra formaterar texten och sparar sina filer

som dokument av typ **`*.docx`** eller annat. Formatering innebär att det läggs till osynliga tecken i texten som interpretatorn resp. kompilatorn inte känner till. Därför är sådana program inte lämpliga för att skriva kod.

Om JavaScript

JavaScript är ett s.k. *scriptspråk* som ursprungligen skapades år 1995 av *Netscape*, ett amerikanskt mjukvaruföretag som året innan hade lanserat den första populära webbläsaren. Scriptspråk är språk som till skillnad från s.k. *universella* programmeringsspråk, som t.ex. C/C++, C#, Java, Python, ... , endast kan användas för att koda webbsidor. Med universella språk däremot kan man programmera vilken applikation som helst. Hos scriptpråken kallas koden för *script*. Man nöjer sig med de enklare elementen i programmering, för att förse webbsidor med vissa funktionaliteter. Scripten bakas in i HTML-kod. Scriptspråk behöver ingen annan utvecklingsmiljö än webben. JavaScript är ett *interpreterande* scriptspråk som kan utföras i en webbläsare med en inbyggd JavaScript interpretator. Alla moderna webbläsare inkluderar en sådan interpretator. JavaScript får inte förväxlas med Java.

Som alla programmeringsspråk är även JavaScript definierat av ett antal nyckelord, även kallade *reserverade ord*, på eng. *keywords*. De är reserverade av och för själva språket, dvs bildar språkets ordförråd. De får inte användas som namn (identifierare) för variabler eller programmets andra delar, t.ex. funktioner osv. Några av dem är samlade i följande tabell:

Reserverade ord i JavaScript				
`break`	`case`	`continue`	`delete`	`do`
`else`	`false`	`for`	`function`	`if`
`in`	`new`	`null`	`return`	`switch`
`this`	`true`	`typeof`	`var`	`void`
`while`	`with`	`default`	`class`	`const`

Det finns fler än i tabellen ovan. Observera att de alla skrivs med små bokstäver. Följande allmän regel gäller för all JavaScript kod:

> JavaScript är case sensitive (skiftlägeskänslig).

Dvs JavaScript skiljer på små och stora bokstäver. Det gör inte HTML.

Om HTML

HTML står för *HyperText Markup Language* och är webbens standardspråk för att utforma presentabla *dokument* som kombinerar text, bild och andra element. Koden genererar dokumentet som ska sedan visas upp dem på webben. Koden är skild från dokumentet – till skillnad från andra formateringsverktyg som t.ex. *Word* som är ett s.k. ***WYSIWYG***-verktyg. Akronymen (förkortningen) står för ***W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et*. Men eftersom HTML är ett icke-*WYSIWYG*-verktyg måste koden först tolkas av en interpretator, innan dokumentet kan visas. Webbläsare är sådana interpretatorer, dvs program som kan tolka HTML-kod. Dessutom har HTML möj-

ligheten att bädda in andra scriptspråk i sin kod som t.ex. JavaScript. Därför är webbläsaren den naturliga utvecklingsmiljön för JavaScript. För att skriva och testa JavaScript kod behöver man bara en *editor*, där man skriver kod, och en *webbläsare* där man kör koden. Vi nöjer oss med denna minimalistiska miljö för att förenkla den tekniska hanteringen och koncentrera oss på själva språket.

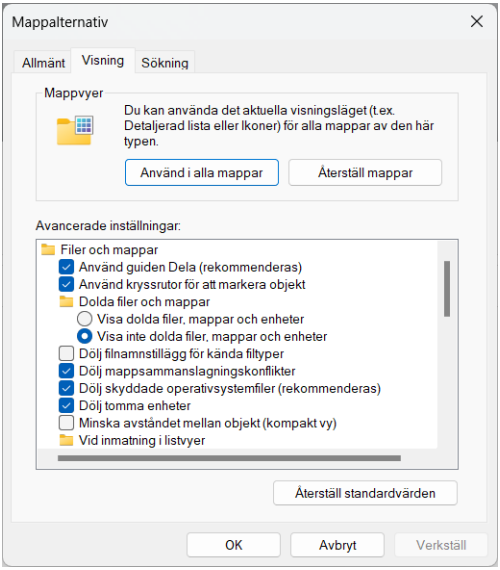
Regler för filändelsen

Skriver du din JavaScript kod i någon editor och sparar filen som **`*.txt`**, kommer du inte kunna exekvera den i en webbläsare, när du (dubbel)klickar på den. Boven i dramat är filändelsen: Operativsystemet identifierar de filer som innehåller kod via filändelsen. All JavaScript kod är inbakad i HTML kod, webbläsarens språk. Ska koden exekveras i en webbläsare måste filen som innehåller koden, ha ändelsen **`html`**, för att kunna identifieras som en JavaScript källkodsfil. Därför måste du antingen från början spara din källkodsfil med ändelsen **`html`** eller i efterhand ändra filändelsen till **`html`**. I Windows kallas filändelser för *Filnamnstillägg*.

Att hantera filändelser

För att kunna följa reglerna för filändelsen som beskrevs ovan, förutsätts att man kan *se* filändelserna när man öppnar en mapp. Men i praktiken är detta ofta inte fallet. Orsaken år på operativsystemets inställningar. I Windows är default inställningen att man i regel *inte* kan se dem. Ta själv reda på hur det är på din dator. Så här kan man göra för att synliggöra filändelserna i Windows:

- Öppna en mapp i Windows.
- Gå i mappens menyrad till Mappalternativ. Om du inte hittar denna meny klicka på de tre små punkterna till höger (Visa mer) och välj Alternativ.
- Du borde få upp dialogrutan Mappalternativ. Välj fliken Visning. Bocka av rutan Dölj filnamnstillägg för ända filtyper. Så här borde nu dialogrutan se ut:
- Klicka på knappen Använd i alla mappar, sedan på Ja och OK.



Nu borde du kunna se dina filers ändelser och kunna följa reglerna på förra sidan. Generellt rekommenderas att ha synliga filändelser på sin dator, när man programmerar.

1.3 Att komma igång med JavaScript

För att komma igång med JavaScript kan vi nu skriva våra koder i en valfri text-editor och spara filen som ren, dvs oformaterad textfil med ändelsen **`html`** (OBS! inte **`txt`**) på datorn. När vi sedan (dubbel)klickar på filen, kommer koden att exekveras i webbläsaren. Anledningen till det är att webbläsaren är ett program som kan tolka och exekvera **`html`**-kod: Webbläsaren är en **`html`**-interpretator. Så här kommer vi att testa alla våra JavaScript koder i denna kurs. Även om man gör detta i en annan miljö är det i grund och botten denna teknik som används i bakgrunden.

Vi sa i början att JavaScript var ett scriptspråk och att koden kallas för script. Men i fortsättningen kommer vi kalla våra JavaScript koder även för *program*.

Programmet `Welcome`

Öppna en texteditor, t.ex. NotePad++, skriv följande kod (utan radnumren) med bibehållen layout:

```
1  <!-- Welcome.html
2       Skriver ut en rad text -->
3
4  <title>Vårt första program i JavaScript</title>
5  <script>                    <!-- Här börjar JavaScript -->
6    document.writeln('<h1>Välkommen till JavaScript!</h1>')
7  </script>                   <!-- Här slutar JavaScript -->
```

Spara den i filen **`Welcome.html`**. (Dubbel)klicka på filen på den plats du sparat den. Din webbläsare kommer att visa körresultatet. Så här ser resultatet ut i min webbläsare (Google Chrome):

Vi kommer i fortsättningen att referera till denna kod som *programmet* **`Welcome`**, medan *filen* i vilken koden är sprarad heter **`Welcome.html`**.

Vi går nu i genom koden genom att referera till radnumren i programmet **`Welcome`**. Huvudjobbet görs av rad **`6`** som skriver ut texten ovan. Men låt oss gå från början:

Kommentar

Raderna **1-2** i programmet **Welcome** är kommentar. Allt som skrivs mellan **<!--** och **-->** betyder i HTML kommentar, dvs utförs inte, utan ska förklara koden. Kommentar börjar med **<!--**, slutar med **-->** och kan sträcka sig över flera rader.

HTML-taggar

Raderna **4-7** består av tre s.k. *HTML-taggar*. All kod som skrivs inom **<** och **>** kallas för *HTML-tagg*. På rad **4** börjar en HTML-tagg med **<title>** och slutar med **</title>**. All text som skrivs mellan är **<title>** och **</title>** kommer att synas på rubriken av webbläsarens flik. I programmet **Welcome** är det texten **Vårt första program i JavaScript** som man kan se i körresultatet längst upp till vänster.

På rad **5** börjar nästa tagg med **<script>** som slutar på rad **7** med **</script>**. Denna tagg, **script**-taggen, betyder att här inbäddas JavaScript i HTML. Allt som står mellan **<script>** och **</script>** utförs av JavaScript-interpretatorn som är integrerad i webbläsaren. JavaScript är standarden bland de scriptspråk som finns i webbläsaren.

Satser i JavaScript

I **script**-taggen (raderna **5-7**) hittar vi följande JavaScript-*sats*:

```
document.writeln('<h1>Välkommen till JavaScript!</h1>')
```

Att vi kallar denna kod för *sats*, beror på att den inte längre är HTML- utan JavaScript-kod, eftersom den står i **script**-taggen. I JavaScript är *satser* motsvarigheten till taggar i HTML. Inte bara koden skiljer sig utan även terminologin. Vi har nu på allvar kommit in i programmeringen. Det visas redan på punkten som står mellan **document** och **writeln()**. Satsen ovan är ett *anrop* av funktionen **writeln()**. En *funktion* är kod som föreskriver vad som ska *göras*. Funktionen **writeln()** ska skriva ut det som står i parentesen på webbläsarens yta och byta rad efteråt. Funktionen är förprogrammerad och finns i **document**, ett s.k. *objekt* tillhörande webbläsaren. För att kunna hitta funktionen **writeln()** måste vi först nämna dess behållare, objektet **document**, sätta sedan en punkt och skriva sist funktionens namn – en slags adressering. Därför blir det slutligen – bortsett från parentesens innehåll:

```
document.writeln()
```

Det här sättet att koda kallas *punktnotation* som vi kommer att använda ofta i fortsättningen. Punkten skiljer två olika kategorier av kod, i det här fallet objektet (före punkten) från funktionen (efter punkten).

Funktioner är karaktäriserade genom parentesen **()**, oavsett parentesen är tom eller inte. När de är definierade i ett objekt kallas de för *metoder*. Så, **writeln()** skulle kunna även kallas för en metod. Alla dessa nya begrepp kommer att behandlas i

detalj senare. Vad gäller **`writeln()`**-funktionens parentes kan man konstatera att följande regel gäller:

> I JavaScript omgärdas strängar av apostrofer **`' '`** eller citationstecken **`" "`**.

Sträng är den programmeringstekniska termen för text. Vi använder i våra exempel apostrofer. Citationstecken går lika bra. I programmet **`Welcome`** (sid 12) står koden **`<h1>Välkommen till JavaScript!</h1>`** inom apostrofer. Därför visar programmets körresultat själva texten i fet stil och i en viss storlek i webbläsaren, medan HTMLs **`<h1>`**-tagg bestämmer textens storlek och stil.

Observera att **`<h1>`**-taggen dvs HTML-kod fungerar i JavaScript (inom **`<script>`**-taggen), men JavaScript-kod inte i HTML (utanför **`<script>`**-taggen).

JavaScript-satser kan även avslutas med semikolon. Men alternativt kan man utelämna semikolonet och skriva varje sats på en ny rad. Dvs det osynliga radavslutningstecknet **`Enter`** kan ersätta semikolonet. Vi kommer att föredra detta alternativ av minimalistiska skäl – för att minska kod. Däremot är det absolut nödvändigt att avsluta **`script`**-taggen med **`</script>`** på rad **`7`**, för att markera att det är slut på JavaScript-kod och att det nu fortsätter igen HTML-kod.

1.4 Konkatenering

```
1  <!-- Concat.html
2       Skriver ut flera rader text i olia storlekar
3       med konkateneringsoperatorn + -->
4
5  <title>Olika storlekar & konkatenering</title>
6  <script>
7     document.writeln('<h1> Välkommen till JS! (med h1) </h1>' +
8                      '<h2> Välkommen till JS! (med h2) </h2>' +
9                      '<h3> Välkommen till JS! (med h3) </h3>' +
10                     '<h4> Välkommen till JS! (med h4) </h4>' )
11 </script>
```

Öppna din favorit editor eller NotePad++, skriv koden ovan och spara den i filen **`Concat.html`**. (Dubbel)klicka på filen när du sparat den. Webbläsaren visar:

Vi kommer att referera i fortsättningen till koden ovan som *programmet* **`Concat`**.

Här skrivs ut fyra rader text i olika storlekar, förorsakat av HTMLs `<hi>`-taggar `(i = 1, 2, 3, 4)` som formaterar textens storlek.

Utskriften görs av ett enda anrop av funktionen **`writeln()`** i raderna **`7-10`**. Dvs vi skriver egentligen ut en enda text, även kallad *sträng*, bara att den är lång och inte ryms på en rad. Därför bryter vi den i fyra delar, men slår ihop strängens delar med tecknet **`+`** i raderna **`7-9`**. Plustecknet betyder här *inte* addition, utan har en annan betydelse som redovisas nedan.

Konkateneringsoperatorn +

To (con)catenate betyder på engelska att slå ihop. Termen används inte bara i JavaScript utan även i en rad olika sammanhang inom IT [*]. I programmet **`Concat`** *konkatenerar* vi strängar med **`+`** som därför kallas för *konkateneringsoperatorn*. Anledningen till att vi använder konkateneringsoperatorn i programmet är följande regel:

> Mitt i en sträng får man inte bryta rad i JavaScript koden.

Man kan i koden bryta rad på alla ställen där ett mellanslag förekommer. Detta gäller dock inte för mellanslag *mitt i en sträng*. T.ex. ger följande radbrytning i koden fel, dvs inget resultat, därför att raden bryts mitt i en sträng:

```
document.writeln('<h1>Välkommen till
                    JavaScript!</h1>')
```

Vill man ändå bryta rad måste man dela upp den i *två* strängar och skicka mellan dem konkateneringsoperatorn:

```
document.writeln('<h1>Välkommen till ' +
                  'JavaScript!</h1>')
```

Observera att mellanslaget i en sträng måste skickas med även i koden. Annars blir det inget mellanskag i utskriften. Därför måste vi lägga till det efter **`till`**.

Konkateneringsoperatorn hjälper oss att undvika det fel som nämns i regeln ovan.

Överlagring

Att kod i olika sammanhang kan ha olika betydelser, kallas i programmeringstermer för *överlagring*, på eng. *overloading*. De multipla betydelserna *överlagrar* varandra. Den aktuella betydelsen träder fram i ett konkret sammanhang och avgörs därmed av sammanhanget – både för oss och för JavaScript-interpretatorn: Står t.ex. **`+`** mellan två *tal* betyder det addition. Står **`+`** mellan två *strängar* betyder det konkatenering. Överlagring är ett generellt koncept inom programmering som används i alla moderna programmeringsspråk.

[*] T.ex. i C/C++ finns funktionen **`strcat()`** som gör **`str`**ing **`cat`**enation, dvs konkatenerar två strängar. Samma sak gör metoden **`concat()`** i Java. I Unix, som är skrivet i C, finns kommandot **`cat`** som konkatenerar data från olika filer och slår ihop dem till en fil. T.ex. kopierar kommandot **`cat file1 file2 file3 > nyfil`** de tre filerna till **`nyfil`**.

1.5 Utskrift i flera rader

```
1  <!-- Break.html
2       Radbrytning i utskriften med HTMLs break-tagg <br> -->
3
4  <title>Utskrift i flera rader</title>
5  <script>
6     document.writeln('<h1> Välkommen till <br> JavaScript-' +
7                      '<br> programmering! </h1>')
8  </script>
```

Observera att vi här pratar om radbrytning inte i koden utan i *utskriften*, dvs i körresultatet, se nedan. Koden producerar tre utskriftsrader med hjälp av HTML-taggen `<br>`, inbakad i utkriftssträngen. Körresultatet blir:

Programmet **`Break`** använder HTML-taggen `<br>`, även kallad *break-taggen* genom att baka in den två gånger i strängen av **`document.writeln()`**-satsen (rad **6** & **7**) för att åstadkomma radbrytning i utskriften.

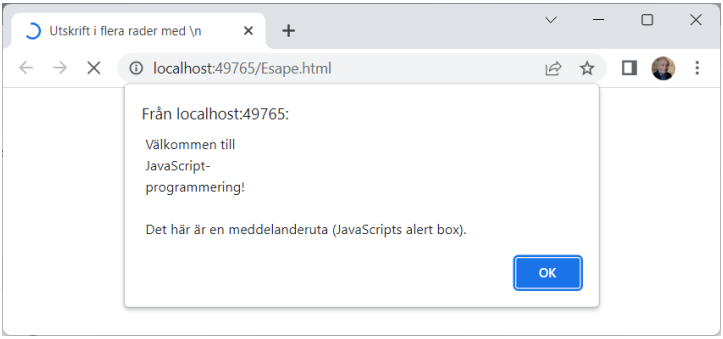
Konkateneringsoperatorn **`+`** används på rad **6**, precis som i programmet **`Concat`** (sid 15), för att inte behöva bryta rad i koden mitt i en sträng, se regeln på förra sidan.

`<br>`-taggen är HTML kod. Vi har använt den i **`document.writeln()`**-satsen som i sin tur finns inom `script`-taggen, dvs där JavaScript kod gäller. Ändå kommer radbrytning i utskriften inte fungera, om vi byter ut HTML koden `<br>` mot JavaScripts motsvarighet till radbrytning, som är **`\n`**. Genomför gärna detta experiment. Längre fram kommer vi att förklara **`\n`** närmare.

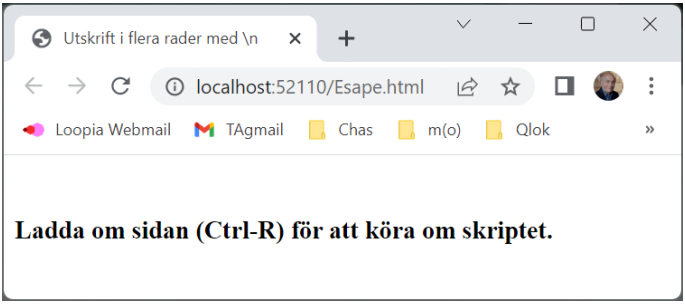
Radbrytning i utskriften med JavaScript

```
1  <!-- Escape.html
2       Radbrytning i utskriften med JavaScripts escapesekvens \n
3       Använder JavaScript funktionen alert() -->
4
5  <title>Utskrift i flera rader med \n</title>
6  <script>
7     alert(' Välkommen till \n JavaScript- \n' +
8           ' programmering! \n\n Det här är '  +
9           ' en meddelanderuta (JavaScripts alert box).' )
10 </script>
11
12 <br>
13 <h3>Ladda om sidan (Ctrl-R) för att köra om skriptet.</h3>
```

Körresultatet är:



Här ser man tydligt att alert boxen visas i en ruta skild från webbdokumentet. Alert boxen är en JavaScript-konstrukt som består av en meddelanderuta och en OK-knapp som stänger rutan när den klickas. Programflödet återgår sedan till webbläsaren som först gör radbyte med `<br>` enligt rad **12** i programmet **Escape** och sedan skriver ut följande instruktion till användaren:



Programkörningen är inte avslutad förrän webbläsaren stängs.

Funktionen `alert()`

Programmet **Escape** använder en annan funktion än programmet **Break** för att skriva ut text, nämligen funktionen **alert()** på rad **7**. Detta för att demonstrera radbrytning i utskrift med JavaScript-koden **\n** som inte fungerade i programmet **Break**.

alert() är en JavaScript-funktion som genererar en meddelanderuta, en s.k. *alert box*. För att åstadkomma radbyte i utskriften används JavaScript-koden **\n** som betyder **n**ewline (rad **7** & **8**), se ***Escapesekvenser*** nedan. Precis som **\n** inte fungerade i programmet **Break**, för att åstadkomma radbrytning i utskriften, kommer **
** inte fungera i programmet **Escape**. Genomför gärna detta experiment genom att byta ut alla **\n** på raderna **7** & **8** i programmet **Escape** mot **
**.

I programmet **Escape** blir inte bara skilnaden utan även samspelet mellan HTML och JavaScript påtaglig.

Escapesekvenser

\n är ett exempel på en escapesekvens. På svenska betyder *to escape* att fly. Escapesekvenser inleds med tecknet backslash **** åtföljt av endast *ett* tecken. Med **** vill man *fly* från tecknets vanliga betydelse och ge det en *annan* betydelse. Med **\n** t.ex. vill man fly från bokstaven **n** och åstadkomma en **n**ewline. På samma sätt fungerar andra escapesekvenser som t.ex. **\t**, **\b** , **\'** , **\0**, **\f** , **\r** , ... Escapesekvensen **\'** t.ex. kan användas för att skriva ut själva apostrofen.

Escapesekvenser är ett generellt koncept som används i alla moderna programmeringsspråk.

Frågor till **Kapitel 1**

1.1 Vad menar *Steve Jobs* med sitt påstående att programmering lär oss att *tänka?*

1.2 Hur tolkar du termen *artificiell intelligens (AI)?* Tror du att maskiner kan lära sig att ”tänka”? Eller är det bara något som människan kan göra?

1.3 Försök att med egna ord beskriva *algoritmiskt tänkande*.

1.4 Vad har algoritmiskt tänkande med programmering att göra?

1.5 Hur skulle du definiera begreppet *algoritm?*

1.6 Är datorprogram det enda sättet att *beskriva* en algoritm?

1.7 Använder du i vardagen algoritmer? Om ja, nämn några exempel.

1.8 Vad innebär *modularisering* och varför är den relevant för programmering?

1.9 Varför kan man inte lära sig programmering genom att endast läsa böcker?

1.10 Vad innebär *kompilering* och hur skiljer den sig från *exekvering?*

1.11 Skriver man *källkod* eller *maskinkod* när man programmerar?

1.12 Vilken egenskap borde editorn ha i vilken man skriver programkoden?

1.13 Är JavaScript ett *universellt* programmeringsspråk? Motivera!

1.14 Är JavaScript ett interpreterande eller ett kompilerande språk?

1.15 Är JavaScript källkod eller maskinkod?

1.16 I vilken miljö exekveras JavaScript kod?

1.17 Vad har JavaScript med HTML att göra?

1.18 Vilka verktyg behöver man för att kunna utveckla JavaScript program?

1.19 Vilka typer av ordbehandlingsprogram är olämpliga för programmering?

1.20 Varför är filändelser relevanta för en programmerare?

Övningar till **Kapitel 1**

1.21 Har du en favorit editor (sid 9)? Om ja, öppna den. Om inte, ladda ned open-source editorn Notepad++ och installera den. Undersök i editorn skillnaderna – vad gäller formen och utseendet – mellan tecknen *apostrof* (**'**), *citationstecken* (**"**), *accent* (´) och *backslash* (****). Ta reda på och kom ihåg deras tangenter på ditt tangentbord.

1.22 Visar din dator filändelserna när du öppnar en mapp? Om inte, genomför instruktionerna ***Att hantera filändelser*** på sid 11 för att synliggöra filändelserna.

1.23 Öppna din favorit editor eller Notepad++ och mata in koden till programmet **`Welcome`** (sid 12). Bibehåll layouten. Spara koden i filen **`Welcome.html`**. (Dubbel)klicka på filen, så att den körs i din webbläsare. Ersätt alla apostrofer i koden med citationstecken och kör om koden. Vilken slutsats drar du?

1.24 Modifiera programmet **`Welcome`** genom att ändra texten i `<title>`-taggen till ditt namn och texten som skrivs ut i dokumentet, till: **`Det här programmet har jag skrivit själv!`** Spara koden i filen **`Mitt.html`** och kör den.

1.25 Ersätt i programmet **`Concat`** (sid 15) **`document.writeln()`**-satsen med fyra olika sådana som ska ge samma utskrift som det ursrungliga programmet. Gör ytterligare ändringar i koden, så att de fyra utkriftsraderna syns i växande textstorlekar istället för minskande.

1.26 Utskrift i flera rader kan kodas på två olika sätt: antingen med HTMLs `<br>`-tagg eller med JavaScripts escapesekven **`\n`**. Ersätt i programmet **`Break`** (sid 17) `<br>` med **`\n`**. Ersätt i programmet **`Escape`** (sid 17) **`\n`** med `<br>`. Beakta funktionerna i vilka dessa koder fungerar. Vilka slutsatser drar du?

1.27 Skriv ett JavaScript program som åstadkommer följande utskrift:

```
*
**
***
****
*****
******
```

1.28 Sätt in följande kod i ett JS program och testa vad den ger för utskrift:

```
document.writeln('****<br>'     +
                 '*****<br>'    +
                 '******<br>'   +
                 '*******<br>'  +
                 '********<br>' +
                 '*******<br>'  +
                 '******<br>'   +
                 '*****<br>'    +
                 '****<br>'     )
```

Kapitel 2

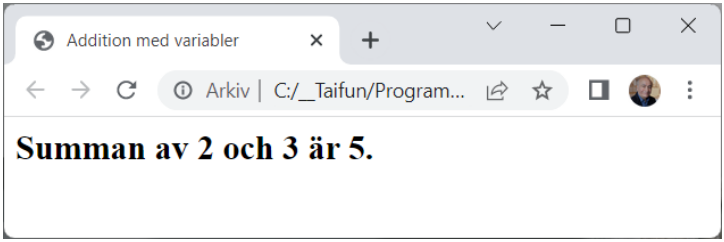
Grundbegrepp i programmering

Ämne	Sida	Program
2.1 Variabler	23	`Variable`
- Vad är en variabel?	23	
- Tilldelningsoperatorn =	24	
2.2 Överskrivning eller kan x = x + 1 vara sant?	26	`Overwrite`
- Prioritet av operatorer	27	
- Tilldelning vs. likhet	27	
2.3 Inläsning av data	28	`Input`
- Funkionerna `prompt()` och `parseInt()`	29	
2.4 Arrays	30	`Arraydef`
- Arrayens initieringslista	33	`ArrayInit`
2.5 Hantering av slumptal	35	`Random`
- Slumptan inom ett intervall	36	
Övningar till kap 2	37	
Inlämningsuppgift	38	

2.1 Variabler

```
1  <!-- Variable.html
2       Adderar två tal med variabler -->
3
4  <title>Addition med variabler</title>
5
6  <script>                  // Radkommentar i JavaScript
7     no1 = 2                // Initiering av variablerna
8     no2 = 3                // no1, no2 och sum
9     sum = no1 + no2
10
11    document.writeln('<h2> Summan av ' + no1 + ' och '
12                     + no2 + ' är ' + sum + '. </h2>')
13 </script>
```

Här förekommer två lika koder för kommentar: Raderna **1-2** är kommentar som är HTML-kod. Den börjar med **<!--** och slutar med **-->**, kan sträcka sig över flera rader och därför kallas för *blockkommentar*. I raderna **6-8** inleds kommentar mitt på en rad med JavaScript-koden **//** som slutar när raden slutar och därför kallas för *radkommentar*. Att vi kan använda den här, beror på att vi gör det *efter* **<script>**-taggen, där JavaScript-kod gäller. I körresultatet syns förstås inte kommentarerna:



I programmet **Variable** skapas på raderna **7-9** tre *variabler* **no1**, **no2** och **sum**.

Vad är en variabel?

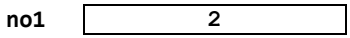
En variabel är en platshållare (minnescell) för ett värde (data).

I koden får variabeln ett namn som används för att komma åt värdet.

I ett program kan variabelns värde ändras, men inte namnet.

Ex.: På rad **7** skapas variabeln **no1** och initieras till värdet **2**.

När vi kör programmet **Variable** reserveras en minnescell i datorns RAM *(Random Access Memory)* vars namn är **no1** och vars innehåll är **2**:



Det är jämförbart med en låda vars etikett är variabelns namn och vars innehåll är variabelns värde. *Värde* är data i största allmänhet, dvs tal, tecken, men även ett sanningsvärde, en sträng, längre text, en fil, ja t.o.m. en bild. Vi kan i programmet komma åt värdet **2** genom att i koden *referera* till variabeln **no1**. Detta görs t.ex. på rad **9**, för att addera variablerna **no1**:s och **no2**:s värden och initiera därmed variabeln **sum**.

Motsatsen till *variabel* är begreppet *konstant*, t.ex. **2**, som inte kan ändra sitt värde under en programkörning. Det kan däremot en variabel göra. Hos en variabel måste man alltid skilja mellan *namnet* och *värdet*, medan konstanter är i regeln namnlösa.

Tilldelningsoperatorn =

I programmet **Variable** kodas initieringen av variabeln **no1** med satsen:

```
no1 = 2                    // Initiering av variabeln no1
```

Här *får* variabeln **no1** värdet **2**. Man skulle kunna beskriva bilden så här:

Dvs likhetstecknet kan snarare jämföras med en pil som pekar från höger till vänster. I RAM-minnet ser bilden ut så som det visades ovan. Variabelns *namn* är i koden den mjukvarumässiga motsvarigheten till minnescellens fysiska adress.

Vi kan i fortsättningen komma åt värdet **2** genom att *referera* till **no1**. T.ex. om vi nu skriver **document.writeln(no1)** får vi *värdet* **2** utskrivet.

Symbolen **=** betyder i matematiken likhet. Men i programmering betyder **=** *inte* likhet utan tilldelning och symbolen kallas för *tilldelningsoperatorn*. Den visar ingen likhet utan *uttför* tilldelning vilket betyder att en variabel *får* ett värde. Det är skillnaden mellan *att vara* och *att bli*. Likhet har i JavaScript symbolen **==** som används i villkor för att testa två värden på likhet.

Samma sak är det förstås med variabeln **no2** som i programmet **Variable** får värdet **3**. Sedan utförs additionen **no1 + no2**. Här adderas *värdena* lagrade i variablerna **no1** och **no2**. Resultatet tilldelas variabeln **sum**. Vi refererar till värdena med hjälp av variablerna. Att additionen **+** görs *först* och tilldelningen **=** *sedan* beror på att **+** binder starkare än **=** .

Utskriftssatsen

Intressant i programmet **`Variable`** är hur koden i utskriftssatsen måste skrivas för att med hjälp av variablerna åstadkomma körresultatet **Summan av 2 och 3 är 5**. Det är en kombination av variabler, strängkonstanter (inom apostrofer) och konkateneringsoperatorn **+** som måste skrivas i parentesen till funktionen **`writeln()`**:

```
document.writeln('<h2> Summan av ' + no1 + ' och '
                 + no2 + ' är ' + sum + '. </h2>')
```

`<h2>`-taggen som styr utskriftstextens storlek samt all text måste bakas in i apostrofer (strängkonstanter), medan variablerna måste stå utanför apostroferna. När de kopplas ihop med **+** är det variablernas aktuella *värden* som skrivs ut.

2.2 Överskrivning eller kan x = x + 1 vara sant ?

```
1  <!-- Overwrite.html
2       = betyder i programmering inte likhet utan tilldelning -->
3
4  <title>Överskrivning</title>
5
6  <script>
7     x = 5            // Initiering av variabeln x
8     document.writeln('<br>Variabeln x har initierats till ' + x)
9
10    x = x + 1        // Överskrivning av variabeln x
11
12    document.writeln(', sedan ökats med 1 och är nu ' + x + '.')
13 </script>
```

” I ett program kan variabelns värde ändras, men inte namnet. ”

Vad är en variabel? (sid 23)

En körning ger:

För tilldelning använder JavaScript samma symbol **=** som för likheten i matematik, vilket kan ge upphov till missförstånd eftersom det handlar om två olika typer av operationer. *Tilldelning* är en instruktion som skall utföras, medan *likhet* är en jämförelse som endast kan testas om den är sann eller falsk. Vid enkel tilldelning, t.ex. på rad **7**, har vi **x = 5**, dvs variabeln **x** förekommer endast på vänster sidan:

Variabeln x ⟵ **Värdet 5**

Men vid en annan tilldelning, t.ex. på rad **10**, finns *samma* variabel **x** på båda sidor:

x = x + 1

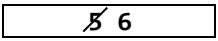
Dvs:

x ⟵ **x + 1**

Om t.ex. **x** har värdet **5** *före* denna sats, innebär satsen ovan att **5** ska adderas med **1** och att det nybildade värdet **6** ska tilldelas variabeln **x**:

x ⟵ **5 + 1**

Efter satsen har **x** värdet **6**. Det nya värdet **6** *skriver över* det gamla värdet **5**:



Detta kallas för *överskrivning* av variabeln **x**. Variabeln **x** är en platshållare vars värde kan ändras medan namnet bibehålls (sid 23). Initialvärdet **5** tilldelas variabeln **x**. Satsen **x = x + 1** ökar värdet till **6** och överskriver det gamla värdet **5** med det nya värdet **6**. Men varför ökas värdet *först*, innan det överskrivs? Det beror på:

Prioritet av operatorer

Två operatorer är inblandade i satsen **x = x + 1**, additionen **+** och tilldelningen **=** . Att JavaScript-interpretatorn utför additionen *först* och tilldelnigen *sedan* beror på att operatorn **+** har högre prioritet, dvs binder starkare, än tilldelningsoperatorn **=** . Operatorernas prioriteter är definierade i alla programmeringsspråk. Därför slipper vi att skriva: **x = (x + 1)**, vilket vi hade varit tvungna att göra om **=** hade samma prioritet som eller högre än **+** . Parentesen bryter prioritetsreglerna. Det är inte fel att skriva **x = (x + 1)** istället för rad **10**, men i det här fallet onödigt.

Tilldelning vs. likhet

Vi har i satsen **x = x + 1** med två olika värden till en och samma variabel **x** att göra, men vid två olika tidpunkter. Det gamla värdet **5** finns i variabeln **x** *före* satsen och det nya värdet **6** finns i variabeln **x** *efter* satsen.

I matematiken betyder tecknet = *likhet*. Därför är det fel att skriva $x = x + 1$ eftersom detta är en ekvation som saknar lösning. Man kan också säga att det är ett falskt påstående som leder till motsägelsen $0 = 1$. Vill man vara matematiskt korrekt måste man använda *två* variabler och skriva så här:

$$x_{\text{nytt}} = x_{\text{gammalt}} + 1$$

I programmeringen däremot betyder tecknet **=** inte likhet utan *tilldelning*. Därför är det helt OK att skriva **x = x + 1** eftersom det inte handlar om ett påstående som kan vara sant eller falskt utan snarare om en *instruktion* som ska utföras. Samma variabel **x** används på båda sidor av tilldelningstecknet. **x** är *en* platshållare (minnescell) vars innehåll (värde) skall *överskrivas* med satsen **x = x + 1** Instruktionen lyder att *tilldela* variabeln **x** ett nytt värde, att öka det gamla värdet med **1**. För *likhet* har an i JavaScript koden **==** som kallas för *jämförelseoperator*, se sid 43.

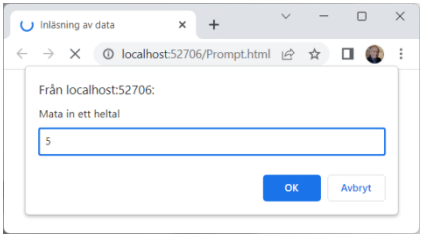
Filosofiskt handlar det om den klassiska skillnaden mellan *att vara* och *att bli*, mellan *tillstånd* och *handling*, mellan den statiska likheten och den dynamiska tilldelningen. Vid tilldelning relateras sanningen till tiden, dvs frågan är inte *om* utan *när* **x = 5**. Jo, precis när variabeln **x** tilldelas värdet **5**. Inte innan och ev. inte heller efteråt, för redan i nästa programsats kan ju variabeln **x** tilldelas ett annat värde. Med andra ord: Tilldelning är likhet relaterad till tiden dvs vid ett visst ögonblick, medan likheten är tidlös.

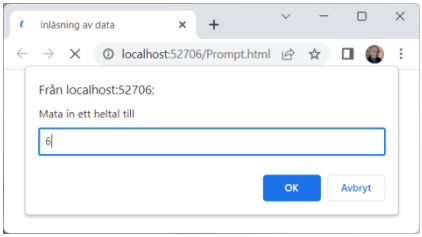
2.3 Inläsning av data

Hittills hade alla våra programexempel handlat om att skriva ut till skärmen. De hade endast utdata och ingen indata. Vill man även läsa in data till programmet, kan man använda sig av JavaScript-funktionen **`prompt()`**.

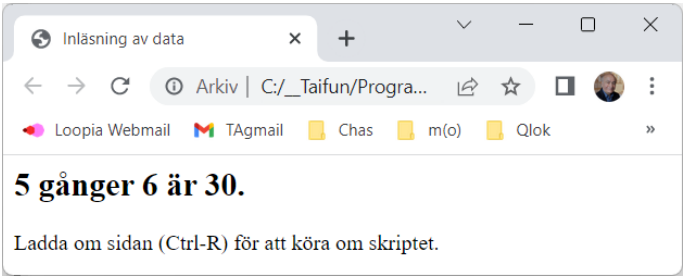
```
1  <!-- Input.html
2       Läser in två tal och skriver ut dem samt deras produkt
3       Funktionen prompt() skriver ut en ledtext och läser in
4       Funktionen parseInt() omvandlar inmatningen till heltal -->
5  <title>Inläsning av data</title>
6
7  <script>
8     no1 = parseInt(prompt('Mata in ett heltal'))  // Inläsning
9     no2 = parseInt(prompt('Mata in ett heltal till'))
10    prod = no1 * no2
11
12    document.writeln('<h2>' + no1 + ' gånger ' + no2 +
13                     ' är ' + prod + '. </h2>')
14 </script>
15
16 Ladda om sidan (Ctrl-R) för att köra om skriptet.
```

I programmet **`Input`** anropas funktionen **`prompt()`** två gånger (rad **8** & **9**). Båda anropen stoppar körningen i väntan på inmatning. En *ledtext* skrivs ut som instruktion till användaren. Följande meddelanderutor genereras:





Först när man matat in och klickat på OK fortsätter programkörningen och vi får:



Programmet **Input** arbetar med tre variabler: **no1**, **no2** och **prod**. Detta kan anses som en vidareutveckling (generalisering) av programmet **Variable** (sid 23). Variablerna **no1** och **no2**:s värden är inte längre hårdkodade utan läses in med godtycklig data. Variablernas initiering sker genom inläsning.

Funktionerna `prompt()` *och* `parseInt()`

Det som åstadkommer inläsningen är anropen av funktionerna **prompt()** och **parseInt()** på rad **8** i satsen:

```
no1 = parseInt(prompt('Mata in ett heltal'))
```

Denna sats gör många saker:

1. Stoppar programkörningen i väntan på inmatning.
2. Genererar en meddelanderuta.
3. Skriver ut en ledtext som instruerar programmets användare.
4. Omvandlar inmatningen till ett heltal med hjälp av funktionen **parseInt()**.
5. Skapar variabeln **no1** och initierar den med heltalet från punkt 4.
6. Fortsätter programkörningen, när användaren klickat på OK.

Inmatningen *returneras* av funktionen **prompt()**. Men eftersom **prompt()** är en fördefinierad funktion i JavaScript som returnerar en sträng, måste returvärdet omvandlas till heltal och tilldelas variabeln **no1**, för att kunna multipliceras på rad **10**.

Det är även möjligt att anropa funktionen **prompt()** utan ledtext. Men det tillhör god programmeringsstil att *inte* göra det, utan att skicka en ledtext, för att underlätta för användaren, när markören står och blinkar. Annars kan situationen tolkas som om programmmet har ”hängt sig”. Kommunikation och tydlighet är uppskattade egenskaper även hos nördiga programmerare.

2.4 Arrays

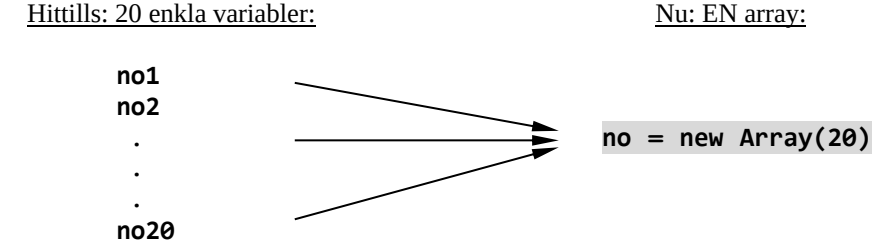
Datorn har några egenskaper som är helt överlägsna motsvarande egenskaper hos människan: snabbheten, noggrannheten och förmågan att effektivt lagra och hantera stora datamängder samt förmågan att aldrig bli trött.

Vi ska i detta avsnitt introducera ett verktyg som utnyttjar en av dessa överlägsna egenskaper, nämligen att kunna effektivt lagra och hantera *stora datamängder*. Detta verktyg heter *array* och betyder *ordnad uppställning* (*battle array* = stridsordning), en ordnad skara av data. Ibland används i litteraturen begreppet *fält* som är identiskt med *array*.

En *array* är en ordnad mängd av variabler grupperade under ETT namn.

Arrayens delar kallas för *element*. Elementens position kallas för *index*.

Vi kan gruppera t.ex. 20 variabler i en array med 20 element:



Hittills behövde vi skriva *20* satser för att skapa 20 variabler. Men nu har vi möjligheten att göra samma sak med endast *en* sats, genom att skapa *en enda* variabel – visserligen inte längre en vanlig variabel utan en *arrayvariabel* – och lägga till informationen om antalet element i den. På så sätt har vi skapat en *arrayvariabel* **no**.

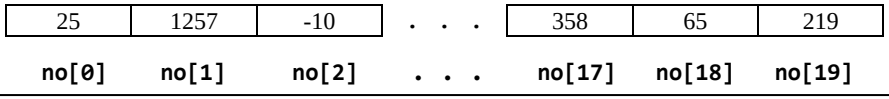
Arrayvariabeln **no** ersätter de 20 vanliga variablerna **no1**, **no2**, …, **no20** och består nu i sin tur av 20 *element*. Varje element är en variabel som kan lagra ett värde. Enda skillnaden är *sättet* dvs *koden* att komma åt dessa värden. Indexet är ett nummer som specificerar varje elements position i arrayen. Varje element i en array kan betraktas som en *indexerad* dvs *numrerad variabel*.

En array är inte längre en enkel utan en s.k. *sammansatt* datatyp. En *enkel datatyp* representerar ETT värde åt gången, t.ex. ett heltal, ett deci-maltal, ett tecken, ett sanningsvärde osv. En *sammansatt datatyp* representerar fler än ett värde åt gången, t.ex. flera heltal, flera flyttal, flera tecken, flera sanningsvärden osv. Man kan gruppera enkla datatyper till den sammansatta datatypen array.

Åtkomst till arrayens element

Följande sats definierar arrayen **no**: `no = new Array(20)`
Den allokerar (reserverar) 20 minnesceller för lagring av 20 värden. Låt oss anta att t.ex. vissa värden tilldelats arrayen **no**:s element, som man ser på bilden nedan. Eftersom elementen i en array alltid lagras i ett sammanhängande minnesområde, uppstår följande minnesbild:

Minnesbild av arrayen no:



Bilden visar hur indexeringen av element i en array organiseras. I raden under minnescellerna står hur JavaScript-kod kommer åt varje element i en array. Det är anmärkningsvärt är att indexnumreringen börjar med **0**, medan vi människor är vana vid att påbörja numreringen av ett antal objekt med **1**. Följande indexregel gäller:

> Indexregeln: I arrays börjar numreringen av index alltid med 0.
>
> Därför gäller: elementets position = index + 1

Med *position* menas numret som människan använder för att räkna elementen, medan kodens numrering – det som står inom hakparenteserna **[ ]** – kallas för *index*.

Det 1:a elementet i arrayen **no** ovan har index **0** och värdet 25, medan positionen är 1. JavaScript kodar elementet med **no[0]**. Det 2:a elementet har index **1** och värdet 1257 medan koden är **no[1]**. Det 3:e elementet: index **2**, värdet –10 och koden **no[2]** osv. Det **n**:e elementet har alltid index **n-1**. Därför har också det 20:e elementet index **19** och värdet 219. Det gäller att hålla isär det mänskliga sättet att numrera som börjar med 1 från JavaScript-kodens sätt att indesera som börjar med **0**. Vi har definierat 20 variabler **no[0], ..., no[19]**. Antalet element är 20. Indexen går från **0** till **19**.

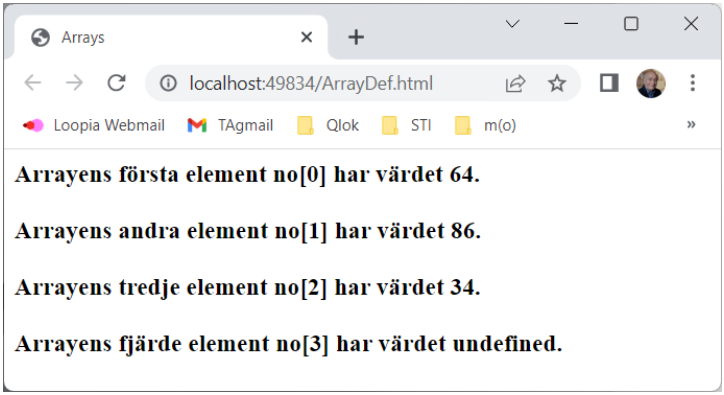
Av indexregeln följer dessutom att negativa index generellt inte är tillåtna.

Definition och initiering av en array

Följande program demonstrerar allt vi sagt om arrays speciellt indexregeln. Dessutom kan vi se, hur JavaScript hanterar överskridningen av de definierade indexgränserna.

```
<!-- ArrayDef.html
     Definierar en array, initierar & skriver ut den elementvis
     JavaScript tar hand om överskridning av indexgränsen -->
<title>Arrays</title>
<script>
   no = new Array(3)              // Definition av en array
                                  // med 3 element
   no[0] = 64                     // 1:a element initieras
   no[1] = 86                     // 2:a element initieras
   no[2] = 34                     // 3:e element initieras

   document.writeln('<h3>Arrayens första element no[0]'  +
                    ' har värdet ' + no[0] + '.<br><br>' +
                    'Arrayens andra element no[1]'       +
                    ' har värdet ' + no[1] + '.<br><br>' +
                    'Arrayens tredje element no[2]'      +
                    ' har värdet ' + no[2] + '.<br><br>' +
                    'Arrayens fjärde element no[3]'      +
                    ' har värdet ' + no[3] + '.</h3>'    )
</script>
```

Vi tittar på en körning:



Körningen visar att icke-definierade arrayelement inte leder till något fel. Index `3` överskrider de definierade indexgränserna **`0`** och **`2`**. Arrayelementet **`no[3]`** är varken definierat eller tilldelat något värde. Ändå kan man skriva det i koden och köra

programmet. Inte ens en varning påpekar att man använt kod som är odefinierad. Anledningen är följande:

> I en array kontrollerar JavaScript endast arraynamnet, inte indexen.
>
> Arrayelement som överskrider indexgränserna blir **"undefined"**.

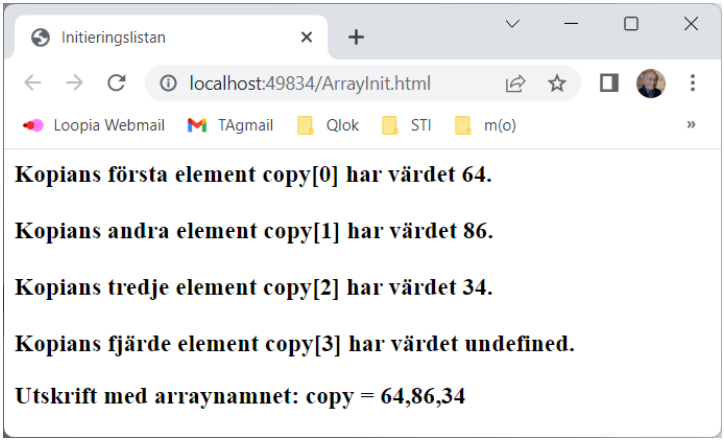
Ett annat namn än det definierade arraynamnet **no** leder till fel. Om vi däremot använder ett index som överskrider de definierade gränserna, kan vi fotfarande exekvera koden. Ansvaret för kontroll av indexgränserna ligger helt och hållet hos programmeraren. Skälet för denna liberala attityd är bl.a. strävan efter snabbhet, vilket förstås är på bekostnad av säkerheten.

Arrayens initieringslista

Precis som det finns skillnader i definitionen av arrayvariabler jämfört med vanliga variabler, finns även skillnader vid initieringen dvs första tilldelningen. T.ex. är initieringen av arrayen **no** i programexemplet **ArrayDef** – en sats för varje element – inte särskilt lämplig för arrays, speciellt om man skulle tillämpa samma teknik på större arrays. Men just hanteringen av stora datamängder var ju motiveringen för att syssla med array. Kan man inte effektivisera initieringen? Jo, till en viss gräns. Det finns i huvudsak två möjligheter: antingen att använda **for**-satser eller att slå ihop definitionen med tilldelningen till en kortform som använder sig av en s.k. *initieringslista*. Båda har vi använt i följande program:

```
1  <!-- ArrayInit.html
2       Kortform för definition och initiering av en array med
3       en initieringslista. Direkt tilldelning till en kopia -->
4  <title>Initieringslista</title>
5  <script>
6     no = [64, 86, 34]       // Kortform på definition och
7                             // initiering med initieringslistan
8     copy = no               // Tilldelning med arraynamnet
9                             // Elementvis utskrift av kopian:
10    document.writeln('<h3>Kopians första element copy[0]'   +
11                     ' har värdet ' + copy[0] + '.<br><br>' +
12                     'Kopians andra element no[1]'          +
13                     ' har värdet ' + copy[1] + '.<br><br>' +
14                     'Kopians tredje element copy[2]'       +
15                     ' har värdet ' + copy[2] + '.<br><br>' +
16                     'Kopians fjärde element copy[3]'       +
17                     ' har värdet ' + copy[3] + '.</h3>'    )
18                   // Utskrift av kopian med arraynamnet:
19    document.writeln('<h3>Utskrift med arraynamnet: '       +
20                                  'copy = ' + copy + '</h3>' )
21 </script>
```

En körning av programexemplet **ArrayInit** visar att värdena från arrayen **no** verkligen kopierats över till arrayen **copy**:



Både definitionssatsen och initieringssatserna i **ArrayDef** – det är de 4 första satserna – kan slås ihop till den enda satsen:

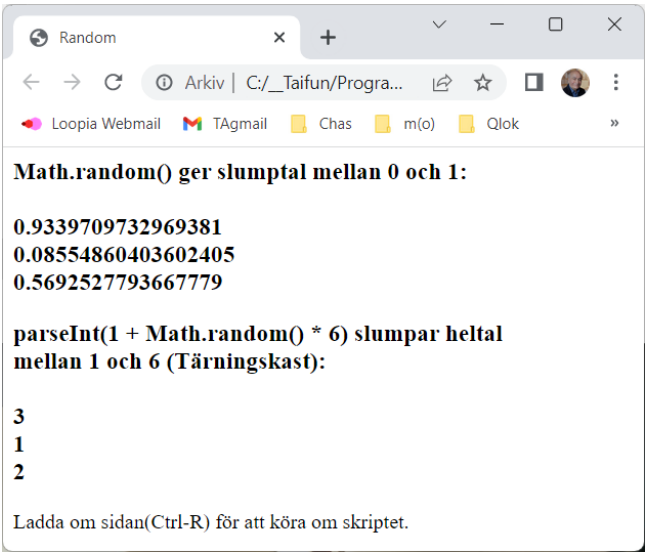
```
no = [64, 86, 34]    // Kortform för definition och
                     // initiering med initieringslistan
```

Satsen gör två saker: Först, fram till tilldelningstecknet definieras arrayen **no** *utan* någon uppgift om arrayens storlek. Sedan, från och med tilldelningstecknet tilldelas arrayen **no**:s element fyra värden som står i en kommaseparerad lista grupperad inom hakparenteserna **[ ]** som kallas arrayens *initieringslista*. Satsen ovan är endast en kortform för de fyra första satserna i `<script>`-taggen till **ArrayDef** och gör precis samma sak som de. JavaScript-interpretatorn får informationen om arrayens storlek i initieringslistan, dvs räknar antalet kommaseparerade element inom hakparenteserna **[ ]**. Observera att man får använda kortformen ovan endast i samma sats som definitionen.

2.5 Hantering av slumptal

```
1  <!-- Random.html
2       Slumpar tal mellan 0 och 1 med funktionen Math.random()
3       parseInt(1+Math.random()*6) slumpar heltal mellan 1 & 6 -->
4  <title>Random</title>
5
6  <script>
7     document.writeln('<h3>Math.random() ger'    +
8          ' slumptal mellan 0 och 1:<br><br>'    +
9          Math.random() + '<br>' + Math.random() +
10               '<br>' + Math.random() + '</h3>')
11
12    document.writeln('<h3>parseInt(1 + Math.random() * 6) '   +
13        ' slumpar heltal<br>mellan 1 och 6 (Tärningskast): '  +
14        '<br><br>' + parseInt(1 + Math.random() * 6) + '<br>' +
15                     parseInt(1 + Math.random() * 6) + '<br>' +
16                     parseInt(1 + Math.random() * 6) + '</h3>')
17 </script>
18
19 Ladda om sidan(Ctrl-R) för att köra om skriptet.
```

En körning ger:



JavaScript-funktionen **`Math.random()`** slumpar decimaltal mellan **`0`** och **`1`** (rad **`9`** & **`10`**). Mer exakt inom intervallet **`[0, 1)`**, dvs från och med **`0`** till, men inte med, **`1`**. Matematiskt uttryckt:

$$0 \leq \texttt{Math.random()} < 1$$

Egentligen kan datorn som en deterministisk maskin inte producera slumptal, Man kan endast *simulera* slumptal genom att *beräkna* tal, vilket sker enligt en viss algortim. Resulatet är förstås inte ”äkta” slumptal. I praktiken måste vi nöja oss med simulerade slumptal, s.k. *pseudoslumptal*.

Programmet `Random`:s utskrift visar tre slumptal mellan `0` och `1`, dessutom decimaltal. Ur användningssynpunkt är det inte särskilt intressant att hantera slumptal med 16 decimaler mellan `0` och `1`. Ofta vill man inte ha decimal- utan heltal och dessutom kunna själv bestämma inom vilket intervall heltalen ska vara.

Slumptal inom ett intervall

Här vill vi konstruera en formel som slumpar heltal inom ett önskat intervall. Låt oss för enkelhetens skull börja med intervallet `[1, 6]`, t.ex. för simulation av tärningskast. Sedan kan man generalisera formeln till ett godtyckligt intervall `[a, b]`.

För att skräddarsy JavaScript funktionen `Math.random()` för vårt ändamål, nämligen att få heltal mellan `1` och `6`, utför vi först en *skalning* med `6` och sedan en *skiftning* med `1`. Slutligen görs en omvandling till heltal. Följande formel fås:

```
parseInt(1 + Math.random() * 6)
```

Med *skalning* menas multiplikation med `6`, dvs en förstoring av intervallet `[0, 1)` till `[0, 6)`, dvs från och med `0` till, men inte med, `6`. Om vi endast tar heltalsdelen ger detta slumptal mellan `0` och `5`.

Med *skiftning* menas en förskjutning av intervallet `[0, 5]` med `+ 1` som ger slumptal mellan `1` och `6`.

Slutligen omvandlas hela uttrycket till heltal med hjälp av JavaScript-funktionen `parseInt()`. Vi får formeln ovan som har använts i programmet `Random` på raderna `14-16`.

Formeln kan generaliseras: Vill man ha slumptal mellan `a` och `b` och `a < b`, kan man transformera talen mellan `0` och `1` till tal mellan `a` och `b`, genom att skriva:

```
parseInt(a + Math.random() * (b – a + 1))
```

För att få intervallet `[a, b]`:s längd måste man bilda uttrycket `b – a + 1`.

Är `a > b` måste formeln ovan ersättas med:

```
parseInt(b + Math.random() * (a - b + 1))
```

Dessa formler skulle kunna användas i program som ska slumpa heltal i intervallet `[a, b]`.

Övningar till Kapitel 2

2.1 Komplettera programmet **Variable** (sid 23) genom att skapa ytterligare variabler, säg **diff**, **prod**, **div**. Tilldela till dem uttryck bildade med de andra räknesätten **-**, ***** och /. Skriv ut resultaten med meningsfulla utskrifter, genom att använda varablernas namn.

2.2 Varför fungerar inte följande kod i JavaScript?

```
1  <!– Ovn_2_2.html
2      Adderar två tal med variabler -->
3  <title>Funkar inte!</title>
4  <script>
5     a = 1
6     sum = sum + a
7     document.writeln('<h2> sum = ' + sum + '. </h2>')
8  </script>
```

Hitta felets orsak och åtgärda felet.

2.3 Ersätt i programmet **OverWrite** (sid 26) satsen **x = x + 1** med **x++**. Blir det samma resultat när du kör? Dra slutsats för betydelsen av satsen **x++**. Gör samma sak med **x--** istället? Förklara skillnaden till förra körningen. Med vilken sats är **x--** identisk?

2.4 Vidareutveckla din lösning till övn 2.1 genom att ersätta den hårdkodade tilldelningen av variablerna **no1** och **no2** med *inläsning*. Använd för inläsningen funktionen **prompt()** med ledtext, se programmet **Input** (sid 28).

2.5 Skriv ett JavaScript program som skriver ut fem slumptal

a) mellan 0 och 1.
b) mellan 10 och 30.
c) som heltal mellan 25 och 50.

2.6 Skriv ett JavaScript program som läser in tre siffror (0-9) och skriver ut dem i omvänd ordning.

2.7 Skriv ett JavaScript program som läser in tre tecken och skriver ut dem i omvänd ordning.

Inlämningsuppgift

Gymnastiktävling Skriv ett JavaScript program som avgör en tävling i gymnastik. Tre tävlande deltar i tävlingen. De får sina poäng av 3 olika domare. Poängen ska ligga mellan 0 och 10. Poängen ska summeras till en totalpoäng för varje tävlande. Programmet ska skriva ut både varje tävlandes totalpoäng och utropa tävlingens vinnare.

Använd tre arrays. Varje array ska lagra poängen för varje tävlande. Varje element i arrayen ska tilldelas en domares poäng. Simulera domarnas poänggivning med slumptal inom intervallet [0, 10]. Slumpvärdena kan vara decimaltal.

Ledning:

Steg 1 Läs i *kursboken*, avsn. *2.4 Arrays* (sid 30).

Steg 2 Skapa tre arrays, en till varje tävlandes poäng.

Steg 3 Läs i *kursboken*, avsn. *2.5 Hantering av slumptal* (sid 35), speciellt om *Slumptal inom ett intervall* (sid 36).

Steg 4 Fyll varje array från ***Steg 2*** med slumpvärden i intervallet mellan 0 och 10 (domarnas poänggivning).

Steg 5 Skapa tre variabler, en för varje tävlandes totalpoäng, och initiera dem med summan av varje tävlandes poäng (från ***Steg 4***).

Steg 6 Bestäm den största bland de tre tävlandes totalpoäng. Använd funktionen `max()` som behandlas i *kursboken* (sid 46) resp. på *lektion 5*.

Steg 7 Skriv ut både de tävlandes totalpoäng och vilken av dem som vunnit gymnastiktävlingen.

Kapitel 3

Kontrollstrukturer

Ämne	Sida	Program
3.1 Vad är kontrollstrukturer?	40	
3.2 Enkel selektion: `if`-satsen	41	`SimpleIf`
- Villkor	42	
- Jämförelseoperatorer	43	
- Bestämning av max/min	44	`Max`
- Modularisering	45	
- Funktionen max()	46	`MaxFct`
- Om funktioner	46	
3.3 Tvåvägsval: `if`-else-satsen	47	`IfElse`
- Modulooperatorn	49	
- Tillämpningar av modulo	49	
3.4 Flervägsval	50	
- `if-else`-stegen	51	`GissaTal`
- `switch`-satsen	50	`Switch`
3.5 Efter-testad repetition: `do`-satsen	55	`Collatz`
3.6 För-testad repetition: `while`-satsen	59	`Sum_while`
- Evighetsloop	60	
3.7 Bestämd repetition: `for`-satsen	61	`Sum_for`
- `for`-satsens struktur	61	
- En tillämpning av `for`-satsen	63	`Borr`
Övningar till kap 3	65	

3.1 Vad är kontrollstrukturer?

Kontrollstrukturer är algoritmers byggstenar och programmeringens mest grundläggande verktyg. Det finns generella strukturer i alla algoritmer som är oberoende av det aktuella problemet. Därför kan de användas som byggstenar vid beskrivning av *alla* algoritmer som i sin tur ligger till grund för alla datorprogram, oberoende av programmeringsspråk.

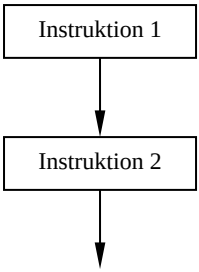
Kontrollstrukturer består av tre grundläggande typer:

- **Sekvens (följd)**
- **Selektion (val)**
  - Enkel selektion
  - Tvåvägsval
  - Flervägsval
- **Repetition (upprepning)**
  - Förtestad repetition
  - Eftertestad repetition
  - Bestämd repetition

Alla datorprogram är kombinationer av dessa tre typer av kontrollstrukturer. I detta kapitel ska vi gå igenom alla tre och lära oss hur de kodas i JavaScript. Kontrollstrukturer används och är i princip uppbyggda enligt samma logik i alla programmeringsspråk. Både C/C++:s, Javas och C#:s kontrollstrukturer har – när det gäller syntaxen – tagits över från och är i princip identiska med Algol/Pascal bortsett från några detaljer. Ännu längre tillbaka i historien kan man hitta deras spår i de första strukturerade språken.

Sekvens (följd)

En *sekvens* är en följd av instruktioner (bilden till höger) – den enklast möjliga strukturen som tänkas kan. Alla våra program hittills består endast av sekvenser. Varje instruktion kan i sin tur innehålla andra kontrollstrukturer. Så även om sekvensen är en enkel struktur, kan nästlade sammansättningar av den med sig själv (underinstruktioner) och andra kontrollstrukturer ändå ge en ganska invecklad bild.



Selektion (val)

Kontrollstrukturen *selektion* är mer komplex än sekvens. Beroende på antalet alternativ man kan välja mellan tre olika varianter: *Enkel selektion, två- eller flervägsval*. Vi börjar med den första.

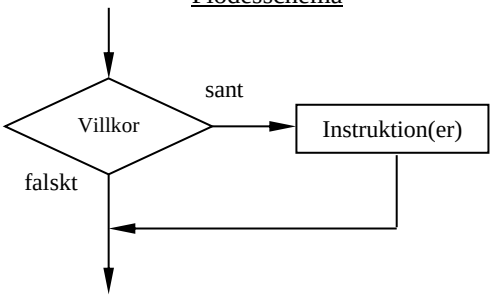
3.2 Enkel selektion: `if`-satsen

Enkel selektion är ett val utan alternativ. Ett villkor avgör valet. Är villkoret sant, utförs en eller flera instruktioner. Är villkoret falskt, görs ingenting.

Pseudokod

OM *villkor* uppfyllt
instruktion(er)

Flödesschema



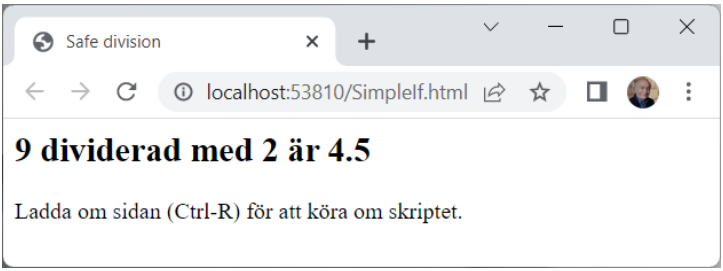
I JavaScript kallas den enkla selektionen för **if**-sats och kodas generellt på följande sätt:

```
if (villkor)
{
    sats(er)
}
```

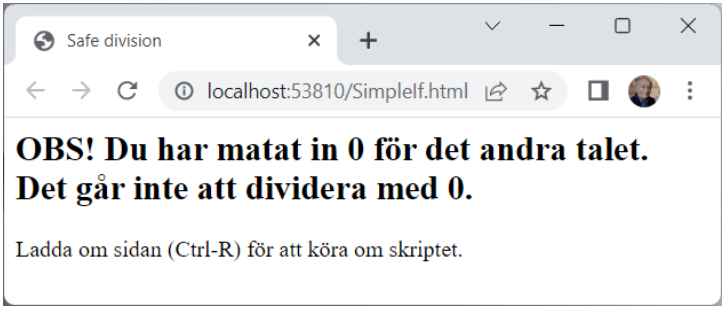
Första raden är **if**-satsens *huvud*. Resten är **if**-satsens *kropp* som omsluts av klammerparenteserna **{** och **}** som vi i fortsättningen kommer att kalla kort *klamrar*, ibland *måsvingar*. Om kroppen består endast av en sats kan klamrarna utelämnas vilket vi utnyttjar i följande program:

```
1  <!-- SimpleIf.html
2       Dividerar endast om det som ska divideras med, inte är 0
3       Enkel selektion: if-satsen med EN sats: utan klamrar -->
4  <title>Safe division</title>
5  <script>
6     no1 = parseInt(prompt('Mata in ett tal'))   // Inläsning
7     no2 = parseInt(prompt('Mata in ett tal till'))
8
9     if (no2 != 0)
10       document.writeln('<h2>' + no1 + ' dividerad med ' + no2 +
11                        ' är ' + no1 / no2 + '</h2>')
12    if (no2 == 0)
13       document.writeln('<h2>OBS! Du har matat in 0 för det' +
14                        ' andra talet.<br>Det går inte att ' +
15                        ' dividera med 0.</h2>')
16 </script>
17 Ladda om sidan (Ctrl-R) för att köra om skriptet.
```

Programmet läser in två tal och dividerar dem med varandra. `if`-satserna gör att division endast sker om det andra talet `no2` (det som ska divideras med) *inte* är `0`, för att förhindra den matematiskt odefinierade divisionen med `0`. Följande resultat får man när man matar in ett värde skilt ifrån `0` till det andra talet:



Matas in däremot `0` till det andra talet uppstår följande:



Inmatning av `0` till det andra talet genererar ett egendefinierat ”felmeddelande”. Låt oss titta närmare på den första `if`-satsens huvud i programmet `SimpleIf` (rad `9`):

```
if (no2 != 0)
```

betyder i termer av pseudokod: `OM no2` *är skilt ifrån* `0`

Satsen inleds med det reserverade ordet `if` följt av ett *villkor (condition)* inom parentes. Observera att parenteserna tillhör syntaxen och inte får inte utelämnas.

Villkor

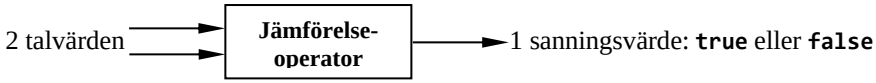
`if`-satsens huvudingrediens är alltid ett *villkor*, t.ex. `no2 != 0`. Dubbeltecknet `!=` betyder *icke lika med* och måste skrivas *utan* mellanslag: Är `no2` skilt ifrån `0`, ja eller nej? Man kan alltså uppfatta ett villkor som en *fråga* som endast kan besvaras med ja eller nej. En annan aspekt är att uppfatta ett villkor som en *utsaga* som endast kan vara sann eller falsk. Till skillnad från en *sats* som är en instruktion som ska *utföras*, kan ett villkor inte utföras, utan endast *testas*, för att få ut svaret sant eller falskt. T.ex. testar villkoret `no2 != 0` om `no2` är skilt ifrån `0`. Variabeln `no2`:s värde *jämförs* med `0`. Finns det icke-likhet mellan dem är villkoret sant, annars falskt. Därför kallas `!=` för en *jämförelseoperator*. Det finns fler sådana:

Jämförelseoperatorer

Jämförelseoperatorer sätts mellan *två* variabler för att jämföra deras värden. De används endast i *villkor*, inte i instruktioner. Det är avgörande att skilja mellan begreppen *villkor* och *instruktion*. Här är de vanligaste jämförelseoperatorerna:

<	mindre än
<=	mindre än eller lika med
>	större än
>=	större än eller lika med
==	lika med
!=	icke lika med

De jämför två talvärden med varandra och returnerar jämförelsens resultat som ett s.k. *sanningsvärde* dvs sant eller falskt, **`true`** eller **`false`** som är reserverade ord.



Sanningsvärdena **`true`** och **`false`** är de enda värden som villkor kan anta varför jämförelseoperatorer används för att skriva villkor. Exempel på villkor formulerade med jämförelseoperatorer är:

```
number == 0
number != 0
7 > 5
guessedNo <= 17
```

Observera att de jämförelseoperatorer som är dubbeltecken, inte får innehålla mellanslag, annars tolkas de som respektive tecken och inte som jämförelseoperatorer. T.ex. är == symbolen för *lika med*. Redan på sid 27 pratade vi om skillnaden mellan likhet och tilldelning och poängterade att = i JavaScript inte betyder likhet utan tilldelning. Här har vi symbolen **`==`** för *likhet*. Medan tilldelningsoperatorn = förekommer i instruktioner (satser), används jämförelseoperatorn **`==`** i villkor, t.ex. i villkoret till **`if`**-satsen i programmet **`SimpleIf`**, rad **`12`** (sid 41).

Så långt om **`if`**-satsens *huvud*. Sedan kommer **`if`**-satsens kropp som i programmet **`SimpleIf`** består av *en* enda utskriftssats. Därför kan klamrarna **`{ }`** kring kroppen utelämnas. Men det vore inte heller fel att skriva dem. Villkorets sanningsvärde avgör nu om kroppen dvs utskriftssatsen utförs eller ej. Är variabeln **`no2`**:s värde icke lika med **`0`**, utförs kroppen. Observera också att hela utskriftssatsen är indragen för att markera att denna tillhör **`if`**-satsen och att den bildar **`if`**-satsens kropp – en kodstil som hör till god programmeringssed och höjer kodens läslighet.

Den andra **`if`**-satsens huvud i programmet **`SimpleIf`**:

```
if (no2 == 0)
```

betyder i termer av pseudokod: **`OM no2`** *är lika med* **`0`**

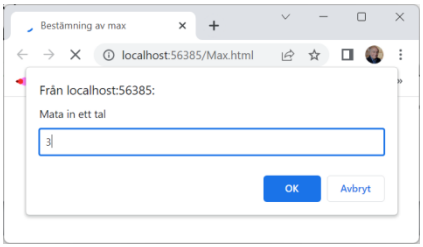
Precis som **!=** är även dubbeltecknet == (utan mellanslag) en jämförelseoperator, men står för *lika med*. Observera skillnaden mellan likhet som kodas med *två* likhetstecken == och tilldelning vars kod är *ett* likhetstecken =. Även den andra **if**-satsens kropp är en utskriftssats som skriver ut ett felmeddelande om värdet **0** matas in som andra tal. På så sätt utförs inte division med **0**, för divisionen förekommer endast i den första **if**-sats som inte utförs eftersom dess villkor blir falskt, när man matar in **0** som andra tal.

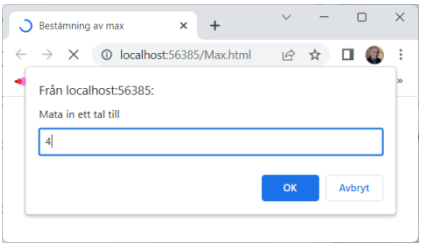
Bestämning av max/min

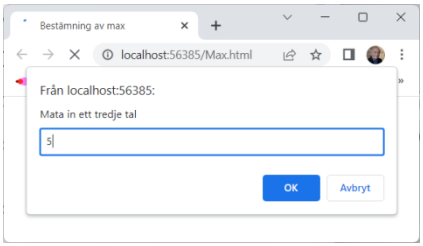
I programmet **SimpleIf** (sid 41) användes två **if**-satser, för att avgöra om ett tal var jämnt eller udda. Nu ska vi skriva ett nyttigt program som bestämmer det största (minsta) värdet bland 3 inmatade tal. Nyttigt, därför att vi kommer att ha användning av det bl.a. i inlämningsuppgiften (sid 38). Sedan ska vi använda detta exempel för att precisera vår kunskap om *modularisering* som nämndes inledningsvis i boken (sid 7), och lära oss att själva definiera *funktioner* i JavaScript.

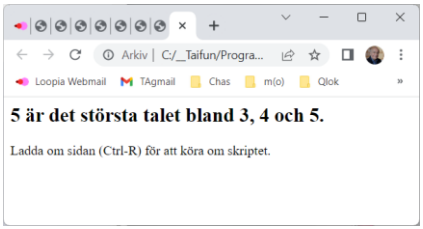
```
1  <!-- Max.html
2       Läser in 3 tal och bestämmer det största bland dem
3       Två enkla if-satser löser problemet  -->
4  <title>Bestämning av max</title>
5  <script>
6    no1 = parseInt(prompt('Mata in ett tal'))     // Inläsning
7    no2 = parseInt(prompt('Mata in ett tal till'))
8    no3 = parseInt(prompt('Mata in ett tredje tal'))
9
10   max = no1          // Vi antar att no1 är störst
11   if (no2 > max)
12      max = no2       // Byter till no2 om no2 är större
13
14   if (no3 > max)
15      max = no3       // Byter till no3 om no3 är större
16
17   document.writeln('<h2>' + max + ' är det största talet ' +
18         'bland ' + no1 + ', ' + no2 + ' och ' + no3 + '.</h2>')
19 </script>
20
21 Ladda om sidan (Ctrl-R) för att köra om skriptet.
```

Själva algoritmen att hitta det största bland tre tal, är kodad på raderna **10-15** med två enkla **if**-satser: Först antar vi att **no1** är det största talet och tilldelar det variabeln **max**. Det behöver inte stämma. Den första **if**-satsen (rad **11-12**) testar detta antagande genom att kolla om **no2** är större än **max** och därmed även större än **no1**. Om det är fallet byts **max**-”rollen” från **no1** till **no2**. Samma sak gör den andra **if**-satsen med **no3** (rad **14-15**). Slutligen kommer **max**-”rollen” ges till det tal som är störst av alla tre. Resten är inläsning och utskrift:



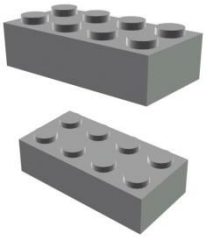






För att hitta det *minsta* talet bland tre inmatade behöver man i **`if`**-satsernas villkor (rad **`11`** & **`14`**) bara byta ut jämförelseoperatorn **`>`** mot **`<`** . Självklart borde man, för att följa god programmeringsstil, även byta ut variabelnamnet **`max`** mot **`min`** och ändra texten i utskriftssatsen.

Modularisering



Modularisering innebär att bryta ner ett problem i mindre, återanvändbara delar, s.k. *moduler*, jämförbart med Legobitar. I JavaScript kallas de för *funktioner*.

Programmet **`Max`** (sid 44) löser problemet att bestämma det största talet bland tre givna tal. Men detta problem kan även förekomma i andra sammanhang. Och då vill man helst använda den redan befintliga algoritmen som är kodad på raderna **`10-15`**, utan att behöva återuppfinna hjulet.

Ett exempel på ett sådant behov är vår inlämningsuppgift (sid 38). Där ska man bestämma den största bland tre tävlandes totalpoäng. Som ledning anges i uppgiften att man ska använda funktionen **`max()`**. Denna funktion ska vi skriva nu genom att ta raderna **`10-15`** från programmet **`Max`**, definiera dem som en funktion, binda in funktionen i ett program och anropa den därifrån. Det nya programmet **`MaxFct`** ska åstadkomma samma sak som programmet **`Max`**. På så sätt modulariserar vi programmet **`Max`**. Samtidigt blir funktionen **`max()`** vår första *egendefinierade* funktion i JavaScript. Hittills hade vi endast *anropat* redan fördefinierade funktioner.

Funktionen max( )

```
1  <!-- MaxFct.html
2       Definierar och anropar funktionen max() som bestämmer
3       det största bland tre tal  -->
4  <title>Max med funktion</title>
5  <script>
6
7     function max(a, b, c)   // Definierar funktionen max()
8     {
9        tmp = a              // Antar att a är störst
10       if (b > tmp)
11          tmp = b           // Byter till b om b är större
12       if (c > tmp)
13          tmp = c           // Byter till c om c är större
14       return tmp           // Returnerar tmp till max()
15    }
16
17    no1 = parseInt(prompt('Mata in ett tal'))  // Inläsning
18    no2 = parseInt(prompt('Mata in ett tal till'))
19    no3 = parseInt(prompt('Mata in ett tredje tal'))
20    noMax = max(no1, no2, no3)       // Anropar funktionen max()
21    document.writeln('<h2>' + noMax + ' är det största talet ' +
22         'bland ' + no1 + ', ' + no2 + ' och ' + no3 + '.</h2>')
23 </script>
24
25 Ladda om sidan (Ctrl-R) för att köra om skriptet.
```

Om funktioner

Rad **7** kallas för funktionens *huvud* och inleds med det reserverade ordet **`function`** (sid 10). Funktionens *namn* är **`max()`**. Parentesen **`(a, b, c)`** kallas för *parameterlistan*. **`a, b`** och **`c`** är funktionens *formella parametrar*, medan **`no1, no2`** och **`no3`** som står i funktionsanropet (rad **20**), kallas för *aktuella parametrar*. Vid anropet kopieras de inlästa värdena från de aktuella till de formella parametrarna. På så sätt hamnar de i funktionen, där deras största värde bestäms.

Efter huvudet står funktionens *kropp* inom måsvingar (rad **8-15**). Kroppen avslutas med en s.k. **`return`**-sats som med hjälp av variabeln **`tmp`** returnerar det största värdet till namnet **`max()`**. På så sätt hamnar funktionens *returvärde* i programmet, när funktionen anropas på rad **20**. Eftersom namnet **`max()`** bär returvärdet måste anropet inbakas i en tilldelningssats, så att variabeln **`noMax`** kan ta emot detta värde som slutligen skrivs ut (rad **21**). Att funktionen **`max()`** innehåller en **`return`**-sats ger upphov till att kalla **`max()`** för en *funktion med returvärde*. Det finns i JavaScript även *funktioner utan returvärde*. Dessa saknar **`return`**-sats.

Prorammet **`MaxFct`** producerar samma utskrift som programmet **`Max`** (sid 44).

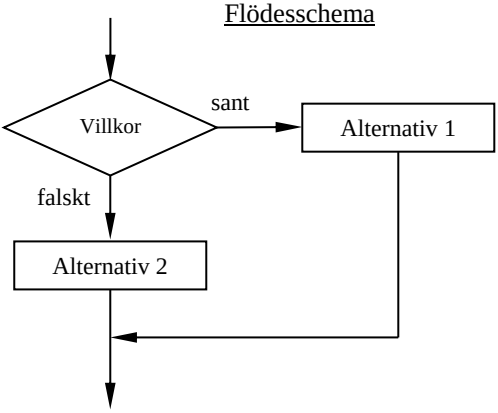
3.3 Tvåvägsval: if-else-satsen

Tvåvägsval är ett val mellan två alternativ. Valet görs med ett enda villkor. Är villkoret sant, utförs en eller flera instruktioner som vi kallar för *alternativ 1*. Är villkoret falskt, utförs – till skillnad från **`if`**-satsen – en annan uppsättning instruktioner som vi kallar för *alternativ 2*. Så här kan tvåvägsvalet beskrivas:

Pseudokod

`OM` *villkor* uppfyllt
    alternativ 1
`ANNARS`
    alternativ 2

Flödesschema



Alternativ 1 och
Alternativ 2 är två olika instruktioner.

Endast ett av de två alternativen kommer att utföras, beroende på villkorets sanningsvärde. Sanningsvärdena sant och falskt utesluter varandra – och därmed även de båda alternativen. Därför går flödet i flödesschemat, som visas med pilarna, efter alternativ 1 inte till eller före utan *efter* alternativ 2. Det vore logiskt fel att leda pilen till ett ställe *före* alternativ 2.

I JavaScript kallas tvåvägsvalet för **`if-else`**-sats och kodas på följande sätt:

```
if (villkor)
{
    sats(er)1
}
else
{
    sats(er)2
}
```

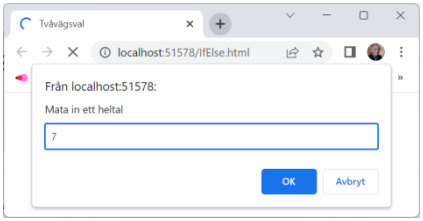
Om **`if`**- eller **`else`**-blocket består endast av en sats kan klamrarna **`{`** och **`}`** utelämnas. Anta att båda block består bara av en sats, då förenklas formen:

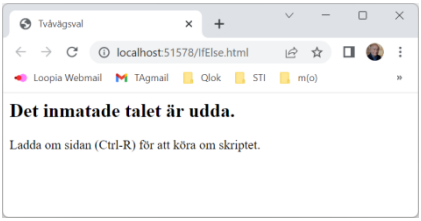
```
if (villkor)
    sats1
else
    sats2
```

Följande exempel behandlar **`if-else`**-satsen med endast en sats i resp. del:

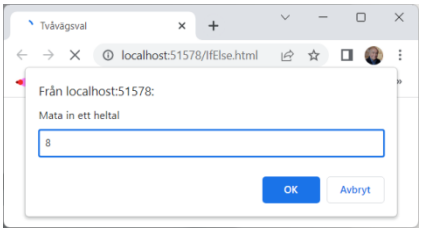
```
1  <!-- IfElse.html
2       Läser in ett heltal och avgör om det är jämnt eller udda
3       Tvåvägsval: if-else-satsen med EN sats i if-else-delen -->
4
5  <title>Tvåvägsval</title>
6
7  <script>
8     no = parseInt(prompt('Mata in ett heltal'))   // Inläsning
9
10    if (no % 2 == 0)
11       document.writeln('<h2>Det inmatade talet är jämnt.</h2>')
12    else
13       document.writeln('<h2>Det inmatade talet är udda.</h2>')
14
15 </script>
16 Ladda om sidan (Ctrl-R) för att köra om skriptet.
```

Körexempel av programmet **`IfElse`** med ett udda tal som inmatning ger:





Med ett jämnt tal som inmatning får vi:



Det egentliga jobbet – nämligen att avgöra mellan *jämnt* och *udda* – har gjorts med hjälp av en operator som kallas för *modulooperatorn:*

Modulooperatorn %

Symbolen **%** har i JavaScript ingenting med procenträkning att göra utan står för ett nytt räknesätt som kallas för *modulo*. Modulo är en heltalsoperation. Man dividerar två heltal, tar resten och ignorerar resultatet: **9** dividerat med **2** ger **4**, rest **1**. Därför: **9** *modulo* **2** ger **1**. Med symboler: **9 % 2 = 1**. Modulooperationen ignorerar **4** och tar resten **1**. En användning av modulo är: P.g.a. **9 % 2 = 1** är **9** udda. Däremot är **8 % 2 = 0**, eftersom **8** dividerat med **2** ger resultatet **4** och resten **0**. Därför är **8** ett jämnt tal. Alla jämna tal ger rest **0** vid heltalsdivision med **2**. Alla udda tal ger rest **1** vid heltalsdivision med **2**. Modulo ger resten vid heltalsdivision. Man kan uppfatta modulo även som en upprepad subtraktion: Man drar av **2** från **9** så många gånger det bara går och tar det som blir kvar. Fyra gånger går det att ta bort **2** från **9**, kvar blir **1**. Därför är **9 % 2 = 1**. Generellt innebär att *räkna modulo* **a** att man drar av alla multipler av **a** och behåller resten: **33** *modulo* **6** ger **3**, därför att man får **3**, när man drar av **5** gånger **6**, dvs **30**, från **33**.

Tillämpningar av modulo

Det finns många tillämpningar av modulooperatorn:

1. I progammet **IfElse** (rad **10**) tillämpas modulo i **if**-satsens villkor:

   ```
   no % 2 == 0
   ```

   för att avgöra att talet **no** är jämnt: Delar man **no** med **2** och resten är **0**, så är **no** jämnt delbart med **2** och därmed jämnt.

2. En rolig och enkel användning av modulooperatorn är följande exempel:

   > Idag är fredag och du vill träffa din kompis om **11** dagar.
   > Vilken veckodag blir det?

   Vi numrerar veckodagarna stigande från **1** med början på måndag, så att fredag blir den **5**:e veckodagen. Man får svaret på frågan ovan genom att *räkna modulo* **7**:

   ```
   (5 + 11) % 7 = 2
   ```

   Dvs veckodagen i frågan är **2**:a veckodagen, nämligen **tisdag**. Med andra ord man lägger till aktuell veckodag, antalet dagar och räknar modulo **7**. I själva verket handlar det om en omvandling av det decimala talsystemet med basen **10** och siffrorna **0-9** – det system vi är vana vid att räkna med – till *veckodagarnas system* dvs till *talsystemet med basen* **7** som använder sig av siffrorna **0-6**.

3. En annan tillämpning av modulo är omvandling mellan olika talsystem, t.ex. mellan det decimala och binära talsystemet. Generellt är modulo nyckeloperationen vid omvandling mellan olika system.

4. I matematiken används modulo bl.a. för att bestämma den största gemensamma delaren av två heltal (Euklides algoritm).

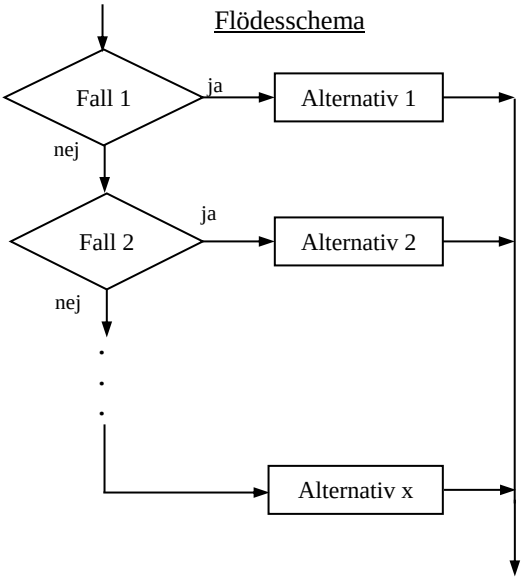
3.4 Flervägsval

Flervägsval är ett val mellan fler än två alternativ. Strukturen och logiken kan beskrivas så här:

Pseudokod

`VÄLJ` *fall* ur
  fall 1: alternativ 1
  fall 2: alternativ 2
    .
    .
    .
  Annars: alternativ x

Flödesschema



Alternativ 1, 2, … innebär olika *instruktioner* eller olika uppsättningar instruktioner och Fall 1, 2, … motsvarar olika *villkor*.

Observera att det logiska flödet – symboliserat med pilarna – går efter varje *fall* till ett *alternativ*, för att därefter *lämna* hela flödesschemat. Dvs flödet går efter varje fall *inte* till nästa fall. I slutet, när alla fall är avklarade, behöver inget nytt villkor formuleras, därför att Alternativ x utförs när Fall 1, Fall 2, … *inte* föreligger.

Det finns olika sätt att implementera flödesschemat ovan i kod. I praktiken har det visat sig att följande två koncept är mest effektiva och användbara i programmeringen oavsett programmeringsspråk:

- **`if-else`-stegen**
- **`switch`-satsen**

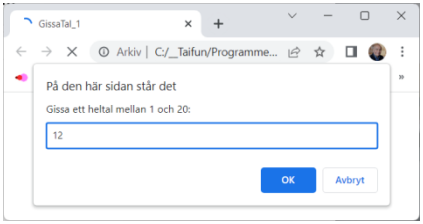
`if-else`-*stegen*

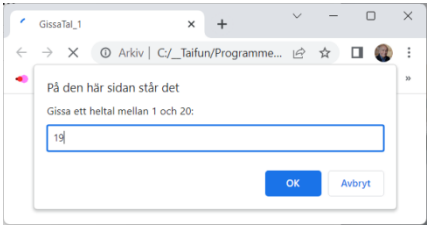
Låt oss titta på följande exempel av ett trevägsval som använder den s.k. **`if-else`**-stegen. Användaren ska gissa fram programmets hemliga tal **`17`**. Man gissar inom intervallet **`[1, 20]`**, får sedan hjälp om det gissade talet var mindre än, större än eller lika med det hemliga talet. Just nu måste vi nöja oss med en spelomgång, därför att vi inte lärt oss ån att koda loopar.

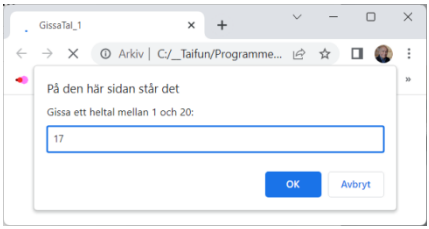
Här kommer *Gissa tal-spelet* i en första version, som innehåller ett val mellan tre alternativ, *fall 1:* det gissade talet är lika med, *fall 2:* mindre än, *fall 3:* större än programmets hemliga tal **17**:

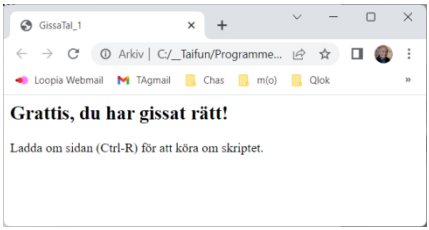
```
<!-- GissaTal.html
     Låter användaren gissa programmets hemliga tal secret
     Trevägsval med en if-else stege -->
<title>GissaTal</title>
<meta charset="UTF-8">          <!-- För de svenska tecknen -->
<script>
   secret = 17                  // Programmets hemliga tal
                                // Inläsning av en gissning:
   guess = parseInt(prompt('Gissa ett heltal mellan 1 och 20:'))

   if (guess == secret)
      document.writeln('<h2>Grattis, du har gissat rätt!</h2>')
   else if (guess < secret)     // Ger hjälp för nästa körning:
      document.writeln('<h2>Fel: ' + guess + ' < hemliga ' +
                       ' talet<br>Gissa högre nästa gång.</h2> ')
   else
      document.writeln('<h2>Fel: ' + guess + ' > hemliga ' +
                       ' talet<br>Gissa lägre nästa gång.</h2> ')
 </script>

Ladda om sidan (Ctrl-R) för att köra om skriptet.
```

De tre relevanta testen av programmet **GissaTal** med gissningar mindre än, större än och lika med **17** ger:









`switch`-*satsen*

Flervägsvalets flödesschema som visades på sid 50 kan kodas på olika sätt. Ett sätt var **`if-else`**-stegen som demonstrerades i programmet **`GissaTal`** på förra sidan. Ett annat sätt är **`switch`**-satsen vars generella struktur kan beskrivas så här:

```
switch (uttryck)
{
    case konstant1 :
        sats(er)1
        break
    case konstant2 :
        sats(er)2
        break
            .
            .
            .
    default:
        sats(er)x
}
```

Första raden är **`switch`**-satsens *huvud*. Resten är **`switch`**-satsens *kropp* som består av ett block. All kod som skrivs mellan måsvingarna **`{ }`** kallas för *block*.

Med *uttryck* i huvudet menas ett aritmetiskt uttryck vars värde får bara vara av typ tal eller tecken.. När **`switch`**-satsen exekveras, jämförs detta uttryck en i taget med de konstanter som står efter **`case`**. Jämförelsen görs på likhet och innebär följande när man översätter alla **`case`** till **`if`**:

```
if (uttryck == konstant1)
if (uttryck == konstant2)
            .
            .
            .
```

Så blir villkoren som är dolda i **`switch`**-satsen avslöjade: Man ser att de är hård-kodade med operatorn == och inte kan ersättas med andra jämförelseoperatorer.

Uttryckets och konstanternas värden jämförs med varandra enbart på likhet. Om likhet föreligger, kommer man in i **`switch`**-satsens kropp. Alla satser fr.o.m. **`case`** utförs, tills **`break`** kommer eller kroppen slutar.

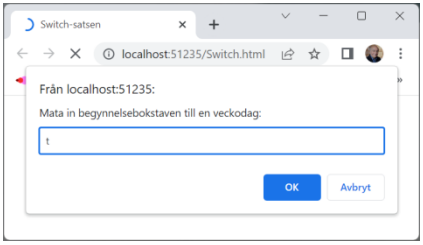
Följande programexempel demonstrerar **switch**-satsen: Vi läser in begynnelsebokstaven till en veckodag och det fullständiga veckodagsnamnet skrivs ut. I **switch**-satsen väljs ett alternativ av sex. Tisdag och torsdag behandlas i ett fall.

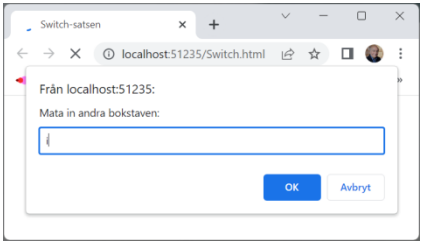
```
<!-- Switch.html
     Demonstrerar flervägsval med switch-satsen
     Kompletterar veckodagen efter inmatning av första bokstaven
     För t(isdag/torsdag) krävs den 2:a bokstaven -->
<title>Switch-satsen</title>
<script>
   letter1 = prompt('Mata in begynnelsebokstaven ' +
                    'till en veckodag:')
   switch (letter1)
   {
      case 's':
         weekday = 'söndag'
         break
      case 'm':
         weekday = 'måndag'
         break
      case 't':
         letter2 = prompt('Mata in andra bokstaven: ')
         if (letter2 == 'i')
            weekday = 'tisdag'
         else
            weekday = 'torsdag'
         break
      case 'o':
         weekday = 'onsdag'
         break
      case 'f':
         weekday = 'fredag'
         break
       case 'l':
          weekday = 'lördag'
          break
       default:
          weekday = 'ingen veckodag'
   }
   document.writeln('<h2>' + letter1 + ' är första ' +
                    'bokstaven till ' + weekday + '. </h2>')
</script>
Ladda om sidan (Ctrl-R) för att köra om skript
```

Programmet utför inte bara de satser som omedelbart följer det **case** där likheten inträffar, utan *alla* satser som följer, ända tills en **break**-sats kommer eller **switch**-satsen avslutas. Har man en gång kommit in i **switch**-satsen via något **case**, stan-

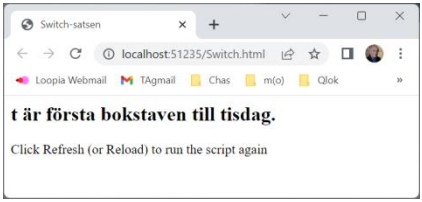
nar man i den utan att likhet mellan uttrycket och konstanten som finns i de efterföljande **case**-satserna testas. Om **switch**-satsen ska välja endast ett enskilt värde bland flera, borde varje **case** avslutas med **break**.

Här ett körresultat för inmatningen av **t** som första bokstav, där en andra inmatning p.g.a. konflikten **t**isdag/**t**orsdag krävs. Testa gärna alla andra alternativ också:





break-satsen



break är både ett reserverat ord i JavaScript och en sats i prgrammet **Switch**. **break** bryter programflödet, dvs i det här fallet lämnar **switch**-satsen. Alla satser mellan **break** och blockets avslutande klammer **}** hoppas över. Detta garanterar ett entydigt val mellan flera alternativ. Användningen av **break** i **switch**-satsen är, vad gäller den formella syntaxen, frivillig dvs man begår inget syntaxfel om man utelämnar **break**. Men om det blir så som man tänkt sig är en helt annan historia, dvs det kan bli *logiskt* fel. Utelämnandet av **break** leder i alla fall att programflödet ”faller ned” till nästa **case**, utan att testa den nya **case**-satsens villkor. I vissa fall kan det dock finnas även logiska skäl att utelämna **break**, där ett entydigt val mellan enstaka värden inte är önskvärt, t.ex. när valet står mellan olika *intervall* och man vill använda ”tomma” **case**-satser.

default på rad **33** är motsvarigheten till **else**. Om ingen likhet påträffats i någon **case**-sats, utförs istället de satser som följer efter **default**. På så sätt har man möjligheten att skriva kod som dokumenterar det just inträffade. Ofta väljer man att skriva ut någon form av felmeddelande. Användningen av **default**-satsen är frivillig. Den kan utelämnas i **switch**-satsen, men rekommendationen är att utnyttja möjligheten till ett alternativ till alla **case**-satser. Även användningen av **break** som sista sats i **default**-blocket, är frivillig. Den avslutande klammer **}** i **switch**-satsen ersätter **break**, vilket vi har unyttjat i programmet **Switch**.

3.5 Efter-testad repetition: do-satsen

Datorn har några egenskaper som är helt överlägsna motsvarande egenskaper hos människan: snabbheten, noggrannheten och förmågan att effektivt lagra och hantera stora datamängder samt förmågan att inte bli trött. Datorn kan upprepa en sak miljardtals gånger utan att tappa i noggrannhet. Denna förmåga utnyttjas i stor skala av alla möjliga datorprogram. Och därför har man en speciell kontrollstruktur i algoritmer som beskriver den: *repetitionen* [*], även kallad *loop*. ”Att låta datorn göra jobbet” innebär som regel att datorn utför en repetition. Beroende på hur repetitionen, speciellt hur avslutningsvillkoret, kort kallat *villkoret*, formuleras och var det placeras i loopen skiljer man mellan tre typer av repetition:

- **Efter-testad repetition**
- **För-testad repetition**
- **Bestämd repetition**

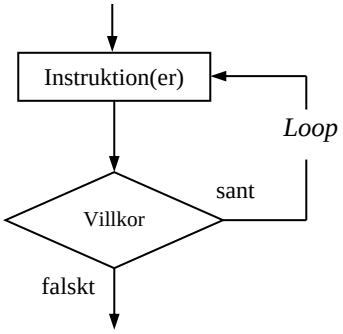
Efter-testad repetition

Det är en loop (upprepningsslinga) där avslutningsvillkoret testas *efter* slingans instruktioner dvs *efter* det som egentligen ska upprepas. Så här kan den formuleras i pseudokod och som flödesschema:

Pseudokod

```
REPETERA
    instruktion(er)
SÅ LÄNGE villkor uppfyllt
```

Flödesschema



I JavaScript inleds den efter-testade repetitionen med det reserverade ordet **do**:

```
do
{
    sats(er)
} while (villkor)
```

do-satsen är en loop där villkoret testas *efter* loopens instruktioner, därför *efter*-testad. Första raden är **do**-satsens *huvud*. Resten är **do**-satsens *kropp* som omsluts av måsvingar **{ }**. Dessa kan utelämnas när kroppen består endast av en sats.

[*] I några böcker kallas repetitionen även för *iteration*. Vi undviker denna term eftersom den används som fackterm i andra sammanhang, t.ex. i numerisk analys.

För att motivera nödvändigheten av loopar tar vi här upp följande känd algoritm som ett exempel på hur **do**-satsen kan komma till användning när man implementerar (skriver koden för) algoritmen:

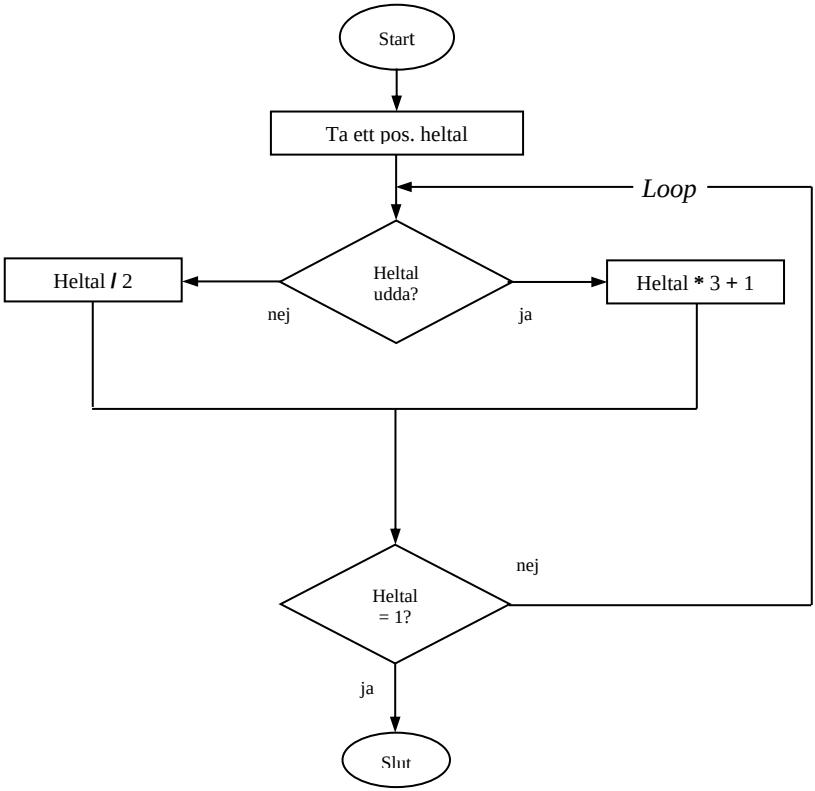
Collatz algoritmen

Lothar Collatz (1910-1990) var professor för tillämpad matematik vid Hamburgs Universitet på 60-talet. Som ung student ställde han upp följande uppgift:

> Tänk dig ett positivt heltal (startvärde).
> Är talet udda multiplicera det med 3 och addera 1.
> Är talet jämnt dividera det med 2.
> Gör samma sak med resultatet. Fortsätt tills du fått 1.

Det visar sig att talföljderna i denna algoritm, även känd som *Collatz-förmodan* alltid slutar med 1 oavsett startvärde. Förmodan heter det eftersom påståendet är matematiskt hittills obevisat. Så här kan flödesschemat för denna algoritm se ut:

Flödesschema



Flödesschemat visualiserar algoritmens logiska struktur som är grundläggande för en korrekt implementering. Men för att slutligen koda kan det vara fördelaktigt att formulera algoritmen även som pseudokod som ligger närmare programkoden än flödesschemat.

Pseudokoden till Collatz algoritmen *

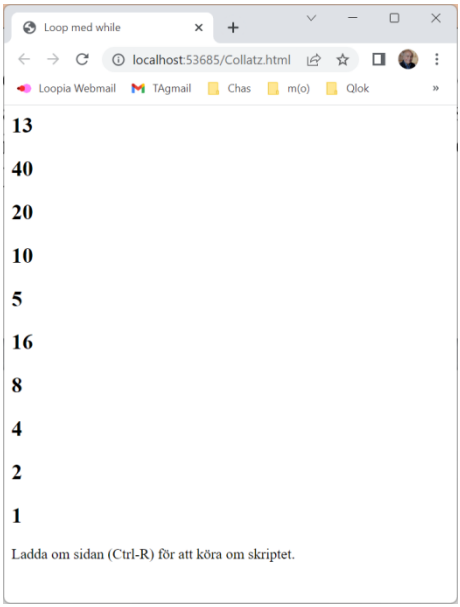
```
Läs in ett positivt heltal
REPETERA
   OM talet är udda
      multiplicera med 3, addera 1
   ANNARS
      dividera talet med 2
   Skriv ut talet
SÅ LÄNGE talet ≠ 1
```

Som man ser har vi redan anpassat pseudokoden till programmering, t.ex. med formuleringar som `Läs in...`, **`REPETERA`** och `Skriv ut...`. I följande program implementeras Collatz algoritmen i JavaScript. För **`REPETERA`** väljer vi **`do`**-satsen:

```
<!-- Collatz.html
     Läser in ett pos. heltal, tar det gånger 3 och adderar 1,
     om det är udda. Delar det med 2 om talet är jämnt.
     Upprepar samma sak med resultatet, tills det blir 1.
     Använder do-sats för repetitionen -->
<title>Loop med do-satsen</title>
<script>
   no = parseInt(prompt('Mata in ett pos.heltal')) // Startvärde
   document.write('<h2>' + no + '</h2>')
   do                                  // do loop börjar
   {
      if (no % 2 == 1)                 // Om no är udda
         no = 3 * no + 1
      else
         no = no / 2
      document.write('<h2>' + no + '</h2>')
   } while (no != 1)                   // do loop slutar
</script>

Ladda om sidan (Ctrl-R) för att köra om skriptet.
```

* Man kan testa Collatz algoritmen i appen *Mattekollen* där den är kodad i Python. Ladda ned appen eller kör den som Webbapp: **app.mattekollen.se → En mobil pythonmiljö**. Eller kör den direkt som webbapp: beta.mattekollen.se/#/app/coding. Prova koden med olika startvärden för att kolla om algoritmens talföljder alltid slutar med 1.

do-satsen är framhävt med vit bakgrund. Talföljden som produceras här, kommer att alltid avslutas med 1, vilket är ett rent empiriskt påstående, som dock varken har motbevisats hittills eller bevisats teoretiskt. Att den avslutas med 1 är oberoende av startvärdet. Här har vi ett körresultat med startvärdet **13**:



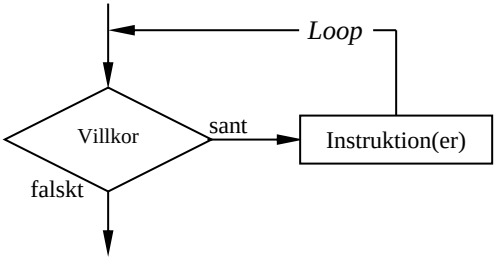
do-satsens arbetssätt, dvs *repetitionen* skiljer sig grundläggande från kontrollstrukturen *selektion* (val) som vi lärde känna tidigare. Medan en selektions alltid går *framåt*, efter den har avgjort valet p.g.a. det styrande villkoret, återvänder en repetition alltid till kontrollstrukturens början, dvs går *tillbaka* och utför koden som står i kroppen en gång till, även detta p.g.a. sitt avslutningsvillkor. Tydligast ser man detta i flödesschemat på sid 56 där programflödet (pilen) går från avslutningsvillkoret tillbaka, för att utföra det hela en gång till.

3.6 För-testad repetition: `while`-satsen

`while`-satsen är en upprepningsslinga där avslutningsvillkoret testas *före* slingans instruktioner dvs *innan* det som ska upprepas. Enda skillnaden gentemot den eftertestade repetitionen med **`do`**-satsen är ordningen mellan villkor och instruktioner. Denna ordning blir nu omvänd:

Pseudokod och ritas som Flödesschema

SÅ LÄNGE *villkor* uppfyllt
instruktion(er)



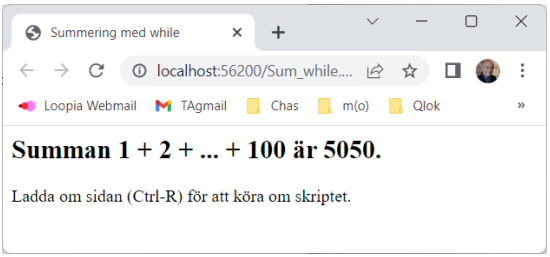
I JavaScript inleds den för-testade repetitionen med det reserverade ordet **`while`** och skrivs generellt på följande sätt:

```
while (villkor)
{
    sats(er);
}
```

Första raden är **`while`**-satsens *huvud*. Resten är **`while`**-satsens *kropp* som omsluts av måsvingar **`{ }`**. Om kroppen består endast av en sats kan måsvingarna utelämnas. Här följer ett exempel med två satser i kroppen och därför med måsvingar:

```
1  <!-- Sum_while.html
2       Beräknar och skriver ut summan 1 + 2 + ... + 100
3       För-testad repetition: while-satsen -->
4  <title>Summering med while</title>
5  <script>
6     sum = 0
7     term = 1
8     while (term <= 100)            // while loop börjar
9     {
10       sum = sum + term
11       term++                      // term ökar med 1
12    }                              // while loop slutar
13    document.write('<h2>Summan 1 + 2 + ... + ' + (term - 1) +
14                                      ' är ' + sum + '.</h2>')
15 </script>
16 Ladda om sidan (Ctrl-R) för att köra om skriptet.
```

Hela **while**-loopen är framhävt med vit bakgrund i programmet **Sum_while**.



Här ett körexempel:

Det är enkelt att ändra sluttermen **100** till lägre eller högre. Ännu bättre vore det förstås att låta sluttermen vara en variabel som läses in, så att man kan beräkna vilka summor som helst, se övn 3.13 (sid 66).

Raden **11** innehåller koden **term++** sm betyder amma sak som **term = term + 1**, dvs ökning av variabeln **term**:s värde med **1**. Koden **++** kallas för ökningsoperatorn och kan sättas före eller efter ett variabelnamn. Att vi i utskriftssatsen på rad **13** använt uttrycket **term - 1**, för att skriva ut sluttermen **100**, beror på att variabeln **term** har värdet **101** när koden har lämnat **while**-loopen på rad **12**. Det är just därför att **101** inte längre är **<= 100** stoppas loopen. Därför måste vi, för att skriva ut **100**, skicka uttrycket **term - 1** till utskrift.

while-satsen är den enklaste varianten av loop i JavaScript. Vi vill använda den för att illustrera en företeelse som man brukar råka ut för när man jobbar med loopar:

Evighetsloop

I programmet **Sum_while** är **while**-satsens avsutningsvillkor **term <= 100**. Om detta villkor vore sant från början och förblev sant hela tiden, skulle satserna på raderna **10-11** att utföras i all evighet, vilket kallas för *evighetsloop*.

För att undvika en evighetsloop, måste villkoret och satserna formuleras på ett sätt att villkorets sanningsvärde *ändras* i loopens kropp. Villkoret måste *bli* falskt efter några varv. I programmet **Sum_while** har vi åstadkommit detta genom att ha **term++** på rad **11**. Samtidigt är villkoret formulerat som **term <= 100**. Dvs, har man med en lämplig initiering av **term** kommit in i **while**-loopen, kommer **term** att öka med **1** i varje varv, så att den någon gång blir **> 100**. Då stoppas loopen. Glömmer man ökningen **++** och initierar man **term** med ett värde mindre än **100** blir **while**-loopen en evighetsloop.

Omvänt: Är **while**-villkoret falskt från början, görs ingenting. Initieras **term** till ett värde större än **100**, blir villkoret falskt från början och man kommer aldrig in i kroppen (”aldrigslinga”). Programflödet fortsätter vid första satsen *efter* **while**-loopen.

3.7 Bestämd repetition: `for`-satsen

För att snabbt visa **`for`**-satsens arbetssätt vill vi börja med en ren översättning av programmet **`Sum_while`** (sid 59) till en **`for`**-variant. Båda summerar alla heltal från **`1`** till **`100`** och ger samma utskrift som på förra sidan.

```
1  <!-- Sum_for.html
2       Beräknar och skriver ut summan 1 + 2 + ... + 100
3       Bestämd repetition: for-loop (översätting av Sum_while) -->
4  <title>Summering med for</title>
5  <script>
6     sum = 0
7     for (term = 1; term <= 100; term++)    // for-loop börjar
8         sum = sum + term                   //          slutar
9     document.write('<h2>Summan 1 + 2 + ... + ' + (term - 1) +
10                                        ' är ' + sum + '.</h2>')
11 </script>
12 Ladda om sidan (Ctrl-R) för att köra om skriptet.
```

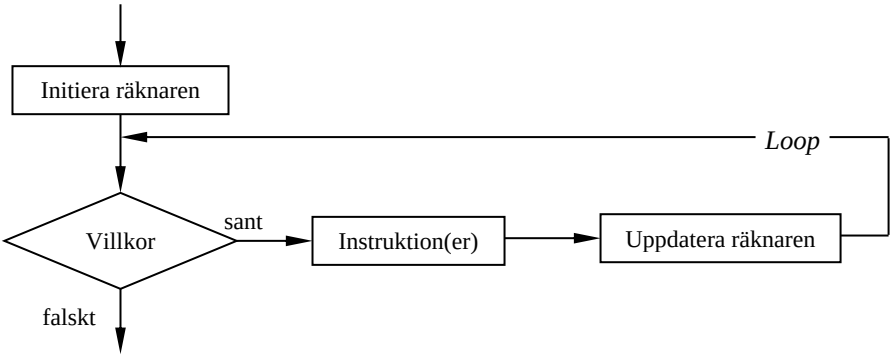
`for`-satsen har endast *en* sats i sin kropp, rad **`8`**. Huvudet, rad **`7`**, är mera invecklat: Initieringen **`term = 1`** och uppdateringen **`term++`** av variabeln **`term`** har flyttats i **`for`**-satsens huvud. Avslutningsvillkoret **`term <= 100`** däremot är kvar på plats: det fanns även i huvudet på **`while`**-loopen, se rad **`8`** i programmet **`Sum_while`** (sid 59).

Eftersom parentesen i **`for`**-satsens huvud (rad **`7`**) nu består av tre delar – initieringen, villkoret och uppdateringen av variabeln **`term`** – måste dessa delar skiljas från varandra med semikolon **`;`** som i JavaScript är skiljetecknet mellan satser. Att vi kan utelämna det i våra andra program beror på att vi skriver våra satser på separata rader. Radslutstecknet kan ersätta semikolonet. Men i **`for`**-satsens huvud är det inte möjligt att bryta rad mellan dessa tre delar. Därför måste vi sätta **`;`**

`for`-satsens struktur

`for`-satsens struktur skiljer sig markant från de hittills behandlade repetitionerna **`do`** och **`while`**. Hos dessa styr endast villkoret antalet repetitioner och man kan få reda på antalet repetitioner endast i efterhand, dvs efter att ha kört programmet. **`for`**-satsen kallas för den *bestämda repetitionen* därför att programmeraren redan vid kodningen *bestämmer* antalet repetitioner. **`for`**-satsen används helst som en loop vars antal repetitioner är känt i förväg. Det kan vara användbart i de fall där man vet hur många gånger en sak ska upprepas. Visserligen finns även i den bestämda repetitionen ett villkor som testas i varje varv, men det finns även en inbyggd möjlighet att styra villkoret och därmed antalet repetitioner med hjälp av en *räknare*, även kallad *styrvariabel*.

Flödeschemat



Flödesschemat åskådliggör den *logiska strukturen* av **`for`**-satsen, medan pseudokoden ligger närmare programkoden.

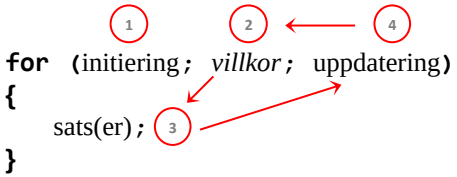
Pseudokoden

Initiera räknaren
`SÅ LÄNGE` *villkor* är uppfyllt
    utför instruktion(er)
    uppdatera räknaren

Nyckelordet **`SÅ LÄNGE`** i denna pseudokod visar att den bestämda repetitionen alltid kan översättas till en **`while`**-sats om man själv tar hand om räknaren. Precis som i **`while`**-satsen har man i princip friheten att formulera villkoret hur som helst. Men eftersom räknaren är inbyggd i strukture, kan man i villkoret jämföra räknaren med slutvärdet, t.ex. så här: *”räknare är mindre än eller lika med slutvärde”*.

Programkoden

`for`-satsen inleds med det reserverade ordet **`for`** och skrivs generellt så här:



De rödmarkerade ringarna och pilarna samt numreringen ska visa i vilken *ordning* de respektive delarna utförs. Denna ordning är nämligen inte identisk med kodbitarnas ordning. Pilarna markerar loopens förlopp. Initieringen görs endast en gång och ingår ej i loopen.

Första raden är **`for`**-satsens *huvud*. Resten är **`for`**-satsens *kropp* som omsluts av klamrarna **`{`** och **`}`**. Om kroppen endast består av en sats kan klamrarna utelämnas.

Räknaren sätts före repetitionen till ett önskat startvärde, för det mesta något heltal, ofta **1**. Detta kallas *initiering* av räknaren dvs den allra första tilldelningen av ett värde till räknaren. Sedan testas ett villkor där man brukar lägga in ett önskat *slutvärde* på räknaren. Därmed är antalet repetitionerna fastlagt, t.ex. till slutvärde minus startvärde om räknaren ökats med **1**. Om villkoret är uppfyllt, t.ex. om räknaren är mindre än slutvärdet, utförs ett antal instruktioner. Sedan görs en *uppdatering* av räknaren, oftast en ökning med **1**, men det är även möjligt att räkna nedåt eller välja ett annat steg än **1**. Allt detta händer i varje varv.

En tillämpning av `for`*-satsen*

Följande problem ska lösas:

> En borrutrustning för bergvärme kan borra 25 m i en viss tomtmark under den 1:a timmen.
>
> Under de följande timmarna minskar borrens prestation med uppskattningsvis 10-20% per timme. Den exakta minskningen är inte känd. Borren går oavbrutet i 8 timmar.
>
> Skriv ett program som simulerar minskningen av borrens prestation efter den1:a timmen med slumptal mellan 10 och 20 och beräknar ett närmevärde till det totala borrdjupet.
>
> Uppskatta borrhålets totala djup efter 8 timmar.
>
> Skriv ut även borrprestationens procentuella minskning vid aktuell körning.

Lösningen:

```
1  <!-- Borr.html
2       Uppskattar det totala borrdjupet för en borrutrustning som
3       går i 8 timmar. Simulerar minskningen av borrens prestation
4       med slumptal inom ett intervall -->
5
6  <title>Uppskattning av borrhål</title>
7
8  <script>
9     totalDepth = 0
10    hDepth = 25                          // 1:a timmens borrdjup
11    a = 10                               // Intervall för pro-
12    b = 20                               // centuell minskning
13    procent = a + parseInt(Math.random()*(b-a+1))   // 10-20%
14    FF = 1 - procent / 100               // Förändringfaktorn
15
```

```
16      for (h = 1; h <= 8; h++)              // 8 timmar
17      {
18         totalDepth = totalDepth + hDepth   // Varje timmes totaldjup
19         hDepth = FF * hDepth               // Varje timmes borrdjup
20      }                                     // efter 1:a timmen
21
22      document.writeln('<h2>Hålet för bergvärmen är ca. ' +
23               parseInt(totalDepth) + ' meter djupt.</h2>')
24      document.writeln('Denna uppskattning baseras på ' + procent +
25               '% minskning av borrprestationen per timme.<br><br>')
26  </script>
27
28  Ladda om sidan (Ctrl-R) för att köra om skriptet.
```

En körning ger följande reultat:

Övningar till kapitel 3

3.1 Marcus som är 1,75 m stor och väger 76 kg vill veta om han är överviktig. Enligt *Body Mass Index (BMI)* anses man vara överviktig om BMI > 25. BMI beräknas med formeln:

$$\text{BMI} = \frac{\text{Vikt i kg}}{\left(\text{Längd i m}\right)^2}$$

Skriv ett program – en *BMI Calculator* – som läser in vikten i kg och längden i cm som heltal och skriver ut **`Överviktig`** om BMI > 25, annars **`OK`**. Som kontroll skriv även ut BMI-värdet.

3.2 Skriv ett program som läser in två hetal och skriver ut **`Rätt ordning`** om det första är mindre än det andra. Skriv ut **`Lika stora`** om de är lika stora och **`Fel ordning`** om det första är större än det andra.

3.3 Vidareutveckla programmet **`Max`** (sid 44) så att det läser in *fyra* tal, hittar och skriver ut det största. Vilken ändring i koden leder till det minsta talet?

3.4 Modularisera din lösning från övn 3.3 genom att definiera den delen av kod som hittar det största talet, som en funktion. Anropa sedan funktionen från ett program. Låt dig inspireras av programmet **`MaxFct`** (sid 46).

3.5 Modularisera din lösning från övn 3.1 genom att definiera BMIs beräkningsformel som en funktion. Anropa funktionen från ett program.

3.6 Ersätt i programmet **`SimpleIf`** (sid 41) de två enkla **`if`**-satserna med en enda **`if-else`**-sats. I övrigt ska programmet göra samma sak som tidigare, nämligen att förhindra division med **`0`**, när man matar in **`0`** för det andra talet.

3.7 Skriv ett JavaScript program som läser in två heltal till variablerna **`a`** och **`b`** och med hjälp av en **`if-else`**-sats avgör om **`a`** är jämnt delbart med **`b`**. Glöm inte att skicka ledtext vid inmatningar. Skriv ut användarvänligt. Testa programmet och visa att **`4592`** är jämnt delbart med **`7`**.

3.8 Idag är det onsdag. Julia vill träffa sin kompis om 13 dagar och vill veta vilken veckodag det blir. Lös problemet generellt:

Skriv ett program som frågar efter aktuell veckodag. Mata in en siffra för veckodagen. Anta att veckans dagar är numrerade från 1-7 med början på måndag. Sedan ska programmet fråga när användaren vill träffa sin kompis och få som svar ett antal dagar. Beräkna och skriv ut den planerade träffens veckodag som nummer. Kör programmet för att lösa Julias problem.

Tips: Läs lösningen till *Tillämpningar av modulo*, ex. **2** (sid 49).

3.9 Följande pseudokod beskriver hur man tar på sig sjal, mössa och handskar beroende på hur kallt det är ute:

```
Start Vinterklädsel
Läs av temperaturen
OM  temperatur <  0
    ta sjal, mössa och handskar
ANNARS OM  temperatur <  5
    ta sjal och mössa
ANNARS OM  temperatur < 10
    ta sjal
ANNARS
    slipper du vinterklädsel
Slut Vinterklädsel
```

Översätt pseudokoden *Vinterklädsel* till ett JavaScript program med hjälp av en **`if-else`**-stege. Låt programmet läsa in ett värde för *temperatur* och avgöra val av klädsel genom att skriva ut ”Ta …”.

3.10 Modifiera programmet **`Collatz`** (sid 57) genom att ersätta **`do`**-loopen med en **`while`**-loop. Modifiera även algoritmens pseudokod och flödesschema, så att de återpeglar algoritmens implementering med **`while`**-loopen.

3.11 Ändra koden i programmet **`Collatz`** (sid 57) så att körningen genererar en evighetsloop.

3.12 Modifiera programmet **`Sum_while`** (sid 59) genom att ersätta **`while`**-loopen med en **`do`**-loop.

3.13 Generalisera programmet **`Sum_while`** (sid 59) genom att ersätta den hårdkodade sluttermen **`100`** med en variabel **`last_term`** som läses in, så att man kan beräkna vilka summor som helst. Testa programmet med olika inläsningar för **`last_term`**, bl.a. med **`10`**, **`1 000`** och **`10 000`**.

3.14 Ändra koden i programmet **`Sum_while`** (sid 59) så att körningen genererar en evighetsloop.

3.15 a) Använd en **`while`**-loop för att skriva ut de första 10 positiva heltalen.

b) Vilken ändring ändring i koden till a) måste göras för att få fram de första 20 positiva heltalen?

3.16 a) Använd en **`for`**-loop för att skriva ut 10 slumptal mellan 0 och 1.

b) Skräddarsy JavaScripts funktion **`Math.random()`** för att slumpa 20 heltal mellan 1 och 50.

3.17 Vi vill simulera tärningskast. Generera i en **`for`**-loop 10 slumptal mellan 1 och 6 och skriv ut dem. Fortsätt med att skriva ut 50 tärningskast.

3.18 a) Skriv ett program som skriver ut de första 10 *jämna* talen.

b) Modifiera a) så att endast de första 10 *udda* talen skrivs ut.

3.19 a) Skriv ett program som summerar de första 10 positiva heltalen.

b) Generalisera a) så att programmet beräknar summan av de första n positiva heltalen där n kan matas in. Testa för n = 100 och 1 000.

c) Skriv ett program som summerar de första n pos. heltalen med formeln:

$$summa = n\,(n+1)\,/\,2$$

Testa om du får samma svar i b) och c) för n = 1 000, 5 000 och 1 000 000.

3.20 Skriv ett program som läser in ett heltal som stegvariabel för att skriva ut tal från 1 till 5 000. Om steget är t.ex. 5 skrivs var femte tal ut.

3.21 Skriv ett program som omvandlar tiden i antal år, månader och veckor till antal dagar. Läs in tre heltal till antal år, månader och veckor. Beräkna och skriv ut sedan användarvänligt hur många dagar det blir totalt.

3.22 Vänd på problemet från övn 3.21: Skriv ett program som läser in ett antal dagar, omvandlar det till antal år, månader, veckor samt resterande dagar och skriver ut resultatet. Använd för denna omvandling följande algoritm och pseudokod.

Algoritmen:

1. Kalla den givna tiden i dagar för totaldagar.
2. Dividera totaldagar med 365 och strunta i resten, så får du det sökta antalet år.
3. Ta resten vid divisionen ovan. Dividera denna rest med 30 och strunta i resten så får du det sökta antalet månader.
4. Ta resten vid divisionen i punkt 3. Dividera denna rest med 7 och strunta i resten så får du det sökta antalet veckor.
5. Resten vid divisionen i punkt 4 är det sökta antalet resterande dagar.

Operationen ”Dividera och strunta i resten” är heltalsdivision och operationen ”Ta resten vid heltalsdivision” är modulo..

Pseudokoden:

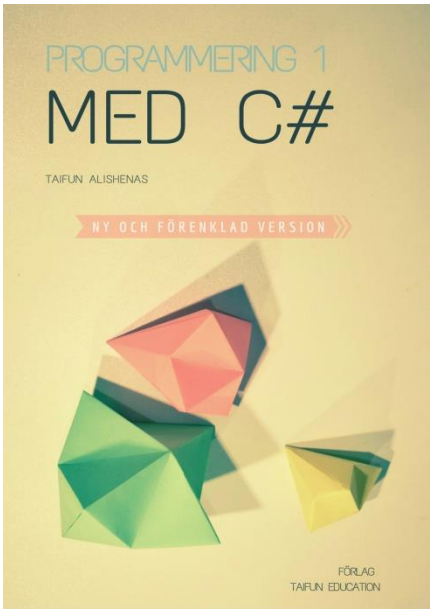
år	=	totaldagar heltalsdividerad med 365
månader	=	(totaldagar modulo 365) heltalsdividerad med 30
veckor	=	((totaldagar modulo 365) modulo 30) heltalsdividerad 7
Resterande dagar	=	((totaldagar modulo 365) modulo 30) modulo 7

3.23 Tillämpa den logiska strukturen i algoritmen och pseudokoden till övn 3.22 för att lösa följande uppgift:

Efter inköp av en vara i en automat ska växeln ges tillbaka i form av ett antal föreskrivna myntslag: 10-kronor, 5-kronor, 1-kronor, 50-öringar [1] och en rest i ören < 50. Skriv ett program som läser in ett växelbelopp i ören, omvandlar det till ett antal 10-kronor, 5-kronor, 1-kronor och 50-öringar samt skriver ut resultatet. Resten i ören < 50 kan vi försumma (resp. avrunda).

[*] 50-öringen finns inte längre i det svenska myntsystemet. Att vi ändå inkluderar den i uppgiften beror inte på nostalgi utan på internationalisering. Vi vill hålla öppen möjligheten för en övergång till andra valutor, t.ex. Euro. Behandlingen av en halv enhet vid omvandling av växelbeloppet till automatens tillåtna mynt inkluderar en programmeringsteknisk finess som kan vara värd att lära sig. Så kan våra program även användas t.ex. för Euron där 50 Cent ersätter 50-öringen.

Programmering 1 med C#



Ur innehållet:

Grundbegrepp i programmering
Datatyper, variabler & tilldelning
Utskrift till grafisk miljö
Windowsprogrammering
C# Console & Windows Applications
Interaktiva grafiska gränssnitt
Kontrollstrukturer
Klasser, objekt och referenser
Metoder
Rekursiva metoder
Sammansatta datatyper: Arrays
Dynamiska arrays: Listor
Sökning & sortering
Kryptering av text
Hantering av slumptal
Undantagshantering
Vad är objektorienterad programmering?
Installation av Visual Studio.NET
Konfiguration av Visual Studio.NET
Projekt i Visual Studio.NET
Övningar & projektuppgifter
Fullständiga lösningar till övningar

www.taifun.se

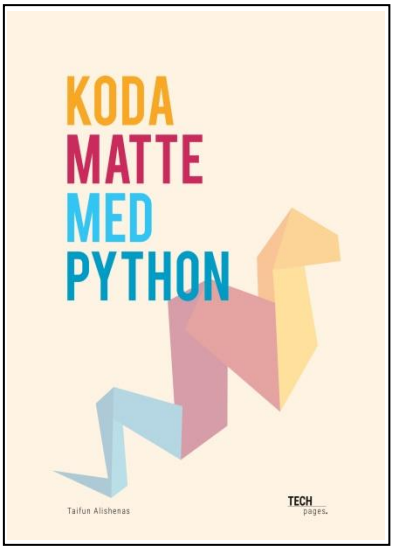
Ladda ned gratis smakprov.

Koda matte med Python

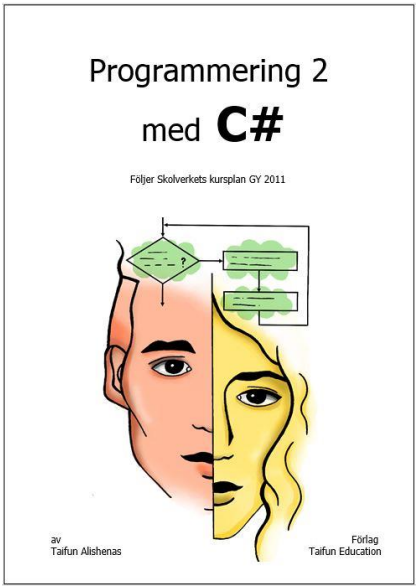
Programmering i matematik

En enkel, pedagogisk lärobok som kompletterar matematikundervisningen med inslag av programmering. Den vägleder både lärare och elever genom att kombinera teori med praktiska övningar och fullständiga lösningar. Boken presenterar ett pedagogiskt koncept om hur programmering kan integreras i kurserna Matematik 1 (a,b,c) och Matematik åk 7-9.

www.kodamatte.se



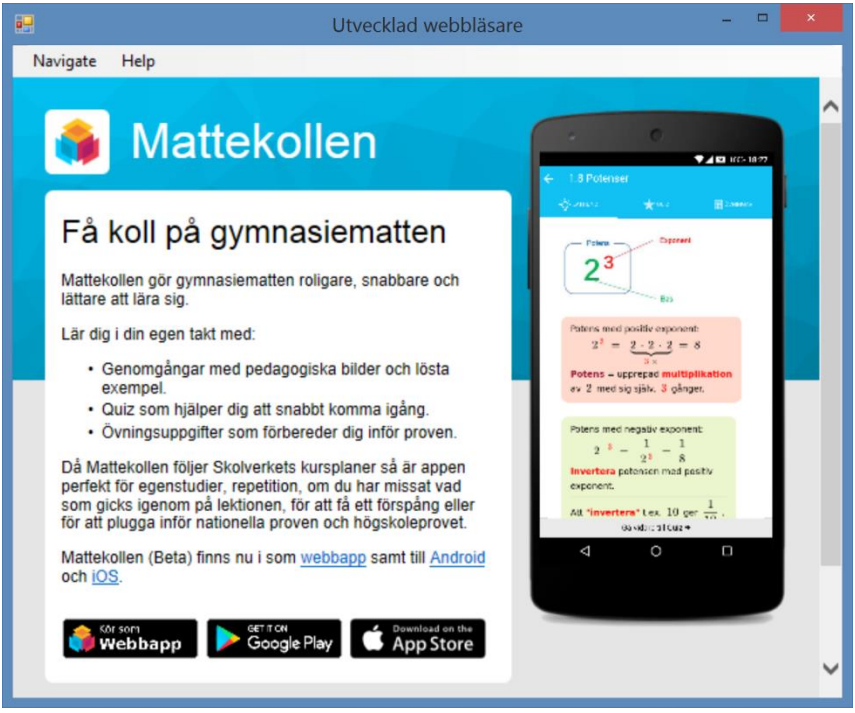
Programmering 2 med C#



Ur innehållet:

Windowsprogrammering
Grafiskt gränssnitt mot Internet (webbläsare)
Grafiskt gränssnitt med menyval
Multiple Document Interface
Objektorienterad programmering
Objektorient. modellering & implementation
Metoder i OOP / Generics
LINQ / Lambdauttryck
Delegater / Metodgrupper
Arv och polymorfism
Abstrakta klasser & metoder
Virtuella metoder
Filhantering / Slumplösenord
Kryptering av filer / Tabellhantering i filer
Databaser / Relationsdatabasmodellen
Introduktion till SQL databaser
Visual Studios SQL-Server
Grafiskt gränssnitt mot databasen
En SQL-klient i C#
Att skapa och designa en databas
Databas med egna funktionaliteter
Projektuppgifter & övningar
Fullständiga lösningar till alla övningar

Utveckla en egen webbläsare (ex. ur boken ovan):



Programmering i matematik

Tio lektioner

Ett läromedel som integrerar programmering i matematikundervisningen.

Kan användas för självständigt arbete i klassrummet eller på distans.

Kräver inga förkunskaper i programmering.

För gymnasiets kurser i Matematik 1 (a, b, c) och för högstadiets åk 7-9.

Ur innehållet

Varför är såpbubblor runda?

Eftersom de följer naturens lag och antar den minst möjliga ytan vid samma volym. Detta kan uppnås endast som klot (sfär), en geometrisk figur som saknar hörn och är dessutom vacker.

Naturen minimerar energin. Effektiviteten möter estetiken.

Genom att kombinera programmering med matematik kan du lyfta hemligheterna bakom samma naturlag som gör såpbubblorna runda.

Koda direkt i vår mobila pythonmiljö. Ladda ned appen *Mattekollen*.