Webbutveckling 2

med JavaScript

Med övningar och
projektupppgifter

Förlag: Lieta AB

Titel: Webbutveckling 2 med JavaScript

Författare: Taifun Alishenas
info@taifun.se

Juni 2024

Innehåll

Kapitel 1

Introduktion till Programmering

1.1 Om programmering

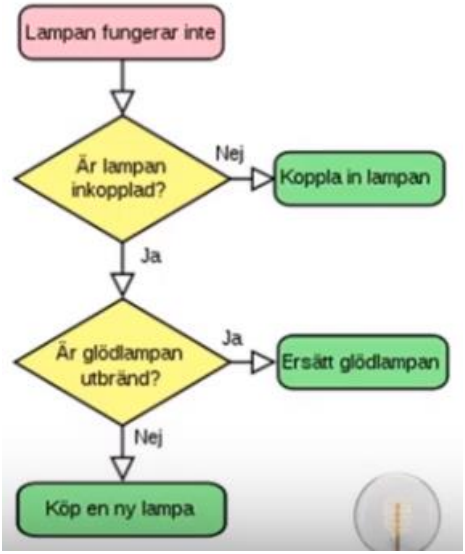
Världen vi lever i är full med prylar som är programmerade. De kallas för "intelligenta". Man pratar om *artificiell intelligens*. Men prylarna kan inte tänka själva. Någon har programmerat dem, närmare bestämt de elektroniska komponenterna i dem – små datorer. Det är de som styr all funktionalitet.

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 miljöer

> 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*, dvs koda och testa koden – precis som bilkörning. Och för att programmera behöver man en miljö, där man kan skriva och testa kod. Denna miljö är själv en programvara som måste laddas ned, om den inte redan finns i datorn och installeras. Det finns en uppsjö av programmeringsmiljöer för de olika programmeringsspråken. Ofta kallas de för IDE, *Integrated Development Environment*. En enklare variant – ofta en del av en IDE – är en editor.

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 minst en editor förinstallerad. För att skriva kod och spara den i en fil behövs en editor. Men kod får innehålla endast sådana tecken som kan 'förstås' av programmeringsspråket. Därför måste editorn spara filen som *oformaterad textfil*, dvs utan styr- och kontrollkoder som i vissa ordbehandlingsprogram används för formatering (typsnitt, stil, sorlek osv.). Ett exempel på sådana program är Word som formaterar texten och sparar sina filer som dokument av typ **`*.docx`**. Formatering innebär att det läggs till osynliga tecken i texten som programmeringsspråket inte känner till. Motsvarigheten på Mac-datorer är Pages. Sådana ordbehandlingsprogram är inte lämpliga för att skriva kod. Däremot kan t.ex. Notepad (Anteckningar), Notepad++ eller TextPad på Windwos-datorer och Textredigerare eller TextEdit på Mac-Datorer vara lämpliga texteditorer för programmering, eftersom de sparar alla filer som rena textfiler utan formatering. För textfiler kan filändelser av typ **`*.txt`**, men även andra väljas, beroende på operativsystemet resp. programvaran som filen ska användas i.

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 IDE som har utvecklingsverktyg för ett antal språk. Men JavaScript behöver ingen IDE. Det räcker med 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.

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. Med scriptspråk menar man sådana språk som endast kan köras på webben. En annan kategori är *universella* språk, som t.ex. C, C++, C#, Java, Python, ... som kan användas för att programmera vilken applikation som helst.

Hos scriptspråken nöjer man sig med de enklare elementen i programmering, för att förse webbsidor med vissa funktionaliteter. Därför kallas koden även för *script*. Scripten bakas in i HTML-kod, varför de enast kan exekveras på webben.

JavaScripts exekveringsmiljö är webbläsaren (web browser).

Utvecklingsmiljön däremot – dvs där man skriver koden – kan vara vilken editor som helst.

JavaScript är ett *interpreterande* språk, dvs koden tolkas till maskinkod (datorns språk) av en interpretator som är inbyggd i webbläsaren. Maskinkoden utförs direkt av datorns processor utan att den mellanlagras. De mest använda webbläsarna är Google Chrome på Windwos-datorer och Safari på Mac-datorer. I båda är en interpretator för JavaScript inbyggd.

JavaScript får inte förväxlas med Java. Det handlar om två olika programmeringsspråk som dessutom tillhör två olika kategorier av programmeringsspråk: Medan JavaScript är ett scriptspråk är Java ett universellt programmeringsspråk.

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 reserverade ord än i tabellen ovan. Man ser att de skrivs alla med små bokstäver. Generellt gäller följande regel 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 möjligheten att bädda in andra scriptspråk i sin kod som t.ex. JavaScript. Därför är webbläsaren den naturliga exekveringsmiljön för JavaScript. För att *skriva* JavaScript kod behöver man en *editor*, och för att exekvera behöver man en *webbläsare*. 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*.

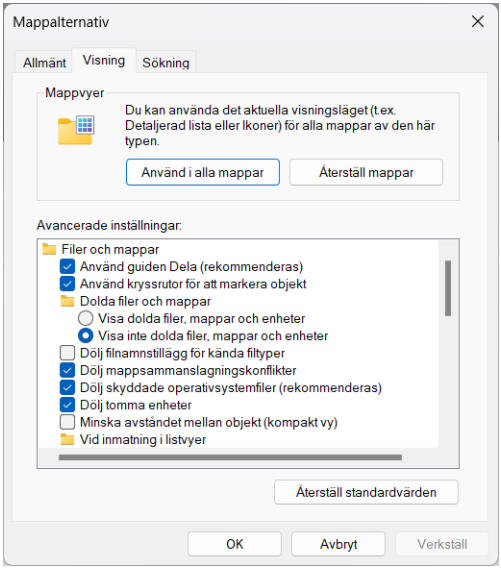
Att hantera filändelser

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:

Kommentarer

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 15) 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.

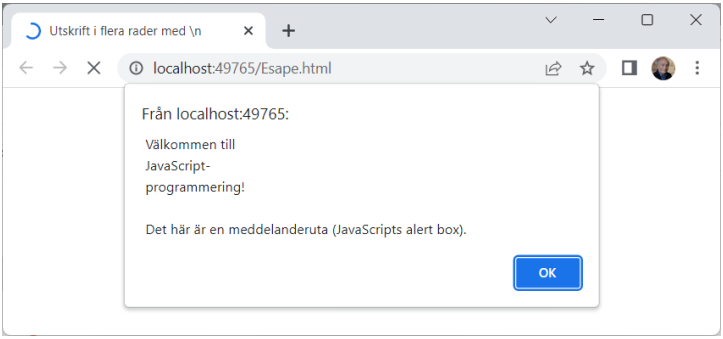
Konkateneringsoperatorn **`+`** används på rad **6**, precis som i programmet **`Concat`** (sid 18), för att inte behöva bryta rad i koden mitt i en sträng, se regeln på förra sidan.

`<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.

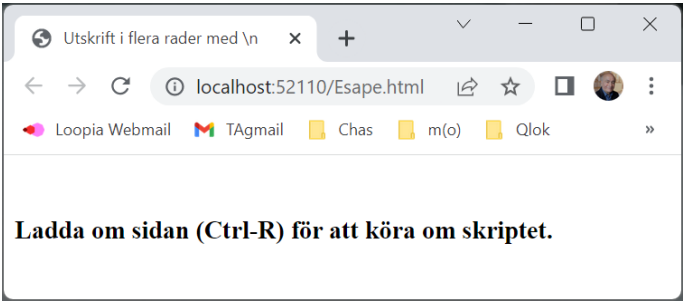
Radbrytning i utskriften med JavaScript

```
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 11)? 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 13 för att synliggöra filändelserna.

1.23 Öppna din favorit editor eller Notepad++ och mata in koden till programmet **`Welcome`** (sid 15). 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 **`document.writeln()`**-satsen i programmet **`Concat`** (sid 18) med fyra olika satser. De ska ge samma utskrift som det ursrungliga programmet. Ändra desutom 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 20) `<br>` med **`\n`**. Ersätt i programmet **`Escape`** (sid 20) **`\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>'     )
```

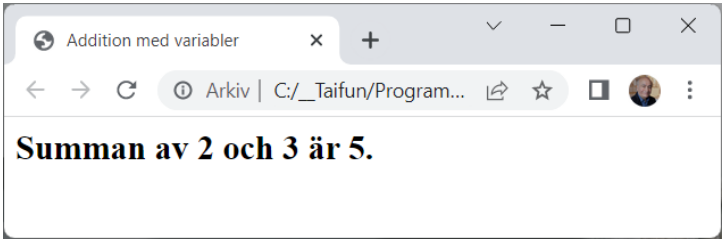
Kapitel 2

Grundbegrepp i programmering

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?

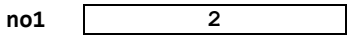
En variabel är en platshållare (minnescell) för ett värde (data).

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 26)

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

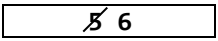
Dvs:

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 26). 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 42.

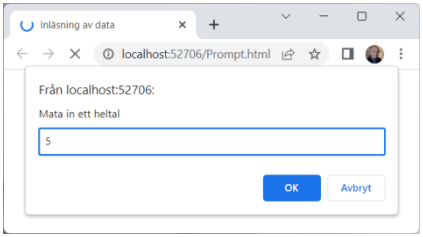
Filosofiskt handlar det om den klassiska skillnaden mellan *att vara* och *att bli*, mellan *tillstånd* och *handling*, mellan den statiska likheten och den dynamiska tilldelningen. Vid tilldelning relateras sanningen till tiden, dvs frågan är inte *om* utan *när* **x = 5**. Jo, precis när variabeln **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.

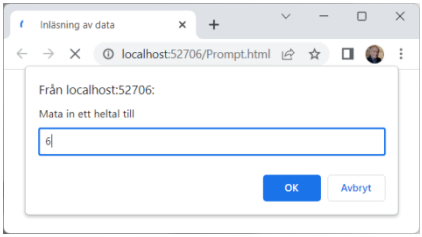
2.3 Inläsning av data

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()`**.

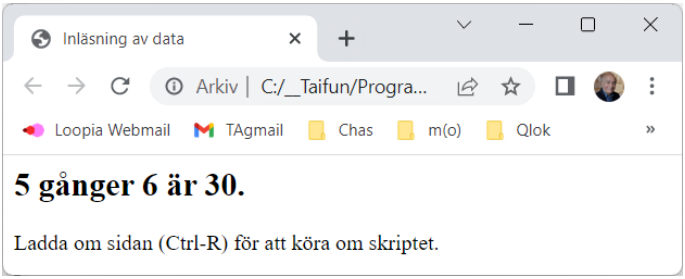
```
1  <!-- Input.html
2       Läser in två tal och skriver ut dem samt deras produkt
3       Funktionen prompt() skriver ut en ledtext och läser in -->
4  <title>Inläsning av data</title>
5
6  <script>
7     no1 = prompt('Mata in ett heltal')        // Inläsning
8     no2 = prompt('Mata in ett heltal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

I programmet **`Input`** anropas funktionen **`prompt()`** två gånger (rad **7** & **8**). 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 26). Variablerna **no1** och **no2**:s värden är inte längre hårdkodade utan läses in med godtycklig data. Variablernas initiering sker genom inläsning.

Funktionen `prompt()`

Det som åstadkommer inläsningen är anropet av funktionetnt **prompt()** på rad **8** i satsen:

```
no1 = 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. Skapar variabeln **no1** och initierar den med heltalet från punkt 4.
5. Fortsätter programkörningen, när användaren klickat på OK.

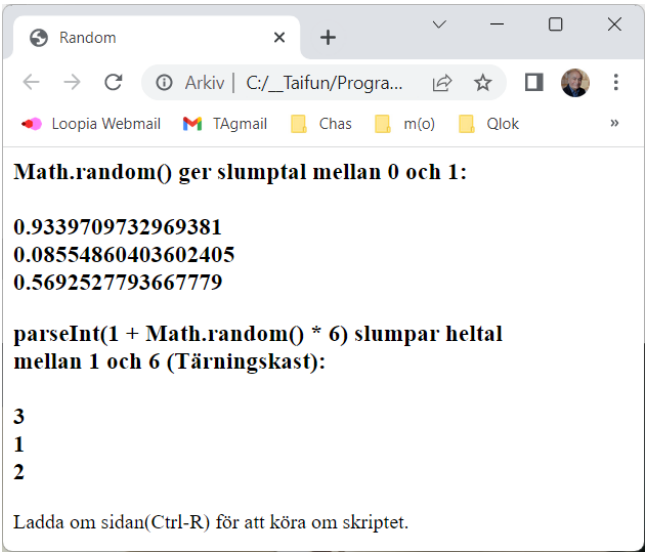
Inmatningen *returneras* av funktionen **prompt()**. Men eftersom **prompt()** är en fördefinierad funktion i JavaScript som returnerar en sträng, som tilldelas variabeln **no1**, som automatiskt omvandlas till tal när det multipliceras med **no2** på rad **9**.

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 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))
```

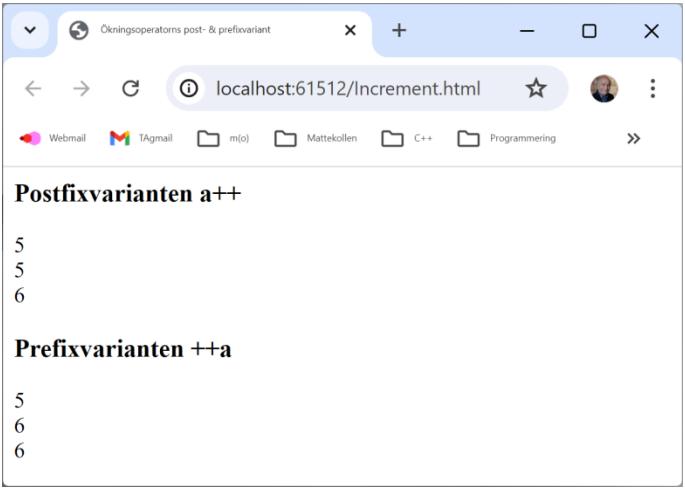
Dessa formler skulle kunna användas i program som ska slumpa heltal i intervallet `[a, b]`.

2.5 Ökningsoperatorn ++

Denna operator kommer från C++, har gett namnet till språket. Den har i JavaScript samma betydelse. Det finns två varianter av ökningsoperatorn, eng. *increment operator*: Man kan skriva den *efter* operanden, så här **a++**, eller *före* operanden, så här **++a**. **a** är operanden, **++** operatorn. Sätts den *efter* operanden talar man om ökningsoperatorns *postfix*variant. Skrivs den *före* operanden blir det *prefix*varianten. Följande program demonstrerar skillnaden mellan båda dessa varianter:

```
<!-- Increment.html -->
<title>Ökningsoperatorns post- & prefixvariant</title>
<script>
   var a = 5;
   document.writeln('<h3>Postfixvarianten a++</h3>');
   document.writeln(a);              // Skriver ut 5
   document.writeln('<br>' + a++);   // Skriver ut först 5, ökar sedan
   document.writeln('<br>' + a);     // Skriver ut 6

   a = 5;
   document.writeln('<h3>Prefixvarianten ++a</h3>');
   document.writeln(a);              // Skriver ut 5
   document.writeln('<br>' + ++a);   // Ökar först, skriver ut 6 sedan
   document.writeln('<br>' + a);     // Skriver ut 6
</script>
```

Ett körresultat ser ut så här:



Postfixvarianten **a++** betyder:

”Utför satsen med det aktuella värdet på variabeln **a** och öka den därefter med **1**”.

Närmare bestämt ökar **a**:s värde *efter* satsen dvs efter *semikolonet*. Satsen **a++** är en kompakt kod för ökning med **1** genom överskrivning, dvs:

a++ gör samma sak som **a = a + 1**

Observera att **a++** *inte* gör samma sak som **a + 1**. I **a++** ingår dessutom en tilldelning, medan **a + 1** endast innehåller en addition. Ökningsoperatorn består alltså av *två* operationer, addition *och* tilldelning. Därför kallas den också för en *dubbeloperator*.

P.g.a. att tilldelning implicit ingår i ökningsoperatorn *överskriver* **a++** variabeln **a**:s värde. Det gör inte utrycket **a + 1**.

Prefixvarianten **++a** betyder:

”Öka först variabeln **a**:s värde med **1** och utför därefter satsen med det nya ökade värdet på **a**”.

Även **++a** gör samma sak som **a = a + 1**. Skillnaden med postfixvarianten blir påtaglig först när det finns något som händer innan och/eller efteråt. Dvs det är snarare *sammanhanget* i vilket operatorn används, som gör skillnaden. I programmet **Increment** på förra sidan är detta sammanhang utskriftssatsen med funktionen **document.writeln()**.

Övningar till Kapitel 2

2.1 Komplettera programmet **Variable** (sid 26) 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 29) 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 31).

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.

Kapitel 3

Kontrollstrukturer

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.

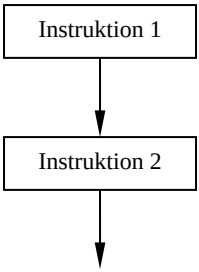
Kontrollstrukturer består av tre grundläggande typer:

- **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.

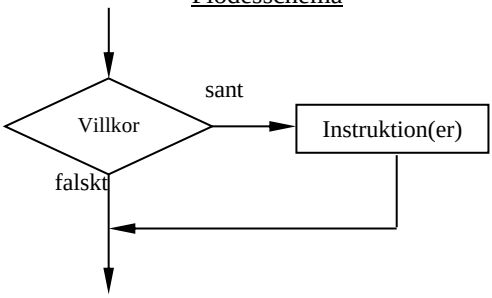
3.2 Enkel selektion: `if`-satsen

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



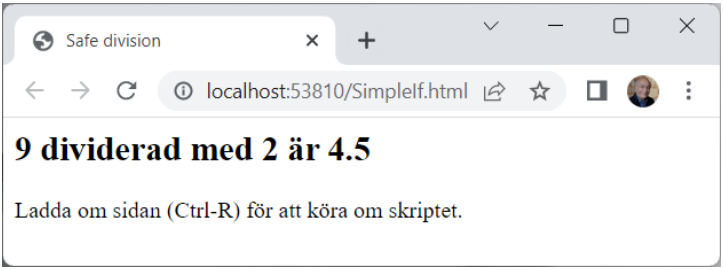
I JavaScript kallas den enkla selektionen för **if**-sats och kodas generellt på följande sätt:

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

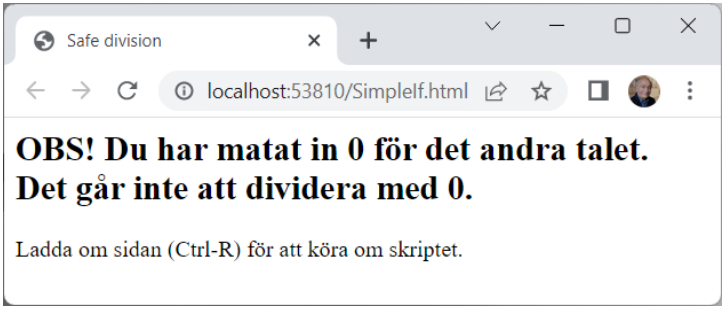
Första raden är **if**-satsens *huvud*. Resten är **if**-satsens *kropp* som omsluts av klammerparenteserna **{** och **}** som vi i fortsättningen kommer att kalla kort *klamrar*, ibland *måsvingar*. Om kroppen består endast av en sats kan klamrarna utelämnas vilket vi utnyttjar i följande program:

```
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

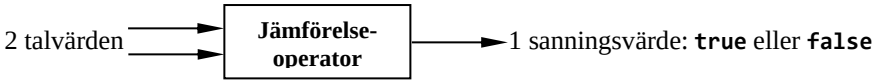
`if`-satsens huvudingrediens är alltid ett *villkor*, t.ex. `no2 != 0`. Dubbeltecknet `!=` betyder *icke lika med* och måste skrivas *utan* mellanslag: Är `no2` skilt ifrån `0`, ja eller nej? Man kan alltså uppfatta ett villkor som en *fråga* som endast kan besvaras med ja eller nej. En annan aspekt 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 30 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 40).

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`**

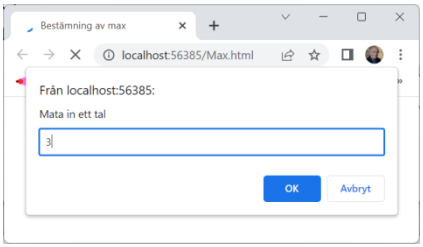
Precis som **!=** är även dubbeltecknet == (utan mellanslag) en jämförelseoperator, men står för *lika med*. Observera skillnaden mellan likhet som kodas med *två* likhetstecken == och tilldelning vars kod är *ett* likhetstecken =. Även den andra **if**-satsens kropp är en utskriftssats som skriver ut 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.

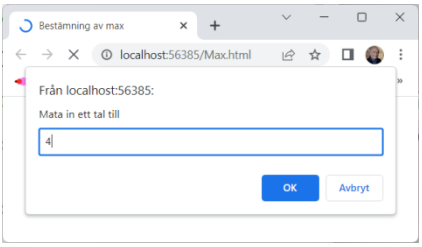
Bestämning av max/min

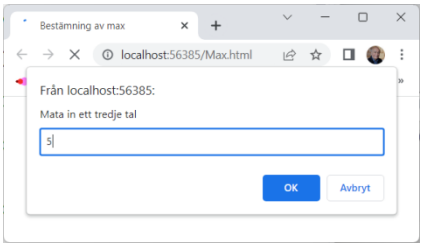
I programmet **SimpleIf** (sid 40) användes två **if**-satser, för att avgöra om ett tal var jämnt eller udda. Nu ska vi skriva ett nyttigt program som bestämmer det största (minsta) värdet bland 3 inmatade tal. Nyttigt, därför att vi kommer att ha användning av det bl.a. i projektet Gymnastiktävlingen. Sedan ska vi använda detta exempel för att precisera vår kunskap om *modularisering* som nämndes inledningsvis i boken (sid 9), och lära oss att själva definiera *funktioner* i JavaSript.

```
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.

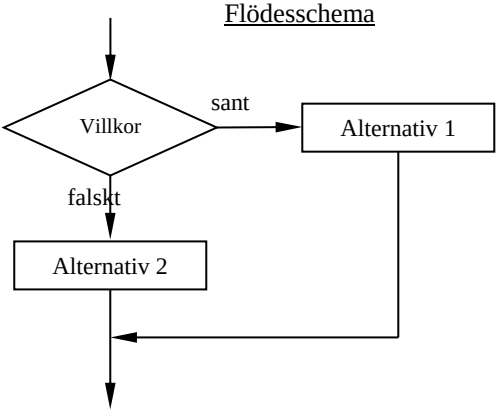
3.3 Tvåvägsval: if-else-satsen

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
}
```

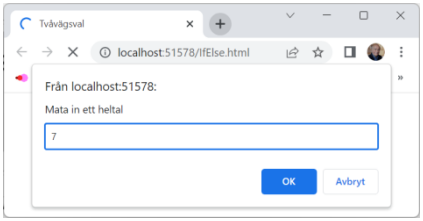
Om **`if`**- eller **`else`**-blocket består endast av en sats kan klamrarna **`{`** och **`}`** utelämnas. Anta att båda block består bara av en sats, då förenklas formen:

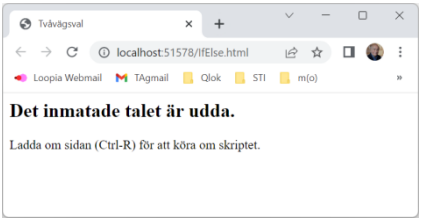
```
if (villkor)
    sats1
else
    sats2
```

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

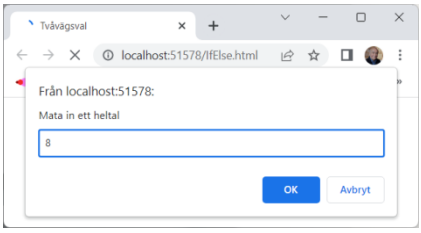
```
1  <!-- IfElse.html
2       Läser in ett heltal och avgör om det är jämnt eller udda
3       Tvåvägsval: if-else-satsen med EN sats i if-else-delen -->
4
5  <title>Tvåvägsval</title>
6
7  <script>
8     no = parseInt(prompt('Mata in ett heltal'))   // Inläsning
9
10    if (no % 2 == 0)
11       document.writeln('<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.

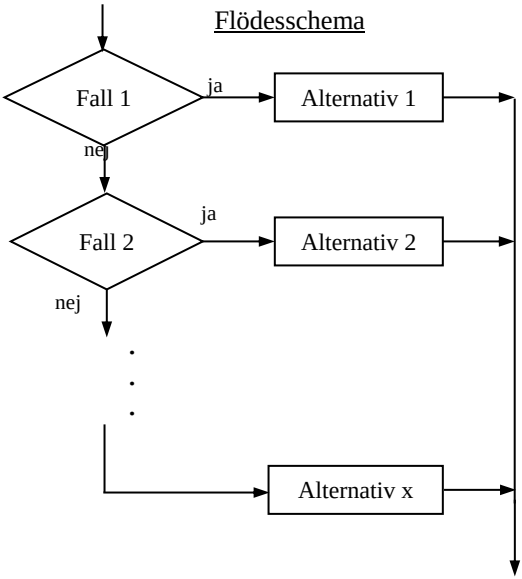
4. I matematiken används modulo bl.a. för att bestämma den största gemensamma delaren av två heltal (Euklides algoritm).

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



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**

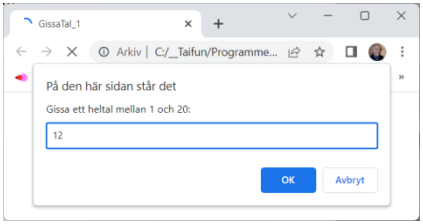
`if-else`-*stegen*

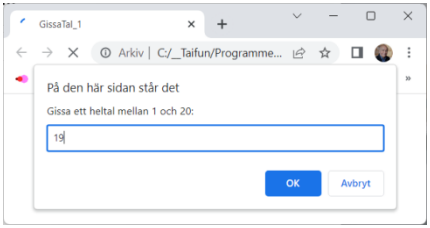
Låt oss titta på följande exempel av ett trevägsval som använder den s.k. **`if-else`**-stegen. Användaren ska gissa fram programmets hemliga tal **`17`**. Man gissar inom intervallet **`[1, 20]`**, får sedan hjälp om det gissade talet var mindre än, större än eller lika med det hemliga talet. Just nu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

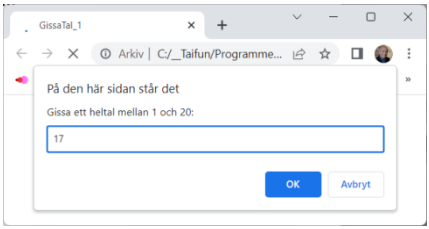
Här kommer *Gissa tal-spelet* i en första version, som innehåller ett val mellan tre alternativ, *fall 1:* det gissade talet är lika med, *fall 2:* mindre än, *fall 3:* större än programmets hemliga tal **17**:

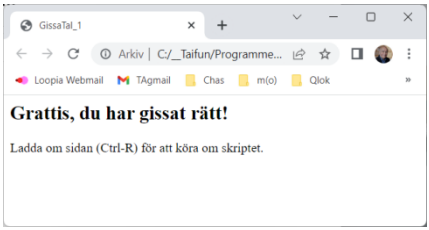
```
<!-- GissaTal.html
     Låter användaren gissa programmets hemliga tal secret
     Trevägsval med en if-else stege -->
<title>GissaTal</title>
<meta charset="UTF-8">          <!-- För de svenska tecknen -->
<script>
   secret = 17                  // Programmets hemliga tal
                                // 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 48 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.

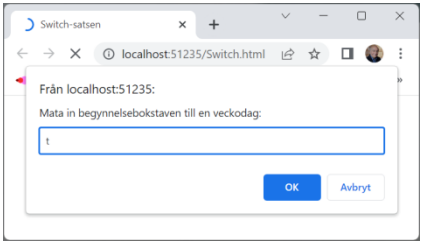
Följande programexempel demonstrerar **switch**-satsen: Vi läser in begynnelsebokstaven till en veckodag och det fullständiga veckodagsnamnet skrivs ut. I **switch**-satsen väljs ett alternativ av sex. Tisdag och torsdag behandlas i ett fall.

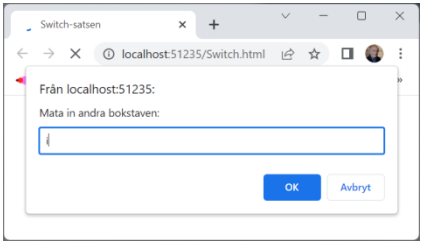
```
<!-- Switch.html
     Demonstrerar flervägsval med switch-satsen
     Kompletterar veckodagen efter inmatning av första bokstaven
     För t(isdag/torsdag) krävs den 2:a bokstaven -->
<title>Switch-satsen</title>
<script>
   letter1 = prompt('Mata in begynnelsebokstaven ' +
                    'till 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-

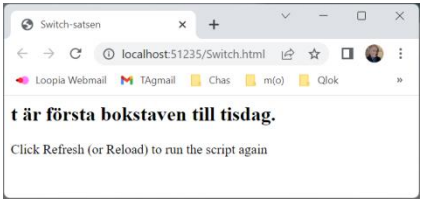
nar man i den utan att likhet mellan uttrycket och konstanten som finns i de efterföljande **case**-satserna testas. Om **switch**-satsen ska välja endast ett enskilt värde bland flera, borde varje **case** avslutas med **break**.

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**

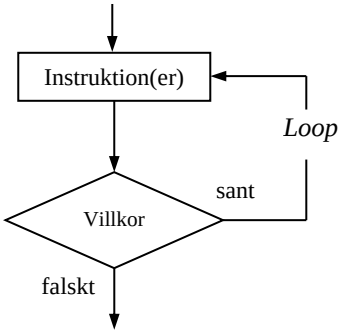
Efter-testad repetition

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:

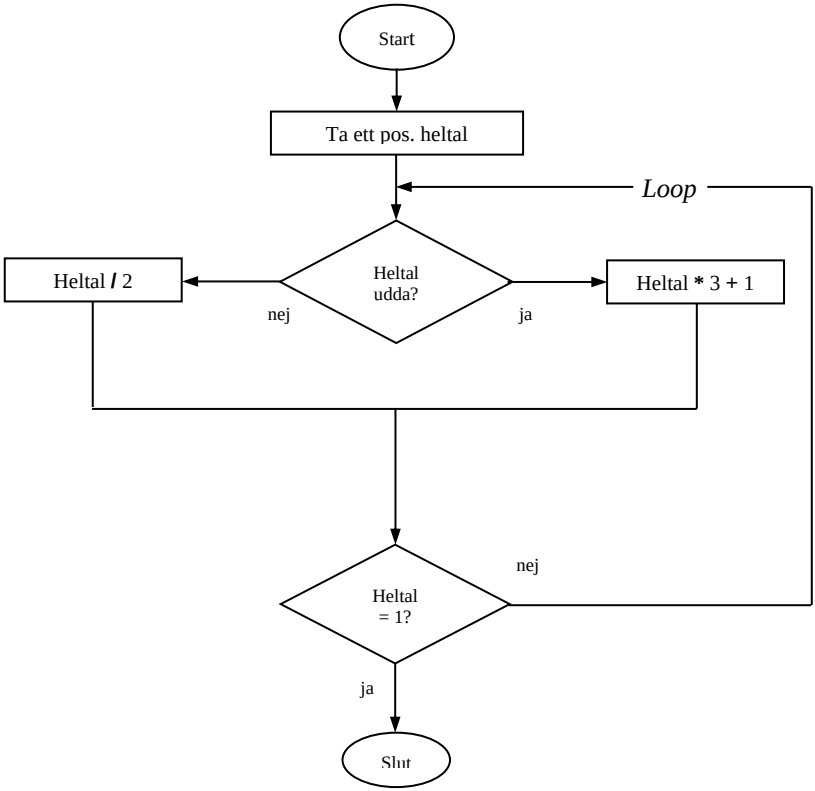
Collatz algoritmen

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 *

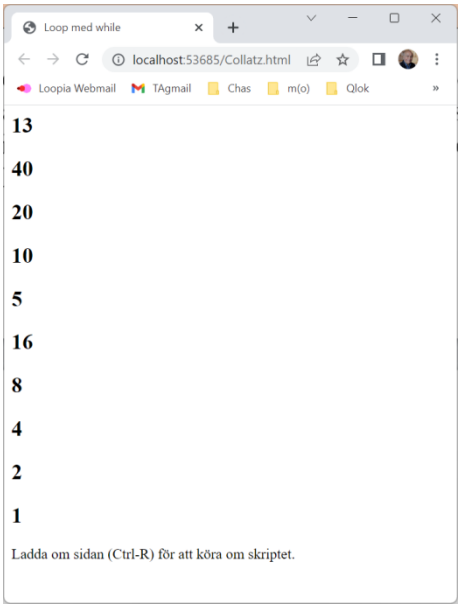
```
Läs in ett positivt heltal
REPETERA
   OM talet är udda
      multiplicera med 3, addera 1
   ANNARS
      dividera talet med 2
   Skriv ut talet
SÅ LÄNGE talet ≠ 1
```

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**:



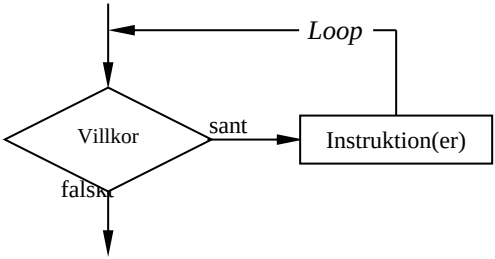
do-satsens arbetssätt, dvs *repetitionen* skiljer sig grundläggande från kontrollstrukturen *selektion* (val) som vi lärde känna tidigare. Medan en selektions alltid går *framåt*, efter den har avgjort valet p.g.a. det styrande villkoret, återvänder en repetition alltid till kontrollstrukturens början, 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 54 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)



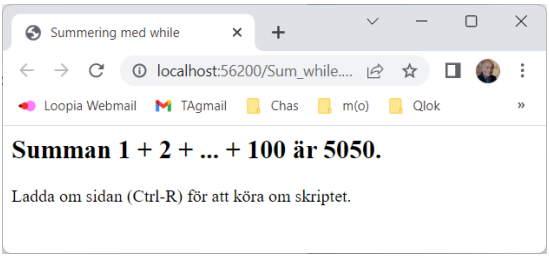
I JavaScript inleds den för-testade repetitionen med det reserverade ordet **`while`** och skrivs generellt på följande sätt:

```
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 74).

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

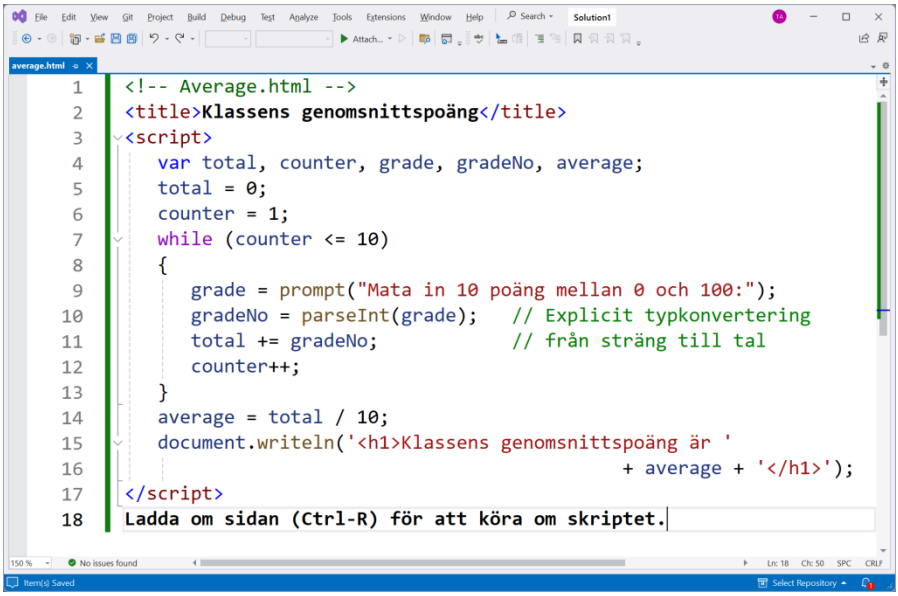
I programmet **Sum_while** är **while**-satsens avsutningsvillkor **term <= 100**. Om detta villkor vore sant från början och förblev sant hela tiden, skulle satserna på raderna **10-11** att utföras i all evighet, vilket kallas för *evighetsloop*.

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 Räknar-styrd repetition

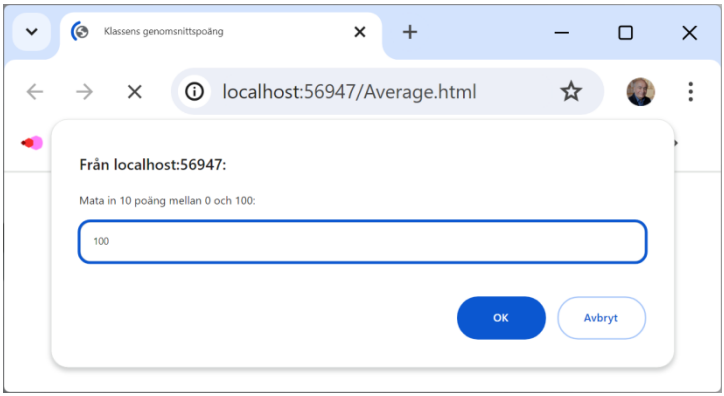
Som exempel för repetitioner väljer vi **while**-satsen. Programmet **Average** kombinerar summeringen av termer (programmet **Sum_while**, sid 57) med inläsning av data. Man beräknar genomsnittet (medelvärdet) av elevernas poäng i en klass:



Samtidigt introduceras dubbeloperatorerna **+=** och **++** i JavaScript som vi känner till från C++. De gör förstås samma sak här som där: **+=** adderar först sina operander och tilldelar sedan resultatet till operanden till vänster. Dubbeloperatorn **++** ökar sin operand med 1. Det som man måste se upp med här, är att de involverade variablerna **total** och **counter** måste initieras innan de uppdateras på raderna **11** och **12**. Initieringen som sker på raderna **5** och **6**, är nödvändig, eftersom deras gamla värden används innan de uppdateras.

Ett körexempel efter 10 inmatningar 100, 88, 93, 55, 68, 77, 83, 95, 73 och 62 ger:

Rutan ovan är den sista efter 10 inmatningsdialoger som genereras av JavaScript-funktionen **`prompt()`** på rad **`9`**:



Vi vet från tidigare att **`prompt()`** returnerar en sträng som vi på rad **`10`** omvandlar till tal med en annan JavaScript-funktion, nämligen **`parseInt()`**. Tekniken kallas för *explicit typkonvertering*, dvs självgjord omvandling av datatypen, till skillnad från *automatisk typkonvertering* som JavaScript utför p.g.a. vissa regler som är inbyggda i språket. Här är den explicita varianten nödvändig.

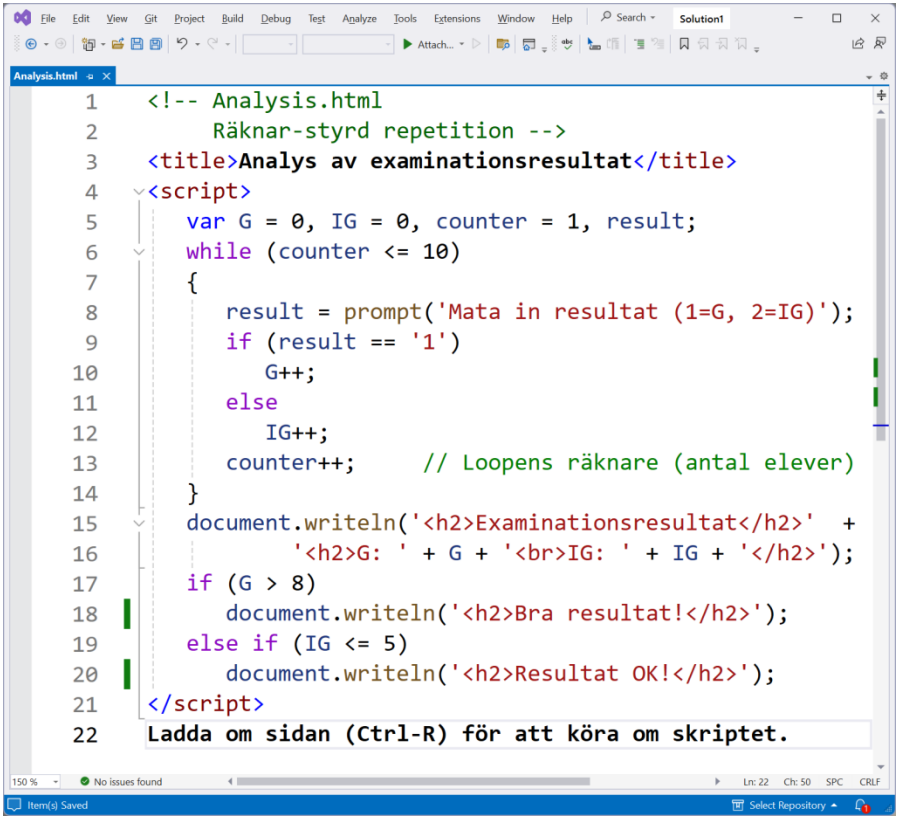
En annan nyhet i programmet **`Average`** på förra sidan är att vi för första gången deklarerar våra variabler med **`var`** på rad **`4`**, vilket inte är obligatoriskt i JavaScript. Vi kommer att fortsätta med det, när våra program blir längre och behöver struktureras mer. Strukturering är redan avgörande i detta programexempel, speciellt vad gäller frågan, vilka delar av koden måste stå *före*, vilka *i* och vilka *efter* loopen:

Observera att både inläsningen och summeringen av data, inkl. typkonverteringen görs *i* **`while`**-loopen, mellan allt annat står utanför den, speciellt beräkningen av genomsnittspoängen (medelvärdet) som står *efter* loopen på rad **`14`**.

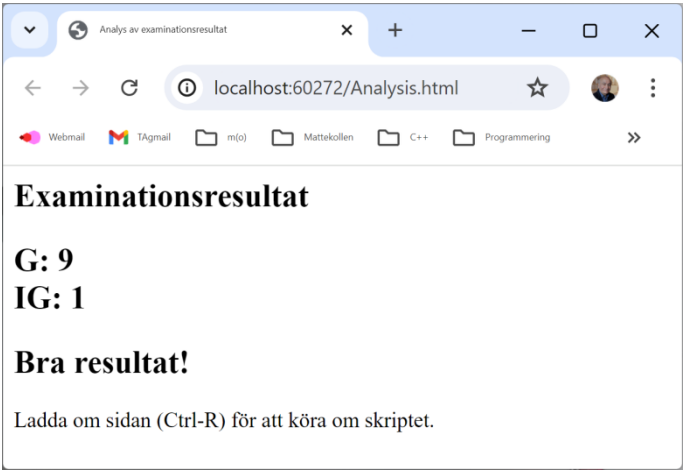
En speciell roll spelar variabeln **`counter`** som tar reda på antal varv. Den initieras *före* och uppdateras *i* loopen, för att användas i avslutningsvillkoret och förhindra evighetsloop. Det är **`counter`**n och antalet 10 som vi har fastskrivit i koden som styr **`while`**-loopen och därmed antalet inmatningar. Man talar om räknar-styrd repetition.

Analys av examinationsresultat

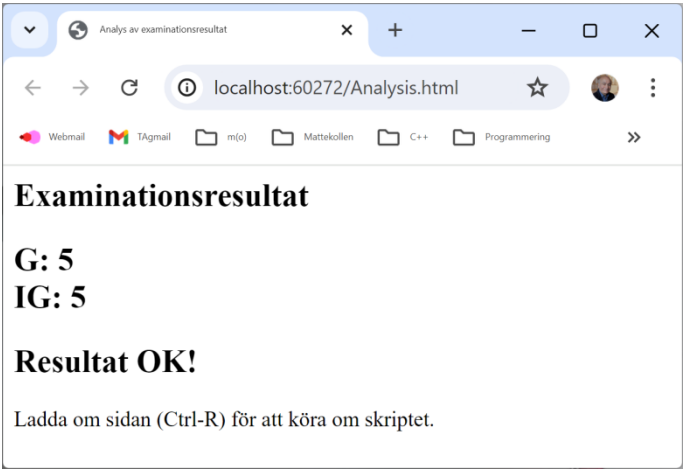
Ett annat exempel på räknar-styrd repetition är programexemplet **`Analysis`** på nästa sida. Här försöker man att få en sammanfattning eller analys av 10 elevers resultat i ett prov. Beroende på analysen som baseras på en policy skrivs ut vissa slutsatser.



En körning med 10 inmatningar 1, 2, 1, 1, 1, 1, 1, 1, 1 och 1 ger:



Medan en annan körning med inmatningarna 1, 2, 1, 2, 2, 1, 2, 2, 1 och 1 resulterar i:

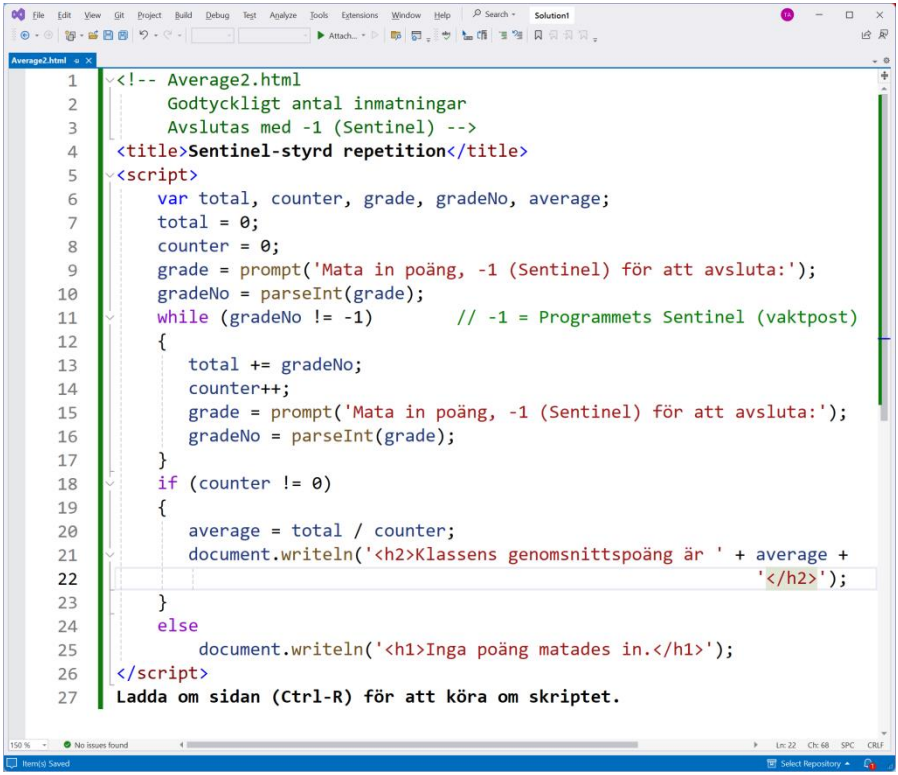


En övning i räknar-styrd repetition vore att byta ut loopens räknare **`counter`** i programmet **`Analysis`** med summan **`G + IG`** av alla poäng, för att spara en variabel och förkorta koden. Ev. blir det nödvändigt att även justera **`while`**-loopens avslutningsvillkor. Denna uppgift är överlåten till övn 3.15 (sid 74).

3.8 Sentinel-styrd repetition

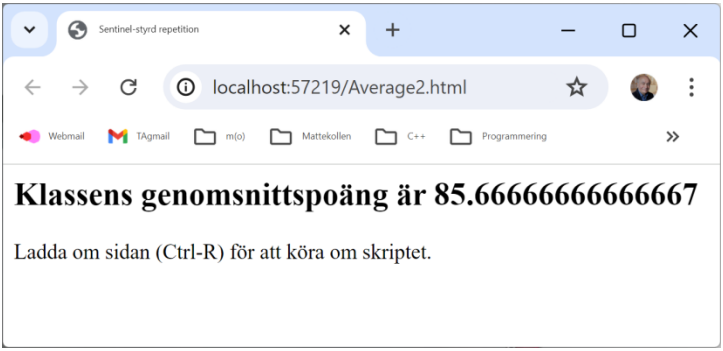
Programmet **Average** (sid 59) har en stor nackdel: Antalet inmatningar är statiskt. **while**-loopens antal varv är hårdkodat. Bara klasser med 10 elever kan använda programmet. Självklart är det önskvärt att användaren ska kunna bestämma antalet inmatningar, inte vi som kodar. Programmet borde vara användbart för alla möjliga storlekar på klasser.

Nu vill vi generalisera programmet **Average** genom att låta antalet inmatningar vara fritt väljbart. Programmet **Average2** som följer, är ett exempel på en lösning av detta problem. Det som skiljer dessa två program är i huvudsak loopens avslutningsvillkor, dvs sättet att styra repetitionen. **Average**:s **while**-loop styrs av en räknare – variabeln **counter** – som successivt uppdateras. Repetitionen är räknarstyrd. **Average2**:s repetition däremot styrs av ett speciellt *värde* som kallas för *Sentinel* (vaktpost) som är nyckeln till att avsluta programmet. Man pratar om *Sentinel-styrd repetition*. Detta värde som bestäms i koden, får inte finnas bland programmets ”legitima” data. Vad vi menar med det kommer vi snart att se.



Som man ser har vi valt värdet *-1* som *Sentinel*. *-1* kan inte vara en elevpoäng.

Matar vi t.ex. in: 97, 88, 72 och slutligen *-1* får vi följande utskrift:



Inmatningen av *-1* avslutar körningen därför att programmets Sentinel har valts till *-1* på rad **11**. Decimaltalet är resultatet av divisionen på rad **20**.

Matar vi in däremot *-1* från början blir det följande utskrift:

Anledningen är att vi p.g.a. **`gradeNo`** = **`-1`** aldrig kommer in i **`while`**-loopen och p.g.a. **`counter`** = **`0`** hamnar i **`else`**-delen av **`if-else`**-satsen på rad **25**.

Logiken bygger på programmets struktur: en **`while`**-loop följd av en **`if-else`**-sats. Dessutom nödvändigheten att ha inläsningen av data *två gånger* i programmet, en gång *före* (rad **9**) och en gång *i* **`while`**-loopen (rad **15**). Före loopen, för att kunna bilda loopens avslutningsvillkor. I loopen, för att kunna mata in data flera gånger. Och faktiskt gäller även det omvända: Programmets struktur är vald för att just implementera den önskade logiken.

Nödvändigheten av *två* inläsningar kan anses som en nackdel av programmet. Frågan är: kan man hitta en alternativ struktur som möjliggör endast *ett* inläsningstillfälle, t.ex. genom att byta till en annan loop-typ? Frågan är föremål för en övning, se övn 3.16 (sid 74).

3.9 HTML-element i loopar

För att förstå hur HTML-element fungerar i loopar vill vi börja med att titta på HTML-element utan loopar och sedan fortsätta att sätta dem i loopar. Vi tar som exempel **p**-elementet där **p** står för paragraf dvs stycke. Men vad är ett *element* i HTML? Det blir en liten repetition i HTMLs grunder från *Webbutveckling 1*:

HTMLs taggar, innehåll, element och attribut

En *tagg* i HTML inleds med **<** och avslutas med **>** . Taggar används i regel parvis: en *starttagg* markerar *början* och en *starttagg* markerar *slutet*. Ex.:

```
<p>Välkommen till HTML!</p>
```

Raden ovan har två taggar: starttaggen ** ** och sluttaggen ** **.

Det som står mellan start- och sluttagg, texten **Välkommen till HTML!**, kallas för *innehåll*, närmare bestämt innehåll till **p**-elementet.

p-elementet skapar ett nytt stycke (paragraf) i textflödet som inkluderar radbyte efteråt. Innehållet skrivs ut i paragrafen.

Generellt har ett *element* i HTML följande ingredienser:

Starttagg + innehåll + sluttagg = **Element**

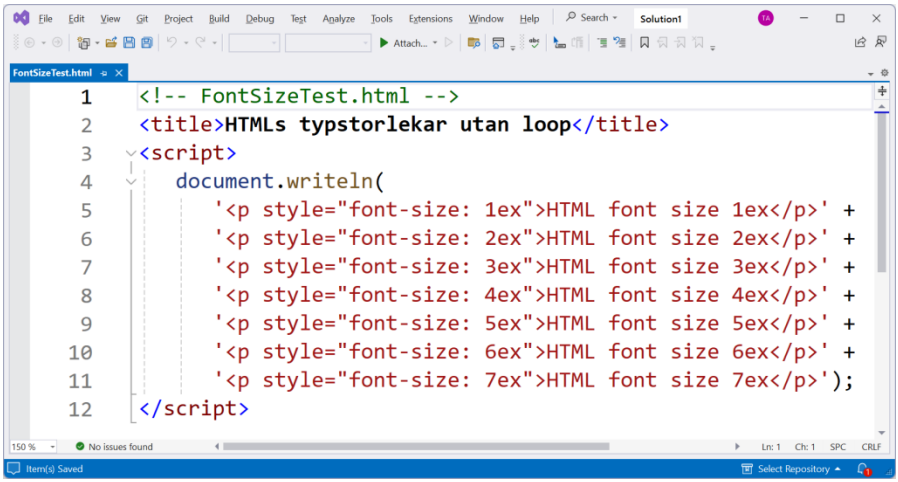
Ett annat exempel på ett **p**-element som är lite mer invecklat är:

```
<p style="font-size: 4ex">HTML font size 4ex</p>
```

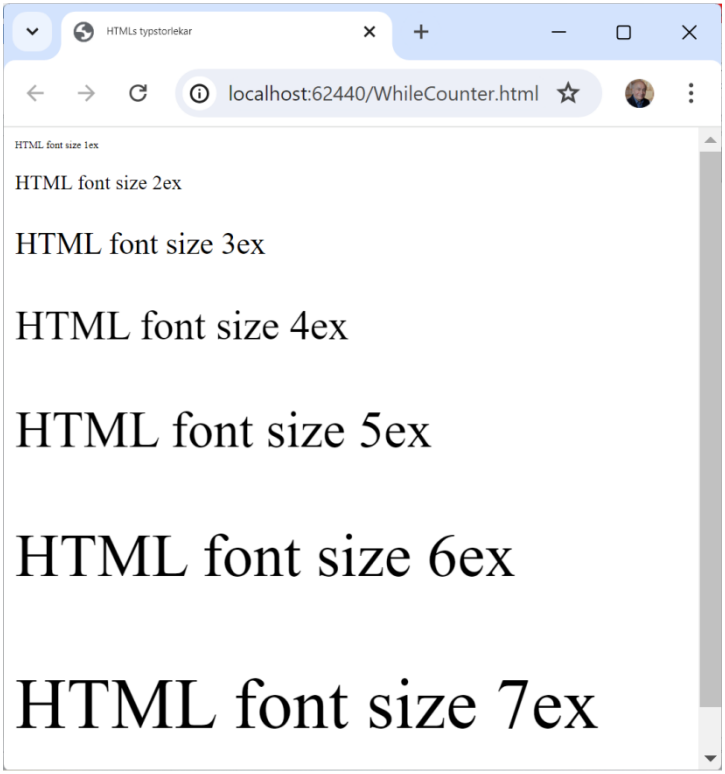
Innehållet i det här **p**-elementet som skrivs ut, är texten **HTML font size 4ex**, eftersom det är den som står mellan start- och sluttagg. Men starttaggen har fått ett nytt utseende: den har utvidgats och fått ett s.k. *attribut*. Attributets *namn* är **style** och attributets *värde* är **"font-size: 4ex"**. Värdet måste anges som sträng dvs inom citationstecken. I exemplet bestämmer **style**-attributets värde att texten som skrivs ut, ska ha HTML-typsnittet **4ex**. Både attributets namn och värde är del av **p**-elementet, koden som står *inom* taggarna, medan innehållet är texten som ska skrivas ut och står *utanför* taggarna.

Vilka typsnitt som finns i HTML och hur de ser ut, ska vi snart titta på.

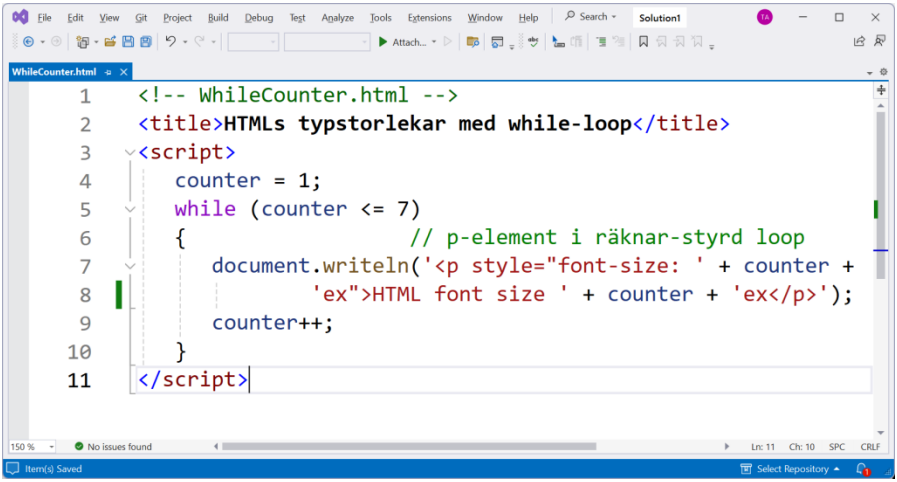
I scriptet **FontSizeTest** på nästa sidan skriver vi ut alla 7 typsnitt utan loop. Sedan ska vi gå över att skriva om koden till en **while**-loop.



Här har vi tagit över **p**-elementets syntax som förklarades på förra sidan. Koden skriker efter att skrivas om till en loop, vilket vi tar upp på nästa sida. En körning av scriptet **FontSizeTest** visar HTMLs 7 typstorlekar:



I scriptet **WhileCounter** har vi effektiviserat koden med en **while**-loop som sträcker sig över raderna **5-10**. Loopen styrs av räknaren **counter**.



Den svåraste delen av koden utgörs av raderna **7-8**, i **document.writeln**-parentesen. Problemet består av att vi måste baka in typsnittens storlekar 1-7 som representeras av räknaren **counter**, i den text som **p**-elementets innehåll ska skriva ut. För att göra det måste vi konkatenera varaiabeln **counter** med strängkonstanter.

Men det dyker upp en konflikt mellan JavaScript-funktionen **document.writeln**:s sätt att markera strängar (både med citationstecken och apostrofer) och HTML-attribute **style**:s värde som måste omgärdas med citationstecken.

Apostrof vs. citationstecken

Vi tar raderna **7-8** från scriptet **WhileCounter** (ovan) och skriver parameterlistan av **document.writeln()**, på en rad, för att analysera den:

```
'<p style="font-size: ' + counter + 'ex">HTML font size ' + counter + 'ex</p>'
```

Det är ett **p**-element, och det som är framhävt med grå bakgrund, är elementets innehåll som kommer att skrivas ut. Värdet till attributet **style** är skrivet inom citationstecken, medan andra strängar har markerats med apostrofer. Detta för att åstadkomma korrekt parning. Några strängar är nästlade i andra. Dessutom överlappar vissa strängmarkeringar med apostrofer andra strängmarkeringar med citationstecken. Överlappning förekommer även i början och slutet med **p**-elementets start- och sluttagg. Alla **+** tecken betyder konkatenering mellan variabeln **counter** och strängkonstanter.

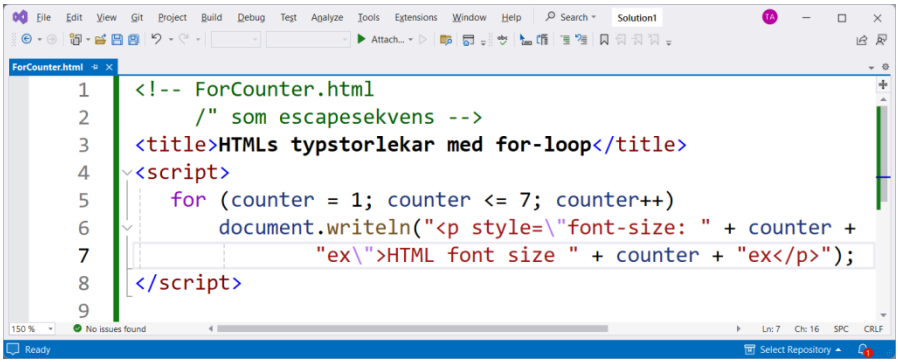
Vid den här konstruktionen har vi dragit nytta av att vi i JavaScript har möjligheten att markera strängar både med citationstecken och apostrofer. Hade det funnits endast ett tecken för strängmarkering, hade vi varit tvungna att använda *escapese-*

kvensen `\"` som alternativ. Testa gärna att ersätta alla apostrofer med citationstecken och de två citationstecknen kring attributets värde med `\"`. Vi har gjort så i nästa script. Escapesekvenser har i JavaScript samma *betydelse* som i C++. Om de *fungerar* i alla sammanhang beror ofta på samspelet mellan JavaScript och HTML.

Körresultatet av scriptet **`WhileCounter`** är identisk med **`FontSizeTest`**s resultat på förförra sidan.

3.10 Bestämd repetition: `for`-satsen

För att snabbt visa **`for`**-satsens arbetssätt vill vi börja med en ren översättning av scriptet **`WhileCounter`** (förra sid) till en **`for`**-variant:



`for`-satsen har endast *en* sats i sin kropp, raderna **`6-7`**. huvudet på rad **`5`** består efter det reserverade ordet **`for`** av: *Initieringen* **`counter = 1`** , *uppdateringen* **`counter++`** och *avslutningsvillkoret* **`counter <= 7`** som även fanns i huvudet på **`while`**-loopen i scriptet **`WhileCounter`**. Dessa tre är ingredienser i alla **`for`**-satser.

Eftersom parentesen i **`for`**-satsens huvud (rad **`5`**) nu består av de tre ingredienserna initiering, villkor och uppdatering, 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. Därför måste vi sätta **`;`**

Raderna **`6-7`** borde vara identiska med **`while`**-satsens rader rader **`7-8`** i scriptet **`WhileCounter`**. Men vi har valt – som det diskuterades på förra sidan – att endast använda citationstecknet för strängar och skilja de olika funktionaliteterna med *escapesekvensen* **`\"`**, så att den motsvarande koden i scriptet **`ForCounter`** blir:

```
"<p style=\"font-size: " + counter + "ex\">HTML font size " + counter + "ex</p>"
```

Nu är attributet **`style`**s värde omgärdat av escapesekvensen **`\"`** (som ger citationstecknet) för att para ihop attributvärdet som en sträng. De andra strängarna kodas med vanliga citationstecken. Nödvändigheten att använda escapesekvenserna motiveras av att para ihop de rätta strängmarkeringarna, eftersom det förekommer både överlappning och nästling av strängmarkeringarna. Alternativt skulle man kunna skriva allt endast med apostrofer och använda escapesekvensen **`\'`** (som ger apostrof) för **`style`**s värde. Testa gärna även detta alternativ.

Scriptet **`ForCounter`** producerar samma utskrift som på sid 66.

Summering med for

Följande script **Sum_For** som är en ren översättning av scriptet **Sum_while** (sid 57) till en **for**-variant. Båda summerar alla heltal från **1** till **100** och skriver ut summan.

```
1  <!-- Sum_for.html) -->
2  <title>Summering med for</title>
3  <script>
4     sum = 0
5     for (term = 1; term <= 100; term++)     // for-loop
6         sum = sum + term
7     document.write('<h2>Summan 1 + 2 + ... + 100 är ' +
8                                     + sum + '.</h2>')
9  </script>
10 Ladda om sidan (Ctrl-R) för att köra om skriptet.
```

for-loopen på raderna **5-6** förenar både initierigen och uppdateringen av räknaren **term** samt avslutningsvillkoret i sitt huvud, så att koden, jämfört med **Sum_while** (sid 57), blir kortare. Fördelen är effektiviteten, medan nackdelen är läsligheten.

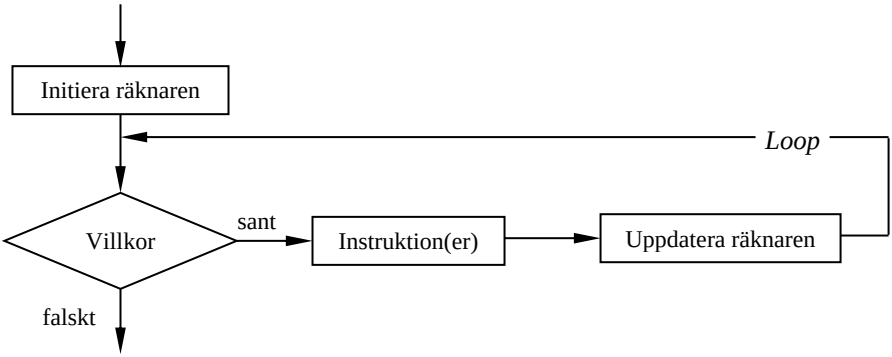
Så här ser det ut när vi kör scriptet.

Mer om **for**-satsens generella struktur samt flödesplan och pseudokod följer.

for-satsens struktur

for-satsens struktur skiljer sig 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.

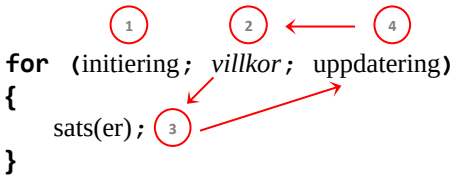
Pseudokoden

```
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:



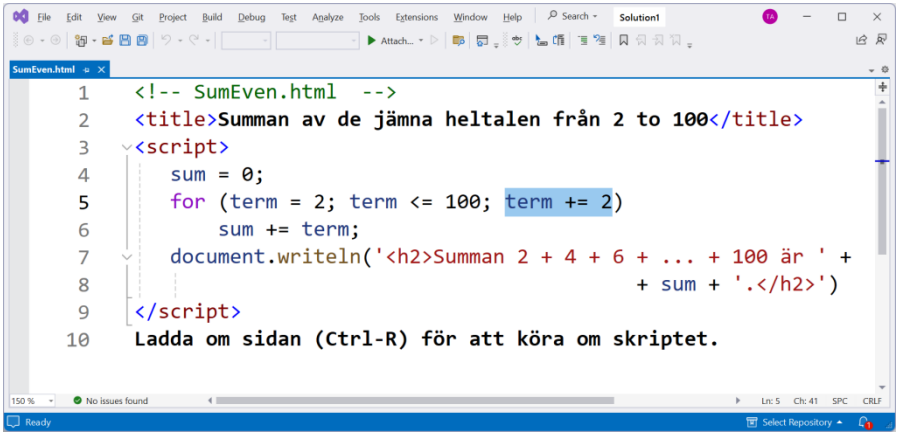
De rödmarkerade ringarna och pilarna samt numreringen ska visa i vilken *ordning* de respektive delarna utförs. Denna ordning är nämligen inte identisk med kodbitarnas ordning. Pilarna markerar loopens förlopp. Initieringen görs endast en gång och ingår ej i loopen.

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.

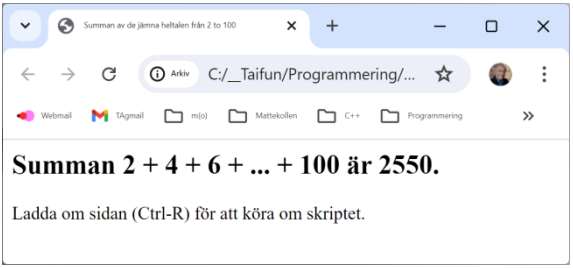
Räknaren sätts före repetitionen till ett önskat startvärde, för det mesta något heltal, ofta **1**. Detta kallas *initiering* av räknaren dvs den allra första tilldelningen av ett värde till räknaren. Sedan testas ett villkor där man brukar lägga in ett önskat *slutvärde* på räknaren. Därmed är antalet 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.

Kontroll via räknaren

for-satsens effektiva kod tillåter oss att modifiera våra script och ändra resp. vidareutveckla dem på ett enkelt sätt. Ett utmärkt verktyg för sådana modifieringar är **for**-satsens räknare. Om vi t.ex. vill ändra lite i scriptet **Sum_for** (sid 70) så att programmet summerar endast de jämna heltalen, räcker det med följande manipulation av räknaren (och lite mer) för att åstadkomma denna summering:



Vi behöver alltså bara låta loopens räknare **term** ta *två* steg i taget, för att involvera endast de jämna talen i summeringen, vilket vi gör med den framhävda koden på rad **5**. Självklart måste vi även starta räknaren med **2**, dvs ändra initieringen på samma rad. Satsen



term += 2 gör samma sak som **term = term + 2**

dvs adderar först, tilldelar sedan. Samma sak är med **sum += term** på rad **6** som gör samma sak som **sum = sum + term**. Exemplet visar programkontroll via räknaren.

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 43) 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.

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 40) 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 47).

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 55) 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 55) så att körningen genererar en evighetsloop.

3.12 Modifiera programmet **Sum_while** (sid 57) genom att ersätta **while**-loopen med en **do**-loop.

3.13 Generalisera programmet **Sum_while** (sid 57) 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 57) så att körningen genererar en evighetsloop.

3.15 Modifiera programmet **Analysis** (sid 61) genom att ersätta **while**-loopens räknare **counter** (antalet elever) med summan **G + IG**, dvs totala antalet poäng. Justera loopens avslutningsvillkor efter detta byte. På så sätt kan vi minimera antalet variabler i programmet.

3.16 Programmet **Average2** (sid 63) har inläsningen av data *två gånger* i programmet, en gång *före* (rad **9**) och en gång *i* **while**-loopen (rad **15**). Modifiera programmet genom att hitta en alternativ struktur som möjliggör endast *ett* inläsningstillfälle i programmet. **Tips:** Byt ut loop-typen.

3.17 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?

3.18 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.

3.19 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.

3.20 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.

3.21 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.

3.22 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.

3.23 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.

3.24 Vänd på problemet från övn 3.22: 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

3.25 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).

[1] 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.

Tre projektuppgifter

1. Bergvärme – simulering av borrutrustning

En borrutrustning för bergvärme kan borra 25 m under den 1:a timmen i en viss tomtmark.

Under de följande timmarna minskar borrens prestation med uppskattningsvis 10-20% per timme. Den exakta minskningen är inte känd, då den är beroende av markförhållandena. Borren ska gå oavbrutet i 8 timmar.

Skriv ett JavaScript program som uppskattar det totala borrdjupet.

Ledning: Börja med att simulera minskningen av borrens prestation efter den 1:a timmen med slumptal mellan 10 och 20. Summera borrhålets djup efter den 1:a timmen baserad på denna simulation.

Skriv ut slutligen ett närmevärde för borrhålets totala djup efter 8 timmar.

Skriv ut även borrprestationens procentuella minskning per timme vid den aktuella körningen, t.ex.:

”Denna uppskattning baseras på 12% minskning av borrprestationen per timme.”

P.g.a. simuleringen med slumptal borde man få vid olika körningar olika procentsatser för minskningen av borrens prestation och därmed även andra uppskattnningar av det totala borrdjupet. I praktiken duger dock ofta en sådan uppsattning, då den kan vara en värdefull information för planeringen av arbetet.

En körning kan t.ex. se ut så här:

2. Palindrom – en lek med ord

En *palindrom* är en sträng som inte ändras när den läses baklänges. T.ex. är orden *rar*, *död* och *radar* palindromer, även namnet *Hannah* när det stavas så. Men även en text som *ni talar bra latin* är en palindrom om man ignorerar mellanslagen. Och det ska man göra. Därför: vid behandling av sådana texter i ett program låt koden först ta bort alla mellanslag.

Skriv en funktion **`palindrom()`** som avgör om en sträng är en *palindrom* eller ej. Anropa sedan funktionen i ett JavaScript program som hanterar strängar. Låt användaren mata in strängar – med eller utan mellanslag – så länge tills man hittat en palindrom. Bjud på möjligheten att avsluta om ingen palindrom hittas och föreslå användaren ett antal palindromer.

3. Frekvenstabell – stämmer sannolikhetsläran?

Skriv ett JavaScript program som simulerar tärningskast, dvs genererar slumptal mellan **`1`** och **`6`**, och ställer upp en frekvenstabell enligt följande beskrivning:

Frekvens är antalet förekomster av ett resultat (utfall) bland tärningens 6 möjliga.

Låt programmet genomföra olika antal simuleringar och räkna vid varje simulering frekvensen för varje resultat **`1`**, ...,**`6`** av tärningskastet. T.ex. ska man kunna läsa av från tabellen hur många gånger resultatet **`1`** förekommer när man kastar tärningen 50 gånger, 100 gånger, 1 000 gånger, 5 000 gånger, 10 000 gånger, osv. Avgör själv hur långt du går. Samma information ska man kunna läsa av från tabellen om tärningskastets andra resultat **`2`**, ... ,**`6`**.

Infoga i tabellen även en kolumn som för varje resultat av tärningskastet visar kvoten:

Frekvens / Antalet tärningskast

Denna kvot är den experimentella sannolikheten för ett visst resultat. Undersök på vilket sätt den experimentella sannolikheten närmar sig den ideala sannolikheten för varje resultat, som enligt sannolikhetsläran borde vara 1/6 eller 0,16667.

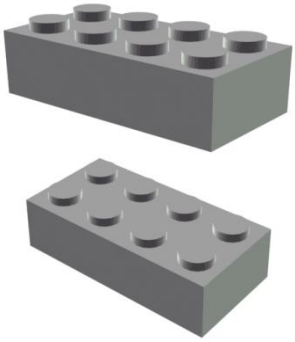
Kapitel 4

Funktioner

4.1 Funktionsbegreppet i programmering

Begreppet *funktion* härstammar från matematiken: Man har en formel y = f(x) som beräknar ett tal y utgående från ett annat tal x och säger: y är en funktion av x. Denna matematiska syn på funktion har tagits över till programmering som ett underliggande koncept och som en historisk utgångspunkt. Men under tiden har begreppet vidareutvecklats och fått en bredare tolkning då den inom programmering tillämpats på all datoriserad problemlösning. I programmering inkluderar funktioner även matematiska problem, men är inte begränsade till dem.

Modularisering eller Lego-principen

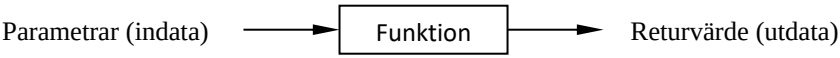


De flesta har väl någon gång som barn, eller tillsammans med sina barn, byggt ett hus, en bil eller liknande med Lego-bitar. Efter ett tag har huset kanske rasat och nya tekniska underverk har konstruerats. Men även de har någon gång plockats isär. Det enda som blivit kvar är själva Lego-bitarna som man så småningom samlat i en kartong för att kunna återanvända dem senare.

Vill man lösa ett komplext problem, t.ex. bygga ett hus eller en bil, bryter man ned det i ett antal mindre problem som är enklare att lösa. Sedan sätter man ihop de små enkla lösningarna till den stora komplexa lösningen. Principen heter *modularisering* och kan användas vid nästan all problemlösning. Ett stort komplext problem bryts ned i mindre *moduler* – motsvarande Lego-bitarna – och bearbetas en i taget. Varje modul löser ett delproblem som är oberoende av andra, är mindre än det stora problemet och därmed enklare att lösa. Sedan gäller det att sätta ihop modulerna till den stora lösningen. I programmering är dessa moduler *funktioner*:

En funktion är kod som definieras som en namngiven modul.
Koden utförs inte förrän funktionen anropas.

Vid anropet kan funktionen ta emot indata, s.k. parametrar,
bearbeta dem och returnera utdata, s.k. returvärde.



Man kan jämföra en funktion med en ”svart” låda i vilken man stoppar indata och får ut utdata: Indata kallas även *parametrar* (argument) och utdata *returvärde*.

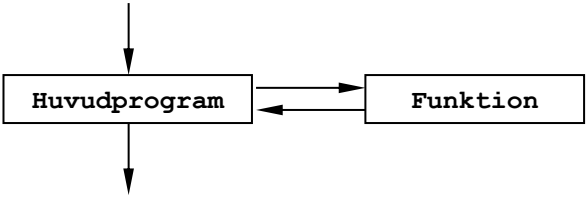
En funktion kan ha inga, en eller flera parametrar. Den kan ha inget eller endast *ett* returvärde, dvs en funktion kan inte ha flera returvärden, vilket är ett arv från matematiken. Både parametrarna och returvärdet kan vara tal, tecken, strängar, sanningsvärden, objekt eller andra datatyper. Funktionen bearbetar de inkommande parametrarna och returnerar returvärdet eller inget alls. I denna bemärkelse är en funktion ett *underprogram*, på eng. *subroutine* eller *procedure*.

”Svart” är lådan så länge vi inte vet hur den fungerar ”inuti”, dvs så länge vi inte själva definierat funktionen. I så fall använder vi den endast för att lösa ett visst problem. Det gäller t.ex. för de biblioteksfunktioner som vi hittills använt i våra program: **`document.writeline()`**, **`parseInt()`**, **`prompt()`**, **`alert()`** och andra. De är förprogrammerade och lagras i bibliotek. Vi anropar dem i våra program för att dra nytta av deras funktionalitet, utan att behöva veta hur de i detalj är konstruerade. Bibliotek som består av sådana ”svarta” lådor finns i alla programmeringsspråk.

Gränssnitt

För att att sätta ihop det hela måste varje modul kommunicera med sin omgivning. Även här kan man lära av Lego: Varje Lego-bit är konstruerad så att den passar in i en annan Lego-bit. Därför har varje Lego-bit två uppsättningar av *taggar*, en på varje sida. Taggarna är de delar av Lego-biten som tillåter denna passning och kan anses som Lego-bitens *gränssnitt* mot andra Lego-bitar. På samma sätt har en funktion ett gränssnitt mot andra delar av koden, för att kunna kommunicera med dem.

Även detta gränssnitt har två delar: För det första funktionens *parametrar* som importerar värden från omgivningen och för det andra funktionens *returvärde* som exporterar ett värde till omgivningen. Men sedan måste Lego-bitarna ”sättas ihop” vilket i programmeringstermer innebär att *anropa* den ena från den andra. Ett *anrop* av en funktion innebär att *aktivera* funktionen. Detta sker genom att ev. skicka till den parametrar, utföra koden som står i funktionen och ev. få tillbaka returvärdet. Generellt finns det i ett program flera funktioner som anropar varandra. Det enklast tänkbara exemplet är att huvudprogrammet anropar en **`Funktion`** Då kan programflödet mellan dem se ut så här:



Varför funktioner?

Kan man inte helt enkelt skriva kod rakt ned? Är detta med funktioner inte att krångla till det hela? Föreställ dig en verksamhet som växer med tiden, ett expanderande företag eller en organisation med stigande antal medlemmar. Hur

organiserar man jobbet? Man gör arbetsdelning. Man delegerar uppgifterna. Var och en får en väl definierad arbetsuppgift. Annars skulle man inte kunna klara av jobbet. Samma sak gör man med program vars kod växer. Lösningen är: man delar upp det stora programmet i mindre, logiskt meningsfulla delproblem, för att kunna klara av komplexiteten. Det finns i huvudsaken två viktiga skäl för användning av funktioner:

1. Återanvändning av kod

Samma idé finns bakom Lego-biten som minsta återanvändbara modul för att bygga i princip vad som helst. Har man i ett program löst ett litet delproblem som även dyker upp i andra sammanhang och vars kod kan vara relevant i andra program, så vill man ju helst inte satsa tid och resurser för att koda det en gång till. Man vill undvika att återuppfinna hjulet. Detta är inte bara av teoretiskt-estetiskt intresse utan även av stort ekonomiskt intresse. Det man gör är att lösa koden för det lilla delproblemet från det aktuella programmet och skriva den som en funktion för att kunna återanvända koden i vilket annat program som helst. Man behöver då endast anropa den från andra program. Det kräver förstås att den ursprungliga koden som kanske var skräddarsydd för just det speciella programmet då, nu som funktion måste formuleras på ett mer generellt sätt och förses med möjligheten att kunna kommunicera med andra program. Därför måste koden kompletteras med parametrar och returvärdet. Hela tanken bakom standardbibliotek – inte bara i C++ utan i alla programspråk – bygger på idén om återanvändning av kod. Även om man väljer att inte skriva egna funktioner kan man i alla fall inte komma ifrån att använda redan fördefinierade funktioner från standardbiblioteket.

2. Strukturering av program

Genom att modularisera ett komplext problem som ska lösas med hjälp av datorn underlättar man inte bara själva lösningen (innehållet) utan kan även lättare få en strukturering av programkoden (formen). Det enklast tänkbara sättet att strukturera vilket program som helst är t.ex. att dela in det i ***inmatning – bearbetning – utmatning***. Dessa tre delar kan skrivas i var sin funktion vilka sedan anropas av `main()`. Denna huvudfunktion kan då bestå av ett få antal satser som endast anropar programmets olika funktioner. På så sätt har man från huvudprogrammet en övergripande kontroll över hela programflödet. Så kan man så småningom bygga upp sitt eget bibliotek av egendefinierade funktioner.

Funktionen `max` (nedan) 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 en återanvändbar modul, utan att behöva återuppfinna hjulet.

Vår första funktion

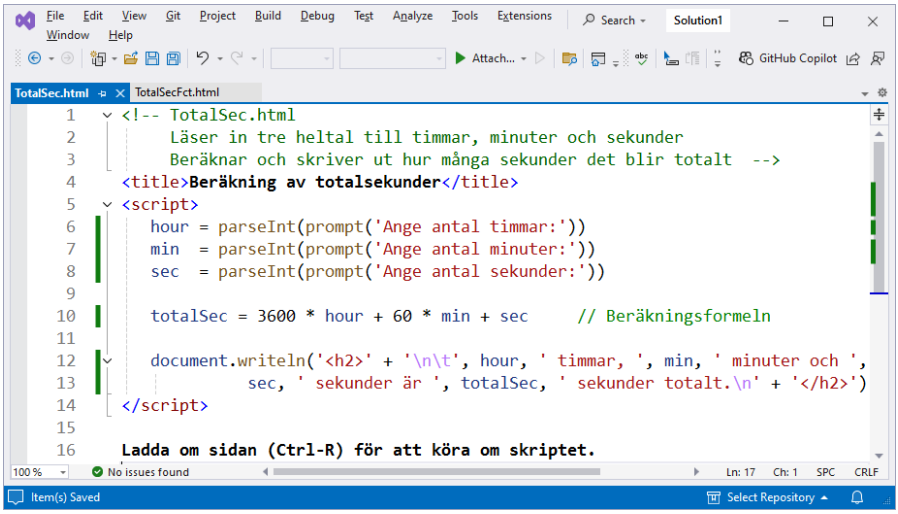
```
1  <!-- MaxFct.html
2       Definierar och anropar funktionen max() som bestämmer
3       det största bland tre tal  -->
4  <title>En egendefinierad funktion</title>
5  <script>
6
7     function max(a, b, c)   // Definierar funktionen max()
8     {
9        tmp = a              // Antar att a är 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.
```

Raderna **7-15** definierar funktionen max() och rad **20**n anropar den.

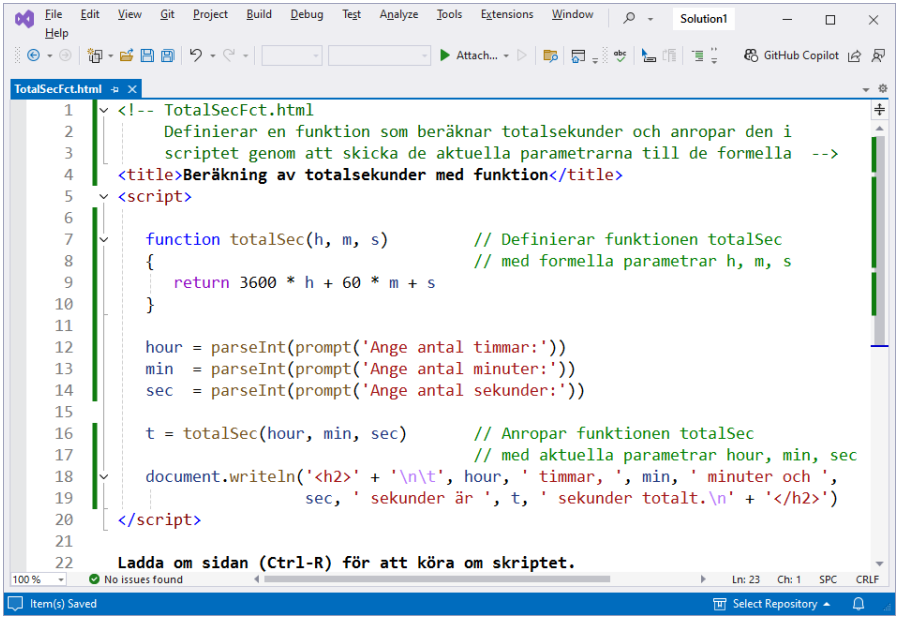
Rad **7** kallas för funktionens *huvud* och inleds med det reserverade ordet **function** (sid 12). Funktionens *namn* är **max()**. Parentesen **(a, b, c)** kallas för *parameterlistan*. **a, b** och **c** är funktionens *formella parametrar*, medan **no1, no2** och **no3** som står i funktionsanropet (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.

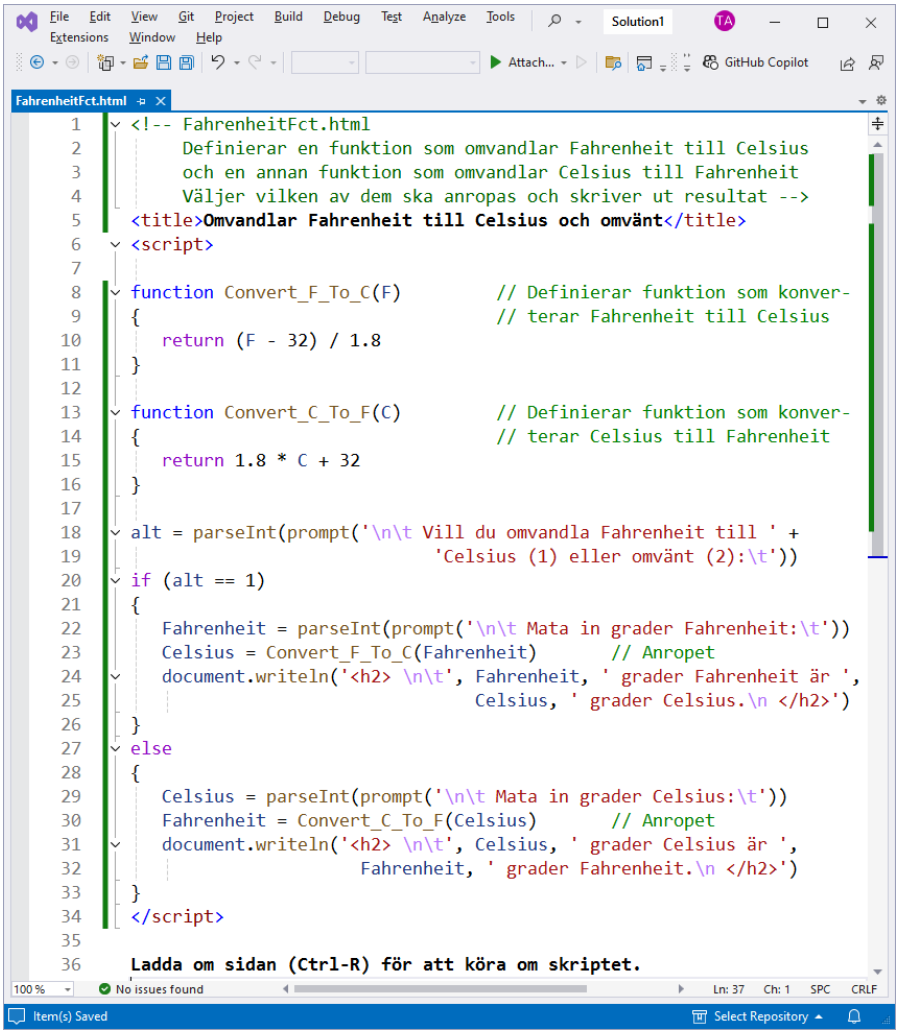
4.2 Formella och aktuella parametrar



```
<!-- TotalSec.html
     Läser in tre heltal till timmar, minuter och sekunder
     Beräknar och skriver ut hur många sekunder det blir totalt  -->
<title>Beräkning av totalsekunder</title>
<script>
   hour = parseInt(prompt('Ange antal timmar:'))
   min  = parseInt(prompt('Ange antal minuter:'))
   sec  = parseInt(prompt('Ange antal sekunder:'))

   totalSec = 3600 * hour + 60 * min + sec     // Beräkningsformeln

   document.writeln('<h2>' + '\n\t', hour, ' timmar, ', min, ' minuter och ',
            sec, ' sekunder är ', totalSec, ' sekunder totalt.\n' + '</h2>')
</script>

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



```
<!-- TotalSecFct.html
     Definierar en funktion som beräknar totalsekunder och anropar den i
     scriptet genom att skicka de aktuella parametrarna till de formella  -->
<title>Beräkning av totalsekunder med funktion</title>
<script>

   function totalSec(h, m, s)          // Definierar funktionen totalSec
   {                                   // med formella parametrar h, m, s
      return 3600 * h + 60 * m + s
   }

   hour = parseInt(prompt('Ange antal timmar:'))
   min  = parseInt(prompt('Ange antal minuter:'))
   sec  = parseInt(prompt('Ange antal sekunder:'))

   t = totalSec(hour, min, sec)        // Anropar funktionen totalSec
                                       // med aktuella parametrar hour, min, sec
   document.writeln('<h2>' + '\n\t', hour, ' timmar, ', min, ' minuter och ',
                    sec, ' sekunder är ', t, ' sekunder totalt.\n' + '</h2>')
</script>

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

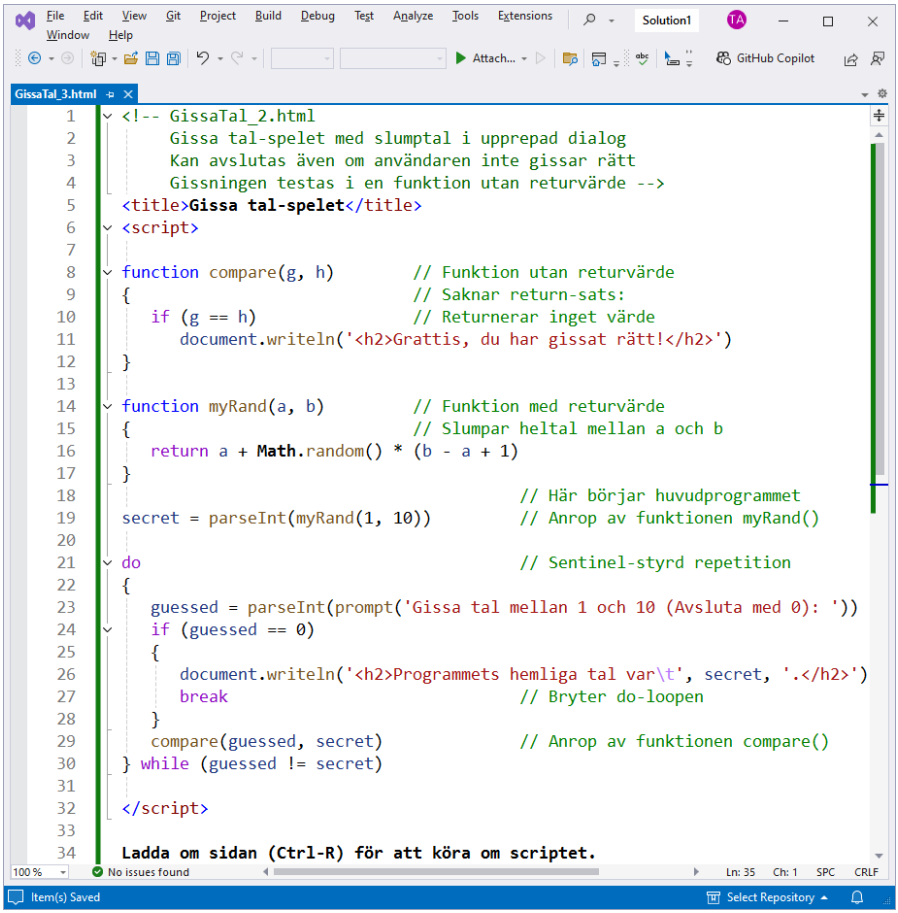

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Solution1
Attach...
GitHub Copilot
FahrenheitFct.html
<!-- FahrenheitFct.html
Definierar en funktion som omvandlar Fahrenheit till Celsius
och en annan funktion som omvandlar Celsius till Fahrenheit
Väljer vilken av dem ska anropas och skriver ut resultat -->
<title>Omvandlar Fahrenheit till Celsius och omvänt</title>
<script>
function Convert_F_To_C(F) // Definierar funktion som konver-
{ // terar Fahrenheit till Celsius
return (F - 32) / 1.8
}
function Convert_C_To_F(C) // Definierar funktion som konver-
{ // terar Celsius till Fahrenheit
return 1.8 * C + 32
}
alt = parseInt(prompt('\n\t Vill du omvandla Fahrenheit till ' +
'Celsius (1) eller omvänt (2):\t'))
if (alt == 1)
{
Fahrenheit = parseInt(prompt('\n\t Mata in grader Fahrenheit:\t'))
Celsius = Convert_F_To_C(Fahrenheit) // Anropet
document.writeln('<h2> \n\t', Fahrenheit, ' grader Fahrenheit är ',
Celsius, ' grader Celsius.\n </h2>')
}
else
{
Celsius = parseInt(prompt('\n\t Mata in grader Celsius:\t'))
Fahrenheit = Convert_C_To_F(Celsius) // Anropet
document.writeln('<h2> \n\t', Celsius, ' grader Celsius är ',
Fahrenheit, ' grader Fahrenheit.\n </h2>')
}
</script>
Ladda om sidan (Ctrl-R) för att köra om skriptet.
100 %
No issues found
Ln: 37 Ch: 1 SPC CRLF
Item(s) Saved
Select Repository

4.3 Funktioner utan returvärde

Hittills hade alla våra funktioner returvärden. Det var *en* typ av funktion, en ganska viktig sådan. Men funktioner med returvärde har en begränsning: De kan returnera endast *ett* värde, *ett* tal, *ett* tecken, *ett* sanningsvärde, *en* sträng eller *ett* objekt, inte flera. En annan typ av funktioner är sådana som inte har något returvärde alls. I denna bemärkelse pratar vi nu om *funktioner utan returvärde*.

Exempel på en funktion utan returvärde

Funktionen `compare()` i scriptet **`GissaTal2`**, raderna **`8-12`** (nedan) , är ett exempel på en funktion utan returvärde. Den har ingen **`return`**-sats. Istället skriver den ut resultatet av en jämförelse mellan två tal – en typisk användning av en funktion utan returvärde.

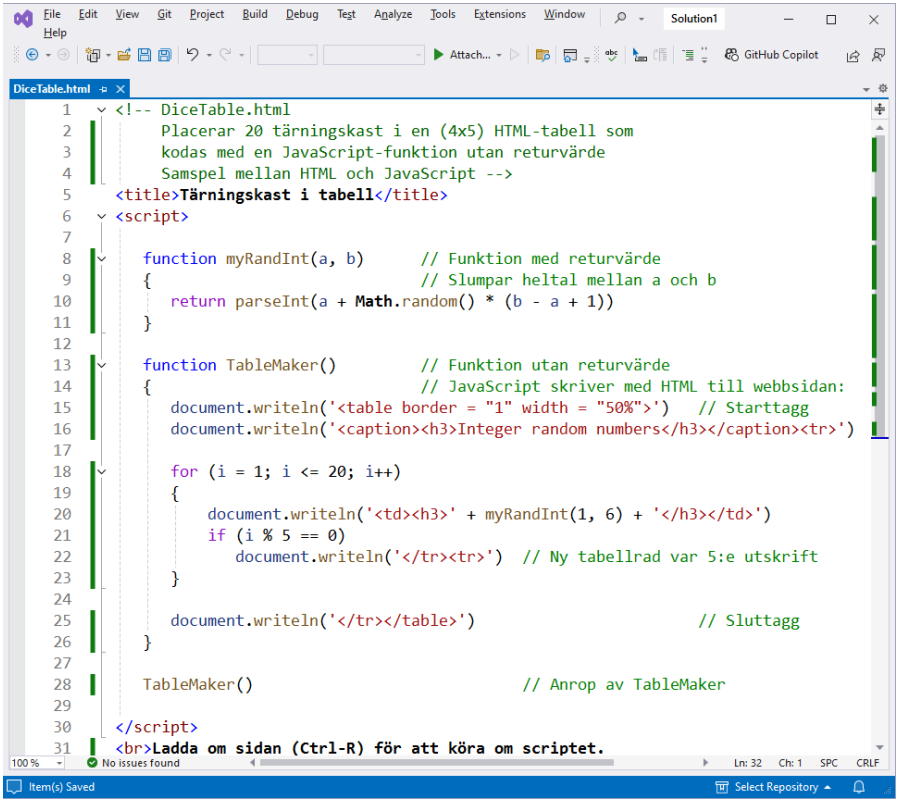


Man hade även kunna skriva en tom **`return`**-sats i funktionen som skulle i så fall avsluta funktionen. Testa gärna att lägga in en tom **`return`**-stas.

Så här kan vi sammanfatta:

> En funktion utan returvärde returnerar inte något värde. Den kan ha ingen **`return`**-sats alls eller en tom **`return`**-sats, dvs: **`return`**, som i så fall avslutar funktionen.

4.4 Tärningskast i tabell



Funktionen TableMaker()

Integer random numbers

3	5	6	2	2
2	1	3	6	5
6	6	2	6	2
6	2	6	2	5

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

Denna funktion som gör huvudjobbet i scriptet **`DiceTable`** (ovan), returnerar inget värde. Den konstruerar en tabell och skriver i den 20 tärningskast. Tabellen har 4 rader och 5 kolumner och skrivs ut med en enkel `for`-sats. Loopen är inte nästlad, för att åstadkomma 2D-tabellstrukturen. Istället använder den sig av en `if`-sats och modulooperatorn % för att byta rad var 5:e utskrift. Men i själva verket ”byter” den inte rad på ett traditionellt sätt med `\n`, utan skapar nya rader var 5:e utskrift medr **`table`**-elementets **`tr`**-tagg *(table row*. Det är *HTML-kod* som skickas till *JavaScript*-funktionen `document.wri-`

`teln()`, se rad **22**, vilket är anmärkningsvärt. Därför vill vi titta närmare på denna konstruktion:

Samspel mellan JavaScript och HTML

I scriptet **DiceTable** på rad **15** öppnas HTML-elementet **table** i `document.writeln()`:s parameterlista och går över flera rader. Det stängs på rad **25**. **table**-elementet finns fördelat över funktionen `TableMaker()` i de olika satser som anropar `document.writeln()`, inkl. i `for`-loopen. Som vanligt används **tr** *(table row)* i **table**-elementet, för att skapa rader och **td** *(table data)* för att skapa dataceller.

På rad **22** i `for`-loopen stängs i varje varv förra raden och öppnas en ny rad. Och genom att lägga den i `if`-satsen görs detta var 5:e utskrift. Man skulle kunna extrahera all HTML-kod från `document.writeln()`:s parameterlistor och skriva den i ett separat script, för att se hur en cell i tabellen uppstår, se övn 4.5 b).

Scriptet `DiceTable`:s överordnade struktur

Scriptet består av de två funktionerna `myRandInt()` i och `TableMaker()` samt huvudprogrammet som endast består av anropet av funktionen `TableMaker()` på rad **28**. Anropet av funktionen `myRandInt()` däremot finns inte i huvudprogrammet. Denna funktion anropas i funktionen `TableMaker()`, närmare bestämt i `for`-satsen på rad **20**.

Övningar till Kapitel 4

4.1 Skriv om scriptet **MaxFct** (sid 83) så att funktionen **max()** byggs in (tillbaka) i scriptet och varken definieras eller anropas. Dvs skriv ett nytt script som inte innehåller någon funktion, men gör samma sak som scriptet **MaxFct**. I slutet ska scriptet **MaxFct** framstå som en modularisering av ditt nya script.

4.2 Modifiera scriptet **TotalSecFct** (sid 84) så här:

a) Spara undan i funktionen **totalSec()** beräkningen av totalsekunder i en variabel `resultat`. Sätt sedan alla formella parametrar till `1` och skriv ut dem från funktionen. Returnera `resultat`.

b) Skriv ut de aktuella parametrarna från huvudprogrammet. Anropa funktionen. Skriv ut de aktuella parametrarna igen efter anropet.

Vilka slutsatser drar du från experimentet ovan?

4.3 Ta scriptet **FahrenheitFct** (sid 85) och skriv om det, för att omvandla enheterna *grader* och *radianer* till varandra enligt följande:

a) Det finns två olika enheter för att mäta vinklar: *grader* och *radianer*.
1 grad = (π / 180) radianer
1 radian = (180 / π) grader
Ta själv reda på hur man kodar π i JavaScript. Annars ta π = 3,14.

b) Definiera en funktion som omvandlar grader till radianer och en annan som gör det omvända. Anropa båda funktionerna från huvudprogrammet. Testa olika inmatningar.

I övrigt ta över upplägget från scriptet **FahrenheitFct** (sid 85).

4.4 Modifiera scriptet **GissaTal_2** (sid 86) på tre olika sätt:

a) Bygg in en räknare i scriptet som får reda på antalet gissningar. Skriv ut efter hur många gissningar användaren lyckats gissa rätt.

b) Ersätt huvudprogrammets `do`-sats med en `while`-sats. Hur måste du i så fall placera inläsningen av användarens gissning? Vilken lösning, tror du, är bättre?

c) Vidareutveckla funktionen `compare()` så att den jämför gissningen `g` med programmets hemliga tal `s` inte bara på likhet utan även om `g` är mindre eller större än `s`. Skriv ut vid varje gissning en hjälp till användaren, om den ska gissa högre eller lägre nästa gång. Om detta går att implementera i JavaScript, presentera din lösning. Annars ange orsaken.

4.5 a) Modifiera scriptet **DiceTable** så att det producerar en (10 x 7)-tabell över tärningskast. Bibehåll scriptets överordnade struktur.

b) Extrahera från scriptet **DiceTable** (sid 88) HTML-elementet **table** som sträcker sig över funktionen `TableMaker()` i de olika satser som anropar `document.writeln()`, inkl. i `for`-loopen. Skriv hela elementet sammanhängande utan loop i ett script och kör. Vad får du för utskrift?

c) I JavaScript kan man markera strängar både med citationstecken och apostrofer. Ersätt i scriptet **DiceTable** alla apostrofer som förekommer i parameterlistan av `document.writeln()`, med citationstecken. Gör de ändringar i koden som blir nödvändiga som en konsekvens av denna åtgärd. Vilka ändringar, förklaras under rubriken *Apostrof vs. Citationstecken* på sid 67.

4.6 Definiera funktionen $y = f(x) = x^3$ i Javascript.

a) Inkludera funktionen i ett script som anropar den för att skriva ut en värdetabell för alla heltal *x* i intervallet [-5, 5].

b) Skriv värdena från a) i arrays: Skapa en array för alla *x* och en för alla *y*. Lägg in resp. värdena i dem och skriv ut dem. Markera med kommentar funktionens definition och anrop.

4.7 Skriv de fyra räknesätten och heltalsdivisionen samt modulo som funktioner i JavaScrip. Läs in två heltal. Anropa funktionerna och skriv ut resultaten så att du får t.ex. följande utskrift när du läser in `5` och `3`:

```
5  +  3  ger  8
5  -  3  ger  2
5  *  3  ger  15
5  /  3  ger  1.6666666666666667
5  // 3  ger  1
5  %  3  ger  2
```

4.8 Modularisera scriptet **IfElse** (sid 46) så genom att flytta kodn som avgör om det inmatade talet är jämnt eller udda, till en funktion. Välj olika variabler för den formella och den aktuella parametern. Anropa funktionen. Testa ditt program för olika inmatningar.

4.9 Modularisera programmet **GissaTal** (sid 49) genom att definiera if-else-stegen som en funktion.

Kapitel 5

Arrays

5.1 Deklaration och initiering av arrays

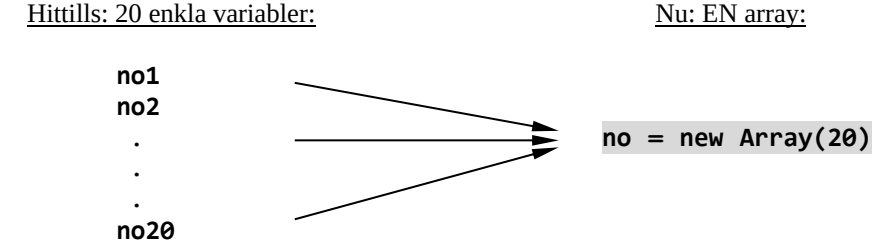
Datorn har några egenskaper som är helt överlägsna motsvarande egenskaper hos människan: snabbheten, noggrannheten och förmågan att effektivt lagra och hantera stora datamängder samt förmågan att aldrig bli trött.

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**.

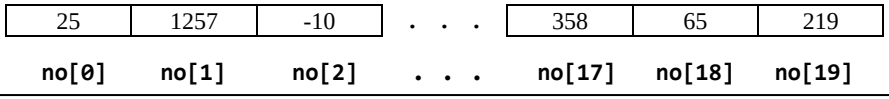
Arrayvariabeln **no** ersätter de 20 vanliga variablerna **no1**, **no2**, …, **no20** och består nu i sin tur av 20 *element*. Varje element är en variabel som kan lagra ett värde. Enda skillnaden är *sättet* dvs *koden* att komma åt dessa värden. Indexet är ett nummer som specificerar varje elements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

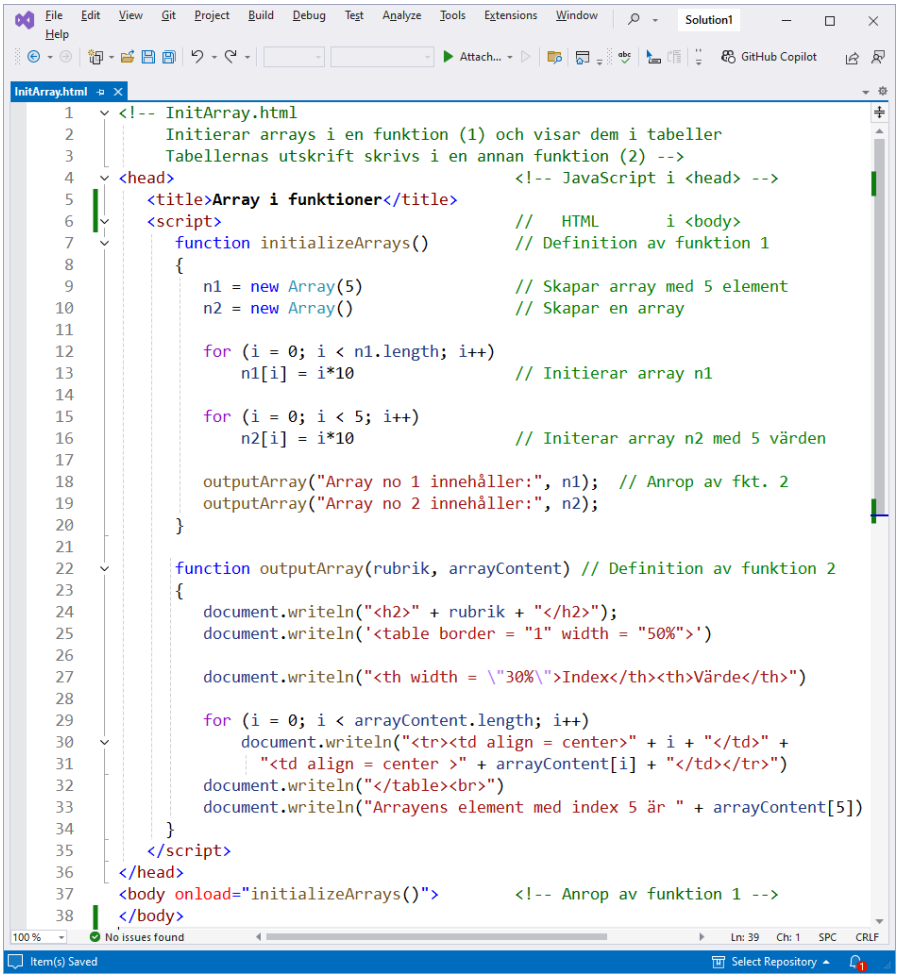
Av indexregeln följer dessutom att negativa index generellt inte är tillåtna.

Array i funktioner

Programmet **InitArray** på nästa sida demonstrerar allt vi sagt om arrays speciellt indexregeln. Dessutom kan vi se, hur JavaScript hanterar överskridningen av de definierade indexgränserna.

En annan egenskap av **InitArray** är att den är modularierad. Scriptet kombinerar arrays med funktioner, genom att definiera deklarationen och initieringen av arrays i en funktion (1) och tabellutskriften av arrayens innehåll i en annan funktion (2).

Medan funktion 1 anropar funktion 2, anropas den själv i scriptets **body**-del. Annars är funktionernas definition placerade i scriptets **head**-del.



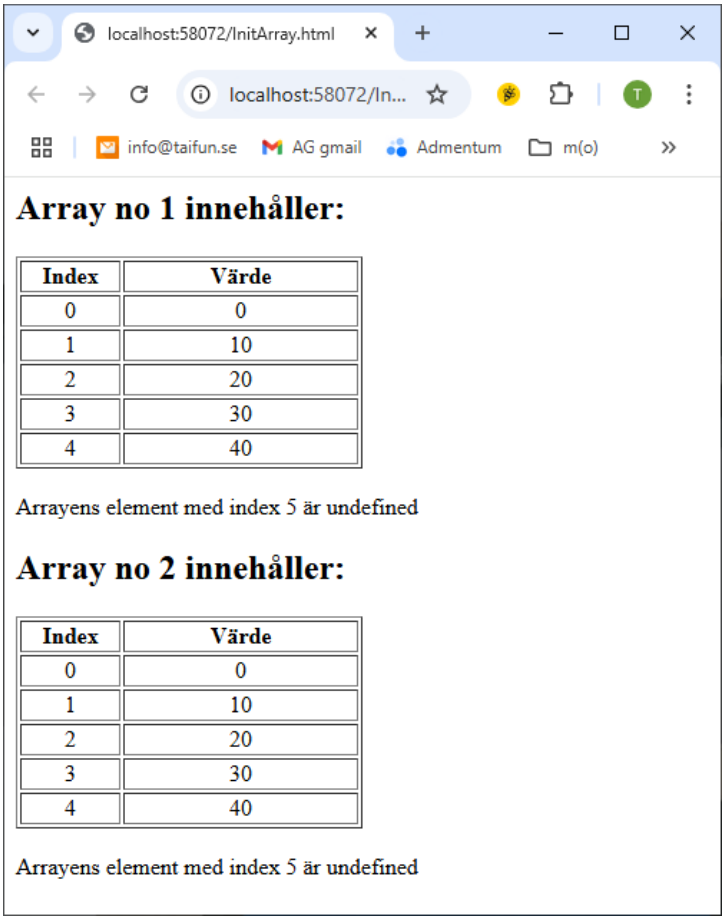
Odefinierade element i en array

Körningen visar att icke-definierade arrayelement inte leder till något fel. Index **5** överskrider de definierade indexgränserna. Arrayelementet **arrayContent[5]** på rad **33** 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”**.

En körning visar detta:



Ett annat namn än det definierade arraynamnet **`arrayContent`** leder till fel. Om vi däremot använder ett index som överskrider de definierade gränserna, kan vi fortfarande 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.

5.2 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:

```
<!-- InitLista.html
     Kortform för definition och initiering av en array med
     en initieringslista. Direkt tilldelning till en kopia -->
<title>Initieringslista</title>
<script>
   no = [64, 86, 34]       // Kortform på definition och
                           // initiering med initieringslistan
   copy = no               // Tilldelning med arraynamnet
                           // Elementvis utskrift av kopian:
   document.writeln('<h3>Kopians första element copy[0]'   +
                    ' har värdet ' + copy[0] + '.<br><br>' +
                    'Kopians andra element no[1]'          +
                    ' har värdet ' + copy[1] + '.<br><br>' +
                    'Kopians tredje element copy[2]'       +
                    ' har värdet ' + copy[2] + '.<br><br>' +
                    'Kopians fjärde element copy[3]'       +
                    ' har värdet ' + copy[3] + '.</h3>'    )
                  // Utskrift av kopian med arraynamnet:
   document.writeln('<h3>Utskrift med arraynamnet: '       +
                                  'copy = ' + copy + '</h3>' )
</script>
```

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

Kopians första element copy[0] har värdet 64.

Kopians andra element copy[1] har värdet 86.

Kopians tredje element copy[2] har värdet 34.

Kopians fjärde element copy[3] har värdet undefined.

Utskrift med arraynamnet: copy = 64,86,34

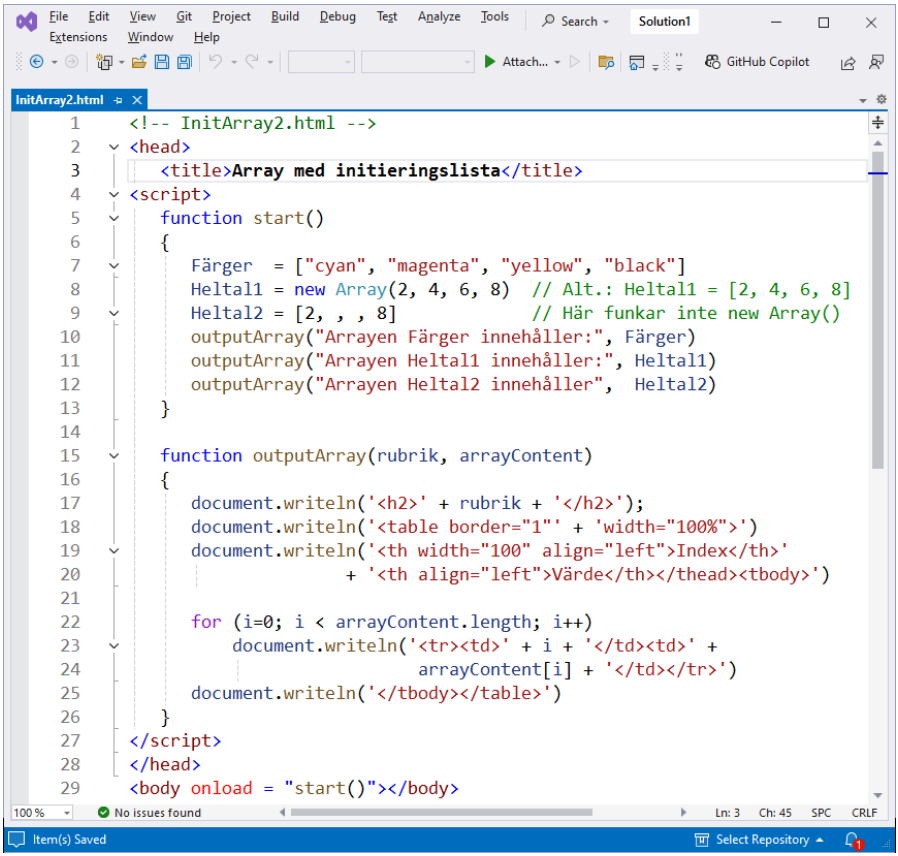
Både definitionssatsen och initieringssatserna i **ArrayDef** – det är de 4 första satserna – kan slås ihop till en enda sats.

Initieringslista

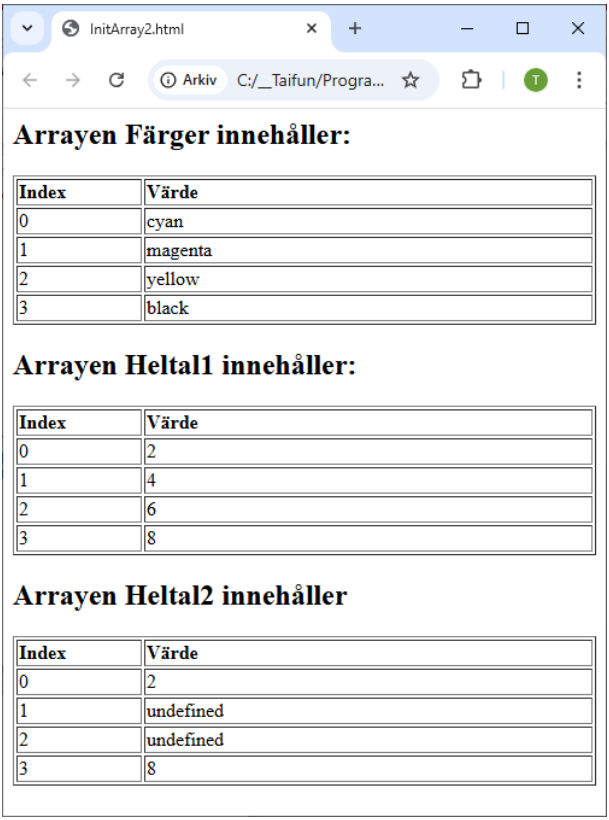
```
no = [64, 86, 34]    // Kortform för definition och
                     // initiering med initieringslistan
```

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 **for**-satserna i scriptet **InitArray** (sid 95) som initierar arrayen elementvis och gör precis samma sak som de. JavaScript-interpretatorn får informationen om arrayens storlek i initieringslistan, genom att räkna antalet kommaseparerade element inom hakparenteserna **[ ]**. Observera att man får använda kortformen ovan endast i samma sats som deklarationen av variabeln **no**.

Exempel med funktion



Ett körexempel visas på nästa sida:



Arrayen Färger innehåller:

Index	Värde
0	cyan
1	magenta
2	yellow
3	black

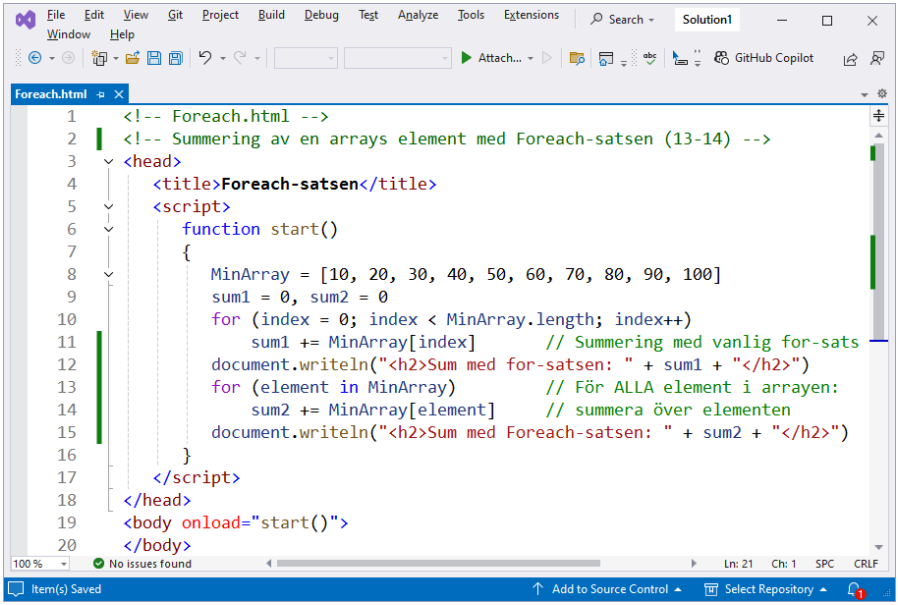
Arrayen Heltal1 innehåller:

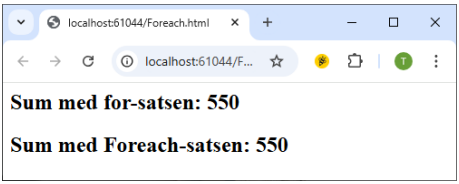
Index	Värde
0	2
1	4
2	6
3	8

Arrayen Heltal2 innehåller

Index	Värde
0	2
1	undefined
2	undefined
3	8

5.3 Foreach-satsen, en ny kontrollstruktur





Scriptet **Foreach** summerar värdena i arrayen **MinArray** på två olika sätt: en gång med en vanlig **for**-sats på raderna **10-11** och en annan gång med en annan sats på raderna **13-14** som vi kallar för **Foreach**-satsen. Dess syntax skiljer sig från den vanliga **for**-satsen. Den nya satsen är i själva verket en ny kontrollstruktur som vi inte använt hittills. Att den nu introduceras i samband med arrays är inte en tillfällighet. Vi kommer att se varför. Ovan visas ett körexempel.

Foreach-*satsen*

```
for (element in MinArray)
    sum2 += MinArray[element]
```

Översatt till svenska:

För varje **element** av arrayen **MinArray**
summera elementen till **sum2**.

Denna sats som används på raderna **13-14** summerar värdena i arrayen **MinArray**. Vi kunde inte ta upp den i kapitlet om kontrollstrukturer därför att den förutsätter array-begreppet eller liknande sammansatta datatyper, som vi då inte hade hunnit gå igenom. Den är t.o.m. idealisk för att hantera arrays. Den gör samma sak som

for-satsen, men har en lite annorlunda – ja t.o.m. enklare syntax, om man är förtrogen med arrays.

element – ett namn som är valt av oss – kallas för **Foreach**-satsens *iterationsvariabel*. **element** pekar på värdet (innehållet) som står i arrayen. Iteration betyder upprepning och innebär här att satsens kropp upprepas: Programflödet fortskrider från element till element tills alla element är genomgångna. Det reserverade ordet **in** betyder *av* eller *element av*. **MinArray** pekar på arrayen som ska loopas igenom. Därför: ” För varje **element** *av* arrayen **MinArray**”.

Foreach-satsens enkelhet består i att den till skillnad från **for**-satsen varken behöver ett start-, steg- eller slutvärde resp. avslutningsvillkor. Den går helt enkelt igenom arrayens *alla* element, från det första till det sista. Det är själva arrayen som bestämmer start-, steg- och slutvärdena. Variabeln **element** pekar i varje varv av loopen på resp. arrayelementets värde och kan sedan användas i loopens kropp för att göra det man önskar. I vårt exempel för att summera arrayens element.

En viktig egenskap av iterationsvariabeln är att den inte kan ändra arrayelementens värden i **Foreach**-satsens kropp. Den är så att säga *read only*. I praktiken innebär detta att iterationsvariabeln inte får förekomma till vänster om tilldelningsoperatorn (**=**) i någon sats i **Foreach**-satsens kropp. Vill man i **Foreach**-satsens kropp ändra på arrayelementens värden måste man använda **for**-satsen istället med arrayens index som räknare.

Att iterera över en arrays alla element

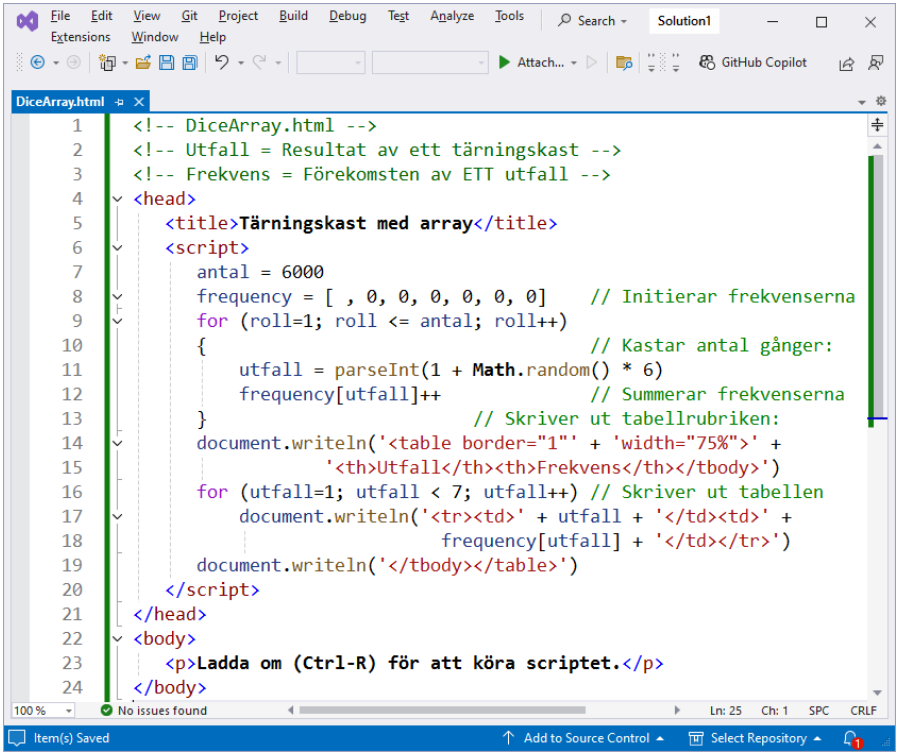
Att **Foreach**-satsens syntax är trots enkelhet mer sofistikerad än så, kan man se när man tittar på **Foreach**-satsens kropp på rad **14**:

```
sum2 += MinArray[element]
```

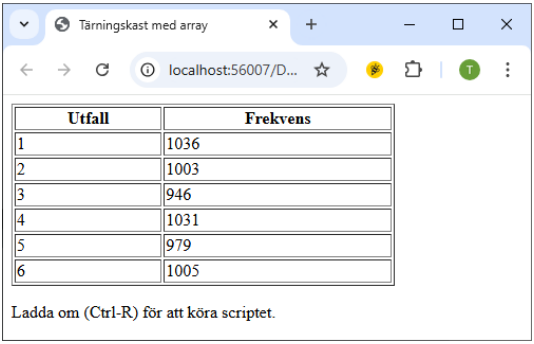
Här står iterationsvariabeln **element** på en plats som egentligen är reserverad för räknaren (indexet). Men **element** är ju inte något index, utan en variabel som i varje varv av loopen pekar på resp. arrayelementets värde, utan att vara identisk med själva värdet. Just här hade det varit kanske bättre att välja ett annat namn för iterationsvariabeln än **element**, för att framhäva skillnaden.

I själva verket tilldelas **element** i varje varv av loopen det aktuella arrayelementets värde – dvs ett annat värde i varje varv. Därför antar **element** här *rollen* av en ”räknare”. Man kan också säga att vi har att göra med en ny betydelse av hakparentesen: Den kan innehålla **Foreach**-satsens iterationsvariabel, för att iterera över en arrays alla element genom att använda iterationsvariabeln istället för räknare.

5.4 Tärningskast med array



Vi har redan kodat tärningskast tidigare, men då var det med funktioner (sid 88). Här ska vi använda en array, för att simulera 6 000 tärningskast, skriva ut resultaten *(utfall)* och räkna även antalet förekomster av ett utfall *(frekvens)*. Detta görs i scriptet **`DiceArray`** i `for`-satsen på raderna **`9-13`**.



Att ignorera index 0

I scriptet **`DiceArray`** ovan har vi på rad **`8`** initierat arrayen **`frequency`** så att det första elementet, det med index 0, är odefinierat:

```
frequency = [ , 0, 0, 0, 0, 0, 0]
```

Vi har alltså ignorerat index 0 för att kunna simulera de 6 möjliga utfallen av en tär-

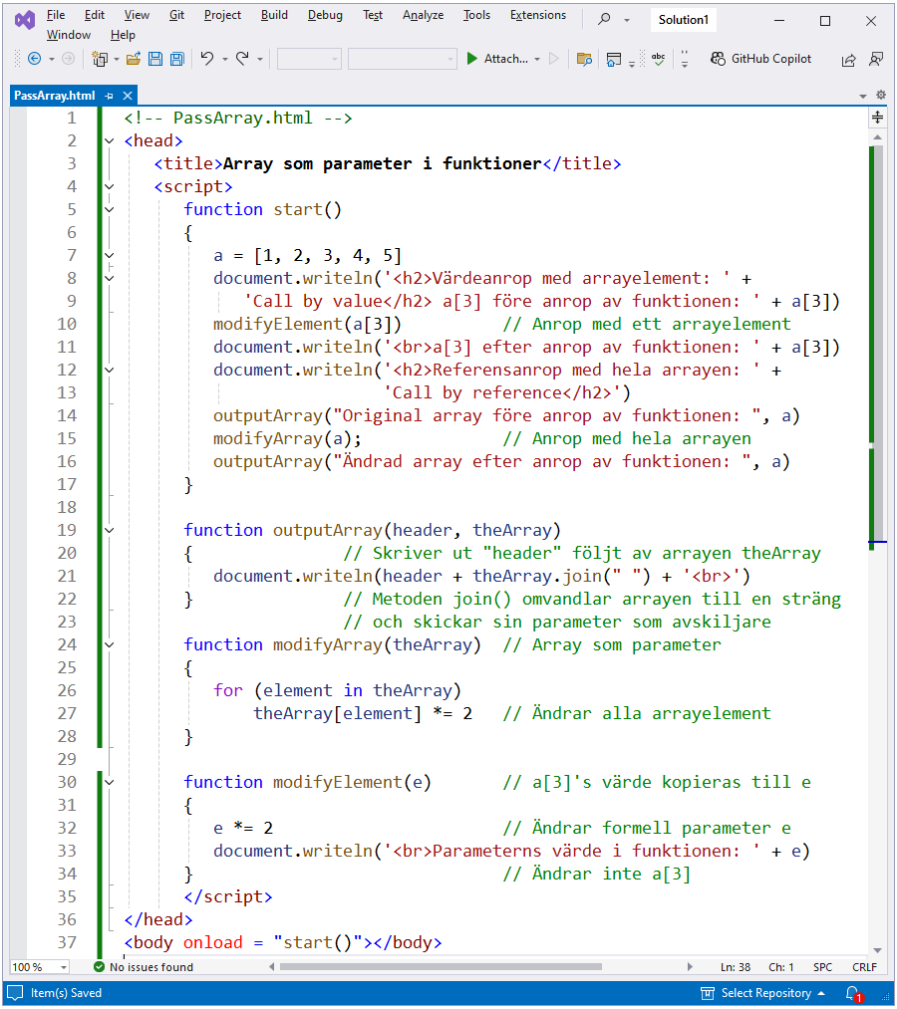
ning som är `1, 2, 3, 4, 5, 6`, som inte innehåller `0` i resten av scriptet. Detta har gjorts speciellt på rad `12` där frekvenserna av ett utfall summeras i en `for`-sats:

```
frequency[utfall]++
```

Här får utfall inte vara `0`, för ett sådant utfall finns inte hos tärningen. Dessutom har vi i satsen innan låtit slumpgeneratorn att slumpa fram ett slumptal mellan `1` och `6`.

Vi använder alltså ett ”otillåtet” verktyg, nämligen ett odefinierat element, för att anpassa arrayens sätt att hantera sina index (att börja med 0) till vårt syfte att simulersa tärningskast.

5.5 Array i funktioner



Scriptet **PassArray** skapar en array på rad **7** och initierar den samtidigt. Ett av arrayens element skickas till en funktion (rad **10**) och hela arrayen skickas till en (annan) funktion (rad **15**). Det första anropet på rad **10** är som vanligt, eftersom `a[3]` är *ett* värde. Men i det andra anropet på rad **15** är parametern en array, vilket visar att man kan skickar en array som parameter till en funktion genom att skriva arrayens namn, i det här fallet `a`, i parameterlistan. Och detta utan hakparenteser och utan uppgiften om arrayens storlek – som om arrayen vore en vanlig variabel.

Körresultatet visas på nästa sida.

Array som parameter i funktioner

I funktionens definition på raderna **24-28** tas den aktuella parametern `a` emot av den formella parametern `theArray`. Även där finns inga spår av varken hakparenteser eller av arrayens storlek. I funktionen `modifyArray()` multipliceras arrayens alla element med `2` i en `foreach`-sats (sid 100).

Som man ser i körresultatet till höger på sista raden syns denna ändring även efter anropet av funktionen i programmet. I resten av scriptet **PassArray** undersöks om denna ändring av arrayens element i funktionen även sysns utanför funktionen, även gäller när man gör samma sak med arrayens enskilda element `a[3]`. Körresultatet visar nämligen att detta *inte* är fallet. Vi ska nu förklara varför?

Värdeanrop: Call by value

Funktionen `modifyElement()` anropas på rad **10** och är definierad på raderna **30-34**. Vid anropet skickas elementet `a[3]` som aktuell parameter till funktionen och tas emot av den formella parameter `e`. I funktionen multipliceras `e` med `2` (rad **32**). Men denna ändring sysns inte *efter* anropet, när `a[3]` skrivs ut på rad **11**. Det är egentligen inte konstigt, eftersom `a[3]` och `e` är två olika variabler som har sin giltighet i var sitt kodområde (scope). Närmare bestämt är `e` en lokal variabel i funktionen `modifyElement()` och är inte giltig utanför funktionen. `a[3]` däremot är arrayen `a`:s fjärde element som har initierats där och gäller i hela scriptet.

Det som händer vid anropet på rad **10** är att `a[3]`:s *värde* som är `4`, kopieras över från minnescellen `a[3]` till minnescellen `e`. Kopians värde ändras, men detta påverkar inte originalet, vilket man ser i körresultatet. Denna metod för parameteröverföring kallas i programmering för *Värdeanrop*, på eng. *Call by value*. Beteckningen motiveras av att det är *värdet* på den aktuella parametern `a[3]` som överförs till den formella parametern `e`. Värdeanrop tillämpas automatiskt, när parametern är av enkel datatyp. En helt annan metod för parameteröverföring tillämpas när parametern är en *array*:

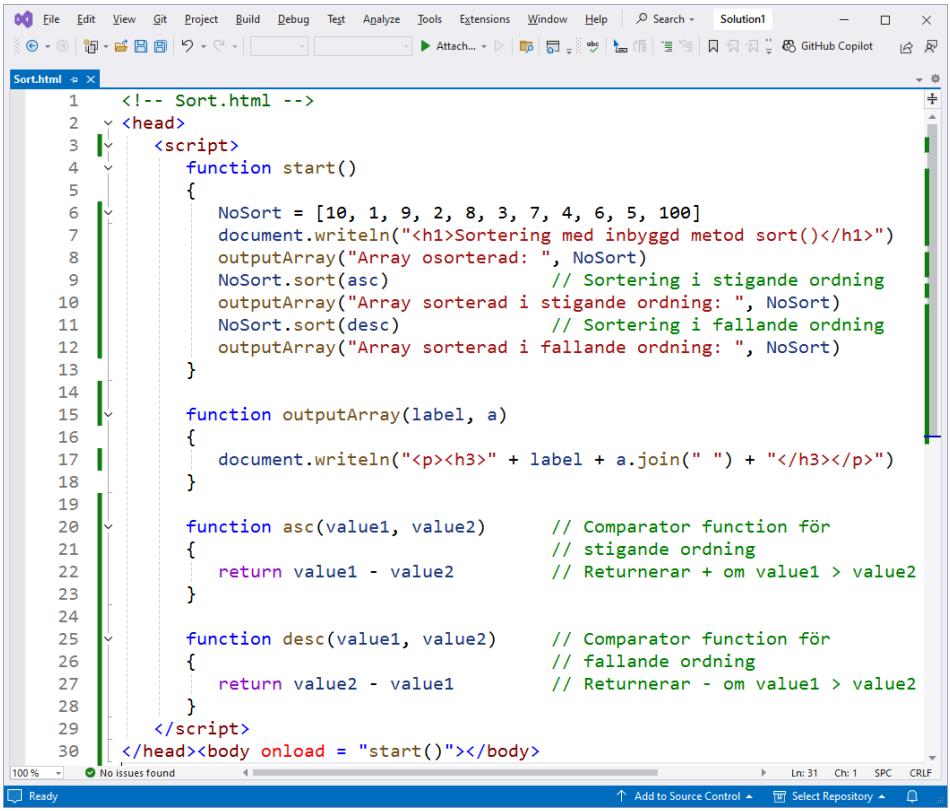
Referensanrop: Call by reference

Funktionen `modifyArray()` anropas på rad **15** och är definierad på raderna **24-28**. Vid anropet skickas arrayen `a` som aktuell parameter till funktionen och tas emot av den formella parameter `theArray`. I funktionen multipliceras alla element av `a` med `2` med hjälp av en `foreach`-sats (rader **26-27**). Sista raden i körresultatet visar att denna ändring sysns *efter* anropet, när `a` skrivs ut på rad **16**: Arrayens alla element är

fördubblade. Detta är märkligt, speciellt med den argumentation som framfördes vid värdeanropet och med tanke på att den aktuella och den formella parametern har olika namn: `a` och `theArray`. Förklaringen är datatypen array som inte längre är en enkel, utan en sammansatt datatyp:

Det som händer vid anropet på rad **15** är att inte arrayen `a`:s värden, utan dess *adress* skickas till funktionen. Ett annat namn för adress är *referens*. Den formella parametern `theArray` blir en referens till samma array som `a`. Minnescellerna till arrayen `a` får nu ett slags alias (referens) som pekar på samma array. När arrayens element fördubblas med hjälp av referensen `theArray`, skrivs ut dem med `a` och har med sig ändringen, vilket man ser i körresultatet. Denna metod för parameteröverföring kallas i programmering för *Referensanrop*, på eng. *Call by reference*. Beteckningen motiveras av att det är den aktuella parametern `a`:s *referens* (adress) som överförs till den formella parametern `theArray`. Referensanrop tillämpas automatiskt när parametern är en *array*, dvs en *sammansatt datatyp*.

5.6 Sortering av arrays



Scriptet **Sort** skapar och initierar en array på rad **6** samt skriver ut den på rad **8** i samma ordning som den initierades. På rad **9** skickas arrayen till funktionen **sort()** för sortering. Syntaxen för detta anrop visar att **sort()** inte är en vanlig funktion:


```
NoSort.sort(asc)
```

För det första har vi ingenstans i scriptet definierat en funktion vid namnet **sort()**. För det andra anropas den med punkt efter arrayen **NoSort** och för det tredje anropas

den med en parameter (**asc**) som vi inte har definierat som en variabel. Faktiskt är **sort()** en s.k. *metod* som är fördefinierad i JavaScript, närmare bestämt i *klassen* **Array**. Därför kallar vi den för en *Array-metod*. Den anropas med punktnotation, vilket i sin tur innebär att **NoSort** är ett *objekt* av JavaScript-klassen **Array**. Vi kommer att förklara dessa begrepp senare och nöjer oss med att närmare gå in på *metoder*.

Vad är en metod?

En *metod* är en funktion som är definierad i en *klass*. Dvs den enda skillnaden mellan metod och funktion är *placeringen* av definitionen. Så man skiljer mellan *funktioner* som är *fristående* och *metoder* som är placerade i klasser. Därför kan metoder endast anropas i ett *objekt* (ett exemplar) av klassen. Närmare bestämt kan metoder anropas endast genom att först kalla på objektet och sedan med hjälp av punktnotation anropa metoden, t.ex. **NoSort.sort()**. För att anropa **sort()** måste vi skriva så, därför att metoden **sort()** är definierad i klassen **Array** och **NoSort** är ett objekt av denna klass. Det går inte att anropa **sort()** utan objekt, som en fristående funktion.

Array-metoden sort()

En annan egenskap av den inbyggda metoden *(built-in method)* **sort()** är att den kan ta som parameter namnet på en s.k. *comparator function* som jämför sina två parametrar och returnerar:

- ett positivt värde om den första parametern är större än den andra
- noll om båda parametrarna är lika stora eller
- ett negativt värde om den första parametern är mindre än den andra.

I scriptet **Sort** har vi definierat två comparator functions: **asc()** på raderna **20-23** och **desc()** på raderna **25-28**. Båda anropas i parameterlistan av metoden **sort()** för att bestämma om sorteringen ska göras i stigande eller i fallande ordning.

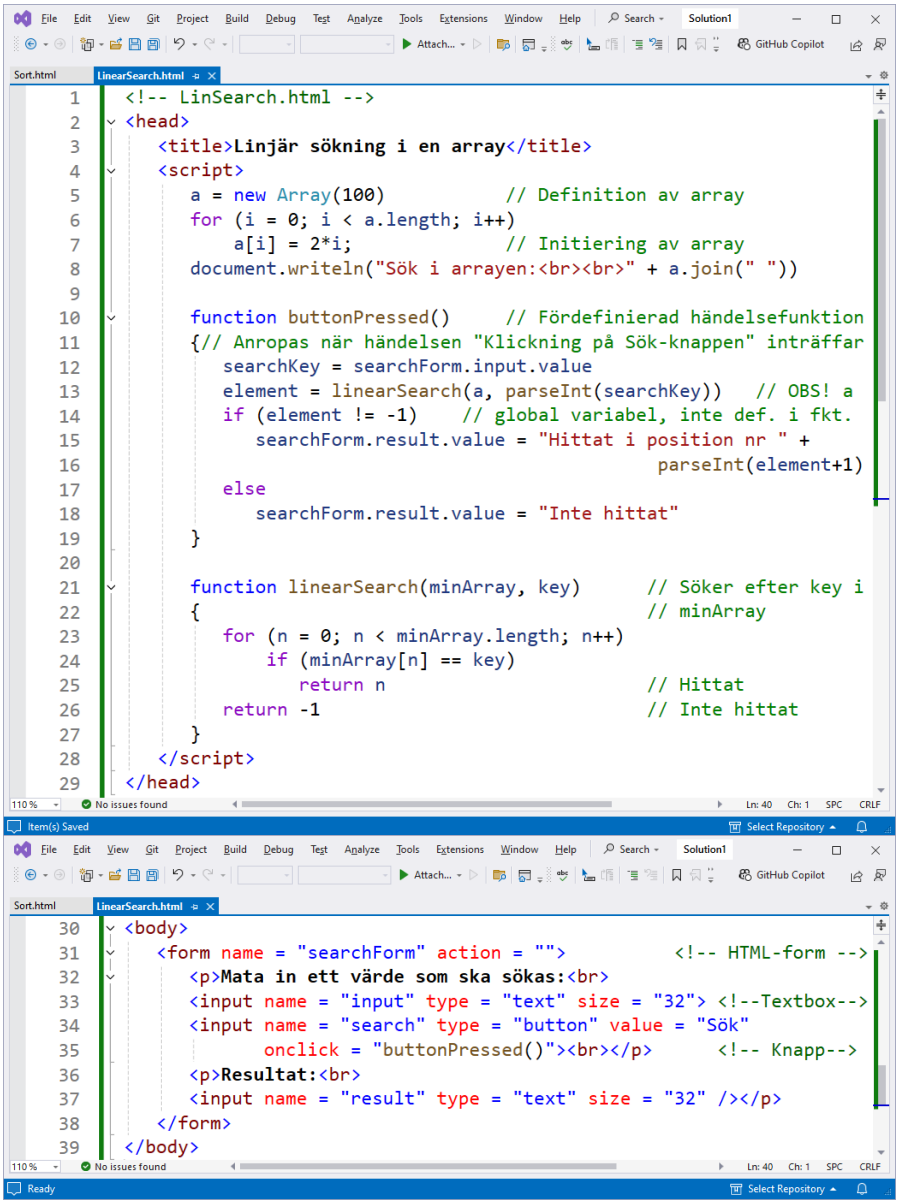
Det är anmärkningsvärt att anropen av comparator functions görs utan de sedvanliga parenteserna som är obligatoriska vid funktionsanrop. Anledningen är att funktioner anses i JavaScript som *data* som kan tilldelas variabler och skickas som parametrar till andra funktioner, precis som vilken annan data som helst. Eftersom anropet sker i parameterlistan av en *annan* funktion vore det fel att skriva parenteserna. Därför skriver vi i scriptet **Sort** som anrop av metoden **sort()**

på rad **9**: **NoSort.sort(asc)**

och på rad **11**: **NoSort.sort(desc)**

Dvs våra comparator functions **asc()** och **desc()** anropas i parameterlistan av metoden **sort()** *utan* parenteser och överför informationen om sorteringsordningen (stigande eller fallande) som en flagga till metoden. **asc** och **desc** anses här vara *variabler*. Skriver man parenteserna blir det fel syntax.

5.7 Sökning i arrays



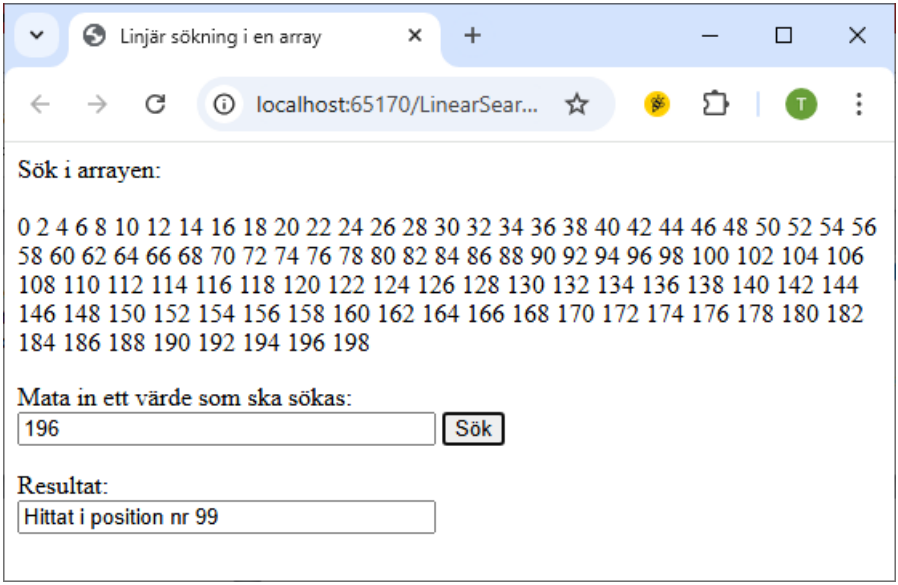
```
<!-- LinSearch.html -->
<head>
   <title>Linjär sökning i en array</title>
   <script>
      a = new Array(100)           // Definition av array
      for (i = 0; i < a.length; i++)
          a[i] = 2*i;              // Initiering av array
      document.writeln("Sök i arrayen:<br><br>" + a.join(" "))

      function buttonPressed()     // Fördefinierad händelsefunktion
      {// Anropas när händelsen "Klickning på Sök-knappen" inträffar
         searchKey = searchForm.input.value
         element = linearSearch(a, parseInt(searchKey))   // OBS! a
         if (element != -1)    // global variabel, inte def. i fkt.
            searchForm.result.value = "Hittat i position nr " +
                                                 parseInt(element+1)
         else
            searchForm.result.value = "Inte hittat"
      }

      function linearSearch(minArray, key)      // Söker efter key i
      {                                         // minArray
         for (n = 0; n < minArray.length; n++)
             if (minArray[n] == key)
                return n                        // Hittat
         return -1                              // Inte hittat
      }
   </script>
</head>
```

```
<body>
   <form name = "searchForm" action = "">          <!-- HTML-form -->
      <p>Mata in ett värde som ska sökas:<br>
      <input name = "input" type = "text" size = "32"> <!--Textbox-->
      <input name = "search" type = "button" value = "Sök"
             onclick = "buttonPressed()"><br></p>      <!-- Knapp-->
      <p>Resultat:<br>
      <input name = "result" type = "text" size = "32" /></p>
   </form>
</body>
```

Linjär sökning

I scriptet **`LinSearch`** ovan har vi implementerat en algoritm för sökning av data i en

array vars körresultat kan t.ex. se ut så här:



Vi letar efter värdet 196 i arrayen ovan som har 100 element. Svaret är att 196 finns på den 99:e positionen, där med ”position” menas platsen när man börjar räkna från 1 – så som vi brukar göra (inte från 0 som datorn brukar göra). Hade vi matat in värdet 0 hade vi fått position nr 1 osv. Detta har implementerats i koden på rad **16**. Matar man in ett värde som inte finns i arrayen får man svaret ”Inte hittat”.

Fördefinierad händelsefunktion

Intressant och ny för oss är en funktion som definieras på raderna **10-19** med huvudet:

```
function buttonPressed()
```

För det första är det inte vi som definierar funktionen utan JavaScript som redan har definierat den. Men vi *definierar om* den. Dvs vi bibehåller *namnet* som JavaScript har föreskrivit, men skriver *kroppen* själva. Det får vi göra eftersom JavaScript tillåter det – ett programmeringstekniskt koncept som används i många språk och kallas för *överlagring* (eng. *Overloading*). Vi överlagrar funktionen `buttonPressed()` genom att definiera en egen variant av den. I så fall får vi inte ändra namnet på den, inte ens vad gäller stora/små bokstäver. Överlagring är ett ofta använt koncept i programmeringen.

För det andra är funktionen `buttonPressed()` inte en vanlig funktion, dvs den anropas inte genom att kalla på dess namn, utan genom att initiera en händelse. Med ”händelse” menas en musklickning, en tangenttryckning eller en systemåtgärd. I det här fallet anropas funktionen genom att vi vid körning klickar på en knapp. Det är

anledningen varför sådana funktioner kallas för *händelsefunktioner*. Närmare bestämt anropas funktionen `buttonPressed()` i scriptet `LinSearch` när vi klickar på knappen Sök. Global variabel som parameter

Global variabel som parameter

En annan programmeringsteknisk nyhet är att variabeln `a` används i funktionen `buttonPressed()`, men inte är definierad som en lokal variabel i funktionen. Detta sker i anropet av funktionen `linearSearch()` på rad **13**, där `a` är den första parametern:

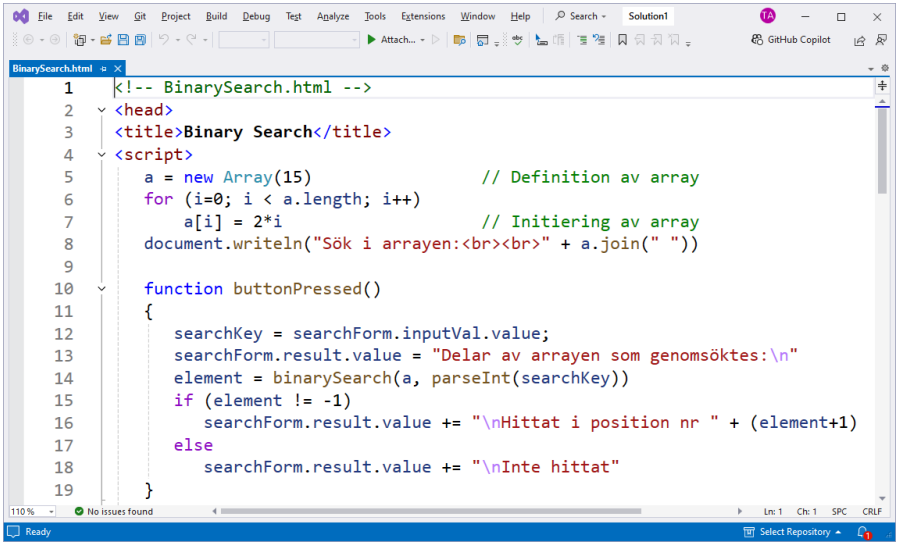
```
element = linearSearch(a, parseInt(searchKey))
```

`a` är i själva verket en *global* variabel, därför att den är definierad på rad **5** dvs *utanför* funktionen `buttonPressed()` och dessutom utanför *alla* funktioner. Detta är möjligt eftersom globala variabler är giltiga i hela programmet och därmed även i alla funktioner som definieras där. `a` är ju arrayen som vi initierar med värden på raderna **6-7** och skickar sedan till funktionen `linearSearch()` för sökning.

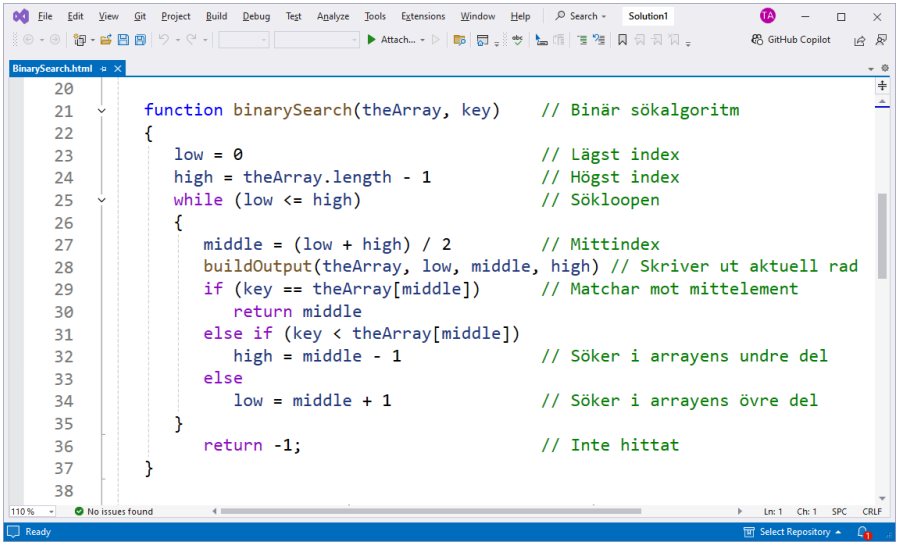
Det ”vanliga” sättet att överföra arrayen `a` till funktionen `buttonPressed()` hade varit att definiera `buttonPressed()` med parametern `a`, typ `buttonPressed(a)`, och sedan använda den i kroppen. Men vi får inte ändra namnet på funktionen `buttonPressed()` eftersom den är fördefinierad i JavaScript och vi får endast definiera om kroppen, som vi sade tidigare. `buttonPressed(a)` fungerar faktiskt inte i koden.

Det verkar som om användningen av `a` som global variabel är den enklaste lösningen i det här fallet. Annars rekommenderas generellt en restriktiv policy för användningen av globala variabler eftersom de motverkar modulariseringen. Funktioner med globala variabler som prametrar kan inte användas som fristående moduler i andra program. Löser man dem från sitt sammanhang förlorar den globala variabeln sin giltighet.

5.8 Binär sökning

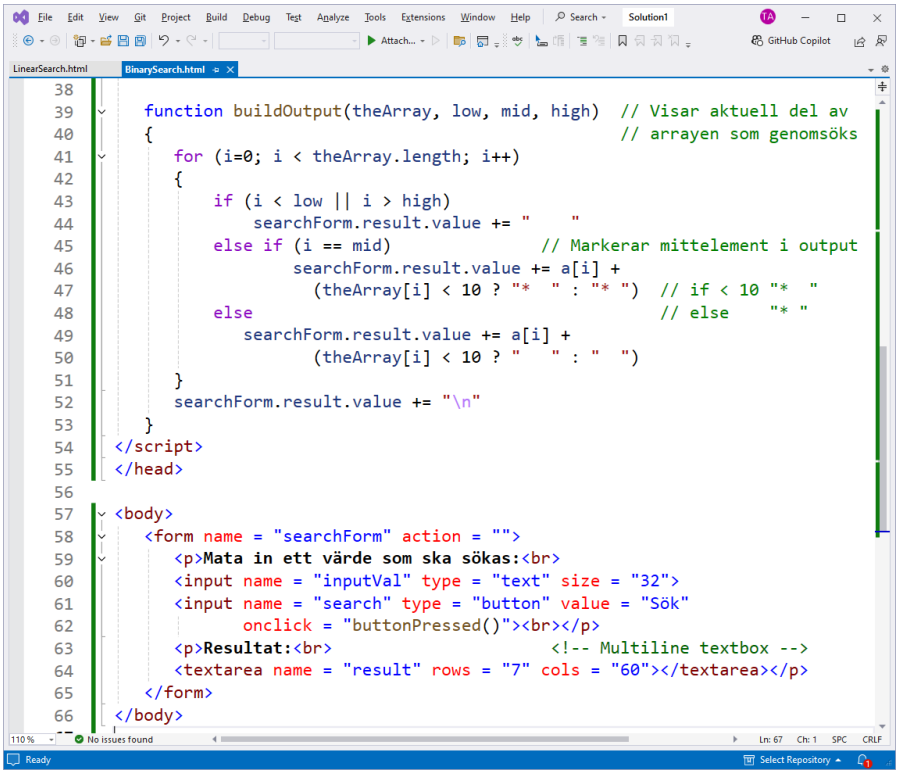


```
1  <!-- BinarySearch.html -->
2  <head>
3  <title>Binary Search</title>
4  <script>
5     a = new Array(15)                  // Definition av array
6     for (i=0; i < a.length; i++)
7         a[i] = 2*i                     // Initiering av array
8     document.writeln("Sök i arrayen:<br><br>" + a.join(" "))
9
10    function buttonPressed()
11    {
12       searchKey = searchForm.inputVal.value;
13       searchForm.result.value = "Delar av arrayen som genomsöktes:\n"
14       element = binarySearch(a, parseInt(searchKey))
15       if (element != -1)
16          searchForm.result.value += "\nHittat i position nr " + (element+1)
17       else
18          searchForm.result.value += "\nInte hittat"
19    }
```

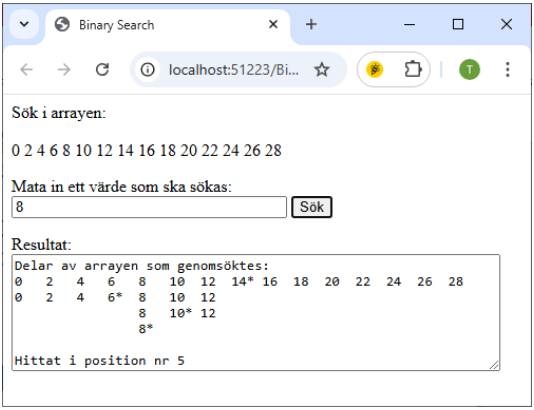


```
20
21    function binarySearch(theArray, key)    // Binär sökalgoritm
22    {
23       low = 0                              // Lägst index
24       high = theArray.length - 1           // Högst index
25       while (low <= high)                  // Sökloopen
26       {
27          middle = (low + high) / 2         // Mittindex
28          buildOutput(theArray, low, middle, high) // Skriver ut aktuell rad
29          if (key == theArray[middle])      // Matchar mot mittelement
30             return middle
31          else if (key < theArray[middle])
32             high = middle - 1              // Söker i arrayens undre del
33          else
34             low = middle + 1               // Söker i arrayens övre del
35       }
36          return -1;                        // Inte hittat
37    }
38
```

Koden fortsätter på nästa sida. Följ radnumren.



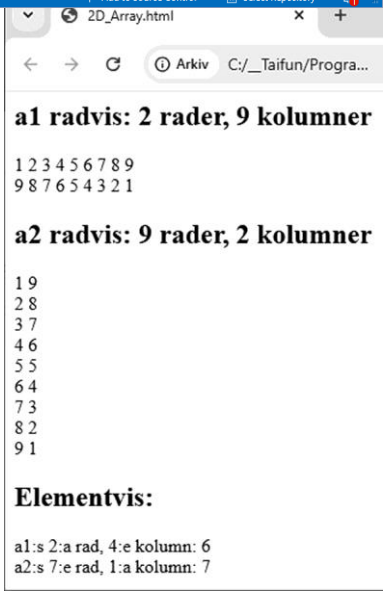
Algoritmen



Algoritmen för binär sökning är implementerad i funktionen `binarySearch()` i scriptet **`BinarySearch`** på sid 112, raderna **`21-37`**. Grundtanken liknar den i intervallhalveringsalgoritmen. Efter varje matchning (rad **`29`**) eliminerar algoritmen halva arrayen: Arrayens mitt lokaliseras och jämförs med det sökta elementet *(key)*. Om de är lika har man hittat *key* vars position (index + 1) returneras. Annars reduceras sökandet till hälften av arrayen. Om *key* är mindre än arrayens mittelement sökes endast i den första halvan av arrayen, annars i den andra halvan. Om *key* inte är lika med den aktuella delarrayens mittelement, upprepas algoritmen i en kvart av arrayen osv. Sökandet fortsätter tills *key* är lika med en delarrays mittelement eller en delarray består endast av ETT element som inte är lika med *key*. I så fall har *key* inte hittats.

5.9 2D arrays

```
1   <!-- 2D_Arrays.html -->
2   <head>
3   <title>2D arrays</title>
4   <script>
5      function start()
6      {
7         a1 = [ [1, 2, 3, 4, 5, 6, 7, 8, 9],        // En (2 x 9)-array
8                [9, 8, 7, 6, 5, 4, 3, 2, 1] ]
9         a2 = [ [1, 9], [2, 8], [3, 7], [4, 6],     // En (9 x 2)-array
10               [5, 5], [6, 4], [7, 3], [8, 2], [9, 1] ]
11        outputArray("a1 radvis: 2 rader, 9 kolumner", a1)
12        outputArray("a2 radvis: 9 rader, 2 kolumner", a2)
13        document.writeln("<h2>Elementvis:</h2>" +
14                         "a1:s 2:a rad, 4:e kolumn: " + a1[1][3] + "<br>" +
15                         "a2:s 7:e rad, 1:a kolumn: " + a2[6][0])
16     }
17     function outputArray(header, a)
18     {
19        document.writeln("<h2>" + header + "</h2>")
20        for (row in a)                              // Nästlad foreach-sats
21        {                                           // Skriver ut rader
22           for (col in a[row])                      // Skriver ut kolumner
23              document.write(a[row][col] + " ")
24           document.writeln("<br>")
25        }
26     }
27  </script>
28  </head>
29  <body onload="start()"></body>
```



2D Array = Array av arrays

2D arrays med två index används ofta för att representera tvådimensionella strukturer som tabeller eller matriser, dvs information som är ordnad i form av *rader* och *kolumner*. För att identifiera ett värde i tabellen måste vi ange rad- och kolumnnumret (indexet). Det är allmän konvention att först ange rad- och sedan kolumnindexet. Det finns även flerdimensionella arrays med flera index som vi däremot inte tar upp här.

JavaScript (och även andra programmeringsspråk) beskriver en 2D array som en enkel array av flera enkla arrays. Dvs vi ersätter en enkel arrays element med andra enkla element. I scriptet **`2D_Arrays`** ovan har vi på rad **7** och **8** skapat en 2D array `a1` bestående av 2

”element” som i sin tur är arrays bestående av 9 element. Därför kallar vi `a1` för en (2 x 9)-array. I den matematiska disciplinen *Linjär algebra* har den här konstruktionen beteckningen *(2 x 9)-matris*, medan vanliga, dvs endimensionella arrays, kallas för *vektorer* som består av endast EN rad resp. kolumn. 2D arrayen `a1` består av 2 rader och 9 kolumner, vilket vi har försökt att påminna om i scriptets körresultat ovan.

På raderna **`9-10`** skapas en annan 2D array `a2` bestående av 9 ”element” som är arrays bestående av 2 element. Därför är `a2` en (9 x 2)-array. Även dess utskrift har vi i körresultat utformat som en tabell med 9 rader och 2 kolumner. Att vi kallar dessa två framställningar (`a1` och `a2`) för ”radvis” har att göra med koden på raderna **`20-25`** som skriver ut dem och med konventionen som nämndes, nämligen att först ange rad- och sedan kolumnindexet.

I slutet av körresultatet ser vi under rubriken ”Elementvis:” utskriften av två enskilda element. Detta för att visa med vilken JavaScript-kod man tar fram enskilda värden ur en 2D array. Och då menar vi med *element* verkligen enskilda värden, inte subarrays. I scriptet **`2D_Arrays`** kan man se detta på rad **`14`** i koden:

```
a1[1][3]
```

där man kommer åt arrayen `a1`:s element som finns på rad `2`, kolumn `4` (motsvarande indexen `1` och `3`) som är talet 6 och på rad **`15`** i koden:

```
a2[6][0]
```

där man kommer åt arrayen `a2`:s element som finns på rad `7`, kolumn `1` (motsvarande indexen `6` och `0`) som är talet 7. Generellt refererar koden:

```
a[i][j]
```

till arrayen `a`:s element som finns på rad `i+1`, kolumn `j+1` (motsvarande indexen `i` och `j`).

Övningar till Kapitel 5

5.1 Modifiera scriptet **InitArray** (sid 95) så att funktionen **InitializeArrays()** skapar ytterligare en tredje array med 10 element. Initiera denna array med värden som är multipla av 100. Skriv ut den i tabellform under de två första arrays. Visa i slutet även några odefinierade element av den nya arrayen.

5.2 I scriptet **InitArray_2** (sid 98) finns dett två funktioner som tar hand om initiering och utskrift av arrays. Undersök om det är bra att slå ihop dessa funktioner till en funktion, genom att testa det i ett nytt script. Det nya programmet ska generera samma utskrift som det gamla.

5.3 Modifiera scriptet **InitArray_2** (sid 98) genom att ersätta den vanliga **for**-satsen på raderna **22-24** med en **Foreach**-sats enligt modellen i scriptet **Foreach** (sid 100), raderna **13-14**. Skriv endast ut arrayvärdena. Strunta i att skriva ut indexen.

5.4 Lägg till i utskriftstabellen av scriptet **DiceArray** (sid 102) en tredje kolumn som visar den experimentella sannolikheten av varje utfall. Utöka i scriptet antalet tärningskast som för närvarande är 6 000, till 60 000, 600 000 osv. Observera och förklara vad som händer med de experimentella sannolikheterna i den tredje kolumnen.

Kan man ersätta **for**-satserna i scriptet **DiceArray** med **Foreach**-satser (sid 100)? Besvara frågan både teoretiskt och experimentellt. Förklara ditt svar.

5.5 Ta scriptet **InitArray_2** (sid 98) som utgångspunkt. Lägg till en funktion `modifyHeltal1()` som kastar om ordningen av elementen i arrayen `Heltal1` genom att sortera arrayen baklänges. Skriv ut arrayen före och efter anropet av funktionen, för att visa effekten av referensanrop. Gör samma sak med de andra två arrayerna `Heltal2` och `Färger`.

5.6 I scriptet **InitArray_2** (sid 98) skapas och initieras tre arrays. Kasta om först deras ordning. Använd sedan Array-metoden `sort()` för att sortera dem både i stigande och i fallande ordning. Skriv ut de sorterade arrayerna. I scriptet **Sort** (sid 107) demonstreras hur man gör det för en heltalsarray. För att sortera arrayen `Färger` vars element är strängar måste du anropa metoden `sort()` *utan* parameter.

5.7 I scriptet **LinSearch** (sid 109) skapas och initieras arrayen **a** som en *global* variabel som sedan används i funktionen `buttonPressed()` för att skicka den som parameter till anropet av funktionen `linearSearch()` på rad **13**.

Modifiera scriptet genom att flytta definitionen och initieringen av arrayen **a** till funktionen **buttonPressed()**, så att **a** blir en *lokal* variabel där.

Genomför alla ändringar i koden som blir nödvändiga efter denna ändring, så att det nya scriptet fungerar på samma sätt som det gamla.

Besvara följande frågor: Är denna ändring kodmässigt genomförbar? Om ja, är det nya upplägget bättre än det gamla?

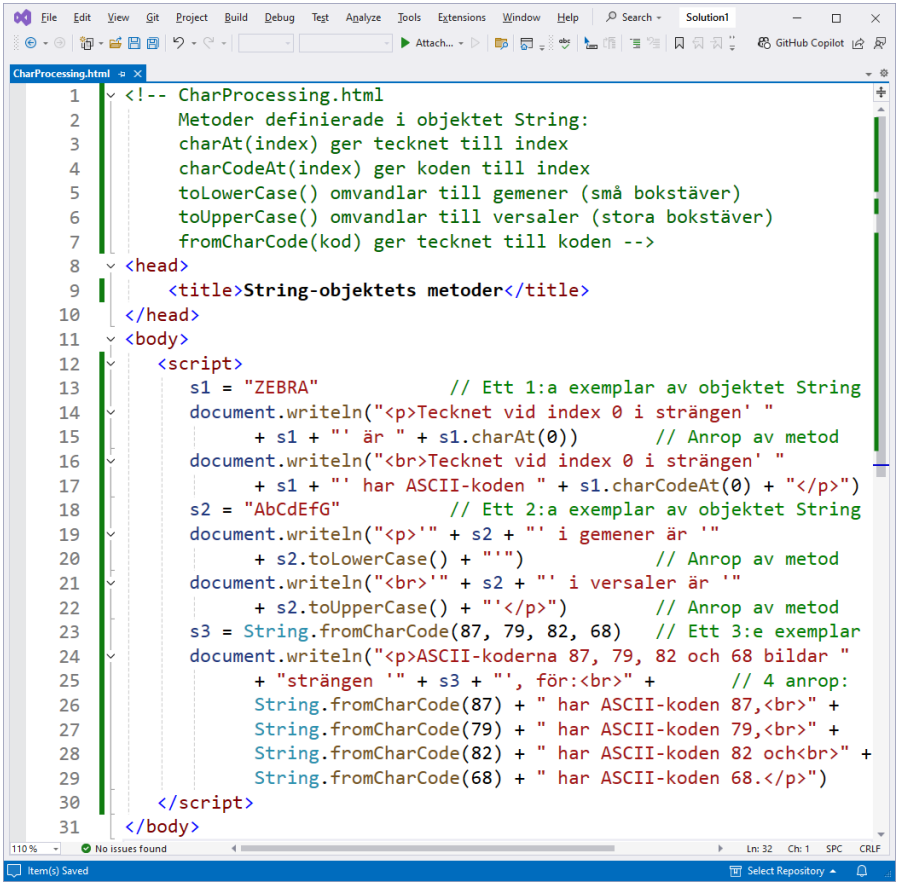
5.8 Modifiera scriptet **BinarySearch** (sid 112) genom att utöka arrayens storlek till 60. Ersätt dessutom koderna på raderna **47** och **50** med vanliga **if-else**-satser. Fundera på om det vore bättre att ersätta de **if-else**-stegen i funktionerna **binarySearch()** i och **buildOutput()** med andra konstruktioner. Testa dina idéer.

5.9 Scriptet **2D_Arrays** (sid 114) ger en inblick hur man i JavaScript kan hantera 2D arrays. Skriv ett script som med hjälp av en 2D array skriver ut multiplikationstabellen. Dvs skapa en (9 x 9)-array och initiera den med värden, så att utskriften ger den lilla multiplikationstabellen (1-9).

Kapitel 6

Objekt

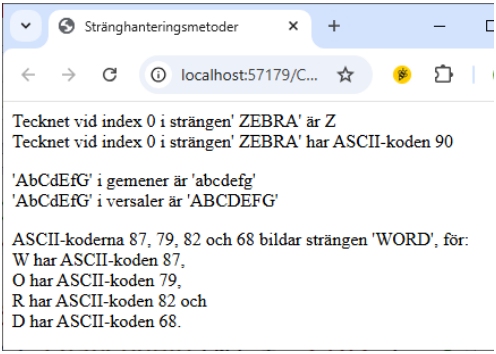
6.1 Stränghantering med String-objekt



Objektet `String`

`String` är ett *fördefinierat* objekt i JavaScript. I scriptet **`CharProcessing`** (sid 119) skapar vi några exemplar av objektet **`String`** och anropar metoder som är definierade i dem. På rad **11** skapas ett första exemplar:

```
s1 = "ZEBRA"
```



Detta verkar vara en vanlig deklaration och initiering av variabeln **`s1`**. Det stämmer också – med tillägget att variabeln är ett exemplar av *objektet* **`String`**, därför att citationstecknen **`" "`** gör

ZEBRA till en **String**. Och **String** är ett objekt som är definierat i JavaScript. Vi använder det bara i scriptet **CharProcessing** för att dra nytta av dess metoder. Det gör vi för första gången på rad **13**:

```
s1.charAt(0)
```

Dvs vi anropar metoden **charAt()** och skickar parametern **0**, för att få den första bokstaven (index **0**) till variabeln **s1** som i satsen innan initierats till strängen **ZEBRA** (rad **11**). Därför returnerar **charAt()** den första bokstaven **Z** som skrivs ut sedan, se körresultatet på sid 119.

Samma sak händer med de övriga anropen av metoderna **charCodeAt()** på rad **15**, **toLowerCase()** på rad **18**, **toUpperCase()** på rad **20** och **fromCharCode()** på raderna **24-27**. Vad de gör framgår av koden samt kommentarerna, se scriptet **CharProcessing** samt körresultatet på sid 119.

Objektbaserad programmering

JavaScript är ett *objektbaserad programmeringsspråk.* Vad betyder det? Man kan säga att det är en Light-variant av de objekt*orienterade* programmeringsspråken (OOP), så som vi känner till det från C++ eller från andra objektorieneterade språk. Anledningen är att JavaScript är ett interpreterande (inte kompilerande) språk som används på webben. Exekveringsmiljön är webbläsaren som måste ge respons i realtid.

En given definition på programmering är problemlösning med hjälp av datorn. Om man då beskriver problemets lösning i form av en *algoritm* kan man kort säga:

Program = algoritm + data

Denna definition ställdes upp av Niklaus Wirth på 60-talet och återspeglar den procedurala synen på programmering. Fokuset ligger på *algoritmen* dvs att inte bara hitta utan även *beskriva* tillvägagångssättet (proceduren) för att lösa ett problem. Sedan återstår bara att koda denna beskrivning. En annan definition som kom upp på 80-talet och återspeglar den objektbaserade synen på programmering är:

Program = Modell av verkligheten

Om man i Wirths formel *Program = algoritm + data* lägger vikten på data istället för på algoritmen och inte längre betraktar data som ett slags bihang till algoritmen utan som *objekt*, kommer man till *objektbaserad programmering*.

Vad är ett objekt i JavaScript?

Objektbaserad programmering syftar åt att efterlikna verkligheten. Man vill avbilda den reala världen – åtminstone den del som tillåter datorisering – och konstruera en modell av den i sina datorprogram för att kunna simulera verkligheten genom att tes-

ta modellen. För att undvika filosofiska diskussioner kan vi anta att den reala världen består kort sagt av *objekt*. Världen kring oss är full med sådana objekt: Människor, byggnader, bilar, tåg, flygplan, träd, möbler, böcker, butiker, skolor, bibliotek, kontor, anställda, kunder, varor, fakturor, order, bokningar, kurser osv. Objekten kan vara verkliga eller virtuella. Ett datorprogram försöker att beskriva dessa objekt.

Ett *objekt*, t.ex. en bil, har vissa egenskaper. Man kan t.o.m. säga att bilen är summan av alla sina egenskaper. Ett annat ord för egenskap är *attribut*. Summan av alla attribut utgör objektet. Bilen har som attribut: fabrikat, modell, färg, årsmodell, antal körda mil, antal hästkrafter, maximala hastigheten, antal och storlek på cylindrar i motorn osv. Alla dessa data utgör objektet bil och ger svar på frågan ”Vad är det för bil?”. Alla bilar har sådana attribut. Därför abstraherar man – dvs bortser från bilarnas olikheter – och samlar bilarnas gemensamma egenskaper (attribut) i något som man kallar för *objektet Bil*. När man programmerar, deklarerar man objektet *Bil* och skriver upp alla dessa bilattribut som objektets *egenskaper*. Vi sammanfattar:

> I JavaScript är objekt behållare för logiskt sammanhängande variabler och funktioner. Variablerna bildar objektets *attribut* (*egenskaper*) och funktionerna objektets *metoder*. Ett JavaScript-objekt organiserar data, kapslar in dem och ger ett gränssnitt till sina medlemmar (både attribut och metoder) via sitt namn.

Metoder

Bilden vore ofullständig om vi nöjde oss med objektets intressanta, men statiska egenskaper (attribut). Vi vill också veta vad man kan *göra* med dem. Ett objekt med alla sina attribut kan i regel även utföra vissa aktioner eller operationer. I den objektbaserade programmeringens terminologi kallas dessa aktioner för *metoder*. Typiska metoder för t.ex. en bil är att köra fram, att backa, att accelerera, att bromsa, att parkera, att byta olja osv. Den fullständiga definitionen på en bil vore alltså att ange *både* dess attribut *och* metoder. Bilfabrikanten måste förse bilen med alla dessa färdigheter för att kunna sälja den som en bil. Därför går man i bilfabriken efter en plan när man tillverkar bilen. Denna plan för konstruktion av bilen är *objektet Bil*. Konstruktörerna, mest ingenjörer, måste skapa denna plan, innan bilen kan byggas. När vi skriver ett program måste vi först definiera *objektet Bil* för att sedan kunna anropa objektets metoder. I definitionen måste vi ta upp alla attribut och metoder som är relevanta eller av någon anledning önskvärda för en bil.

En *metod* är en funktionalitet som definieras i ett objekt. Den talar om vad ett objekt kan *göra*. Det finns två steg i hantering av metoder: Först definierar man dem dvs skapar man deras kod i ett objekt. Sedan *anropar* dvs aktiverar man dem i exemplar av detta objekt. Ofta är det första steget redan genomfört av andra, så vi behöver bara anropa en redan *fördefinierad* metod. I objektet *Bil* t.ex. är metoderna att köra fram, att backa, att accelerera, att bromsa osv. definierade i huvuden på

bilkonstruktörerna och i deras konstruktionsritningar och dokumentationer. Sedan har man tillverkat massor med exemplar av objektet Bil i fabriken och byggt in dessa metoder i alla bilar. Vi behöver bara anropa dem i den bil vi kör. Den bil vi kör är ett specifikt exemplar av objektet Bil. Låt oss kalla det för **`minVolvo`**. Exemplaret **`minVolvo`** har ett antal attribut som t.ex. fabrikat, modell, färg, årsmodell osv., men också ett antal metoder, bl.a. metoden **`kör()`**. Parenteserna i metodens namn brukar man skriva för att karakterisera **`kör()`** som en *metod*. Man skriver ett anrop av metoden **`kör()`** så här:

```
minVolvo.kör()
```

Observera att *före* punkten står ett exemplar av objektet. Det är ju den specifika bil som jag använder just nu som ska köras. Först *efter* punkten står själva anropet av metoden **`kör()`**. Det här sättet att skriva kallas för *punktnotation*. Metoder måste alltid anropas med punktnotation, vilket har sin grund i att de endast är deklarerade i objektet. Till skillnad från fristående *funktioner* kan metoder varken definieras eller anropas utanför objekt. I JavaScript finns både metoder och funktioner.

En annan variant av metoden **`kör()`** kan anropas på följande sätt:

```
minVolvo.kör(40)
```

Det kan t.ex. betyda: Kör bilen med hastigheten **`40`** km/h. Värdet **`40`** kallas då en *parameter* som skickas till metoden när den anropas. I så fall måste även metoden **`kör()`** vara definierad så att den har beredskapen att ta emot denna parameter. Så det kan inte vara samma metod som anropades *utan* parameter. Det måste vara en annan variant av den, exakt talat en annan metod med samma namn. Konceptet kallas *överlagring av metoder* och innebär två eller flera metoder med samma namn, men olika parametrar.

I slutet vill vi dra parallellen till det vi lärt oss tidigare, nämligen *funktioner*. Vad är egentligen skillnaden mellan metoder och funktioner? Metoder är inget annat än funktioner vars definition placerats i ett objekt. Därför kan vi anropa metoder endast i exemplar av sådana objekt som innehåller metodens definition. Vi ska nu titta på några exempel på anrop av metoder som är fördefinierade i JavaScript-objektet **`String`** och som vi använt i vårt script **`CharProcessing`**.

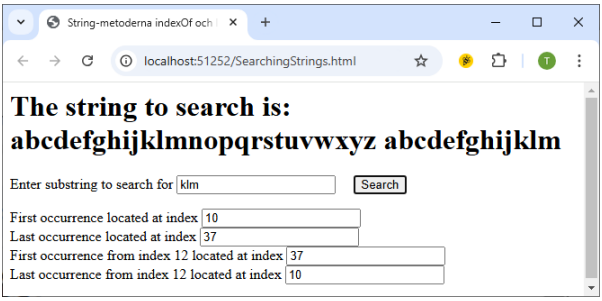
6.2 Strängsökningsmetoder

Med *Strängsökningsmetoder* menas sökning i strängar efter tecken eller delsträngar. Två strängsökningsmetoder definierade i JavaScript-objektet **`String`** demonstreras i följande script, **`indexOf()`** och **`lastIndexOf()`**:

```
<!-- SearchingStrings.html -->
<head>
   <title>String-metoderna indexOf och lastIndexOf</title>
   <script>                                  // String-objekt:
      letters = "abcdefghijklmnopqrstuvwxyz abcdefghijklm"
      function buttonPressed()
      {
         searchForm.first.value =          // String-metoder:
            letters.indexOf(searchForm.inputVal.value)
         searchForm.last.value =
            letters.lastIndexOf(searchForm.inputVal.value)
         searchForm.first12.value =
            letters.indexOf(searchForm.inputVal.value, 12)
         searchForm.last12.value =
            letters.lastIndexOf(searchForm.inputVal.value, 12)
      }
   </script>
</head>
<body>
   <form name = "searchForm">
   <h1>The string to search is:<br>
      abcdefghijklmnopqrstuvwxyz abcdefghijklm</h1>
   <p> Enter substring to search for            <!-- Textbox 1 -->
      <input name="inputVal" type="text">     
      <input name="search" type="button" value="Search"
         onclick="buttonPressed()"><br></p>    <!-- Sök-knapp -->
   <p>First occurrence located at index         <!-- Textbox 2 -->
      <input name = "first" type = "text"><br>
      Last occurrence located at index          <!-- Textbox 3 -->
      <input name = "last" type = "text"><br>
      First occurrence from index 12 located at index
      <input name = "first12" type = "text"><br> <!-- Textb 4 -->
      Last occurrence from index 12 located at index
      <input name = "last12" type = "text"></p>  <!-- Textb 5 -->
   </form>
</body>
```

Ett körresultat med sökning efter delsträngen **`klm`** ger följande utskrift:

Metoden `indexOf()`



På rad **9** i scriptet **SearchingStrings** (förra sidan) anropas första gången metoden **`indexOf()`**. När vi matar in delsträngen **`klm`** returnerar metoden värdet **`10`**, därför att den första förekomsten av delsträngens första bokstav **`k`** befinner sig vid index **`10`** av strängen **`letters`**. Dvs metoden **`indexOf()`** returnerar indexet av den första förekomsten av strängen **`searchForm.inputVal.value`**, dvs den sträng användaren matar in i den första textboxen **`inputVal`**, i körexemplet ovan **`klm`**. Om delsträngen hittas returneras indexet, annars returneras **`-1`**.

Nästa anrop av metoden **`indexOf()`** finns på rad **13**. Men är det samma metod som anropas som på rad **9**? Anropet på rad **13** sker med *två* parametrar:

```
letters.indexOf(searchForm.inputVal.value, 12)
```

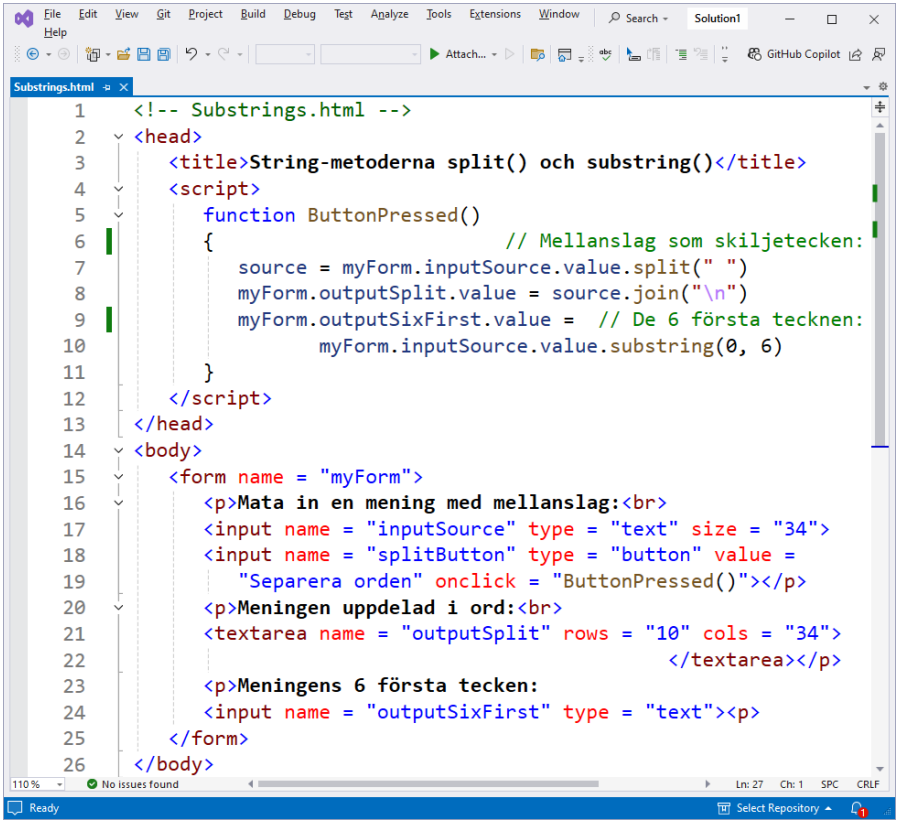
Medan det första anropet på rad **9** hade bara *en* parameter. Dvs det är inte samma metod utan en överlagrad variant av den. Den överlagrade metoden **`indexOf()`** med två parametrar returnerar indexet av den första förekomsten av delsträngen användaren matar in i textboxen **`inputVal`**, genom att börja räkna indexen från index **`12`** i strängen **`letters`**. Om delsträngen hittas returneras indexet av, med start från index **`12`**, annars returneras **`-1`**.

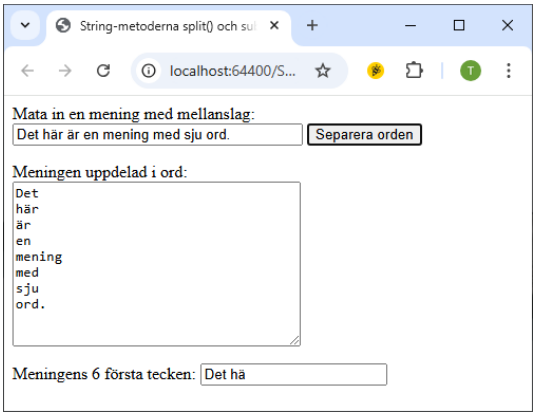
Metoden `lastIndexOf()`

På rad **11** i scriptet **SearchingStrings** (sid 123) anropas första gången metoden **`lastIndexOf()`**. När vi matar in delsträngen **`klm`** returnerar metoden värdet **`37`**, därför att den sista förekomsten av delsträngens första bokstav **`k`** befinner sig vid index **`37`** av strängen **`letters`**. Dvs metoden **`lastIndexOf()`** returnerar indexet av den sista förekomsten av den sträng användaren matar in i den första textboxen **`inputVal`**, i körexemplet ovan **`klm`**. Om delsträngen hittas returneras indexet, annars returneras **`-1`**.

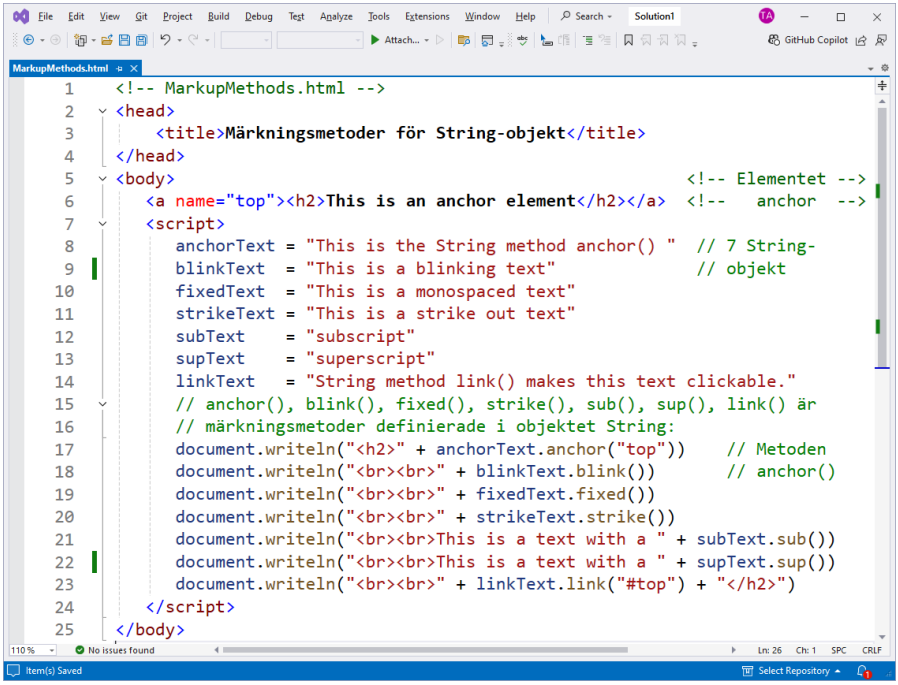
Vad gäller den överlagrade varianten av **`lastIndexOf()`** med *två* parametrar vars anrop vi ser på rad **15**, kan vi hänvisa till det som sades för metoden **`indexOf()`** ovan. Samma sak gällaer här: den överlagrade varianten börjar räkna från index **`12`** uppåt i strängen **`letters`**.

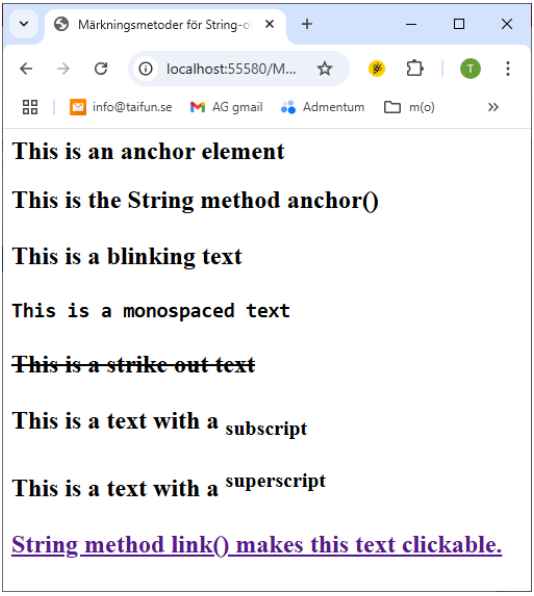
6.3 Delsträngar



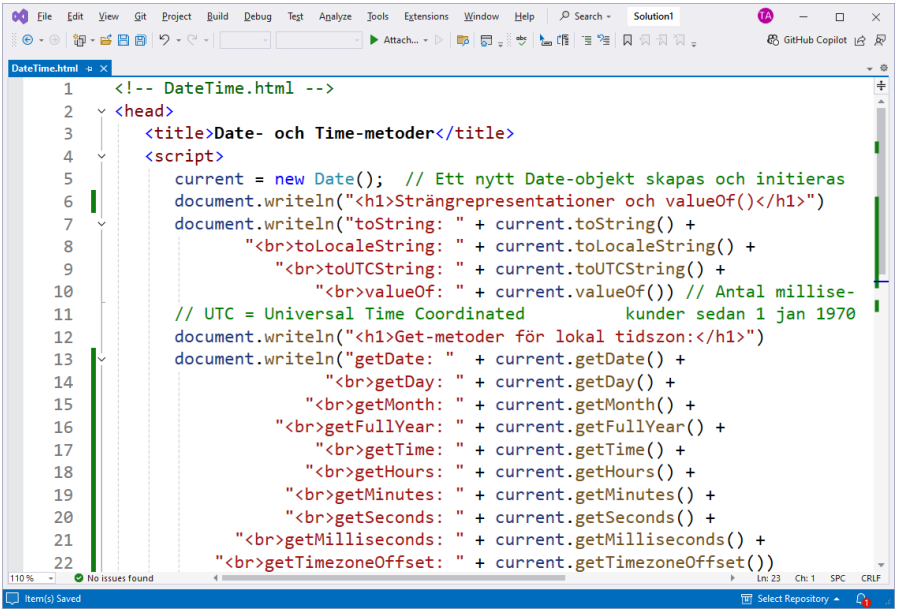


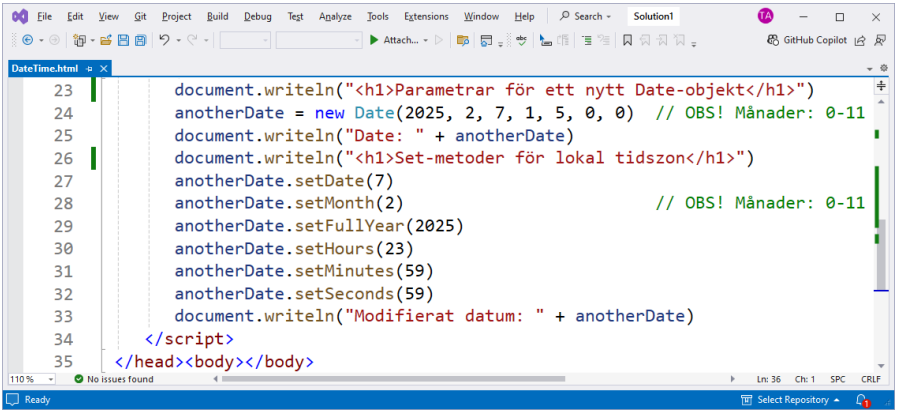
6.4 Märkningsmetoder för String-objekt

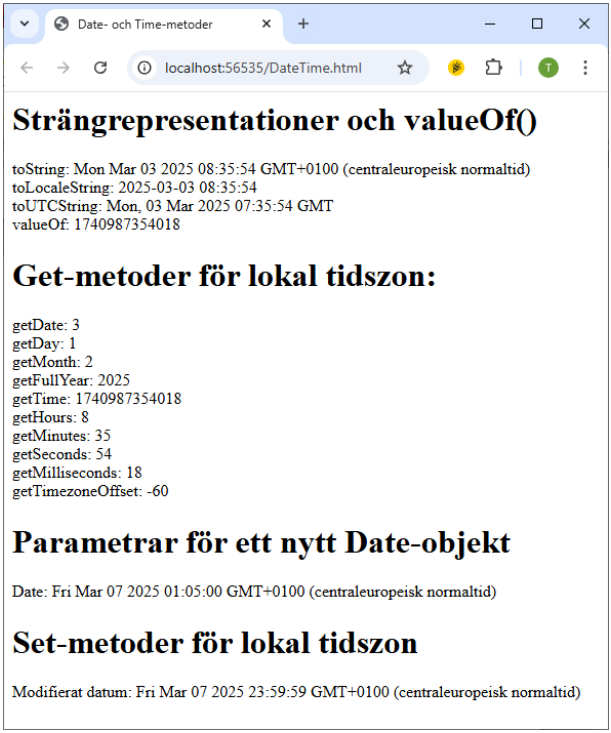




6.5 Objektet Date






Date- och Time-metoder
localhost:56535/DateTime.html
Strängrepresentationer och valueOf()
toString: Mon Mar 03 2025 08:35:54 GMT+0100 (centraleuropeisk normaltid)
toLocaleString: 2025-03-03 08:35:54
toUTCString: Mon, 03 Mar 2025 07:35:54 GMT
valueOf: 1740987354018
Get-metoder för lokal tidszon:
getDate: 3
getDay: 1
getMonth: 2
getFullYear: 2025
getTime: 1740987354018
getHours: 8
getMinutes: 35
getSeconds: 54
getMilliseconds: 18
getTimezoneOffset: -60
Parametrar för ett nytt Date-objekt
Date: Fri Mar 07 2025 01:05:00 GMT+0100 (centraleuropeisk normaltid)
Set-metoder för lokal tidszon
Modifierat datum: Fri Mar 07 2025 23:59:59 GMT+0100 (centraleuropeisk normaltid)

Övningar till Kapitel 6

6.1 Modifiera scriptet **CharProcessing** (sid 119) genom att slå ihop alla **document.writeln()**-satser till en enda. Gör så här: Samla först alla objekt i början av scriptet. Skapa sedan ett 4:e exemplar av objektet **String** som innehåller allt som ska skrivas ut. Dumpa detta exemplar till utskrift genom att placera det i **document.writeln()**-satsen i slutet av scriptet.

Kommentera din lösning så att kommentarerna hjälper att förstå koden utan att störa kodens läslighet. Går det inte, samla dina kommentarer i en extern textfil som en dokumentation till scriptet, som t.ex. **readme.txt**.

6.2 I scriptet **SearchingStrings** (sid 123) visas värdet **-1** när **String**-metoderna **indexOf()** och **lastIndexOf()** inte hittar den sökta delsträngen i strängen **letters**. Detta är inte särskilt användarvänligt. Vidareutveckla scriptet genom att istället skriva ut ett meddelande i textform av typ ”Inte hittat”, när sökmetoderna returnerar **-1**. Efterlikna scriptet **LinSearch** (sid 109), där vi redan hade implementerat detta på rad **18**. Ta även för strängen **letters** den omvända ordningen till alfabetet.

6.3 I scriptet **SubStrings** (sid 125) används i inmatningssträngen **source** *mellanslaget* som skiljetecken mellan ord när strängen ska delas upp i ord. Ersätt skiljetecknet med *@*. Glöm inte att modifiera ledtexten som instruerar användaren. Testa om ”skiljetecknet” i metoden **split()** kan även bestå av *flera* tecken, dvs om det i själva verket kan vara en *skiljesträng*.

Testa vad som händer om man inte skickar någon parameter till **String**-metoden **join()**?

Minska bredden och öka höjden till textarean **outputSplit**. Fundera samt testa vilken bredd/höjd som är rimliga. Visa i textboxen **outputSixFirst** de 6 *sista* tecknen i strängen **source**. Ändra textboxens namn till **outputSixLast**. Flytta knappen **splitButton** till efter textboxen **inputSource**.

6.4 I scriptet **MarkupMethods** (sid 126) är en av texterna klickbar. Ta reda på vart denna text länkar till. Skriv om scriptet så att scriptets alla texter blir klickbara. Låt dem länkas till en plats i slutet av scriptet som du kallar för *Target*. Den i sin tur behöver inte vara klickbar.

Alternativet till länkbara texter som kodas med **String**-märkningsmetoder, är HTML-elementet ankare. Skriv om ditt script genom att ersätta alla klickbara texter med detta alternativ.

Om du jämför båda metoderna ovan, vilken av dem föredrar du?

6.5 Skriv ett script som med hjälp av objektet **Date** (sid 127) hämtar datorns tid. Avgör med **Date**-objektets metoder om det är dags för dagens lunch om tiden ligger mellan kl 11 och 14. Skriv i så fall ut ett meddelande om att dagens lunch serveras. Skriv ut att det är för tidigt för dagens lunch om tiden är före kl 11 och att det är för sent för dagens lunch om tiden ligger efter kl 14.

6.6 I scriptet **DateTime** (sid 127) sätts en ny tid på rad **24**. För att få reda på vad parametrarna i **Date**-objektets konstrukor står för, skriv ut dem med hjälp av get-metoder som demonstreras på raderna **13-22**.

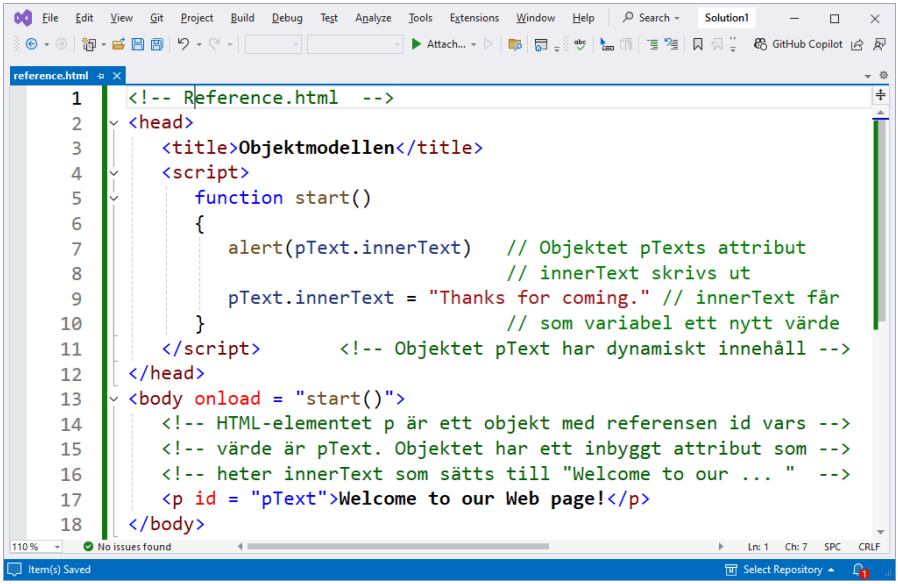
Skapa en egen ny tid med valfria parametrar till konstruktorn. Skriv ut dem och ändra på dem med de set-metoder som visas på raderna **27-32**.

Kapitel 7

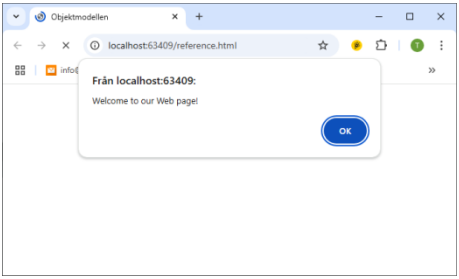
Objektmodellen

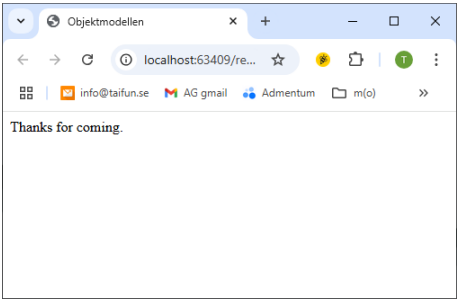
och Collections

7.1 Objektmodellen



För att förstå detta avsnitt bättre rekommenderas att läsa om objekt i JavaScript i avsnitt 6.1 (sid 119).





JavaScripts objektmodell

I detta avsnitt behandlar vi den s.k. *objektmodellen* i JavaScript, vilket innebär att JavaScripts objektbaserade karaktär utvidgas till HTML. Det betyder bl.a. att HTML-element behandlas som objekt. Det gör man genom att tilldela elementen attributet **id** och ge det ett värde som sedan kan användas som referens (namn) till objektet. Dessutom har JavaScript definierat attribut (egenskaper) och metoder till objektet som kan användas för att manipulera objektet.

HTML-element som objekt

I scriptet **Reference** (ovan) på rad **17** har t.ex. HTML-elementet **p** definierats som objekt genom attributet **id** vars värde **pText** kan anses som en referens till objektet. Funktionen **start()** refererar till detta objekt två gånger, en gång på rad **7** och en annan gång på rad **9**. Dessutom har JavaScript definierat attributet (egenskapen) **innerText** till objektet **p**. Scriptet **Reference** använder detta attribut på rad **7** för att hämta (läsa) den aktuella texten som är definierat i **p** HTML-elementet som innehåll (rad **17**) och skriva ut den i **alert**-boxen. Sedan används attributet **innerText** för att ändra (skriva) texten på rad **9**. Att denna ändring är möjlig beror på att **innerText** är som attribut i ett objekt en *variabel* vars värde är dynamisk och kan ändras. **innerText** pekar på texten i innehållet av HTML-elementet **p** som i sin tur har referensen (namnet) **pText**. Dvs man skriver:

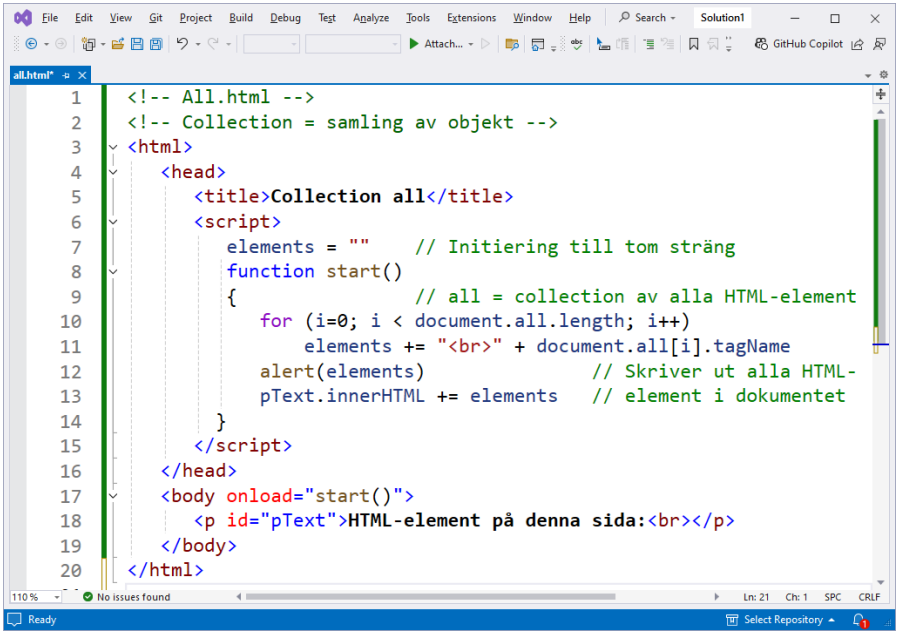
```
pText.innerText
```

för att komma åt texten i innehållet av HTML-elementet **p**, både för att hämta den och för att ändra den. Detta hade utan JavaScriptets objektmodell inte varit möjligt.

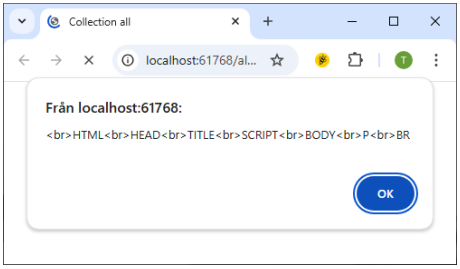
Referens till objekt

I scriptet **Reference** har HTML-elementet **p** blivit ett objekt i JavaScript. *Referensen* till detta objekt, dvs koden man använder för att komma åt objektet, är **pText**, som är värdet till **p**-elementets attribut **id**. Det är därför vi använder denna referens för att via attributet **innerText** ge ett annat innehåll till HTML-elementet **p**.

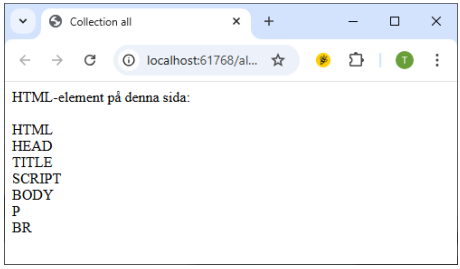
7.2 Collection all



Vad är en Collection?



En *Collection* i JavaScript är en samling av objekt, liknande en array av element. Skillnaden är att en arrays element måste alla vara av samma datatyp, medan en collections ”element” är *objekt*. Olika objekt är olika datatyper. Ett exempel på collection är **`all`** som behandlas nedan. Andra exempel följer i de kommande avsnitten.



Collection all

I scriptet **`All`** (ovan) används på raderna **`10`** och **`11`** collection **`all`** som är samlingen över alla HTML-element i ett dokument resp. script, i den ordning de förekommer.

Med collection **`all`**:s attribut **`tagName`** kan man referera till dessa element, speciellt när elementen själva saknar attributet **`id`**. Scriptet **`All`** loopar igenom collection **`all`** och skriver ut alla HTML-element i dokumentet. Det gör det genom att skriva

dem till **p**-elementets attribut **innerHTML** på rad **13**. Attribut **innerHTML** liknar attributet **innerText** som användes i scriptet **Reference** (sid 132).

Övningar till **Kapitel 7**

7.1 Modifiera scriptet **Reference** (sid 132) genom att ändra texterna i koden av scriptet.

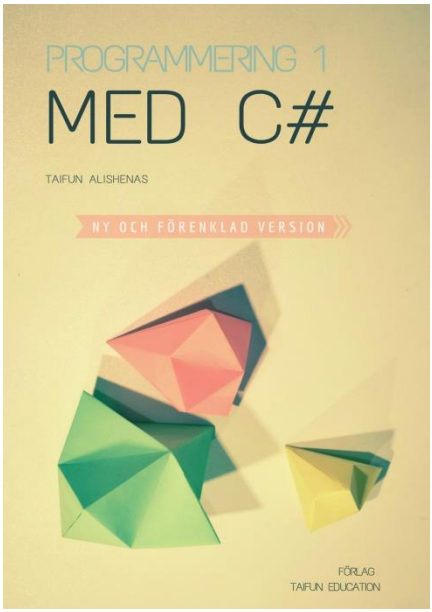
Byt sedan ut texterna i funktionen **start()** och innehållet av HTML-elementet **p** mot varandra och kör. Tolka resultatet.

Skriv ett andra element **p** med en annan text i innehållet samt hämta och ändra den i funktionen **start()**.

7.2 Byt ut i scriptet **All** (sid 134) **p**-elementets attribut **innerHTML** mot **innerText** som användes i scriptet **Reference** (sid 132). Vad exakt är skillnaden? Avgör vilket alternativ som är bättre.

Modifiera scriptet **All** genom att skriva ut dokumentets HTML-element i omvänd ordning, jämfört med dokumentets ordning, t.ex. genom att ändra **for**-satsen.

Programmering 1 med C#



Ur innehållet:

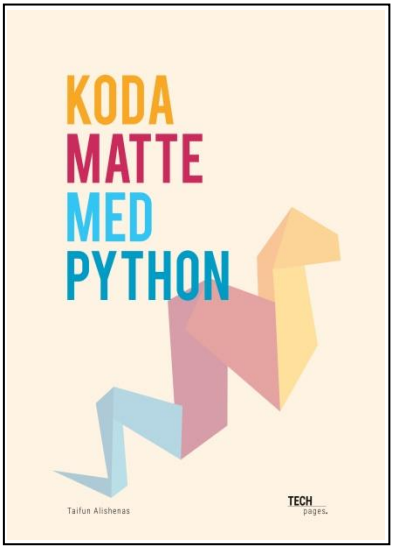
Grundbegrepp i programmering
Datatyper, variabler & tilldelning
Utskrift till grafisk miljö
Windowsprogrammering
C# Console & Windows Applications
Interaktiva grafiska gränssnitt
Kontrollstrukturer
Klasser, objekt och referenser
Metoder
Rekursiva metoder
Sammansatta datatyper: Arrays
Dynamiska arrays: 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.**

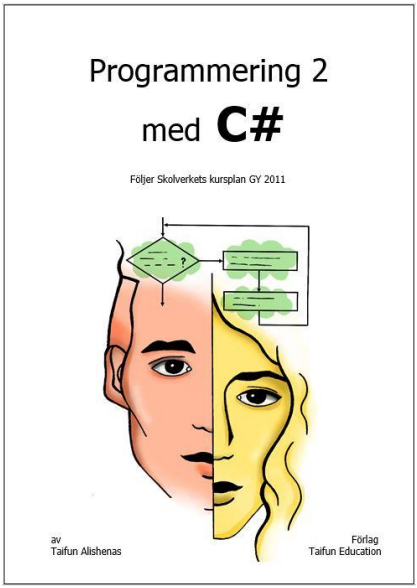
Koda matte med Python

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.



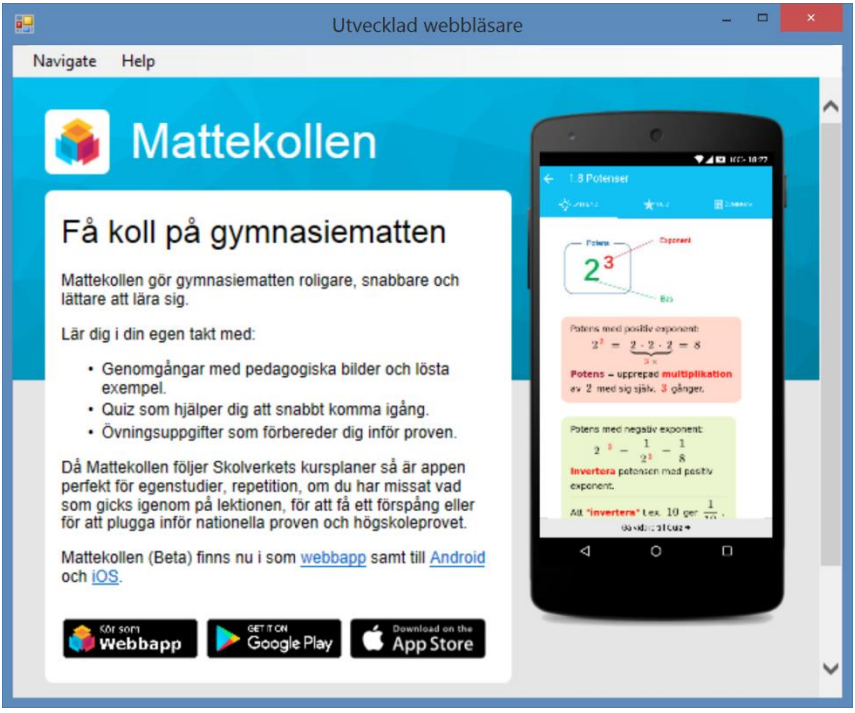
Programmering 2 med C#



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*.