Webbutveckling 1 med HTML, CSS och JavaScript

Med övningar och projektupppgifter

Förlag: Lieta AB

Titel: Webbutveckling 1 med HTML, CSS och JavaScript

Författare: Taifun Alishenas
info@taifun.se

August 2025

Innehåll

Kapitel 1

Introduktion till Webbutveckling

1.1 Om webbutveckling

Världen vi lever i är full med prylar som kallas för "intelligenta". Man talar om *artificiell intelligens*. Men prylarna kan inte tänka själva. Någon har programmerat dem, närmare bestämt de elektroniska komponenterna i dem – små datorer som styr all funktionalitet och interaktivitet. De programmerade små prylarna kallas för *embedded systems*. Är de dessutom uppkopplade mot Internet pratar man om *Internet of Things (IoT)*, resultat av en utveckling som tog fart på 90-talet och sedan dess har revolutionerat alla områden i det sociala livet och människans sätt att leva i hela världen – på gott och ont.

Programmering är ett av de mest spännande kapitlen i teknologihistorien. Inte bara därför att den har lagt grunden till den moderna IT-industrin. Den har bidragit till att förverkliga den urgamla mänskliga drömmen att förenkla mödosamma arbeten. Istället för att plåga sig instruerar man en maskin med idéer och låta den göra jobbet, för att ha mer tid över för annat roligt i livet. Det är roligare att köra en bil än att bara åka med. Det är kreativiteten och det fria skapandet som lockar. Man kan testa helt nya egna idéer.

Webbutveckling är en speciell form av programmering, bara att språket man kodar med – och även resultatet man får, är lite annorlunda. När man tröttnat på att använda program som andra skrivit – surfa, chatta, maila eller lyssna på musik – är det dags att börja programmera själv. Och varför inte skapa och publicera en egen webbsida?

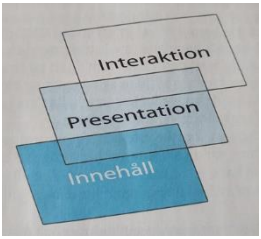
Vad behöver man för webbutveckling?

Mindre än för programmering. Det räcker med en dator samt följande:

1. Lite programmeringstänkande
2. En vanlig texteditor
3. En webbläsare (web browser)
4. Ett bildbehandlingsprogram (helst)

För 1 läs vidare! För 2-4 spelar det ingen roll om du har en Mac-, Windows- eller Linux-dator. En editor och en webbläsare finns förinstallerade på alla datorer. Annars rekommenderas Google Chrome som webbläsare som kan gratis laddas ned. Andra exempel ges senare (sid 13).

De tre skikten



Man brukar tala om tre olika skikt när det gäller webbsidor. Och det rekommenderas att hålla isär dessa och behandla dem till tre olika webbutvecklingsvertyg:

- *Innehållet* kodas med **HTML**.
- *Presentationen* formges med **CSS**.
- *Interaktionen* programmeras med **JavaScript**.

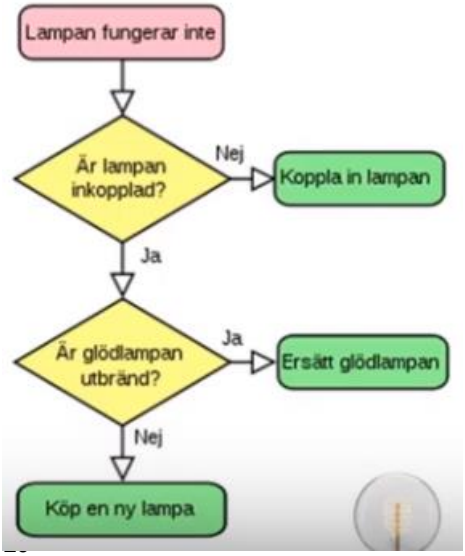
Verktygen kommer att tas upp i detalj senare. De utgör huvudinnehållet i kurserna webbutveckling 1 och 2.

Förr i tiden blandades dessa tre skikt, vilket ledde till svårigheter. Under tiden har man lärt sig att modularisera, dvs bryta ned och strukturera utvecklingsarbetet genom att separera de här tre skikten. Som kriterium för en webbsidas kvalitet anses primärt sidans användarvänlighet. Kommunikationen med användaren står i centrum. Andra viktiga kriterier är enkelhet och möjligheten att underhålla och uppdatera sidan.

Programmeringstänkande

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

Flödesschema



Egentligen programmerar vi varje dag utan att vara medvetna om det. Är t.ex. en lampa trasig hemma följer vi ungefär ett handlingsförlopp 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.

Algoritmer

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 webbutveckling 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 webbutveckling hittat på andra, speciella språk 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 flödes-

planen 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 webbutvecklarens uppgifter.

1.2 Webbutvecklingens miljöer

Webbutveckling är i allra högsta grad ett praktiskt ämne.

Man kan inte lära sig webbutveckling genom att endast läsa böcker. För att lära sig webbutveckling måste man skriva kod och testa koden – precis som när man lär sig att köra bil. Och för att göra det behöver man en miljö, där man kan skriva och en annan där man kan köra och testa koden. Den första är en en *editor*, den andra en *webbläsare (browser)*. Det räcker fullt med dessa två verktyg för att bedriva webbutveckling. Så se till att du skaffar dessa två verktyg, så att de är kompatibla (passande) för din dator och för ditt operativsystem.

Editorer

En *editor* är ett skrivverktyg på datorn, där man kan skriva text och spara den i en fil. Vi kommer att skriva våra HTML-koder i en editor. Däremot kan man inte *exekvera* HTML-kod i en editor. För det behövs en webbläsare, se nästa paragraf. Det finns två olika typer av editorer:

1. Sådana som skriver ren text *utan formatering*, dvs utan att bestämma typsnitt, stil, storlek, färg osv., jämförbara med gamla mekaniska skrivmaskiner. Exempel på sådana enkla editorer är: Anteckningar (Notepad), Notepad++, TextPad osv. på Windwos-datorer och Textredigerare, TextEdit, Emacs osv. på Mac-datorer. Dessa editorer sparar sina texter i textfiler, ofta med filändelsen **`*.txt`**. Sådana editorer kan vara lämpliga för webbutveckling och programmering, eftersom de sparar text *utan* formatering, dvs utan ytterligare kod.

2. Mer avancerade editorer, s.k. *ordbehandlingsprogram*. Exempel på sådana program är **`Google Docs`**, en online editor eller **`Word`** på Windwos-datorer. Båda formaterar texten. **`Word`** sparar sina filer som dokument av typ **`*.docx`**. Motsvarigheten på Mac-datorer är Pages. Sådana ordbehandlingsprogram är **inte** lämpliga för att skriva kod. Anledningen är att **v**i vill skriva i editorn HTML-kod som sedan ska exekveras av webbläsaren. Men formatering innebär att det automatsiskt läggs till osynliga formateringskoder i texten som webbläsaren inte känner till. Det är osynliga styr- och kontrolltecken som ordbehandlingsprogrammet använder för att bestämma typsnitt, stil, sorlek, färg osv. och som ”stör” vår egen HTML-kod.

Webbläsare

(engl. *web browser*) är ett program som bl.a. kan *tolka* och *exekvera* HTML-kod samt andra scriptspråkens koder. Att *tolka* är synonym till att *interpretera*, dvs omvandla HTML-koden till maskinkod. Att *exekvera* betyder *utföra* maskinkoden, vilket sker i datorns processor. Webbläsaren är alltså i huvudsak en HTML-interpretator. Själva webbläsaren är ett program som har skrivits i något av de universella språken, ofta i C/C++. Exempel på sådana webbläsare är Google Chrome, Internet Explorer, Firefox, Safari, Netscape, ... En webbläsare finns förinstallerad på alla datorer.

HTML-kod kan exekveras av webbläsaren både lokalt och från Internet. I båda fall måste koden vara sparad i en fil. Ska HTML-koden exekveras i en webbläsare måste filen som innehåller koden, ha ändelsen **`html`**, vare sig vi har lagrat filen lokalt eller hämtat den från en server på Internet. När vi testar våra koder i webbutvecklingskursen gör vi det lokalt.

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. Alla webbutvecklingsspråk är interpreterande, inkl. HTML.

Till skillnad från en interpretator är en *kompilator* ett program som *översätter* källkod till maskinkod och lagrar maskinkoden på hårddisken. Det uppstår en mellanprodukt – en fil av typ **`*.exe`**. Först när man exekverar skickas den kompilerade maskinkoden (mellanprodukten) till datorns processor och utförs där. Vissa programmeringsspråk är kompilerande (C/C++), andra är interpreterande (Python).

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 HTML, CSS och JavaScript behöver ingen IDE. Det räcker med en enkel editor 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 (IDEs). Vi kommer att skriva våra koder i en editor och exekvera dem i en webbläsare. Självklart kan man även använda editorn i en IDE för att skriva kod och IDEns webbläsare för att exekvera koden.

HTML – webbens standardspråk

HTML står för ***H****yper****T****ext* ***M****arkup* ***L****anguage* och är webbens standardspråk. ***H****yper****T****ext* betyder att det är text som innehåller länkar till andra delar av sig själv och till s.k. externa mål, dvs andra dokument på webben,. ***M****arkup* betyder att det är ett språk med markeringar, dvs med formateringar som kodas med s.k. *taggar*. En viktig egenskap av HTML är att koden inte skiljer mellan små och stora bokstäver, dvs:

> HTML är **inte** case sensitive (skiftlägeskänslig).

Syftet är att producera presentabla *dokument* som kombinerar text, bild och andra element. Därför kan man kalla HTML även för ett ordbehandlingsprogram. Man skriver kod som genererar dokumentet som sedan visas på webben. Koden är separerad från dokumentet – till skillnad från andra ordbehandlingsprogram som

t.ex. **Word**, där man direkt skriver i dokumentet och formaterar texten med vissa inbyggda verktyg. Koden genereras automatiskt i bakgrunden och är osynlig för användaren. Därför är **Word** ett s.k. ***WYSIWYG***-verktyg, dvs ***W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et*. I **Word** *ser* man, vad man *får* i dokumentet.

Till skillnad från **Word** är HTML ett icke-*WYSIWYG*-verktyg, dvs man skriver koden – blandad med texten – utan att se dokumentet. Koden måste först *tolkas* av en interpretator, innan dokumentet kan uppstå. Webbläsaren är en programvara som kan tolka HTML, dvs en interpretator som exekverar koden och visar dokumentet. Andra exempel på icke-*WYSIWYG*-verktyg är TeX/LaTeX, ett prgram för typsättning och presentation av text, speciellt matematiska uppsatser.

Script vs. program

Med *script* menas all kod som kan köras på webben. T.ex. är HTML-kod inkl. all kod som kan inbäddas i HTML script. Och språk som används i en sådan kod kallas för *scriptspråk*. Exempel på scriptspråk är *HTML, CSS, JavaScript, PHP, ...* . JavaScript används i regel på klientdatorer, medan PHP är webbservrarnas scriptspråk.

Det finns två olika kategorier av språk i datavärlden: *scriptspråk* och *programmerings-* eller *universella* språk som t.ex. C, C++, C#, Java, Python, ... Koder till universella språk kan användas för vilken applikation som helst och är inte begränsade till webben. Deras koder kallas för *program*. HTML är ett *scriptspråk*, inget programmeringsspråk. HTML har möjligheten att bädda in andra scriptspråk i sin kod som t.ex. JavaScript. I koden avgränsar man språken genom tydliga markeringar. Webbläsaren är den naturliga exekveringsmiljön både för HTML och alla andra scriptspråk, medan särskilda verktyg behövs för att exekvera universella språk. För att *skriva* scriptkod behöver man endast en *editor*, och för att exekvera den, en *webbläsare*. Vi nöjer oss med denna minimalistiska miljö vad gäller scriptspråken, för att förenkla den tekniska hanteringen och koncentrera oss på själva språket.

Det mest kända scriptspråket är *JavaScript* som skapades år 1995 av *Netscape*, ett amerikanskt mjukvaruföretag som 1994 lanserade den första grafiska webbläsaren *Mosaic* som snabbt blev en jättesuccé. Netscape integrerade JavaScript i Mosaic, för att göra webben interaktiv, se ***De tre skikten*** sid 10.

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.

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

Scriptspråkens *exekverings*miljö är webbläsaren (web browser).

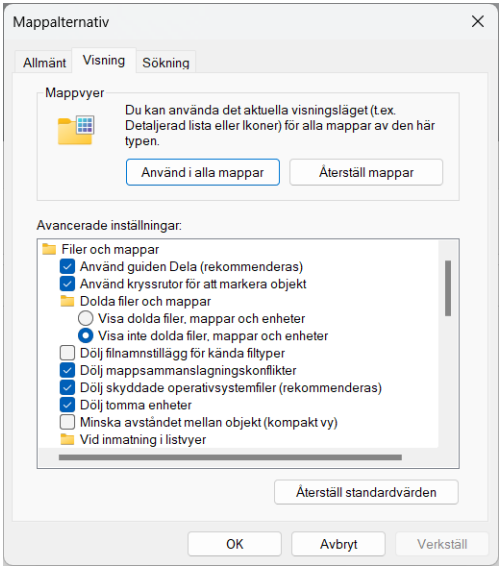
Scriptspråken är *interpreterande* språk, dvs koden tolkas till maskinkod (datorns språk) av en interpretator som är inbyggd i webbläsaren. Maskinkoden utförs direkt av datorns processor utan att den mellanlagras. De mest använda webbläsarna är Google Chrome på Windwos-datorer och Safari på Mac-datorer.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

Filändelser

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*.

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 HTML

För att komma igång med HTML kan vi nu skriva våra koder i en valfri texteditor 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 HTML-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.

Vårt första HTML-script

Öppna din favorit editor (t.ex. Notepad, Notepad++, TextPad i Windwos eller Textredigerare, TextEdit, emacs på Mac) och skriv följande kod. Ta bort radnumren **`1-7`** som har satts endast för att underlätta kodförklaringen. I övrigt bibehåll layouten:

```
1  <!-- Welcome.html
2       Skriver ut en rad text i fet stil & storleken h1 -->
3
4  <head>
5     <title>Vårt första HTML-script</title>
6  </head>
7
8  <body>
9     <h1>Välkommen till HTML!</h1>
10 </body>
```

Spara koden 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:

Vi kommer i fortsättningen att referera till koden som *scriptet* **`Welcome`** och till filen som **`Welcome.html`**. I följande går vi igenom koden i detalj.

Kommentarer

Raderna `1-2` i scriptet `Welcome` är *kommentar*. dvs de utförs inte. De ska förklara koden. Allt som skrivs mellan `<!--` och `-->` betyder kommentar, HTML koden `<!--` inleder och `-->` avslutar en kommentar som kan sträcka sig över flera rader eller stå mitt på en rad. Att skriva kommentarer tillhör god programmeringsstil. Även indragningarna, på eng. *indentations*, på raderna `5` och `9` är element av god programmeringsstil som ska hjälpa oss att förstå att den indragna koden är nästlad i en övergripande struktur som inte är indragen, när vi läser scriptet.

HTML-taggar

En *tagg* i HTML inleds med `<` och avslutas med `>` . Taggar används i regel parvis: en *starttagg* markerar *början* och en *starttagg* markerar *slutet*. På rad `9` har vi:

```
<h1>Välkommen till HTML!</h1>
```

Denna rad har två taggar: starttaggen `<h1>` och sluttaggen `</h1>`.

Det som står mellan start- och sluttagg, texten `Välkommen till HTML!`, kallas för *innehåll*, närmare bestämt innehåll till `<h1>`-elementet (se nästa rubrik).

Taggar är HTML-språkets minsta byggstenar. HTML är ett s.k. *taggat märkspråk*. Det som skrivs *inom* en tagg, dvs mellan `<` och `>`, är HTML-kod. Allt som står *utanför* taggen kommer att visas som text på webbsidan, se regeln på nästa sida.

Taggar kan nästlas i varandra.

HTML-element

Ett HTM-*element* har följande ingredienser:

Starttagg + innehåll + sluttagg = **Element**

`<h1>Välkommen till HTML!</h1>` är i sin helhet ett HTMLelement, närmare bestämt ett `<h1>`-element. Element kan nästlas i varandra. I scriptet `Welcome` är `<h1>`-element nästlat i ett `body`-element (raderna `8-10`).

Ett annat exempel på element hittar vi på rad `5`:

```
<title>Vårt första HTML script</title>
```

Även detta element är nästlat i ett annat element, nämligen i `head`-elementet på raderna `4-6`.

Om taggar är HTML-språkets minsta byggstenar, så är element språkets näst minsta byggsten.

En övergripande struktur

`head`- och `body`-elementen i scriptet **`Welcome`** kallas för *strukturingselement*. De delar in scriptet **`Welcome`** i två skilda delar: ett huvud och en kropp. Medan i kroppen kodas webbsidans huvudinnehåll, är huvudet tänkt för att innehålla bl.a. `title`-elementet, men även andra ingredienser som vi kommer att lära känna senare. Alla HTML-script följer den här övergripande `head`- och `body`-strukturen.

På raderna **`4-6`** står `head`-elementet som nästlar `title`-elementet. All text som skrivs som innehåll i `title`-elementet på rad **`5`** kommer att synas på rubriken av webbläsarens flik längst upp till vänster, kallad *title bar*, när man kör scriptet. Det är texten **`Vårt första HTML-script`** som kan ses där (lite svårläst). Utelämnar man `title`-taggen kommer scriptets filnamn att skrivas i title bar.

På raderna **`8-10`** står `body`-elementet som nästlar `h1`-elementet på rad **`9`** som omgärdar texten som ska skrivas ut: **`Välkommen till HTML!`**. `h1`-taggen bestämmer textens stil och storlek: Texten skrivs ut i fet stil och i den storlek som man ser i körresultatet på förra sidan. Med `h`-taggar kan man åstadkomma rubriker i texten. `h` står för *header*, dvs rubrik. Det finns flera header-taggar i HTML som vi kommer att lära känna i nästa avsnitt. `h1` är den största av dem.

Observera att texten som skrivs ut, inte behöver sättas inom varken apostrofer eller citationstecken i koden. Detta är en konsekvens av den regel som nämndes inledningsvis och som vi nu sammanfattar som en övergripande regel:

> Inom en tagg står **HTML-kod**. Allt som står utanför taggar anses vara **text** som ska visas på webbsidan.

Denna regel återspeglar HTMLs karaktär som ett icke-*WYSIWYG*-verktyg (sid 14).

Headers

```
1  <!-- Headers.html
2       Skriver ut flera rader text i olia stilar och storlekar -->
3  <head>
4     <title>Olika stilar och storlekar för rubriker</title>
5  </head>
6  <body>
7     <h1> Rubrik i HTML med h1 </h1>
8     <h2> Rubrik i HTML med h2 </h2>
9     <h3> Rubrik i HTML med h3 </h3>
10    <h4> Rubrik i HTML med h4 </h4>
11    <h5> Rubrik i HTML med h5 </h5>
12    <h6> Rubrik i HTML med h6 </h6>
13 </body>
```

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

Vi kommer att referera i fortsättningen till koden ovan som *scriptet* **`Headers`**. Som man ser skriver sciptet **`Headers`** ut fyra rubriker i olika storlekar, förorsakat av HTMLs `<hi>`-taggar `(i = 1, 2, 3, 4, 5, 6)` som formaterar textens storlek.

En fråga dyker upp här: Vad har förorsakat radbyten i körresultatet ovan? Vi har inte skrivit i scriptet någon kod som ska åstadkomma radbyten. Slutsatsen är:

> `<hi>`-taggarna `(i = 1, 2, 3, 4, 5, 6)` avslutar varje rubrik med radbyte.

Utskrift i flera rader

Frågan som automatiskt dyker upp efter förra avsnitts slutsats, är: Hur kan vi själva åstadkomma radbyten. Med radbryte menar vi förstås inte radbryte i koden utan i *utskriften*, dvs i körresultatet. Scriptet **`Break`** nedan visar *ett* sätt att i HTML åstadkomma egna radbyten och kombinerar detta med resultatet från förra avsnitt:

```
1  <!-- Break.html
2       Radbrytning i utskriften med HTMLs break-tagg <br> och
3       genom byte av h-taggens rubrikstil -->
4
5  <head>
6     <title>Utskrift i flera rader</title>
7  </head>
8
9  <body>
10    <h1>Välkommen till<br>Webbutveckling 1</h1><h2>med HTML och CSS!</h2>
11 </body>
```

Det första radbytet kodas med HTML-taggen `<br>` som bakas in i texten. Det andra radbytet genereras automatiskt genom att vi stänger `<h1>`-taggen och öppnar `<h2>`-taggen, se slutsatsen på förra sidan.

Körresultatet blir en utskrift på tre rader:

`br`-*taggen*

Scriptet **`Break`** använder HTML-taggen `<br>`, även kallad *break-taggen* genom att baka in den i texten **`Välkommen till`**`<br>`**`Webbutveckling 1`** (rad **`8`**) för att åstadkomma radbrytning i utskriften. break-taggen kan placeras var som helst.

Man kan koda taggen med `<br>` eller med `<br />`, så att även sluttaggen syns. Det är lite smaksak. Anledningen är att **`br`** bildar ett s.k, *tomt element*, som har vissa regler, vilket förklaras senare (sid 29). Vi nöjer oss än så länge med `<br>` för enkelhetens skull.

Frågor till **Kapitel 1**

1.1 Vad är *embedded systems* och hur skiljer de sig från *Internet of Things (IoT)*?

1.2 Hur tolkar du termen *artificiell intelligens?* Kan maskiner ”tänka”?

1.3 Varför tror *Steve Jobs* att programmering lär oss hur vi ska *tänka?*

1.4 Vad är relationen mellan webbutveckling och programmering?

1.5 Vilka verktyg behöver man för webbutveckling?

1.6 Nämn webbutvecklingens tre skikt. Varför ska man skilja dem från varandra?

1.7 Vad menas med programmeringstänkande?

1.8 Hur skulle du definiera begreppet *algoritm?*

1.9 Är datorprogram ett sätt att *beskriva* algoritm? Om ja, är det det enda sättet?

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

1.11 På vilket sätt kan man visualisera en algoritms logiska struktur?

1.12 Vad har algoritmiskt tänkande med programmering att göra?

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

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

1.15 Vilka är algoritmers byggstenar?

1.16 Vad är skillnaden mellan *instruktioner* och *val* i en algoritm?

1.17 Vad är kontrollstrukturer? Nämn tre exempel på dem.

1.18 Varför kan man inte lära sig webbutveckling genom att endast läsa böcker?

1.19 Vilken egenskap borde editorn ha i vilken man skriver kod?

1.20 Vad är en IDE? Vad är skillnaden till en editor?

1.21 Vad är en webbläsare?

1.22 Vad är en *interpretator* och hur skiljer den sig från en *kompilator?*

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

1.24 Skriver man *källkod* eller *maskinkod* när man kodar i något språk?

1.25 Vilka två kategorier av språk är relevanta för webbutveckling?

1.26 Är HTML ett *universellt* programmeringsspråk?

1.27 Är HTML ett interpreterande eller ett kompilerande språk?

1.28 Är HTML case sensitive?

1.29 Vad är ett *script* och hur skiljer det sig från ett *program*?

1.30 Nämn några exempel för *scriptspråk* och några för *universella* språk.

1.31 I vilken miljö exekveras HTML-kod?

1.32 Vad är ett WYSIWYG-verktyg? Nämn några exempel.

1.33 Är HTML ett WYSIWYG-verktyg?

1.34 Vilka verktyg behöver man för att kunna utveckla HTML-script?

1.35 Vilka typer av ordbehandlingsprogram är olämpliga för webbutveckling?

1.36 Varför är filändelser relevanta för en webbutvecklare?

1.37 Hur kodar man kommentar i HTML?

1.38 Varför är kommentarer viktiga när man skriver kod?

1.39 Vilken övergripande struktur bör man följa i alla HTML-script?

1.40 Var någonstans i dokumentet hamnar texten man skriver i **`<title>`**-taggen?

1.41 Vilken text kommer att skrivas om man utelämnar **`<title>`**-taggen?

1.42 Nämn några exempel på *headers* (rubriktaggar) i HTML. Hur många finns?

1.43 Vilka egenskaper får en text som formateras av **`<h1>`**-taggen?

1.44 Vad är en bieffekt av headers som inte syns i koden?

1.45 Nämn några exempel på HTML-taggar som saknar sluttagg.

1.46 Vad består ett HTML-element av?

1.47 Vad menas med ett *tomt* HTML-element?

1.48 Hur kodar man radbyte i HTML?

1.49 Nämn två olika sätt att koda break-taggen.

1.50 Varför bildar break-taggen ett tomt element?

Övningar till **Kapitel 1**

1.1 Om du har en favorit editor, öppna den (sid 13). Om inte, ladda ned en editor (t.ex. Notepad++ eller Emacs)och installera den. Undersök i editorn skillnaderna – vad gäller formen och utseendet – mellan tecknen *apostrof* (**'**), *citationstecken* (**"**), *accent* (´), *slash* (**/**) och *backslash* (****), så att du kan skilja dem från varandra. Försök att memorera deras tangenter på din dators tangentbord.

1.2 Visar din dator filändelserna när du öppnar en mapp? Om inte, genomför instruktionerna ***Filändelser*** på sid 16, för att synliggöra filändelserna.

1.3 Öppna din favorit editor och mata in koden till scriptet **`Welcome`** (sid 17). Bibehåll layouten. Spara koden i filen **`Welcome.html`**. (Dubbel)klicka på filen, så att den körs i din webbläsare.

1.4 Modifiera scriptet **`Welcome`** genom att ändra texten i **`<title>`**-taggen till ditt namn och texten som skrivs ut i dokumentet, till: **`Det här scriptet har jag skrivit själv!`** Spara koden i filen **`Mitt.html`** och kör den i din webbläsare. Välj en lämplig mapp på din dator för att spara filen. Skapa en mappstruktur för scriptfiler du kommer att ha användning av för kursens kodexempel.

1.5 Modifiera koden i scriptet **`Headers`** (sid 19), så att de fyra utkriftsraderna syns i växande textstorlekar istället för minskande.

1.6 Skriv ett HTML-script som åstadkommer följande utskrift:

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

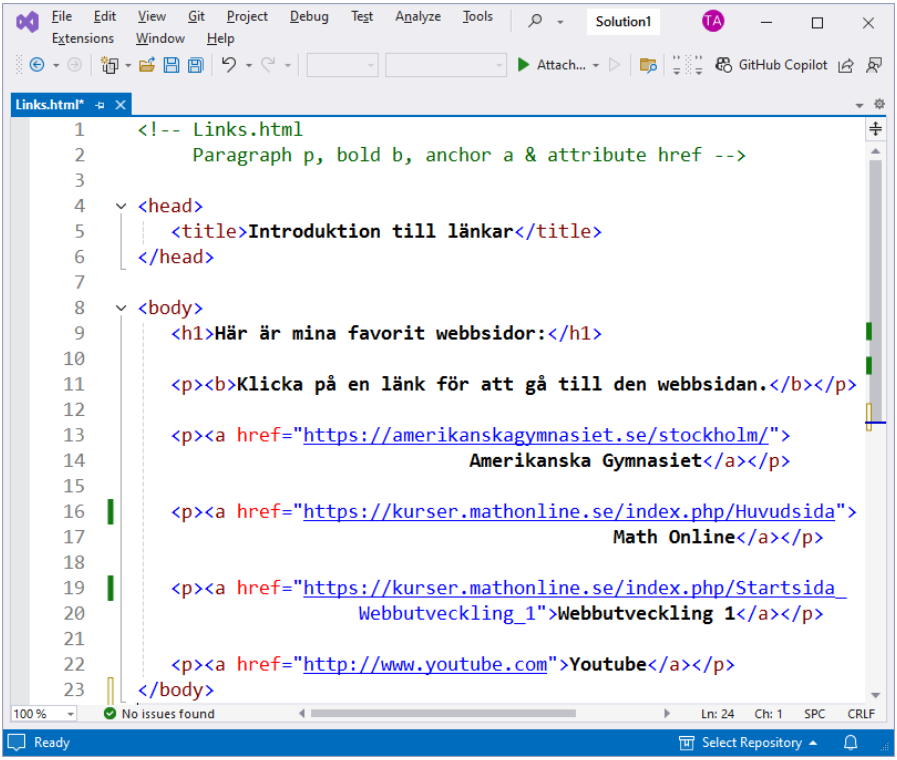
1.7 Skriv kod som ger följande utskrift:

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

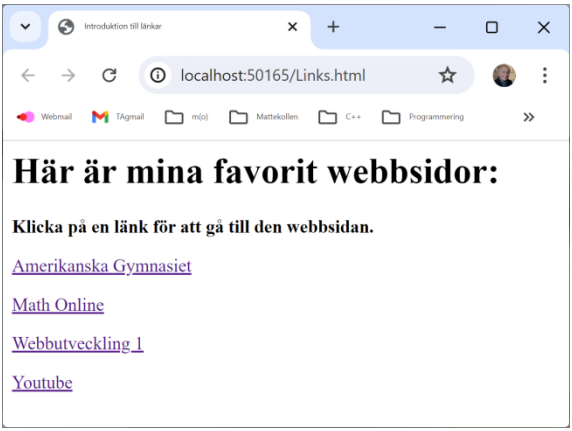
Kapitel 2

Grundbegrepp i webbutveckling

2.1 Länkar

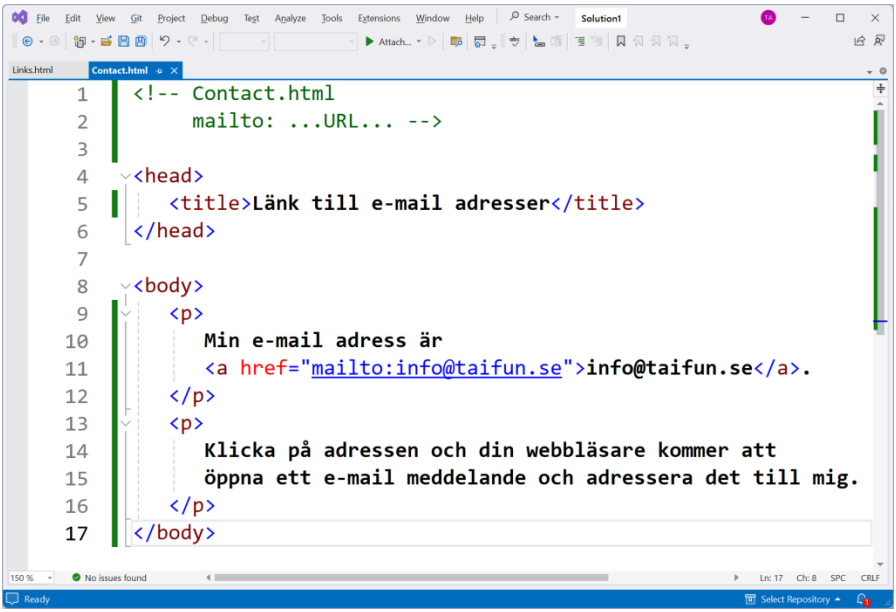


Scriptet **Links** introducerar elementtyperna *paragraph* **p** för stycke, *bold* **b** för fet stil och **a** för *anchor*. **p** skapar ett nytt stycke (paragraf) i textflödet. **b** byter textens stil till fet. Ankaret **a** skapar en länk. **a**:s *attribut* href specificerar webbadressen. Så här ser körresultatet ut där man kan navigera till de specificerade webbsidorna:



Attribut*namnet* href refererar till attribut*värdet* "http: ... (URL) ... " som skrivs inom citationstecken. URL står för Uniform Resource Locator och är formatet för giltiga webbadresser på Internet.

Låter man inleda attributvärdet istället för `http` med `mailto` kan man på samma sätt länka till mailadresser: `"mailto: ... (mailadress) ... "`. Scriptet **`Contact`** demonstrerar detta:



Så här ser körresultatet av scriptet **`Contact`** ut:

Elementet ankare

Den generella formen på HTML-elementet *ankare* (eng. *anchor*) är:

```
<a href="mailto:emailadress">...text/bild/emailadress...</a>
```

där **`...text/bild...`** är innehållet i elementet som blir en klickbar länk när man kör scriptet. Raden **`11`** i scriptet **`Contact`** ovan är ett exempel på denna form. Där skapas en länk till e-mail. `href` är ankarets attribut, närmare bestämt dess *namn*, medan **`"mailto:`***`emailadress`***`"`** är attributets *värde*.

Attribut

Observera att attributnamnet `href` samt dess värde är en del av ankar-elementets *starttagg*, inte av innehållet. Dvs det skrivs innan man avslutar starttaggen med `>` . Så här ser starttaggen ut i sin helhet, dvs med attributet:

`<a href="mailto:`*emailadress*`">`.

Att attributet inte är del av innehållet utan av starttaggen, beror på den regel som vi tidigare lärde oss om HTMLs övergripande sttruktur (sid 19):

> Inom en tagg står **HTML-kod**. Allt som står utanför taggar anses vara **text** som ska visas på webbsidan.

Attributet `href` är HTML-kod. Därför måste det stå *inuti* ankarens starttagg. Det är innehållet **`...text/bild...`** som ska visas på webbsidan. Därför måste det stå *utanför* taggarna, närmare bestämt mellan ankarens start- och starttagg, se raden **`11`** i scriptet **`Contact`** på förra sidan eller raderna **`13-22`** i scriptet **`Links`** (sid 26).

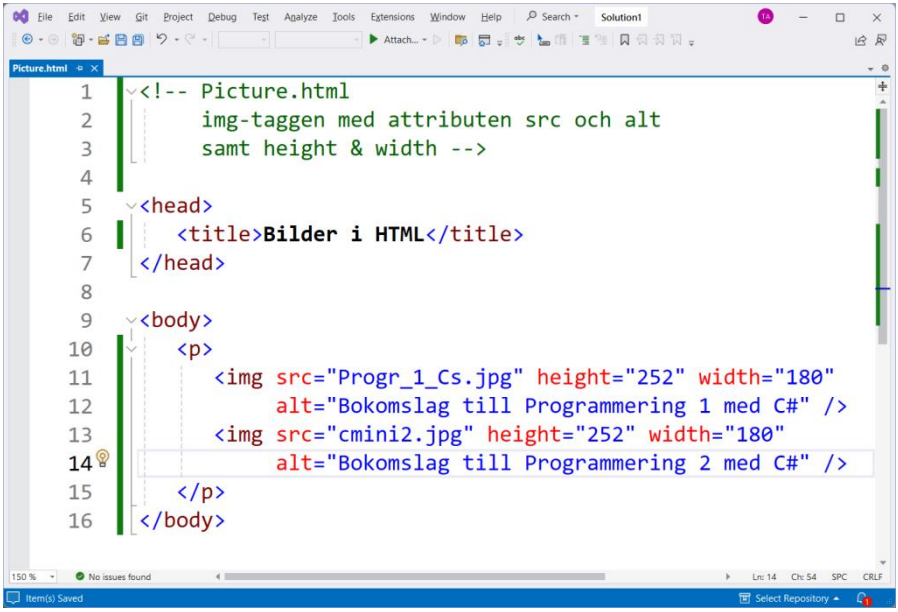
Varför heter det ankare?

Ett ankare anses vara en *förankringspunkt* för en förbindelse mellan två platser på webben: vårt HTML-dokument som vi aktuellt kör och webbsidan som Internet-platsen (URLen) i attributet `href` länkar till. Det är även möjligt att man länkar till samma HTML-dokument, men på en annan plats i dokumentet. Elementtypen **`a`** som står för engelskans *anchor* är den fasta punkten – *ankaret* – för dessa två platser. Den binder ihop scriptet med dokumentet.

2.2 Bilder

Hittills ha vi använt endast text i våra scriptexempel. Men en webbsida innehåller i regel text *och* bild. Att formge webbsidor med bilder är en väsentlig del av webbdesign. Bilder lagras i datorn i ett visst *format*, vilket framgår av filändelsen.

De mest anända bildformaten på webben är *Joint Photographic Experts Group* (JPG/JPEG) och *Graphics Interchange Format* (GIF). För att hantera bilder på webben rekommenderas, som det sades i början (sid 9), att ha ett bildbehandlingsprogram, t.ex. *Photoshop* eller motsvarande. Vi kommer att ha nytta av det i detta avsnitt, då vi kommer att behöva ta reda på t.ex. bildernas storlek, deras upplösning och ev. att anpassa deras storlek till våra varierande behov på webben. Följande script introducerar oss till att använda bilder i HTML:



`img` – *ett tomt element*

Raderna **11-12** använder **`img`**-elementet för att placera bilder i dokumentet. Den generella formen på ett **`img`**-element är:

```
<img src="filnamn" height="höjd" width="bredd" alt="text" />
```

Anledningen till att det är tomt, är att det egentligen borde se ut så här:

```
<img src="filnamn" height="höjd" width="bredd" alt="text"></img>
```

Och mellan starttaggen `<img ... >` och sluttaggen `</img>` finns inget innehåll `><`, därför är det tomt. Elementet ska inte skriva ut text, utan endast infoga en bild.

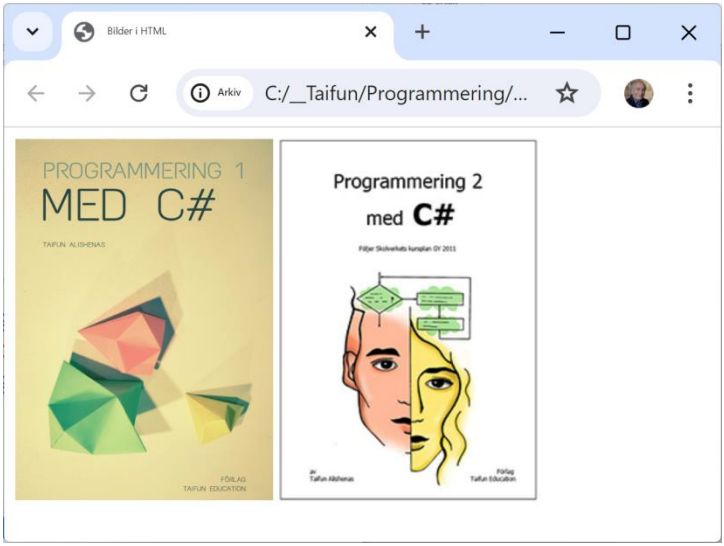
Förkortningsregeln för tomma element

Är ett element tomt kan man tillämpa följande regel i HTML för att förkorta koden:

> Ett **tomt element** i HTML kan avslutas på tre sätt:
>
> 1. med sluttaggen: `<starttagg></sluttagg>` eller
> 2. endast med `/>`: `<starttagg />` eller
> 3. endast med `>`: `<starttagg>`

`><` är det tomma innehållet. Punkt 1 visar det vanliga sättet att avsluta elementet: t.ex. **`img`**-elementet kan avslutas på vanligt sätt med `</img>`. Punkt 2 är ett sätt att förkorta koden: som en förenkling kan man avsluta endast med `/>`. Vi har i scriptet **`Picture`** på raderna **`12`** och **`14`** valt att använda den förenklade varianten 2. Mellanslaget ␣ före `/>` har lagts in endast för bättre läslighetens skull. Punkt 3 är ännu kortare: endast med `>` . Dvs bara starttaggen står där fullständigt. I praktiken har dock starttaggen ofta ett antal attribut, innan den avslutas, som vi utelämnat här. Vi har använt punkt 3 i scriptet **`Nav`**: s **`img`**-element på sid 32.

Annars gör **`img`**-elementet även med de förenklade varianterna exakt samma sak som med den vanliga, nämligen att infoga en bild i dokumentet. Testa gärna båda varianterna. Så här ser körresultatet av scriptet **`Picture`** ut:



Bildernas `jpg`-filer som är värden till attributet `src` (source) måste ligga i samma mapp som scriptfilen `Picture.html`, för att webbläsaren ska kunna ladda dem i dokumentet. Ligger de inte där eller kan webbläsaren av andra skäl inte ladda dem, kommer texten som är tilldelad attributet `alt` (alternative) att skrivas ut istället. Testa gärna!

Val av bildstorlek

Bildernas storlek i dokumentet bestäms i koden av **`img`**-elementets attribut `height` och `width`. Enheten till storleken är *pixlar (picture elements)*. Tittar man på bildernas originalstorlek i resp. `jpg`-fil – antingen i filens Egenskaper → Information (i Windows) eller i Photoshop (Image → Image Size) hittar man förstås andra värden. Hur ska man anpassa dem till dokumentet? Man bildar kvoten höjd / bredd av bildernas originalstorlek. I vårt fall får man ca. **1,4** . För att behålla bildernas rätta proportion i HTML-dokumentet, måste denna kvot tas över till koden. Dvs **`img`**-elementets attribut `height` och `width` måste väljas så att även `height` / `width` = **1,4** . Väljer man andra värden tappar bilderna sina urprungliga proportioner och blir förvrängda i dokumentet, vilket inte kommer att se bra ut. Utelämnar man attributen `height` och `width` kommer webbläsaren välja bildernas originalstorlek.

2.3 Bilder som länkar

Nästlade element

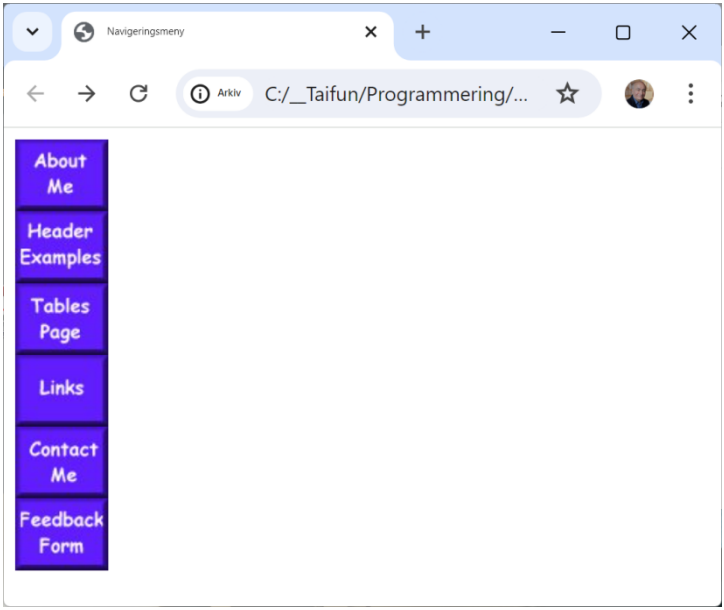
Scriptet **Nav** nästlar **img**-elementet i **a**-elementet (ankare), dvs sätter in **img** i ankarets innehåll. Det gör det 6 gånger, första gången på raderna **8-11**. Kör man **Nav** visas 6 små bilder. Deras samling kallar vi för Navigeringsmeny i title-taggen. Bilderna hämtas från undermappen **buttons** som ligger i samma mapp som scriptfilen **Nav.html**, eftersom **buttons** är den mapp där vi har placerat alla bildfiler av typ ***.jpg**. Det är **img** som visar dem på raderna **9, 14, 19, 24, 29** och **34**.

P.g.a. nästlingen i ankaret **a** blir de 6 små bilderna klickbara. Dvs de länkar till de webbsidor som **a**-elementets attribut **href** refererar till. T.ex. hämtar **img**-elementet bildfilen **Form.jpg** från mappen **buttons** till dokumentet (raderna **34-35**). Genom att **img** är nästlat som innehåll i ankaret **a**, blir bilden en länk som leder till **a**:s attribut **Picture.html** (rad **33**). Dvs scriptet **Picture**:s körresultat, de två bokomslagen, visas. Se nästa sida.

Alla andra delar av scriptet **Nav** fungerar på samma sätt. Tre scriptfiler av typ ***.html** ligger i undermappen Old (raderna **8, 13, 18**). De andra finns direkt i samma mapp som scriptfilen **Nav.html**.

Navigeringsmeny

Så här ser ett körresultat av scriptet **Nav** ut:



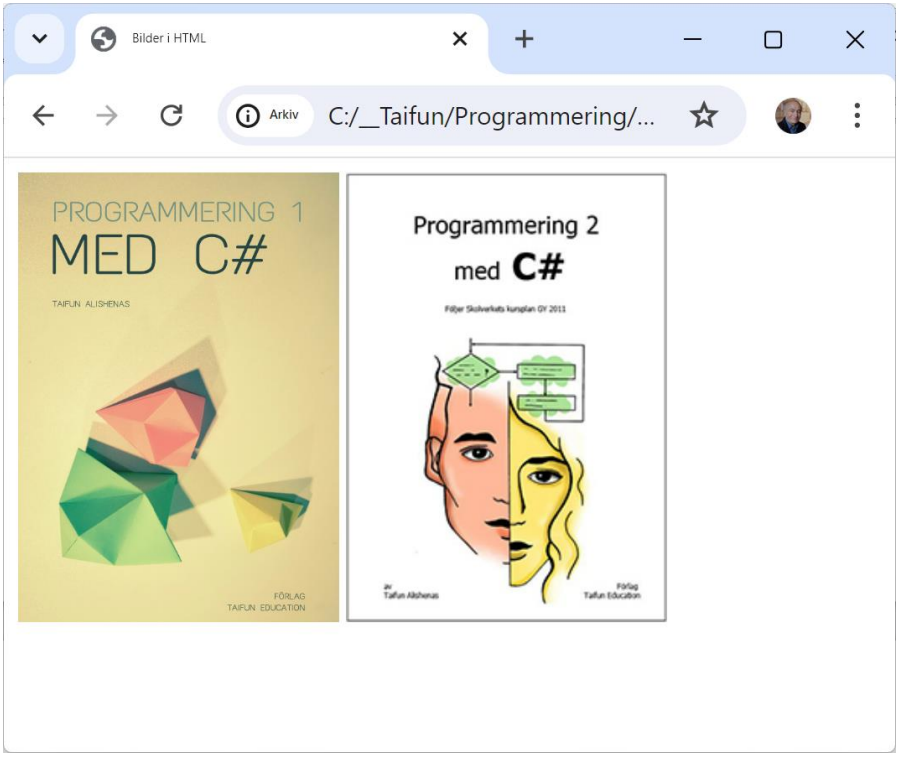
Sex små bilder visas som hämtats med **img**-elementet. Bilderna bildar en slags meny för navigering till andra hemsidor vars länkar finns bakom bilderna. Implemen-

terat är detta i scriptet **Picture** med **img**-element nästlade i **a**-element, vilket gör att bilderna blir klickbara. För varje bild används samma konstruktion.

Texterna på bilderna är lite godtyckliga och har mindre betydelse. Se över dem. Man kan lika bra använda andra bilder. Testa gärna!

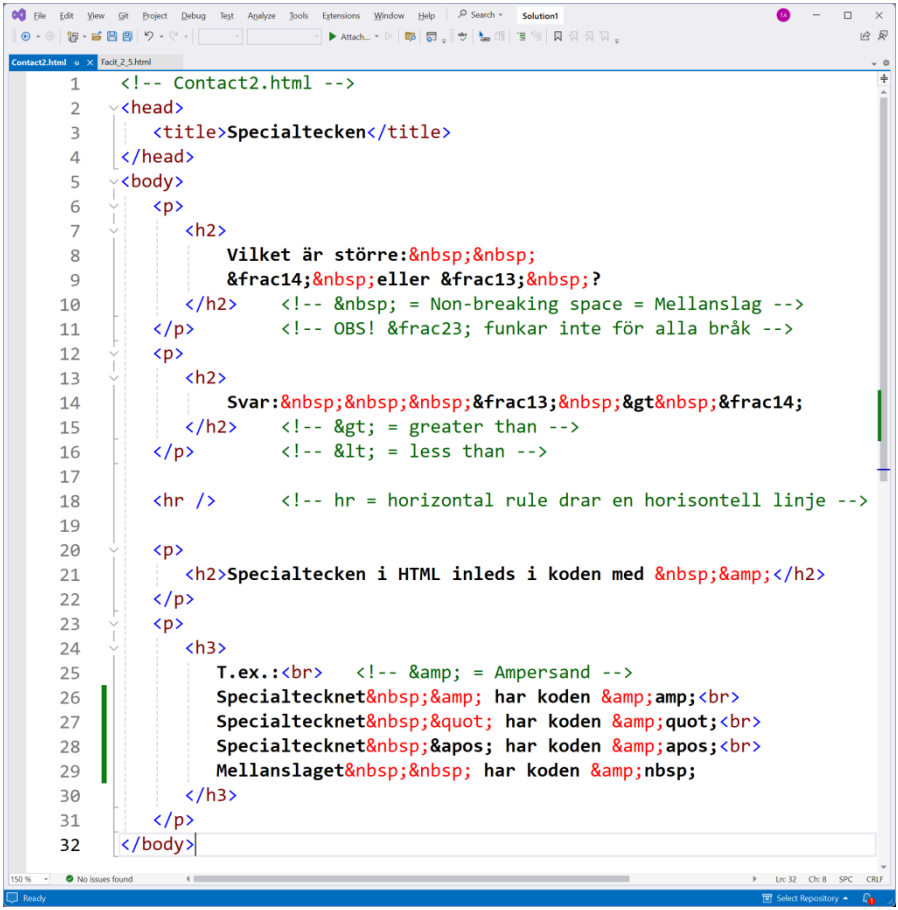
Interaktion

Men så länge vi inte interagerar med dokumentet händer ingenting. Klickar vi däremot i navigeringsmenyn ovan på den sista bilden som har texten **Feedback Form**, visas dokumentet som tidigare kodats i scriptet **Picture** (sid 29):



Vi klickade i navigeringsmenyn (förra sidan) på den sista bilden som hade texten **Feedback Form**. Eftersom denna bild enligt rad **36** är lagrad i filen **buttons/-Form.jpg** och enligt rad **35** länkar till filen **Picture.html**, blir det samma bild (sid 30) som tidigare producerats av scriptet **Picture**.

2.4 Specialtecken



Namnkoder för specialtecken

Alla tecken på tangentbordet förutom bokstäver och siffror är *specialtecken*. Några av dem är i HTML reserverade för att koda. T.ex. tecknen **<** och **>** används för att koda HTML-taggar. För att skriva ut själva *tecknen* **<** och **>** eller andra reserverade specialtecken till dokumentet behövs en speciell åtgärd. Man kan inte skriva dem direkt, eftersom då tolkas de av webbläsaren som kod. Denna åtgärd definieras här:

> Koden för specialtecken i HTML har följande form: **&code;**

där man för **code** kan sätta in de koder som är definierade för specialtecken, t.ex. **gt** som står för **g***reater* **t***han*:

>

Denna kod producerar tecknet **>** . Rad **14** i scriptet ovan använder den.

Ett annat exempel för definitionen ovan är specialtecknet **&**, kallat *ampersand*, som enligt definition inleder koden till just de reserverade specialtecknen. För att skriva ut själva tecknet **&** måste man koda **&**

När man kodar specialtecken följer efter ampersand ett *namn*, i exemplen: **amp, quot, apos, nbsp**. Alla koder avslutas med semikolon. T.ex. producerar koden **⅓** bråket **1/3**. Men tyvärr är koden inte generell, dvs andra bråk kan inte bildas på liknande sätt.

Andra, mer eller mindre intressanta specialtecken ges exempel på i scriptet **Contact2** på förra sidan vars körresultat ser ut så här:

Talkoder för specialtecken

Det finns möjligheten att ange *talkoder* istället för *namn*, när man kodar tecken i HTML, inkl. specialtecken. T.ex. ger **&** samma som **&** nämligen tecknet **&**, där **38** är ASCII-koden till tecknet **&**. ASCII är den äldsta och mest använda teckenstandarden och en del av, närmare bestämt, början av *Unicode*-tabellen. Faktiskt kan även vanliga tecken, inte bara specialtecknen, kodas med denna metod, dvs:

&#Unicode;

Unicodes **0-127** kallas för ASCII-koder. I övn 2.5 kan du öva dig på talkoder. Där hittar du även fler exempel på talkoder (sid 46).

En bra översikt över specialtecken med massor av nyttig information ger Internetlänken:

https://www.enur.se/tecken.html

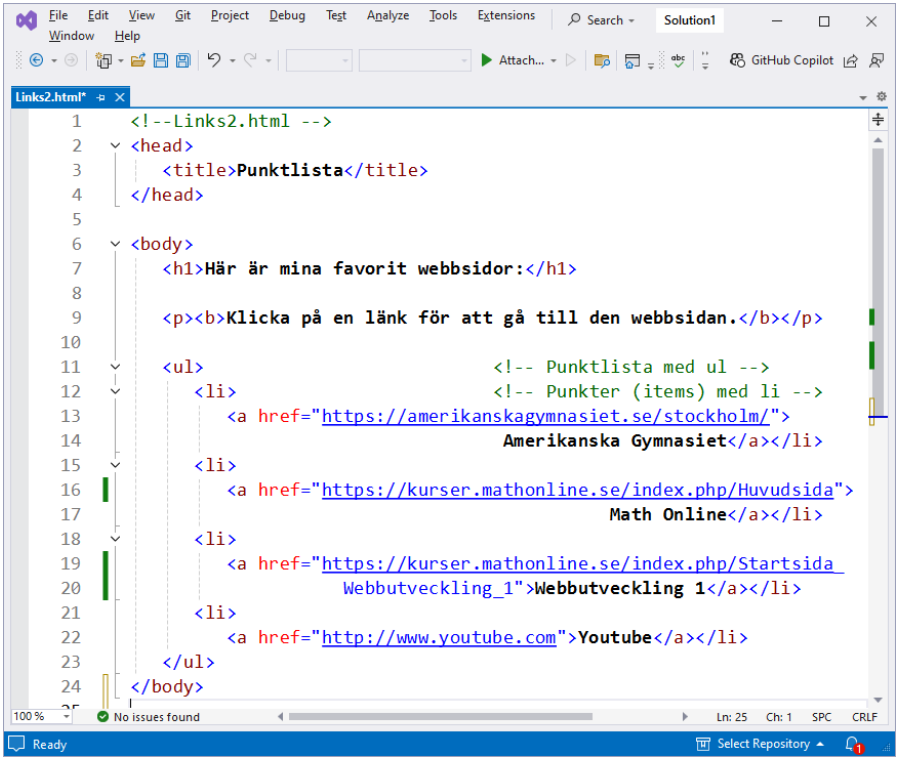
hr-*taggen*

Raden **18** i scriptet på förra sidan har inget med specialtecken att göra, utan introducerar **hr**-taggen *(horizontal rule)* som drar en horisontell linje i dokumentet, vilket kan beskådas i körresultatet ovan.

Man kan koda denna tagg med **<hr>** eller med **<hr />**. Det är lite smaksak. Anledningen är att **hr** bildar ett tomt element, för vilket gäller förkortningsregeln som nämndes tidigare (sid 30), när vi introducerade det tomma **img**-elementet (sid 29).

Samma sak är det med **br**-taggen: **
** eller **
** (sid 21).

2.5 Punktlistor



Elementet unordered list `ul`

Scriptet **`Links2`** (ovan) introducerar elementtypen *unordered list* **`ul`** för punktlista. Listan är *unordered* (oordnad) eftersom den inte ställer upp sina items med ordnande symboler, typ bokstäver eller siffror, utan med punkter.

- **Point A**
- **Point B**
- **Point C**

`ul` förser sina items med s.k. *bullets* (punkter). Därför betecknas oordnade listor även som *bulleted lists* (punktlistor). I scriptet **`Links2`** sträcker sig **`ul`**-elementet över raderna **`11-23`**.

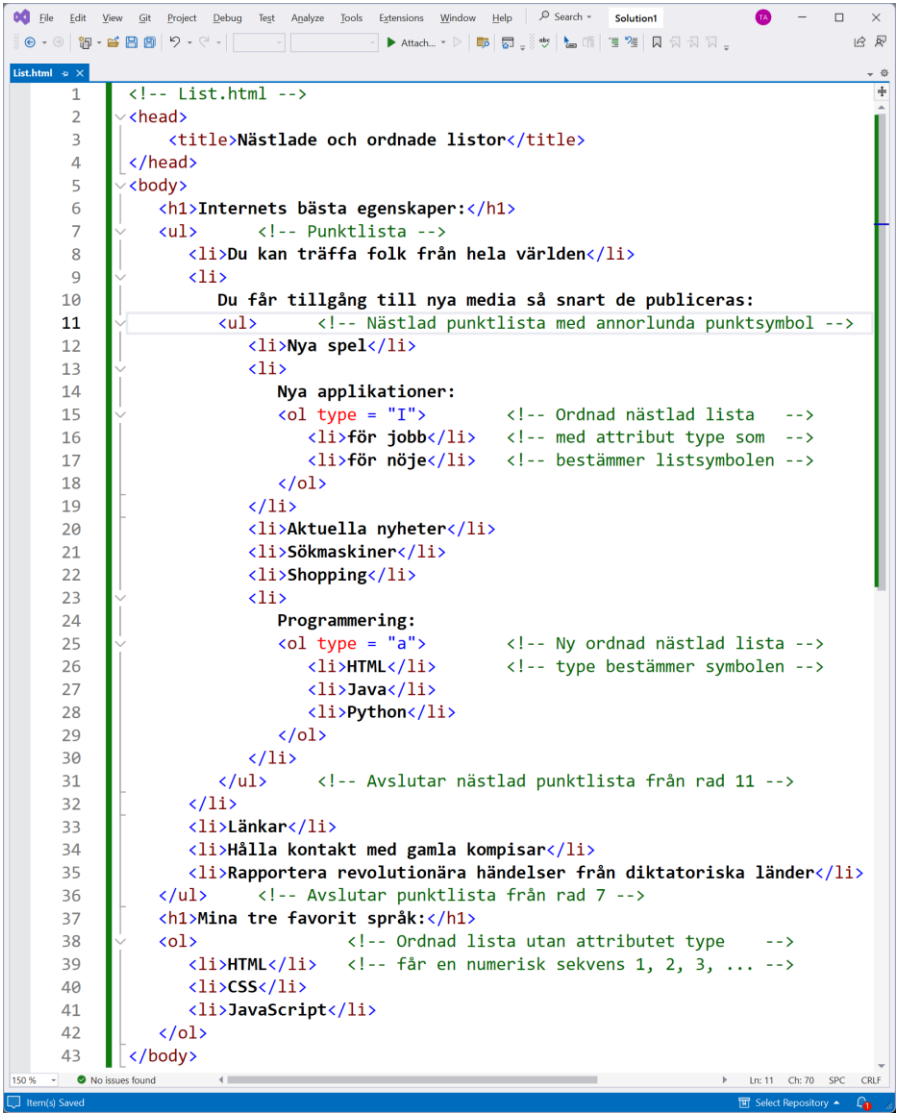
`li`*-taggen*

Dessvärre genererar **`ul`** sina items inte automatiskt. De måste kodas med en speciell tagg: **`li`**-taggen som står för *list item*. I scriptet **`Links2`** finns **`li`**-taggen på raderna **`12, 15, 18`** och **`21`**. Varje **`li`**-tagg i sin tur inleder ett element vars innehåll är en ankare.

Jämför man vårt aktuella script **Links2** med ett tidigare script **Links** (sid 26) kan man se att samma länkar där finns nu i en punktlista **ul** och att paragraferna där har bytts ut mot punktlistans items som kodas med **li**-taggen.

Körresultatet ser ut så här. De tidigare webblänkarna från scriptet **Links** finns nu i en punktlista:

2.6 Nästlade och ordnade listor



Elementet ordered list `ol`

Scriptet `List` (ovan) introducerar elementtypen *ordered list* `ol`, se exempel till höger. Listan är *ordered* (ordnad) eftersom den ställer upp sina items med ordnande symboler, t.ex. bokstäver eller siffror. I scriptet `List` börjar den första ordnade listan på rad `15` med elementtypen `ol`.

List of Fruits

1. Apple
2. Mango
3. Banana
4. Grapes
5. Orange

Attributet `type`

Elementtypen **`ol`** på rad **`15`** använder sitt attribut `type` för att bestämma listsymbolen. `type` har satts till ”I”, vilket betyder det romerska talet 1, så att listan börjar med romerskt I och fortsätter automatiskt med nästa romerska tal II, se körresultatet på nästa sida.

På rad **`25`** börjar den andra ordnade listan som sätter `type` till ”**`a`**”, så att listsymbolen blir det lilla latinska alfabetet. Den sista ordnade listan har inget `type`-attribut, så att den numeriska sekvensen 1, 2, 3, ... används by default. Varje nästlad lista blir automatiskt indragen, så att den hierarchiska strukturen blir tydlig.

Nästlade listor

Scriptet **`List`** (förra sid) innehåller ett antal nästlingar, dvs olika listtyper är inblandade och nästlade i varandra. De åstadkommer utskriften på nästa sida som återspeglar nästlingarna med olika nivåer av indragningar från vänstra kanten.

Om vi börjar titta på den överordnade strukturen i scriptet **`List`** kan vi konstatera att en oordnad punktlista **`ul`** börjar på rad **`7`** och slutar på rad **`36`**. Nästlad i den finns en annan punktlista som börjar på rad **`11`** och slutar på rad **`31`**. Intressant är nu att den nya, nästlade punktlistan automatiskt väljer en annan än den överordnade punktlistan. **`<ul>`** som börjar på rad **`7`** får en vanlig punkt (bullet), även kallad **`disc`**, som listsymbol, medan den nästade punktlistan som börjar på rad **`11`** får ihåliga ringar, kallade **`circles`**, se körresultat på nästa sida.

Nästa nästling – som inte finns med i scriptet – skulle få rutor, kallade *square*s. Det är även möjligt att ange ett `type`-attribut till punktlistor genom att använda dessa namn: ”**`disc`**”, ”**`circle`**”, eller ”**`square`**”.

De andra nästlingarna kan spåras i körresultatet av scriptet `List` som följer. Studera noga alla andra nästlingar genom att jämföra körresultatet på nästa sida med koden på förra sidan.

God programmeringsstil

Av skäl som beträffar *God programmeringsstil* måste koden återspegla nätlingarnas logik och struktur genom att göra exakt de kodindragningar du ser i scriptet `List` (sid 40). Utan dessa indragningar blir koden värdelös ur de kriterirernas synpunkt som definierar God programmeringsstil, nämligen:

- Läslighet
- Förståelighet
- Ändringsbarhet

Frågor till Kapitel 2

2.1 Med vilket HTML-element kan man skapa länkar i sitt dokument?

2.2 Vad åstadkommer elementtyperna p, b och a i HTML?

2.3 Är ankaret ett tomt HTML-element?

2.4 Vad är ett *attribut* och var någonstans placeras det i HTML-kod?

2.5 Vad gör attributet `href` i ett ankare?

2.6 Vilket är det viktigaste attributet av HTML-elementet *ankare*?

2.7 Ge tre exempel på attributvärde till attributnamnet `href`.

2.8 Hur inleds ett attributvärde till en webbadress?

2.9 Hur inleds ett attributvärde till en mailadress?

2.10 Vad är förklaringen till beteckningen *ankare* för HTML-elementet **`a`**?

2.11 Vilken egenskap får texten i dokumentet som skrivs i ett ankare?

2.12 Kan man även ange namnet på en bildfil som innehåll i ett ankare?

2.13 Vilka bildformat är de mest vanliga på webben?

2.14 Vilka är de vanligaste bildformat i ett HTML-script?

2.15 Till vilket HTML-element hör attributen `height` och `width`?

2.16 Vad gör **`img`**-elementet i HTML?

2.17 Vilket är det viktigaste attributet av HTML-elementet **`img`**?

2.18 Hur bestämmer man värdena till attributen `height` och `width`?

2.19 Vad gör attributet `alt` i **`img`**-elementet?

2.20 Var någonstans måste du lagra dina bilder när du vill hämta dem med **`img`**?

2.21 Vilka är de mest använda attribut till **`img`**-elementet?

2.22 Hur anger man bildstorleken i **`img`**-elementet?

2.23 Koda några exempel på **`img`**-elementet.

2.24 På vilken mapp pekar **`img`**-elementets attribut `src` primärt?

2.25 Vad exakt händer när man nästlar ett **`img`**-element i ett ankare?

2.26 Är **`img`**-elementet tomt? Om ja, varför? Om nej, vad är dess innehåll?

2.27 Ange två varianter att avsluta **`img`**-elementet.

2.28 Vad händer om man inte specificerar bildstorleken i img-elementet?

2.29 Formulera med egna ord regeln för att avsluta tomma element.

2.30 Ange några exempel på tomma element.

2.31 När man i navigeringsmenyn producerad av scriptet **`Nav`** (sid 32) klickar på bilden som har texten **`Tables Page`**, visas ett dokument som tidigare skapats av scriptet **`Break`** (sid 20). Förklara varför?

2.32 Ange några specialtecken som är reserverade för att skriva HTML-kod.

2.33 Ange den generella syntaxen för kodning av specialtecken i HTML.

2.34 Ange några specialtecken som introduceras i scriptet **`Contact2`** (sid 35).

2.35 Kan specialtecken även kodas med sina resp. teckenkoder istället för namn?

2.36 Vilken teckenstandard används för kodning av specialtecken med koder?

2.37 Ange ett exempel på kodning av specialtecken med teckenkoder.

2.38 Vad är *ASCII* och i vilken relation står den till *Unicode*?

2.39 Hur ritar man en rak linje i ett HTML-dokument?

2.40 Varför kan man koda **`hr`**-taggen på två olika sätt?

2.41 Kodas en oordnad punktlista med **`ul`** eller **`ol`**?

2.42 Hur ställer upp **`ul`** sina items?

2.43 Genererar **`ul`** sina items automatiskt?

2.44 Med vilken elementtyp kodar man *items* i en punktlista?

2.45 Vad gör **`li`**-taggen?

2.46 Vad är skillnaden (i kod och i utskrift) mellan scripten **`Links`** (sid 23) och **`Links2`** (sid 38)?

2.47 Vilket element har **`li`**-elementet som innehåll i scriptet **`Links2`** (sid 38)?

2.48 Var någonstans finns i scriptet **`Links2`** nästlade element?

2.49 Vilka listsymboler finns i HTML till förfogande för punktlistor?

2.50 Vad är skillnaden mellan elementtyperna **`ul`** och **`ol`**?

2.51 Vad händer med listsymbolen i punktlistor när man nästlar dem?

2.52 Hur bör den hierarkiska strukturen i nästlade listor understrykas i kod?

2.53 Kan man själv bestämma listsymbolen i listor?

2.54 Vad gör attributet `type` i listor? Kan det användas i både **`ul`** och **`ol`**?

2.55 Vad händer om man i ordnade listor om man utelämnar attributet `type`?

2.56 Med vilket värde till `type` kan man få stora bokstäver som listsymboler?

Övningar till **Kapitel 2**

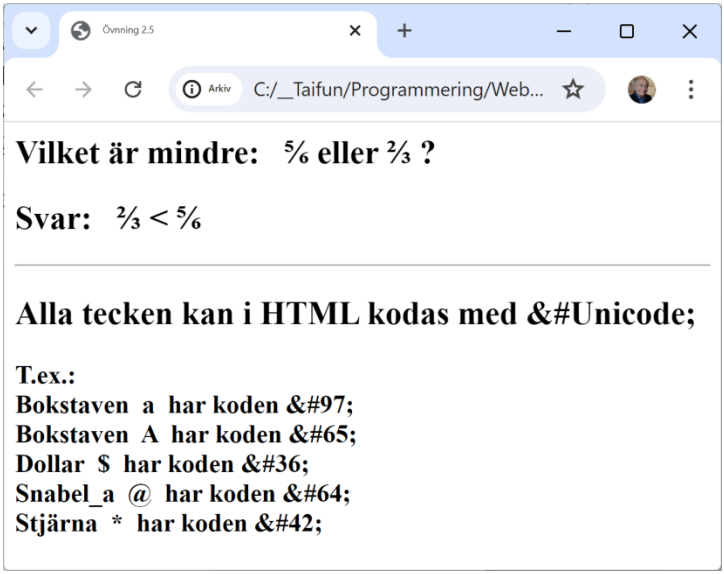
2.1 Kör scriptet **Links** (sid 26) i din webbläsare. Ladda scriptet i din favorit editor. Avlägsna alla paragraftaggar och kör igen. Ser du någon skillnad i körresultatet? Svaret är beroende på vilken webbläsare du använder. Försök att genomföra experimentet i olika webbläsare. Vilken slutsats drar du?

2.2 Scriptet **Contact** (sid 27) har på rad **14** ett radbyte i koden. Medför detta även radbyte i dokumentet? Testa! Lägg in flera tomma rader i koden och testa igen. Lägg in slutligen HTML-kod för radbyte på samma ställe och kör. Testa de olika varianterna av break-taggen.

2.3 Scriptet **Picture** (sid 29) kommer inte att visa några bilder i din webbläsare, så länge bildfilerna ***.jpg** som är angivna som värden till attributet **src** inte finns på din dator och dessutom inte på rätt plats. Vad ser du istället? Ersätt bildfilerna med egna bilder och placera dem på rätt plats. Justera bildernas storlek. Modifiera attributet **alt** meningsfullt och testa.

2.4 För scriptet **Nav** (sid 32) gäller samma sak som sades i övn 2.3 ovan om bilder. Det tillkommer här även att scriptfilerna ***.html** inte heller finns på de angivna platserna hos dig. Modifiera scriptet **Nav** med egna bilder och scriptfiler på korrekta platser, så att allt fungerar på ett meningsfullt sätt.

2.5 Modifiera scriptet **Contact2** (sid 35) så att det producerar nedanstående körresultat. Dvs använd *talkoder* istället för *namn* för att koda tecken i HTML. Förfarandet beskrivs under ***Talkoder*** på sid 36.



2.6 Vidareutveckla scriptet **`Links2`** (sid 38) så att det producerar de fem sista raderna i utskriften av övn. 2.5 (ovan) som en oordnad punktlista. Fortsätt att använda ***Talkoder*** istället för *namnkoder* för att koda specialtecken i HTML enligt förfarandet som beskrivs på sid 36.

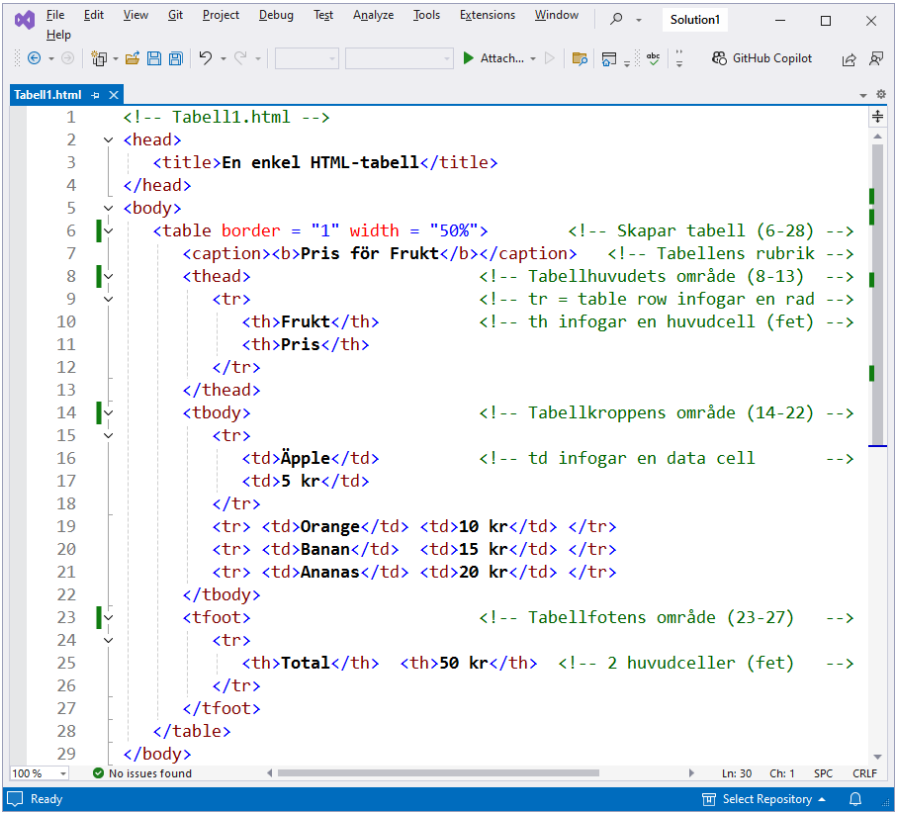
2.7 Skriv ett script **`MyList`** som liknar scriptet **`List`** (sid 40) och med hjälp av ordnade, oordnade och nästlade listor genererar en utskrift liknande den på sid 42. Ditt körresultat ska ge svar på frågan vilka som är Internets *sämsta* egenskaper samt vilka dina tre favoritämnen i skolan är.

Beakta att din kod måste uppfylla alla kriterier på *God programmeringsstil* (sid 42), speciellt när det gäller indragningarna i koden. Dessa måste överensstämma med strukturen och logiken i de nästlingar du gör med dina listor. Försök att lära dig detta genom att jämföra scriptet **`List`** med dess körresultat på på sid 42.

Kapitel 3

Mer om HTML

3.1 Tabeller



`table`-*attributet* `border`

På rad **6** skapar **`table`**-taggen en tabell som har två attribut, av vilka attributet `border` ritar en ram, dvs kantlinjerna till tabellen. Attributets värde bestämmer ramens typ och stil. Utan att definiera attributet `border` inramas tabellen inte alls, vilket inte ger intrycket av en ”riktig” tabell. I scriptet ovan har attributet `border`:s värde satts till **`1`** vilket ritar den ram som ses på bilden ovan. Prova gärna andra värden **`0`**, **`2`**, **`3`**, … för att se andra typer av kantlinjer. Enheten är pixlar. **`0`** betyder ingen ram. Det andra attributet `width` bestämmer tabellens bredd. I scriptet **`Table1`** är breddens enhet angiven i procent av webbläsarens bredd. För att få en uppfattning om

den faktiska storleken, prova att köra scriptet med olika värden för `width` och olika storlekar på webbläsaren.

Elementet `table`

Tabellen ovan initieras i scriptet **`Tabell1`** på förra sidan av starttaggen **`<table>`** på rad **`6`** och avslutas av sluttaggen **`</table>`** på rad **`28`**. Som bilden visar har tabellen 6 rader och 2 kolumner och en rubrik. Rubriken skapas med taggen **`caption`** och hamnar utanför (ovanpå) tabellen. Den första och sista raden är tabellens huvud *(head)* och fot *(foot)* och har annorlunda formateringar: fet stil och centrerad, vilket beror på att deras koder placerats i speciella taggar. Dvs den första raden kodas i tabellhuvudets område **`thead`** med **`th`**-celler (rad **`10-11`**). Och den sista raden kodas i tabellfotens område **`tfoot`**, även den med **`th`**- celler (rad **`25`**). Båda omgärdar tabellens kropp. Ur ett helhetsperspektiv kan vi konstatera:

Den överordnade strukturen i tabellen består av följande tre delar:

- Huvudet *(head)* som definieras med ett **`thead`**-element (raderna **`8-13`**)
- Kroppen *(body)* som definieras med ett **`tbody`**-element (raderna **`14-22`**)
- Foten *(foot)* som definieras med ett **`tfoot`**-element (raderna **`23-27`**).

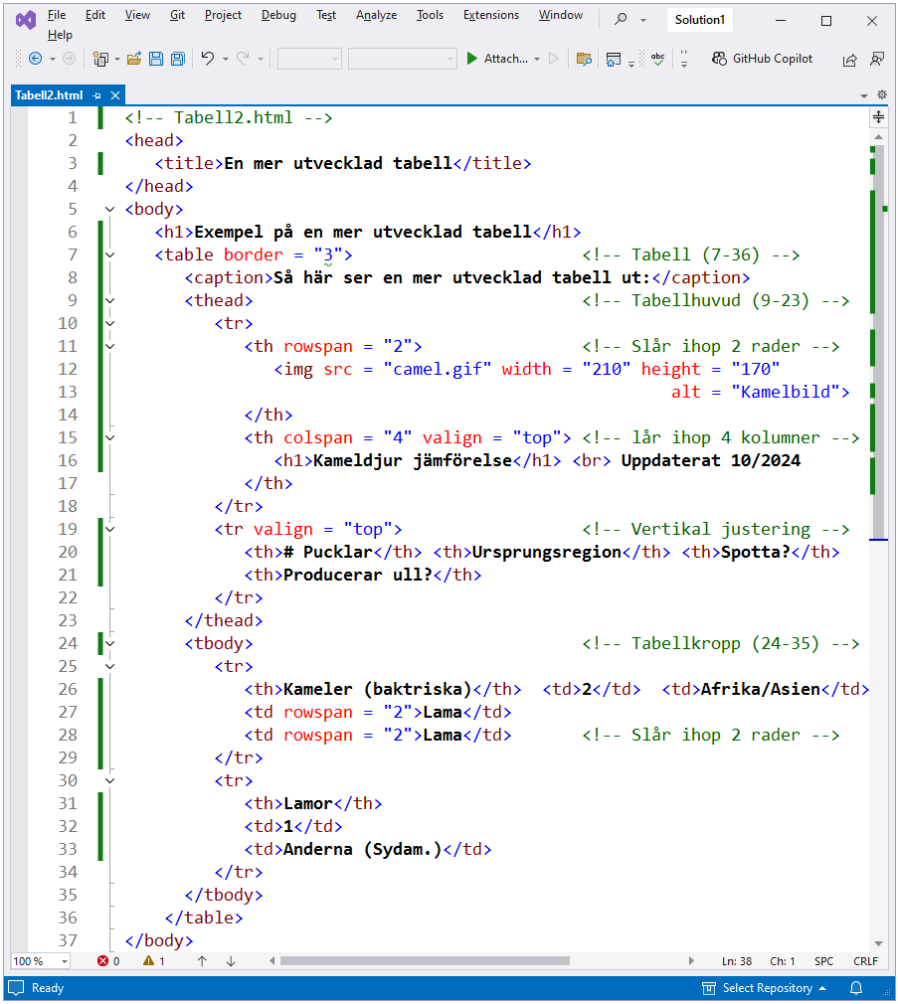
Elementen `th` *och* `td`

`th` står för *table header column* och **`td`** för *table data*. Båda skapar kolumner.

Varje **`tr`**-element (t.ex. rad **`9-12`**) definierar en enskild tabellrad, medan kolumnerna i huvudets område skapas med ett **`th`**-element som formaterar texten i fet stil och centrerad.

Tabellkroppens område (rad **`14-22`**) innehåller tabellens primära data och är definierad med ett **`tbody`**-element. Även där skapas raderna med **`tr`**-element, medan kolumnerna kodas med **`td`**. I vanliga dataceller som skapas med **`td`** sker ingen formatering.

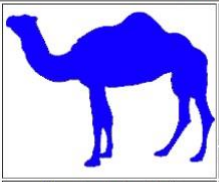
3.2 En mer utvecklad tabell



På rad **7** skapas en tabell med **table**-taggen vars attribut **border** får värdet ”3”, vilket leder till kantlinjer man ser i körresultatet till höger.

Exempel på en mer utvecklad tabell

Så här ser en mer utvecklad tabell ut:



Kameldjur jämförelse

Uppdaterat 10/2024

	# Pucklar	Ursprungsregion	Spotta?	Producerar ull?
Kameler (baktriska)	2	Afrika/Asien	Lama	Lama
Lamor	1	Anderna (Sydam.)		

Två nyheter finns i denna tabell resp. i scriptet **Tabell2** (ovan) som skapar den:

th- och td-a ttributen rowspan & colspan

I regel formateras tabellceller så att de kan visa data som de innehåller. Men utvecklare kan ändra på det och skapa större dataceller med attribut som de skriver till cell-taggarna **td** (*table data*) och **th** (*table header column*) och. Dessa attribut heter rowspan och colspan. Värden de får, bestämmer antalet rader resp. kolumner som reserveras för en cell. Två exempel har vi för detta i scriptet **Tabell2** på förra sidan:

- **th**-elementet på raderna **11-14** använder attributet rowspan = ”2” för att tillåta cellen, som innehåller kamelbilden, att använda 2 vertikala grannceller. Dvs det slår ihop *(spans)* 2 rader.

- **th**-elementet på raderna **15-17** använder attributet colspan = ”4” för att tillåta cellen, som innehåller texten **Kameldjur jämförelse** och **Uppdaterad 10/2024**, att använda 4 horisontella grannceller, dvs att slå ihop *(span)* 4 kolumner.

  Motsvarande gäller för **td**-elementen på raderna **27-28**.

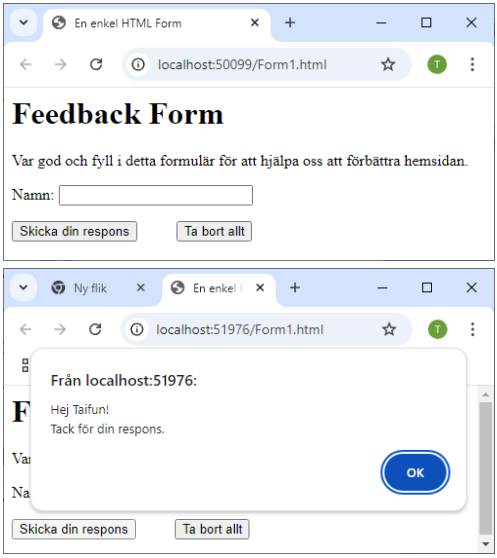
3.3 HTML Forms

```
1   <!-- Form1.html
2        Vår första HTML Form (Formulär på svenska) -->
3   <title>En enkel HTML Form</title>
4   <h1>Feedback Form</h1>
5   <p>Var god och fyll i detta formulär för att hjälpa oss att förbättra hemsidan.</p>
6
7   <form>                      <!-- Initierar formen -->
8      <p>
9         <label>
10           Namn:              <!-- <input type = "text"> infogar en text box -->
11           <input type="text" id="name_tb" size="23">
12        </label>
13     </p>
14
15     <p>
16        <!-- input type="submit" och type="reset" infogar knappar -->
17        <input type="submit" id="submit_btn"
18               value="Skicka din respons" onClick="showData()">  <!-- Anrop av -->
19                                       <!-- JS funktion -->
20        <input type="reset" id="reset_btn" value="Ta bort allt">
21     </p>
22  </form>
23
24  <script>                    // Kommentar i JavaScript
25     function showData()      // JavaScript funktion
26     {
27        user = document.getElementById("name_tb").value;
28        alert("Hej " + user + "!" + "\nTack för din respons.")
29     }
30  </script>
```

Elementet `form`

Scriptet `Form1` ovan genererar det övre dokumentet till höger som kallas *formulär* (eng. *form*). En form består i regel av ett antal klickbara knappar, ett antal textboxar med tillhörande labels (etiketter) och förklarande text. Man kan säga, en form är ett grafiskt gränssnitt som uppmanar användaren att fylla i vissa delar av den och skicka tillbaka formen.

Skriver man sitt namn i textboxen `Namn` och klickar på knappen Skicka din respons visas det undre dokumentet.



Oftast kommer en sådan form från en server på nätet och går också tillbaka till den, efter att användaren har fyllt i den. Vi kan inte genomföra den här kommunikationen utan bara simulera den, för att lära oss hur man kodar former i HTML. Istället för att skicka formuläret till en server skriver vi ut ett svar med hjälp av en JavaScript funktion.

JavaScript funktion

I scriptet **Form1**, raderna **24-30**, har vi definierat funktionen `showData()` som är kodad i *JavaScript* och genererar en meddelanderuta som kallas *alert box*, se förra sidan. JavaScript är ett scriptspråk som kan bäddas in i HTML och har element av de universella programmeringsspråken som saknas i HTML. **<script>**-taggen på rad **24** inleder JavaScript-kod och sluttaggen på rad **30** avslutar den. Kommentar i JavaScript kodas med **//** (raderna **24** & **25**). Närmare bestämt inleds en *radkommentar* med **//** som inte behöver avslutas då den gäller till slutet av raden.

På rad **18** anropas funktionen `showData()` i **input**-attributet `onClick`, närmare bestämt i attributets värde, vilket genererar alert-boxen som man ser på förra sidan (nedan till höger). Anropet sker när användaren klickar på knappen Skicka din respons i formuläret. Detaljerna i definitionen till JavaScript-funktionen `showData()` kommer att tas upp senare resp. i kursen Webbutveckling 2.

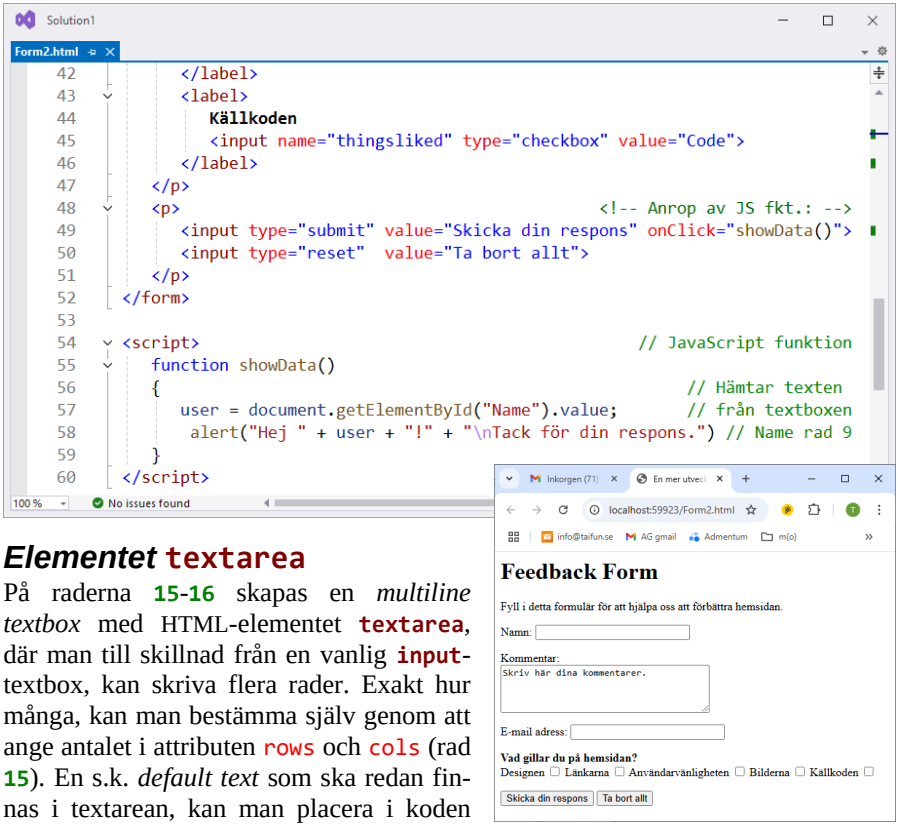
Textboxar med elementet `input`

Det är elementet **input** med sina attribut som i huvudsak designar formen, genom att skapa textboxar och knappar på formen. Det är attributet `type` som bestämmer vilken typ av **input** det ska bli. Medan `type` = "`text`" (rad **11**) infogar en textbox, skapar `type` = "`submit`" (rad **17**) en knapp som skickar formen och `type` = "`reset`" (rad **20**) en annan knapp som tömmer allt.

3.4 En mer utvecklad Form

I förra avsnitt introducerades grundläggande HTML Forms med enkla element som **`form`**, **`input`** samt deras olika attribut. I detta avsnitt vill vi fortsätta med lite mer avancerade element. Scriptet **`Form2`** innehåller elementet **`textarea`** och nya varianter av textboxar av typ `password` (maskerad textbox) samt `checkbox`:

• • • (Koden fortsätter på nästa sida)



Elementet `textarea`

På raderna **15-16** skapas en *multiline textbox* med HTML-elementet **`textarea`**, där man till skillnad från en vanlig **`input`**-textbox, kan skriva flera rader. Exakt hur många, kan man bestämma själv genom att ange antalet i attributen `rows` och `cols` (rad **15**). En s.k. *default text* som ska redan finnas i textarean, kan man placera i koden som innehåll till elementet **`textarea`**, dvs mellan start- och sluttaggen (rad **16**).

Maskerad textbox

På rad **22** skapas med elementet **`input`** en – till att börja med – vanlig textbox. Men genom att ge dess attribut `type` värdet "`password`" blir texten som man sedan skriver i den i dokumentet, maskerad. Det innebär att alla bokstäver man matar in, ersätts i dokumentet med asterisk (*), medan textboxen sparar den inmatade texten i sitt variabelnamn (i `id`-attributets värde). Det är så man vill ha det i en textbox av typ `password`. Här har man valt att inte offentligt visa mailadressen.

Checkboxar

En checkbox är en liten ruta (□) som kan bockas. På rad **29** skapas den första checkboxen i scriptet **`Form2`** (sid 55). För det används en vanlig textbox. Men genom attributet `type` = "`checkbox`" blir denna textbox en *checkbox*. I ett formulär kan man bocka *flera* checkboxar.

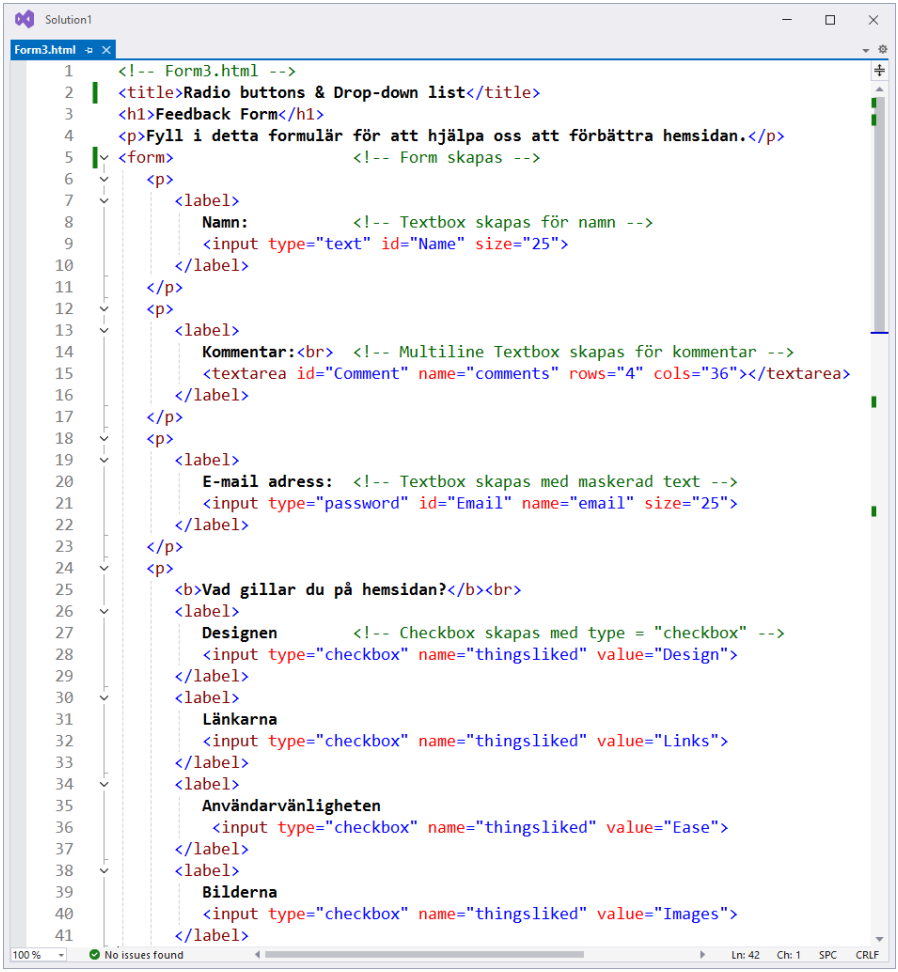
På raderna **33**, **37**, **41** och **45** skapas ytterligare 4 checkboxar. Deras namn bestäms i elementet **`label`** som omgärdar checkboxarna. I dokumentet (ovan) visas de under rubriken **Vad gillar du på hemsidan?** i nästsista raden.

3.5 Radioknappar och Dropp-down list

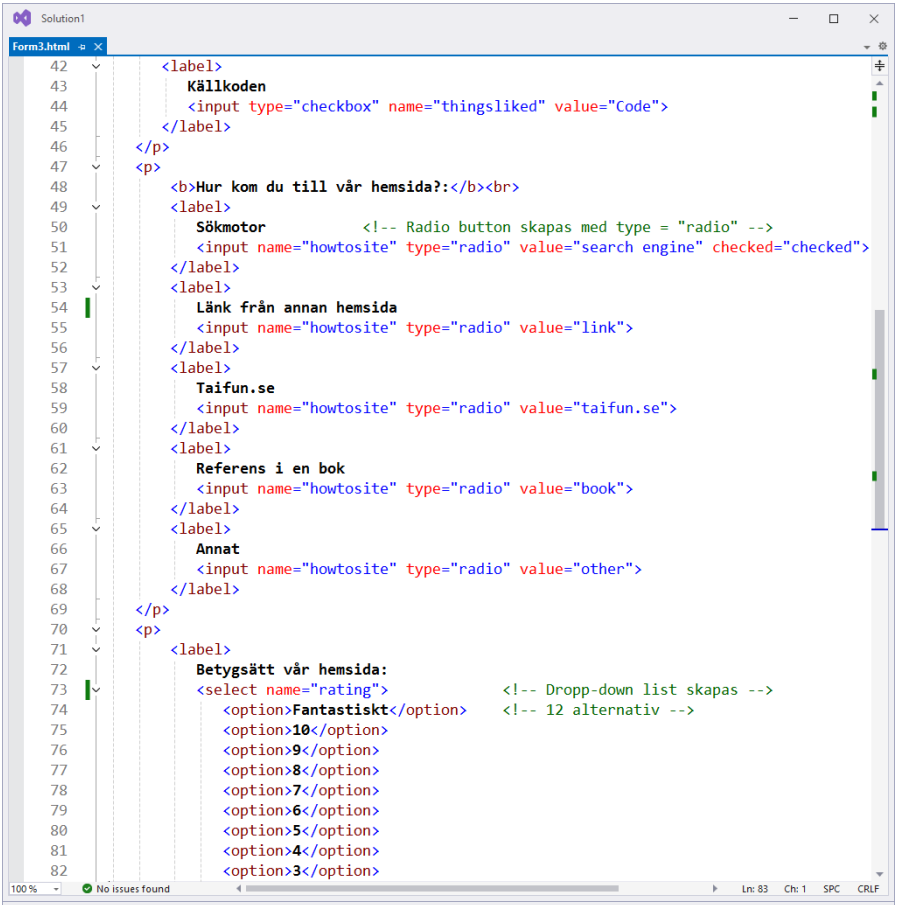
Den nya, utvidgade formen **Form3** nedan tar över **Form2**, modifierar den lite och lägger till två nya element: ***radioknappar*** och en ***dropp-down list***.

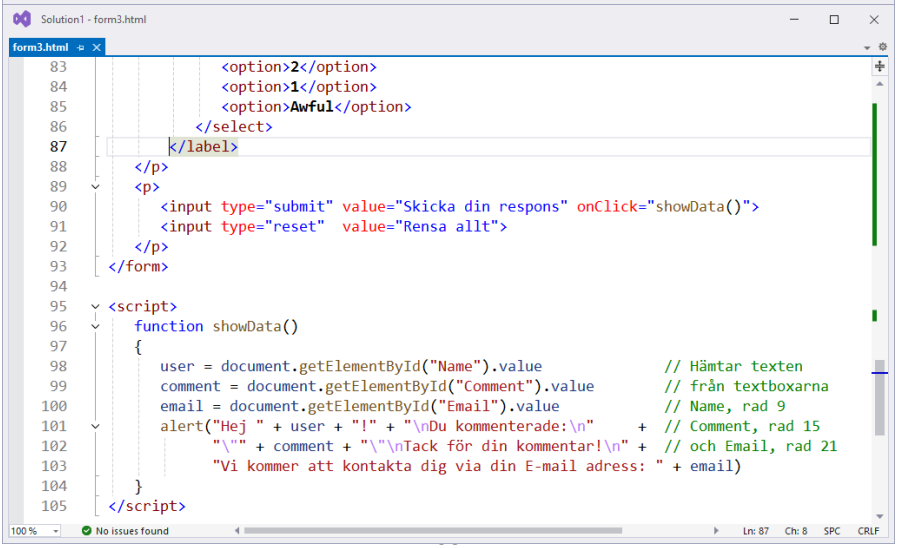
Radioknappar

Till skillnad från checkboxar (□), se förra avsnitt, ser en radioknapp *(radio button)* ut som en ring (○). I ett formulär kan man bocka *flera* checkboxar, medan en radioknapp kan markeras endast en gång. I scriptet **Form3** nedan skapas den första radioknappen på rad **51** (sid 55) med **<input** type = "radio" ... **>** .



. . . (OBS! Koden fortsätter på nästa sida.)


Solution1
Form3.html
42 <label>
43 Källkoden
44 <input type="checkbox" name="thingsliked" value="Code">
45 </label>
46
47
48 Hur kom du till vår hemsida?:
49 <label>
50 Sökmotor <!-- Radio button skapas med type = "radio" -->
51 <input name="howtosite" type="radio" value="search engine" checked="checked">
52 </label>
53 <label>
54 Länk från annan hemsida
55 <input name="howtosite" type="radio" value="link">
56 </label>
57 <label>
58 Taifun.se
59 <input name="howtosite" type="radio" value="taifun.se">
60 </label>
61 <label>
62 Referens i en bok
63 <input name="howtosite" type="radio" value="book">
64 </label>
65 <label>
66 Annat
67 <input name="howtosite" type="radio" value="other">
68 </label>
69
70
71 <label>
72 Betygsätt vår hemsida:
73 <select name="rating"> <!-- Dropp-down list skapas -->
74 <option>Fantastiskt</option> <!-- 12 alternativ -->
75 <option>10</option>
76 <option>9</option>
77 <option>8</option>
78 <option>7</option>
79 <option>6</option>
80 <option>5</option>
81 <option>4</option>
82 <option>3</option>
100 %
No issues found
Ln: 83 Ch: 1 SPC CRLF

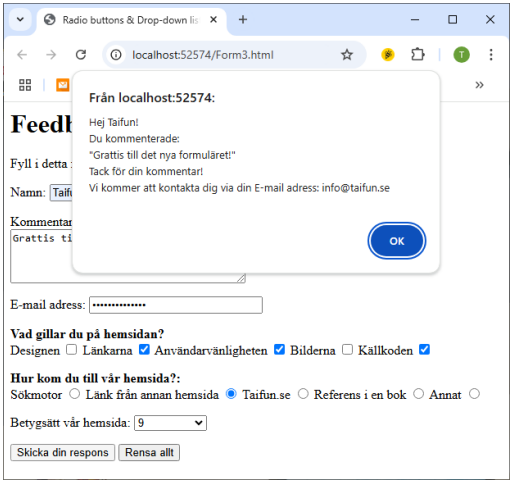

Solution1 - form3.html
form3.html
83 <option>2</option>
84 <option>1</option>
85 <option>Awful</option>
86 </select>
87 </label>
88
89
90 <input type="submit" value="Skicka din respons" onClick="showData()">
91 <input type="reset" value="Rensa allt">
92
93 </form>
94
95 <script>
96 function showData()
97 {
98 user = document.getElementById("Name").value // Hämtar texten
99 comment = document.getElementById("Comment").value // från textboxarna
100 email = document.getElementById("Email").value // Name, rad 9
101 alert("Hej " + user + "!" + "\nDu kommenterade:\n" + // Comment, rad 15
102 "\"" + comment + "\"\nTack för din kommentar!\n" + // och Email, rad 21
103 "Vi kommer att kontakta dig via din E-mail adress: " + email)
104 }
105 </script>
100 %
No issues found
Ln: 87 Ch: 8 SPC CRLF

Dropp-down list

I scriptet **Form3** (ovan) skapas en *dropp-down list* med:

```
<select name = "rating" ... >
```

(rad **73**). I dokumentet (till höger) ser man den på nästsista raden höger om texten **Betygsätt vår hemsida:** . Det är en rullgardinsmeny som låter användaren, efter man har klickat på den lilla pilen, välja mellan ett antal alternativ som i koden skapas med **option**-taggen inbäddad i **select** (raderna **74-85**). På så sätt ges användaren möjligheten att betygsätta hemsidan med 12 olika alternativ.



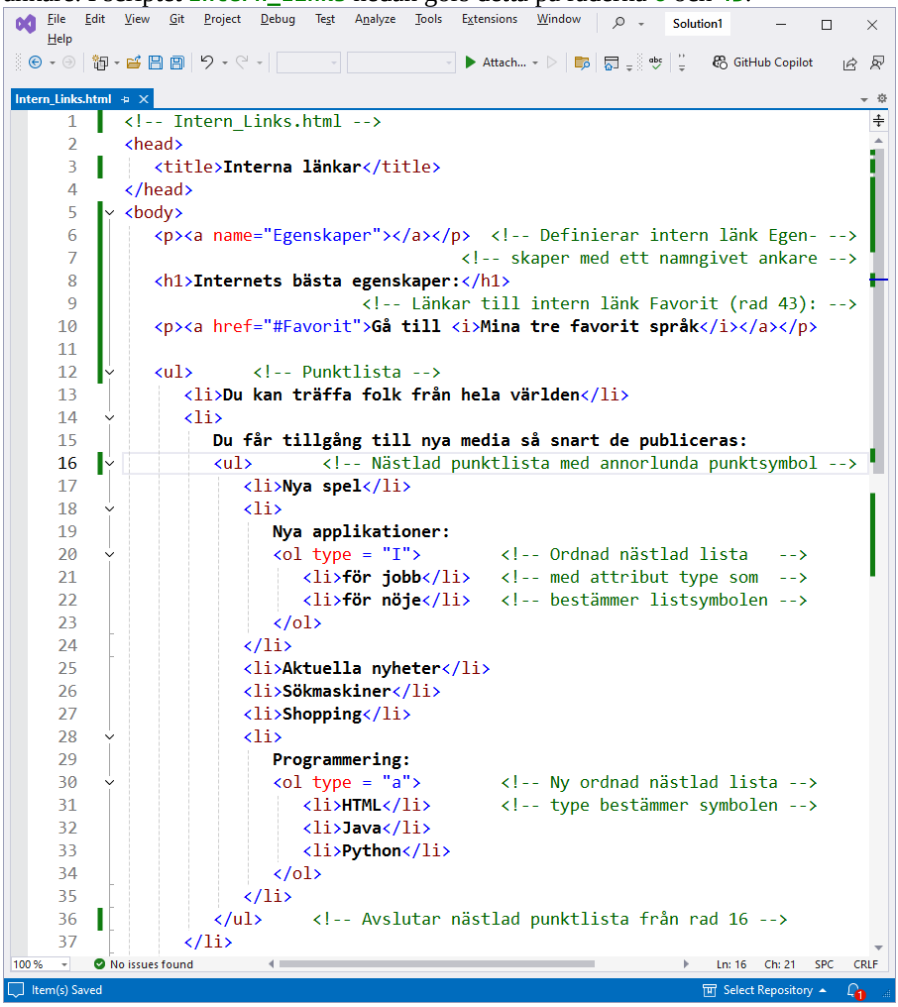
Dessutom har implementerar scriptet **Form3** övn 3.5:s (sid 71) krav på att hämta data inte bara från textboxen Namn (rad **9**), utan även från textarean Kommentar (rad **15**) och från textboxen E-mail adress (rad **22**) samt visa dessa data i JavaScript funktionens alert-box, se utskriften ovan. Koden för dessa tillägg finns på de angivna raderna.

3.6 Interna länkar

I detta avsnitt tar vi scriptet **List** (sid 40) som har körresultatet på sid 41 och modifierar den genom att lägga till två *interna länkar*. Vanliga länkar hade vi lärt känna i scriptet **Links** (sid 26) som vi kodade med HTML-elementet ankare `a` (sid 27). Medan vanliga länkar går till andra dokument, hoppar interna länkar mellan olika ställen i *samma* dokument, vilket är av intresse i stora dokument. Interna länkar skapas med ankare som har ett namn, s.k.:

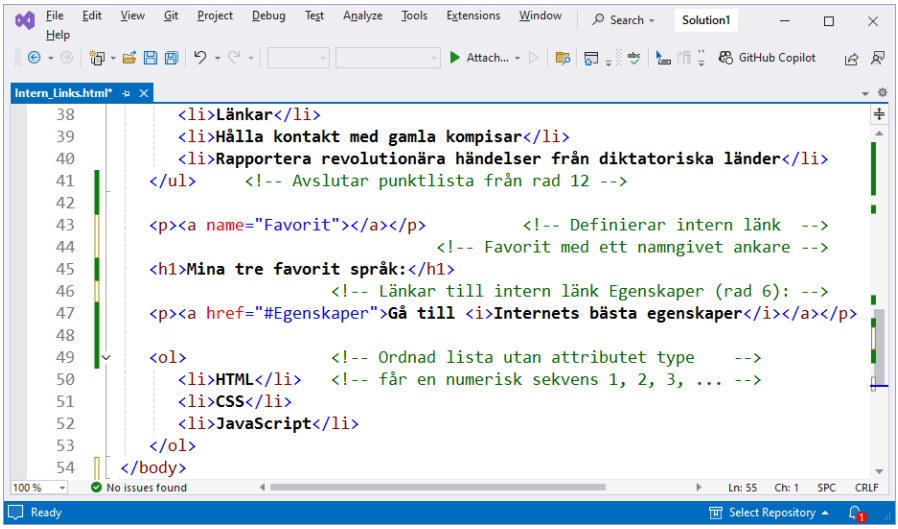
Namngivna ankare

Genom att definiera ett värde till attributet `name` i ett ankare, skapas ett namngivet ankare. I scriptet **Intern_Links** nedan görs detta på raderna **6** och **43**.




```
1  <!-- Intern_Links.html -->
2  <head>
3     <title>Interna länkar</title>
4  </head>
5  <body>
6     <p><a name="Egenskaper"></a></p>  <!-- Definierar intern länk Egen- -->
7                                       <!-- skaper med ett namngivet ankare -->
8     <h1>Internets bästa egenskaper:</h1>
9                                 <!-- Länkar till intern länk Favorit (rad 43): -->
10    <p><a href="#Favorit">Gå till <i>Mina tre favorit språk</i></a></p>
11
12    <ul>      <!-- Punktlista -->
13       <li>Du kan träffa folk från hela världen</li>
14       <li>
15          Du får tillgång till nya media så snart de publiceras:
16          <ul>       <!-- Nästlad punktlista med annorlunda punktsymbol -->
17             <li>Nya spel</li>
18             <li>
19                Nya applikationer:
20                <ol type = "I">        <!-- Ordnad nästlad lista   -->
21                   <li>för jobb</li>   <!-- med attribut type som  -->
22                   <li>för nöje</li>   <!-- bestämmer listsymbolen -->
23                </ol>
24             </li>
25             <li>Aktuella nyheter</li>
26             <li>Sökmaskiner</li>
27             <li>Shopping</li>
28             <li>
29                Programmering:
30                <ol type = "a">        <!-- Ny ordnad nästlad lista -->
31                   <li>HTML</li>       <!-- type bestämmer symbolen -->
32                   <li>Java</li>
33                   <li>Python</li>
34                </ol>
35             </li>
36          </ul>     <!-- Avslutar nästlad punktlista från rad 16 -->
37       </li>
```

• • • (Koden fortsätter på nästa sida)



Kör man scriptet **Intern_Links** ovan, måste man minska utskriftsfönstret *innan* man klickar på länkarna, för att se bytet av platsen i dokumentet.

Den interna länken som skapas med ankare på rad **10**, leder till den del av dokumentet som är rubricerad med **Mina tre favorit språk**. Medan länken som skapas med ankare på rad **47**, leder till den del av dokumentet som är rubricerad med **Internets bästa egenskaper**.

Observera att varje intern länk har *två* olika platser i koden: en gång definitionen av den plats i dokumentet som länken ska leda *till* med ett ankare och attributet

`name` (raderna `6` och `43`), en annan gång referensen som man ska klicka på med ett ankare och attributet `href` (raderna `10` och `47`). Det är som i all kommunikation: det finns en sändare och mottagare. Först måste mottagarens plats definieras (`name`), sedan kan sändaren länka till mottagaren (`href`).

3.7 Interna länkar i andra dokument

Scriptet **Intern_Links** i förra avsnitt (sid 60) definierade interna länkar som hoppade mellan olika ställen i *samma* dokument.

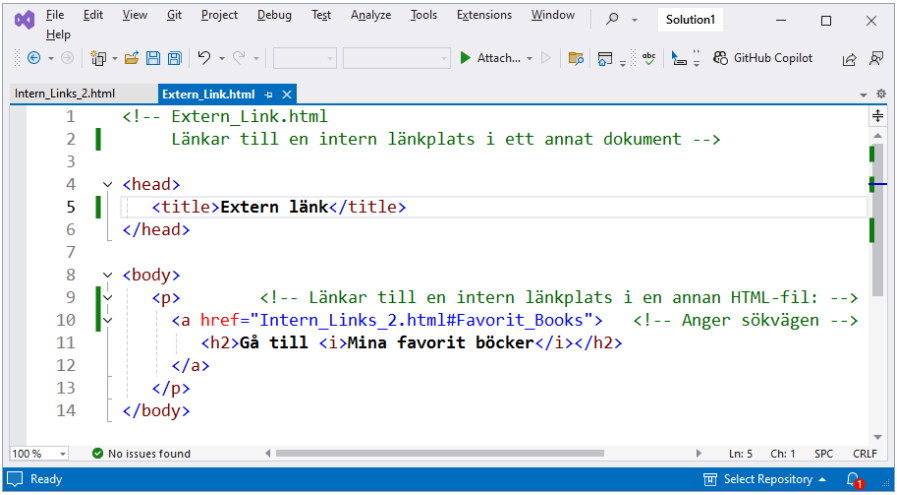
Här ska vi skapa länkar som går till interna länkar i andra dokument. Scriptet **Extern_Link** nedan skapar ett ankare vars attribut **href** refererar till en länkplats som är definierad i ett annat dokument. Länkplatsen heter **Favorit_Books** och är definierad i filen **Intern_Links_2.html**.

Sökväg som referens

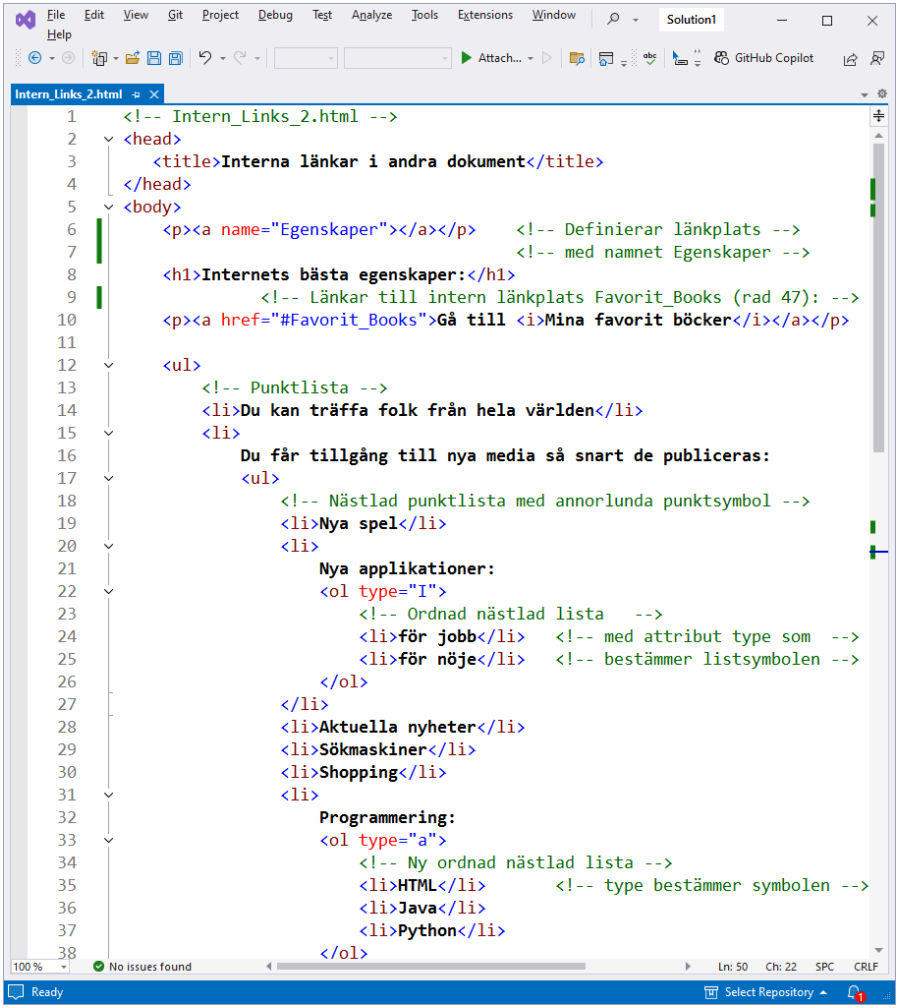
På rad **10** nedan skapas ett ankare med attributet **href** vars värde är sökvägen till länkplatsen **Favorit_Books**. Observera sökvägens syntax:

```
Intern_Links_2.html#Favorit_Books
```

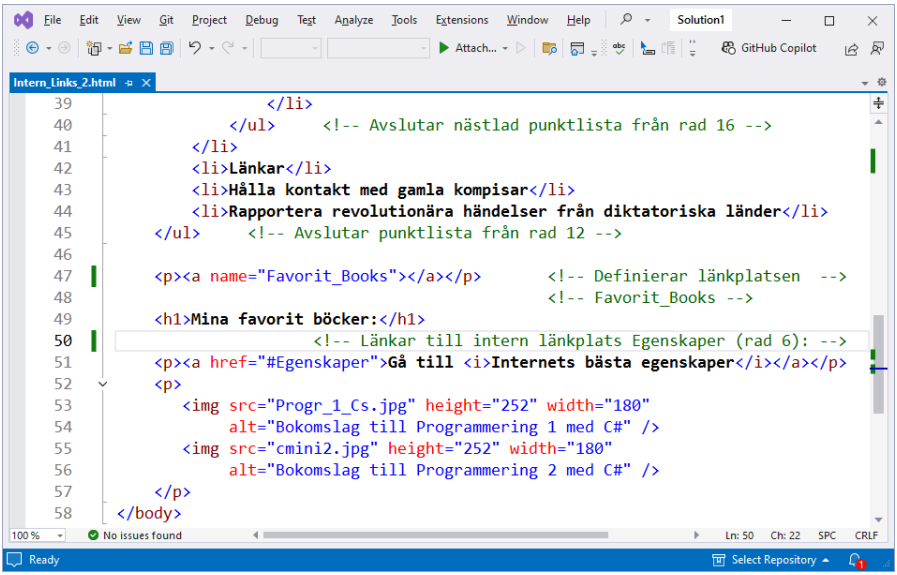
Filen **Intern_Links_2.html** måste placeras i samma mapp som scriptet **Extern_Link** nedan. Väljer man en annan plats måste motsvarande sökväg anges.



Scriptet **Intern_Links_2** nedan som scriptet **Extern_Link** hänvisar till är en modifikation av scriptet **Intern_Links** i förra avsnitt (sid 60). Den första delen med rubriken **Internets bästa egenskaper** är oförändrad. I den andra delen har vi ersatt **Mina favorit språk** med **Mina favorit böcker** och döpt om namnet på ankaret från **Favorit** till **Favorit_Books**.

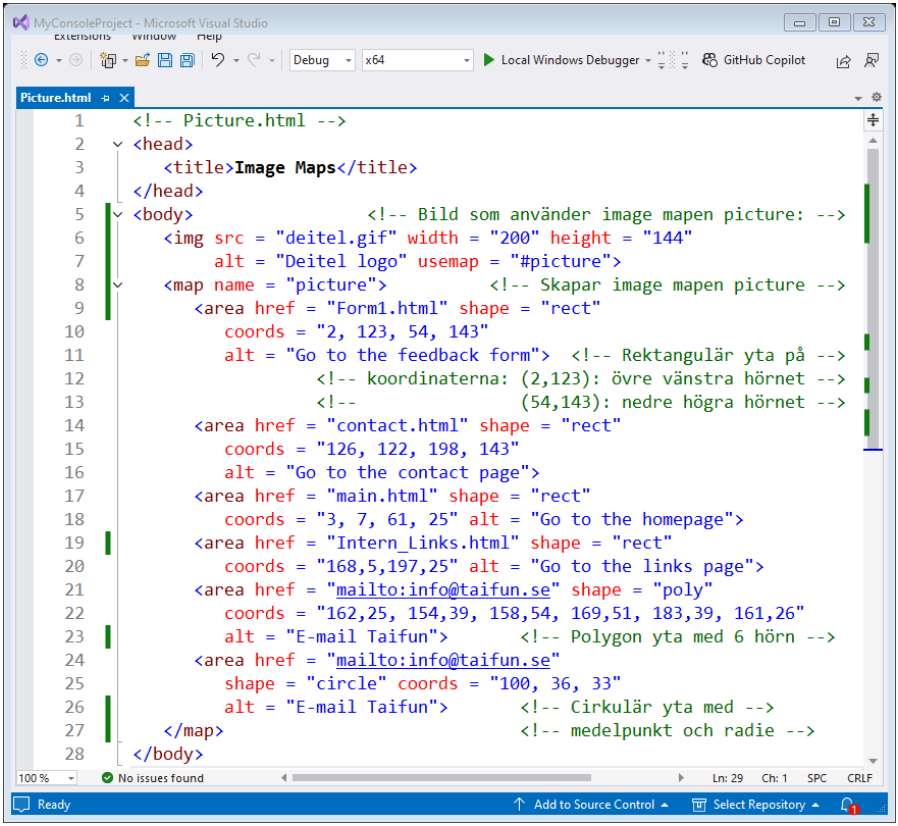


. . . (Koden fortsätter på nästa sida)


File Edit View Git Project Debug Test Analyze Tools Extensions Window Help
Solution1
Attach...
GitHub Copilot
Intern_Links_2.html
39
40 <!-- Avslutar nästlad punktlista från rad 16 -->
41
42 Länkar
43 Hålla kontakt med gamla kompisar
44 Rapportera revolutionära händelser från diktatoriska länder
45 <!-- Avslutar punktlista från rad 12 -->
46
47 <!-- Definierar länkplatsen -->
48 <!-- Favorit_Books -->
49 <h1>Mina favorit böcker:</h1>
50 <!-- Länkar till intern länkplats Egenskaper (rad 6): -->
51 Gå till Internets bästa egenskaper
52
53 <img src="Progr_1_Cs.jpg" height="252" width="180"
54 alt="Bokomslag till Programmering 1 med C#" />
55 <img src="cmini2.jpg" height="252" width="180"
56 alt="Bokomslag till Programmering 2 med C#" />
57
58 </body>
100 %
No issues found
Ln: 50 Ch: 22 SPC CRLF
Ready
Select Repository

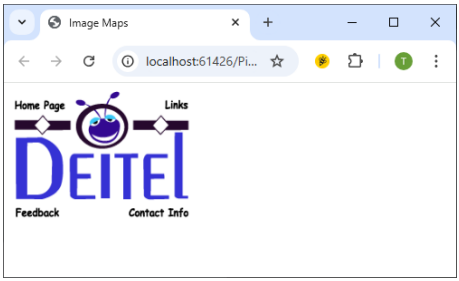
3.8 Image maps

Image maps, på svenska bildkartor, är ett verktyg i HTML som skapar klickbara ytor på en bild. Ytorna är osynliga och kallas för *hotspots*. En image map läggs som en osynlig karta på en bild eller *”bilden använder en image map”*. Så, själva bilden måste skiljas från image mapen. I scriptet **`Picture`** nedan använder bilden som skapas med **`img`** på raderna **`6-7`**, image mapen som med **`map`** skapas på raderna **`8-27`**.



Elementet `map`

Image maps definieras med HTML-elementet **`map`**. I scriptet ovan skapas en image map och döps med attributet **`name`** till **`picture`** (rad **`8`**). Bilden som skapas på raderna **`6-7`**, använder denna map, genom att referera till den med attributet **`usemap`** (rad **`7`**). Bilden till höger har sex klickbara ytor (hotspots).



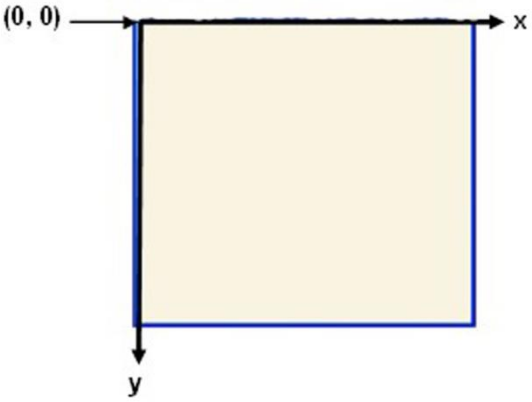
Elementet `area`

De klickbara ytorna i image mapen `picture` skapas med HTML-elementet `area` som nästlas in i `map`-elementet. Så, en `map` kan innehålla en eller flera `area`:s. Ytorna kan ha olika former. Dessa anges med attributet `shape`. I scriptet `Picture` har vi fyra rektanglar, en cirkel och en sexhörning (hexagon). `area`-attributet `href` refererar till de dokument som ska visas, när man klickar på de hotspots i image mapen.

De geometriska figurers position och storlek som är definierade med attributet `shape`, anges i attributet `coords`. För rektangeln anges det övre vänstra och nedre högra hörnets koordinater (raderna `10`, `15`, `18 och 20`). För sexhörningen anges de sex hörnens koordinater (rad `22`) och för cirkeln medelpunktens koordinater samt radiens längd. Hur koordinaterna definieras förklaras nedan:

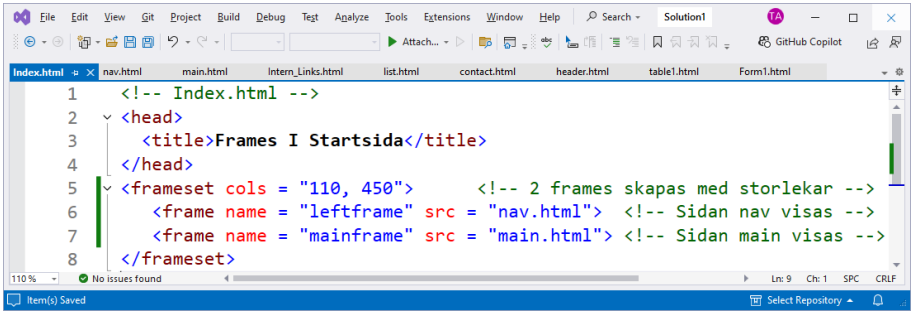
Koordinatsystem för geometriska figurer

För att kunna rita geometriska figurer och placera dem behöver vi ange deras storlek och position, vilket förutsätter ett koordinatsystem på den grafiska ritytan. Följande koordinatsystem är automatiskt definierat i alla grafiska miljöer: Origo dvs positionen (0, 0) är placerad i fönstrets övre vänstra hörn. x-koordinaten växer i horisontell led åt höger och y-koordinaten i vertikal led nedåt:



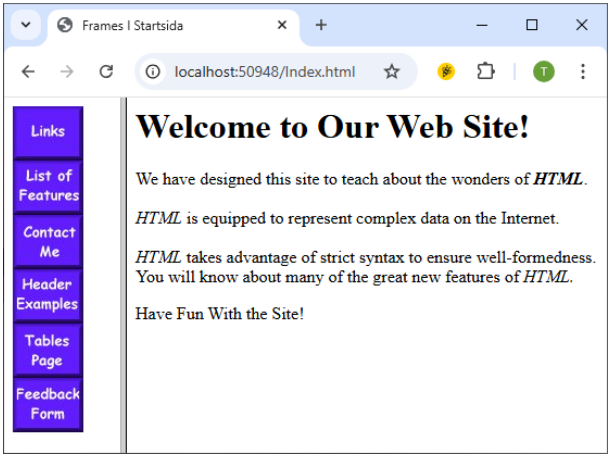
Denna bild borde man ha i minnet när man arbetar med koordinater. Enheten är en *pixel* som står för *picture element*. En pixel är en digital bilds minsta komponent – datorgrafikens atom så att säga. Pixelns faktiska storlek är beroende av den aktuella tekniska utrustningen dvs bildskärmen och dess upplösning. Vill vi placera en punkt i koordinatsystemet ovan anges punktens x-koordinat som antalet pixlar som den är borta från formens vänstra kant. Punktens y-koordinat anges som antalet pixlar som den är borta från formens övre kant.

3.9 Framesets



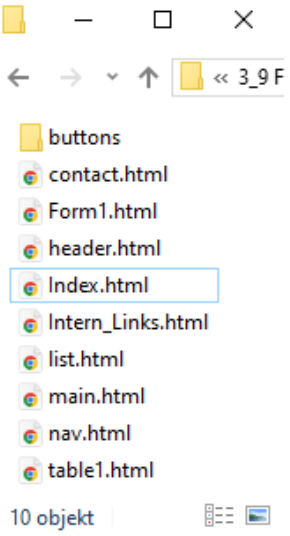
Elementet `frame-set`

Scriptet `Index` producerar i kombination med andra script som vi återkommer till, sidan till höger. Först skapar elementet `frame-set` på rad `5` två frames, dvs delar webbsidan i två kolumner, den ena av storleken **`110`**, den andra av **`450`** pixlar. Hela webbsidan (till höger) består av *två* frames som bildar ett *frameset*. Med navigeringsmenyn i den vänstra framen kan man komma till webbsidans andra delar.

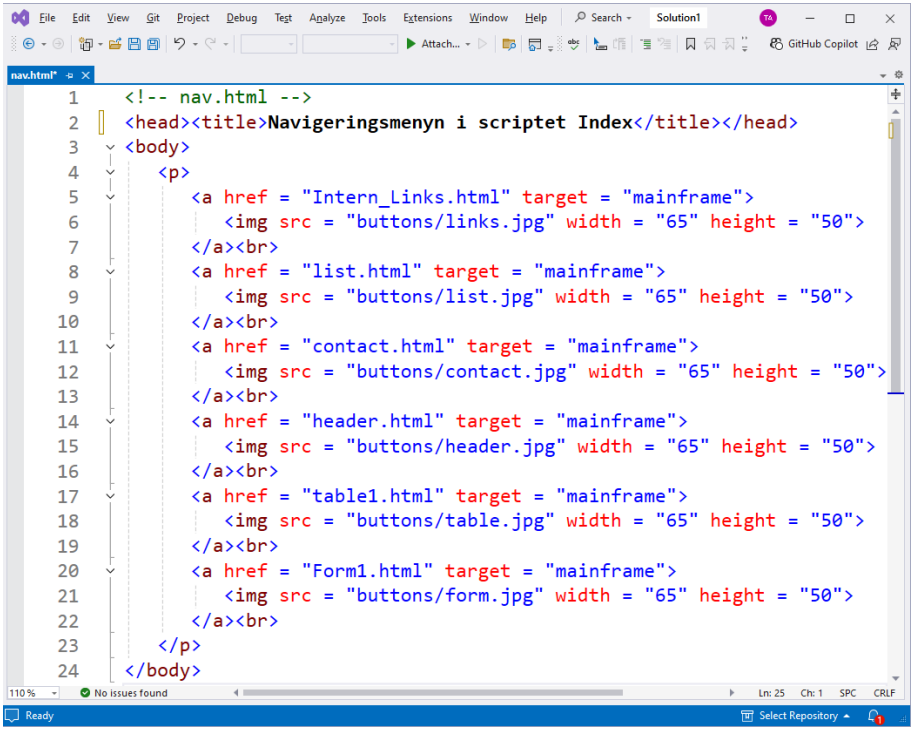


Elementet `frame`

Varje frame får med elementet `frame` ett namn och ett innehåll på raderna `6` och `7`. Den vänstra framen med namnet **`leftframe`** fylls med innehållet från scriptet **`nav`**, medan i den högre med namnet **`mainframe`** visas scriptet **`main`**. Båda script lagras som HTML-filer i samma mapp som scriptet `Index`, se mappen till höger. Medan **`main.html`** innehåller endast vanlig text som visas ovan, har filen **`nav.html`** lite mer intressant innehåll. Vi ska titta lite närmare på koden som återges på nästa sida.



Webbsidan består av 9 html-filer och mappen **`buttons`** som innehåller alla bilder, se bilden till höger.



Navigeringsmeny i en frame

Redan i scriptet **Nav** på sid 32 har vi introducerat en navigeringsmeny vars körresultat visas på sid 33. Koden har anpassats till att fungera här som navigeringsmeny i den vänstra ramen av scriptet **Index**.

Meningen med uppdelningen av dokumentet i två frames är att efterlikna designen i verkliga webbsidor där man kan navigera till olika delar av webbsidan.

Största nyheten i den här versionen jämfört med scriptet **Nav** på sid 32 är attributet **target** som lagts till i alla ankare, och som placerar resultatet av musklickningen på resp. bild i navigeringsmenyn i den frame som bär namnet **mainframe**, se scriptet **Index**, rad **7**. Meningen med detta är att sidan som bilden man klickar på, hänvisar till, visas i den högre (större) framen av startsidan, utan att det öppnas en ny flik, samtidigt som navigeringsmenyn finns kvar. Den ska ju användas för att navigera till andra delar av webbsidan.

Alla koder kan inte visas här. Hela mappen (avbildad på förra sidan) är zippad till filen **Framesets.zip** som kan laddas ned från kursens hemsida, se den aktuella lektionens övningar.

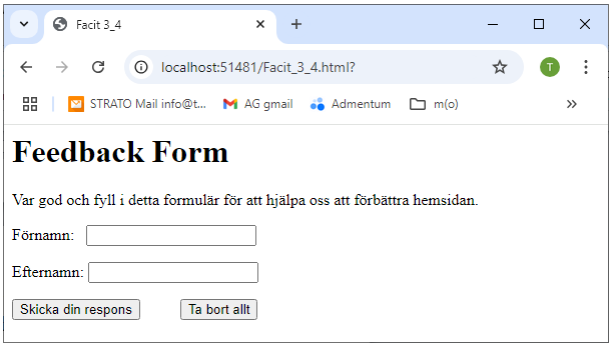
Övningar till **Kapitel 3**

3.1 Kör scriptet **`Tabell1`** (sid 49) i din webbläsare. Ladda scriptet i din favorit editor. Ändra värdet till **`table`**-attributet `border` till **`0`**, **`2`**, **`3`**, … för att se andra typer av kantlinjer. Ändra även värdet till attributet `width`. Blir tabellens bredd den procentuella delen av din webbläsares bredd?

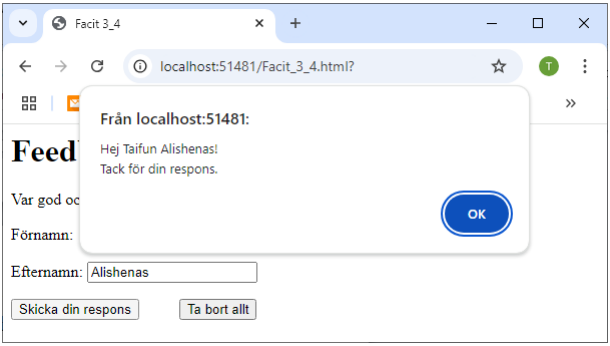
3.2 Vidareutveckla scriptet **`Tabell1`** (sid 49) genom att lägga in ytterligare en rad till tabellen som beskriver en valfri frukt. Ändra även det totala priset i tabbellens fot som en konsekvens av tillägget.

3.3 Modifiera scriptet **`Tabell2`** (sid 51) genom att ändra koden, så att körresultatet ser ut så här:

3.4 Lägg till scriptet **`Form1`** (sid 53) en textbox till som ska innehålla efternamnet. Ändra även koden så att formen får det här utseendet:

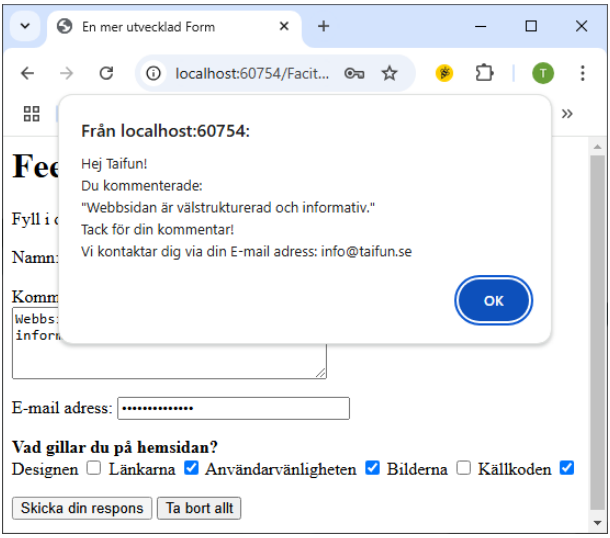


Man ska sedan fylla i sitt för- och efternamn och få följande svar när man klickar på knappen Skicka din respons:



För att få detta svar måste du även ändra koden i funktionen `showData()`.

3.5 Vidareutveckla scriptet **Form2** (sid 55) så att formen får det här utseendet:



Dvs modifiera koden, så att JavaScript funktionen `showData()` hämtar data inte bara från textboxen Namn (rad **9**), utan även från multiline textarean Kommentar (rad **15**) och från textboxen E-mail adress (rad **22**). För att uppnå resultatet ovan, bör du modifiera inte bara scriptets HTML-kod utan även JavaScript funktionen.

Intressant blir att se hur den maskerade E-mail adressen ”avslöjas” i JavaScripts *alert box* (sid 54).

3.6 Modifiera scriptet **Form3** (sid 55) i följande punkter:

1) Minska analet alternativ i rullgardinsmenyn till tre.
   Ge valfria namn till dina betygssteg.

2) Döp om radioknappen Referens i en bok till Tips från en kompis .

3) Ta bort radioknappen Taifun.se .

4) Döp om checkboxen Bilderna till Strukturen .

5) Lägg till i slutet en multiline textbox som frågar:

   Hur skulle vi kunna förbätra vår hemsida? Ge oss tips:

3.7 Scriptet **Intern_Links** (sid 60) har två huvudrubriker som båda är försedda med interna länkar. Komplettera scriptet med en tredje rubrik och lista upp dina favoritämnen där:

Mina tre favoritämnen i skolan:

Definiera en intern länk som leder till denna plats. Länka från de andra rubrikerna till den nya rubriken. Länka även från denna plats till de två andra rubrikerna.

3.8 I scriptet **Picture** (sid 66) finns en länk till Taifuns mail som är definierad i **area**-elementet med attributet **shape** = ”circle”. Flytta i ett nytt script denna länk (hotspot) till en annan plats, närmare bestämt till fyrkanten på vänstra sidan av bilden, under texten **Home Page**.

3.9 Ladda ned zipp-filen **Framesets.zip** och expandera den. Bygg med dessa filer webbsidan som scriptet **Index** är startsidan för (sid 68).

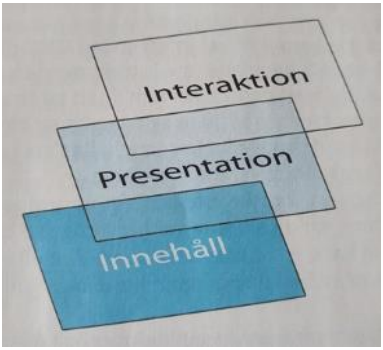
Modifiera sedan sidan genom att lägga ytterligare menyer (minst en) till navigeringsmenyn med egna bilder och ev. scriptfiler. Placera dina nya moduler på korrekta platser, så att allt fungerar på ett meningsfullt sätt.

Kapitel 4

Cascading Style Sheets (CSS)

4.1 Inline Styles

De tre skikten



I början av kursen (sid 10) talade vi om:

De *tre olika skikten* inom Webbutveckling:

- *Innehållet* kodas med **HTML**.
- *Presentationen* formges med **CSS**.
- *Interaktionen* programmeras med **JavaScript**.

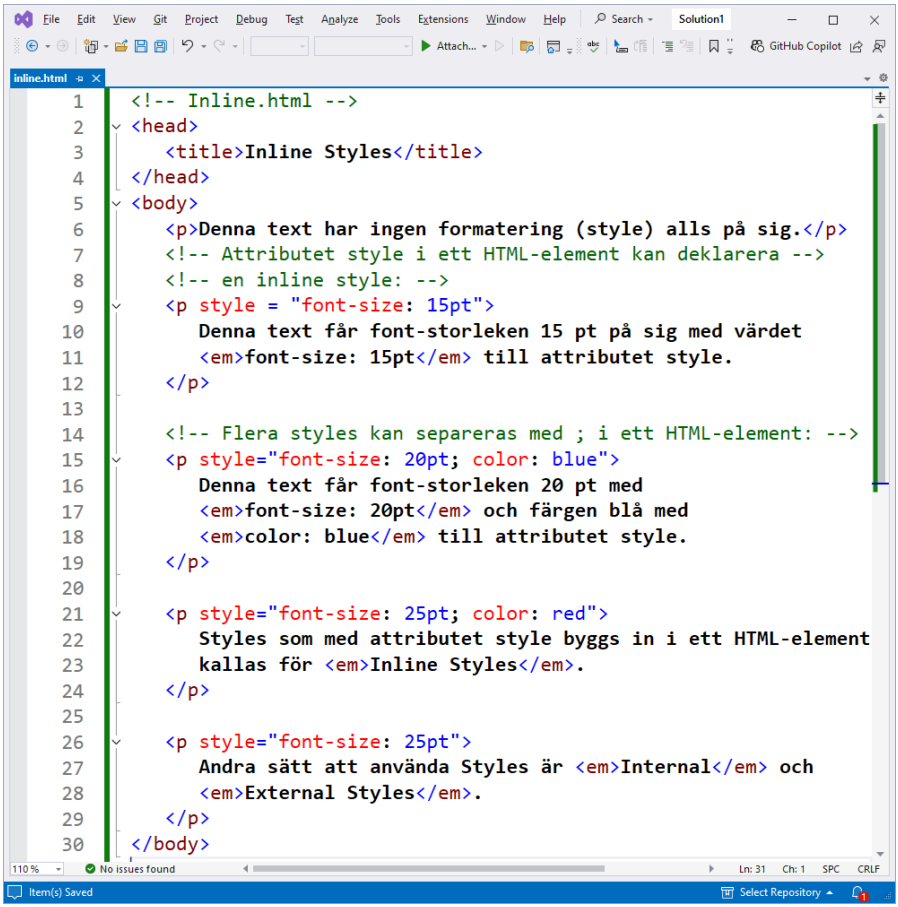
I tre kapitel har vi gått igenom det första skiktet **HTML** som ger *innehållet* eller *strukturen* åt webbsidan. Nu ska vi ägna oss åt det andra skiktet **CSS** som är tänkt att *presentera* webbsidor på bäst möjliga sätt. I det här skiktet står layouten, formen, utseendet och det visuella intrycket i centrum. För dessa står oss ett specifikt verktyg till förfogande som skiljer sig från HTML och som skulle kunna betecknas som ett nytt språk med egna syntaxregler. Vi menar CSS. I praktiken inbakas CSS i HTML, så att det blir svårt att skilja de två språken från varandra.

Vad är CSS?

CSS är ett *stilspråk* som används för att skapa s.k. *Style Sheets*. *Sheet* betyder på engelska blad papper, alltså i datasammanhang *fil*. I denna fil skriver man kod som definierar stilreregler som används för att formatera text i HTML-element. Med *formatering* menas allt som påverkar textens layout som t.ex. färger, typsnitt, storlekar, radavstånd, indrag, marginaler, bakgrundsfärger, bakgrundsbilder osv. Dessa regler fungerar på samma sätt som kommandon som används i ordbehandlingsprogram, vare sig de är av typ WYSIWYG (som t.ex.Word) eller skrivs som kod.

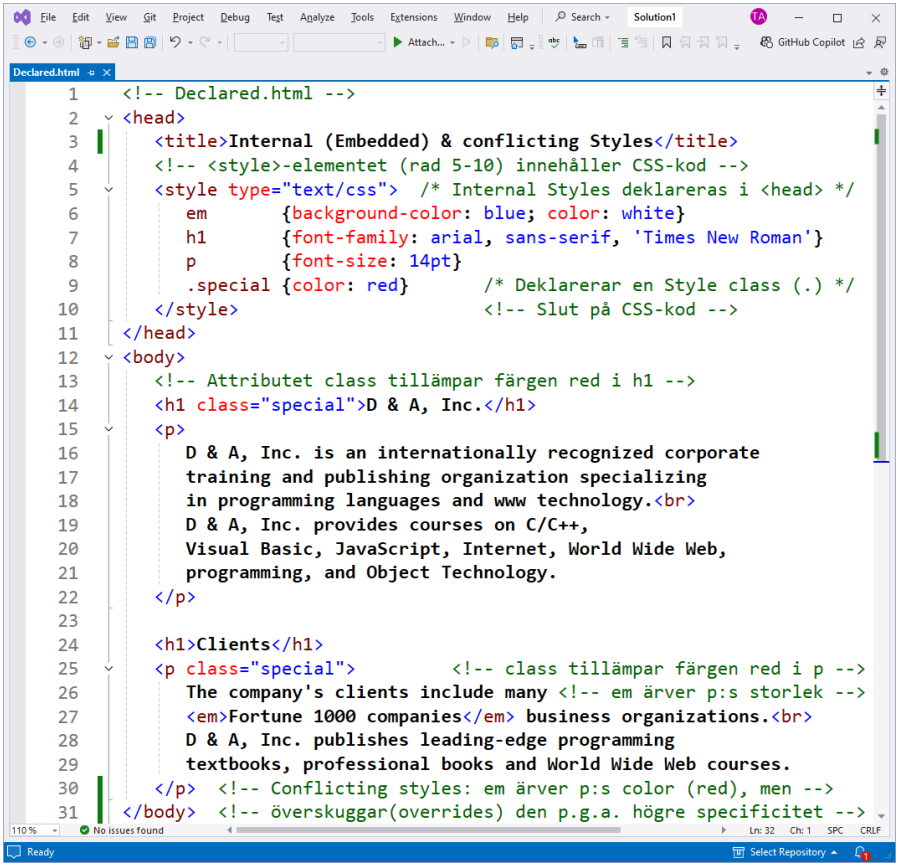
Historien bakom CSS

CSS introducerades redan 1994, blev standard 1996 och rekommenderades av W3C, det officiella internationella organet som tar fram standarder inom HTML. En fullständig specifikation för HTML och CSS kan hittas på W3C:s webbsida.

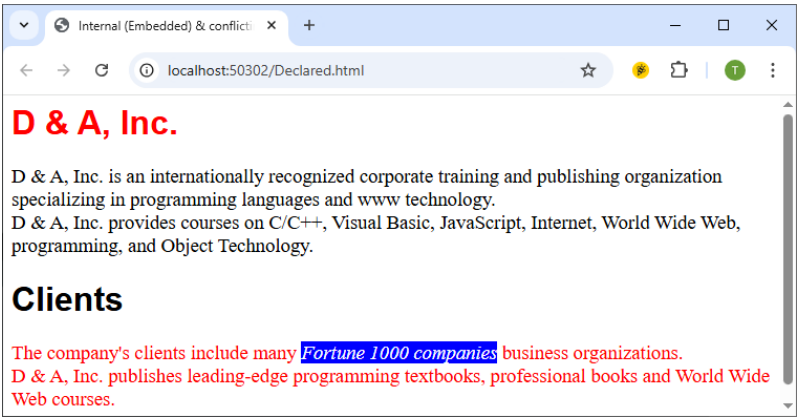

File Edit View Git Project Debug Test Analyze Tools Extensions Window Help
Search
Solution1
Attach...
GitHub Copilot
inline.html
1 <!-- Inline.html -->
2 <head>
3 <title>Inline Styles</title>
4 </head>
5 <body>
6 Denna text har ingen formatering (style) alls på sig.
7 <!-- Attributet style i ett HTML-element kan deklarera -->
8 <!-- en inline style: -->
9
10 Denna text får font-storleken 15 pt på sig med värdet
11 font-size: 15pt till attributet style.
12
13
14 <!-- Flera styles kan separeras med ; i ett HTML-element: -->
15
16 Denna text får font-storleken 20 pt med
17 font-size: 20pt och färgen blå med
18 color: blue till attributet style.
19
20
21
22 Styles som med attributet style byggs in i ett HTML-element
23 kallas för Inline Styles.
24
25
26
27 Andra sätt att använda Styles är Internal och
28 External Styles.
29
30 </body>
110 %
No issues found
Ln: 31 Ch: 1 SPC CRLF
Item(s) Saved
Select Repository


Inline Styles
localhost:59166/inline.html
Denna text har ingen formatering (style) alls på sig.
Denna text får font-storleken 15 pt på sig med värdet font-size: 15pt till attributet style.
Denna text får font-storleken 20 pt med font-size: 20pt och färgen blå med color: blue till attributet style.
Styles som med attributet style byggs in i ett HTML-element kallas för Inline Styles.
Andra sätt att använda Styles är Internal och External Styles.

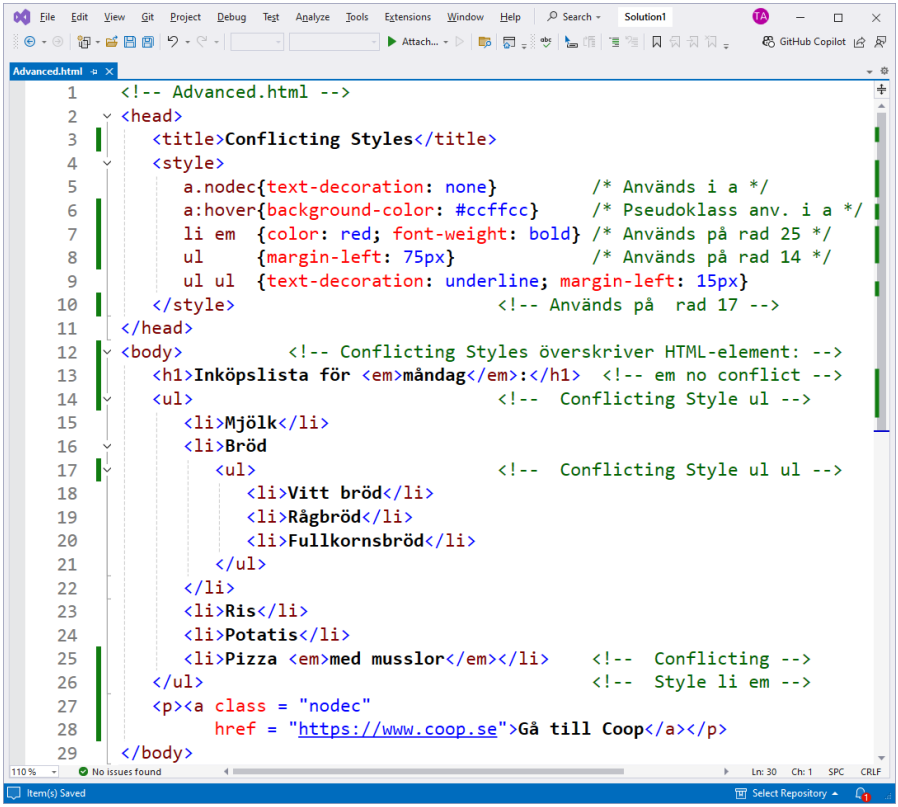
4.2 Internal Styles



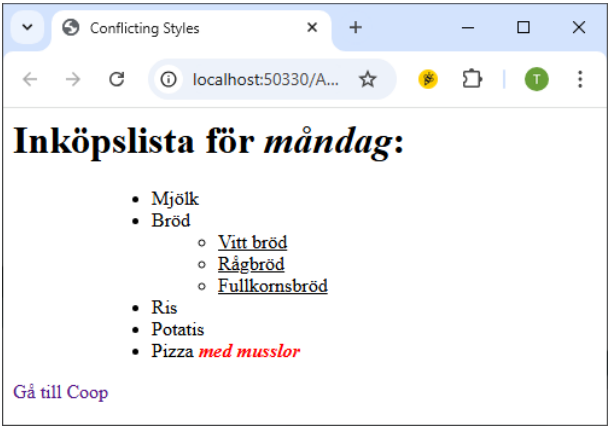
```
<!-- Declared.html -->
<head>
   <title>Internal (Embedded) & conflicting Styles</title>
   <!-- <style>-elementet (rad 5-10) innehåller CSS-kod -->
   <style type="text/css">  /* Internal Styles deklareras i <head> */
      em       {background-color: blue; color: white}
      h1       {font-family: arial, sans-serif, 'Times New Roman'}
      p        {font-size: 14pt}
      .special {color: red}       /* Deklarerar en Style class (.) */
   </style>                       <!-- Slut på CSS-kod -->
</head>
<body>
   <!-- Attributet class tillämpar färgen red i h1 -->
   <h1 class="special">D & A, Inc.</h1>
   <p>
      D & A, Inc. is an internationally recognized corporate
      training and publishing organization specializing
      in programming languages and www technology.<br>
      D & A, Inc. provides courses on C/C++,
      Visual Basic, JavaScript, Internet, World Wide Web,
      programming, and Object Technology.
   </p>

   <h1>Clients</h1>
   <p class="special">         <!-- class tillämpar färgen red i p -->
      The company's clients include many <!-- em ärver p:s storlek -->
      <em>Fortune 1000 companies</em> business organizations.<br>
      D & A, Inc. publishes leading-edge programming
      textbooks, professional books and World Wide Web courses.
   </p>  <!-- Conflicting styles: em ärver p:s color (red), men -->
</body>  <!-- överskuggar(overrides) den p.g.a. högre specificitet -->
```



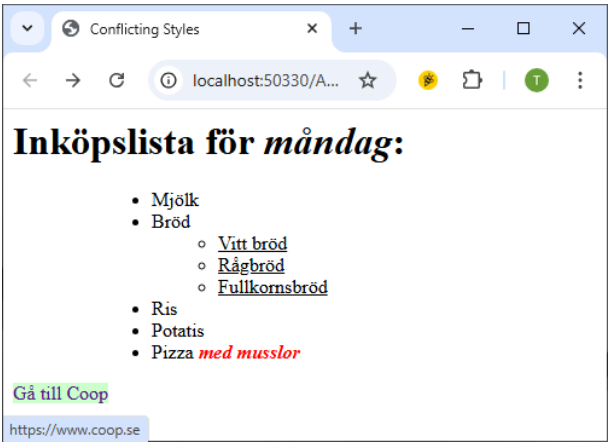
4.3 Conflicting Styles



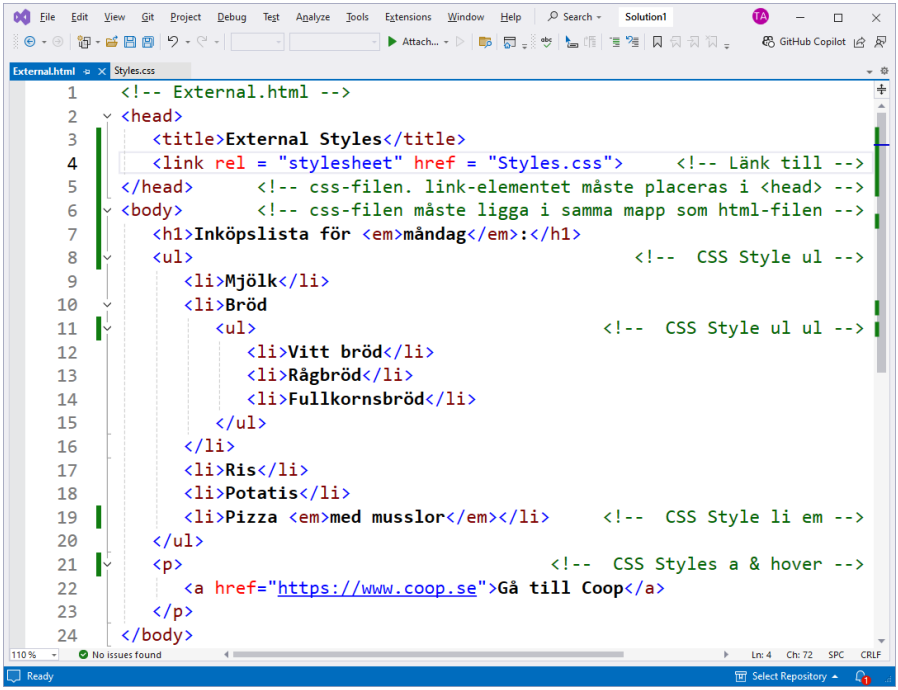
```
<!-- Advanced.html -->
<head>
   <title>Conflicting Styles</title>
   <style>
      a.nodec{text-decoration: none}         /* Används i a */
      a:hover{background-color: #ccffcc}     /* Pseudoklass anv. i a */
      li em  {color: red; font-weight: bold} /* Används på rad 25 */
      ul     {margin-left: 75px}             /* Används på rad 14 */
      ul ul  {text-decoration: underline; margin-left: 15px}
   </style>                              <!-- Används på  rad 17 -->
</head>
<body>          <!-- Conflicting Styles överskriver HTML-element: -->
   <h1>Inköpslista för <em>måndag</em>:</h1>  <!-- em no conflict -->
   <ul>                                  <!--  Conflicting Style ul -->
      <li>Mjölk</li>
      <li>Bröd
         <ul>                            <!--  Conflicting Style ul ul -->
            <li>Vitt bröd</li>
            <li>Rågbröd</li>
            <li>Fullkornsbröd</li>
         </ul>
      </li>
      <li>Ris</li>
      <li>Potatis</li>
      <li>Pizza <em>med musslor</em></li>    <!--  Conflicting -->
   </ul>                                    <!--  Style li em -->
   <p><a class = "nodec"
         href = "https://www.coop.se">Gå till Coop</a></p>
</body>
```



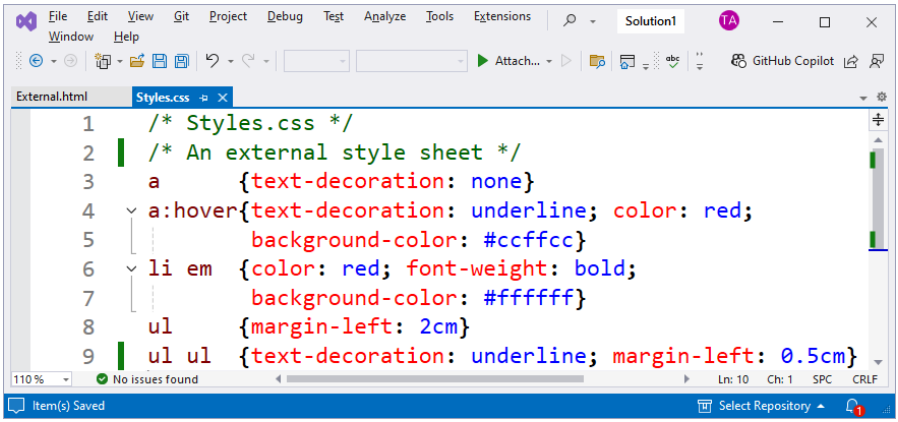
Mus över Gå till Coop :



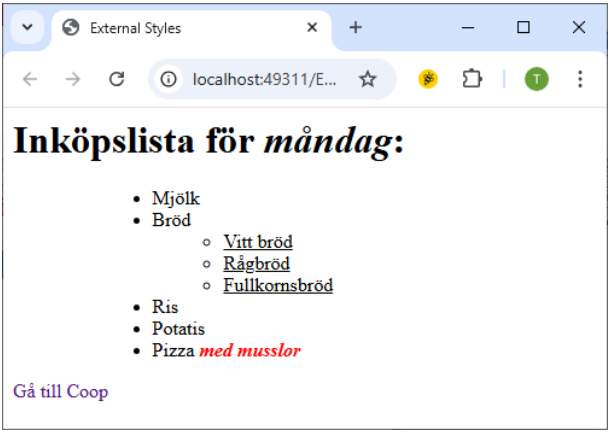
4.4 External Styles



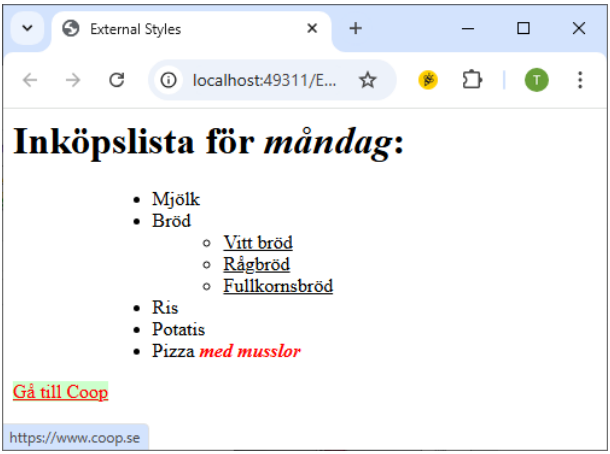
```
<!-- External.html -->
<head>
   <title>External Styles</title>
   <link rel = "stylesheet" href = "Styles.css">     <!-- Länk till -->
</head>      <!-- css-filen. link-elementet måste placeras i <head> -->
<body>       <!-- css-filen måste ligga i samma mapp som html-filen -->
   <h1>Inköpslista för <em>måndag</em>:</h1>
   <ul>                                          <!--  CSS Style ul -->
      <li>Mjölk</li>
      <li>Bröd
         <ul>                                 <!--  CSS Style ul ul -->
            <li>Vitt bröd</li>
            <li>Rågbröd</li>
            <li>Fullkornsbröd</li>
         </ul>
      </li>
      <li>Ris</li>
      <li>Potatis</li>
      <li>Pizza <em>med musslor</em></li>     <!--  CSS Style li em -->
   </ul>
   <p>                                    <!--  CSS Styles a & hover -->
      <a href="https://www.coop.se">Gå till Coop</a>
   </p>
</body>
```



```
/* Styles.css */
/* An external style sheet */
a       {text-decoration: none}
a:hover{text-decoration: underline; color: red;
        background-color: #ccffcc}
li em  {color: red; font-weight: bold;
        background-color: #ffffff}
ul      {margin-left: 2cm}
ul ul  {text-decoration: underline; margin-left: 0.5cm}
```



Mus över Gå till Coop :



4.5 Absolut positionering

```
1   <!-- Positioning.html -->
2   <head>
3      <title>Absolut positionering</title>
4   </head>
5   <body>
6      <p><img src = "i.gif" style = "position: absolute;
7         top: 0px; left: 0px; z-index: 1"
8         alt = "Positionerad bild 1"></p>
9
10     <p style = "position: absolute; top: 50px; left: 15px;
11        z-index: 3; font-size: 20pt;">Absolut positionering</p>
12
13     <p><img src = "circle.gif" style = "position: absolute;
14        top: 25px; left: 100px; z-index: 2"
15        alt = "Positionerad bild 2"></p>
16  </body>
```

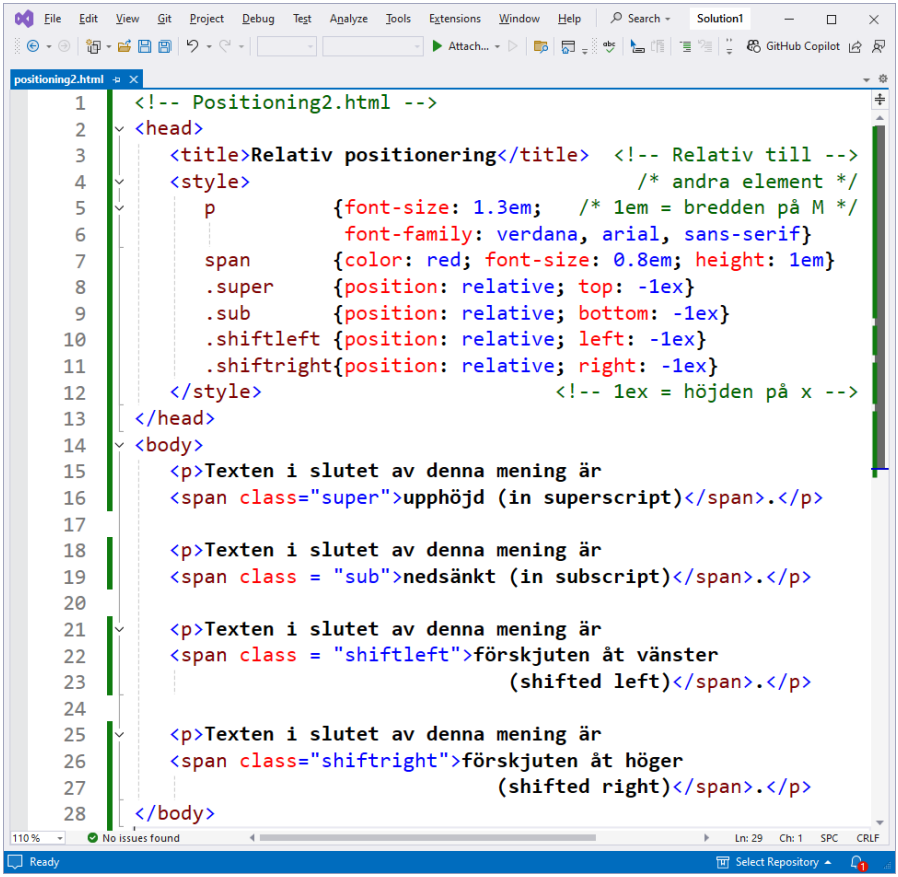
Attributet `position`

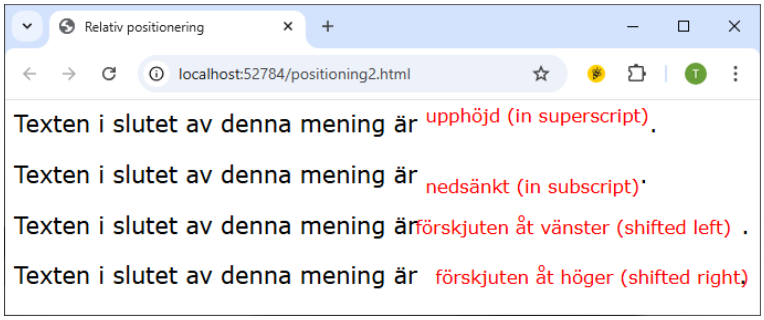
Före CSS var det svårt att kontrollera positioneringen av elementen i ett HTML-dokument. Det var webbläsaren som ofta bestämde positioneringen. CSS introducerade attributet **`position`** som ger utvecklaren kontrollen över positioneringen av element, genom att bl.a. bjuda på möjligheten att använda *absolut positioning*.

Scriptet **`Positioning`** (ovan) använder två bilder, i.gif och circle.gif, och en text för att sätta ihop dem till dokumentet ovan. Raderna **`6-8`** placerar den första bilden (i.gif) så att den hamnar, oberoende av den normala ordningen i koden, på den absoluta positionen **`0`** pixlar från den vänstra och övre kanten.

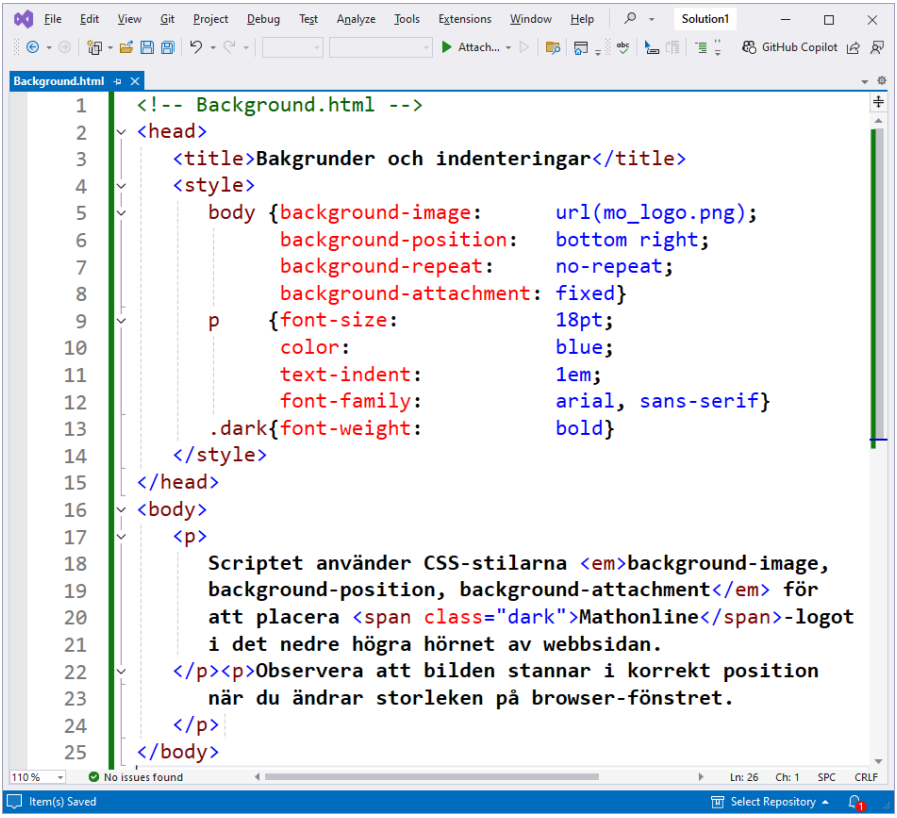
Attributet **`z-index`** ordnar överlappningen av elementen. Element med högre **`z-index`** placeras framför element med lägre **`z-index`**. I vårt exempel har bilden i.gif det lägsta **`z-indexet`**, nämligen **`1`**. Därför hamnar den i bakgrunden. circle.gif har index **`2`** och texten index **`3`**. Om man inte anger ett värdre tilt **`z-index`** placeras elementen från bakgrunden framåt i den ordning de skrivs till dokumentet,

4.6 Relativ positionering

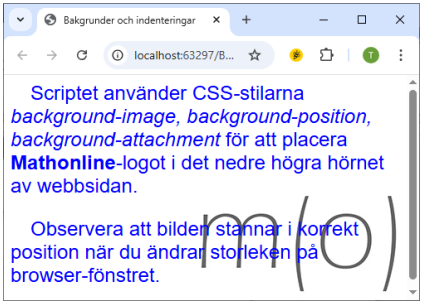




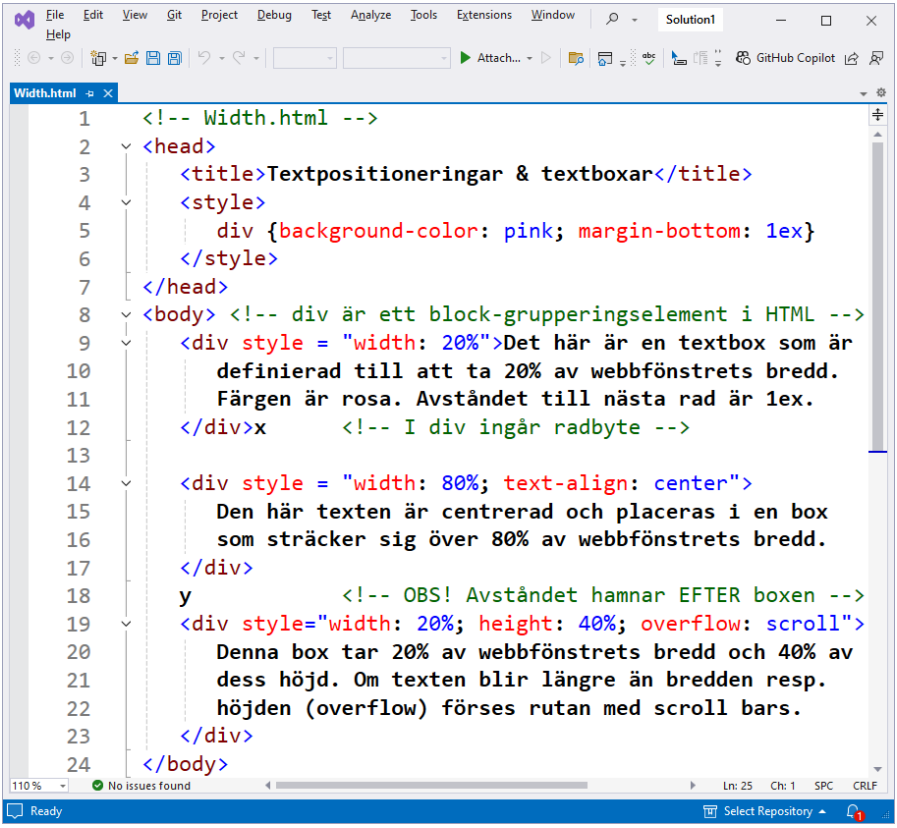
4.7 Bakgrunder



```
<!-- Background.html -->
<head>
   <title>Bakgrunder och indenteringar</title>
   <style>
      body {background-image:      url(mo_logo.png);
            background-position:   bottom right;
            background-repeat:     no-repeat;
            background-attachment: fixed}
      p    {font-size:             18pt;
            color:                 blue;
            text-indent:           1em;
            font-family:           arial, sans-serif}
      .dark{font-weight:           bold}
   </style>
</head>
<body>
   <p>
      Scriptet använder CSS-stilarna <em>background-image,
      background-position, background-attachment</em> för
      att placera <span class="dark">Mathonline</span>-logot
      i det nedre högra hörnet av webbsidan.
   </p><p>Observera att bilden stannar i korrekt position
      när du ändrar storleken på browser-fönstret.
   </p>
</body>
```

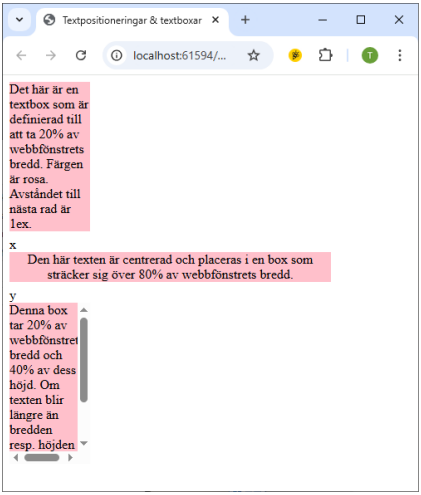


4.8 Textpositioneringar



Elementet `div`

Detta element liknar elementet **`span`** som vi lärde känna i avsnitt ***4.6 Relativ positionering***, scriptet **`Positioning2`** (sid 82). Båda är s.k. *grupperingselement*. Dvs de gör inget annat än att gruppera andra HTML-element samt CSS-formateringar. Skillnaden är att **`span`** är ett s.k. *inline-* grupperingselement, medan **`div`** är ett s.k. *block-*grupperingselement. **`div`** inkluderar radbyte, medan **`span`** inte gör det. Scriptet **`Width`** (ovan) använder denna egenskap av elementet **`div`**.



Övningar till Kapitel 4

4.1 Kör scriptet **Inline** (sid 75) i din webbläsare. Ladda scriptet i din favorit editor. Ändra värdena till attributet **style** så att texterna skrivs i omvänd fontstorlek, dvs den största fonten först och den minsta sist. Ändra även värdena till attributet **color** genom att experimentera med andra färger.

Byt ut färgvärdena blue och red mot färg*koder* genom att ta reda på koderna. T.ex. är #0000ff färgkoden till blue.

4.2 Modifiera scriptet **Declared** (sid 76) genom att ändra namnet till Style-klassen **special** till ditt namn. Testa om scriptet fungerar precis som förut. Det borde det göra om du ändrat korrekt. Kasta om ordningen av fonterna i Stylen **h1** och testa. Vilken slutsats drar du?

Ta bort attributet **color** från Stylen **em**. Vilken färg får den framhävda texten? Från vilket element ärver texten sin färg?

Byt även ut den framhävda textens bakgrundsfärg till andra färger. Experimentera både med färgnamnen och färgkoder.

4.3 Mata in scriptet **Advanced** (sid 77) i din favorit editor och kör det i din webbläsare. Gå med musen över texten **Gå till Coop**. Vilken Style i scriptet ändrar textens bakgrundsfärg? Ändra bakgrundsfärgen till red. Vilken Style i scriptet är ansvarig för indragningen av item **Mjölk**? Ändra indragningen till 50 pixlar. Ändra även indragningen av item **Vitt bröd** till 10 pixlar.

Byt ut färgen av texten ***med mussllor*** till blue och låt den inte längre skrivas i fet utan i normal stil. Experimentera med ytterligare stiländringar i scriptet. Varför har texterna ***måndag*** och ***med mussllor*** olika formateringar, fast båda ligger i koden i HTML-elementet ****?

4.4 Dela upp scriptet **Declared** (sid 76) i två filer genom att flyttta den interna csskoden till en extern css-fil, så att alla internal Styles blir external Styles.

Först ska scriptet generera samma dokument som det ursprungliga scriptet **Declared** (sid 76) gjorde.

Modifiera sedan scriptet genom att göra de ändringar som föreslås i övn. 4.2 (på webben: Övningar 29).

4.5 Läs igenom texten och gå igenom koden på sid 81. Skriv ett script som positionerar elementen i scriptet **Positioning** så att hela bildgruppen hamnar i det nedersta högre hörnet av browserfönstret på sid 81.

4.6 Dela upp scriptet **Positioning2** (sid 82) i två filer genom att flyttta all css-kod till en extern css-fil.

Först ska scriptet generera samma dokument som det ursprungliga scriptet **Positioning2** gjorde.

Modifiera sedan scriptet genom att göra experiment med de relativa positioneringarna, t.ex. med font-storlekarna, textförskjutningarna osv.

4.7 Modifiera scriptet **Background** (sid 83) så att bilden hamnar i browser-fönstrets andra tre hörn.

Gör experiment med de andra CSS-stilar som är definierade i scriptet. Öka t.ex. indenteringarnas storlek, textens typsnitt, storlek och färg.

Ändra stilklassens innehåll genom att leta efter information om attributen som är tillgängliga för stilklassen. Vad händer om du tar bort den inledande punkten i stilklassens namn?

4.8 Centrera i scriptet **Width** (sid 84) texten i den första textboxen och justera textens bredd så att man ser centreringen.

Låt texten i den andra textboxen vara ocentrerad och lite längre. Förse den med rullningslistor (scroll bars) så att de blir synliga.

Placera en längre valfri text i den tredje textboxen. Ta bort rullningslistorna och justera boxens bredd och höjd, så att man ser hela texten även utan rullningslistor. Ge i den tredje textboxen hela paragrafen en indentering och ändra textens storlek samt färg. Dra nytta av dina kunskaper från förra avsnitt, scriptet **Background** (sid 83).

Gör experiment med elementen **span** och **div** för att lära känna dera skillnader. Läs om detta under rubriken ***Elementet* div** på sid 84.

Kapitel 5

Introduktion till JavaScript

5.1 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.

5.2 Att komma igång med JavaScript

För att komma igång med JavaScript kan vi nu skriva våra koder i en valfri texteditor 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, skriv följande kod med bibehållen layout (utan radnumren):

```
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, för att visa körresultatet i webbläsaren. 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:

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 webbläsarens flik. I programmet `Welcome` är det texten **`Vårt första program i JavaScript`** som man kan se i körresultatet längst upp till vänster. Än så länge har vi ren HTML-kod.

<script>-taggen

På rad **5** börjar nästa tagg med `<script>` som slutar på rad **7** med `</script>`. `<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.

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 i HTML börjar alltid med **`<!--`**, slutar med **`-->`** och kan sträcka sig över flera rader. Samma syntax för kommentar används på rad **7**, då vi befinner oss i HTML-området av scriptet. På rad **5** däremot har kommentaren skrivits annorlunda, eftersom där har i och med `<script>`-taggen JavaScript-kod börjat gälla. I JavaScript inleds en radkommentar med koden `//` och gäller till slutet av raden.

Satser i JavaScript

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

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

Den kan även avslutas med semikolon (**`;`**), speciellt om man vill fortsätta att skriva nästa sats. Men semikolonet kan utelämnas om man skriver den på en separat rad. Man skiljer alltså satser i JavaScript från varandra genom att sätta semikolon mellan dem eller skriva dem på separat rad.

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 17) 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.

5.3 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`**.

5.4 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.

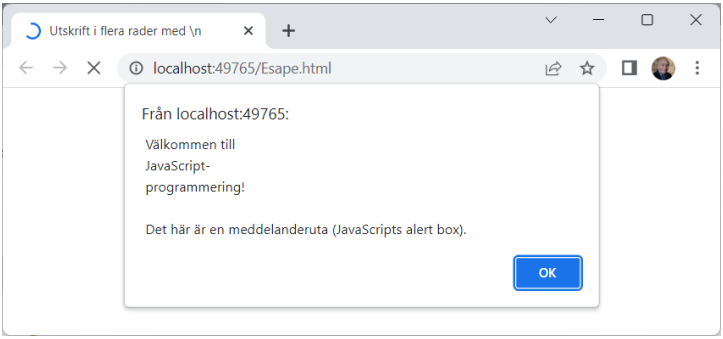
Konkateneringsoperatorn **`+`** används på rad **`6`**, precis som i programmet **`Concat`** (sid 19), för att inte behöva bryta rad i koden mitt i en sträng, se regeln på förra sidan.

`<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.

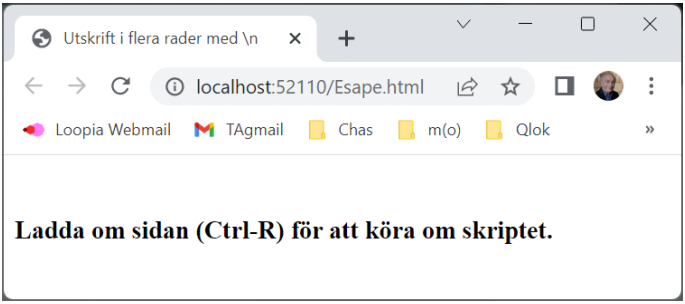
Radbrytning i utskriften med JavaScript

```
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 `<br>` inte fungera i programmet **`Escape`**. Genomför gärna detta experiment genom att byta ut alla **`\n`** på raderna **`7`** & **`8`** i programmet **`Escape`** mot `<br>`.

I programmet **`Escape`** blir inte bara skillnaden 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 5**

5.1 Är JavaScript ett *universellt* programmeringsspråk? Motivera!

5.2 Är JavaScript ett interpreterande eller ett kompilerande språk?

5.3 Är JavaScript källkod eller maskinkod?

5.4 I vilken miljö exekveras JavaScript kod?

5.5 Vad har JavaScript med HTML att göra?

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

5.7 Vilka typer av ordbehandlingsprogram är olämpliga för programmering?

5.8 Varför är filändelser relevanta för en programmerare?

Övningar till **Kapitel 5**

5.1 Öppna din favorit editor. Annars ladda ned och installera Open-source editorn Notepad++. Skriv i editorn tencken *apostrof* (**'**), *citationstecken* (**"**), *accent* (´), *slash* (**/**), *backslash* (****) och *pipe* (**|**). Bekanta dig med deras skillnader vad gäller form och utseende. Ta reda på och kom ihåg deras plats på ditt tangentbord – som kan vara olika på olika datortyper.

5.2 Visar din dator filändelserna när du öppnar en mapp? Om inte, genomför instruktionerna *Filändelser* på sid 16 för att synliggöra filändelserna.

5.3 Öppna en texteditor som inte formaterar text och mata in koden till programmet **`Welcome`** (sid 17). Bibehåll layouten. Spara koden i filen **`Welcome.html`**. Kör filen i din webbläsare. Ersätt alla apostrofer i koden med citationstecken och kör om koden. Vilken slutsats drar du?

5.4 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.

5.5 Ersätt **`document.writeln()`**-satsen i programmet **`Concat`** (sid 92) 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.

5.6 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 94) `<br>` med **`\n`**. Ersätt i programmet **`Escape`** (sid 94) **`\n`** med `<br>`. Beakta funktionerna i vilka dessa koder fungerar. Vilka slutsatser drar du?

5.7 Skriv ett JavaScript program som åstadkommer följande utskrift:

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

5.8 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>'     )
```

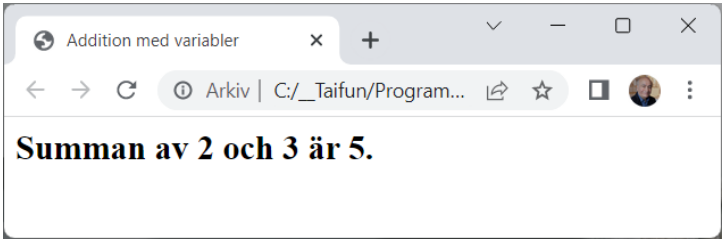
Kapitel 6

Grundbegrepp i JavaScript

6.1 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?

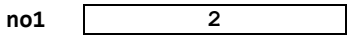
En variabel är en platshållare (minnescell) för ett värde (data).

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.

6.2 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 100)

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

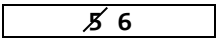
Dvs:

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 100). 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*.

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.

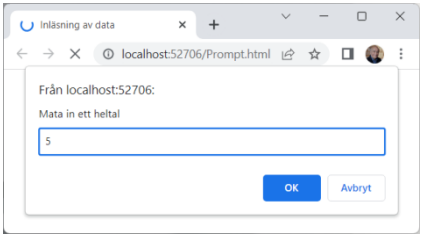
Med andra ord: Tilldelning är likhet relaterad till tiden dvs vid ett visst ögonblick, medan likheten är tidlös.

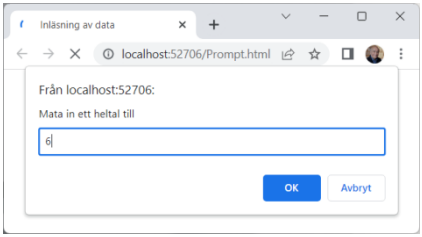
6.3 Inläsning av data

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()`**.

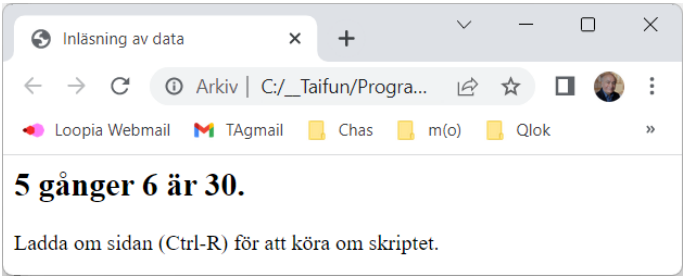
```
1  <!-- Input.html
2       Läser in två tal och skriver ut dem samt deras produkt
3       Funktionen prompt() skriver ut en ledtext och läser in -->
4  <title>Inläsning av data</title>
5
6  <script>
7     no1 = prompt('Mata in ett heltal')        // Inläsning
8     no2 = prompt('Mata in ett heltal 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 100). 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.

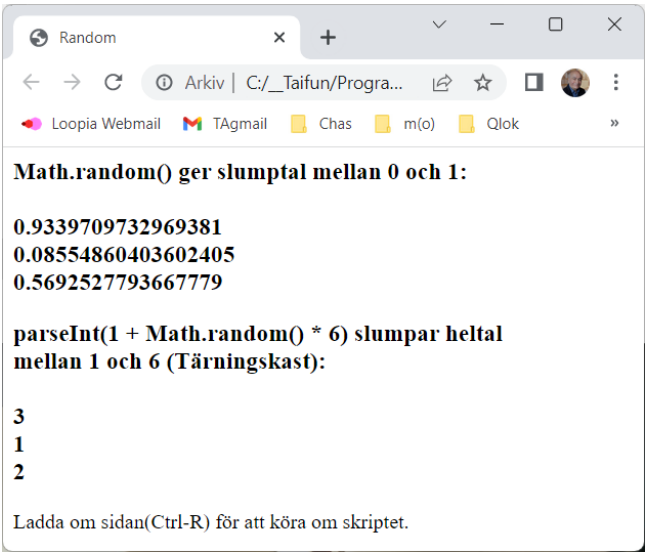
Inmatningen *returneras* av funktionen **prompt()**. Men eftersom **prompt()** är en fördefinierad funktion i JavaScript som returnerar en sträng, som tilldelas variabeln **no1**, som automatiskt omvandlas till tal när det multipliceras med **no2** på rad **9**.

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.

6.4 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`.

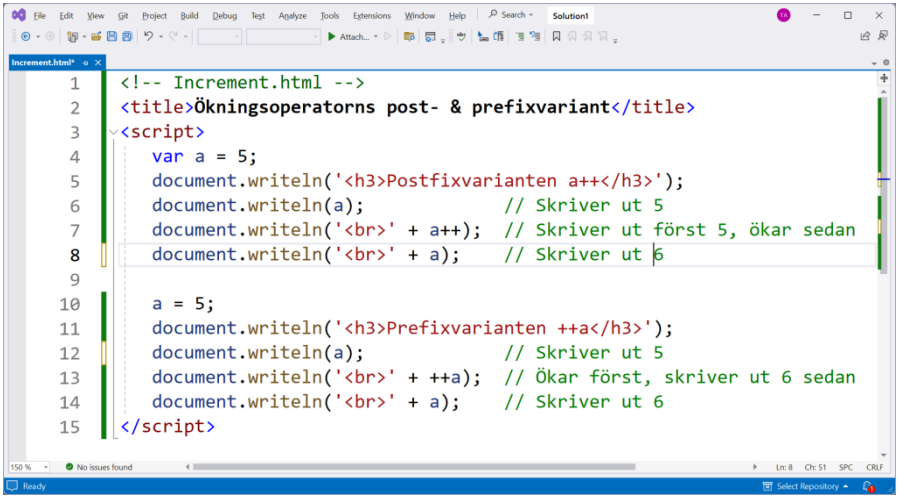
Är `a > b` måste formeln ovan ersättas med:

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

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

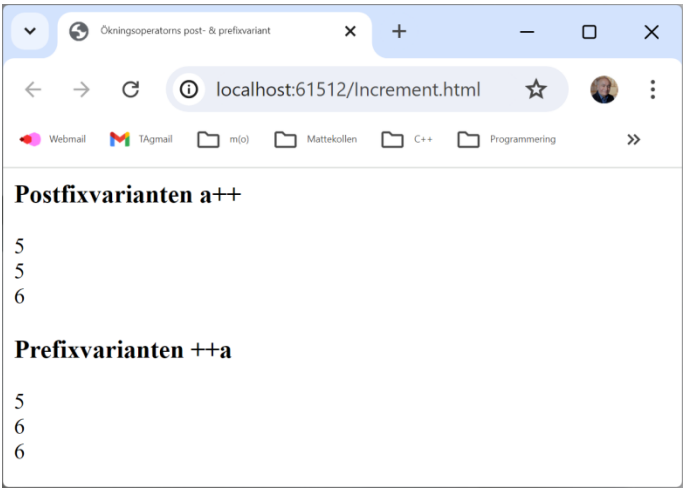
6.5 Ökningsoperatorn ++

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:



```
1  <!-- Increment.html -->
2  <title>Ökningsoperatorns post- & prefixvariant</title>
3  <script>
4     var a = 5;
5     document.writeln('<h3>Postfixvarianten a++</h3>');
6     document.writeln(a);              // Skriver ut 5
7     document.writeln('<br>' + a++);   // Skriver ut först 5, ökar sedan
8     document.writeln('<br>' + a);     // Skriver ut 6
9
10    a = 5;
11    document.writeln('<h3>Prefixvarianten ++a</h3>');
12    document.writeln(a);              // Skriver ut 5
13    document.writeln('<br>' + ++a);   // Ökar först, skriver ut 6 sedan
14    document.writeln('<br>' + a);     // Skriver ut 6
15 </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 6**

6.1 Komplettera programmet **Variable** (sid 100) 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.

6.2 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.

6.3 Ersätt i programmet **OverWrite** (sid 103) 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?

6.4 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 106).

6.5 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.

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

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

Kapitel 7

Kontrollstrukturer i JavaScript

7.1 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.

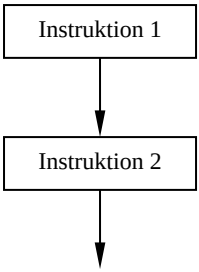
Kontrollstrukturer består av tre grundläggande typer:

- **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.

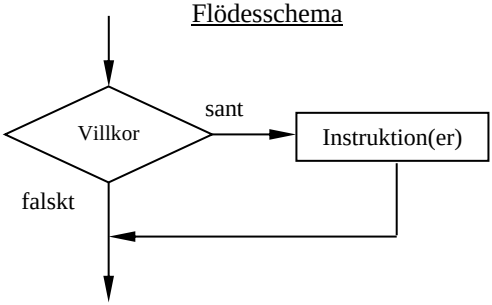
7.2 Enkel selektion: `if`-satsen

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



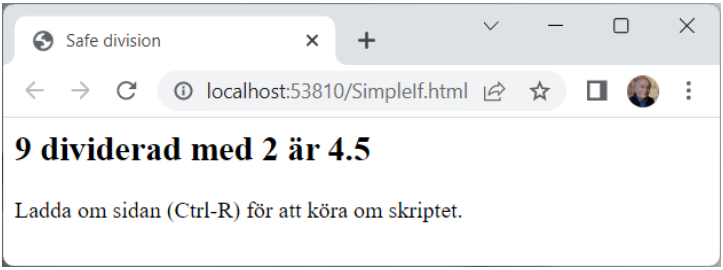
I JavaScript kallas den enkla selektionen för **if**-sats och kodas generellt på följande sätt:

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

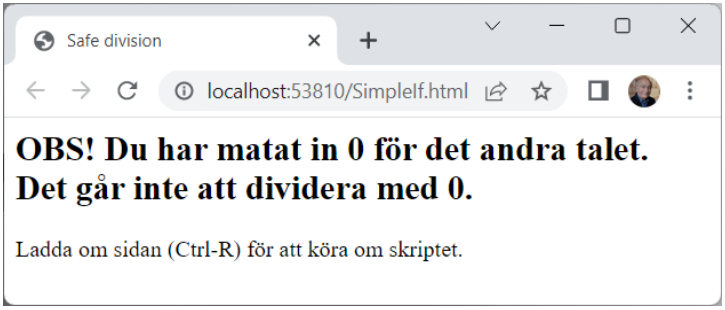
Första raden är **if**-satsens *huvud*. Resten är **if**-satsens *kropp* som omsluts av klammerparenteserna **{** och **}** som vi i fortsättningen kommer att kalla kort *klamrar*, ibland *måsvingar*. Om kroppen består endast av en sats kan klamrarna utelämnas vilket vi utnyttjar i följande program:

```
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

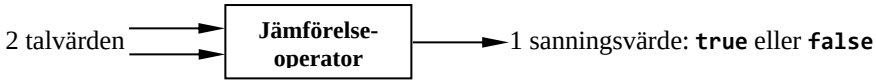
`if`-satsens huvudingrediens är alltid ett *villkor*, t.ex. `no2 != 0`. Dubbeltecknet `!=` betyder *icke lika med* och måste skrivas *utan* mellanslag: Är `no2` skilt ifrån `0`, ja eller nej? Man kan alltså uppfatta ett villkor som en *fråga* som endast kan besvaras med ja eller nej. En annan aspekt 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 104 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 115).

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.

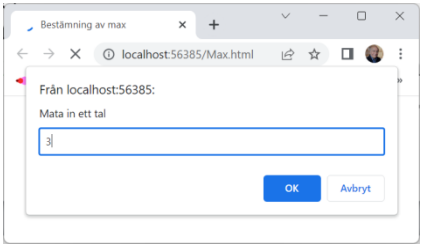
Bestämning av max/min

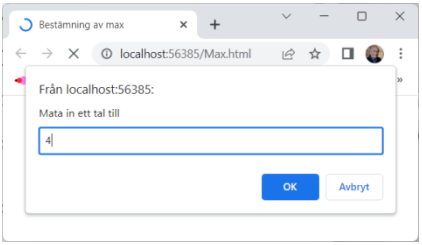
I programmet **SimpleIf** (sid 115) användes två **if**-satser, för att avgöra om ett tal var jämnt eller udda. Nu ska vi skriva ett nyttigt program som bestämmer det största (minsta) värdet bland 3 inmatade tal. Nyttigt, därför att vi kommer att ha användning av det bl.a. i projektet Gymnastiktävlingen. Sedan ska vi använda detta exempel för att precisera vår kunskap om *modularisering* som nämndes inledningsvis i boken (sid 11), och lära oss att själva definiera *funktioner* i JavaScript.

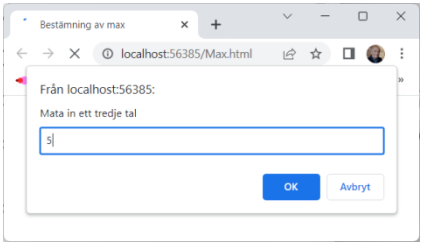
```
1  <!-- Max.html
2       Läser in 3 tal och bestämmer det största bland dem
3       Två enkla if-satser löser problemet  -->
4  <title>Bestämning av max</title>
5  <script>
6    no1 = parseInt(prompt('Mata in ett tal'))     // Inläsning
7    no2 = parseInt(prompt('Mata in ett tal till'))
8    no3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

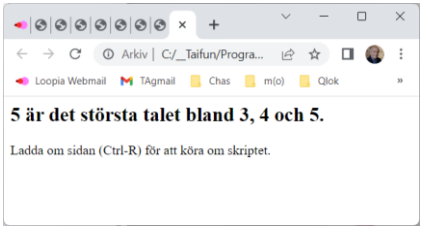
Själva algoritmen att hitta det största bland tre tal, är kodad på raderna **10-15** med två enkla **if**-satser: Först antar vi att **no1** är det största talet och tilldelar det variabeln **max**. Det behöver inte stämma. Den första **if**-satsen (rad **11-12**) testar detta antagande genom att kolla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

satsen med **no3** (rad **14-15**). Slutligen kommer **max**-”rollen” ges till det tal som är störst av alla tre. Resten är inläsning och utskrift:









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

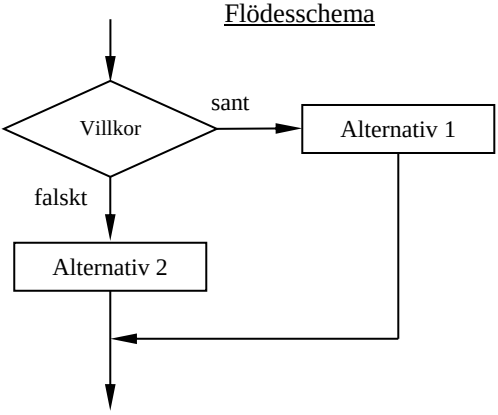
7.3 Tvåvägsval: if-else-satsen

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
}
```

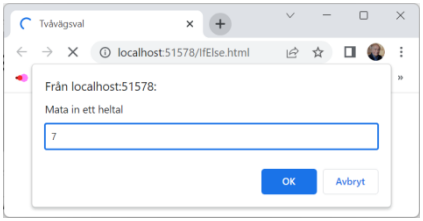
Om **`if`**- eller **`else`**-blocket består endast av en sats kan klamrarna **`{`** och **`}`** utelämnas. Anta att båda block består bara av en sats, då förenklas formen:

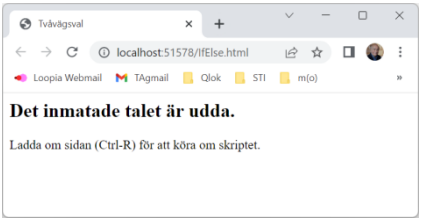
```
if (villkor)
    sats1
else
    sats2
```

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

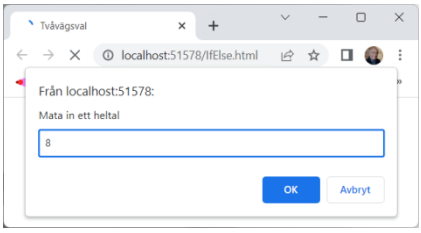
```
1  <!-- IfElse.html
2       Läser in ett heltal och avgör om det är jämnt eller udda
3       Tvåvägsval: if-else-satsen med EN sats i if-else-delen -->
4
5  <title>Tvåvägsval</title>
6
7  <script>
8     no = parseInt(prompt('Mata in ett heltal'))   // Inläsning
9
10    if (no % 2 == 0)
11       document.writeln('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

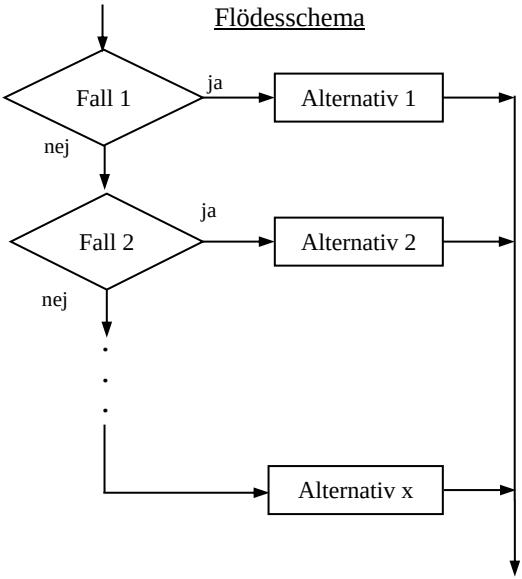
7.4 Flervägsval

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

Pseudokod

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

Flödesschema



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

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

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

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

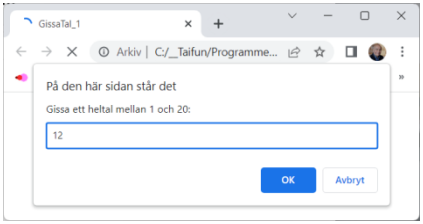
`if-else`-*stegen*

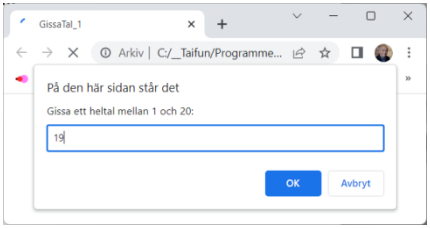
Låt oss titta på följande exempel av ett trevägsval som använder den s.k. **`if-else`**-stegen. Användaren ska gissa fram programmets hemliga tal **`17`**. Man gissar inom intervallet **`[1, 20]`**, får sedan hjälp om det gissade talet var mindre än, större än eller lika med det hemliga talet. Just nu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

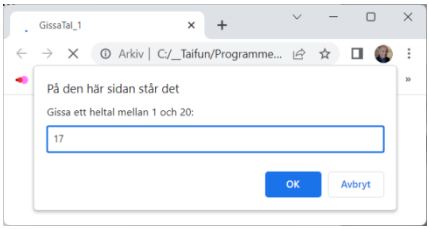
Här kommer *Gissa tal-spelet* i en första version, som innehåller ett val mellan tre alternativ, *fall 1:* det gissade talet är lika med, *fall 2:* mindre än, *fall 3:* större än programmets hemliga tal **17**:

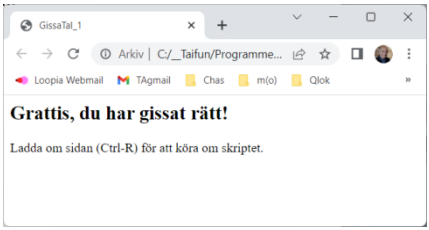
```
<!-- GissaTal.html
     Låter användaren gissa programmets hemliga tal secret
     Trevägsval med en if-else stege -->
<title>GissaTal</title>
<meta charset="UTF-8">          <!-- För de svenska tecknen -->
<script>
   secret = 17                  // Programmets hemliga tal
                     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

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

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

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









`switch`-*satsen*

Flervägsvalets flödesschema som visades på sid 123 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.

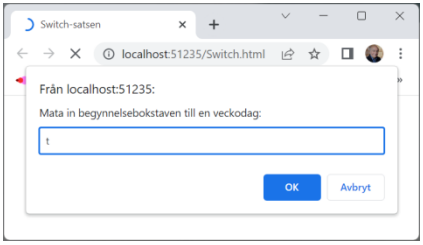
Följande programexempel demonstrerar **switch**-satsen: Vi läser in begynnelsebokstaven till en veckodag och det fullständiga veckodagsnamnet skrivs ut. I **switch**-satsen väljs ett alternativ av sex. Tisdag och torsdag behandlas i ett fall.

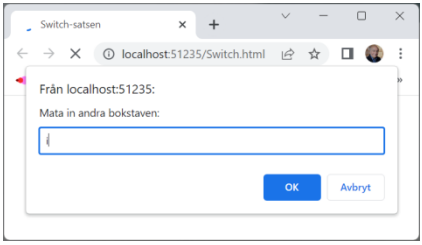
```
<!-- Switch.html
     Demonstrerar flervägsval med switch-satsen
     Kompletterar veckodagen efter inmatning av första bokstaven
     För t(isdag/torsdag) krävs den 2:a bokstaven -->
<title>Switch-satsen</title>
<script>
   letter1 = prompt('Mata in begynnelsebokstaven ' +
                    'till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

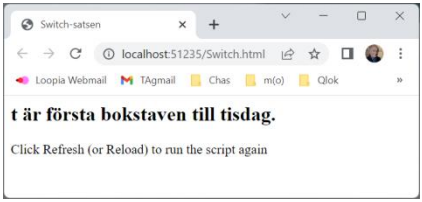
nar man i den utan att likhet mellan uttrycket och konstanten som finns i de efterföljande **case**-satserna testas. Om **switch**-satsen ska välja endast ett enskilt värde bland flera, borde varje **case** avslutas med **break**.

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**.

7.5 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**

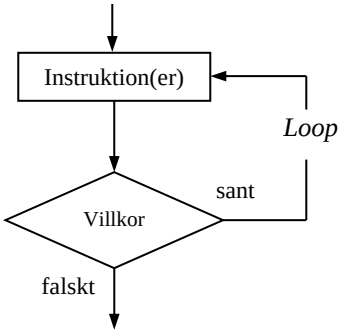
Efter-testad repetition

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:

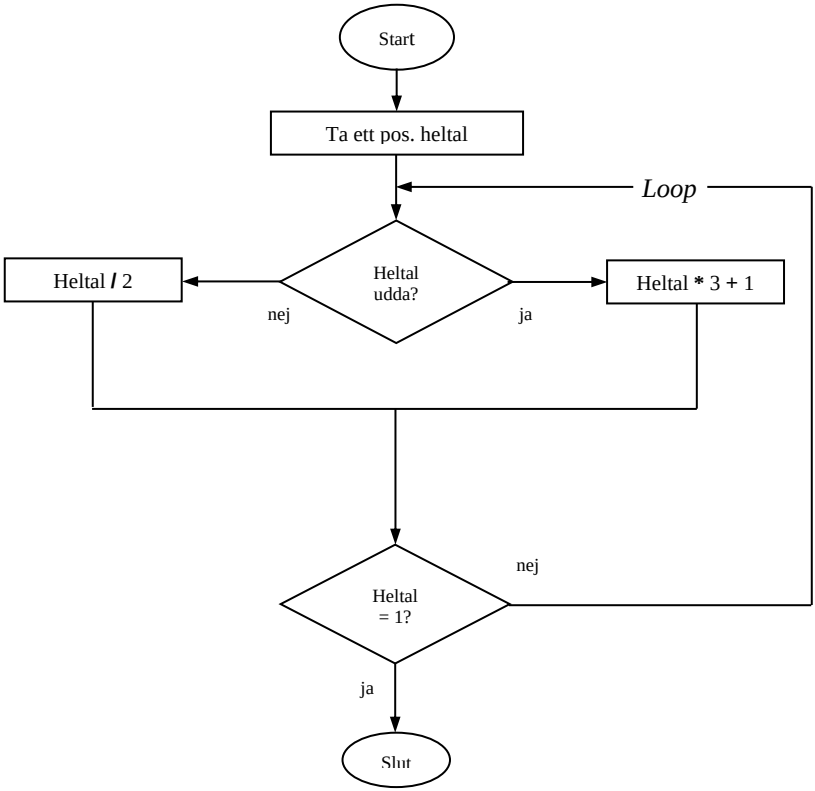
Collatz algoritmen

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.

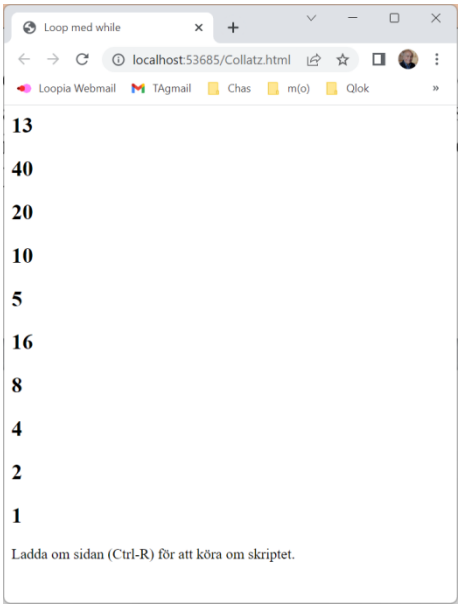
Pseudokoden till Collatz algoritmen

```
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:

```
1  <!-- Collatz.html
2       Läser in ett pos. heltal, tar det gånger 3 och adderar 1,
3       om det är udda. Delar det med 2 om talet är jämnt.
4       Upprepar samma sak med resultatet, tills det blir 1.
5       Använder do-sats för repetitionen -->
6  <title>Loop med do-satsen</title>
7  <script>
8     no = parseInt(prompt('Mata in ett pos.heltal')) // Startvärde
9     document.write('<h2>' + no + '</h2>')
10    do                                  // do loop börjar
11    {
12       if (no % 2 == 1)                 // Om no är udda
13          no = 3 * no + 1
14       else
15          no = no / 2
16       document.write('<h2>' + no + '</h2>')
17    } while (no != 1)                   // do loop slutar
18 </script>
19
20 Ladda om sidan (Ctrl-R) för att köra om skriptet.
```

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**:



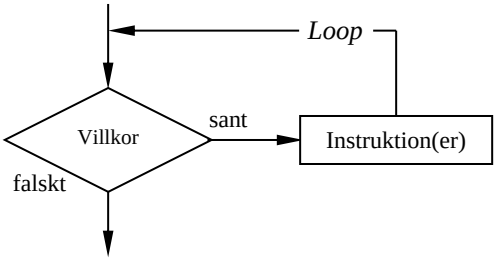
do-satsens arbetssätt, dvs *repetitionen* skiljer sig grundläggande från kontrollstrukturen *selektion* (val) som vi lärde känna tidigare. Medan en selektions alltid går *framåt*, efter den har avgjort valet p.g.a. det styrande villkoret, återvänder en repetition alltid till kontrollstrukturens början, 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 129 där programflödet (pilen) går från avslutningsvillkoret tillbaka, för att utföra det hela en gång till.

7.6 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)



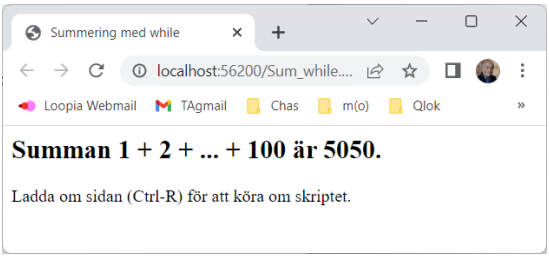
I JavaScript inleds den för-testade repetitionen med det reserverade ordet **`while`** och skrivs generellt på följande sätt:

```
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 149).

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

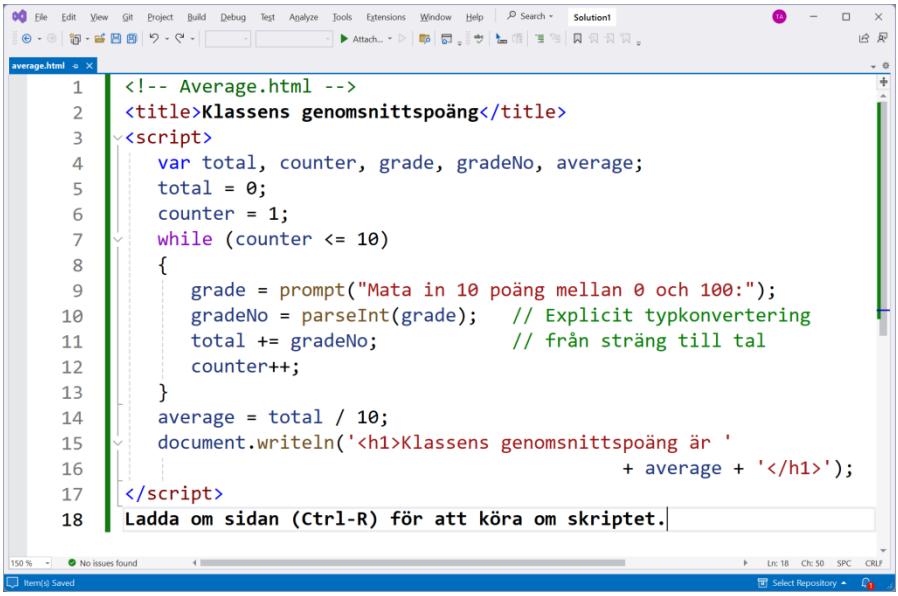
I programmet **Sum_while** är **while**-satsens avsutningsvillkor **term <= 100**. Om detta villkor vore sant från början och förblev sant hela tiden, skulle satserna på raderna **10-11** att utföras i all evighet, vilket kallas för *evighetsloop*.

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.

7.7 Räknar-styrd repetition

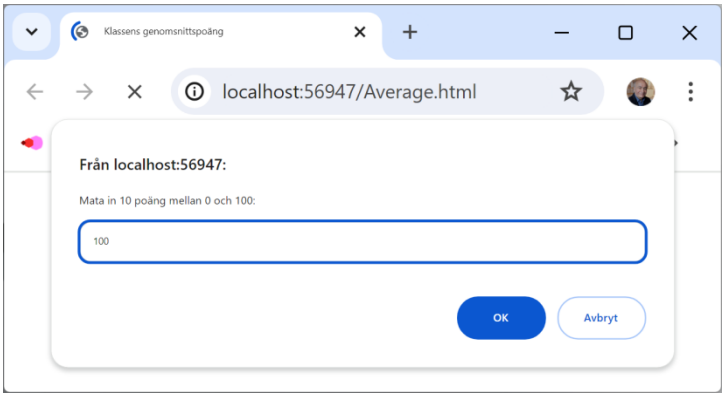
Som exempel för repetitioner väljer vi **while**-satsen. Programmet **Average** kombinerar summeringen av termer (programmet **Sum_while**, sid 132) med inläsning av data. Man beräknar genomsnittet (medelvärdet) av elevernas poäng i en klass:



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.

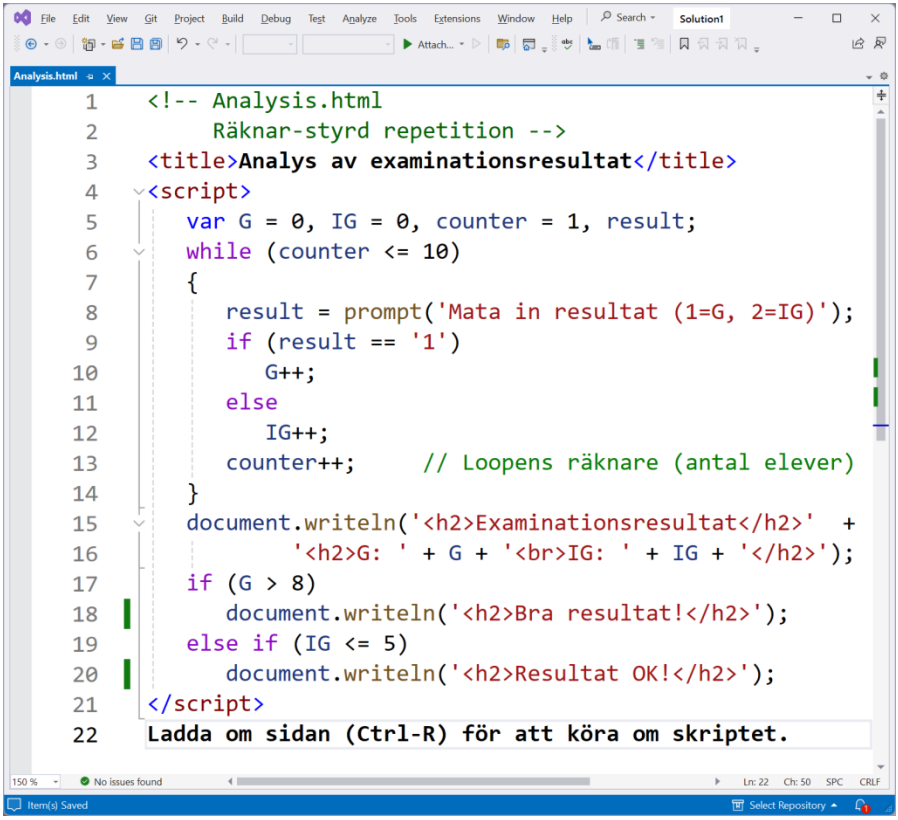
En annan nyhet i programmet **`Average`** på förra sidan är att vi för första gången deklarerar våra variabler med **`var`** på rad **`4`**, vilket inte är obligatoriskt i JavaScript. Vi kommer att fortsätta med det, när våra program blir längre och behöver struktureras mer. Strukturering är redan avgörande 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`**.

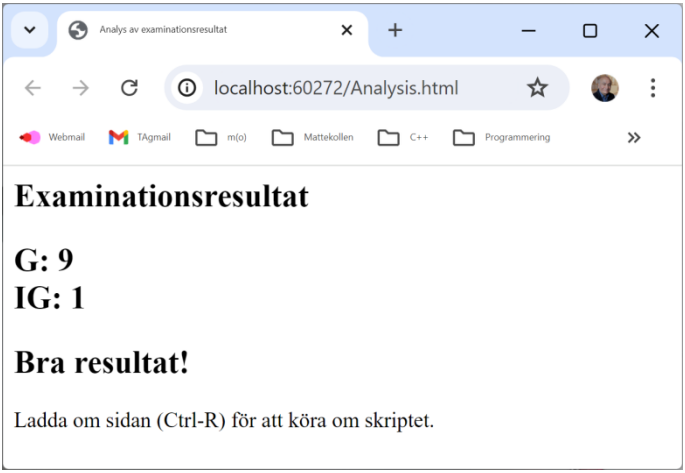
En speciell roll spelar variabeln **`counter`** som tar reda på antal varv. Den initieras *före* och uppdateras *i* loopen, för att användas i avslutningsvillkoret och förhindra evighetsloop. Det är **`counter`**n och antalet 10 som vi har fastskrivit i koden som styr **`while`**-loopen och därmed antalet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

Analys av examinationsresultat

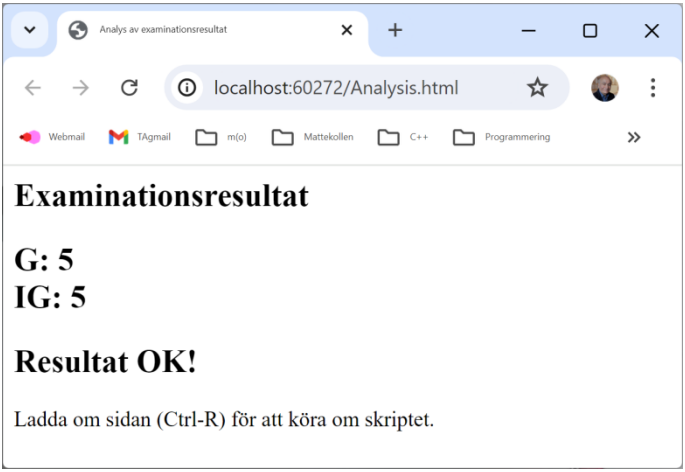
Ett annat exempel på räknar-styrd repetition är programexemplet **`Analysis`** på nästa sida. Här försöker man att få en sammanfattning eller analys av 10 elevers resultat i ett prov. Beroende på analysen som baseras på en policy skrivs ut vissa slutsatser.



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:

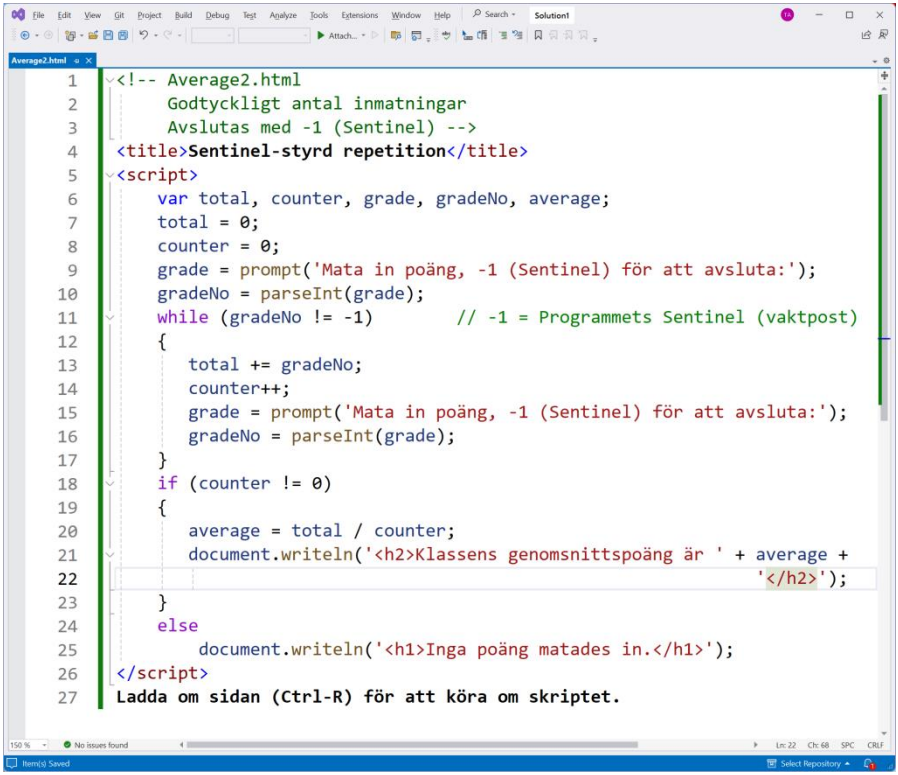


En övning i räknar-styrd repetition vore att byta ut loopens räknare **`counter`** i programmet **`Analysis`** med summan **`G + IG`** av alla poäng, för att spara en variabel och förkorta koden. Ev. blir det nödvändigt att även justera **`while`**-loopens avslutningsvillkor. Denna uppgift är överlåten till övn 3.15 (sid 149).

7.8 Sentinel-styrd repetition

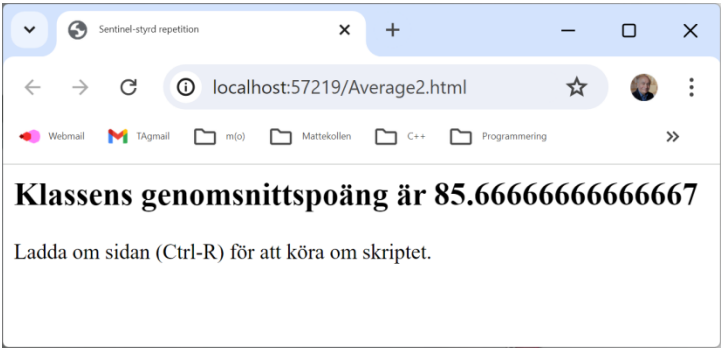
Programmet **Average** (sid 134) har en stor nackdel: Antalet inmatningar är statiskt. **while**-loopens antal varv är hårdkodat. Bara klasser med 10 elever kan använda programmet. Självklart är det önskvärt att användaren ska kunna bestämma antalet inmatningar, inte vi som kodar. Programmet borde vara 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 149).

7.9 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**

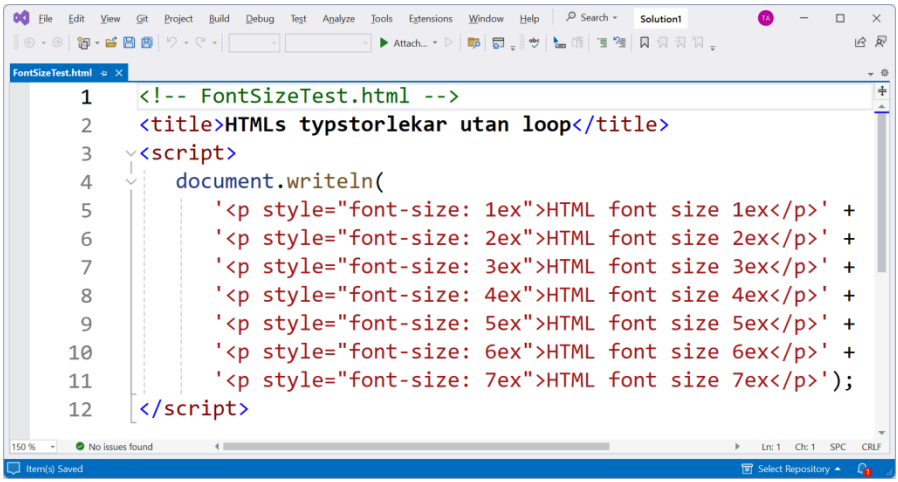
Ett annat exempel på ett **p**-element som är lite mer invecklat är:

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

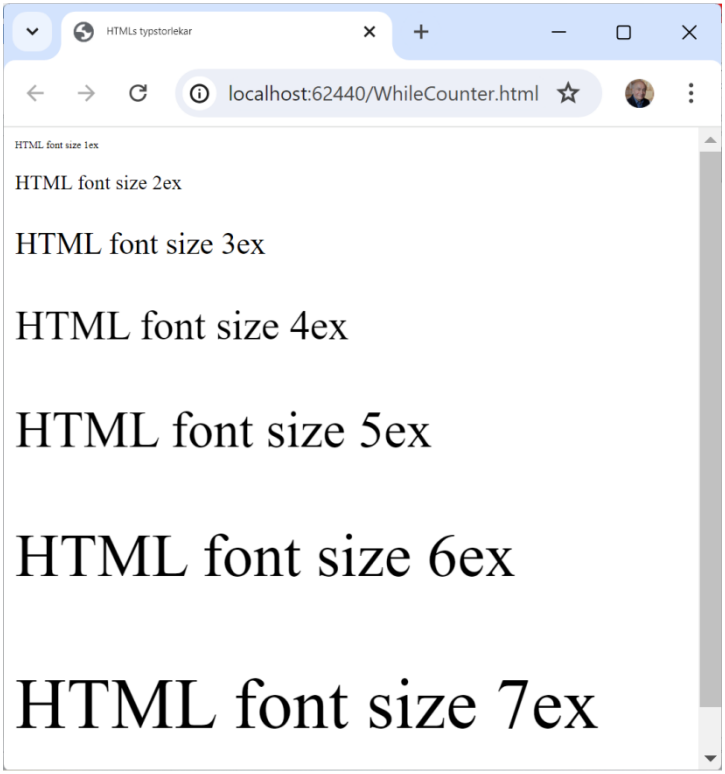
Innehållet i det här **p**-elementet som skrivs ut, är texten **HTML font size 4ex**, eftersom det är den som står mellan start- och sluttagg. Men starttaggen har fått ett nytt utseende: den har utvidgats och fått ett s.k. *attribut*. Attributets *namn* är **style** och attributets *värde* är **"font-size: 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.

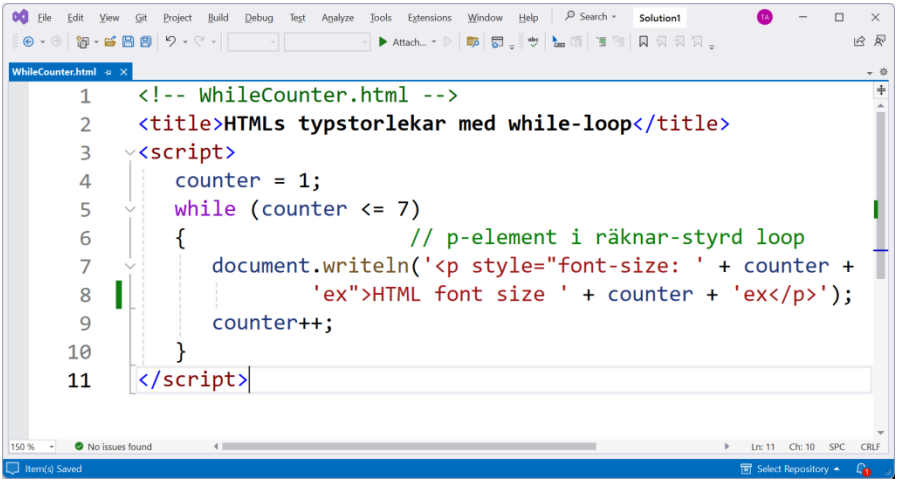
I scriptet **FontSizeTest** på nästa sidan skriver vi ut alla 7 typsnitt utan loop. Sedan ska vi gå över att skriva om koden till en **while**-loop.



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.

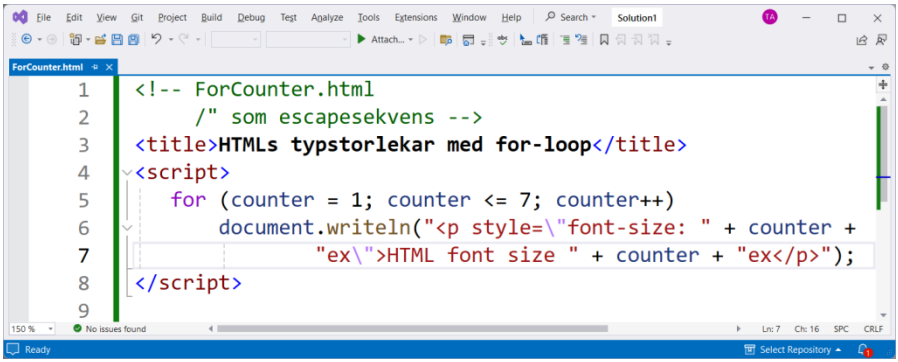
Vid den här konstruktionen har vi dragit nytta av att vi i JavaScript har möjligheten att markera strängar både med citationstecken och apostrofer. Hade det funnits endast ett tecken för strängmarkering, hade vi varit tvungna att använda *escapese-*

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.

7.10 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 141.

Summering med for

Följande script **Sum_For** som är en ren översättning av scriptet **Sum_while** (sid 132) 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 132), blir kortare. Fördelen är effektiviteten, medan nackdelen är läsligheten.

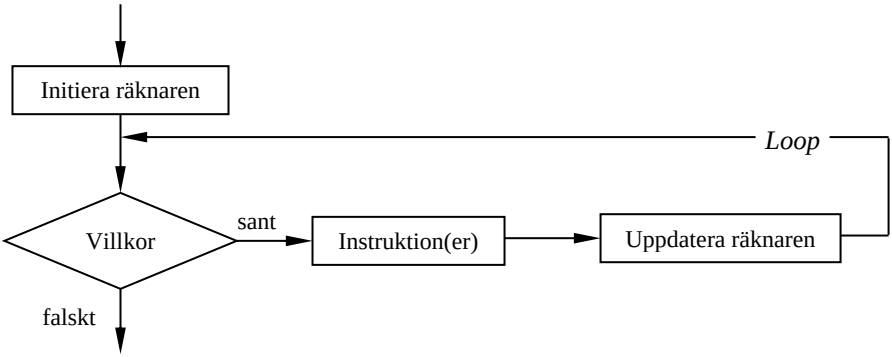
Så här ser det ut när vi kör scriptet.

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.

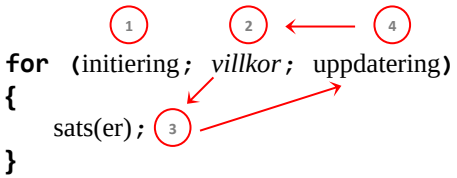
Pseudokoden

```
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:



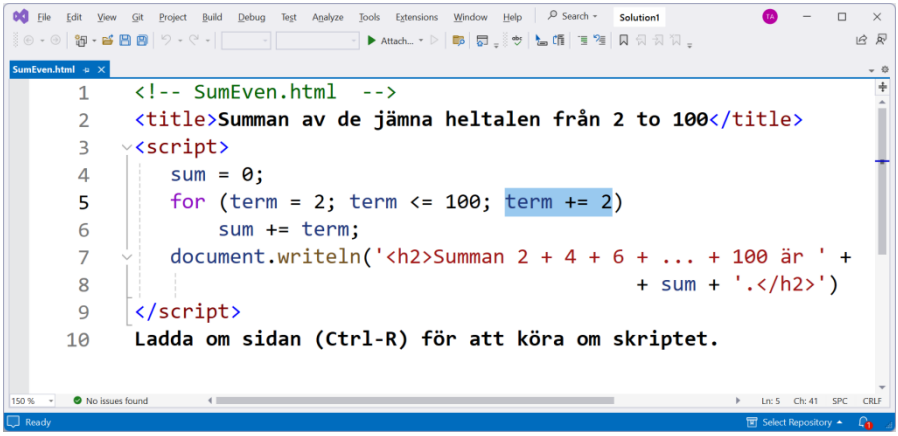
De rödmarkerade ringarna och pilarna samt numreringen ska visa i vilken *ordning* de respektive delarna utförs. Denna ordning är nämligen inte identisk med kodbitarnas ordning. Pilarna markerar loopens förlopp. Initieringen görs endast en gång och ingår ej i loopen.

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.

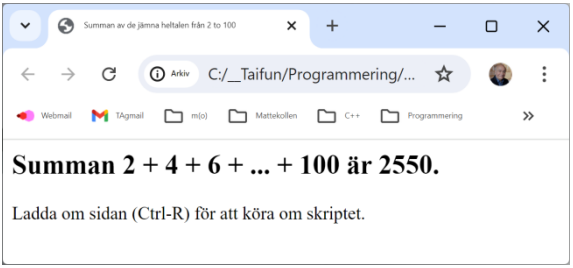
Räknaren sätts före repetitionen till ett önskat startvärde, för det mesta något heltal, ofta **1**. Detta kallas *initiering* av räknaren dvs den allra första tilldelningen av ett värde till räknaren. Sedan testas ett villkor där man brukar lägga in ett önskat *slutvärde* på räknaren. Därmed är antalet 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 145) 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 7

7.1 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.

7.2 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.

7.3 Vidareutveckla programmet **`Max`** (sid 118) så att det läser in *fyra* tal, hittar och skriver ut det största. Vilken ändring i koden leder till det minsta talet?

7.4 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.

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

7.6 Ersätt i programmet **`SimpleIf`** (sid 115) 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.

7.7 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`**.

7.8 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 122).

7.9 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 …”.

7.10 Modifiera programmet **Collatz** (sid 130) genom att ersätta **do**-loopen med en **while**-loop. Modifiera även algoritmens pseudokod och flödesschema, så att de återpeglar algoritmens implementering med **while**-loopen.

7.11 Ändra koden i programmet **Collatz** (sid 130) så att körningen genererar en evighetsloop.

7.12 Modifiera programmet **Sum_while** (sid 132) genom att ersätta **while**-loopen med en **do**-loop.

7.13 Generalisera programmet **Sum_while** (sid 132) 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**.

7.14 Ändra koden i programmet **Sum_while** (sid 132) så att körningen genererar en evighetsloop.

7.15 Modifiera programmet **Analysis** (sid 136) 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.

7.16 Programmet **Average2** (sid 138) 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.

7.17 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?

7.18 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.

7.19 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.

7.20 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.

7.21 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.

7.22 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.

7.23 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.

7.24 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

7.25 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.

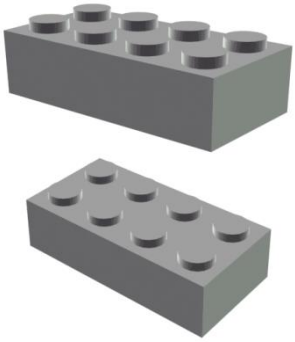
Kapitel 8

Funktioner

8.1 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

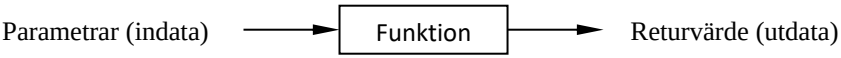


De flesta har väl någon gång som barn, eller tillsammans med sina barn, byggt ett hus, en bil eller liknande med Lego-bitar. Efter ett tag har huset kanske rasat och nya tekniska underverk har konstruerats. Men även de har någon gång plockats isär. Det enda som blivit kvar är själva Lego-bitarna som 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*.

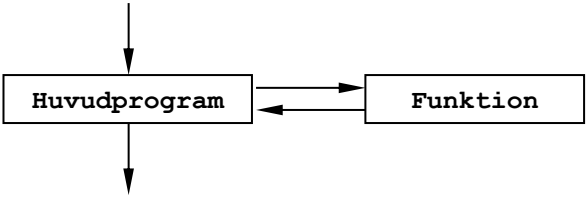
En funktion kan ha inga, en eller flera parametrar. Den kan ha inget eller endast *ett* returvärde, dvs en funktion kan inte ha flera returvärden, vilket är ett arv från matematiken. Både parametrarna och returvärdet kan vara tal, tecken, strängar, sanningsvärden, objekt eller andra datatyper. Funktionen 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

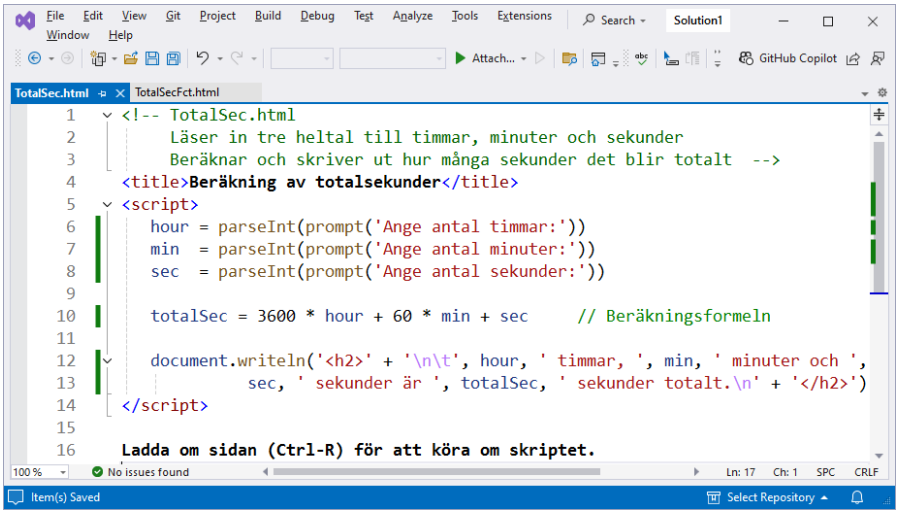
```
1  <!-- MaxFct.html
2       Definierar och anropar funktionen max() som bestämmer
3       det största bland tre tal  -->
4  <title>En egendefinierad funktion</title>
5  <script>
6
7     function max(a, b, c)   // Definierar funktionen max()
8     {
9        tmp = a              // Antar att a är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

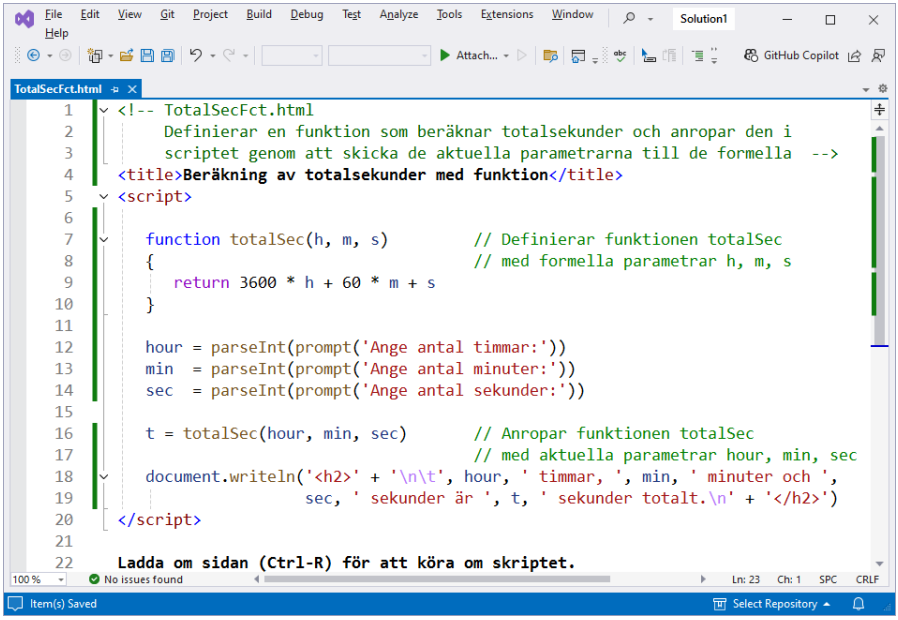
Rad **7** kallas för funktionens *huvud* och inleds med det reserverade ordet **function** (sid 88). Funktionens *namn* är **max()**. Parentesen **(a, b, c)** kallas för *parameterlistan*. **a, b** och **c** är funktionens *formella parametrar*, medan **no1, no2** och **no3** som står i funktionsanropet (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.

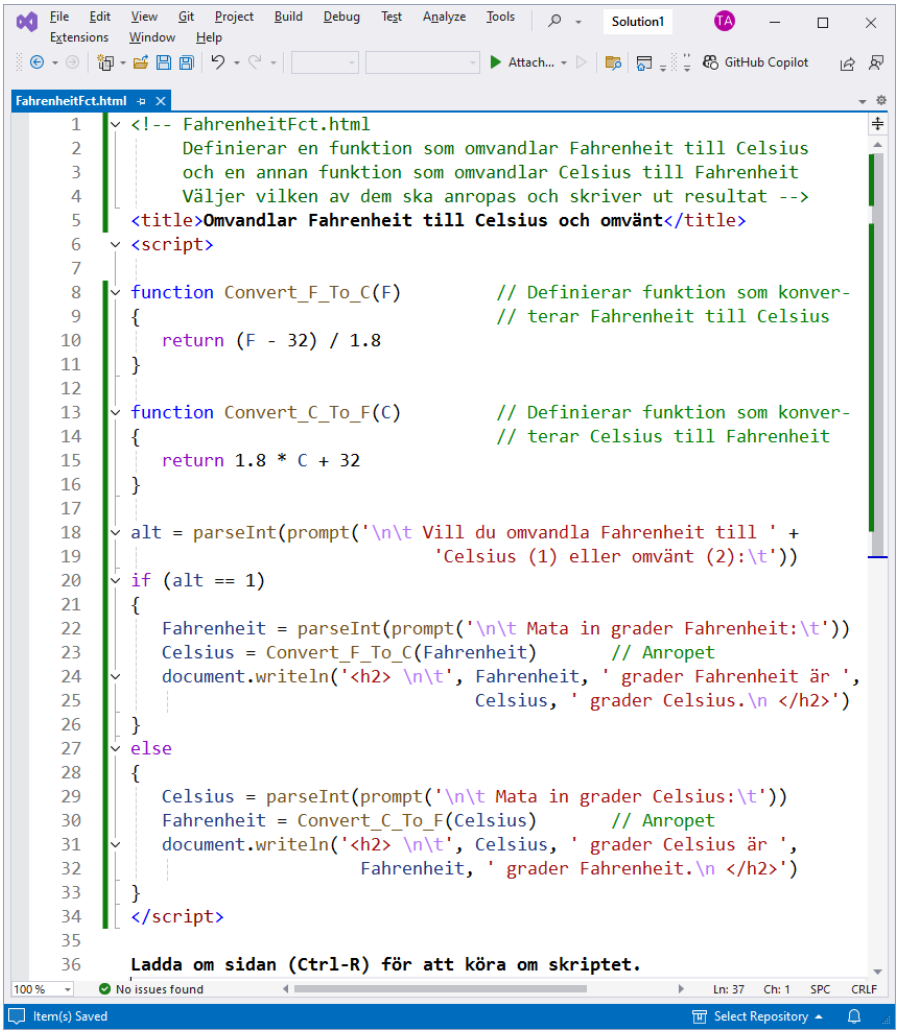
8.2 Formella och aktuella parametrar



```
<!-- 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.
```

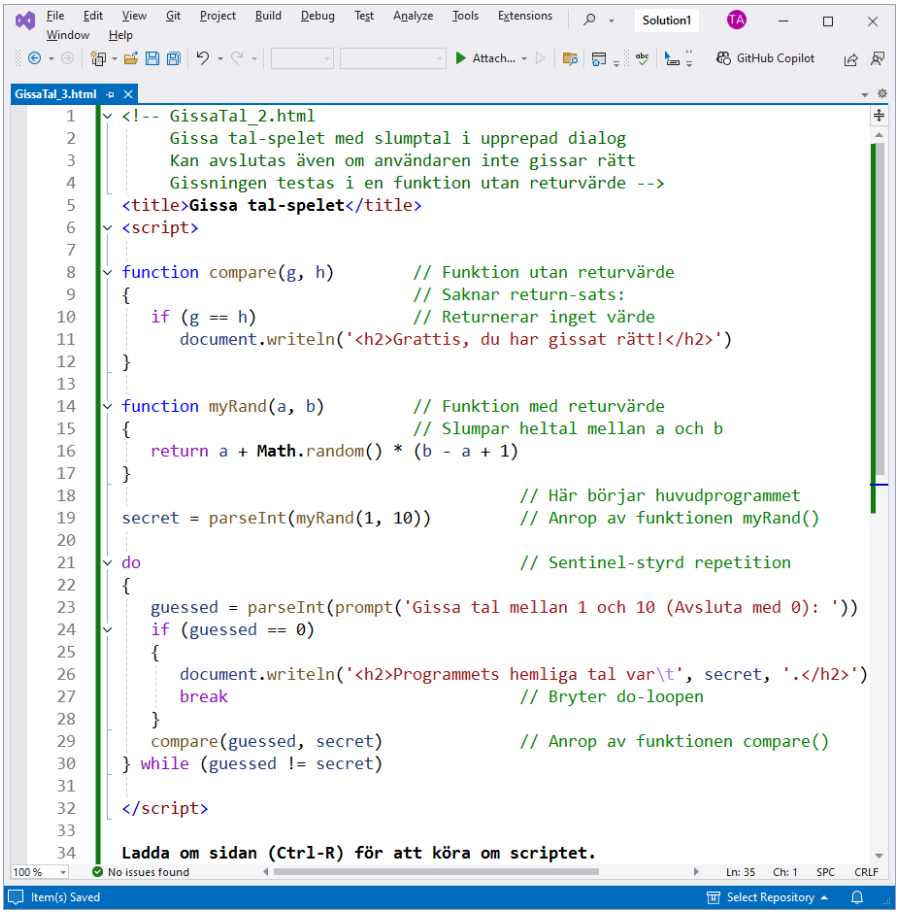

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Solution1
Attach...
GitHub Copilot
FahrenheitFct.html
<!-- FahrenheitFct.html
Definierar en funktion som omvandlar Fahrenheit till Celsius
och en annan funktion som omvandlar Celsius till Fahrenheit
Väljer vilken av dem 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

8.3 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.

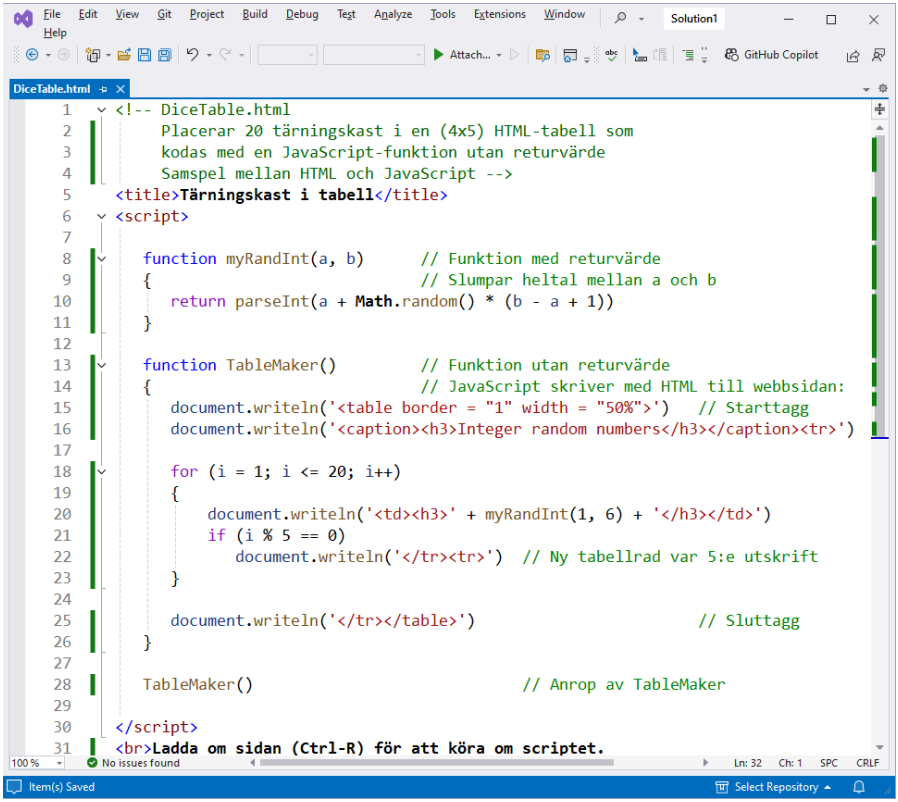


Man hade även kunna skriva en tom **`return`**-sats i funktionen som skulle i så fall avsluta funktionen. Testa gärna att lägga in en tom **`return`**-stas.

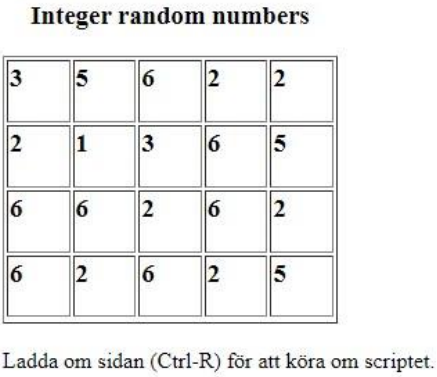
Så här kan vi sammanfatta:

> 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.

8.4 Tärningskast i tabell



Funktionen TableMaker()



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 8

8.1 Skriv om scriptet **MaxFct** (sid 158) 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.

8.2 Modifiera scriptet **TotalSecFct** (sid 159) 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?

8.3 Ta scriptet **FahrenheitFct** (sid 160) 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 160).

8.4 Modifiera scriptet **GissaTal_2** (sid 161) 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.

8.5 a) Modifiera scriptet **DiceTable** (sid 163) så att det producerar en (10 x 7)-tabell över tärningskast. Bibehåll scriptets överordnade struktur.

b) Extrahera från scriptet **DiceTable** 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 142.

8.6 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.

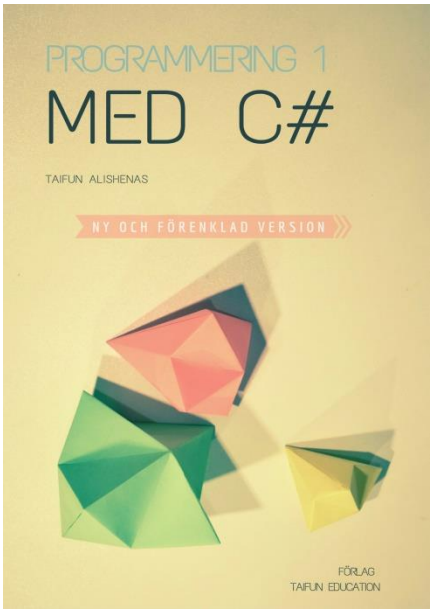
8.7 Skriv de fyra räknesätten och heltalsdivisionen samt modulo som funktioner i JavaScrip. Läs in två heltal. Anropa funktionerna och skriv ut resultaten så att du får t.ex. följande utskrift när du läser in `5` och `3`:

```
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
```

8.8 Modularisera scriptet **IfElse** (sid 121) genom att flytta koden 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.

8.9 Modularisera programmet **GissaTal** (sid 124) genom att definiera if-else-stegen som en funktion.

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

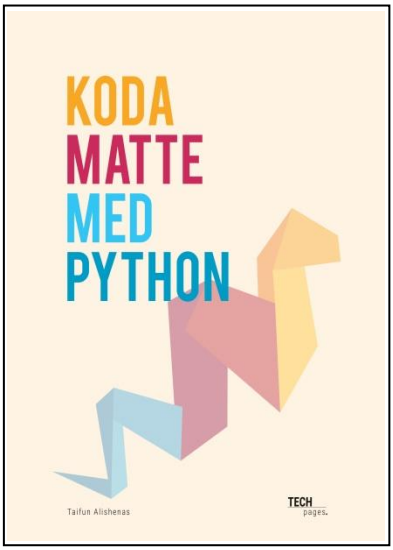
www.taifun.se **Ladda ned gratis smakprov.**

Koda matte med Python

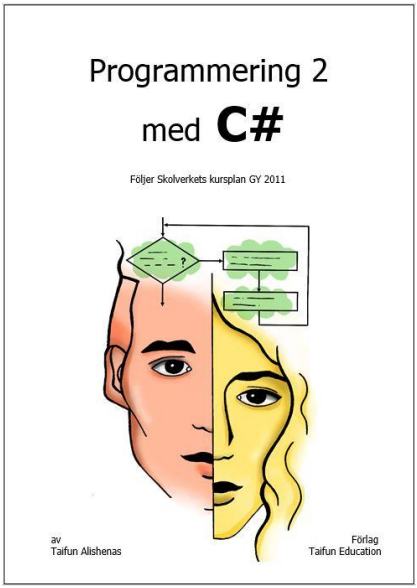
Programmering i matematik

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 och Matematik åk 7-9.



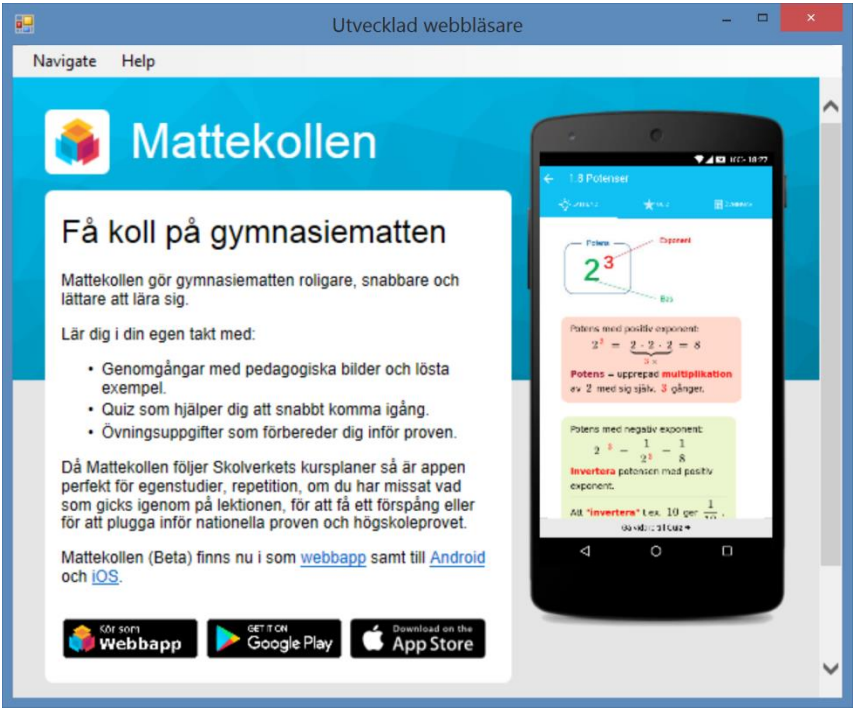
Programmering 2 med C#



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*.