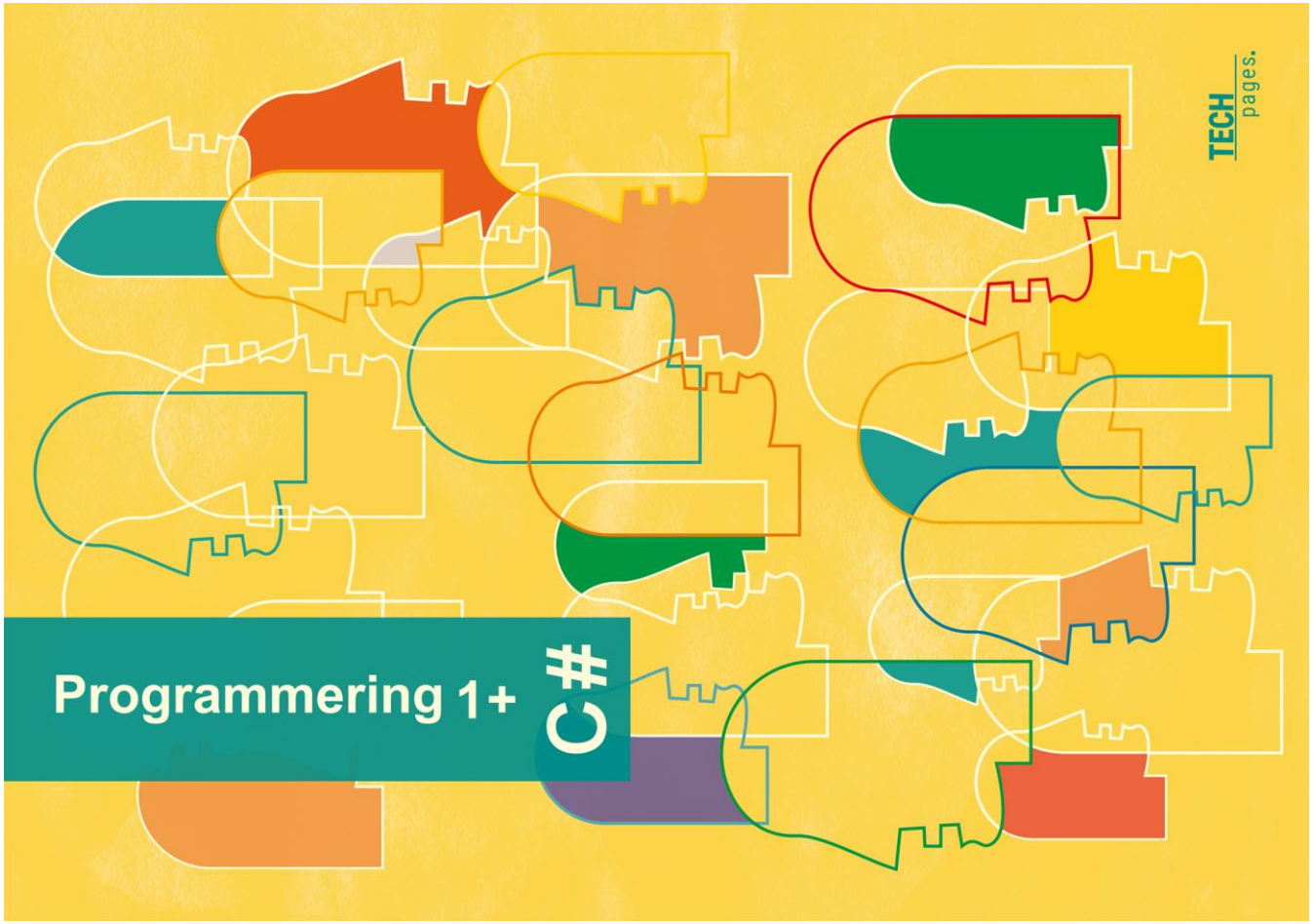

TECH pages.
Programmering 1+
C#

Programmeringsspråket C# (uttalat ”si sharp”) bygger på allt som är bra i C, C++ och Java och lägger till en hel del nytt. Tillägget ++ (ökning med 1) i C++ betyder att man lagt till 1 utvecklingssteg till C, men behållit den gamla kärnan C. Tillägget # i C# däremot står för en ny- och vidareutveckling som inte längre bär på ballasten från C och bjuder på en högre programmeringsupplevelse. I musiken är C# (eng. *C sharp*, sv. *Ciss*) en halv ton högre än tonen C, vilket har inspirerat namngivningen av programmeringsspråket C#, där man med tillägget # syftar på att man ”höjt” språket C till C#.

Koden är enkelt att lära sig. Har man kommit över en viss tröskel – vilket denna bok hjälper dig att göra – blir det roligt att skriva program som kan lösa praktiska problem och förenkla människans arbete – allt för att ha mer tid över för annat i livet. Det är just det IT och programmering handlar om. Dessutom får man genom att lära sig programmering, en bättre förståelse för hemligheterna bakom IT.

Andra programmeringsböcker publicerade i samma förlag:

- *Programmering **2** med **C#*** – En fortsättning på denna bok.
- *Programmering **3** med **C#*** – En fortsättning på *Programmering 2 med **C#***.
- *Programmering **1** med **C#*** – En lightversion av denna bok.
- *Programmering 1 med **Java*** – En introd. till programmeringens grunder.
- *Programmering för nybörjare med **C++*** – Introd. till programmering.
- *Koda matte med **Python***.
- *Programmering i matematik – Tio lektioner* (under bearbetning).

Alla böcker innehåller övningar, projektuppgifter och fullst. lösningar till alla övn.

Tacksam för kritik, synpunkter, påpekande av fel och förslag om korrektur eller förbättringar av innehåll eller form till **`info@techpages.se`** .

Printed in Sweden
Published by TechPages Förlag AB
ISBN 9789197420471

`www.techpages.se`

Programmering 1+

med **C#**

Täcker Skolverkets ämnesplan för kursen Programmering 1

Med övningar,
fullständiga lösningar
&
projektuppgifter

Titel: Programmering 1+ med C#
ISBN 9789197420471

Förlag: TechPages Förlag AB
info@techpages.se

www.techpages.se

Skrivet av TechPages Förlagets författarkollektiv.
Tryckeri: Eprint, Stockholm
Februari 2022

Vad boken handlar om

Välkommen till programmeringens spännande värld! När man tröttnat på att bara använda program som andra skrivit, är det dags att börja programmera själv. Visst är det roligare att köra en bil än att bara åka med. Det är kreativiteten och det fria skapandet som lockar. Programmering kan vara en naturlig fortsättning för dig som hittills endast har mailat, surfat eller lyssnat på musik på datorn och nu vill veta mer om vad som händer bakom kulisserna i en dator. Man lär sig nämligen på ett helt nytt plan hur datorer fungerar när man programmerar själv. Dessutom kan man testa sina egna, nya idéer.

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 också bidragit till att förverkliga den urgamla mänskliga drömmen att förenkla mödosamma arbeten. Istället för att plåga sig kodar man en maskin med idéer, för att ha mer tid över för annat i livet.

Meningen med boken är att lära ut programmering. Detta kan dock praktiskt åstadkommas endast genom att skriva och testa program, dvs använda ett programmeringsspråk. I denna bok används C# som medel, verktyg och medium för att presentera programmering. Men medlet är av underordnad betydelse. Målet är att förmedla *tankesättet* och *tekniken* att programmera, oberoende av språk. Har man en gång förstått de grundläggande principer som är gemensamma för alla programmeringsspråk, blir det närmast en teknikalitet att på egen hand lära sig ett nytt språk. Denna bok är en introduktion till programmering som inte bara täcker Skolverkets kursplan för *Programmering 1* utan innehåller även en hel del annat smått och gott från datalogin för att göra pliktlektyren mer intressant, därför 1+. Några förkunskaper inom ämnet förutsätts dock inte.

Vi som skrivit boken har många års erfarenhet av undervisning i programmering, databaser, matematik, numerisk analys och andra ämnen både på skol- och högskolenivå i olika länder. I vårt material eftersträvar vi enkelhet och klarhet som resulterar i strukturerade och logiska program så att man lätt kan se *idén* och förstå *tanken* bakom koden.

I början av våra banor som pedagoger antog vi att vissa begrepp, sammanhang och förutsättningar var självklara, men den dagliga undervisningen i klassrum fick oss snart på andra tankar. Våra elevers frågor, kritik och kommentarer fick oss att förstå var de begreppsmässiga luckorna i våra resonemang fanns. De var våra *elever* i programmering, matematik osv. men blev våra *lärare* i pedagogik.

Röda trådens pedagogik

Böcker i tekniska ämnen är ofta rena faktasamlingar vilket kan vara en konsekvens av ämnenas komplexitet. När de är skrivna för experter behöver det inte heller vara

av nackdel. Men när nybörjare ska introduceras till ett ämne blir det problem om boken inte kombinerar kunskap med pedagogik. Då blir läroböcker ofta en ambitiös samling fakta som inte framhäver det väsentliga. Oftast handlar det om elementär kunskap som experten tar för given, men blir den bristande länken i förståelsekedjan hos nybörjaren. Bokens ambition är att förverkliga den röda trådens pedagogik genom att stiga ned till nybörjarens kunskapsnivå och steg för steg bygga upp kunskapens hus av små lösa, logiskt härledbara pusselbitar så att till slut allt faller på plats.

Learning by doing – teaching by example

Programmering är i allra högsta grad ett praktiskt ämne. Därför är *Learning by doing* det enda sättet att lära sig det. I detta avseende liknar programmering bilkörning. Du kommer aldrig att lära dig programmering enbart genom att läsa böcker. Men att bara ”pröva sig fram” räcker inte heller. Ämnet är alltför omfattande. En handledning behövs, inte minst i början, som kombinerar sakkunskap med pedagogik, belyser det väsentliga och tillämpar ett helhetskoncept.

Boken håller inga abstrakta lektioner utan använder *teaching by example* dvs exempelorienterad teoriundervisning i kombination med praktiska övningar: All teori, även de mest abstrakta begreppen åskådliggörs med enkla praktiska exempel. Fullständiga små program med körexempel gås igenom i detalj för att förmedla viktiga koncept inom programmering. Ännu mer material presenteras i övningarna inkl. praktiska programmeringsprojekt. Fullständiga lösningar till alla öningsuppgifter finns i slutet av boken.

Gör så här:

- Ladda ned och installera programvaran Visual Studio (sid 12-13).
- Gå igenom boken avsnitt för avsnitt, program för program. En fullständig programförteckning finns i slutet av boken (sid 253).
- Gör övningarna i slutet av varje kapitel. Kolla lösningarna (sid 219).
- Genomför programmeringsprojekt som finns bland övningarna.
- Pröva dina idéer i egna program och återvänd till teorin.

All form av kritik, korrekturanmärkningar såväl som förslag till förbättringar av både form och innehåll tas tacksamt emot på adressen **`info@techpages.se`**.

Stockholm februari 2022 TechPages Förlag

Innehållsförteckning

Kapitel 1

C# programmeringens miljö

1.1 Installation av Visual Studio

1) Gå till webbadressen: https://visualstudio.microsoft.com/vs/

Webbsidan **Visual Studio 2019** visas. Gå med musen över den lila knappen:

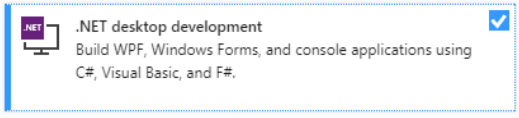
Download Visual Studio 

En dropplista dyker upp. Välj Community 2019 .

2) Installationsfilen vs_community__12….exe laddas ner. Strunta i resten på den nya webbsidan. Dubbelklicka på den just hämtade installationsfilen. Svara Ja på frågan om du ska tillåta att den här appen får göra ändringar på din dator. Klicka på Continue om det dyker upp rutan Visual Studio Installer.

3) Det tar ett tag tills Visual Studio Installer öppnar ett stort vitt fönster dyker upp med den lilla rubriken Installing – Visual Studio Community 2019 … och den blåmarkerade fliken **Workloads**. I den finns till vänster ett antal rutor. Leta efter följande ruta:

4) Markera rutan med rubriken .NET desktop development genom att bocka den lilla blå rutan i det övre högra hörnet.



5) Klicka sedan i det stora vita fönstret Installing – Visual Studio Community 2019 … på knappen **Install** i det nedre högra hörnet. Detta kan ta ett tag, ev. ganska länge – beroende på din dators prestation. Du kan själv avgöra om du under tiden vill besvara några frågor av mindre intresse för installationen. Eller välj t.ex. Do'nt show … resp. Not now. Starta om din dator om du uppmanas till det.

6) När du lyckats med installationen startas Visual Studio antingen automatiskt eller du kan göra det själv från Start-knappen. Stäng rutan Visual Studio Installer. Följande eventualiteter kan dyka upp:

- Om du uppmanas att skapa ett Microsoft-konto (Sign in), gör det. Det är gratis, går fort och är inte problematiskt. Anteckna ditt lösenord för senare uppdateringar.

- Om du får upp en ruta med bl.a. dropplistan Development Settings välj C#. Om alternativet inte finns låt General stå där. Klicka sedan på knappen Start Visual Studio.

7) Beroende på vilken typ av applikation du vill skapa fortsätt enligt instruktionerna på sid 15 för Console Application eller sid 23 för Windows Forms Application.

1.2 Konfiguration och användning av Visual Studio

Efter lyckad installation av Visual Studio enligt anvisningarna i förra avsnitt – eller om du har tillgång till en redan färdiginstallerad version av Visual Studio – kan du i detta avsnitt läsa hur man *använder* programvaran. För att kunna göra det, närmare bestämt kompilera och exekvera C#- program krävs nämligen en korrekt konfiguration av Visual Studio, vilket i början kan verka lite invecklad. Anledningen till det är att Visual Studio är en integrerad programutvecklingsmiljö (IDE) som är skapad för professionella utvecklare och därför är ganska stor och komplex. Därför vill vi i denna beskrivning hålla oss till det absolut minimala som är nödvändigt för att klara av miljön och kunna koncentrera oss på själva *språket C#*. De viktigaste momenten är följande:

- Att välja rätt typ av applikation
- Att skapa ett projekt (helst endast ett till alla C# program)
- Att lägga till en C#-källkodsfil till projektet
- Att kompilera och exekvera C#-koden i projektet

Det finns olika typer av C# program, även kallat *applikation.*

Två olika typer av applikation

I Visual Studio finns det många olika typer av applikation. Av dessa behandlas endast två här:

1. ***Console Application*** är ett C# program vars körresultat är en utskrift i textform som hamnar i *Windows Kommandotolk*, den s.k. *konsolen*, ett svart fönster, ibland även kallat för DOS-fönstret. Ett sådant program har inga grafiska komponenter. Programexemplen i denna bok domineras av Console Applications.

2. ***Windows Forms Application*** involverar både text och grafik och producerar fönster samt dialogrutor av olika slag. Med sådana program kan användaren kommunicera via grafiska gränssnitt. I denna bok introduceras de på sid 23. Fler Programexempel av typ Windows Forms Applications finns i boken *Programmering 2 med C#*.

Följande tre steg måste alltid tas för att kunna köra ett program i Visual Studio – vare sig det är en *Console Application* eller en *Windows Forms Application*:

1. Att skapa eller öppna ett befintligt projekt
2. Att lägga till en C#-källkodsfil till projektet
3. Att kompilera och exekvera

För program av typ *Console Application* går vi igenom dessa tre steg på nästa sida. Men först: Vad exakt är ett projekt i Visual Studio och varför behöver vi det?

Projekt i Visual Studio

För att kunna köra ett C# program i Visual Studio måste koden infogas i ett s.k. *projekt*. Ett *projekt* är en samling filer – alltid själva C#-källkoden, men också andra relaterade filer inkl. ev. bilder – som sammanlagt utgör ett C# program. Denna samling filer bildar både en fysisk mapp på hårddisken och en virtuell arbetsplats i Visual Studio. De kommunicerar med varandra hela tiden när vi utvecklar och testar våra program. Visual Studio kan endast kompilera och köra C# program som är inbäddade i projekt, även om det är det enklast tänkbara program som består av endast en fil. Det är inte möjligt att kompilera C#-källkod utanför ett Visual Studio-projekt. Så, innan vi kan börja skriva C#-kod måste vi antingen skapa ett nytt eller öppna ett befintligt projekt.

Den övergripande termen till projekt i Visual Studio är *solution*. Dvs flera projekt kan samlas i en solution. Självklart kan en solution även bestå av ett enda projekt. Vi kommer till att börja med inte att använda flera projekt i en solution utan endast *ett* projekt. Ändå kommer vårt projekt att automatiskt vara paketerat i en solution.

1.3 C# Console Applications

Starta Visual Studio från Windows *Start*-meny genom att klicka fram dig till:

Start → Visual Studio 2019

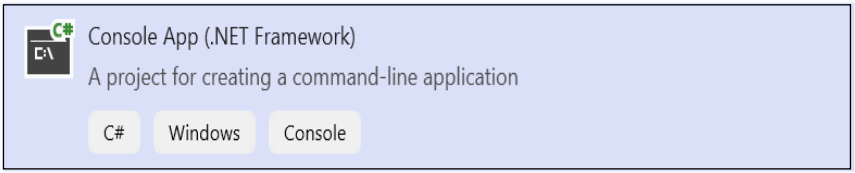
Ett vitt fönster öppnas med rubriken Visual Studio 2019. I kolumnen till höger under rubriken Get started finns ett antal rutor.

1. **Att skapa eller öppna ett befintligt projekt:** Beroende på om vi vill skapa ett nytt eller öppna ett befintligt projekt, tar vi ett av följande alternativen **a)** eller **b)**:

   a) Om vi vill skapa ett nytt projekt – och det vill vi nu – klickar vi i det vita Visual Studio 2019-fönstret på rutan

   Create a new project

   En ny dialogruta dyker upp med rubriken Create a new project. Scrolla ned den högra kolumnen i dialogrutan Create a new project och leta efter en ruta med rubriken Console App (.NET Framework) som ser ut så här:

   OBS! Det kan vara lite svårt att hitta denna ruta, eftersom det finns många alternativ och många rutor som ser likadana ut. Det är lättgjort att man väljer fel ruta. Var extra noga med att du har **C#** ikonen och den exakta rubriken:

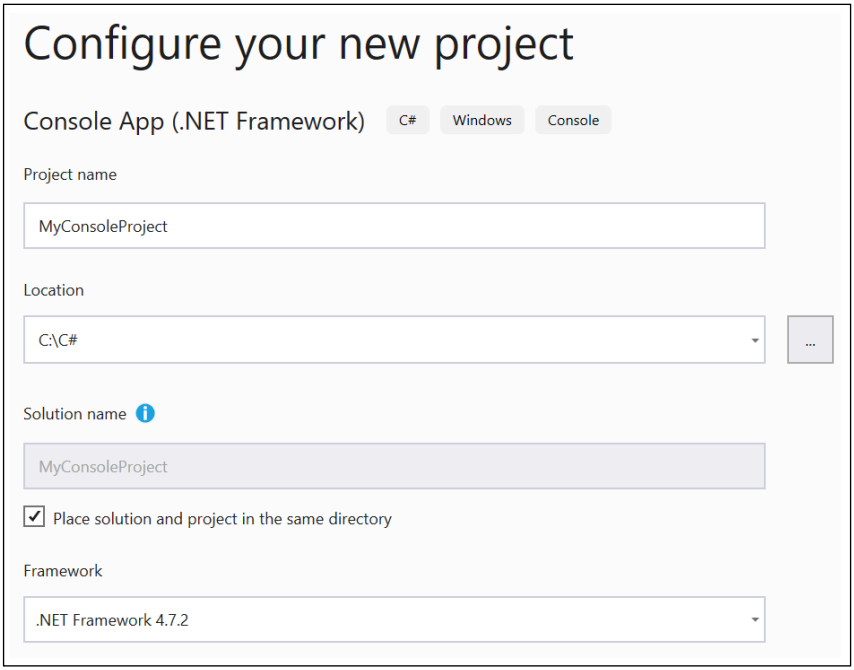
   Console App (.NET Framework)

   Och inget annat! Annars kommer våra program inte kunna köras med de instruktioner som ges i boken. Och då kommer hela installation av Visual Studio att behöva göras om.

   Markera rutan ovan. Klicka sedan på knappen Next.

   En ny dialogruta dyker upp med rubriken Configure your new project.

   Fyll i den uppgifterna enligt följande:



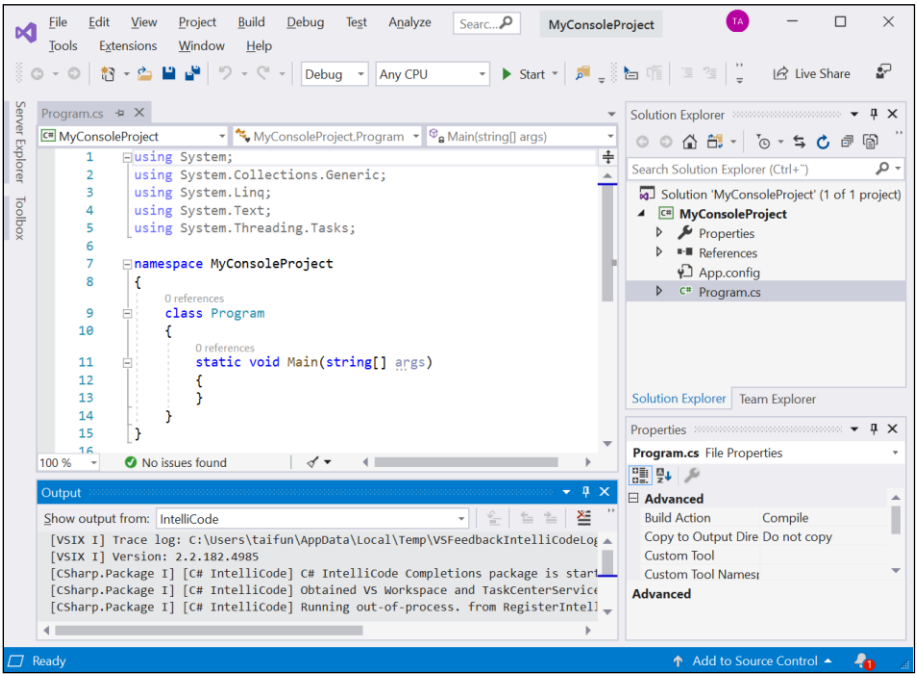
Dvs i den övre delen av dialogrutan döper vi vårt projekt till MyConsoleProject. I textrutan Location anger vi den fullständiga sökvägen till den mapp vi vill placera vårt projekt i. Låt oss säga vi vill samla våra C# program i en mapp som vi kallar C# och placerar i enheten C:\ på vår dator. I så fall anger vi som Location C:\C#. I denna mapp kommer nu projektmappen MyConsoleProject placeras. Visual Studio skapar automatiskt både den nya mappen och projektfilen. Bocka för den lilla rutan Place solution and project in the same directory. Klicka på knappen Create. Gå till punkt **2**.

b) Om vi vill öppna ett redan befintligt projekt – det gör vi kanske senare – klickar vi i det vita Visual Studio 2019-fönstret på rutan

Open a project or solution

Vi får upp dialogrutan Open Project/Solution. För att öppna det projekt vi vill jobba med, navigerar vi i datorns filsystem till projektmappen och öppnar där filen med ändelsen `.csproj`. Gå till punkt **2**.

2. Att lägga till en C#-källkodsfil till projektet: Efter att ha lämnat dialogrutan Configure your new project med Create-knappen enligt **1. a)** eller dialogrutan Open Project/Solution med Open-knappen enligt **1. b)** öppnas projektet. Ett grafiskt gränssnitt kommer upp som liknar en webbsida bestående av en massa menyer, flikar, länkar och fönster som ser ut så här:



Man ser ett antal fönster: till höger ovan fönstret Solution Explorer där projektets innehåll visas med ett antal automatiskt skapade filer, bl.a. filen Program.cs som vi har markerat i bilden ovan. Till vänster ser man det stora kodfönstret som visar denna fils innehåll som är en mall för ett C# program. Den är lämplig för dem som vill använda mallen för att snabbt kunna utveckla en applikation. Vi däremot ska lära oss C# från grunden och vill inte använda kod som vi inte skrivit själva. Därför: Markera Program.cs, högerklicka och välj:

Exclude From Project

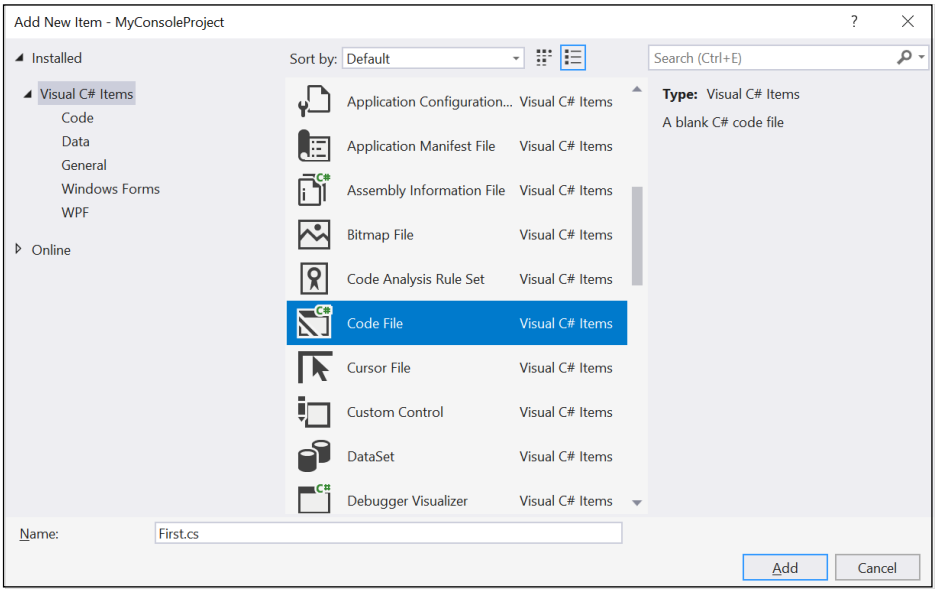
Därmed har vi avlägsnat denna fil från projektet för att kunna infoga vårt eget C# program i projektet. Det finns två alternativ att göra det: Antingen vill vi skapa ett helt nytt program, skriva in koden, spara den i en fil och infoga den i projektet eller vi vill lägga till en redan befintlig fil som innehåller ett C# program, som vi kanske har skrivit tidigare. Vi ska behandla båda varianter och börjar med den första:

a) Att skapa en ny fil och infoga det i projektet:

Markera i Solution Explorer projektnamnet **MyConsoleProject**, högerklicka på det och välj:

Add → New Item…

Dialogrutan Add New Item – MyConsoleProject dyker upp. Scrolla ner fönstret i mitten tills du ser filtypen Code File. Markera Code File i mittfönstret:



Ange i den undre delen av dialogrutan i textrutan Name: First.cs. Därmed har du skapat en fil av typ Code File och döpt den till First.cs. Klicka på Add-knappen. Så snart du gjort det läggs den tomma filen First.cs till projektet. Samtidigt skapas denna fil i projektmappen MyConsoleProject. Och när du i Solution Explorer markerar filen visas till vänster ett stort vitt fönster som du kan använda som en editor för att skriva C#-kod i. Skriv in där t.ex. följande kod:

```
using System;

class First
{
   static void Main()
   {
      Console.WriteLine("\n\tMitt första C#
program!\n");
   }
}
```

Det rekommenderas att bibehålla kodens layout, för att följa *God programmeringsstil*, se sid 47. Visual Studio har stöd för detta. Koden kan sparas och lagras t.ex. i filen First.cs så snart du kompilerar projektet, se punkt **3**. Vi kommer att referera till den med programmet **`First`** som samtidigt är klassnamnet i koden, vilket dock inte är obligatoriskt utan en konvention vi följer.

b) Att lägga till en befintlig fil till projektet:

Har du redan en C#-källkodsfil bland dina filer på hårddisken, markera i Solution Explorer projektnamnet **MyConsoleProject**, högerklicka och välj:

Add → Existing Item…

Dialogrutan Add Existing Item – MyConsoleProject dyker upp som tillåter dig att navigera genom datorns filsystem för att ladda en existerande C#-källkodsfil. Gå till den fil du vill ladda, markera den och klicka på knappen Add i dialogrutan Add Existing Item – MyConsoleProject. I Solution Explorer kan du konstatera att den fil du valde har kommit till projektet MyConsoleProject. Markera den för att se innehållet i kodfönstret till vänster som nu kan användas som en editor.

3. **Att kompilera och exekvera:** Nu när projektet är skapat och innehåller en C#-källkodsfil kan man kompilera det vilket innebär att även källkoden ovan kompileras. Om det inte redan finns ett Output-fönster längst ned på sidan under kodfönstret, klicka i menyraden längst upp på menyn:

View → Output

Du får ett nytt Output-fönster för att kunna se resultatet av kompileringen och även se eventuella kompileringsfel. Akta på vad som skrivs i det när du kompilerar koden från menyraden längst upp med:

Build → Build Solution

Om du får följande meddelande i Output-fönstret har kompileringen gått bra:

```
1>------ Build started: Project: MyConsoleProject, Configuration: Debug
Any CPU ------
1>                                   MyConsoleProject                ->
C:\C#\MyConsoleProject\bin\Debug\MyConsoleProject.exe
======== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =========
```

Meddelandet ovan säger att koden inte innehåller några kompileringsfel. Har du syntaxfel i koden kommer du att få felmeddelanden i Output-fönstret. Åtgärda alltid endast det allra första kompileringsfelet och kompilera om med kommandot ovan, eftersom de andra kan innehålla följdfel. Ett möjligt kompileringsfel kan vara att du glömt att exkludera filen Program.cs från projektet, se sid 17.

För att exekvera koden, klicka i menyraden längst upp på menyn:

Debug → Start Without Debugging

Om allt har gått bra bör det se ut så här på din skärm:

Får du detta på skärmen har du lyckats med att kompilera och exekvera den kod du matade in på sid 18 och skapa en C# Console Application: Programmet **`First`** finns nu lagrad i filen First.cs och i projektet MyConsoleProject**`.csproj`**.

Grattis!

Vill du skapa nya konsolapplikationer behöver du inte göra om hela proceduren. Du behöver bara ladda projektet MyConsoleProject i Visual Studio, exkludera filen First.cs från det och infoga nya filer resp. skriva ny kod, spara och köra enligt instruktioner på sid 17-18. *Ett* projekt räcker för alla konsolapplikationer. Så det var värt mödan.

ETT projekt för alla konsolapplikationer

Det är jobbigt att behöva skapa ett separat projekt varje gång man vill testa ett litet program. Och vi kommer att hela tiden skriva och testa små program. Av dessa kommer många – speciellt i början – bestå av endast en fil där skapandet av ett nytt projekt varje gång kan uppfattas som overkill. Det finns följande möjlighet att slippa detta och ändå uppfylla Visual Studios krav på att utveckla program endast inom ett projekt:

För att underlätta arbetet och inte behöva skapa för *varje* program ett *nytt* projekt, kommer vi att skapa ett enda projekt dvs ta steg **1** (sid 15) bara en gång i början och i fortsättningen endast upprepa stegen **2a** (sid 17) och **3** (sid 19). Dvs vi kommer att skapa *ett* projekt i vilket vi sedan lägger en aktuell C#-källkodsfil, jobbar med den och avlägsnar den från projektet när vi är klara. Nästa gång öppnar vi samma projekt, lägger till en annan källkodsfil i det, jobbar och tar bort filen från projektet osv. För alla C# program används ett och samma projekt.

Så här kan man realisera detta förfarande:

I fortsättningen, när du vill testa ett annat C# program, laddar du det redan skapade projektet MyProject, markerar först den gamla källkodsfil som finns i projektet från tidigare, exkluderar den från projektet genom att högerklicka på filnamnet i Solution Explorer och välja:

Exclude From Project

Filen tas bort och är inte längre med i Visual Studio-projektet, men finns kvar på hårddisken i projektmappen.

Sedan fortsätter du så här: För att ladda och testa nästa program markerar du i Solution Explorer projektnamnet **MyProject**, högerklicker och väljer:

→ Add → New Item…

Här följer du instruktionerna i stegen **2a** (sid 17) och **3** (sid 19). Så kan du hela tiden använda *samma* projekt för att kompilera *alla* dina C# program, så länge de är av typ Console Application. På så sätt slipper vi att skapa ett separat projekt för varje C# program.

Detta förfarande rekommenderar vi dock endast så länge som vi arbetar med konsolapplikationer (kap. 3, avsn. 3.1 – 3.3 & kap. 4-11).

Organisera dina C#-kodsfiler

Det är upp till dig hur du organiserar dina filer. Men för att underlätta arbetet rekommenderas följande förfarande:

Du kan samla och spara alla dina C# program tillhörande kapitel 3 *Att komma igång med C#* genom att skapa en undermapp som heter 03 KomIgång i en valfri mapp, t.ex. i C:\C# och spara filen First.cs i mappen C:\C#\03 KomIgång. Detta kan göras från Visual Studios FILE-huvudmeny med:

→ FILE → Save First.cs As…

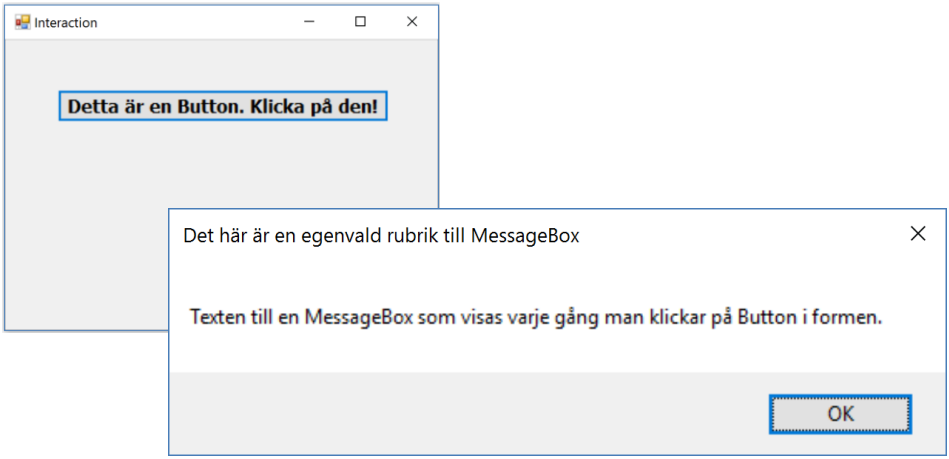
Anledningen till denna rekommendation är följande: Har du fått körresultatet på förra sidan efter flera försök där du rättat till kompileringsfel och kompilerat om och därmed ändrat C#-kodsfilen, har alla dina ändringar sparats i filen First.cs som tillhör projektet MyProject. Men eftersom vi enligt instruktioner nedan kommer att exkludera filen First.cs från projektet för att sedan kunna infoga och köra nästa program i samma projekt är det bra för säkerhets skull att ha alla sina testade program samlade i en egen mapp som ligger utanför projektmappen. På liknande sätt kan du spara dina efterföljande C#-kodsfiler i mappar du skapar under C:\C# och betecknar enligt bokens kapitelindelning.

Självklart fungerar bokens alla programexempel även i alla tidigare versioner av Visual Studio än 2019 vars installation beskrevs på sid 12.

1.4 C# Windows Forms Application

Ett grafiskt användargränssnitt, på eng. *Graphical User Interface (GUI)*, är en yta som kan användas för att kommunicera med programmet när det körs. Och detta i båda riktningar, dvs från användaren till programmet och tvärtom. Det är ett slags användarvänligt mellanskikt (*gräns*) mellan användaren och den icke-användarvänliga koden. För att kunna kommunicera måste vi väcka de grafiska komponenterna till liv och *interagera* med dem, när applikationen körs, vilket kräver att vi förser dem med egenskriven kod och/eller med komponenter som är förprogrammerade i Visual Studio. I regel ingår i sådana program mer grafik än kod. En konsekvens av denna nya form av program blir att körningen till skillnad från konsolapplikationer inte längre till 100% är förbestämd av utvecklarens kod utan kan även styras – åtminstone delvis – av användaren under programkörningen genom musklickningar och tangenttryckningar, s.k. *händelser*. Exekveringen startar i ett fönster med grafiska komponenter, som visas när programmet körs. Efter en händelse återgår kontrollen till operativsystemet, vilket dock inte betyder att körningen är avslutad, utan att programmet är redo att ta emot nästa händelse osv. – därför: *händelsestyrd programmering*.

I detta avsnitt vill vi bygga en *Windows Forms Application* som genererar nedanstående två fönster. Till vänster har vi det s.k. *formfönstret*, kort kallat *formen*, som i sin tur innehåller en knapp (Button). Först när man klickar på knappen (händelse) får man en meddeladeruta (MessageBox), avbildad till höger:



Controls

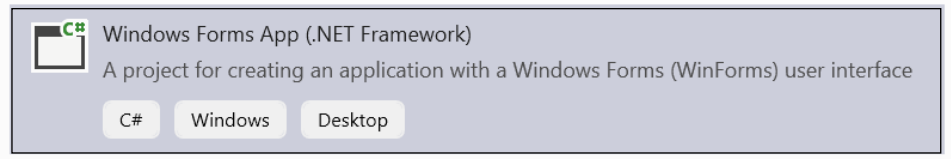
Knappen i formfönstret är en s.k. Control i Visual Studio. Den ingår i en stor grupp av återanvändbara grafiska komponenter i verktygslådan Toolbox. Ett exempel på en sådan Control är Button – en klickbar knapp som finns i Toolbox och som man med musen kan placera i formfönstret. För att få fram rutan till höger måste vi skriva kod ”bakom” Button och lägga den i applikationen. Vilken kod och framför hur

denna kombination av grafik och kod är organiserad i Visual Studio, ska vi nu gå igenom. Om *projekt* i Visual Studio läs på sid 14.

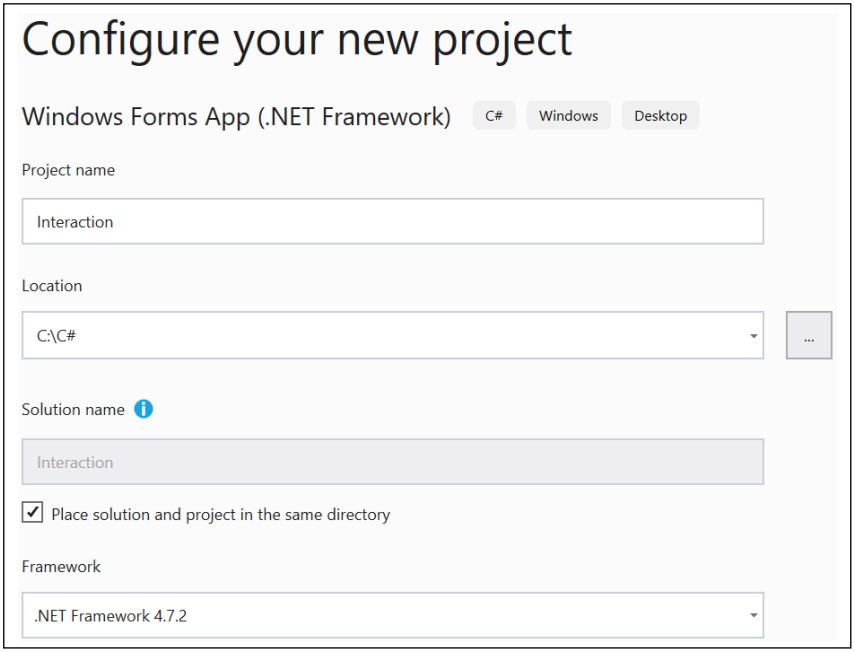
C# Windows Forms Application

Starta Visual Studio från Windows *Start*-meny: Start → Visual Studio 2019. Ett vitt fönster öppnas med rubriken Visual Studio 2019. I kolumnen till höger under rubriken Get started finns ett antal rutor. Klicka på rutan Create a new project .

En ny dialogruta dyker upp med rubriken Create a new project. Markera i den rutan med rubriken Windows Forms App (.NET Framework) som ser ut så här:



Klicka på knappen Next. Dialogrutan Configure your new project dyker upp:

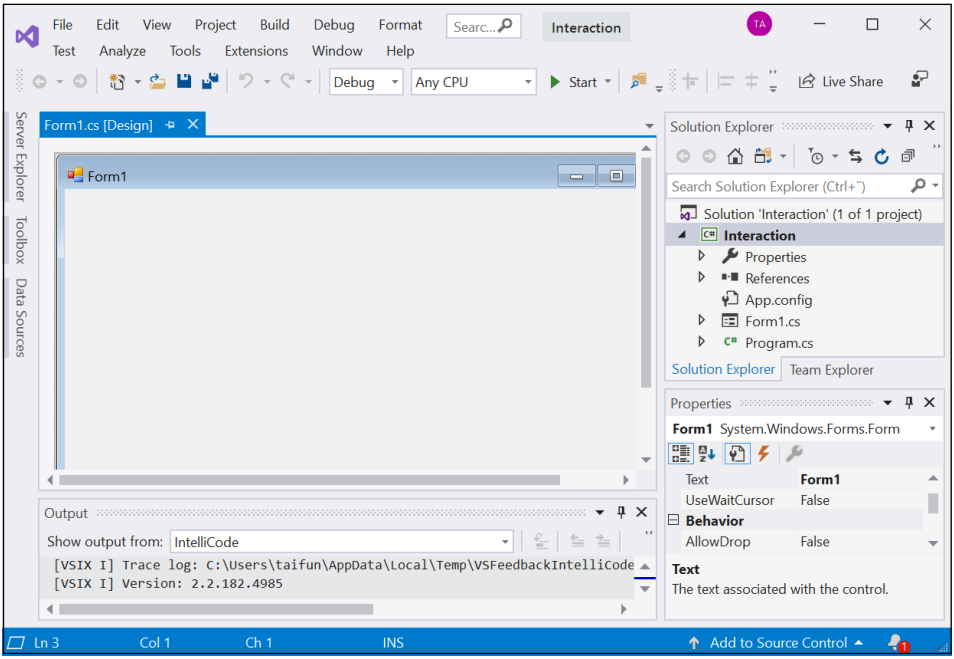


.

Fyll i den uppgifterna enligt ovan. Dvs i den övre delen av dialogrutan döper vi vårt projekt till Interaction. I textrutan Location anger vi den fullständiga sökvägen till den mapp vi vill placera vårt projekt i. Låt oss säga vi vill samla våra C# program i en mapp som vi kallar C# och placerar i enheten C:\. I så fall anger vi som Location C:\C#. I denna mapp kommer nu projektmappen Interaction placeras.

Visual Studio skapar automatiskt både den nya mappen och projektfilen. Bocka för den lilla rutan Place solution and project in the same directory. Klicka på Create.

Ett grafiskt gränssnitt kommer upp som liknar en webbsida bestående av en massa menyer, flikar, länkar och fönster som ser ut så här:



Huvudingrediensen i denna samling av komponenter är fliken Form1.cs [Design] som i sin tur visar ett fönster med rubriken Form1. Detta fönster är en s.k. *Windows Form*, kort kallad för *form* – ett grafiskt användargränssnitt som kommer att utgöra den visuella delen av vår grafiska applikation. Denna *form* – ibland även kallad formfönstret – är huvudfönstret (en slags Container) till alla grafiska applikationer som vi kommer att placera i den och som visas när programmet körs.

Markera formfönstret, gå med musen till Properties-fönstret i formfönstrets nedre högra hörn, markera egenskapen Text och ändra dess värde från Form1 till Interaction. Observera att formfönstrets rubrik nu ändrats till Interaction. Scrolla ner Properties-fönstret till egenskapen Size och sätt dess värde till 930; 660. Därmed har vi gett vårt formfönster en ny rubrik och en ny storlek.

Gå till menyraden längst upp och välj menyn: View → Toolbox

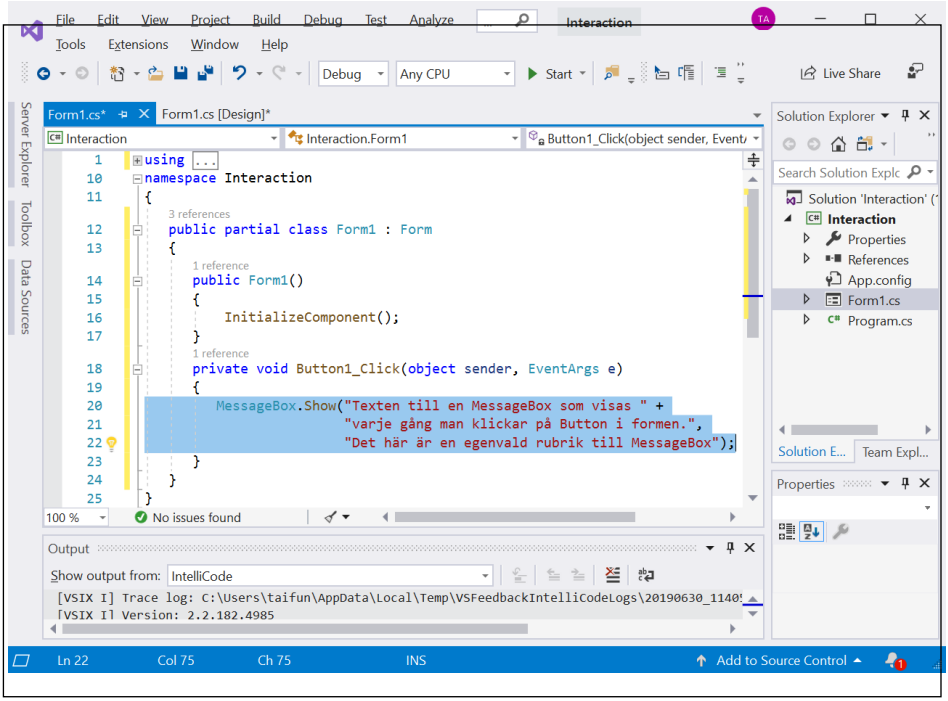
Expandera Common Controls och dubbelklicka på kontrollen Button, så att den hamnar i formfönstret. När du flyttar markören till formen stängs Toolbox-fönstret. Markera den nya kontrollen button1 på din form för att få fram dess egenskaper i Properties-fönstret.

Egenskaperna i Properties-fönstret är by default grupperade i kategorier (Categorized). Ändra detta genom att i Properties-fönstrets lilla menyrad strax under button1 klicka på ikonen (Alphabetical) för att lättare kunna hitta de egenskaper angivna i tabellen nedan. Ändra button1-egenskapernas värden enligt följande:

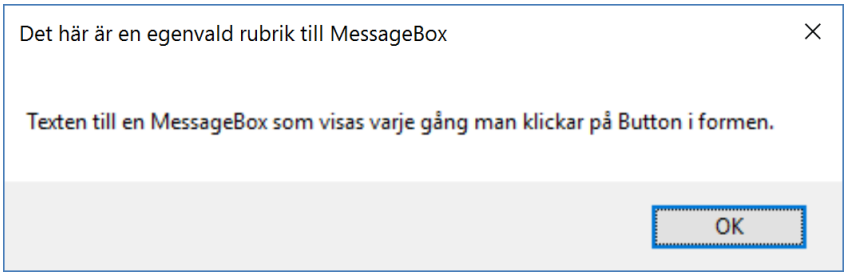
button1:

Egenskap	Värde
AutoSize	True
Font	Tahoma; 12pt; style=Bold
Location	110; 100
Text	Detta är en Button. Klicka på den!

Markera knappen med texten Detta är en Button. Klicka på den! och dubbelklicka på den. En ny flik Form1.cs uppstår till vänster om den gamla fliken Form1.cs [Design]. Den nya fliken visar kod som lagras i filen Form1.cs. Impandera den första raden som inleds med **`using`**. Skriv på det stället där markören står och blinkar, de tre rader kod som är markerade på denna bild (raderna 20-22):



Kompilera med Build → Build Solution och kör med Debug → Start Without Debugging applikationen Interaction. Klicka på knappen för att få fram detta:



Nedan följer den fullständiga koden i filen Form1.cs samt kodens förklaring:

```
// Form1.cs
using System;
using System.Windows.Forms;
namespace Interaction                    // Namnutrymme
{
   public partial class Form1 : Form     // Form1 ärver Form
   {
      public Form1()                     // Klassens konstruktor
      {
         InitializeComponent();
      }

      private void button1_Click(object sender, EventArgs e)
      {
         MessageBox.Show("Texten till en MessageBox som "   +
           "visas varje gång man klickar på Button i formen.",
           "Det här är en egenvald rubrik till MessageBox")  ;
      }
   }
}
```

I C# är **`namespace`** ett reserverat ord som skapar ett namnutrymme, en slags behållare för klasser. C#:s programbibliotek är organiserat i sådana namnutrymmen som innehåller fördefinierade klasser. Dessa placeras i namnutrymmen som får samma namn som projektet. T.ex. kan man komma åt klassen **`Form1`** med **`Interaction.Form1`** osv. De **`using`**-direktiven i början inkluderar två namnutrymmen ur C#:s programbibliotek som behövs för att kompilera denna enkla grafiska applikation. Ursprungligen genererar Visual Studio några onödiga **`using`**-direktiv till som vi tagit bort.

Klasshuvudet **`public partial class Form1 : Form`** säger för det första att koden är en *del* av klassdeklarationen (**`partial`**). För det andra säger det att klassen **`Form1`** som vi skapar, ärver bibliotheksklassen **`Form`**. I C# är **`:`** koden för arv*. Klassen **`Form`** i sin tur är deklarerad i namnutrymmet **`System.Windows.Forms`**. Där

* Läs om *metoder* på sid 61 och om *arv* och *konstruktorn* på sid 63.

finns en hel del fördefinierad kod som behövs för att skapa formfönstret. Alla klasser som skapar formfönstret måste ärva denna fördefinierade kod. Den del av klassen **Form1** som deklareras här, innehåller endast två metoder. Den första är klassens konstruktor **Form1()**. Den andra metod i vilken vi lade tre rader egen kod, heter **button1_Click()**. Denna kod gör att MessageBoxen visas vid musklickning när man kör programmet. Medan konstruktorn **Form1()** är en automatisk metod för att initiera klassen **Form1**:s egenskaper, är **button1_Click()** en helt ny typ av metod som kallas för *händelsemetod*. Den förekommer inte i konsolapplikationer utan är ett verktyg för händelsestyrd programmering och därför typisk för interaktiva grafiska applikationer.

Händelsemetoder

Vanliga metoder definieras först och anropas sedan. Både definitionen och anropet sker med kod. En *händelsemetod* (eng.: *event handler*) definieras också precis som en vanlig metod, men anropas inte explicit med en vanlig anropskod utan genom en s.k. *händelse*. En händelse är en aktion som utförs antingen av användaren eller av ett program, vare sig en applikation eller datorns operativsytem. Exempel på händelser är musklickning, musdragning eller tangenttryckning. Men även en kod kan utlösa en händelse. När händelsen inträffar, anropas metoden som är associerad med händelsen. Metoden **button1_Click()** är associerad med musklickning på button1, en kontroll av typ Button. Så snart vi skapar en sådan kontroll i formen, t.ex. button1 (sid 24), genereras kod: Huvudet till metoden **button1_Click()** i klassen **Form1** (filen Form1.cs). Med dubbelklick på den nya kontrollen (i designläge) får vi fram denna kod i editfönstret och kan skriva kroppen till metoden. Vi är fria att skriva där vilken kod som helst, för att få den exekverad när man i körläge klickar på knappen button1. Eftersom vi vill få ut ett meddelande i ett fönster, skriver vi ett anrop av metoden **MessageBox.Show()** som vi stiftade bekantskap med tidigare. Händelsemetoden **button1_Click()** har två parametrar som vi dock inte använder i kroppen i just denna applikation. Ändå måste vi ha dem med i metodens huvud, för huvudet är fördefinierat i superklassen **Form**.

Metoden **MessageBox.Show()**

Till skillnad från **button1_Click()** är metoden **Show()** ingen händelsemetod, utan en vanlig metod, fördefinierad i klassen **MessageBox**. Därför anropas den med kod, inte med en händelse (musklickning). Den anropande koden står i händelsemetoden **button1_Click()**. Musklick på knappen med texten Detta är en Button. Klicka på den! (i körläge) anropar händelsemetoden och den i sin tur metoden **Show()**. I den version som används här har metoden **MessageBox.Show()** två parametrar: Den första står för själva meddelandet som ska visas i den lilla rutan, den andra för rubriken som ska stå på rutans ram. Att vi i koden med **+** konkatenerar två strängar på den 1:a parameterplatsen, beror på att meddelandet vi vill skriva ut, inte ryms på en rad i editfönstret resp. på sidan i boken. I koden är det som vanligt kommat som skiljer åt metodens två parametrar.

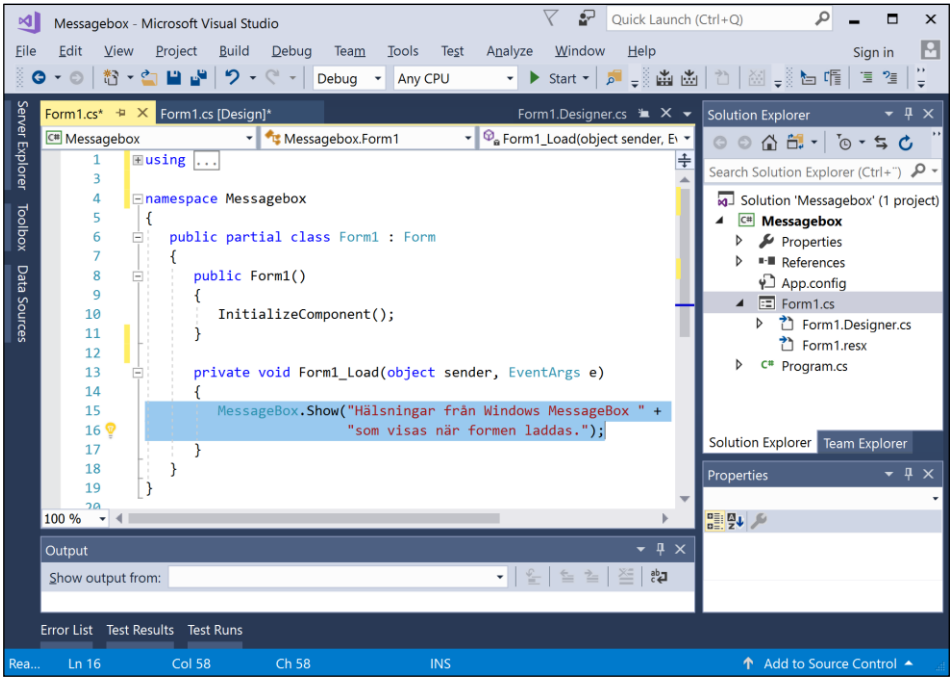
1.5 Utskrift till en grafisk miljö

Alla våra program hittills har varit *C# Console Applications* (sid 15). De skriver ut sina körresultat till konsolen. Det är det svarta fönstret som i Windows kallas för *Kommandotolken*. Konsolen är en ren textmiljö som är uppbyggd av små rutor. I varje ruta hamnar *ett* tecken, antingen bokstav, siffra eller specialtecken. Till skillnad från en textmiljö är en grafisk miljö uppbyggd av små *pixlar*, där en pixel (*picture element*) är bildskärmens minsta ljuspunkt – mycket mindre än en textruta. I en grafisk miljö ”ritas” t.o.m. bokstäver och siffror med pixlar.

I avsnitt *1.4 C# Windows Forms Application* hade vi behandlat ett exempel på interaktion (sid 22). Här ska vi med enkla medel skriva ut till en grafisk miljö, en meddelanderuta som heter MessageBox. Till skillnad från konsolen är MessageBox en grafisk miljö, dock utan möjligheten till interaktion. Gör så här:

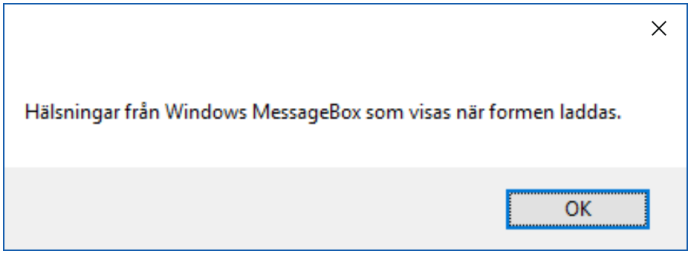
Skapa enligt instruktionerna på sid 23 en *Windows Forms Application* och döp projektet till **`Messagebox`**. Beakta att du stavar rätt, speciellt det lilla **`b`**. Vi återkommer till det på nästa sida. Ett nytt fönster liknande på sid 24 dyker upp med en ny flik: Form1.cs [Design] som i sin tur visar det lilla formfönstret.

Markera formfönstret och dubbelklicka på det. En ny flik Form1.cs innnehållande kod uppstår som lagras i filen Form1.cs. Skriv på det stället där markören står och blinkar, de två rader kod som är markerade på denna bild:



Ta koden från nedan om den markerade koden på bilden ovan inte är tydlig nog.

Kompilera projektet **Messagebox**. Exekveringen ger meddelanderutan till höger. Ett klick på OK stänger rutan, och programflödet går tillbaka till det tomma formfönstret. Den fullständiga koden i filen *Form1.cs* som utgör koddelen av programmet **Messagebox** ser ut så här:




```
// Form1.cs
using System;
using System.Windows.Forms;

namespace Messagebox
{
  public partial class Form1 : Form
  {
    public Form1()
    {
      InitializeComponent();
    }

    private void Form1_Load(object sender, EventArgs e)
    {
      MessageBox.Show("Hälsningar från Windows MessageBox " +
                      "som visas när formen laddas.");
    }
  }
}
```

Anledningen till att vi döpte vårt projekt till **Messagebox** och stavade det med ett litet **b** är att **MessageBox** med stort **B** är en klass i biblioteket **System.Windows.Forms** som är fördefinierad där och som vi använder för att kunna anropa metoden **Show()**. Klassen **MessageBox** har därmed samma status som ett reserverat ord som vi inte får använda som namn för våra egna variabler, klasser, projekt osv., se namngivningsreglerna på sid 59. Hade vi döpt projektet till **MessageBox** med stort **B** hade vi fått kompileringsfel.

För alla andra frågor angående koden ovan och projektets egenskaper hänvisas till förklaringarna i nästa avsnitt som behandlar en *Windows Forms Application* med interaktion (sid 26). Där tas bl.a. upp metoden **MessageBox.Show()** samt hur en ny typ av metoder kallade *händelsemetoder* fungerar. Metoden **Form1_Load()** i koden ovan är en sådan hädelsemetod som automatiskt anropas när formen laddas vid exekvering, vilket i sin tur gör att **MessageBox** visar sin text.

Övningar till kapitel 1

1.1 Modifiera programmet **MessageBox** (sid 29) så att meddelanderutan får rubriken ”Övning 1.1”. Hur man skriver ut en MessageBox med en egenvald rubrik har vi lärt oss i programmet **Interaction** (sid 26).

1.2 Skapa en Windows Forms Application (sid 23) och döp den till **WindowsForm**. Utveckla den med förprogrammerade grafiska komponenter i Visual Studio (*Controls*, sid 22) till ett grafiskt program utan att skriva någon kod. När man exekverar programmet ska följande bild genereras:

För att få tag i den färgglada bilden ovan med texten ***Välkommen till Windows programmering!*** ladda ner bildfilen så här:

a) Gå till webbsidan **www.taifun.se**, klicka på bokomslaget *Programmering 2 med C#* . Scrolla ner och klicka på länken **Välkomst.gif**. Extrahera **gif**-filen från zip-filen.

Ladda ner filen Välkomst.gif

b) Lägg gif-bilden i en *Control* som heter PictureBox som du kan hämta från Toolbox. Sätt värdet StrechImage till PictureBoxens egenskap SizeMode, för att få in gif-bilden i PictureBoxen.

Förse din applikation med en Label med den text du ser över bilden.

1.3 Modifiera programmet **WindowsForm** från övn 1.2 så här: Rita i något av dina favoritritprogram en bild efter egen smak och spara den i en bild-

format. Alternativt kan du med din mobilkamera ta ditt porträtt och överföra det till datorn i gif- eller jpg-format. Skapa sedan en Windowsapplikation i Visual Studio, infoga din bild i den och förse den med en förklarande text med en Label under bilden. Se till att layouten blir så snygg som möjligt.

1.4 Modifiera din lösning från övn 1.3 genom att ändra formens storlek och bakgrundsfärg. Dessutom ska Labeln få en annan bakgrundsfärg än formen.

Kapitel 2

Programmeringsspråket C#

2.1 Integrated Development Environment (IDE)

Visserligen är *”Programmering är i allra högsta grad ett praktiskt ämne.”* (sid 6). Dvs du kommer aldrig lära dig programmering genom att endast läsa böcker. Men lika sant är att du inte kan fortsätta programmera utan att känna till de mest grundläggande *begreppen*. Utan denna kunskap blir det inte särskilt roligt att koda. Det roliga är just att *förstå* vad man gör när man skriver och testar sina program.

I detta kapitel tar vi upp några begrepp om både programmeringsmiljön och -språket C#. Ett av dem är kompilering. En *kompilator* är en programvara som översätter *källkod* (som vi förstår) till *maskinkod* (som endast datorn förstår). För att köra C# kod måste en C# kompilator vara installerad på datorn. Har du redan tillgång till en sådan programvara kan du gå vidare till nästa kap 3, mata in vårt första C# program **`First`** (sid 42) och fortsätta. Annars borde du installera och konfigurera C# kompilatorn, som ingår i Visual Studio, enligt anvisningarna i förra kap 1. Mer om detta kommer vi att ta upp i avsn. *2.3 Kompilering och exekvering* (sid 37).

Visual Studio – en IDE

För att underlätta utvecklingsarbetet har Microsoft integrerat C#-kompilatorn i sin programvara Microsoft Visual Studio, en s.k. *Integrated Development Environment (IDE)*, en integrerad programutvecklingsmiljö. En IDE är ett grafiskt gränssnitt som inkluderar en editor, en kompilator och andra verktyg för programutveckling i en samlad miljö. Fördelen med en IDE är att man slipper byta miljö mellan editering och kom-pilering, när man utvecklar kod. Man behöver inte bry sig om sökvägar i systemet. Dessutom har IDEn stödverktyg för språket osv. Vi kommer att använda Visual Studio för att skriva, kompilera och exekvera alla våra programexempel i C#. Programvaran som även har kompilatorer för ytterligare språk som Python, Visual Basic, C++, F# och andra verktyg, är väldigt stor och komplex – ett verktyg för professionella utvecklare. Den är inte gjord för nybörjare. För att kunna koncentrera oss på själva språket C#, som är vårt egentliga mål, kommer vi att endast ta upp den delen av miljön som just behövs för att kunna testa våra C# program.

Vad är .NET?

För att integrera de fyra programmeringsspråken C#, Python, Visual Basic, C++ och F# i en enda miljö, nämligen i IDEn Visual Studio, så att deras koder enkelt kan bytas ut mot varandra och bli portabla på olika plattformar, har Microsoft utvecklat ett speciellt språk som kallas *Common Intermediate Language (CIL)*. Vid kompileringen översätts alla koder i Visual Studio först till CIL. CIL-koden tolkas sedan till maskinkod – kallad *bytecode* eller *object code* – när programmet exekveras. Kombinationen av datortypen (hårdvara) och operativsystemet (mjukvara) kallas *plattform*. Medan CIL är plattforms*oberoende* och kan fortfarande läsas och förstås av människan, är *object code* plattforms*beroende* och kan endast tolkas av datorns CPU. För att realisera CIL-konceptet behöver Windows ett tillägg till operativsystemet som heter Microsoft .NET Framework, även kallat .NET-plattform.

2.2 Vad är C# ?

C# är ett modernt universellt programmeringsspråk som används för att skriva program för datorer. Modernt därför att man – lite förenklat – kan säga att Pascal och C var 70-talets, C++ 80-talets, Java och Python 90-talets och C# 2000-talets programmeringsspråk. Universellt därför att C# har obegränsade tillämpningsmöjligheter, från spel till webbapplikationer, från text- eller databasrelaterade program till interaktiva grafiska gränssnitt. Det finns inga applikationer på datorn som inte skulle kunna skrivas i C#. Den allra första versionen av C# släpptes år 2000.

C# har sina rötter i programmeringsspråken C, C++ och Java. Man har byggt på det gamla, beprövade och välkända, tagit över allt som var bra i de äldre språken och samtidigt tagit bort allt som var lite krångligt, ibland t.o.m. instabilt. Samtidigt har man försett det hela med en ny överordnad struktur vars objektorientering redan var renodlad hos C++. På så sätt behövde man inte ta hänsyn till rötterna från C. Men det mest intressanta i C# är att det är utvecklat för en plattform där man bjuder på en programutvecklingsmodell som tillåter olika språk att kommunicera med varandra – den s.k. ***.NET***-plattformen (sid 34). Tillägget # till C har tagits från musikvärlden där tonen C# (eng. *C sharp*, sv. *Ciss*) är en halv ton högre än C, vilket syftar på att man ”höjt” språket C till C# (uttalat ”si sharp”).

C# är objektorienterat

C#:s viktigaste programmeringstekniska egenskap är att det från början är konstruerat som ett renodlat objektorienterat språk. Allt i C# är objekt. Ett C# program, även det minsta, är inget annat än en samling av objekt som ”pratar” med varandra när programmet körs. Man kan även säga att de skickar meddelanden *(messages)* till varandra. I själva verket anropar de varandras metoder. Hela programbiblioteket är skrivet på objektorienterat sätt. Så det består i stort sett av tusentals klasser som är organiserade i ett antal namnutrymmen *(namespaces)*. Vill man använda dem inkluderar man dem i sitt program och skapar objekt av dem. Självklart kan man även skriva egna klasser. Minst en sådan *måste* man t.o.m. skriva för att därifrån starta programkörningen. Man lagrar i regel varje klass i en separat fil för att kunna använda den även i andra program. Så ett C# program består ofta av ett antal filer. Varje fil innehåller en modul som återanvänds i olika sammanhang, se Appendix *Vad är objektorienterad programmering*.

Reserverade ord

Dagens version av C# som denna bok bygger på, är i skrivande stund Visual C# som ingår i Visual Studio 2019. Denna programvara behövs för att kunna utveckla och testa C#-kod. Som alla programmeringsspråk är även C# definierat av ett antal nyckelord, även kallade *reserverade ord*, eftersom de inte får användas som namn för programmets andra komponenter. De är reserverade av och för själva språket och bildar språkets ordförråd. Denna kärna i C# består av 78 reserverade nyckelord och är samlade i följande tabell som vi kommer ofta att referera till:

Reserverade ord i C# *(Sök kolumnvis!)*				
abstract	**do**	**in**	**protected**	**true**
as	**double**	**int**	**public**	**try**
base	**else**	**interface**	**readonly**	**typeof**
bool	**enum**	**internal**	**ref**	**uint**
break	**event**	**is**	**return**	**ulong**
byte	**explicit**	**lock**	**sbyte**	**unchecked**
case	**extern**	**long**	**sealed**	**unsafe**
catch	**false**	**namespace**	**short**	**ushort**
char	**finally**	**new**	**sizeof**	**using**
checked	**fixed**	**null**	**stackalloc**	**using static**
class	**float**	**object**	**static**	**virtual**
const	**for**	**operator**	**string**	**void**
continue	**foreach**	**out**	**struct**	**volatile**
decimal	**goto**	**override**	**switch**	
default	**if**	**params**	**this**	
delegate	**implicit**	**private**	**throw**	

Observera följande allmän regel som gäller för all C#-kod:

C# är case sensitive (skiftlägeskänslig).

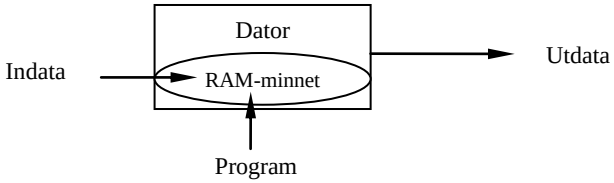
Dvs C# skiljer på gemener och versaler. Om vi t.ex. skriver det reserverade ordet **`new`** som **`New`**, kommer C#-kompilatorn generera ett felmeddelande. Den skiljer på **`new`** och **`New`** men känner bara till **`new`**. Alla reserverade ord är giltiga endast med små bokstäver. Denna regel får beaktas när man är van vid Windows som inte är case sensitive.

C# har ett stort programbibliotek

Om man nu tittar på ett C# program, kommer man att upptäcka flera ord som inte finns i ovanstående tabell. T.ex. **`WriteLine()`** är ett sådant i vårt första program **`First`** (sid 42). Detta beror på att C# dessutom har ett s.k. *bibliotek* av fördefinierade program (klasser) som man använder i sina egna program för att utföra rutinmässiga uppgifter som t.ex. in- och utmatning. C#:s kärna innehåller inga instruktioner för t.ex. att skriva ut data till bildskärmen eller läsa in data från tangentbordet. Man måste använda biblioteksprogram för det. **`WriteLine()`** är en metod för utskrift till konsolen som är definierad i klassen **`Console`** som i sin tur finns fördefinierad i biblioteket **`System`**. Men även andra instruktioner för vissa ofta använda standardrutiner är redan kodade i biblioteksprogram: T.ex. att rita geometriska figurer, att bestämma längden på en text eller generera slumptal osv. är rutiner som man inte ska behöva skriva kod för när man behöver dem. C#:s programbibliotek ligger som ett skal kring den inre kärnan av reserverade ord. Om vi vill använda dem i våra egna program, måste vi anropa dem med deras namn. Vi måste därför tala om för kompilatorn vilka fördefinierade program vi tänker att använda och i vilka bibliotek de ligger, t.ex. genom att inleda med **`using System;`** .

2.3 Kompilering och exekvering

Innan vi ger oss i kast med själva C#-kodningen ska vi på ett mera detaljerat sätt gå igenom hur programkoden hamnar i datorn och hur den körs där. I de första datorerna var den enda möjligheten att skapa ett ”program”, att lägga in instruktionerna i hårdvaran – ett väldigt osmidigt förfarande. Först 1944 lyckades *John von Neumann* att konstruera en dator som kunde lagra både indata och programinstruktionerna i minnet:



Arbetssättet används än idag: Programmet laddas från hårddisken till datorns primärminne, även kallat *RAM*-minne (*Random Access Memory*) när ett program körs. På hårddisken finns bara filer och mappar. Vi måste alltså se till att vårt program hamnar i en fil som vi placerar i en mapp på hårddisken. En sådan fil heter *källkodsfil* eftersom den innehåller *källkoden* till det körbara programmet. Följande steg måste tas för att få in källkoden i och köra programmet på datorn:

- Editering
- Kompilering
- Exekvering

Editering

Eftersom programmet ska översättas av en kompilator till maskinkod, måste vi vid skapandet av källkodsfilen se till att den endast innehåller tecken resp. teckenkombinationer som kompilatorn förstår. Om det ska bli ett C# program får källkodsfilen endast innehålla de reserverade ord som definierar språket (sid 36) samt biblioteksnamn men inget annat. Därför måste vi skriva programkoden i en texteditor, ordbehandlare eller annat skrivverktyg som sparar källkodsfilen som *oformaterad textfil*. I praktiken måste man ha tillgång till Visual Studio som är en integrerad programutvecklingsmiljö (IDE, se sid 34) för att utveckla och testa C# kod. Med integrerad menar man att flera verktyg är samlade i Visual Studio, bl.a. en texteditor. Fördelen med en IDE är att man inte behöver bry sig om sökvägen till C#-installationen i systemet, att den har olika stödverktyg för språket och att man slipper byta miljö mellan editering och kompilering. Längre fram i boken ges anvisningar om hur man laddar ner och installerar Visual Studio (sid 12) samt hur man konfigurerar och använder denna miljö för att testa sina C# program (sid 13).

Regeln för filändelsen

Har du skrivit din programkod t.ex. i Notepad eller en annan editor och sparat filen som **`*.txt`** eller med en annan ändelse, kommer du att få kompileringsfel även om din kod är felfri. Boven i dramat är filändelsen: kompilatorn accepterar den inte. Kompilatorn måste nämligen kunna identifiera de filer som innehåller C#-kod via filändelsen. Därför kräver C#-kompilatorn filändelsen **`cs`** på källkodsfilen. Antingen måste du spara din källkodsfil med korrekt filändelse eller ge den korrekt ändelse i efterhand. Denna enhetliga regel för filändelsen gäller självklart även om du arbetar i Visual Studio där den väljs automatiskt.

Kompilering av C#-källkod

Kompilering innebär översättning av källkod till maskinkod. Har du skrivit din programkod i en texteditor och sparat den som ren textfil med filändelsen **`cs`**, måste du kompilera din källkod innan du kan köra programmet. Anledningen är att datorns processor inte förstår källkod, endast maskinkod. För att kunna kompilera måste C#-kompilatorn vara installerad på din dator. C#-kompilatorn är själv ett program som lagras i filen **`csc.exe`** där det andra **`c`** står för **`compile`** och **`exe`** visar att det är en exekverbar dvs körbar fil. Denna fil ingår i Visual C# 2019. Kompilering innebär att köra programmet **`csc.exe`** som översätter C#-kod till .NET-plattformens ”mellan”språk *Common Intermediate Language (CIL)* (sid 34):

När du har installerat Visual Studio (sid 12) kan du hitta filen **`csc.exe`** i mappen **`C:\Windows\Microsoft.NET\Framework\v2.0.50727`**. Du kan, om du vill, t.o.m. köra från Kommandotolken kommandot **`csc First.cs`**, dvs kompilera innehållet i filen **`First.cs`**, om du ställer dig i mappen ovan och om även filen **`First.cs`** finns där. Då kompileras vårt första C# program (sid 42) som lagras i filen **`First.cs`**. Kompilatorn skapar då den exekverbara filen **`First.exe`**. Samma kompilator ingår i Visual Studio och körs när man kör Build Solution (sid 19).

Hur C#-kompilatorn åstadkommer denna översättning vet vi inte och behöver vi inte heller veta när vi programmerar. Det enda vi behöver veta är hur man *använder* kompilatorn och hur man skriver C#-kod så att kompilatorn accepterar den. C#-kompilatorn producerar samma CIL-kod från en källkod oavsett datortyp. Om vi alltså preciserar C#:s plattformsoberoende måste vi säga: Kompileringen av C#-källkod är plattformsoberoende. Däremot är CIL fortfarande i textform. Koden översätts i en ytterligare process till en slags maskinkod som kallas *Byte Code*.

Exekvering av Byte Code

Det som ”förstår” CIL är ett program som kallas *Virtual Machine (VM)*. Detta program ingår i Visual Studio. Vad det gör är också en slags översättning, fast man kallar det snarare för en *interpretering* dvs en tolkning. Skillnaden i begreppen är att en översättning (kompilering) lämnar ifrån sig ett nytt dokument medan en si-

multan tolkning inte gör det. C#-interpretatorn tolkar Byte Code och omvandlar denna speciella typ av maskininstruktioner till ettor och nollor som datorns processor förstår och kan utföra. Detta kallas *exekvering*. Till skillnad från C#-kompilatorn finns för varje plattform en speciell C#-interpretator som exekverar Byte Code. Detta sista extrasteg som måste tas för att köra ett C# program är alltså beroende av datortyp och operativsystem. För att precisera C#:s plattformsoberoende måste man lägga till: C#-interpretatorn (VM) är plattformsberoende.

C#-interpretatorn måste alltså vara kompatibel med datortypen och ingå i operativsystemet för att kunna köras. Det räcker att den kompilerade filen lagras. Källkoden behöver inte lagras alls.

Fel

kan uppstå i alla ovan beskrivna steg. Därför skiljer vi mellan olika typer av fel:

- **Kompileringsfel** som kan uppstå pga att vi har brutit mot språkets regler. Dvs i kod som vi själva skrivit finns ”ortografiska” fel, något felstavat nyckelord eller ett utelämnat semikolon osv. Det kan även handla om ”grammatiska” fel, även kallade *syntaxfel* som t.ex. en felanvänd kod eller fel struktur i koden. Kompileringsfel innebär tvärstopp dvs man kan inte gå vidare till nästa steg utan måste först hitta och korrigera felet samt kompilera om.

- **Exekveringsfel** uppstår endast om processorn inte kan utföra dina instruktioner. Ett typiskt exempel på exekveringsfel i program som involverar beräkningar är division med 0. Ett annat exempel är användningen av minnesutrymme som är redan upptaget av ett annat program i datorn. Ett tredje exempel är skadade eller obefintliga filer som det hänvisas till i den egna programkoden.

Vid felsökning är det avgörande att man först identifierar *typen* av fel innan man vidtar någon åtgärd.

Observera även att man som nybörjare ofta får *inte ett* – utan en hel samling av felmeddelanden. Bli inte desperat! Det är helt normalt. Glöm alla felmeddelanden utom det allra *första*. De kan nämligen vara *följdfel* orsakade av första felet. Åtgärda endast det första och kompilera om. Om några fel är kvar, upprepa förfarandet. Du kommer att se: efter två tre gångar har du blivit av med alla fel.

Kapitel 3

Att komma igång

med C#

3.1 Vårt första C# program

Efter de inledande kapitlen om programmeringsmiljön och -språket är det dags för praktik – att skriva och testa vårt första C# program. För att göra det behöver du ha installerat Visual Studio enligt instruktionerna på sid 12. Så, låt oss sätta igång!

```
// First.cs                          Filnamnet
// Skriver ut text till konsolen (svarta fönstret)
// Metoden Main() anropas automatiskt när programmet körs
// Den i sin tur anropar metoden WriteLine() i klassen
Console
// Klassen Console finns förprogrammerad i biblioteket System

using System;                    // Krävs för klassen Console

class First                      // Klassnamnet
{
   static void Main()            // Metoden Main()
   {
      Console.WriteLine("\n\tMitt första C# program!\n");
   }
}
```

En körning visar följande utskrift i konsolen (Windows Kommandotolk-fönstret):

```
        Mitt första C# program!
```

I själva verket ser utskriften ut så som är avbildad på sid 19. Dvs konsolen visar dessutom meddelandet **`Press any key to continue . . .`** som dock inte producerats av C#-koden utan av Kommandotolken, för att hålla kvar fönstret. Därför visar vi i fortsättningen inte detta meddelande när vi avbildar programmens körresultat. Vi kommer i fortsättningen att referera till programmet ovan med klassnamnet **`First`**. Liknande konvention använder vi i bokens alla program: Ett program **`Abc`** skrivs i klassen **`Abc`** och lagras i filen **`Abc.cs`**.

Kommentar i C#

De första raderna börjar med två snedstreck (eng. *slash*) **`//`**. Detta teckenpar betyder *kommentar*, närmare bestämt *radkommentar*. En radkommentars giltighet börjar med **`//`** och sträcker sig till slutet av raden. **`//`** kan stå i början av en rad, men också någonstans mitt på raden. En *blockkommentar* kan gå över flera rader och ska inledas med **`/*`** och avslutas med **`*/`**. Alla kommentarer kommer att ignoreras av kompilatorn. De är endast till för att förklara koden. I den första kommentarraden står alltid i vilken fil koden lagras, här **`First.cs`**. I den andra raden brukar stå vad programmet gör. Sedan följer kommentar om de olika programmeringstekniska koncept som behandlas i programmet. I följande ska vi reda ut några begrepp:

Sats

Termen *instruktion* kommer i fortsättningen att ersättas av programmeringstermen *sats* (eng. *statement*). Innehållet är samma sak: ett kommando till datorn att utföra något. En sats i C# måste avslutas med semikolon. Det lilla tecknet **`;`** är ett av de vanligast förekommande tecknen i C#-kod och samtidigt det oftast glömda tecknet vid kodningen. Semikolonet är en *obligatorisk del* av en C#-sats, det allra sista tecknet i satsen. Semikolonet är C#-språkets *satsavslutningstecken* vars utelämnande leder till kompileringsfel. I **`First`** avslutas alla satser – det finns endast två – med semikolon, även satsen strax före klammern **`}`**. Klammern ersätter inte semikolonet utan avslutar **`Main()`**-metodens kropp.

`using`-*direktivet*

Programmet **`First`**:s första sats efter kommentarerna är ett s.k. **`using`**-direktiv:

```
using System;
```

`System` är ett s.k. *namnutrymme* (eng. *namespace*), en slags behållare för klasser. C#:s programbibliotek är organiserat i sådana *namespaces* som innehåller fördefinierade klasser, paketerade i namnutrymmen. Här ges kompilatorn direktivet att ladda namnutrymmet **`System`** till vårt program så att vi kan använda klasser som finns där och anropa metoder som är definierade i dem, här klassen **`Console`** och metoden **`WriteLine()`**. Alternativt skulle man kunna slippa **`using`**-direktivet och istället anropa metoden **`WriteLine()`** med **`System.Console.WriteLine(...)`** vilket man dock i regel inte gör eftersom det gör koden onödigt tung. Speciellt när man anropar metoder flera gånger ur samma klass eller använder flera klasser i samma namnutrymme, är det bättre att skriva **`using`**-direktivet en gång för alla i början. Men det gäller att placera det rätt: **`using`**-direktiv måste alltid skrivas *utanför* klassen, närmare bestämt *före* klassen eftersom informationen behövs *i* klassen. Att placera ett **`using`**-direktiv i en klass ger kompileringsfel. Man kan inte inkludera *flera* namnutrymmen med *ett* **`using`**-direktiv. För varje namnutrymme krävs ett separat **`using`**-direktiv. Dessa särregler gör att **`using`**-satsen kallas för *direktiv*.

Vad är ett C# program?

> Ett C# program är en samling av klasser, av vilka en och endast en måste innehålla metoden **`Main()`**.
>
> När programmet körs startar exekveringen i **`Main()`**.

Alla C# program måste innehålla metoden **`Main()`** för att kunna exekveras, annars har exekveringen ingen startpunkt. För att exekveringen ska automatiskt kunna börja i **`Main()`** måste namnet vara känt för C#-interpretatorn och därmed obligatoriskt. Det är nämligen C#-interpretatorn (*Virtual Machine*, sid 38), som automatiskt anropar metoden **`Main()`** när vi exekverar programmet. **`First`** är det enklast

tänkbara C# program därför att det består av en klass som innehåller **`Main()`**. Men **`Main()`** kan aldrig skrivas fristående dvs utanför en klass utan måste alltid inbäddas *i* en klass. Det beror på att klasser är C# programmens primära byggstenar, medan metoder inkl. **`Main()`** är *delar* av dessa klasser. I andra programmeringsspråk som C++ finns även *funktioner* som är fristående. En *metod* gör samma sak som en funktion, men är placerad i en klass. Därför finns det i C# inga funktioner utan endast metoder: I vårt första C# program är **`Main()`** en metod i klassen **`First`** vars huvud är föreskriven för att kunna kännas igen av C#, men vars innehåll vi kan bestämma själva. Mer om **`Main()`** följer längre fram.

Klassen `First`

Efter kommentarerna och **`using`**-direktivet följer i programmet **`First`** (sid 42):

```
class First
```

som är rubriken – man säger också *huvudet* – till en klass. En *klass* i C# är en kodmodul som man bygger programmet med, jämförbar med en Legobit eller en tegelsten som man bygger ett hus med. Klasser är C# programmens minsta beståndsdelar. Den allmänna strukturen hos en klass i C# ser ut så här:

```
class className
{
    ...
    ...
}
```

Första raden är klassens *huvud* och resten är klassens *kropp*. Det avgörande nyckelord som gör den här biten kod till en klass är **`class`** som vi kan hitta bland C#:s reserverade ord i tabellen på sid 36. Sedan står **`First`** i klassens huvud. Till skillnad från **`class`** är **`First`** inget reserverat ord utan ett namn som vi själva hittat på. Dock har namngivningen vissa enkla regler som vi så småningom kommer att ta upp. En av dem känner vi redan till: Det valda namnet får inte vara ett reserverat ord. Namnet måste direkt följa nyckelordet **`class`**. I vårt program är **`First`** namnet på klassen som vi också använder som programnamn. Men klassnamnet `className` behöver inte vara relaterat till filnamnet. Klassens kropp består av kod inom *klammer*paret **`{ }`** där den inledande klammern **`{`** markerar början och den avslutande klammern **`}`** slutet på klassen. Klamrarnas uppgift är att avgränsa klassen från andra delar av programmet. Vi kommer att använda ordet *klammer* som beteckning för tecknen **`{`** eller **`}`** på engelska *curly brackets* till skillnad från **`[ ]`** som kallas *hakparenteser*, på eng. *brackets*.

Metoden `Main()`

Efter den inledande klammern **`{`** som öppnar klassen **`First`**:s kropp (sid 42) står:

```
static void Main()
```

Det är huvudet till en *metod* vars namn är **`Main()`**. Vi kommer att ägna ett helt kapitel åt metoder. Just nu räcker det att känna till att en *metod* är kod som skrivs inuti kroppen av en klass. Metoden **`Main()`** har följande allmänna struktur:

```
static void Main()
{
    statement(s);
}
```

Första raden är metodens *huvud* och resten är metodens *kropp*. I kroppen skrivs en eller flera satser som *gör* någonting, i vårt exempel skriver ut text till konsolen på skärmen. Alla metoder i C# innehåller satser som *utför* vissa instruktioner.

Metodens huvud består av de reserverade orden **`static`** och **`void`**, metodens namn **`Main`** och parentesen **`()`** som är tom. Allmänt kan en metod ha ingen, en eller flera s.k. *parametrar*. Därför kallas parentesen *parameterlistan*. I vårt exempel har **`Main()`** ingen parameter. Vi får inte ändra **`Main()`**-metodens huvud därför att den inte är en egendefinierad metod. Vi har ju inte ens fått välja namnet, till skillnad från klassen **`First`**. Närmare bestämt är det *huvudet* till metoden **`Main()`** som vi inte har någon frihet att bestämma över. Huvudet är fördefinierat av systemet och måste skrivas som ovan, fast det finns lite andra varianter också. I kroppen däremot kan vi skriva vilken kod som helst.

En detaljerad förklaring till de reserverade orden **`static`** och **`void`** skjuter vi upp till senare. Här ska det räcka att bara kort nämna följande:

`static` innebär att den kan anropas utan att skapa objekt av klassen **`First`**,
`void` innebär att metoden **`Main()`** inte returnerar något värde.

Det är Virtual Machine (VM) som exekverar vårt program **`First`** genom att anropa metoden **`Main()`**. För att detta anrop ska kunna utföras behövs *modifieraren* **`static`** i metodens huvud. Även om det känns lite tråkigt att inte kunna säga mycket mer måste vi, för att inte tappa den röda tråden, rekommendera att du i alla dina egna program helt enkelt skriver av huvudet till metoden **`Main()`** precis som det står ovan. Medan **`static`** är en modifierare som reglerar reserveringen av minnesutrymme, är **`void`** en s.k. *returtyp* dvs en datatyp till metodens returvärde. **`void`** betyder ”inget returvärde” dvs metoden **`Main()`** returnerar inget värde när den anropas. Vad den gör är att den utför koden som står i dess kropp.

Liknande klassens kropp består även en metods kropp av ett antal satser inom klammerparet **`{ }`** där den inledande klammern **`{`** markerar början och den avslutande klammern **`}`** slutet på metoden. Klamrarnas uppgift är att gruppera satserna och avgränsa dem från andra delar av programmet. I kroppen till **`Main()`**-metoden kan vilka satser som helst stå. Satserna innehåller instruktioner till datorn. I programexemplet **`First`** står endast en sats som skickar text till konsolen.

Metoden `WriteLine()`

Körresultatet av programmet **`First`** dvs utskriften **`Mitt första C# program!`** har producerats av följande sats som står i klassen **`First`**:s **`Main()`**-metod:

```
Console.WriteLine("\n\tMitt första C# program!\n");
```

Detta är ett anrop av metoden **`WriteLine()`** som är fördefinierad i C#:s klassbibliotek, närmare bestämt i klassen **`Console`** som i sin tur finns i biblioteket **`System`**. För att kompilatorn ska kunna hitta metoden **`WriteLine()`** måste vi ange dess klass med sedvanlig punktnotation: **`Console.WriteLine()`**. Vad den gör är att skriva till konsolen och därefter byta rad pga tillägget **`Line`**. Det finns även metoden **`Write()`** som skriver ut text utan radbyte, vilket kommer att visas i programmet **`Output`** i nästa avsnitt (sid 49). Koderna **`\n`** och **`\t`** behandlas i nästa avsnitt.

Texten **`Mitt första C# program!`** som ska ska skickas till konsolen skrivs inom citationstecken – även kallade ”dubbelfnuttar” i populär svenska – eftersom all text i C#-kod måste sättas inom citationstecken. I utskriften kommer texten självfallet att visas utan citationstecken. *Citationstecknet* är koden för strängar:

> Strängar omgärdas i C#-kod av citationstecken `"  "`.

En *sträng* som är datatermen för text består av ett antal tecken där antal kan vara 0, 1, 2, ... Ett vanligt exempel på sträng är text som består av ett antal bokstäver. Allmänt kan dock vilka specialtecken som helst ingå i en sträng. När antalet tecken är 0 talar man om en *tom sträng* som i kod skulle kunna skrivas som `""`, medan koden `" "` är en sträng som inte är tom utan består av *ett* tecken, nämligen mellanslaget. Det är onödigt att skriva endast ett tecken som sträng eftersom det i C# finns ett annat, enklare sätt att lagra enstaka tecken. Med onödigt menas inte bara att det är slöseri med minnet utan att det även programmeringstekniskt sett, är dålig stil och dålig vana att blanda ihop dessa två olika typer av data. Vi kommer att förstå det bättre när vi tar upp *datatyper* (sid 56). Medan strängar måste specificeras med citationstecken `" ... "` används *apostrofer* `'…'` för att markera tecken:

> Enstaka tecken omgärdas i C#-kod av apostrofer `' '`, t.ex. **`'a'`**.

Apostroferna kring **`a`** – även kallade ”enkla fnuttar” i populär svenska – innebär att **`a`** är ett tecken, i detta fall en bokstav. Observera att det verkligen handlar om *apostrofer* kring **`a`** och inte om accent ´ eller ` som används i vissa bokstäver som é eller è. Så, i regel bör det enstaka mellanslaget skrivas som `' '` och inte som `" "`. Att `" "` kompileringsmässigt *kan* skrivas för mellanslaget beror på att ett tecken också är en sträng (OBS! Beakta dock vår programmeringstekniska anmärkning i slutet av förra sidan). Men det gäller inte det omvända: en sträng är inte alltid ett tecken, t.ex. när den består av flera tecken. Därför får apostrofer endast stå kring enstaka tecken, inte kring strängar som innehåller fler än 1 tecken.

3.2 God programmeringsstil

Hur gick det när du kompilerade ditt första C#-program? Om du hade fel märkte du kanske att felsökning kunde vara jobbigt. Det är den också, speciellt när programvolymen växer. Innan vi går vidare och utökar våra koder ska vi lära oss en teknik som gör felsökning enklare. Men vi gör det inte bara för att underlätta felsökning. Frågan är mer av generell karaktär: Hur skriver man bra strukturerade program och hur vänjer man sig vid att göra det *från början*? Att göra det från det allra första programmet är nämligen avgörande för att utveckla en god programmeringsstil när man fortsätter skriva kod. Titta på följande kod skriven i filen **`First_bad.cs`**. Känner du igen den?

```
using System; class First {static void Main() {Console.Write-
Line ("\n\tMitt första C# program!\n"); } }
```

Det är vårt första program **`First`** bortsett från kommentarerna. Koden ovan ”fungerar”, dvs den kan både kompileras och exekveras och producerar exakt samma utskrift som programmet **`First`** (sid 42). Kompilatorn struntar nämligen fullständigt i layouten. Den kontrollerar endast kodens *syntax*. Men det gör inte en människa som ska läsa din kod. Skulle du lämna in den till mig som din lösning på en övningsuppgift skulle du inte bli godkänd. Så här borde koden ovan istället se ut layoutmässigt och med kommentarer:

```
// First.cs                              Filnamnet
// Skriver ut text till konsolen (svarta fönstret)
// Metoden Main() anropas automatiskt när programmet körs
// Den i sin tur anropar metoden WriteLine() i klassen Console
// Klassen Console finns förprogrammerad i biblioteket System

using System;                     // Krävs för klassen Console

class First                       // Klassnamnet
{
   static void Main()             // Metoden Main()
   {
      Console.WriteLine("\n\tMitt första C# program!\n");
   }
}
```

Förutom de krav som kompilatorn ställer för att överhuvudtaget kunna få programmet i exekverbar form, finns andra krav på vårt sätt att skriva kod. Det handlar om krav på god programmeringsstil. Dessa krav är minst lika viktiga som kompileringskraven.

God programmeringsstil innebär att man skriver kod så att *andra* kan använda och underhålla den. Alla professionella program som du använder på din dator, operativsystemet, editorer, skriv-, rit-, kalkyl-, spel- och andra applikationer har skrivits

med detta i åtanke. Program måste vara användarvänliga. God programmeringsstil innebär att vi lämnar ifrån oss kod som andra kan modifiera och vidareutveckla. Program måste vara lätt ändringsbara. Vi kommer själva att ha glädje av det, om vi vid ett senare tillfälle vill förbättra våra program. Därför ställs följande krav på god programmeringsstil:

- Förståelighet
- Användarvänlighet
- Strukturering
- Ändringsbarhet

Och det är därför vi har skrivit vårt första, och kommer att skriva alla våra programexempel, med följande stilelement:

1. **Indragningar** är ett stilelement som används för att uppfylla de ovannämnda kraven på god programmeringsstil. I programmet **`First`** ser man att vissa rader är indragna, närmare bestämt de rader som utgör **`Main()`**-metoden. Dessa indragningar ska markera att raderna tillhör **`Main()`**. Ett exempel på dålig programmeringsstil ser vi på förra sidan där koden komprimerats till tre rader. Genom att låta koden ta mer plats blir den mer lättläst. Den allmänna regeln är att indrag ska återspegla programmets logiska struktur. Rekommendationen är att göra tydliga indragningar dvs inte alltför små. Tumregeln är: mellan tre och fem mellanslag.

2. **Separata rader** tillämpas för att öka kodens läslighet. Varje sats ska som regel stå på en separat rad. Men även klamrarna **`{`** och **`}`** står på egna rader. Detta markerar klamrarnas utomordentligt stora betydelse för att gruppera vissa satser och avgränsa dem från andra delar av programmet. Klamrarna utgör alltså gränser som ska vara mycket tydliga. Dessutom står klass- och metodhuvuden alltid på separat rad, i vårt fall klasshuvudet **`class First`** och metodhuvudet **`static void Main()`**.

3. **Kommentarer** ska förklara koden. Hur mycket och på vilket sätt ska man skriva dem? Rekommendationen är att kommentarerna ska vara korta och inte blandas med koden. Detta gäller speciellt radkommentarerna som annars skulle göra koden mindre lättläst. Vill man skriva längre kommentarer ska man helst skriva en dokumentation till programmet. Denna kan antingen ligga helt separat från koden, t.ex. i en textfil, eller skrivas som *blockkommentar* i början eller på andra ställen av programmet. En blockkommentar i C# kan bestå av flera rader och ska inledas med **`/*`** och avslutas med **`*/`**.

Slutligen ska än en gång påpekas att programfel ur stilsynpunkt inte får bedömas som mindre allvarliga än kompileringsfel. Attityden ”först ska jag lära mig koda, god programmeringsstil kan jag lära mig senare” är ett allvarligt misstag som nybörjare gör pga oerfarenhet, vilket kan leda till slöseri med tid och energi vid felsökning och till dåligt strukturerade program i längre perspektiv. Man kan tröttna på programmering – speciellt vid felsökning – om man inte *från början* lägger stor vikt vid god programmeringsstil.

3.3 Radbyte och tabulator

I programexemplet **First** fanns bara en sats i koden. Den resulterade också i en rad på skärmen. Men en sats i koden kan producera flera rader i utskriften med en speciell kodsymbol som åstadkommer radbyte i utskriften: **\n** är en sådan där **n** står för **n**ewline och **** är ett speciellt styrtecken som kallas *escapetecknet* och gör att **n** tolkas som **n**ewline och inte som bokstaven **n**. Båda tillsammans, **\n**, är *en escapesekvens*. På svenska betyder *to escape* att fly. Escapesekvenser inleds med tecknet backslash **** åtföljt av *ett* tecken. Med **** vill man *fly* från tecknets vanliga betydelse och ge det en *annan* innebörd. Skriver man i koden **\n** inom en sträng blir det radbyte i utskriften.

En annan escapesekvens är **\t** där **t** står för **t**abulator där **** gör att **t** inte tolkas som bokstaven **t** utan åstadkommer en horisontell indragning med åtta mellanslag. Båda escapesekvenserna **\n** och **\t** kan bakas in i strängar, men skulle kunna även kodas som enskilda tecken och skickas separat till utskrift. Samma sak är det med mellanslag. Skriver man ett mellanslag inom en sträng skickas den med till utskrift, annars inte. Följande program använder radbyte för att skriva ut på flera rader:

```
// LineBreak.cs
// Skriver ut text med radbyte och indragning
// FLERA rader utskrift producerad av EN sats i koden
// Radbyte med \n och tabulator med \t
using System;

class LineBreak
{
   static void Main()
   {
      Console.WriteLine("\n\tC#\n\tär\n\tkul !\n");
   }
}
```

Precis som i programmet **First** anropas även här metoden **WriteLine()** endast en gång. Men i strängen som ska skrivas ut förekommer **\n** *fyra gånger* på fyra ställen: Den första ger en tom rad, den andra radbyte efter **C#**, den tredje efter **är** och den fjärde efter **!** vilket man ser när man kör programmet **LineBreak**:

```
        C#
        är
        kul !
```

Den sista tomma raden produceras av **WriteLine()** som byter rad efter utskrift pga **ln** som står för **line**. Dessutom ingår ett mellanslag i strängen som skickas till utskrift. Det står inom **WriteLine**-metodens parentes och kommer i utskriften

att hamna på samma ställe som i koden, innan ordet **är**. Utskriften visar också att **\n** åstadkommer samma sak som en tryckning på **Enter**-tangenten. **\n** är kodsymbolen för **Enter**.

Metoden `Write()`

Medan programmet **LineBreak** producerar flera rader utskrift med en sats i koden, skriver följande program ut en enda sammanhängande rad med flera satser i koden. På så sätt visar programmet även skillnaden mellan **WriteLine()** och **Write()**.

```
// Output.cs
// Två utskriftssatser, men endast EN rad utskrift
// Metoden Wrtite() skriver ut text utan radbyte efteråt
using System;

class Output
{
   static void Main()
   {
      Console.Write("\n\tDetta är EN rad text produc");
      Console.WriteLine("erad av två utskriftssatser.\n");
   }
}
```

Här finns 2 satser som skriver ut till konsolen. Men de producerar en rad utskrift, vilket beror på att det inte finns något **\n** i slutet av den första utskriftssatsen. Det ser man när man kör programmet ovan:

```
    Detta är EN rad text producerad av två utskriftssatser.
```

Observera också att utskriften blir **... producerad ...** och inte **... produc erad ...** vilket beror på att inget mellanslag skrivs i kodens första utskriftssats efter **produc** och inte heller i andra före **erad**. Man får intrycket att det endast är en utskrift på skärmen. Och intrycket är rätt: Hur många utskriftssatser man än skriver, det handlar om en enda utskrift som initieras av metoden **Write()** som skriver ut utan att göra radbyte efteråt. Metoden **WriteLine()** fortsätter sedan exakt där **Write()** slutat, men gör radbyte efteråt.

Men vad gör man om en sträng är för lång och inte ryms på en rad i koden. Lösningen är att dela upp strängen i två eller flera delsträngar, bryta rad på ett lämpligt ställe och använda symbolen **+** för att slå ihop delsträngarna. Självklart kommer tecknet **+** i detta sammanhang inte längre att ha betydelsen som addition utan en annan, vilket tas upp i nästa avsnitt.

3.4 Konkatenering med +

```
// Concat.cs
// Ritar en hjärtlig hälsning
// Anropar metoden Write() utan radbyte
// Slår ihop strängar med konkateneringsoperatorn +
using System;

class Concat
{
   static void Main()
   {
      Console.Write("\n\t     *       *            " +
                    "\n\t   *   *   *   *          " +
                    "\n\t *       *       *        " +
                    "\n\t*                 *       " +
                    "\n\t*                 *       " +
                    "\n\t*     Grattis     *       " +
                    "\n\t *                *       " +
                    "\n\t   *             *        " +
                    "\n\t     *         *          " +
                    "\n\t       *   *              " +
                    "\n\t         *                " +
                    "\n\n");
   }
}
```

Här finns endast en utskriftssats med **`Write()`**-metoden och vi använder oss av **+** för att bryta rad i koden. Operatorn **+** betyder här inte addition av tal utan sammanslagning av strängar. Konceptet kallas *överlagring av operatorer* och förekommer ofta i programmering: Symbolen **+** har flera olika betydelser. Vilken betydelse som gäller aktuellt beror på sammanhanget. Finns det tal på *båda* sidor av operatorn **+** tolkas den som vanlig addition. Står det däremot en sträng på *någon* sida av operatorn tolkas **+** som sammanslagning av strängar. **`Concat`** ger följande utskrift:

```
        *       *
      *   *   *   *
    *       *       *
   *                 *
   *                 *
   *     Grattis     *
    *                *
      *            *
        *        *
          *   *
            *
```

Concatenation

betyder *sammanslagning* och förekommer i olika sammanhang inom datalogin, inte bara i C#[*]. När tecknet **+** sätts mellan strängar eller där bara en sida är en sträng, tolkas det som *konkateneringsoperator*. T.ex.: `Console.Write(25 + " katter");` ger utskriften **25 katter**. Det räcker att *någon* sida av **+** är en sträng. Konkateneringsoperatorn **+** omvandlar även automatiskt den andra operanden till en sträng om endast en operand en sträng. Dvs i satsen ovan omvandlas *talet* **25** till *strängen* `"25"` och slås ihop med strängen `" katter"`. Däremot ger `Console.Write(25 + 5);` utskriften **30** eftersom båda operander tolkas som tal och **+** som additionsoperatorn. Mer om skillnaderna mellan *tal*, *tecken* och *sträng* kommer att behandlas senare när vi lär oss *datatyper* (sid 56).

I programmet **Concat** slås ihop flera strängar till en och skrivs som en enda lång sträng. Metoden **Write()** anropas endast en gång, istället upprepas konkateneringsoperatorn **+** mellan de olika delar som ska skrivas ut. Den allmänna strukturen kan se ut så här:

```
Console.Write(... + "..." + ... + "...");
```

där **...** står för de delar – text eller tal – som ska skrivas ut. Koden ovan kan sträcka sig över flera rader men måste avslutas med ett enda semikolon då det är en enda sats. Glöm inte heller att stänga parentesen. Radbrytningar i koden får inte göras mitt i en sträng eller i mitt i ett ord. Det gäller regeln:

> Mitt i en sträng eller ett ord får man inte bryta raden i C#-kod.

T.ex. ger följande radbrytning i koden kompileringsfel:

```
Console.Write("Detta är en
                utskriftsrad.");
```

Även detta ger kompileringsfel:

```
Console.Write("Detta är en"
                "utskriftsrad.");
```

Lösningen är konkatenering med **+** :

```
Console.Write("Detta är en " +
                "utskriftsrad.");
```

---

[*] T.ex. i Java finns metoden **concat()** som konkatenerar strängar. I C++ finns metoden **strcat()** som står för **str**ing **cat**enation och gör samma sak. I Unix, 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 filen **nyfil**.

Övningar till kapitel 3

3.1 Mata in koden till programmet **`First`** (sid 42), kompilera och kör.

a) Skriv om programmet **`First`** genom att ta bort **`using`**-direktivet och modifiera istället utskriftssatsen så att den kan kompileras och ger samma resultat som programmet **`First`**.

b) Undersök skillnaderna mellan *apostrof*, *citationstecken* och *accent*.

3.2 Sätt in följande kod i ett C# program för att testa vad den ger för utskrift:

```
Console.Write  ("**\n");
Console.WriteLine("***");
Console.WriteLine("****");
Console.Write  ("*****\n");
Console.WriteLine("******");
Console.WriteLine("*****");
Console.Write  ("****\n");
Console.WriteLine("***");
Console.WriteLine("**\n");
```

a) Ersätt alla anrop av **`Console.Write()`** med **`Console.WriteLine()`** och ändra lite i koden utan att utskriften ändras.

b) Lägg till lite kod i varje sats så att hela den utskrivna figuren hamnar lite längre bort från konsolfönstrets vänstra och övre rand.

3.3 Skriv ett program och testa vilken utskrift följande utskriftssatser ger:

```
Console.Write    ("Jag");
Console.Write    ("heter");
Console.WriteLine("K.\n Vad heter du?\n");
```

Lägg till resp. ta bort mellanslag, radbyte och tabulator på lämpliga ställen för att få en snygg utskrift, utan att slå ihop de tre satserna till en.

3.4 Skriv ett C# program som en gång skickar till **`WriteLine()`** koden:

```
"Resultatet är " + 8 + 3
```

och en annan gång: `"Resultatet är " + (8 + 3)`

Förklara skillnaden i utskrifterna. Hur måste **`+`** tolkas på de olika ställena?

3.5 Vilka utskrifter ger följande satser? Sätt in dem i ett program och testa.

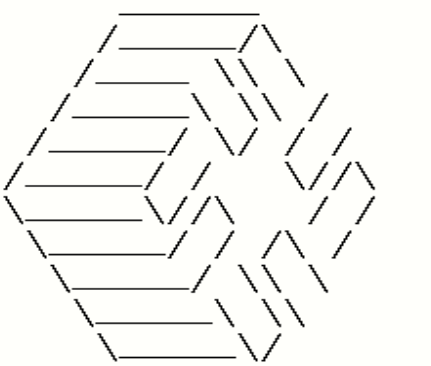
```
Console.WriteLine("*\n**\n***\n****\n*****");
Console.WriteLine("*****\n****\n***\n**\n*");
```

Skriv om koden så att du får samma utskrift med en enda utskriftssats i koden.

3.6 Skriv in koden till programmet **`Concat`** (sid 51), kompilera och kör det. Modifiera det till att skriva ut en oval byggd av stjärnor (**`*`**).

3.7 Skriv ett program som skriver ut en triangel byggd av stjärnor (**`*`**).

3.8 Rita figuren till höger i konsolen med en enda utskriftssats genom konkatenering:



Se upp för skillnaden mellan slash **`/`** och backslash **`\`**. Använd två backslash **`\\`** i koden – som en escapesekvens inbakad i den konkatenerade strängen – för att åstadkomma en backslash **`\`** i utskriften (Läs om escapesekvenser på sid 97).

Kapitel 4

Grundbegrepp i programmering

4.1 Datatyper

Hittills har vi i våra program skrivit ut endast text eller tecken. Datatermen för text är *sträng*, ett antal tecken. Det vanligaste exemplet är ett antal bokstäver. Men även alla möjliga specialtecken kan ingå i en sträng. I koden har vi avgränsat tecken med apostrofer `' '` och strängar med citationstecken `" "` (sid 46). Mer exakt handlar det om tecken- och sträng*konstanter*. Data som inte kan ändras kallas *konstanter*. De skickas som de är, från programkod till bildskärm. T.ex. `'a'` är en teckenkonstant. Apostroferna kring `a` talar om att `a` ska tolkas som tecken. Men hur är det med siffror? De kan vara tal, tecken eller sträng. T.ex. `9` är en talkonstant. I kod måste den skrivas utan apostrofer för att tolkas som tal. *Utan* apostrofer tolkar kompilatorn `9` som tal. *Med* apostrofer `'9'` tolkas `'9'` som ett tecken. Ytterligare en tolkning är `"9"` som sträng. På skärmen ser man ingen skillnad. Alla dessa tre koder `9`, `'9'` och `"9"` skriver ut en `9` på skärmen.

Meningen med att skilja åt dem är att kan man *räkna* med tal, inte med tecken eller strängar. Strängar kan man konkatenera , vilket resulterar i text. Konkatenering av siffror däremot ger tal som av människan tolkas enligt det decimala talsystemet, men av datorn som en sträng bestående av siffror. Apostrofer, citationstecken eller ingenting kring `9` leder till att datorn *tolkar* data på det sätt som vi menar.

Följande program demonstrerar skillnaderna mellan olika typer av data, närmare bestämt mellan tal, tecken och sträng för att introducera begreppet *datatyp*:

```
// Datatype.cs
// Utskrift av olika typer av data: tal, tecken och text
// I kod skrivs talkonstanter så här:                  9
//              teckenkonstanter inom apostrofer:     '9'
//              strängkonstanter inom citationstecken: "9"
using System;

class Datatype
{
  static void Main()
  {
    Console.WriteLine(
    "Detta är talet " + 9                          + '\n'   +
    "Talet 9 + talet 9 ger "     + (9 + 9)         + "\n\n" +

    "Detta är tecknet " + '9'                      + '\n'   +
    "Tecknet 9 + tecknet 9 ger " + ('9' + '9')     + "\n\n" +

    "Detta är stängen " + "9"                      + '\n'   +
    "Strängen 9 + tecknet 9" +
                " + talet 9 ger "  + ("9" + '9' + 9) + '\n');
   }
}
```

En körning av programmet `Datatype` ger följande utskrift:

```
Detta är talet 9
Talet 9 + talet 9 ger 18

Detta är tecknet 9
Tecknet 9 + tecknet 9 ger 114

Detta är stängen 9
Strängen 9 + tecknet 9 + talet 9 ger 999
```

Koden **9 + 9** ger utskriften **18**. Det är självklart: Talet **9** adderas med talet **9** och resultatet **18** skrivs ut. Men koden **('9'+'9')** ger utskriften **114**, vilket beror på att **'9'** inte är tal utan tecken. Hur lagras tecken i datorn? När vi trycker på en tangent överförs en kod i form av ett binärt heltal – en sekvens av ettor och nollor – till datorn. Varje tecken har sin speciella kod, s.k. *ASCII-kod*. *ASCII* är en standard för omvandling mellan tecken och heltalskoder. Vi kommer att ta upp detta mer detaljerat senare (sid 93). Tecknet **'9'** har ASCII-koden 57 som adderas med 57, så att **('9'+'9')** blir **114**. Här tolkas nämligen plustecknet som vanlig addition. Saker och ting sker i följande ordning: *Först* tolkas **'9'** som ASCII-koden 57, *sedan* adderas båda tecknens ASCII-koder vilket resulterar i **114**, *sist* skrivs ut resultatet. Facit: Det är en väsentlig skillnad mellan *talet* **9** och *tecknet* **'9'**.

Varför gäller inte samma resonemang i koden **"Detta är tecknet " + '9'** dvs varför resulterar utskriften av **'9'** inte i 57 så att det skrivs ut **Detta är tecknet 57**? Här tolkas plustecknet *inte* som vanlig addition utan som konkatenering (sid 52). Anledningen är att före **+** står strängen **"Detta är tecknet "** dvs operationen är initierad av en sträng. Därfö omvandlas även **'9'** till en sträng. Alltså skrivs ut hela den konkatenerade *strängen* **Detta är tecknet 9**. När det gäller **('9'+'9')** står både till vänster och höger om plustecknet ASCII-koderna till tecknen **'9'**. Alltså bildas summan av dem som är ett *tal*. Då summan bildas *före* utskriften, står i parentesen redan talet **114** innan det skrivs ut.

Även plustecknet i **"9" + "9"** är till skillnad från **9 + 9** och **'9'+'9'** inte vanlig addition utan konkatenering av strängarna **9** och **9**. Därför sätts dessa strängar mekaniskt ihop till strängen **99** innan den skrivs ut. Här är det citationstecknen som talar om vilken *typ av data* det är, nämligen sträng.

Anledningen till skillnaden mellan *talet* **9** och *tecknet* **'9'** är att de lagras i datorn på olika sätt och har olika stora minnesutrymmen. Tal lagras direkt medan tecken måste kodas först. Tal omvandlas till ettor och nollor med hjälp av olika algoritmer beroende på om det är heltal eller decimaltal. Datorn måste ha informationen om vilken *typ av data* det handlar om, för att kunna välja rätt algoritm. Allt som kan göras med tal kan inte göras med tecken och omvänt: T.ex. kan tal adderas medan tecken inte kan det. Samma sak är det med *strängar* som inte heller kan adderas, de kan däremot konkateneras. De tillhör en tredje typ av data som varken är tal eller tecken, fast de är sammansatta av tecken. Det finns ännu fler typer av data som vi inte lärt känna ännu.

För att internt kunna skilja mellan olika typer av data, digitalisera dem och åter presentera dem i ursprungligt skick har man i programmering begreppet *datatyp*.

Vad är en datatyp?

En datatyp är en föreskrift om

1. hur en viss typ av data ska lagras i datorn,
2. hur mycket minne denna typ av data tar och därmed hur stora värden den kan lagra (det tillåtna värdeområdet),
3. vilka operationer man får utföra med denna typ av data.

Olika programmeringsspråk behandlar sina datatyper på lite olika sätt. C# är ett *strikt typbestämt* språk (eng. *strongly typed language*) vilket innebär att kontrollen över datatyper är väldigt hård. All data som behandlas i ett C# program måste utan undantag vara typbestämd. Man måste explicit ange datatypen till alla värden man arbetar med. Data utan uppgift om datatypen kan inte bearbetas.

Redan våra C# program i förra kapitel innehöll symboler som gav information om datatypen: Tecken avgränsas med apostrofer `' '` och strängar avgränsas med citationstecken `" "`. Dvs `' '` är symbolen för *datatypen tecken* och `" "` symbolen för *datatypen sträng*. T.o.m. avsaknaden av dessa symboler är själv en symbol: Förekommer varken apostrofer eller citationstecken, t.ex. hos **`9`**, anses **`9`** vara av *datatypen tal*. Denna symbolik används så länge vi har att göra med konstanter, närmare bestämt med tal-, tecken- eller strängkonstanter.

Skriver vi däremot en *bokstav* utan apostrofer i koden, t.ex. **`a`** blir det kompileringsfel. Orsaken är att C#-kompilatorn inte kan bearbeta **`a`** då den inte kan identifiera **`a`**:s datatyp. Satsen **`Console.Write('a');`** kan kompileras och ger utskriften **`a`** eftersom **`'a'`** tolkas som tecken. T.o.m. **`Console.Write("a");`** kan kompileras och ger samma utskrift eftersom **`"a"`** tolkas som sträng. Även *en* bokstav eller *ett* tecken kan anses som sträng, den minsta möjliga. Men **`a`** utan apostrofer eller citationstecken är i C# varken en tecken- eller en strängkonstant. Talkonstant kan det inte heller vara. Ja, **`a`** är ingen konstant alls. Vad är **`a`** i så fall? Satsen **`Console.Write(a);`** kan inte kompileras och ger kompileringsfelet *The name 'a' does not exist in the current context*, dvs **`a`** är ett okänt namn. *Okänt* därför att det inte har definierats ett sådant namn med hjälp av datatypen. *Namn* därför att kompilatorn i det enklaste fallet förväntar sig här namnet på en *variabel*. Det som förorsakar kompileringsfelet är att datatypen saknas: **`a`** tolkas som en variabel vars datatyp inte är specificerad. Därför anses den som odefinierad. Men varför måste en variabels datatyp vara specificerad? Vad exakt är en variabel och hur definieras en sådan i C# dvs hur specificeras dess datatyp? Kort sagt, en variabel behövs för att lagra data i datorns RAM-minne som ska sedan användas i programmet. Nästa avsnitt berättar i detalj hur man gör det.

4.2 Deklaration och initiering av variabler

C# är ett s.k. *strikt typbestämt programmeringsspråk*, vilket innebär att alla variablers datatyp måste explicit anges i programmet *före* användningen. Kod som innehåller variabler utan uppgift om datatypen kan inte kompileras. Det finns flera goda skäl för det här kravet. Det viktigaste är att kompilatorn måste reservera plats för variabelns värde. En variabel är en platshållare för ett värde. För att kunna lagra detta värde behövs information om platsens storlek, om sättet att omvandla värdet till ettor och nollor och om vilka operationer man får utföra med värdet. All denna information finns samlad i datatypen. Först ska vi precisera begreppet *variabel*.

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.

Man kan jämföra en variabel med en låda och variabelns värde med lådans innehåll. Variabelns namn är då lådans etikett. *Värde* är data i största allmänhet, dvs kan vara – beroende på datatypen (sid 58) – tal, tecken, men även ett sanningsvärde, en sträng, längre text, en fil, ja t.o.m. en bild, … . Till skillnad från en konstant som inte kan ändra sitt värde (sid 56) kan en variabels värde ändras under en programkörning. För att kunna göra det måste variabeln ha ett namn i programmet. Hos en variabel måste man alltid skilja mellan *namnet* och *värdet*. Men vilka namn får vi ge till våra variabler? Vi har en ganska stor frihet för detta val. Dock är som vanligt friheten relativ och vi måste följa vissa enkla regler som gäller för all namngivning i C# och som vi tar upp här.

Regler för namngivning:

Ett *namn*, även kallat *identifierare*, kan bestå av ett eller flera tecken och får endast innehålla

1. **Alla bokstäver (inkl. svenska specialtecken)**
2. **Alla siffror**
3. **Understreck (underscore _)**
4. **Tecknet @**

Men: **Namnets första tecken får inte vara en siffra.**
C#:s reserverade ord (sid 36) får inte användas.

Exempel på identifierare är namn på variabler, konstanter, metoder, klasser, objekt osv. Bland alla specialtecken får endast understreck (underscore _) och tecknet @ användas. Självklart får en identifierare inte innehålla mellanslag för då tolkas de inte som *en* utan *flera* identifierare. Mellanslag är *avskiljare* mellan två ord. T.ex.

är **number1** och **numberOne** giltiga variabelnamn, men inte **1number** eller **number one**. Däremot går det bra med **number_one** eller **number_1**, ja t.o.m. **_number**. Svenska specialtecken är inte förbjudna. Följer man inte reglerna ovan får man kompileringsfel. Men för att göra program lättare att läsa och förstå, finns det också anledning att följa följande

Rekommendation för namngivning:

Välj namn som är *beskrivande* dvs beskriver identifierarens *roll* i programmet. Bibliotekens klassnamn bör inte användas som identifierare.

Denna rekommendation baseras på de krav som god programmeringsstil ställer (sid 48). För att göra våra program lättare att läsa, förstå och göra ändringar i, måste namnen vara *beskrivande*. I programet **Variable** nedan har vi valt **number1** och **number2** som namn för programmets variabler. Namnen kan i princip väljas godtyckliga, dvs skulle lika bra kunna vara t.ex. **a**, **b**, **x**, **no**, **account** eller vad som helst – upp till reglerna för namngivning. Men vårt val grundas även på rekommendationen ovan: Vi ska lagra tal i variablerna **number1** och **number2**.

Att *definiera* eller *skapa* en variabel innebär att reservera plats i datorns RAM-minne åt dess värde. Det gör man i koden genom att ange variabelns datatyp. Att endast ange datatypen kallas även *deklaration*, man det är definitionen som reserverar minne. Med *tilldelning* menar man att ge en variabel ett värde. *Initiering* är den allra första tilldelningen dvs att ge variabeln ett startvärde som sedan kan ändras. Följande program demonstrerar *deklaration* och *initiering* samt *tilldelning* (eng. *assignment*) av variabler. Även *tilldelningsoperatorn (=)* introduceras:

```
// Variable.cs
// Operationerna + och - som är definierade för tal. Defini-
// tion och initiering av variabler med datatypen int. Varia-
// namn står för värdet variabeln har vid aktuell tid
using System;

class Variable
{
  static void Main()
  {
    int number1, number2, sum, diff;     // Deklaration och
    number1 = 9;                  // initiering av variabler
    number2 = 3;
    sum  = number1 + number2;

    Console.WriteLine("\n\tAddition definierad för int:\t" +
                 number1 + " + " + number2 + " ger " + sum);
    number1 = 11;                // Ändring av variabelns värde
    diff = number1 - number2;
    Console.WriteLine("\tSubtraktion definierad för int:\t" +
           number1 + " - " + number2 + " ger " + diff + '\n');
   }
}
```

I programmet **Variable** anropas metoden **WriteLine()** två gånger. Vid första anropet har variabeln **number1** värdet **9** medan variabeln **number2** har värdet **3**. Vid andra anropet har **number1**:s värde ändrats till **11** medan **number2**:s värde fortfarande är **3**. Det är därför vi har **9** och **3** inblandade i additionen medan **11** och **3** ingår i subtraktionen vilket bekräftas av körresultatet:

```
Addition definierad för int: 9 + 3 ger 12
Subtraktion definierad för int: 11 - 3 ger 8
```

Men låt oss följa koden från början. I programmet **Variable** förekommer fyra variabler **number1**, **number2, sum** och **diff**. De behövs för att kunna lagra fyra värden. Namnen har vi hittat på, men enligt rekommendationen för namngivning (sid 59) ska man för läslighetens skull välja beskrivande namn. I **Main()** definieras variablerna genom att inleda med datatypen **int** följd av en kommaseparerad lista:

```
int number1, number2, sum, diff;
```

En sådan konstruktion är endast tillåten om alla variabler är av samma datatyp. **int** är ett reserverat ord som står för ***int****eger number*, heltal på engelska och symboliserar den i C# fördefinierade datatyp som kan lagra heltal. Man skulle kunna även dela upp satsen ovan i fyra separata satser som är helt likvärdiga med den:

```
int number1;
int number2;
int sum;
int diff;
```

Generellt kan deklaration av en variabel i C# beskrivas så här:

datatyp variabel;

Tilldelningsoperatorn

Tilldelning betyder att *ge* variabeln ett värde. I programmet **Variable** tilldelas variabler värden med en symbol som till synes är likhetstecknet vilket är missledande då symbolen **=** i C# *inte* betyder likhetstecknet. I själva verket representerar **=** i C# en operator som utför tilldelning och därför heter *tilldelningsoperator*. Den första sats i programmet **Variable** där tilldelningsoperatorn används är

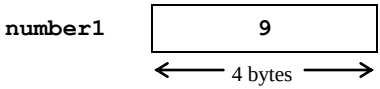
```
number1 = 9;
```

vars innebörd man skulle kunna beskriva med följande pseudokod:

```
Variabel ←—— Värde
```

Variabeln **number1** får värdet **9** dvs minnescellen **number1** får *innehållet* **9**. Tilldelning med **=** kan snarare jämföras med en pil ←—— som går från höger till vänster. Vi måste därför fortsättningsvis vara vaksamma på att vi inte av gammal

vana tolkar likhetstecknet som likhet utan som *tilldelning*. I C# finns en annan symbol för likhet (==) som används i villkor för att jämföra två värden med avseende på likhet. Efter tilldelningen av variabeln **number1** ser RAM-minnet ut så här:



I minnescellen hamnar värdet **9** – omvandlat till ettor och nollor förstås – och vi kan sedan komma åt detta värde genom att referera till **number1** eftersom variabelnamnet är för oss den logiska (mjukvarumässiga) adressen till den fysiska minnescellen. Om vi nu efter tilldelnigen skriver satsen **Console.WriteLine(number1);** får vi variabelns *värde* **9** utskrivet i konsolen.

Samma sak är det förstås med variabeln **number2** som i programmet **Variable** får värdet **3**. Efter tilldelningen av variablerna **number1** och **number2** utförs additionen **number1 + number2**. Här adderas *värdena* (innehållet) lagrade i variablerna **number1** och **number2**. Resultatet tilldelas variabeln **sum**. Vi refererar till värdena med hjälp av variablerna. Att additionen **+** görs *först* och tilldelningen **=** *sedan* beror på parenteserna i satsen **sum = (number1 + number2);** Men även utan parenteser hade vi fått samma resultat då **+** binder starkare än **=** . Slutligen skrivs i utskriftssatsen alla tre variablers värden ut, konkatenerade med lite text för att göra utskriften användarvänlig.

Initiering av variabler

Den allra första tilldelningen av en variabel efter definitionen kallas *initiering*. Det kan ske på olika sätt. I programmet **Variable** har vi gjort det med tilldelningsoperatorn.

Vad händer om man definierar en variabel men glömmer initieringen? Vad händer t.ex. om vi försöker att skriva ut eller på något annat sätt komma åt värdet på en oinitierad variabel genom att referera till den? Man skulle kunna tänka sig att det går bra – i alla fall kompileringsmässigt – då vi åtminstone definierat variabeln och på så sätt skapat minnesutrymme för den. Så är det nämligen i andra språk, t.ex. i C++. Men C# sätter stopp för detta och bannlyser därmed oinitierade variabler från alla C# program:

> Variabler som inte initieras innan de används leder till kompileringsfel.

Fördelen med denna strikta regeln är att man undviker förekomsten av s.k. ”skräpvärden” – godtyckliga slumptal som rent fysiskt råkar finnas på de platser kompilatorn reserverar minne. Detta är möjligt i C++, men C# har stoppat denna möjlighet och tillfört därmed språket mer säkerhet och stabilitet. Vi är alltså tvungna att arbeta med variabler som är *både* definierade *och* initierade, s.k. *väl definierade* variabler. En bra vana att initiera sina variabler är att tilldela dem ett värde direkt i samband med deklarationen. En bra teknik för det är följande:

Deklaration och initiering i samma sats

Ett bra medel mot att glömma variabelinitieringen är att inte avsluta deklarationssatsen förrän man gett variabeln ett värde. Följande program visar att C# tillåter att definiera och initiera variabler i en och samma sats:

```
// DefInit.cs
// Deklaration och initiering i samma sats
// Summan bildas direkt i utskriftssatsen: Sparar en variabel
// Vid utskrift konkateneras vanlig text, variabler & uttryck
using System;
class DefInit
{
  static void Main()
  {
    int number1 = 9;          // Deklaration och initiering
    int number2 = 2;          // Initiering vid deklarationen

    Console.WriteLine("\n\t" + "Summan av " + number1 +
    " och " + number2 + " är " + (number1 + number2) + '\n');
   }
}
```

Programmet ovan producerar denna utskrift:

```
        Summan av 9 och 2 är 11
```

Medan i programmet **`Variable`** (sid 60) deklarationen och initieringen av variabler gjordes i separata satser har dessa satser i programmet **`DefInit`** slagits ihop: Variabeln **`number1`** har blivit definierad och initierad i en och samma sats:

```
int number1 = 9;
```

Samma sak kan man göra med **`number2`**. Detta är möjligt, för man måste inte definiera alla variabler i början av programmet. Man kan göra det när det behövs, bara man definierar en variabel *innan* man initierar den. Det går t.o.m. att slå ihop de två första satserna i **`DefInit`** till en:

```
int number1 = 9, number2 = 2;
```

De två variablers deklaration och initiering kan separeras med komma, vilket endast är möjligt om variablerna har samma datatyp, som då skrivs en gång i början av satsen. Ska båda variablerna ha samma värde kan man göra en *dubbelinitiering*:

```
int number1, number2;       // Separat deklaration
number1 = number2 = 2;      // Dubbelinitiering
```

Men då måste deklarationen stå separat innan. Kom ihåg att tilldelningsoperatorn alltid tilltilldelar som en pil från höger till vänster. Därför får variabeln **`number2`**

först värdet **2**. Variabeln **number1** får samma värde, dvs variabeln **number2**:s värde som redan är **2**.

Deklaration vs. definition

I litteraturen används ofta begreppet *deklaration* istället för *definition av variabler*. Orsaken är att de *skrivs* i en och samma sats. Så är det också i våra egna programexempel: Alla variabeldefinitioner är samtidigt variabel*deklarationer*. Ändå kan det vara av intresse att uppmärksamma deras begreppsmässiga skillnad, speciellt när man tillämpar begreppen på objekt, klasser och metoder. C#:s föreskrivna programarkitektur har eliminerat den praktiska relevansen av denna skillnad – åtminstone när det gäller variabler.

Vad är deklaration?

Vad gör man när man deklarerar skatt? Man *anger* att det finns en inkomst att beskatta. Deklarationen skapar inte inkomsten utan *hänvisar* bara till den. Själva inkomsten har skapats i en helt annan process som inte har det minsta att göra med skattedeklarationen. Samma sak är det när man deklarerar en vara hos tullen. Deklarationen producerar inte varan utan ger endast information om dess existens. Man kan deklarera en sak – vare sig inkomst eller vara – endast om saken redan finns, har skapats eller kommer att skapas. Deklarationen kan inte ersätta skapandeprocessen. Men skapandet kan inkludera deklarationen: Man kan t.ex. ha det som rutin att deklarera vid skapandet.

När det gäller variabler talar deklarationen om för kompilatorn att det i programmet *finns* en variabel som t.ex. heter **number1** och att den är av datatypen **int**. *Deklarationen* är en hänvisning till variabelns existens och dess datatyp. När man däremot pratar om definition menar man alltid *skapandet* av en variabel dvs reservering av minnesutrymme för variabeln. Endast deklarationen skapar inte minnesutrymme utan gör bara att kompilatorn kan *tolka* informationen. Deklarationen gör att kompilatorn förstår vad ordet **number1** är för någonting, nämligen ett namn till en variabel. Uppgiften om datatypen **int** ger kompilatorn möjligheten att *hantera* informationen.

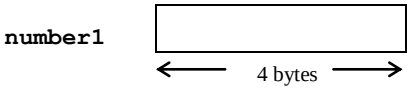
Vad är definition?

Definition skapar en sak – variabel eller objekt – från scratch. I vårt fall skapar den en variabel dvs reserverar minnesutrymme i datorns RAM för lagring av ett värde. Namnet till variabeln används i programmet för att komma åt minnesutrymmets innehåll, för att skriva, läsa eller ändra värdet. Datatypen till variabeln används för att informera kompilatorn om minnesutrymmets storlek, om sättet (algoritmen) att digitalisera data och för att definiera de operationer som får utföras med variabelns värde. I C# inkluderar definitionen deklarationen av variabler så att de sammanfaller i en och samma sats. När vi kommer till objektorienterad programmering blir det relevant att skilja mellan *deklaration av en klass* som inte reserverar minne och *definition av ett objekt* som reserverar minne. Det gäller att använda en konsistent terminologi som man inte behöver revidera.

Vad händer när en variabel definieras?

Genom att besvara denna fråga kommer vi att förstå *varför* man i C# *måste* definiera variabler. Vad händer när t.ex. variabeln **`number1`** definieras till **`int`**?

1. En minnescell reserveras i datorns RAM-minne för lagring av **`int`**-värden. Namnet på minnescellen blir **`number1`**. Storleken på minnescellen bestäms av datatypen **`int`** som i vår C# installation är föreskriven till 4 bytes dvs 4 x 8 = 32 bitar. (En bit kan lagra *en* 0 eller *en* 1). Detta sker vid kompileringen och upprätthålls tills exekveringen är avslutad. Detta kallas *statisk minnesallokering* i den bemärkelse att den inte kan ändras under exekveringen. *Allokering* är bara ett annat ord för reservering. Faktiskt är det *definitionen* som allokerar minne. Följande figur visar förenklat vad som sker i datorns RAM när 4 bytes minne reserveras för variabeln **`number1`**:



2. Specificieringen av datatypen gör att programmet kan tolka innehållet i minnescellen ovan när den fylls med ett värde. Det är 32 ettor och nollor som måste tolkas som ett heltal. Olika datatyper har olika algoritmer för omvandling av data till ettor och nollor och omvänt. Decimaltal omvandlas på ett annat sätt än heltal eller tecken osv. Datatypen är avgörande: *Heltalet* **`1`** t.ex. består av en annan följd av ettor och nollor än *decimaltalet* **`1.0`**. En annan digital sekvens har *tecknet* **`'1'`** för att inte tala om *strängen* **`"1"`**.

3. Namngivning har med *adressering* att göra. Minnescellens *fysiska* adress i RAM kopplas till det *logiska* namn **`number1`** vi valt i koden för att kunna komma åt minnescellen genom att referera till variabelnamnet. Med andra ord, variabler gör minnescellerna i RAM *adresserbara* och därmed åtkomliga via programmet.

Dessa tre punkter borde man ha klart för sig när man använder variabler. De förklarar också varför C# som de flesta programmeringsspråken, är strikt typbestämta språk och varför vi måste definiera alla variabler *innan* vi använder dem.

> Variabler som inte definieras innan de används ger kompileringsfel.

Det finns ingen regel som säger att alla variabeldeklarationer måste stå i början av programmet. Man kan definiera sina variabler när de behövs, bara man gör det *innan* man ger variabeln ett värde, sätter in den i ett aritmetiskt uttryck eller använder den på ett annat sätt. Att vi ändå placerar variabeldeklarationerna i början av våra program har ofta att göra med strukturering, läslighet och god programmeringsstil.

4.3 Inläsning av data

Våra C# program har hittills bara haft utdata, inga indata. Det var *utdata* som skrevs ut från programmet till bildskärmen, närmare bestämt med metoden **`WriteLine()`** till konsolen. Men hur gör man när man vill skicka *indata* till ett program? Följande program visar hur man kan göra det med metoden **`ReadLine()`**:

```
/* Input.cs
   Programmet för en dialog med användaren, läser in text med
   ReadLine() som sedan skrivs ut. Inläsningen föregås av en
   ledtext för att instruera användaren. ReadLine() är en me-
   tod definierad i klassen Console och returnerar den inma-
   tade strängen som lagras i variabler av typ string.
*/
using System;

class Input
{
   static void Main()
   {
      string name, course;                // Datatypen string

      Console.Write("\n\tVad heter du?\t\t"); // Ledtext
      name = Console.ReadLine();          // 1:a inläsning

      Console.Write("\n\tHej på dig, " + name + ','  +
                    "\n\tvilken kurs läser du?   ");
      course = Console.ReadLine();        // 2:a inläsning

      Console.WriteLine("\n\tVälkommen till " + course +
                                              "-kursen!\n");
   }
}
```

Programmet ovan producerar en dialog i två delar. Den första frågar efter **`name`**, läser in det och ger svar, efter att användaren matat in ett namn och tryckt på Enter. Den andra delen gör samma sak med inläsning av **`course`**:

```
        Vad heter du?           Peter

        Hej på dig, Peter,
        vilken kurs läser du?   C#

        Välkommen till C#-kursen!
```

Data som matas in från tangentbordet eller läses in från filer, är indata. Till skillnad från utdata som inte behöver mellanlagras, måste indata lagras i minnet. Hur man får indata in i datorn visar bilden på sid 37: Både indata och programkod måste lagras i RAM-minnet. Programkoden laddas från hårddisken till RAM-minnet när maskinkoden i den exekverbara filen körs. Indata däremot måste matas in under

programmkörning och mellanlagras i en minnescell i RAM-minnet innan den kan vidarebearbetas av programmet. Mjukvarumässigt innebär detta att indata måste tas emot och lagras i en variabel – ytterligare ett skäl till att variabeln måste vara definierad, dvs vara associerad med en minnescell av en viss storlek som är reserverad i datorns RAM-minne. Variabelns namn blir en referens till minnesadressen som sedan kan användas för att komma åt data. Medan allokeringen av minnesutrymme i regel sker under kompilering via variabeldefinition, måste inmatningen göras under exekveringen. Därför avbryts exekveringen när en inmatning ska ske. I koden förorsakas detta temporära avbrott av anropet av metoden **`ReadLine()`** som vi ska nu förklara närmare.

Metoden `Console.ReadLine()`

Vad metoden gör kan vi se när programmet **`Input`** exekveras: Första gången anropas metoden i satsen

```
name = Console.ReadLine();
```

Anropet sker med punktnotation eftersom metoden **`ReadLine()`** är definierad i klassen **`Console`**. Men varför bakas metodens anrop in i en tilldelningssats: **`name = ...`** ? Så är det inte med utskriftsmetoden **`WriteLine()`**. Dess anrop står fritt i en självständig sats (jfr. sid 42). Detta beror på att **`WriteLine()`** läser in data som måste lagras för att vidarebearbetas. Denna lagring görs i en variabel, i exemplet ovan i variabeln **`name`** som tar emot och lagrar den inmatade texten. Vi har i **`ReadLine()`** för första gången att göra med en metod som returnerar ett värde, det s.k. *returvärdet*. **`ReadLine()`** är en metod *med* returvärde. Sådana metoder kan man jämföra med en låda i vilken man stoppar in parametrar och får ut ett returvärde:

`ReadLine()` har ingen parameter och returnerar en sträng, nämligen den av användaren inmatade texten. Denna sträng hamnar i variabeln **`name`** när användaren trycker på Enter. Därför står anropet i en tilldelningssats, just för att ta hand om den returnerade strängen (returvärdet). Att en sträng dvs vanlig text kallas här för retur*värde* är inte något anmärkningsvärt. All form av data betecknas som *värde* som lagras i form av en sekvens av ettor och nollor i en minnescell.

För ett korrekt anrop av en fördefinierad metod är det dessutom avgörande att veta vilka datatyper metodens parametrar och returvärde har. Dessa är nämligen också fördefinierade och kan inte väljas fritt. Vi måste deklarera variabeln som lagrar returvärdet med just den datatyp som metoden föreskriver för sitt returvärde. Faktum är att returvärdet till **`ReadLine()`** är av datatypen **`string`**. Alltså, för att lagra returvärdet i variabeln **`name`** och sedan **`course`** måste dessa variabler deklareras till datatypen **`string`**.

Mer om metoder kommer du att lära dig i kap 7, sid 157.

4.4 Överskrivning eller kan `x = x + 1` vara sant?

I C# har tilldelningsoperatorn (**=**) följande betydelse:

```
variabel    =   värde ;
variabel  <---  värde
```

T.ex. i satsen `a = 9;` tilldelas variabeln `a` värdet `9`. Eller i satsen `sum = r + t;` där `r` och `t` måste ha väl definierade värden, bildas först summan `r + t` som sedan tilldelas till variabeln `sum`. Den gemensamma strukturen hos båda satser är att de tilldelade variablerna `a` och `sum` *inte* förekommer på båda sidor av **=** utan *endast* till vänster av den. Nu ska vi studera en annan struktur där den tilldelade variabeln finns på *båda* sidor av **=** :

```
x      =    x + 1 ;
x   <----   x + 1
```

Här måste variabeln `x` till höger redan ha ett väl definierat värde. Det som denna sats gör är att *ändra* variabeln `x`:s värde – en grundläggande teknik inom programmering som kallas för *överskrivning* och demonstreras i följande program:

```
// Overwrite.cs
// Demontrerar skillnaden mellan likhet och tilldelning
using System;

class Overwrite
{
  static void Main()
  {
    int x;
    string xAsText;

    Console.Write("\n\tMata in ett heltal:");
    xAsText = Console.ReadLine();       // Inläsning

    x = int.Parse(xAsText);             // Omvandling till int

    Console.Write("\nDet inmatade talet " + x);

    x = x + 1;                          // Överskrivning
    // x++;
    Console.WriteLine(" har ökats med 1 och är nu " + x +
                                                    "\n");
   }
}
```

Så här kan en dialog se ut:

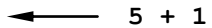
```
        Mata in ett heltal:        5

Det inmatade talet 5 har ökats med 1 och är nu 6
```

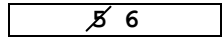
I programmet ovan läses det inmatade värdet **5** in av **ReadLine()** och returneras som en sträng, tilldelas strängvariabeln **xAsText**, omvandlas till heltal av metoden **int.Parse()** och tilldelas heltalsvariabeln **x**. Låt oss återkomma till denna typomvandlingsmetod och utskriftstekniken lite senare och titta först på progammets centrala sats:

```
x = x + 1;
```

som gör att **x**-värdet ökar till **6** och överskriver det gamla inmatade värdet **5**. Har **x** värdet **5** *före* denna sats, innebär satsen att **5** ska adderas med **1** och att det nybildade värdet **6** ska tilldelas variabeln **x** på nytt dvs:



Flöjaktligen har **x** värdet **6** *efter* satsen. Det nya värdet **6** *skriver över* det gamla värdet:



Detta kallas *överskrivning* av variabelvärdet och baseras på egenskapen av variabeln **x** som platshållare vars värde kan ändras medan namnet bibehålls (sid 59).

Vi har i satsen ovan att göra med två olika värden till en och samma variabel **x**, 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. Det beror på likhetstecknets (=) betydelse som *tilldelning* till skillnad från dess matematiska betydelse som *likhet*. Matematiskt är det fel att skriva $x = x + 1$, en ekvation som leder till motsägelsen $0 = 1$. I programmeringen däremot är det helt OK att skriva så, eftersom det inte handlar om en ekvation, utan snarare om en *instruktion* om att ge variabeln **x**) ett nytt värde genom att öka det gamla värdet med **1**. I matematiken borde detta formuleras med *två* variabler: $x_{\text{nytt}} = x_{\text{gammalt}} + 1$. I programmeringen däremot är **x** endast *en* variabel – vars värde byts ut medan namnet bibehålls. Därför används i satsen **x = x + 1;** *en* variabel på båda sidor av tilldelningstecknet och inte två.

I själva verket handlar det om den klassiska, filosofiska 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 inte heller efteråt, för redan i nästa sats kan ju **x** tilldelas ett annat värde. Man kan man säga: Tilldelning är likhet relaterad till tiden, därför är den *dynamisk*.

Att satsen **x = x + 1;** utför additionen *först* och tilldelnigen *sedan* beror på att operatorn **+** binder starkare dvs har en högre prioritet än tilldelningsoperatorn **=** . Därför slipper vi att skriva parenteser: **x = (x + 1);** vilket vi hade varit tvungna att göra om **=** hade samma prioritet som **+** eller högre.

I programmet **Overwrite** (sid 68) kan man ersätta satsen **x = x + 1;** med satsen **x++;** som är bortkommenterad. De gör samma sak: att öka **x** med **1**. Testa gärna genom att aktivera satsen **x++;** och kommentera bort **x = x + 1;** Symbolen **++** (OBS! Utan mellanslag) kallas *ökningsoperatorn* och kommer att behandlas senare (sid 77). Den härstammar från C++ och har en gång gett namnet till språket C++: Tillägget ++ ska antyda att man har lagt till 1 utvecklingssteg till C och därigenom fått fram C++.

Metoden `int.Parse()`

Har man vid exekveringen av programmet **Overwrite** matat in t.ex. **5**, lagras detta värde nu i variabeln **xAsText**. Men pga denna variabels datatyp lagras **5** som sträng, inte som heltal. Så vi kan inte räkna med det, vi kan inte addera eller multiplicera det med ett annat heltal. För att kunna göra det måste vi omvandla det till en **int**. Just detta gör den fördefinierade metoden **int.Parse()** åt oss. Den tar emot i sin parentes en parameter som är av typ **String**, omvandlar den till heltal och returnerar den som en **int**.

Satsen

```
x = int.Parse(xAsText);
```

utför denna omvandling och lagrar resultatet i variabeln **x** som är deklarerad som **int**. Även här är anropet av metoden **int.Parse()** inbakat i en tilldelningssats för att ta hand om metodens returvärde, i det här fallet **5** som heltal.

4.5 Operatorer och uttryck

De fyra grundräknesätten **+**, **–** , *****, **/** är exempel på *aritmetiska operatorer*. Symbolen ***** står för multiplikation och **/** för division. De objekt som en operator tillämpas på, kallas *operander*. I uttrycket **a + b – 4** t.ex. är **a**, **b** och **4** operander. Ett *uttryck* är en kombination av variabler, konstanter, operatorer och vanliga parenteser som till slut, när uttrycket beräknas, returnerar ett värde. På så sätt definierar ett uttryck en *föreskrift* för beräkning av ett värde. När värdet är ett *tal*, pratar man om *aritmetiska uttryck* eller räkneuttryck. Exempel på aritmetiska uttryck är:

```
no1 * no2
a + 6*b – 4/(c+1)
```

Ett annat exempel på ett aritmetiskt uttryck visas i följande program:

```
// Operator.cs
// Läser in tiden i år, månader och veckor, omvandlar den
// till dagar med ett aritmetiskt uttryck och skriver ut dem
// Strukturen Inmatning - bearbetning – utmatning
using System;

class Operator
{
   static void Main()
   {
      int  year, months, weeks, days, totalDays;

      /* I n m a t n i n g */
      Console.Write("\n\tAnge antal år:\t\t");   // Ledtext
      year = int.Parse(Console.ReadLine());      // Inläsning

      Console.Write("\n\tAnge antal månader:\t");
      months = int.Parse(Console.ReadLine());

      Console.Write("\n\tAnge antal veckor:\t");
      weeks = int.Parse(Console.ReadLine());

      Console.Write("\n\tAnge antal dagar:\t");
      days = int.Parse(Console.ReadLine());

      /* B e a r b e t n i n g */          // Aritm. uttryck
      totalDays = 365*year + 30*months + 7*weeks + days;

      /* U t m a t n i n g */
      Console.WriteLine("\n     "     +
             year  + " år, "        + months + " månader, " +
             weeks + " veckor och " + days   + " dagar är " +
                                totalDays + " dagar totalt.\n");
   }
}
```

En körning av programmet **Operator** ger följande dialog:

```
        Ange antal år:          2
        Ange antal månader:     11
        Ange antal veckor:      3
        Ange antal dagar:       6
2 år, 11 månader, 3 veckor och 6 dagar är 1087 dagar totalt.
```

Det aritmetiska uttryck som åstadkommer resultatet i ovanstående program, är:

```
365*year + 30*months + 7*weeks + days
```

Uttrycket använder sig av de aritmetiska operatorerna **+** och *****. Inmatningen i körexemplet ovan leder till följande beräkning enligt uttryckets föreskrift:

```
365*2 + 30*11 + 7*3 + 6
```

Om **year** är antalet år, **months** antalet månader, **weeks** antalet veckor och **days** antalet dagar, beräknas här det totala antalet dagar. Här har man förutsatt att ett år har 365 dagar, en månad 30 och en vecka 7 dagar. Ingen hänsyn har tagits till skottår osv. Att ***** görs först och **+** sedan beror på att i C# multiplikationsoperatorn ***** – precis som i matematiken – har en högre prioritet än additionsoperatorn **+**. Därför behövs inga parenteser.

Inmatning – Bearbetning – Utmatning

Vid sidan om aritmetiska uttryck, introducerar programmet **Operator** ett koncept inom programmering som kan bidra till att uppfylla de krav på förståelighet, användarvänlighet, strukturering och ändringsbarhet som vi ställde upp för god programmeringsstil (sid 48). Det handlar om strukturering av programkod.

Det enklast tänkbara sättet att strukturera ett program är att dela in det i de tre naturliga stegen *inmatning – bearbetning – utmatning* som man kanske helt spontant tar när man utvecklar ett program. I programmet **Operator** *matas in* först indata: år, månader och veckor. Sedan *bearbetas* dessa data genom att beräkna det totala antalet dagar och lagra resultatet i en ny variabel. Slutligen *matas ut* bearbetningens resultat genom att skriva ut den nya variabelns värde. Man borde hålla sig till denna ordning om man inte har någon speciell anledning att avvika från den. Det finns i regel ingen anledning att t.ex. splittra utmatningen och skriva en del av den före och en annan del efter bearbetningen. Inte minst när koden växer rekommenderas att utnyttja åtminstone denna naturligt givna struktur i sina program.

Strukturingen *inmatning – bearbetning – utmatning* kommer vi att ha stor nytta av när vi modulariserar våra program dvs skriver metoder. Då kommer vi nämligen att separera dessa tre delar, skriva dem i var sin metod och sedan anropa dem från **main()**. Därför kan det vara bra att vänja sig vid denna goda sed redan nu. I fort-

sättningen kommer vi att hålla oss i våra programexempel till konventionen att i regel strukturera programkoden i dessa tre delar utan att explicit nämna det.

Nästlat anrop av metoder

Ett nästlat anrop är ett anrop i ett annat anrop. Låt oss ta som exempel den sats som står i programmet **`Operator`** för att läsa in ett heltal:

```
int.Parse(Console.ReadLine())
```

Här anropas metoden **`ReadLine()`** i anropet av metoden **`int.Parse()`**, närmare bestämt i dess parameterlista dvs på en plats där en parameter förväntas. Därmed kommer **`ReadLine()`**:s returvärde, dvs den inlästa strängen, att skickas till **`int.Parse()`**. Det nästlade anropet ovan är helt identiskt med följande två anrop:

```
string yearAsText = Console.ReadLine();
int.Parse(yearAsText);
```

Som man redan ser behöver man *två* satser för att åstadkomma samma sak. Dessutom behövs en variabel (**`yearAsText`**) som måste definieras. Vid nästlingen slipper man detta merarbete. Anledningen till att två anrop är nödvändiga, är att man med metoden **`ReadLine()`** inte kan läsa in heltal eftersom dess fördefinierade returvärde är en **`string`**. Vi måste sedan konvertera till **`int`** med **`int.Parse()`**.

Nästlade anrop av metoder är mycket vanliga inom programmering därför att man spar kod. Självklart går det på bekostnad av läsligheten vilket gör att man måste avväga rimligheten. I vårt fall är det försvarbart med tanke på att vi i programmet **`Operator`** måste läsa in tre heltal så att både inläsningen och omvandlingen till heltal måste ske tre gånger. Man spar alltså här en hel del kod. Använder man sig av nästlade anrop måste följande regel beaktas:

> Nästlade anrop av metoder exekveras inifrån.

Detta innebär i vårt exempel ovan att **`ReadLine()`** anropas först och **`int.Parse()`** sedan. Dvs inläsningen görs först och konverteringen till **`int`** sedan vilket har betydelse för i vilken ordning vi skriver koden. En annan praktisk detalj är att man måste hålla ordning på parenteserna. Just nu när vi har två anrop är det inte så farligt, men flera nivåer av nästling är både tänkbara och möjliga. Dock, som sagt, sätter avvägningen mot läslighet gränser på antalet nivåer.

4.6 Överlagring av operatorer

Programmet **`Operator`** i förra avsnitt använde sig av aritmetiska operatorer och ett enkelt uttryck för att omvandla inmatad tid i antal år, månader, veckor och dagar, till antal dagar (sid 71). I körexemplet matade vi in **`2`** år, **`11`** månader, **`3`** veckor, **`6`** dagar och fick **`1087`** dagar. Lite svårare är det att lösa det omvända problemet dvs att dela upp ett inmatat antal dagar i antal år, månader och veckor, t.ex. att mata in **`1087`** dagar och få ut av programmet uppdelningen i **`2`** år, **`11`** månader, **`3`** veckor och **`6`** dagar. Nyckeln till lösningen är två nya aritmetiska operatorer – *heltalsdivision* och *modulooperatorn* – varav den första är en s.k. *överlagrad* variant av den vanliga divisionsoperatorn och den andra besläktad med den första. Behandlingen av det omvända problemet introducerar oss till ett viktigt koncept i programmering som tillämpas här på operatorer, men kann även generaliseras till metoder.

Överlagring

När två eller flera operatorer betecknas med samma symbol, men ändå betyda olikakallas det *överlagring av operatorerb*. Vi har redan sett ett exempel på det när vi gick igenom konkatenering: Symbolen **`+`** betyder både addition *och* konkatenering (sid 51). Det är sammanhanget där symbolen används, som bestämmer symbolens *aktuella* betydelse. Ett annat exempel på att operatorer kan vara överlagrade är slashtecknet **`/`** som används en gång som symbol för *heltalsdivision*, en annan gång för vanlig division. Vi har i den överlagrade operatorn **`/`** att göra med en ny typ av division. För att förstå det bättre, ska vi lösa

Det omvända problemet

Vänd på problemet i programmet **`Operator`** (sid 71)**.** Dvs Omvandla en tid som är angiven i dagar till år, månader, veckor samt resterande dagar. Skriv ett program, som frågar efter en tid i antal dagar, läser in den, och sedan beräknar samt skriver ut resultatet i antal år, månader, veckor samt resterande dagar. I själva verket handlar det om en omvandling av det decimala systemet till kalenderns system med år, månader, veckor och dagar. För denna omvandling används följande algoritm:

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”* kallas i fortsättningen för *heltalsdividera* och operationen *”Ta resten vid heltalsdivision”* för *modulo*. Algoritmen ovan skrivs nu som pseudokod:

Pseudokoden

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

Programmet **`OverloadOp`** implementerar ovanstående algoritm och pseudokod. I C# är **`/`** operatorn för *heltalsdivision*, om båda operander är heltal. Symbolen **`%`** är operatorn för *modulo*. Pseudokoden ovan återfinns översatt till C# kod i bearbetningsdelen av programmet **`OverloadOp`**.

Heltalsdivision

Det finns två olika typer av division, vanlig division och heltalsdivision. Vanlig division räknar med decimaltal, heltalsdivision bara med heltal. I C# är slashtecknet **`/`** symbolen för *båda*. Vilken av dem som ska gälla i en aktuell situation, avgörs på följande sätt av sammanhanget där **`/`** används: Om heltal finns på bägge sidor av symbolen **`/`** utförs heltalsdivision. Finns heltal på den ena sidan av tecknet **`/`** men decimaltal på den andra, utförs vanlig division. C#-koden **`9/2`** ger *inte* **`4.5`** utan **`4`** vilket beror på att **`9`** och **`2`** båda är heltal. Vill man få den vanliga divisionens resultat **`4.5`** måste man i C# skriva **`9.0/2`** eller **`9/2.0`** eller **`9.0/2.0`** dvs minst en operand måste vara decimaltal. Heltalsdivision däremot, dvs **`9/2`**, trunkerar (klipper av) alla decimaler och returnerar endast heltal – och detta utan någon avrundning. Man dividerar utan att gå vidare till decimaler. Man kan också säga: Divisionens heltalsdel tas, resten ignoreras: **`9`** dividerat med **`2`** ger **`4`** och resten **`1`**. Men resten ignoreras vid heltalsdivision: **`9`** *heltalsdividerat* med **`2`** ger därför **`4`**. En annan aritmetisk operation tar hand om resten som heter *modulo* och är besläktad med heltalsdivision. Vi kommer att använda båda i vårt nästa program.

Modulooperatorn %

`%` har i C# ingenting med procenträkning att göra utan är symbolen för ett räknesätt som kallas *modulo* och innebär *resten vid heltalsdivision*. Man dividerar två heltal utan att gå vidare till decimaler, tar resten och ignorerar resultatet. T.ex. **`16 % 5`** ger **`1`**, därför att **`16`** heltalsdividerat med **`5`** ger **`3`** och en rest på **`1`** blir kvar. Modulooperatorn **`%`** ignorerar **`3`** och returnerar resten **`1`**. Resten vid heltalsdivision kallas *modulo*: **`9`** *modulo* **`2`** ger **`1`**. Man kan uppfatta räknesättet *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`** helt enkelt att man bortser från alla multipler av heltalet **`a`**, att man kastar bort alla multipler av **`a`** och behåller resten. Räknesättet modulo har många tillämpningar, speciellt vid övergång mellan två talsystem, t.ex. mellan det decimala och binära talsystemet. En rolig användning av denna räkneoperation är följande exempel:

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

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

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

Dvs veckodagen i frågan är 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 – till veckodagarnas system dvs till talsystemet med basen **7** och siffrorna 0-6.

Programmet

```
// OverloadOp.cs
// Omvandlar dagar till år, månader, veckor och restdagar
// Överlagring av operatorn / som heltalsdivision
// Modulooperatorn %
using System;

class OverloadOp
{
   static void Main()
   {
      int year, months, weeks, restDays, totalDays;

      /* I n m a t n i n g */
      Console.Write("\n\tAnge antal dagar:\t\t");
      totalDays = int.Parse(Console.ReadLine());

      /* B e a r b e t n i n g */
      year     =   totalDays / 365;
      months   =  (totalDays % 365) / 30;
      weeks    = ((totalDays % 365) % 30) / 7;
      restDays = ((totalDays % 365) % 30) % 7;

      /* U t m a t n i n g */
      Console.WriteLine("\n\t" + totalDays + " dagar är "   +
          year + " år, " + months    + " månader, " + weeks +
                    " veckor och " + restDays + " dagar.\n");
   }
}
```

En körning av programmet **OverloadOp** ger följande dialog:

```
        Ange antal dagar:               1087

        1087 dagar är 2 år, 11 månader, 3 veckor och 6 dagar.
```

4.7 Ökningsoperatorn ++

Denna operator härstammar från programmeringsspråket C++ där den gett namnet till språket. Det finns två varianter av ökningsoperatorn: Man kan skriva den *före* operanden, så här **++a**, eller *efter* operanden, så här **a++**. Sätts den *efter* operanden talar man om ökningsoperatorns *postfix*variant. Skrivs den *före* operanden blir det *prefix*varianten. Följande program demonstrerar skillnaden mellan dessa två:

```
// Increment.cs
// Skillnaden mellan a++ och ++a
using System;

class Increment
{
   static void Main()
   {
      int a, b;

      a = 0;
      b = a++;
   // Samma som:
   // b = a;
   // a = a + 1;

      Console.Write("\n\t a = 0:  Efter b = a++; blir b = " +
                                  b + " och a = " + a + '\n');
      a = 0;
      b = ++a;
   // Samma som:
   // a = a + 1;
   // b = a;

      Console.WriteLine("\t a = 0:  Efter b = ++a; blir " +
                      "b = " + b + " och a = " + a + '\n');
   }
}
```

En körning av programmet **Increment** resulterar i följande utskrift:

```
        a = 0:  Efter b = a++; blir b = 0 och a = 1
        a = 0:  Efter b = ++a; blir b = 1 och a = 1
```

Här jämförs **a** och **b**:s värden efter **b = a++** och efter **b = ++a** med varandra. Anmärkningsvärt i båda fallen är att det inte blir någon skillnad mellan ökningsoperatorns post- och prefixvariant när det gäller själva operanden **a** som **++** tillämpas på. I båda fallen ökar operanden **a**:s värde med **1**. Efteråt är **a = 1** i båda fall. Skillnaden påverkar snarare miljön dvs det som finns kring ökningsoperatorn, i vårt exempel variabeln **b**: Efter postfixvarianten är **b = 0** medan efter prefixvarianten är **b = 1**. För att förklara varför måste vi precisera dessa två varianternas betydelse:

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;`

Nu ser man att ökningsoperatorn består av två operationer, addition *och* tilldelning. Observera att `a++;` *inte* gör samma sak som `a + 1;` I `a++;` ingår även en tilldelning medan `a + 1;` endast innehåller en addition. Ökningsoperatorn tar hänsyn till att det vid överskrivning endast finns *en* variabel vars värde överskrivs. Därför förekommer i `a++;` variabeln `a` bara en gång.

Prefixvarianten `++a;` betyder:

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

Även satsen `++a;` gör samma sak som `a = a + 1;` Skillnaden med prefixvarianten blir påtaglig först när det finns något som händer innan och/eller efteråt, dvs när sammanhanget man använder ökningsoperatorn i är lite mer komplex. Programmet **`Increment`** på förra sidan visar ett enkelt exempel på ett sådant sammanhang.

För att kunna jämföra post- och prefixvarianten och *se* deras skillnad behöver vi i **`Increment`** samma utgångssituation dvs en variabel som har samma initieringsvärde. Därför tilldelas variabeln `a` värdet `0`. Detta för att en gång använda postfixvarianten och en annan gång prefixvarianten på ett och samma värde. I det första fallet görs det i satsen `b = a++;` där tilldelningen utförs *innan* `a`:s värde ökar. Dvs *först* får `b` värdet `0`, *sedan* ökar `a` med `1` och blir `1`. I det andra fallet utförs tilldelningen i satsen `b = ++a;` *efter* att `a`:s värde ökar. Dvs *först* ökar `a` med `1` och blir `1`, *sedan* får `b` detta nyökade värde `1`.

Att det inte blir någon skillnad mellan ökningsoperatorns post- och prefixvariant på själva operanden `a` som `++` tillämpas på utan snarare på miljön dvs variabeln `b` beror på att skillnaden inte ligger i *att* operandens värde ökar med `1` utan *när* detta händer. Skillnaden ligger i saker och tings *ordning*, i vårt fall:

```
b = a++;        gör samma sak som    b = a;
                                     a = a + 1;
```

och

```
b = ++a;        gör samma sak som    a = a + 1;
                                     b = a;
```

Man kan faktiskt i programmet **`Increment`** ersätta satserna till vänster med satserna till höger och får samma resultat. Testa gärna!

Varför, kan man undra, ska man använda `a++;` eller `++a;` istället för `a = a + 1;` om de åstadkommer samma sak nämligen att öka `a`:s värde med 1? Faktum är att

ökningsoperatorn skapar maskinkod som är mycket snabbare och effektivare än maskinkod som skapas av tilldelningssatsen.

Det finns i C# även minskningsoperatorn `--` som fungerar på liknande sätt. Istället för ökning med 1 görs minskning med 1. Även minskningsoperatorn kan sättas antingen efter (postfix) eller före en variabel (prefix):

`a--;` eller `--a;` gör samma sak som `a = a - 1;`

Båda operatorer kan endast öka eller minska med 1, inte med något större värde.

4.8 Sammansatta tilldelningar

Sammansatta operatorer är dubbeloperatorer och består av *två* operatorer varav en är tilldelning: *Ökningsoperatorn* **++** t.ex. består av ökning med 1 *och* tilldelning, *minskningsoperatorn* **--** av minskning med 1 *och* tilldelning. Öknings- och minskningsoperatorn används så här: **a++** eller **a--** där **a** är en operand. De kallas för *unära* operatorer, därför att de endast tar *en* operand. En annan grupp av sammansatta operatorer är de som står mellan *två* operander (*binära*):

+= **-=** ***=** **/=** **%=**

Eftersom tilldelning förekommer hos alla dessa operatorer pratar man om *sammansatt tilldelning*. De är sammansatta av tilldelning *och* en operation till. Därför består deras symbol av två tecken. Observera att alla dessa operatorer (inkl. **++** och **--**) skrivs *utan* mellanslag. *Med* mellanslag tappar de sin speciella betydelse och känns inte igen av kompilatorn. De sammansatta operatorerna är *binära* dvs de har *två* operander, eftersom de används så här: **a += b**. Även de aritmetiska operatorerna **+**, **-**, *****, **/** och **%** är binära.

Operatorn +=

är sammansatt av de två operatorerna addition och tilldelning och betyder:

> Addera först båda sidor av **+=** med varandra och tilldela sedan resultatet till variabeln som står till vänster om **+=** .

Beräkningen utförs helt enkelt från vänster till höger, t.ex.:

sum += a; gör samma sak som **sum = sum + a;**

Dvs addera först **sum** med **a** och tilldela resultatet till **sum**. Operatorn **+=** överskriver variabeln **sum**:s värde direkt. Om a = 1 gör båda satserna ovan samma sak som **sum++;** vilket visar att ökningsoperatorn **++** är ett specialfall av **+=**. På liknande sätt fungerar de andra: **-=**, ***=**, **/=**, **%=**. De utför *först* en aritmetisk operation och *sedan* en tilldelning.

Programmet **CompAssign** på nästa sida demonstrerar alla binära sammansatta tilldelningsoperatorer. Användaren matar in ett värde till variabeln **a** via tangentbordet som kombineras via tilldelningsoperatorerna **+=**, **-=**, ***=**, **/=** med de redan initierade variablerna **sum**, **diff**, **prod** och **div**. Dessa namn utför förstås i sig inga aritmetiska operationer, utan är bara valda för att vara beskrivande. Deras initiering är avgörande, annars kan vi inte kompilera eftersom oinitierade variabler leder till kompileringsfel (sid 62).

Operatorn **+=** fungerar på samma sätt även om **+** tolkas som konkatenering när den initieras med en sträng. Sista exemplet i programmet **CompAssign** demonstrerar detta:

```
// CompAssign.cs
// Sammansatta tilldelningsoperatorer: +=, -=, *=, /= och %=
// Utför FÖRST operationen +, -, *, / och SEDAN tilldelningen
// Gäller även för konkateneringsoperatorn +
using System;

class CompAssign
{
   static void Main()
   {
      int a = 8, sum = 10, diff = 20, prod = 30, div = 40;
      string s = "Slut på", t = " kapitel 4";

      sum   += a;              // Samma som sum  = sum  + a;
      diff  -= a;              //           diff = diff - a;
      prod  *= a;              //           prod = prod * a;
      div   /= a;              //           div  = div  / a;
      s     += t;              // + här konkatenering:
                               // Samma som s    = s   +  t;
      Console.WriteLine(
        "sum  = 10 och a = " + a + ": \nEfter sum += a; "  +
                               "blir sum = "  + sum + "\n\n"   +
        "diff = 20 och a = " + a + ": \nEfter diff -= a; " +
                               "blir diff = " + diff + "\n\n"  +
        "prod = 30 och a = " + a + ": \nEfter prod *= a; " +
                               "blir prod = " + prod + "\n\n"  +
        "div  = 40 och a = " + a + ": \nEfter div /= a; "  +
                               "blir div = "  + div + "\n\n"   +
                                                s + '\n');
   }
}
```

Här ska de binära operatorerna testas genom att kombinera variabeln **a** via **+=**, **-=**, ***=** och **/=** med andra, redan initierade variabler **sum**, **diff**, **prod** och **div**. Dessa namn är förstås godtyckligt valda och har inget att göra med själva aritmetiska operationer som endast utförs pga sina resp. operatorer **+**, **-**, ***** och **/**. Vi har bara försökt att välja beskrivande namn. Satsen **sum += a;** gör exakt samma sak som satsen **sum = sum + a;** dvs adderar först **sum** med **a** och tilldelar sedan resultatet till **sum** igen. Med andra ord, variabeln **sum**:s värde överskrivs av **sum + a**. Samma sak är det med de övriga dubbeloperatorerna.

En körning av programmet **CompAssign** ger följande utskrift:

```
sum  = 10 och a = 8:
Efter sum += a; blir sum = 18

diff = 20 och a = 8:
Efter diff -= a; blir diff = 12


prod = 30 och a = 8:
Efter prod *= a; blir prod = 240

div  = 40 och a = 8:
Efter div /= a; blir div = 5

Slut på kapitel 4
```

Övningar till kapitel 4

4.1 Satsen **`Console.WriteLine(a);`** ger kompileringsfel till skillnad från **`Console.WriteLine('a');`** Sätt in båda i ett C# program och testa. Ger även **`Console.WriteLine(6);`** kompileringsfel? Testa vilka utskrifter följande satser ger:

```
Console.WriteLine(6 + 6);
Console.WriteLine('6' + '6');
Console.WriteLine("6" + "6");
Console.WriteLine(6.6 + 6.6);
Console.WriteLine("6.6" + "6.6");
```

Förklara resultaten.

4.2 Komplettera programmet **`Variable`** (sid 60) så här: Definiera ytterligare variabler, säg **`diff`**, **`prod`**, **`div`**, **`mod`**, tilldela till dem uttryck (sid 71) bildade med de andra räknesätten **`-`**, **`*`**, **`/`** och **`%`**. Skriv ut resultaten med meningsfulla utskrifter. Bibehåll ändringen av variabeln **`number1`**:s värde mellan de två utskrifterna.

4.3 Vidareutveckla din lösning till övn 4.2 genom att ersätta den hårdkodade initieringen av variablerna **`no1`** och **`no2`** med en initiering genom *inläsning* som t.ex. kan göras med en **`ReadLine()`**-sats samt ledtext. Stryk ändringen av variabeln **`no1`**:s värde.

4.4 Skriv ett program som läser in två heltal, multiplicerar dem med varandra och skriver ut resultatet blandat med förklarande text. Om du t.ex. matar in **`3`** till det första och **`4`** till det andra heltalet, ska programmet skriva ut: **`3 gånger 4 är 12.`** Utveckla programmet vidare med ytterligare räkneoperationer, kanske så småningom till en liten kalkylator.

4.5 Ersätt i programmet **`DefInit`** (sid 63) de två satser som definierar och initierar variablerna **`number1`**, **`number2`** med satsen **`int number1 = 9, number2 = 2;`**.

4.6 Modifiera programmet **`Overwrite`** (sid 68) så att variabeln **`x`**:s gamla värde skrivs ut, medan dess nya ökade värde visas senare. Ersätt satsen **`x = x + 1;`** med **`x++;`** Blir det samma resultat om du ersätter den med **`x + 1;`** istället? Förklara!

4.7 Skriv ett program som läser in tre heltal till timmar, minuter och sekunder. Beräkna och skriv ut sedan hur många sekunder det blir totalt. Gör utskriften användarvänlig.

4.8 Varför ger följande program kompileringsfel? Åtgärda felet!

```
using System;

class ExInit
{
   static void Main()
   {
      int a, sum;

      Console.Write("Mata in ett heltal:\t");
      a = int.Parse(Console.ReadLine());

      sum += a;

      Console.WriteLine("sum = "  + sum + "\n\n");
   }
}
```

4.9 Vänd på problemet från övn 4.7: Skriv ett program som läser in ett antal totalsekunder, omvandlar det till antal timmar, minuter och sekunder och skriver ut resultatet. Använd för denna omvandling följande algoritm:

timmar = totalsekunder DIV 3600
minuter = (totalsekunder MOD 3600) DIV 60
sekunder = ((totalsekunder MOD 3600) MOD 60) MOD 60

där DIV betyder heltalsdivision och MOD modulooperation. Om dessa två aritmetiska operationer läs på sid 75.

4.10 Skriv ett program som modifierar algoritmen ovan för att omvandla ett givet belopp i ören till 10-kronor, 5-kronor, 1-kronor och 50-öringar. Programmet ska läsa in ett givet belopp i ören som kan vara växeln efter inköp av en vara i en automat. Sedan ska programmet skriva ut antalet 10-kronor, 5-kronor, 1-kronor och 50-öringar som automaten ska spotta ut. Om 50-öringar läs fotnoten på sidan 193.

Kapitel 5

Enkla

datatyper

5.1 Kan datorn lagra hur stora tal som helst?

Man har ju hört talas om datorernas obegränsade möjligheter. Deras kraft och kapacitet växer med teknikens snabba utveckling. Men hur avancerade de än blir kommer de alltid att ha ett *begränsat* utrymme för lagring av data. Därför har man i alla programmeringsspråk vissa fördefinierade datatyper för att bl.a. ekonomisera och effektivisera minneshanteringen när ett program körs. Datatyperna **`int`** och **`string`** har vi redan stiftat bekantskap med. I detta kapitel ska vi studera kategorin *enkla datatyper* i C# (eng. *primitive types*). Andra kategorier är *sammansatta* datatyper och *objekt* som behandlas senare. I C# finns 13 fördefinierade datatyper av kategorin *enkel*: **`bool`**, **`sbyte`**, **`byte`**, **`char`**, **`short`**, **`ushort`**, **`int`**, **`uint`**, **`long`**, **`ulong`**, **`float`**, **`double`** och **`decimal`**. De kallas *enkla* datatyper därför att de representerar endast *ett* värde, dvs *ett* heltal, *ett* decimaltal, *ett* tecken eller *ett* sanningsvärde. De kan inte lagra mer invecklade data som *objekt*. De är inte heller *sammanatta* som t.ex. datatypen **`string`** som lagrar *flera* tecken och därför inte är en enkel datatyp.

I definitionen av *datatyp* (sid 58) sa vi att en datatyp var en föreskrift om bl.a.

> ” hur mycket minne den tar och därmed hur stora värden
> den kan lagra (dvs det tillåtna värdeområdet) … ”

Det tilldelade minnesutrymmet är förbestämt i datatypens definition. De enkla datatypernas fastlagda minnesstorlekar kan man se i följande utskrift som är producerad av programmet **`Primitives`** på nästa sida:

```
C:\WINDOWS\system32\cmd.exe
De enkla datatyperna i C#:
--------------------------
Datatypen bool    tar  1
          sbyte        1
          byte         1
          char         2
          short        2
          ushort       2
          int          4
          uint         4
          long         8
          ulong        8
          float        4
          double       8
          decimal     16 bytes
```

`bool` representerar sanningsvärdena sant eller falskt. **`char`** lagrar tecken. **`sbyte`**, **`byte`**, **`short`**, **`ushort`**, **`int`**, **`uint`**, **`long`**, **`ulong`** är enkla datatyper för representation av heltal. Prefixet **`u`** som inleder några av dem betyder **`unsigned`** och innebär att dessa endast kan lagra positiva heltal, medan prefixet **`s`** står för **`signed`** som tillåter även negativa heltal. De enkla datatyperna **`float`**, **`double`**, **`decimal`** representerar decimaltal. Alla enkla datatyper hittar man i tabellen över reserverade ord (sid 36). I denna tabell finns också det reserverade ordet **`sizeof`** som används för att mäta minnesstorleken av varje datatyp i antal bytes. 1 byte består av 8 bitar där 1 bit är den minnesatom som kan lagra endast en nolla eller en etta. Som man ser har vi ordnat de enkla datatyperna efter det minnesutrymme som är tilldelat och förbestämt i deras definition. Det tillåtna värdeområdet ligger inom ett intervall som direkt kan härledas från minnesstorleken som varje datatyp har till förfogande.

Egentligen är programmet **Primitives** ur programmeringsteknisk synpunkt inte särskilt intressant och består av en enda utskriftssats. Vi återger den ändå, inte minst för att visa hur man använder operatorn **sizeof**:

```
// Primitives.cs
// Visar alla enkla datatyper i C# och deras minnesstorlekar
// Operatorn sizeof mäter minnesstorleken i antal bytes
using System;

class Primitives
{
  static void Main()
  {
    Console.WriteLine("De enkla datatyperna i C#:\n"  +
                      "--------------------------\n"  +
    "Datatypen bool   tar  " + sizeof(bool)    + '\n' +
    "          sbyte       " + sizeof(sbyte)   + '\n' +
    "          byte        " + sizeof(byte)    + '\n' +
    "          char        " + sizeof(char)    + '\n' +
    "          short       " + sizeof(short)   + '\n' +
    "          ushort      " + sizeof(ushort)  + '\n' +
    "          int         " + sizeof(int)     + '\n' +
    "          uint        " + sizeof(uint)    + '\n' +
    "          long        " + sizeof(long)    + '\n' +
    "          ulong       " + sizeof(ulong)   + '\n' +
    "          float       " + sizeof(float)   + '\n' +
    "          double      " + sizeof(double)  + '\n' +
    "          decimal    "  + sizeof(decimal) + " bytes\n");
  }
}
```

Operatorn `sizeof`

sizeof är en operator i C# som endast har *en* operand och mäter storleken på operandens minnesutrymme. Närmare bestämt returnerar **sizeof** antalet bytes operanden tar i minnesutrymme. Operanden som står i parentes kan vara en datatyp:

sizeof(*datatyp***)**

sizeof returnerar minnesstorleken i antalet bytes för datatypen. C#:s egenskap som ett plattformsoberoende språk medför att de enkla datatypernas minnesstorlekar och därmed deras gränser är enhetligt fastlagda i C#-kompilatorn för alla plattformar (datortyp & operativsystem), vilket är av stor fördel för språkets portabilitet. Så, du kommer att få precis samma värden för de enkla datatypernas gränser på vilken dator än du kör C#, vilket t.ex. inte är fallet i C++.

De enkla datatypernas gränser

De enkla datatypernas gränser som vi egentligen är ute efter i detta avsnitt, kan nu lätt härledas från deras minnesstorlekar. Ett exempel är heltalsdatatypen **short** som enligt ovan har **2** bytes dvs 2x8 = 16 bitar till förfogande. Därför reserverar varje variabel definierad som **short** 16 bitar i minnesutrymme. Ett värde till en sådan variabel kan alltså inte lagras i datorn om det överstiger det största binära tal som kan lagras i 16 – 1 = 15 bitar. 15 därför att en bit behövs för att lagra själva tecknet + eller – därför att en **short**-variabel kan även anta negativa värden. Det största binära heltal som kan lagras i 15 bitar består av 15 ettor dvs 111 1111 1111 1111. I det decimala talsystemet blir det 32 767. Därför är den positiva gränsen för datatypen **short** 32 767. På samma sätt kan de andra datatypernas gränser härledas från deras resp. minnesutrymme. Ingen panik! Vi kommer inte att göra det. Dessa gränser är lagrade i vissa namngivna konstanter. Här skrivs ut dem för alla enkla datatyper som ett körresultat av programmet **Limits** på nästa sida:

```
Enkla datatypernas gränser:
---------------------------
sbyte  finns mellan  -128              och    127
byte                 0                        255
char                 0                        65535
short                -32768                   32767
ushort               0                        65535
int                  -2147483648              2147483647
uint                 0                        4294967295
long                 -9223372036854775808     9223372036854775807
ulong                0                        18446744073709551615
float                -3,402823E+38            3,402823E+38
double               -1,79769313486232E+308   1,79769313486232E+308

decimal              -79228162514264337593543950335 och
                      79228162514264337593543950335

bool tar endast värdena    True    och    False
```

Till skillnad från de andra datatyper som kan anta både positiva och negativa värden, kan de *teckenlösa* datatyperna (**u** = **unsigned** dvs utan tecken **+** eller **-**) endast anta positiva värden: De heter så därför att deras värden varken behöver ha plus- eller minustecknet framför talet. Dessa enkla datatyper har precis lika mycket minnesutrymme till förfogande som sina motsvarande vanliga datatyper med tecken. Detta innebär att nödvändigheten att lagra tecknet faller bort hos **unsigned**-typerna. Om vi resonerar vidare i exemplet med **short** skulle datatypen **ushort** ha alla 16 bitar till förfogande för själva positiva heltalet. Det största binära heltal som kan lagras i 16 bitar består av 16 ettor dvs 1111 1111 1111 1111. I det decimala talsystemet blir detta 65 535. Därför är gränsen för datatypen **ushort** dubbelt så stort (fast +1 pga nollan) som för **short**. Och så är det med alla **unsigned**-typer: deras gränser är dubbelt så stora fast de har lika stort minnesutrymme till förfogande, därför att de inte behöver lagra tecknet och därmed har 1 bit mer för att lagra själva positiva heltalet. Av samma anledning har **byte** en dubbelt så stor övre gräns som **sbyte** fast båda tar endast 1 byte minne. Decimaltalstyperna **float** och **double**:s gränser visas i utskriften ovan i s.k. **E**xponentiellt format, även kallat

grundpotensform (eng.: *Scientific notation*) vilket innebär att t.ex. **`float`**:s positiva gräns **`3.4028235E38`** är lika med 3,4028235 gånger 10 upphöjt till 38 dvs $3{,}4028235 \cdot 10^{38}$.

Overflow

Vad händer nu om man överskrider de ovan angivna gränser dvs om man tilldelar ett värde till en variabel som överskrider det maximalt tillåtna värdet för datatypen? Fenomenet kallas *overflow*. Ja, vad händer om man försöker att hälla mer vatten i ett glas än det ryms? I vissa miljöer blir det exekveringsfel och programkrasch. I C# fortsätter programmet: Det överskridna värdet ”slår runt” och hamnar på andra ändan av det tillåtna talområdet. Overflow innebär förlust eller förfalskning av information. Beroende på datatypen kan det bli felaktigt resultat samt följdfel som är svårt att spåra, om man räknar vidare utan att upptäcka felet. Ett exempel på overflow visas senare i programmet **`AutoConv`** (sid 106). Det enda sättet att undvika overflow är att utveckla en medvetenhet om fenomenet overflow och känna till när det kan inträffa. Programmet nedan som skriver ut de enkla datatypernas gränser (förra sida), kan hjälpa i sådana situationer. Dessutom bekantar vi oss med nya klasser som är associerade till enkla datatyper:

```
// Limits.cs
// Visar enkla datatypernas gränser som är lagrade i
// konstanter definierade i datatypklasserna
using System;

class Limits
{
  static void Main()
  {
    Console.WriteLine("Enkla datatypernas gränser:\n"       +
                      "--------------------------\n"       +
            "sbyte  finns mellan  " + sbyte.MinValue        +
            "               och    " +   sbyte.MaxValue     +
            "\nbyte                  " +    byte.MinValue    +
            "                        " +    byte.MaxValue   +
            "\nchar                  " + (int) char.MinValue +
            "                        " + (int) har.MaxValue +
            "\nshort                 "  + short.MinValue     +
            "                    "      + short.MaxValue     +
            "\nushort                " + ushort.MinValue     +
            "                        " +  ushort.MaxValue   +
            "\nint                   "  +  int.MinValue      +
            "              "            +  int.MaxValue      +
            "\nuint                  "  +  uint.MinValue     +
            "                        " +  uint.MaxValue     +
            "\nlong                  "  +  long.MinValue     +
            "     "                     +  long.MaxValue     +
```

```
            "\nulong                "  + ulong.MinValue     +
            "                        " + ulong.MaxValue     +
            "\nfloat                 "  + float.MinValue     +
            "            "             + float.MaxValue     +
            "\ndouble                "  + double.MinValue    +
            "   "                      + double.MaxValue    +
            "\n\ndecimal\t              " + decimal.MinValue +
            " och \n\t\t      "          + decimal.MaxValue +
            "\n\nbool tar endast värdena    " + true        +
            "    och    "  + false + '\n');
  }
}
```

Inte heller programmet **`Limits`** är ur programmeringsteknisk synpunkt särskilt intressant, men har fördelen att låta datorn göra jobbet och lista upp ett antal intressanta värden – de enkla datatypernas gränser. Värdena hämtas från lagrade konstanter som är definierade i klasser som är identiska med C#:s enkla datatyper.

5.2 Datatypen `char`

I våra programexempel hittills förekom endast datatyperna **`int`** och **`string`**. Nu ska vi lära oss att använda ytterligare en datatyp nämligen **`char`** som står för *character*, tecken på engelska (uttalas därför ”kar”). I en variabel av typ **`char`** kan endast *ett* tecken lagras. Därför är **`char`** en *enkel* dataytyp. Fler än ett tecken bildar en sträng (= text) som då inte längre får tilldelas en variabel av typ **`char`** utan måste lagras i en variabel av den *sammansatta* datatypen **`string`**. Datatypen **`char`** representerar alltså tecken och används i första hand för att definiera teckenvariabler. Men det är bara halva sanningen:

Hur lagrar datorn tecken? All data, även tecken, måste ju slutligen omvandlas till ettor och nollor. Därför måste alla tecken omvandlas till tal. Varje tecken har sin heltalskod enligt ett visst kodsystem, en överenskommen teckenuppsättning eller teckentabell. Det är dessa heltalskoder som omvandlas till ettor och nollor. Bokstaven a t.ex. har enligt den rådande teckentabellen koden 97 som är 1100001 binärt. Redan när man trycker på tangenten a överförs sekvensen 1100001 via tangentbordssladden till datorn. Därför representerar datatypen **`char`** även dessa *koder*. Hela sanningen är alltså att **`char`** är en datatyp som representerar *både* tecken *och* sådana heltal som är koder till tecken dvs tal som ryms i **`char`**-värdenas minnesstorlek på 2 bytes (enligt förra avsnitt). Följande program introducerar datatypen **`char`** genom att definiera en variabel av typ **`char`** och initiera den till bokstaven **`a`**. I själva verket kommer koden 97 i binär form att lagras i RAM-minnet.

```
// Char.cs
// Datatypen char för tecken och tal (mellan 0 och 65535)
// Tolkas som tecken i strängsammanhang (konkateneringar)
//            tal    i räknesammanhang  (aritmetiska uttryck)
using System;

class Char
{
  static void Main()
  {
    char letter = 'a';
//  char letter = (char) 97 ;                // Gör samma sak
    string concat = " " + letter + letter;   // Konkatenering
      int  add    =       letter + letter;   // Addition

      Console.WriteLine("\n\tEn char-variabel har "    +
        "initierats till " + letter + "\n\tTecknet "   +
        letter + " konkatenerat med tecknet " + letter +
        " ger " + concat + "\n\tMen tecknet " + letter +
        "  adderat med tecknet " + letter + " ger "    +
                                              add + "\n");
  }
}
```

En testkörning av programmet **Char** ger följande utskrift:

```
En char-variabel har initierats till a
Tecknet a konkatenerat med tecknet a ger  aa
Men tecknet a  adderat med tecknet a ger 194
```

I programmet **Char** definieras variabeln **letter**, deklareras till datatypen **char** och initieras till **a**. Observera att teckenkonstanten **a** måste i C#-kod skrivas inom apostrofer (sid 46). Annars kommer den att tolkas av kompilatorn som ett nytt variabelnamn och leda till kompileringsfel då en sådan variabel inte är definierad. Teckenkonstanten **a** lagras i teckenvariabeln **letter** i form av koden 97 (fast binärt). Därför är det även möjligt att initiera **letter** med heltalet 97 efter att ha omvandlat det till **char**. Den möjligheten är bortkommenterad i programmet ovan just nu. Men testa gärna genom att aktivera den bortkommenterade raden och kommentera bort raden ovan istället. I raden som följer konkateneras variabeln **letter** med sig själv genom att initiera konkateneringen med en tom sträng och tilldela resultatet till **String**-variabeln **concat**. Vid denna konkatenering omvandlas *tecknet* **a** automatiskt till *strängen* **a**. Därför får vi ut **aa** när den konkatenerade strängen skrivs ut med **concat**.

Teckenaritmetik

Sedan adderas **a** med sig själv utan initiering till sträng. Resultatet tilldelas **int**-variabeln **add**. Vid (och pga) additionen tolkas **char**-variabeln **letter** som tal. Därför får vi ut **194** när summan 97 + 97 skrivs ut med **add**. Att kunna addera två teckenvariabler med varandra är ett exempel på *teckenaritmetik* – att kunna ”räkna” med tecken, i själva verket med deras koder.

Avgörande för tolkningen av variabeln **letter**:s värde **a** som tecken är deklarationen till datatypen **char** och avgörande för dess omvandling till sträng är initieringen till den tomma strängen **" "** i följande tilldelnings högerled:

```
concat = " " + letter + letter;
```

För att vara mer exakt kan man säga att den tomma strängen **" "** gör att operatorn **+** tolkas som konkatenering och inte som addition. Denna tolkning gör att den efterföljande variabeln **letter**:s värde **a** omvandlas till sträng. Samma sak görs i resten av satsen. Strängen **aa** bildas och tilldelas strängvariabeln **concat**.

Avgörande för tolkningen av **char**-variabeln **letter**:s värde **a** som tal i nästa sats är att det inte görs någon initiering till sträng i följande tilldelnings högerled:

```
add = letter + letter;
```

Här tolkas operatorn **+** som addition, inte som konkatenering. Därmed tas heltalsvärdet 97 från variabeln **letter**:s minnescell och adderas med sig själv. Resultatet blir talet **194** som tilldelas **int**-variabeln **add**.

5.3 ASCII - tabellen

I förklaringen till programmet **Char** i förra avsnitt har vi hela tiden lite halvmystiskt talat om ”koder” eller ”den rådande teckentabellen” utan att säga vad det egentligen är för koder och hur de bestäms. Varför är just **97** teckenkoden till bokstaven **a**? En fördjupad insikt i datorns hantering av tecken får vi när vi tar upp datorns *teckenuppsättning*. Den vanligaste teckenuppsättningen är *ASCII-tabellen*.

ASCII (uttalas ”aski”) står för **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange och är en standard för kodning av tecken skapad av det amerikanska standardiseringsorganet ANSI. Innan man hann sätta igång något standardiseringsarbete på internationell nivå, hade ASCII redan erövrat världen. Så idag är ASCII den de facto-standard som används på alla persondatorer, både PC och Mac.

Teckenstandarden ASCII omfattar alla engelska bokstäver, siffrorna 0-9, de vanligaste specialtecknen och en del styr- och kontrolltecken. ASCII använder sig av s.k. 7-bitars kodning vilket innebär att heltalskoden till ett tecken placeras som ettor och nollor i 7 bitar av en byte. Den lediga åttonde biten används för felkontroll och kan därför inte utnyttjas för representation av data. Exempel: Tecknet **a**:s ASCII-kod är **97** vilket omvandlat till ettor och nollor blir 1100001*, dvs ett binär tal som är 7 bitar långt. För att få reda på vilka tal man kan uttrycka med 7 bitar, kan man titta på det minst möjliga talet – det är 7 nollor 0000000 som är **0** – och det störst möjliga – det är 7 ettor 1111111 som är **127** (kontrollera med kalkylatorn). Därför består ASCII-koderna av heltalen mellan **0** och **127** och är definierade enligt:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	null	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	**!**	**"**	**#**	**$**	**%**	**&**	**'**
4	**(**	**)**	*****	**+**	**,**	**-**	**.**	**/**	**0**	**1**
5	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**:**	**;**
6	**<**	**=**	**>**	**?**	**@**	**A**	**B**	**C**	**D**	**E**
7	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**
8	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**
9	**Z**	**[**	****	**]**	**^**	**_**	**`**	**a**	**b**	**c**
10	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**
11	**n**	**o**	**p**	**q**	**r**	**s**	**t**	**u**	**v**	**w**
12	**x**	**y**	**z**	**{**	**|**	**}**	**~**	del		

* För att verifiera omvandlingen till binärt tal utan att gå in på *hur* det görs, kan du använda t.ex. Kalkylatorn som du hittar bland dina appar i Windows (i tidigare versioner: Start → Program → Tillbehör). Välj i menyn Visa undermenyn Avancerad för att kunna omvandla tal till och från olika talsystem (**Bin**är: ettor och nollor, **Dec**imal: vanliga tal, **Hex**adecimal: 0-9 och A-F, **Oct**al: 0-7).

Siffrorna till vänster i tabellens 1:a kolumn anger de två första siffrorna i ASCII-koden och siffran högt upp i tabellens 1:a rad anger den sista siffran i ASCII-koden. Söker man ASCII-koden till t.ex. **`t`** går man på samma rad längst till vänster och hittar **`11`**, går sedan från **`t`** i samma kolumn högst upp och hittar **`6`**. Alltså har **`t`** ASCII-koden **`116`**. ASCII-tabellen visar att endast tecknen med koderna mellan **`33`** och **`126`** är skriv- och läsbara. De andra består av icke-skriv- och icke-läsbara styr- och kontrolltecken eller ”vita” tecken. Dessa specialtecken har symboliska namn. T.ex. kallas mellanslaget för sp = space med ASCII-koden **`32`** och radbyte har beteckningen nl = newline med ASCII-koden **`10`** (en äldre beteckning är lf = line feed). Vi kommer att använda några av dessa specialtecken senare, men en fullständig genomgång ligger utanför denna boks ramar.

ASCII-tabellen visar också denna teckenstandards begränsningar. Specialtecken i andra språk än engelskan saknas, t.ex. de svenska tecknen å, ä, ö, Å, Ä, Ö. Rent tekniskt beror denna begränsning på 7-bitars kodningen. Släpper man kravet på att använda den åttonde biten för felkontroll och utnyttjar även denna bit för representation av data, kan man utvidga ASCII-området bortom **`127`**. Det största binära heltal man kan placera i 8 bitar är 8 ettor 11111111 som ger **`255`**. Det har man gjort i andra teckenuppsättningar och skapat koder mellan **`128`** och **`255`**. Denna utvidgning är däremot inte en standard, inte ens ”de facto”. Ändå pratar man ofta lite slött om ”ASCII”-koder. I själva verket tillämpas i kodintervallet **`128–255`** olika teckenuppsättningar inte bara i olika datorer, utan t.o.m. i olika program på samma dator, t.ex. Windows och Kommandotolken. I C# används teckenstandarden *Unicode* för utvidgningen av ASCII-tabellen, även i Kommandotolken. Andra språk som t.ex. C++ och Java ”förvränger” de svenska tecknen å, ä, ö, Å, Ä, Ö när de skriver ut dem till konsolen (i Windows: Kommandotolken). Förklaringen är att de skriver till konsolen med en annan teckenuppsättning än Unicode. C# visar dock de svenska tecknen korrekt även i konsolen. När det gäller Java kan denna inkonsekvens bero på konkurrenskampen mellan Microsoft och Oracle (ägaren till Java) som tyvärr utkämpas på bekostnad av användarna.

Explicit typkonvertering

Explicit betyder uttrycklig och innebär här att man själv – utan någon fördefinierad metod – omvandlar datatypen. Generellt kan programkoden för explicit typkonvertering beskrivas så här:

`(datatyp)` *uttryck*

där *uttryck* är en kombination av variabler, konstanter, operatorer och vanliga parenteser som i specialfall även kan bestå av en enda variabel eller en enda konstant (sid 71). Det enklast tänkbara uttrycket – ett specialfall – är en konstant eller en variabel. För det mesta kommer vi i våra program att ha en variabel som uttryck.

Programmet **`Int2char`** på nästa sida använder följande exempel på explicit typkonvertering:

`(char) code`

code är en variabel deklarerad som **int**. Genom att sätta typen **char** inom parentes och placera den framför variabeln **code**, omvandlas variabelns *värde* till **char**. Dvs explicit typkonvertering förändrar inte *variabelns* datatyp, utan ändrar endast det aktuella *värdets* datatyp lokalt. När programmet körs och det matas in t.ex. **65** omvandlas **65** som är variabeln **code**:s värde, till tecknet **A**. Det är alltså *talet* **65** som omvandlas till **char**, inte *variabeln* **code**. Den fortsätter att vara av typ **int** enligt deklaration i början av programmet:

```
// Int2char.cs
// Ger tecknet till en inmatad ASCII-kod
// Representation av tecken med ASCII-koder
// Explicit typkonvertering
using System;

class Int2char
{
   static void Main()
   {
      Console.Write("\n\tMata in ett heltal:\t");
      int code = int.Parse(Console.ReadLine());

      Console.WriteLine("\n\t" +
         "Det inmatade talet " + code + " är " +
         "ASCII-koden till tecknet " + (char) code + "\n\n");
         // Explicit typkonvertering: omvandlar till char
   }
}
```

Den med vit bakgrund framhävda explicita typkonverteringen **(char) code** är inbakad i utskriftssatsen: **int**-variabeln **code**:s värde omvandlas till **char** innan det skrivs ut. Därför skrivs ut tecknet **A** tillhörande ASCII-koden 65 när vi matar in 65 till **code**:

```
        Mata in ett heltal:     65

        Det inmatade talet 65 är ASCII-koden till tecknet A
```

Men innan explicit typkonvertering händer följande i programmet **Int2char**: Siffrorna 6 och 5 läses in av metoden **ReadLine()** och returneras som en sträng då denna fördefinierade metod alltid returnerar en sträng (sid 67). Strängen i sin tur omvandlas av metoden **int.Parse()** till heltalet **65** (sid 70) som tilldelas **int**-variabeln **code**. Anropen av dessa två metoder har vi nästlat för att spara en variabel som skulle mellanlagra den returnerade strängen. Därför utförs anropen inifrån dvs först **ReadLine()** och sedan **int.Parse()** (sid 73). Slutligen omvandlar explicit typkonvertering heltalskoden **65** till **char** dvs till det tecken vars kod är **65**, näm-

ligen bokstaven **A**. Genom att köra programmet **Int2char** kan man alltså ta reda på tecken tillhörande vilken ASCII-kod som helst.

Det omvända problemet

Programmet **Int2char** gav oss tecknet när vi matade in ett heltal (ASCII-koden). Programmet **Char2int** på nästa sida löser det omvända problemet: Det skriver ut ASCII-koden till tecknet vi matar in. När vi t.ex. matar in bokstaven **A** läser metoden **ReadLine()** in den som en sträng. Metoden **Convert.ToChar()** omvandlar *strängen* **A** till ett *tecken* som tilldelas **char**-variabeln **letter**. Slutligen omvandlas tecknet **A** till ASCII-koden **65** med explicit typkonvertering och skrivs ut.

```
// Char2int.cs
// Ger ASCII-koden till ett inmatat tecken
using System;

class Char2int
{
   static void Main()
   {
      char letter;

      Console.Write("\n\tMata in ett tecken:\t");
      letter = Convert.ToChar(Console.ReadLine());

      Console.WriteLine("\n\t" +
        "Det inmatade tecknet " + letter + " har " +
        "ASCII-koden " + (int) letter + "\n\n");
   }
}
```

Programmet **Char2int** kan avslöja datorns alla ASCII-koder när man exekverar det med dialoger av typ:

```
        Mata in ett tecken:     A

        Det inmatade tecknet A har ASCII-koden 65
```

Genom att köra programmet **Char2int** kan man alltså ta reda på ASCII-koden till vilket tecken som helst. Hur däremot typomvandlingen från den inlästa strängen till *tecken* genomförs ska vi titta närmare på:

Programmet **Char2int**:s mål var ju att till sist få ut tecknets ASCII-kod. Detta sista steg tas med explicit typkonvertering. Det är generellt så att typomvandlingar inom enkla datatyper kan genomföras med explicit typkonvertering på ett mycket enklare sätt än omvandlingar från eller till sammansatta datatyper som **string**. För dem behövs i regel vissa metoder som måste anropas och som är definierade i klasser som t.ex. metoden **ToChar()** i klassen **Convert**.

5.4 Escapesekvenser

Följande frågor är fortfarande obesvarade när det gäller hantering av tecken:

1. Vilka koder gäller för de tecken som inte finns i ASCII-standarden dvs utanför kodintervallet **0–127** (sid 93)?
2. Hur kommer vi i C#-kod åt de icke-skriv- och icke-läsbara styr- och kontrolltecknen med ASCII-koderna **0–32**?
3. Hur kodar vi inom en strängkonstant "..." själva *tecknen* " eller ' (eller andra) som redan är koder för att omgärda strängar resp. tecken (eller annat)?

För att kunna besvara dessa frågor behövs kunskap om *escapesekvenser* och *Unicode*.

Escapesekvenser är kod som inleds med tecknet backslash **** åtföljt av *ett* tecken eller tecknets kod i ett visst format (Unicode). På svenska betyder *to escape* att fly. Med **** vill man *fly* från tecknets vanliga betydelse och ge det en *annan* innebörd. Med **\n** t.ex. vill man undvika bokstaven **n** och åstadkomma radbyte istället: **n***ewline*. På samma sätt fungerar andra escapesekvenser. Programmet **Escape** (nästa sida) använder denna teknik och visar i följande utskrift några symboliska och kodade escapesekvenser i Unicode-format. Med symboliska escapesekvenser menas **\n**, **\r**, **\t**, **\b** osv. De som inleds med **\u** är kodade i Unicode som kommer att behandlas i detalj inästa avsnitt.

Escapesekvens	Unicode	Tecknet	Direkt
\0	0	nolltecknet	osynligt
\a	7	datorljud	osynligt
\b	8	baksteg	osynligt
\t	9	tabulator	osynligt
\n	10	radbyte	osynligt
\r	13	vagnretur	osynligt
\u0020	32	mellanslag	osynligt
\"	34	"	"
\'	39	'	'
\\	92	\	\
\u0061	97	a	a
\u00E4	228	ä	ä
\u00E5	229	å	å
\u00F6	246	ö	ö
\u00C4	196	Ä	Ä
\u00C5	197	Å	Å
\u00D6	214	Ö	Ö
\u00FC	252	ü	ü
\u03B2	946	ß	oskrivbart

Den första kolumnen visar några escapesekvenser symboliskt med en bokstav och andra i hexadecimalt Unicode-format. Den andra kolumnen anger koderna decimalt. Den tredje kolumnen skriver ut själva tecknen så länge de är skrivbara. Man ser att svenska specialtecken återges korrekt både när vi i programmet på nästa sida anger dem med sina resp. escapesekvenser (3:e kolumnen) och direkt (4:e kolumnen). Osynliga och oskrivbara tecken beskrivs. Den fjärde kolumnen skriver ut de tecken som är skrivbara direkt dvs genom att ange själva tecknen, inte escapesekvenserna.

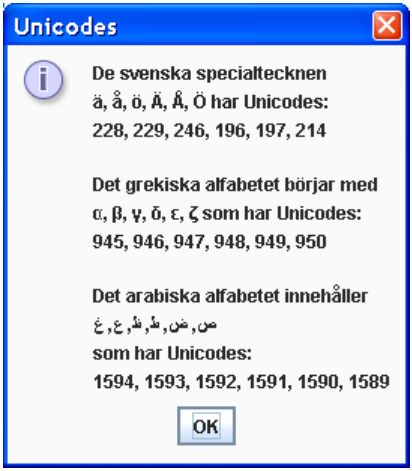
Det är följande C#-kod som producerar utskriften på förra sidan:

```
// Escape.cs
// Skriver ut både symboliska och kodade escapesekvenser
using System;

class Escape
{
  static void Main()
  {
   Console.WriteLine(
   "\n\tEscapesekvens\tUnicode      Tecknet      Direkt \n" +
   "\t-------------------------------------------------- \n" +
   "\t\\0    \t\t" + (int)'\0' + "\t   "                      +
                                 "nolltecknet\t osynligt\n" +
   "\t\\a    \t\t" + (int)'\u0007' + "\t\u0007   "           +
                                  " datorljud\t osynligt\n" +
   "\t\\b    \t\t" + (int)'\b' + "\t   "                     +
                                   "  baksteg\t osynligt\n" +
   "\t\\t    \t\t" + (int)'\t' + "\t   "                     +
                                  " tabulator\t osynligt\n" +
   "\t\\n    \t\t" + (int)'\n' + "\t   " +
                                   "  radbyte\t osynligt\n" +
   "\t\\r    \t\t" + (int)'\r' + "\t   " +
                                  " vagnretur\t osynligt\n" +
   "\t\\u0020  \t" + (int)'\u0020' + "\t   " +
                                 " mellanslag\t osynligt\n" +
   "\t\\\"   \t\t" + (int)'\"'     + "\t\t\"    \t    \"\n" +
   "\t\\\'   \t\t" + (int)'\''     + "\t\t\'    \t    \'\n" +
   "\t\\\\   \t\t" + (int)'\\'     + "\t\t\\    \t    \\\n" +
   "\t\\u0061  \t" + (int)'\u0061' + "\t\t\u0061\t    a \n" +
   "\t\\u00E4  \t" + (int)'\u00E4' + "\t\t\u00E4\t    ä \n" +
   "\t\\u00E5  \t" + (int)'\u00E5' + "\t\t\u00E5\t    å \n" +
   "\t\\u00F6  \t" + (int)'\u00F6' + "\t\t\u00F6\t    ö \n" +
   "\t\\u00C4  \t" + (int)'\u00C4' + "\t\t\u00C4\t    Ä \n" +
   "\t\\u00C5  \t" + (int)'\u00C5' + "\t\t\u00C5\t    Å \n" +
   "\t\\u00D6  \t" + (int)'\u00D6' + "\t\t\u00D6\t    Ö \n" +
   "\t\\u00FC  \t" + (int)'\u00FC' + "\t\t\u00FC\t    ü \n" +
   "\t\\u03B2  \t" + (int)'\u03B2' + "\t\t\u03B2      "     +
                                                 "oskrivbart\n");
  }
}
```

5.5 Unicode

Hur representeras tecken vars kod överskrider ASCII-standardens övre gräns **127**? Det finns många olika utvidgningar av ASCII-tabellen. I C# har man bestämt sig för att använda teckenkodningsstandarden *Unicode* som även används i Windows och i många andra miljöer. Till skillnad från ASCII som lagrar ett tecken i 1 byte (8 bitar) använder sig Unicode av 2 bytes (16 bitar), varför datatypen **char** i C# har 2 bytes till förfogande. Därmed kan man koda ett väsentligt större antal tecken, även sådana från andra språk som arabiska, japanska, kinesiska, hebreiska, kyrilliska osv. Unicode har plats för över en miljon tecken varav drygt 90 000 tecken redan är tilldelade. För att vara bakåtkompatibel har man inkluderat ASCII-standarden i Unicode som en delmängd så att alla tecken upp till koden **127** har samma koder i Unicode som i ASCII, bara att även dessa tar 2 bytes i Unicode. Utöver koden **127** har man infört nya koder. Vilka koder det är, står i Unicode-tabellen som man t.ex. kan hitta på Internet under adressen **http://unicode.coeurlumiere.com**. Några av dem ser man i följande ruta som visar bokstäver från tre olika språk som tagits fram med sina resp. koder enligt Unicode. Utskriften är resultat av javaprogrammet på nästa sida.



Unicode anges som escapesekvensen **\u** följt av ett fyrasiffrigt hexadecimalt tal, dvs så här: **\uAAAA** där **u** står för unicode och **A** för någon hexadecimal siffra. Formatet är föreskrivet vilket bl.a. innebär att ett hexadecimalt tal med mindre än fyra siffror måste inledas med nollor. Alla tecken oavsett skriv- och läsbarhet kan i C# kod skrivas i detta format. I programmet på nästa sida initieras **char**-variabeln **alpha** till tecknet **\u03B1** som är angivet i Unicode-format där **3B1** är den hexadecimala motsvarigheten till det decimala talet **945** som enligt Unicode-tabellen är teckenkoden till den grekiska bokstaven α vilket man kan övertyga sig om med en sökning på Internet. Vi har ingen chans att få in detta tecken i datorn, då det inte finns någon sådan tangent på ett svenskt eller amerikanskt tangentbord. Så, den enda möjligheten att ta fram det är att använda escapesekvensen med Unicode. Apostroferna kring **\u03B1** när det tilldelas **char**-variabeln **alpha** visar att koden **\u03B1** symboliserar endast *ett* tecken – nämligen bokstaven α – och behandlas som vilket tecken som helst. På den nämnda Internet-länken kan man även se att det grekiska alfabetet förekommer som ett sammanhängande block i Unicode-tabellen. Därför kan vi använda teckenaritmetik för att få fram de bokstäver som följer efter α (sid 92). Samma sak görs i programmet med de arabiska bokstäverna genom att initiera **char**-variabeln **arab** till tecknet **\u0635** där **635** är den hexa-

decimala motsvarigheten till det decimala talet **1589**, koden till den arabiska bokstaven ص osv. Anmärkningsvärt är också att programmet automatiskt går över till att skriva från höger till vänster när Unicodes till de arabiska bokstäverna skickas till utskrift. För att få ut de svenska specialtecknen behöver vi inte hämta deras koder från Unicode-tabellen, omvandla dem till hexadecimala och skicka dem som escapesekvenser, därför att vi direkt kan skriva in dem från våra svenska tangentbord.

```
// Unicode.java
                                  // OBS! Detta är ett javaprogram
Import javax.swing.JOptionPane; // från boken Programmering 1
                                  // med Java, se www.techpages.se
class Unicode
{
  public static void Main(String[] a)
  {
    char alpha = '\u03B1', arab = '\u0635';

    JOptionPane.showMessageDialog (null,
      "De svenska specialtecknen\nä, å, ö, Ä, Å, Ö"          +
      " har Unicodes:\n"                                      +
      (int) 'ä' + ", " + (int) 'å' + ", " + (int) 'ö' + ", "+
      (int) 'Ä' + ", " + (int) 'Å' + ", " + (int) 'Ö' + '\n'+

      "\nDet grekiska alfabetet börjar med\n" + alpha + ", "+
      (char)(alpha+1) + ", " + (char)(alpha+2)      + ", "  +
      (char)(alpha+3) + ", " + (char)(alpha+4)      + ", "  +
      (char)(alpha+5) + " som har Unicodes:\n"                +
      (alpha+0) + ", " + (alpha+1) + ", "                     +
      (alpha+2) + ", " + (alpha+3) + ", "                     +
      (alpha+4) + ", " + (alpha+5) + '\n'                     +

      "\nDet arabiska alfabetet innehåller\n" + arab + ", " +
      (char)(arab+1) + ", " + (char)(arab+2)      + ", "    +
       char)(arab+3) + ", " + (char)(arab+4)      + ", "    +
      (char)(arab+5) + "\nsom har Unicodes:\n"               +
      (arab+5) + ", " + (arab+4) + ", "                     +
      (arab+3) + ", " + (arab+2) + ", "                     +
      (arab+1) + ", " + (arab+0),             "Unicodes", 1);
   }
}
```

Programmet **Unicode** använder sig av **(char)(alpha+1)** för att skriva ut β när **alpha** har värdet α fast **alpha** redan är av typ **char**. Anledning till den explicita typkonverteringen till **char** är att additionen gör om parentesens resultat till en **int** så att en explicit omvandling tillbaka till **char** blir nödvändigt. Omvänt när vi behöver Unicode-koden utnyttjar vi just den automatiska typkonverteringen t.ex. i **(alpha+0)** som förorsakas av additionen, för att få fram koden **945** utan explicit typkonvertering. Samma teknik används för **(alpha+1)** osv. för att få koderna till β, γ osv. samt för behandlingen av variabeln **arab**.

5.6 Decimaltalstyperna

C# har tre enkla datatyper för lagring av decimaltal: **`float`**, **`double`** och **`decimal`**. Decimaltal kallas även *flyttal* därför att deras decimalpunkt ”flyter” dvs varierar när de lagras i datorn, beroende på storleken. De omvandlas till ettor och nollor på ett helt annat sätt än heltal och får en form som kallas *flyttalsrepresentation*. Därifrån härstammar också namnet på datatypen **`float`**. Följande program demonstrerar några intressanta egenskaper av decimaltal som delvis är överraskande:

```
// Decimal.cs
using System;

class Decimal
{
  static void Main()
  {
    // float flyt = 4.5;              // Ger kompileringsfel
    float flyt = 4.5f;                // 4.5 som float
    int   hel = (int) flyt;           // Avhuggning till int

    float   flo = 1/3f;               // 3 som float
    double  dou = 1/3d;               // 3 som double
    decimal dec = 1/3m;               // 3 som decimal

    Console.WriteLine(
      "\n\tHeltalsdivision 9 / 2 ger " + (9/2)  + " och "   +
                                       "resten "  +   (9%2)   +
      "\n\tDecimaltalsdivision 9,0 / 2 ger "    + (9.0/2)   +
      "\n\tHeltalsdelen av " +  flyt +  " är "  +  hel      +
      "\n\tfloat variabel   1 / 3   ger " + ' ' + flo       +
      "\n\tdouble variabel  1 / 3   ger " + ' ' + dou       +
      "\n\tKonstanten       1 / 3,0 ger " + ' ' + (1 / 3.0) +
      "\n\tdecimal variabel 1 / 3   ger " + ' ' + dec       +
            "\n\tKonstanta decimaltal lagras som double\n");
  }
}
```

En körning ger följande utskrift:

```
Heltalsdivision 9 / 2 ger 4 och resten 1
Decimaltalsdivision 9,0 / 2 ger 4,5
Heltalsdelen av 4,5 är 4
float variabel   1 / 3   ger  0,3333333
double variabel  1 / 3   ger  0,333333333333333
Konstanten       1 / 3,0 ger  0,333333333333333
decimal variabel 1 / 3   ger  0,33333333333333333333333333
Konstanta decimaltal lagras som double
```

En överraskande egenskap av decimaltal i C# är t.ex. att **4.5** inte direkt kan tilldelas till en **float**-variabel, därför att konstanta decimaltal lagras som **double**. En automatisk typkonvertering nedåt – dvs från **double** till **float** – är inte möjligt (se sid 104), därför att C#-kompilatorn inte tillåter lagringen av ett 8-bytes-värde (**double**) i en 4-bytes-minnescell (**float**) för att undvika overflow (sid 89).

Datatypen **float** lagrar decimaltal i 4 bytes, **double** i 8 bytes och **decimal** i 16 bytes. Därför ser du i körexemplet ovan dubbelt så många siffror i utskriften av **double**-variabeln **dou** jämfört med **float**-variabeln **flo** och dubbelt så många siffror i utskriften av **decimal**-variabeln **dec** jämfört med **double**-variabeln **dou**.

Kan en decimaltalstyp inte specificeras via deklarationen – som är fallet för konstanta decimaltal, så blir datatypen automatiskt **double**. Vill man inte ha det så, utan vill man att det lagras som **float**, måste man använda explicit typkonvertering: **(float) 4.5** eller sätta ett **f** som står för **f**loat utan mellanslag direkt i slutet av värdet: **4.5f** som är bara en kortform för explicit typkonvertering. Även ett heltal med **f** t.ex. **3f** blir omvandlat till **float**. Först **float**-värdet **4.5f** kan tilldelas till **float**-variabeln **flyt**. I nästa sats omvandlas detta **float**-värde till **int** och tilldelas en **int**-variabel:

```
int hel = (int) flyt;
```

Variabeln **hel** ger **4** (se utskriften ovan) vilket visar att typkonverteringen till **int** inte avrundar decimaltalet utan endast tar heltalsdelen dvs trunkerar eller avhuggar decimalerna. På liknande sätt som det lilla **f** kännetecknar det som **float** kan ett **d** i slutet av ett tal karakterisera det som **double**. T.ex. blir **3d** ett **double**-värde så att **1/3d** utförs som en vanlig division mellan två decimaltal som ger ett **double**-värde. **1/3** skulle utföra heltalsdivision och resultera i **0**. Därför ger också **9/2** heltalet **4** och **9%2** resten **1** (sid 75). På samma sätt som suffixet **m** karakterisera ett värde som **decimal**. T.ex. blir **3m** ett **decimal**-värde

I början av kapitlet hade vi skrivit ut de enkla datatypernas gränser (sid 88). Det som hos heltalen är konsekvensen av det begränsade minnesutrymmet – nämligen heltalstypernas gränser – är för decimaltalen deras precision vid den decimala framställningen dvs noggrannheten uttryckt i antal siffror. Vad vi menar är antalet siffror man kan lita på vilket har att göra med den för decimaltal typiska egenskapen att de kan ha oändligt många decimaler. Därför måste de förr eller senare avrundas i alla fall. Det är endast heltal som representeras exakt inom det värdeområde som föreskrivs av datatypen, medan decimaltal representeras i regel bara approximativt (ungefärligt) även om de håller sig inom det tillåtna värdeområdet. Sedan tillkommer avrundningsfelen vars fortplantning vid komplexa beräkningar kan vara förödande, men är relevanta redan vid enkel räkning. T.ex. vid multiplikation av två decimaltal fördubblas antalet decimaler, medan man i regel har samma minnesutrymme för resultatet som för operanderna.

De enkla datatypernas gränser – både för hel- och decimaltal – bestäms av den tilldelade minnesstorleken. Men för att kunna ta reda på decimaltalstypernas noggrannhet behöver man veta hur decimaltal förvandlas till ettor och nollor, vilket

är mer komplicerat än hos heltal. Utan att behöva gå in närmare på detta kan vi konstatera att man vid representation av decimaltal i datorn kan lita på **`7`** siffror om man definierar sina variabler som **`float`**. Utför man inga eller ganska enkla beräkningar borde detta räcka. Är däremot variablerna involverade i lite mer komplexa beräkningar bör man snarare hålla sig till **`double`** där man kan lita på **`15`** siffror vid den interna binära representationen vilket i de flesta praktiska sammanhang är en tillräcklig noggrannhet. Högst noggrannhet har datatypen **`decimal`** med **`28`** siffrors precision. Observera att noggrannheten gäller antal (signifikanta) siffror och inte antal decimaler.

5.7 Automatisk typkonvertering

C# är ett strikt typbestämt programmeringsspråk vilket innebär att alla värden måste ha en datatyp. Data utan datatyp kan inte bearbetas i ett C# program. Detta gäller även för värden som är resultat av ett uttryck. I ett uttryck finns som regel flera värden inblandade både som variabler och konstanter. Man kan inte utesluta att de har olika datatyper. Om flera olika datatyper är inblandade i ett uttryck vilken datatyp ska då beräkningen slutligen resultera i? Vilken datatyp ska t.ex. **`a+b`** få om **`a`** är av typ **`int`**, men **`b`** av typ **`double`**? För att lösa problemet tillämpar C#-kompilatorn automatiskt vissa regler som är begränsade till *enkla* datatyper, med undantag för datatypen **`bool`** vars värden aldrig kan omvandlas till andra typer.

I avsnitt 5.3 behandlades explicit typkonvertering (sid 94) där programmeraren själv uttryckligen kan bestämma när och hur ett värde ska byta datatyp. Vi använde explicit typkonvertering ofta för att få reda på tecknens ASCII-koder genom att omvandla värden av typ **`char`** till värden av typ **`int`** och omvänt. Men typkonverteringar kan även ske automatiskt (implicit) och utan vårt uttryckliga medgivande i aritmetiska uttryck eller vid tilldelningar. Vid tilldelningar därför att tilldelningstecknet **`=`** också är en operator precis som de aritmetiska operatorerna **`+`**, **`-`**, **`*`**, **`/`** eller **`%`**.

Regeln för automatisk typkonvertering

Automatisk typkonvertering kan förekomma vid användning av tilldelningsoperatorn eller i aritmetiska uttryck enligt följande regel:

> Är olika enkla datatyper involverade vid en tilldelning, konverteras till den datatyp som står till vänster om tilldelningstecknet, endast om denne står högre i de enkla datatypernas hierarki:
>
> **`sbyte → byte → char → short → ushort → int → uint → long → ulong → float → double → decimal`**
>
> Är olika enkla datatyper involverade i ett aritmetiskt uttryck, konverteras till den högsta förekommande datatypen enligt hierarkin.

Hierarkin ovan definieras efter datatypernas tilldelade minnesutrymmen samt av att decimaltalstyperna står högre än heltalstyperna. Enligt denna regel får endast automatisk typkonvertering *uppåt* förekomma. Inget värde kan omvandlas automatiskt *nedåt* via tilldelning: Kompilatorn sätter stopp för det. Därför leder t.ex. satsen **`int a = 3.4;`** till kompileringsfelet:

```
Cannot implicitly convert type 'double' to 'int'.
An explicit conversion exists (are you missing a cast?)
```

Med 'double' menar kompilatorn den decimala konstanten **`3.4`** som tolkas som **`double`** och därför med 8 bytes minne (sid 86/105) inte kan omvandlas till och lagras som en **`int`** med 4 bytes minne pga risken för förlust av noggrannhet dvs trun-

kering av siffror. Däremot går det bra att kompilera och även köra **`float b = 3;`** därför att **`3`** som **`int`** kan lagras som en **`float`** enligt de enkla datatypernas hierarki: Här skulle ske en automatisk typkonvertering *uppåt*. Med cast menar kompilatorn explicit typkonvertering och undrar om man har missat en sådan. Det går nämligen lika bra att kompilera och köra **`int a = (int) 3.4;`** där vi själva väljer att trunkera decimaltalet och lagra endast heltalsdelen **`3`** i variabeln **`a`** och ansvara för förlust av noggrannhet.

Regeln för automatisk typkonvertering motiveras av omsorgen för att inte tappa eller förfalska information genom att överföra ett värde från en större minnesplats till en mindre. Vill man ändå ta en sådan risk ska man göra det själv med explicit typkonvertering vilket alltid är möjligt. C# vägrar göra det automatiskt.

En direkt konsekvens av regeln för automatisk typkonvertering är följande regel vars verkan vi redan konstaterade i programmet **`Decimal`** (sid 101):

> Decimaltalskonstanter lagras automatiskt som **`double`**.

Filosofin att inte tappa eller förfalska information leder till att t.ex. den decimala konstanten **`3.4`** får den datatyp som står högst i de enkla datatypernas hierarki, nämligen **`double`**. Man kan tolka regeln även så här: Är ingen datatyp specificerad via deklarationen tas den högsta för att vara på den säkra sidan.

`int`-*regeln*

Det är anmärkningsvärt att automatisk typkonvertering även kan ske när *samma* datatyper är inblandade i ett uttryck, t.ex.: Summan av två **`short`**-variabler omvandlas automatiskt till **`int`** – ett exempel på **`int`**-regeln:

> Är endast heltalstyperna inblandade i ett aritmetiskt uttryck, omvandlas uttryckets värde automatiskt till datatypen **`int`**.

`int`-regeln ska förhindra overflow (sid 89): När t.ex. två **`short`**-värden som var och en ryms i 2 bytes, adderas eller multipliceras med varandra, ska resultatet lagras i en **`int`** dvs i 4 bytes eftersom det finns *risken* att resultatet överskrider **`short`**-gränsen. Därför meddelar också C#-kompilatorn Cannot implicitly convert type 'int' to 'short' … när man försöker att tilldela resultatet av en räkneoperation med **`short`**-variabler till en **`short`**-variabel. Ett exempel på det visas i programmet **`AutoConv`** på nästa sida. En direkt konsekvens av **`int`**-regeln är följande regel:

> Heltalskonstanter lagras automatiskt som **`int`**.

Fattas decimalpunkten i en konstant tolkas den inte längre som decimal- utan som heltal. Men här tas inte den högsta typen bland heltalen som vore **`long`**, utan **`int`**

enligt **int**-regeln. Så, det verkar så som att **int** är en slags favoritdatatyp för heltal därför att alla rutiner kring **int** är optimerade så att C# räknar snabbast med **int**.

En annan sak är det med *namngivna* (symboliska) konstanter som definieras med det reserverade ordet **const** framför datatypen. Deras datatyp specificeras explicit så att reglerna om konstanter borde preciseras så här: Endast icke-namngivna decimaltalskonstanter lagras automatiskt som **double** och ickenamngivna heltalskonstanter lagras automatiskt som **int**. Följande program demonstrerar automatisk typkonvertering samt **int**-regeln:

```
// AutoConv.cs
// int-regeln gör att summan av 2 shortvariabler blir automa-
// tiskt int och ger kompileringsfel när den tilldelas till
// en short. Automatisk typkonvertering uppåt: från short
// till int. Explicit typkonvertering kan leda till felaktigt
// resultat pga overflow
using System;

class AutoConv
{
  static void Main()
  {
    short s1 = 1;
    short s2 = 2;
    // short not_ok = s1 + s2;  // Ger kompileringsfel:
                                // Cannot implicitly convert
                                // type 'int' to 'short'
    short ok = (short) (s1 + s2); // int-regeln gör explicit
                                // typkonvertering nödvändig
    short max = Int16.MaxValue; // short-gränsen
    int hel = max;              // Automatisk typkonvertering
                                // från short till int
    short fel = (short) (hel+1);// Overflow: hel+1 över-
                                // skrider short-gränsen
    Console.WriteLine("\n\t" +
        "Rätt short-värde = "    + ok  + "\n\t"             +
        "short-gränsen = "       + max + "\n\t"             +
        "Overflow: Fel short = " + fel + ": slår runt!\n\t" +
        "Rätt int-värde = " + (hel + 1) + '\n');
  }
}
```

En körning visar följande utskrift:

```
        Rätt short-värde = 3
        short-gränsen = 32767
        Overflow: Fel short = -32768: slår runt!
        Rätt int-värde = 32768
```

Programmet **AutoConv** på förra sidan är ett exempel på hur mycket man kan lära sig av sina misstag: Den bortkommenterade satsen

```
short not_ok = s1 + s2;
```

ger kompileringsfel av följande skäl: För det första bildas uttrycket **s1 + s2** dvs summan av två **short**-variabler med värdena **1** och **2**. Resultatet **3** omvandlas enligt **int**-regeln automatiskt till en **int**. Så långt förekommer inget fel. Men när satsen sedan försöker att tilldela detta **int**-värde till **short**-variabeln **not_ok** blir det fel eftersom **short** står lägre än **int** i de enkla datatypernas hierarki: Automatisk typkonvertering vid tilldelning kan i C# aldrig göras nedåt utan endast uppåt i hierarkin. Additionsoperatorn **+** i uttrycket **s1 + s2** är orsaken till att uttryckets resultat omvandlas till **int**. För att kunna kompilera blir det nödvändigt att explicit typkonvertera ”tillbaka” så att säga till **short** för att kunna tilldela summan till en **short**-variabel:

```
short ok = (short) (s1 + s2);
```

Först nu är det möjligt att skriva ut rätt **short**-värde med variabeln **ok**. Sedan tar programmet **AutoConv** fram övre gränsen till datatypen **short** med hjälp av den lagrade konstanten **Int16.MaxValue** och tilldelar den till **short**-variabeln **max** som sedan skrivs ut korrekt: 32 767 – samma värde vi hade i utskriften av de enkla datatypernas gränser (sid 88) och största värdet som (med tecknet) fortfarande ryms i 2 bytes dvs värdet fyller utrymmet med 16 ettor, 1 för tecknet och 15 för värdet 32 767. I nästa sats vidarebefordras detta värde till en **int**-variabel:

```
int hel = max;
```

Vid denna tilldelning sker nu en automatisk typkonvertering från **short** till **int** och det går bra eftersom det är uppåt: **int** står i de enkla datatypernas hierarki högre än **short**. 32 767 lagras nu både i **max** och **hel**, med skillnaden att det lagras i 2 bytes i **max**, men i 4 bytes i **hel**. När vi bildar **hel+1** är det ok så länge vi lagrar det i en **int**. Men i satsen **short fel = (short) (hel+1);** görs försöket att explicit omvandla det till **short** för att lagra det i en **short**-variabel. Utan den explicita typkonverteringen hade ju koden inte kunnat kompileras därför att C# vägrar att automatisk typkonvertera nedåt från **int** till **short**. Men även om den explicita typkonverteringen kan kompileras genererar den det felaktiga resultatet -32 768 som utskriften av variabeln **fel** visar: **hel+1** dvs 32 768 ryms inte i 2 bytes – ett exempel på overflow (sid 89). Det överskridna värdet ”slår runt” och hamnar på den negativa sidan av **short**-datatypens tillåtna talområdet. Programmet räknar *modulo* 2^n där n är antalet bitar i minnesutrymme som står till förfogande för den aktuella datatypen. Här är n pga **short** 2 bytes dvs 16 bitar. Det blir alltså *modulo* 2^{16} där 2^{16} = 65 536. Värdet ”slår runt” och hamnar på andra ändan av **short**:s talområde innebär här: 32 768 – 65 536 = -32 768. Om *modulo* läs på sid 75. Det farliga med overflow är att kompilatorn inte ger något felmeddelande, inte ens en varning, och att exekveringen av programmet inte påverkas. Om man inte skriver ut värdet till **fel** märker man inte ens felet. Programmet ”fungerar”, men producerar ett felaktigt resultat.

Sammanfattning av kapitel 4 och 5

- *Variabel* = platshållare (minnescell) för ett värde (minnescellens innehåll).
- *Datatyp* = föreskrift om hur en viss typ av data ska lagras i datorn, hur mycket minne den tar och därmed hur stora värden den kan lagra samt vilka operationer man får utföra med denna typ av data.
- *Definition* innebär att *skapa* en variabel genom allokering av minnesutrymme för variabeln. Det görs genom att ange variabelns datatyp.
- *Deklaration* innebär att ange en variabels namn och datatyp.
- *Tilldelning* innebär att ge en variabel ett värde (att placera data i en minnescell) och kan göras med tilldelningsoperatorn eller via inläsning.
- *Initiering* innebär att tilldela en variabel första gången ett värde.
- Regler för användning av variabler:
  - En variabel måste definieras dvs skapas med en *datatyp* innan den kan initieras. Skälet är att C# är ett *strikt typbestämt* programmeringsspråk.
  - En variabel måste vara definierad *och* initierad innan den får användas.
- *Enkla datatyper* representerar endast *ett* värde åt gången dvs *ett* heltal, *ett* decimaltal, *ett* tecken eller *ett* sanningsvärde.

  Följande enkla datatyper är fördefinierade: **`bool`** för sanningsvärden; **`char`** för tecken; **`sbyte`**, **`byte`**, **`short`**, **`ushort`**, **`int`**, **`uint`**, **`long`**, **`ulong`** för heltal samt **`float`**, och **`double`** och **`decimal`** för decimaltal .
- Varje enkel datatyp har en tilldelad storlek på minnesutrymme som i C# är enhetlig för alla plattformar. Överskridandet av de föreskrivna gränserna leder till overflow (sid 89).
- När olika enkla datatyper är inblandade i en tilldelning eller i ett aritmetiskt uttryck utför kompilatorn *automatisk typkonvertering* enligt följande regler:
  1. Vid tilldelningar omvandlas till datatypen till vänster om tilldelningstecknet endast om denne står högre i de enkla datatypernas hierarki.
  2. Vid aritmetiska uttryck omvandlas till den högsta i uttrycket inblandade datatypen enligt datatyphierarkin. Decimaltalskonstanter lagras som **`double`**.
  3. **`int`**-regeln: Är endast heltal inblandade i ett aritmetiskt uttryck, omvandlas uttryckets värde automatiskt till **`int`**. Heltalskonstanter lagras som **`int`**.
- *Explicit typkonvertering* kan göras med koden: **`(`***datatyp***`)`** *uttryck*

Övningar till kapitel 5

5.1 Skriv ett program som läser in 3 tecken och skriver ut dem i omvänd ordning.

Tips: I Python kan detta problem lösas med följande kod:

```
text = input('\nMata in tre tecken skilda med mellanslag:\t')

tecken1 = text[0]
tecken2 = text[2]
tecken3 = text[4]

print('\nTecknen i omvänd ordning:\t\t       ',
               tecken3, tecken2, tecken1, '\n')
```

Försök att hitta en egen lösning, annars översätt pythonkoden ovan till C#.

5.2 Skriv ett program som läser in en gemen och skriver ut dess versal och sedan läser in en versal och skriver ut dess gemen.

5.3 Experimentera med programmet **`Int2char`** (sid 95) för att ta reda på ASCII-koden till datorns ljudsignal. Dvs kör så länge tills du vid inmatning av ett heltal hör ett pip från datorn. Ändra datatypen till variabeln **`code`** från int till **`char`**. Åtgärda kompileringsfelet. Hör du fortfarande pipet när du matar in ASCII-koden för ljudsignal? Förklara.

5.4 Kryptering av tecken: Skriv ett program som läser in ett tecken och förskjuter det i teckentabellen med ett visst antal steg som en slags krypteringsnyckel. Skriv ut både det inlästa och det förskjutna tecknet på ett användarvänligt sätt. Börja med att hårdkoda krypteringsnyckeln och fortsätt med att läsa in den.

5.5 Kryptering av ord: Skriv ett program som läser in fem tecken och skriver ut dem förskjutna med *ett* steg i ASCII-tabellen så att t.ex. inmatningen **`Kalle`** ger utskriften **`Lbmmf`**. Återställ sedan det krypterade ordet. Vidareutveckla programmet genom att utöka och läsa in antalet steg (krypteringsnyckeln).

5.6 Om ökningsoperatorn **`++`** (sid 77) har vi lärt oss:

`letter++;` gör samma sak som **`letter = letter + 1;`**

Ändå ger programmet **`Ovn_5_6`** kompileringsfel om vi ersätter satsen **`letter++;`** med **`letter = letter + 1;`** Varför? Förklara! Försök att åtgärda felet genom att ersätta **`letter++;`** med en modifikation av **`letter = letter + 1;`**

```
using System;

class Ovn_5_6
{
  static void Main()
  {
    char letter = 'Y';
    Console.Write('Tecknet " + letter + " har koden " +
                                          (int) letter);

    letter++;

    Console.WriteLine("\nTecknet " + letter +
                          " har koden " + (int) letter);
   }
}
```

5.7 Ta reda på vilka escapesekvenser som gäller för de svenska specialtecknen genom att köra programmet **Escape** (sid 98) och skriv ut de sammanhängande strängerna **äåöÄÅÖ** och **ä å ö Ä Å Ö** med de hexadecimala escapesekvenserna istället för direkt. Gör samma sak med de decimala Unicodekoderna.

Kapitel 6

Kontrollstrukturer

6.1 Vad är kontrollstrukturer?

Kontrollstrukturer är algoritmers byggstenar – 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:

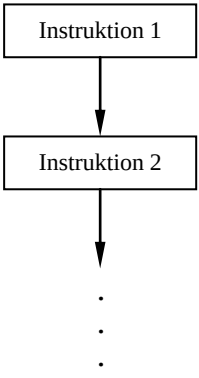
- **Sekvens (följd)**
- **Selektion (val)**
  - Enkel selektion
  - Tvåvägsval
  - Flervägsval
- **Repetition (upprepning)**
  - Förtestad repetition
  - Eftertestad repetition
  - Bestämd repetition

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 C#. Kontrollstrukturer används och är i princip uppbyggda enligt samma logik i alla programmeringsspråk. Både Javas och C#:s kontrollstrukturer har – när det gäller syntaxen – tagits över från och är i princip identiska med *C/C++* bortsett från några detaljer. Ännu längre tillbaka i historien kan man hitta deras spår i de första strukturerade språken som *Algol*, *Simula* och *Pascal*.

Sekvens (följd)

Redan i algoritmdefinitionen förekommer begreppet *följd*:

> ”En algoritm är en *följd* av precisa anvisningar, s.k. elementära instruktioner, som löser ett givet problem . . . ”



En *sekvens* är alltså 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.

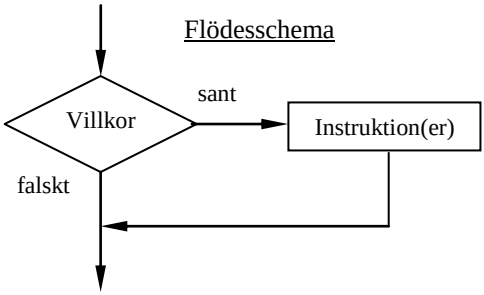
6.2 Enkel selektion: `if`-satsen

Enkel selektion är ett val utan alternativ dvs valet mellan att göra något eller ingenting. Det som avgör valet är ett villkor. Är villkoret sant, utförs en eller flera instruktioner. Är villkoret falskt, görs ingenting.

Pseudokod

OM *villkor* uppfyllt
instruktion(er)

Flödesschema



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

```
if (condition)
{
    statement(s);
}
```

Första raden kallas *huvudet*, resten är *kroppen* som omsluts av *klamrarna* **{** och **}**. Består kroppen endast av *en* sats kan klamrarna utelämnas:

```
if (condition)
    statement;
```

```
// SimpleIf.cs
// Förhindrar division med 0 (& därmed programavbrott) med if
using System;
class SimpleIf
{
  static void Main()
  {
    Console.Write("\n\tMata in ett heltal:       ");
    int no1 = int.Parse(Console.ReadLine());
    Console.Write("\n\tMata in ett heltal till:  ");
    int no2 = int.Parse(Console.ReadLine());

    if (no2 != 0)
       Console.WriteLine("\n\t" + no1 + " heltalsdividerad" +
                  " med " + no2 + " blir " + no1/no2 + '\n');
    if (no2 == 0)
       Console.WriteLine("\n\tOBS!\n\t" +
               "Du har matat in 0 för det andra talet.\n\t" +
               "Det går inte att dividera med 0.\n");
  }
}
```

Programmet läser in två heltal 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 dialog får man när man matar in ett värde skilt ifrån **0** till det andra talet:

```
Mata in ett heltal:       19

Mata in ett heltal till:  5

19 heltalsdividerad med 5 blir 3
```

P.g.a. datatypen **int** till **no1** och **no2** blir det *heltals*division: **19/5** ger **3** hela, resten skrivs inte ut här. Matas in däremot **0** till det andra talet uppstår följande dialog:

```
Mata in ett heltal:       8

Mata in ett heltal till:  0

OBS!
Du har matat in 0 för det andra talet.
Det går inte att dividera med 0.
```

Inmatning av **0** till det andra talet genererar ett ”felmeddelande”, annars får man ut från programmet **SimpleIf** resultatet av divisionen för vilka heltal som helst. Låt oss nu titta närmare på **if**-satserna som åstadkommer distinktionen mellan dessa två alternativ: Det finns två **if**-satser i programmet **SimpleIf**. Den första **if**-satsens huvud

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

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

Huvudet inleds med det reserverade ordet **if** utan att avsluta raden med semikolon. Utan semikolon, därför att **if**-satsen inte är avslutad än i slutet av denna rad. Sedan ska ju kroppen följa. Efter **if** skrivs ett *villkor (condition)* inom parentes. Observera att parenteserna tillhör syntaxen och inte får under några omständigheter utelämnas. Men hur skriver man *villkor* i C#? Vi blir påminda om algoritmer när vi hör begreppet villkor.

Villkor

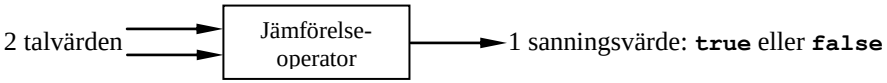
Det är viktigt att skilja mellan begreppen *villkor* och *instruktion*. Enklast kan ett villkor förklaras som en *fråga* som endast kan besvaras jakande eller nekande: är **no2** skilt ifrån **0**, ja eller nej? Närmare bestämt är ett villkor en *utsaga* som endast kan vara sann eller falsk. Medan en instruktion är en handling som ska *utföras* kan ett villkor endast *testas* för att få ut svaret ja eller nej, sant eller falskt. **if (no2 != 0)** testar om **no2** är skilt ifrån **0** eller ej. Variabeln **no2**:s värde jämförs med **0**. Finns icke-likhet mellan dessa värden är villkoret sant, annars är villkoret falskt.

Dubbeltecknet **!=** (utan mellanslag) är en s.k. *jämförelseoperator*. Det är vanligt att formulera villkor med jämförelseoperatorer. *Icke lika med* med symbolen **!=** är en av dem. Det finns fler som används i **if**-satsers villkor. Därför ska vi titta närmare på sådana operatorer.

Jämförelseoperatorer

<	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:

```
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 69 pratade vi om skillnaden mellan likhet och tilldelning och poängterade att = i C# 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 den andra **if**-satsen.

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**

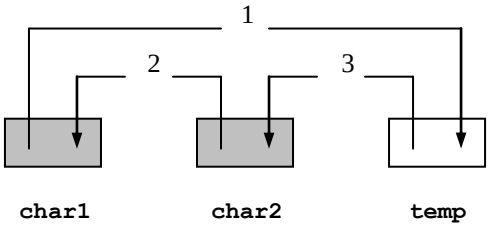
Precis som **!=** är även dubbeltecknet == (utan mellanslag) en jämförelseoperator, men står för *lika med*. Dvs värdet i variabeln **no2** jämförs med **0**. Finns likhet mellan dem är **if**-satsens villkor sant, annars är villkoret falskt. Observera skillnaden mellan likhet som kodas med *två* likhetstecken == 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.

Flera satser i `if`

I programmet **SimpleIf** (sid 113) består båda **if**-satsers kroppar av en enda sats. Därför räcker det med satsens semikolon för att avskilja kroppen från programmets efterföljande satser. Men om **if**-satsens kropp består av flera satser *måste* klamrarna **{** och **}** markera kroppen. Hur ska annars kompilatorn skilja mellan **if**-kroppens och de efterföljande satserna? I programmet **MiniSort** (sid 116) finns ett exempel på detta. Men först ska vi titta på programmets algoritm som handlar om sortering:

Algoritm för platsbyte

Låt oss anta vi har två tecken **char1** och **char2** som vi vill byta pltas på. För att kunna göra det behövs en tredje, temporär plats. Vi börjar med att lägga undan **char1** på den temporära platsen **temp** (steg 1). Sedan byter vi plats på **char2** och lägger det i **char1** som tömdes i steg 1 (steg 2). Och slutligen, i steg 3, lägger vi **char1** som under tiden mellanlagrats i **temp**, in i **char2** som tömdes i steg 2:



Illustrationen ovan är en grafisk beskrivning av algoritmen där 1, 2 och 3 anger ordningen i den. Den tredje platsen **temp**, behövs, för att temporärt lägga undan det felplacerade tecknet. I följande program inplementerar vi algoritmen ovan:

```
// MiniSort.cs
// Läser in 2 tecken och sorterar dem i teckentabellens ord-
// ning med hjälp av en algoritm för platsbyte av två objekt
using System;
class MiniSort
{
   static void Main()
   {
      char char1, char2, temp;
```

```
      Console.Write("\n\tTvå osorterade tecken:\n\n\t" +
              "Mata in två tecken skilda med mellanslag:\t");
      string text = Console.ReadLine();

      char1 = text[0];          // Första tecknet tas ut
      char2 = text[2];          // Andra tecknet tas ut

      if (char1 > char2)        // tecknens ASCII-koder jämförs
      {
         temp  = char1;         // Algoritm för platsbyte
         char1 = char2;         // av två tecken
         char2 = temp;          // Flera satser i if
      }

      Console.WriteLine("\n\tDe två tecknen sorterade:\t\t\t"
                                + char1 + ' ' + char2 + "\n");
   }
}
```

I följande körexempel byts plats på de inmatade tecknen **Z** och **A** som har blivit inmatade i fel ordning. De sorteras enligt teckentabellens ordning:

```
      Två osorterade tecken:

      Mata in två tecken skilda med mellanslag:       Z A

      De två tecknen sorterade:                       A Z
```

Algoritmens kärna ligger i **if**-satsen med sina tre satser. I den första satsen lägger vi undan **char1**:s värde i **temp** (steg 1 i bilden ovan). I den andra satsen byter vi plats på **char2**:s värde och lägger det i **char1** (steg 2). Och slutligen läggs **temp** som under tiden har mellanlagrat **char1**:s värde, in i **char2** (steg 3). Platsbytet på **char1** och **char2** äger endast rum om de inmatade teckenvärdena är felplacerade dvs endast om **char1 > char2**. Annars behåller de sina platser.

I körexemplet ovan jämför **if**-satsens villkor **char1 > char2** värdena **Z** och **A** med varandra. Men tecken kan inte sättas i en relation av typ ”större än” till varandra. I själva verket är det Unicode-koderna till **Z** och **A** som jämförs med varandra. Det är endast tal som kan jämföras med varandra. Jämförelseoperatorn **>** behandlar **char**-variablerna **char1** och **char2** som *tal* precis som aritmetiska operatorer gör.

Block

I C# kallas ett antal satser som omsluts av klamrarna **{** och **}** för ett *block*. Blockets uppgift är att gruppera satserna inom klamrarna och *avgränsa* dem från andra delar av programmet. Klamrarna är *gränser* mellan programmets olika delar. De sätter gräns för variablers räckvidd. För att överskrida dem måste vissa regler om *blockstruktur* beaktas. Ibland kan blockets avslutande klammer t.o.m. ersätta ett ev. efterföljande semikolon. Exempel på block fanns redan i vårt allra första program.

`Main()`-metodens kropp bildar ett block, det s.k. **`Main()`**-blocket. För läslighetens skull brukar blockets satser skrivas indragna. Dessutom placerar man blockets klamrar på separata rader. Alternativt kan man placera den inledande klammern i slutet av huvudets rad och den avslutande i blockets slut på separat rad. Jag väljer dock att använda den förstnämnda stilkonventionen då denna ger en mer lättläst kod: Man kan lättare para ihop de inledande och avslutande klamrarna.

I programexemplet **`MiniSort`** kodas sorteringsalgoritmen i de tre tilldelningssatser som finns i **`if`**-satsens kropp. Avgränsningen innebär här att alla tre satser hör till **`if`**-satsen och att alla tre ska utföras i fall att **`if`**-satsens villkor är sant. Om blockmarkeringen med klamrarna fattades, skulle endast den första av de tre satserna utföras, vilket skulle innebära att sorteringsalgoritmen inte utförs i sin helhet, dvs ingen sortering sker.

Tomma **`if`***-satser*

Vad händer om man av misstag skriver ett semikolon i slutet på **`if`**-satsens huvud? Dvs så här: **`if (no2 == 0);`** Kompilatorn kan tolka koden endast som en tom **`if`**-sats dvs **`OM`** *no2 lika med 0* gör ingenting! Semikolonet avslutar **`if`**-satsen då det inte finns någon sats mellan villkoret och semikolonet, dvs kroppen är tom. Kroppen som följer kommer i alla fall att utföras och inte bara om *no2 är lika med 0*. Genom att skriva semikolonet i slutet på **`if`**-huvudet, kopplar man bort kroppen från **`if`**-satsen och dess villkor. Tomma **`if`**-satser är i regel meningslösa även om de kan kompileras. Se därför upp för regeln vi nämnde inledningsvis: **`if`**-satsens huvud får inte avslutas med semikolon om man inte uttryckligen vill ha en tom **`if`**-sats.

Villkorlig initiering

Även om man i C# har tagit över kontrollstrukturers syntax från C++ förekommer små skillnader. En av dem är villkorlig initiering av variabler som inte får göras i C#, men är tillåten i C++. Det handlar inte om kontrollstrukturers syntax utan om behandlingen av variabler där C# har en striktare policy än C++ som syftar åt mer stabilitet av koden. Variabler deklarerade till enkla datatyper i en metod – och detta gäller förstås även för **`Main()`**-metoden – måste initieras innan (om) de används. I C# får initieringen inte vara villkorlig dvs stå i en **`if`**-sats. Närmare bestämt får initieringen inte skrivas i kroppen till en **`if`**-sats vars villkor involverar variabler. Detta gäller oavsett villkorets sanningsvärde. Även om villkoret är sant kan koden inte kompileras om variabeln initieras i **`if`**-satsen och villkoret är formulerat med variabler. I följande program står initieringen av variabeln **`letter`** i en **`if`**-sats och är därmed beroende av **`if`**-satsens villkor i vilket variabeln **`i`** är involverad. Därför kan koden inte kompileras fast villkoret **`i == 0`** är pga **`i`**:s initiering sant:

```
// CondInit.cs
// Ger kompileringsfel pga villkorlig initiering
// av variabeln tecken i if-satsen
using System;

class CondInit
{
   static void Main()
   {
      char letter;
      int i = 0;

      if (i == 0)
         letter = 'a';              // Villkorlig initiering

      Console.WriteLine(letter);
   }
}
```

Kompilatorn genererar felmeddelandet: Use of unassigned local variable 'letter'
Dvs C#-kompilatorn anser variabeln **letter** som icke-tilldelad. Samma felmeddelande får man om man missar att tilldela en variabel. Problemets lösning är att helt och hållet koppla bort tilldelningen från villkoret och skriva den fristående:

```
// UncondInit.cs                    // Kan kompileras
using System;

class UncondInit
{
   static void Main()
   {
      char letter;
      int i = 0;

//    if (i == 0)
         letter = 'a';              // Ovillkorlig initiering

      Console.WriteLine("\n " +
        "Nu när if är bortkommenterad är variabeln letter " +
           "initierad\ntill " letter + "\n utan villkor!\n");
   }
}
```

Istället för kompileringsfel får vi nu följande utskrift när vi kör:

```
Nu när if är bortkommenterad är variabeln letter initierad
till a
utan villkor!
```

I programmet **`UncondInit`** är initieringen av **`letter`** helt oberoende av något villkor. Raden som inleder **`if`** och därmed hela **`if`**-satsen är bortkommenterad. Även om initieringen av **`letter`** fortfarande står indragen, är den en fristående sats utan villkor.

Anmärkningsvärt är att programmet **`CondInit`** skulle kunna kompileras om man byter ut **`if`**-satsens huvud mot **`if (1 == 1)`** eller **`if (true)`** dvs om endast konstanter är involverade i villkoret. Endast 'variabelt' formulerade villkorliga initieringar sätter C#-kompilatorn stopp för. Därför måste regeln om villkorlig initiering formuleras så här:

> Variabler vars initiering är beroende av icke-konstanta villkor leder i C# till kompileringsfel.

`i == 1` är ett icke-konstant villkor, därför att dess sanningsvärde är beroende av variabeln **`i`**:s värde.

Förbudet mot villkorlig initiering är inte begränsad till **`if`**-satser utan gäller även i andra kontrollstrukturer där villkor är inblandade. T.ex. använder även programmet **`Switch`** i övernästa avsnitt ovillkorlig initiering (sid 125). Ett annat exempel kan man hitta i programmet **`NestedFor`** (sid 142).

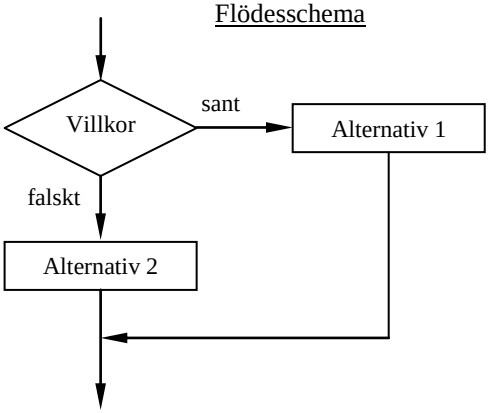
6.3 Tvåvägsval: if-else-satsen

Tvåvägsval är ett val mellan två alternativ. Precis som i **`if`**-satsen görs valet pga ett enda villkor. Är villkoret sant, utförs en eller flera instruktioner. Låt oss kalla dessa *alternativ 1*. Är villkoret falskt, utförs en annan uppsättning instruktioner som vi kallar *alternativ 2*. **`OM-ANNARS`**-satsen är ett exempel på tvåvägsval. Allmänt kan tvåvägsvalet beskrivas så här:

Pseudokod

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

Flödesschema



där
alternativ 1 och
alternativ 2 är två olika instruktioner eller uppsättningar av instruktioner.

Endast ett av de två alternativen kommer att utföras, beroende på villkorets sanningsvärde. Då sanningsvärdena sant och falskt utesluter varandra, utesluter även de båda alternativen varandra. Därför går flödet (pilen i flödesschemat) efter alternativ 1 till flödet *efter* alternativ 2. Det vore logiskt fel att leda pilen till ett ställe *före* alternativ 2. **`if-else`**-sats kodas generellt på följande sätt:

```
if (condition)
{
    statement(s)1;
}
else
{
    statement(s)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 avsevärt:

```
if (condition)
    statement1;
else
    statement2;
```

Observera att varje sats i **`if-else`**-satsen måste avslutas med semikolon enligt semikolonets roll i C# som satsavslutningstecknet. Detta gäller även för den allra

sista satsen i ett block och för statement1 ovan strax före **else**. Därför förekommer flera semikolon – minst två – även om vi pratar om *en* **if-else**-sats, vilket beror på att **if-else**-satsen är en huvudsats som innehåller flera delsatser, minst två. Jämför detta med huvud- och underinstruktioner i algoritmer. Följande exempel visar **if-else**-satsen med endast en sats i respektive **if-else**-del:

```
// IfElse.cs
// Läser in ett heltal och avgör om det är jämnt eller udda
// Tvåvägsval: if-else-satsen med EN sats i resp. if-else-del
using System;

class IfElse
{
   static void Main()
   {
      Console.Write("\n\tMata in ett heltal:\t" );
      int number = int.Parse(Console.ReadLine());

      if (number % 2 == 0)
         Console.WriteLine("\n\t" +
            "Det inmatade talet " + number + " är jämnt.\n");
      else
         Console.WriteLine("\n\t" +
            "Det inmatade talet " + number + " är udda.\n") ;
   }
}
```

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

```
        Mata in ett heltal:     5

        Det inmatade talet 5 är udda.
```

Med ett jämnt tal som inmatning får vi följande dialog:

```
        Mata in ett heltal:     6

        Det inmatade talet 6 är jämnt.
```

Det egentliga jobbet – nämligen att avgöra mellan *jämnt* och *udda* – har gjorts med hjälp av modulooperatorn **%** som används i **if**-satsens huvud:

```
if (number % 2 == 0)
```

och betyder: **OM** *resten vid heltalsdivision av* **number** *med* **2** *är lika med* **0**

9 % 2 t.ex. ger **1** dvs inte **0**, därför är **9** udda. **8 % 2** däremot ger **0**, därför är **8** ett jämnt tal. Alla jämna tal ger resten **0** vid heltalsdivision med **2**. Alla udda tal ger resten **1** vid heltalsdivision med **2**. Om modulooperatorn läs på sid 75.

6.4 Flervägsval

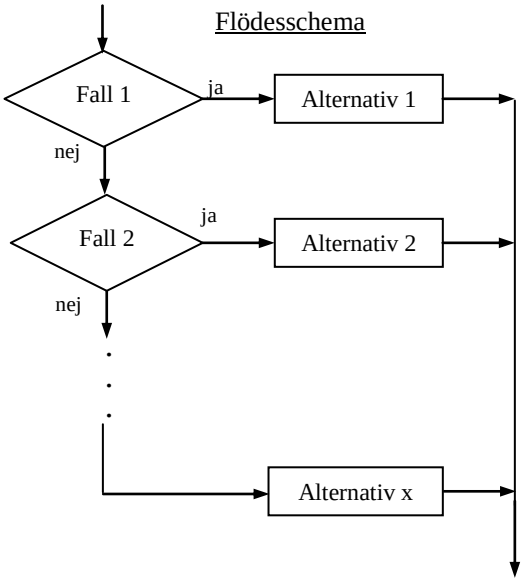
Flervägsval – valet mellan fler än två alternativ – kan programmeras på olika sätt. Det enklaste sättet är att beskriva flervägsval med flera tvåvägsval som antingen följer efter varandra eller är nästlade i varandra. Att rada upp flera **`if`**- eller **`if-else`**-satser efter varandra är inget fel, men kan i praktiken ha sina begränsningar. När du kommit så här långt i boken bör du klara av det på egen hand. Lite svårare är det att nästla **`if`**- eller **`if-else`**-satser vilket vi kommer att ta upp lite längre fram (sid 128). En annan möjlighet att programmera flervägsval är sammansatta villkor. Standard-kontrollstrukturen däremot som ofta används för flervägsval och har motsvarigheter i så gott somm alla programmeringsspråk, är den s.k. **`switch`**-satsen. Vi börjar med den och kommer att fortsätta med de andra alternativen.

`switch`-*satsen*

En nästlad **`if-else`**-sats kan bli komplicerad och oöverskådlig när antalet alternativ man ska välja emellan, växer. För att strukturera komplexiteten vid stora antal alternativ har man i C# tagit fram kontrollstrukturen **`switch`** för flervägsval som i vissa fall är enklare och mer överskådlig och i de flesta fall bättre strukturerad än ett antal nästlade tvåvägsval, även om också den har vissa begränsningar vilket kommer att diskuteras senare. Så här ser **`switch`**-satsens flödesschema och pseudokod ut:

Pseudokod

`VÄLJ` *fall* ur
  Fall 1: Alternativ 1
  Fall 2: Alternativ 2
    .
    .
    .
  Annars: Alternativ x



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

I C# kodas **`switch`**-satsen på följande sätt:

```
switch (expression)
{
    case constant1 :
      statement(s)1;
      break;
    case constant2 :
      statement(s)2;
      break;
     .
     .
     .
    default:
      statement(s)x;
      break;
}
```

Första raden är **switch**-satsens *huvud* och får inte avslutas med semikolon. Resten är **switch**-satsens *kropp* som består av ett block. Kroppens avslutande klammer ersätter här det semikolon som skulle avsluta hela **switch**-satsen. *expression* i huvudet är en slags väljare, ett uttryck (sid 71) som kan innehålla variabler och vars värde får vara av typ **int**, **char** eller **string**. I det enklaste fallet – i våra exempel förekommer bara det enklaste fallet – kan *expression* vara en **int**- eller **char**-variabel. *constant1*, *constant2* osv. däremot måste vara konstanta uttryck som *inte* får innehålla variabler. När **switch**-satsen exekveras, jämförs *expression* i huvudet en i taget med alla konstanter som står efter **case**. Jämförelsen innebär:

```
if (expression == constant1)
if (expression == constant2)
          .
          .
          .
```

Då blir villkoren som är dolda i **switch**-satsen avslöjade: Man ser att de är hårdkodade med operatorn == och inte kan ersättas med andra jämförelseoperatorer. Två enskilda värden kan jämföras med varandra endast på likhet. För att testa om ett värde ligger i ett intervall (olikheter) kan nästlad **if-else** användas. Två identiska konstanter i en **switch**-sats leder till kompileringsfel.

Om likhet föreligger mellan expression och en konstant, så kommer man in i **switch**-satsens kropp och utför alla satser som följer **case** tills **break** bryter **switch**-satsen eller kroppen slutar. Programmet utför alltså inte bara de satser som omedelbart följer det **case** där likheten inträffar, utan *alla* satser som följer ända tills en **break**-sats kommer. **switch**-satsen väljer endast ett fall bland flera, så att varje **case** måste avslutas med **break**. Till skillnad från C++ och Java är **break**-satsen i C# obligatorisk. Det finns inte möjligheten att utelämna **break** eller skriva ”tomma” **case**-satser. Inte ens det allra sista **break** i **default**-satsen får inte utelämnas. Gör man det ger kompilatorn felmeddelandet: Control cannot fall through from one case label ('default:') to another.

Följande program demonstrerar **switch**-satsen:

```
// Switch.cs
// En enkel kalkylator: Flervägsval med switch-satsen
using System;

class Switch
{
  static void Main()
  {
    char  op;
    double no1, no2, answer = 0;

    Console.Write("\n\tMata in no1:\t");
    no1 = Convert.ToDouble(Console.ReadLine());
    Console.Write("\n\tMata in en " +
                  "operator +, -, *, / eller ^ :\t");
    op = Convert.ToChar(Console.ReadLine());
    Console.Write("\n\tMata in no2:\t");
    no2 = Convert.ToDouble(Console.ReadLine());

    switch (op)               // switch börjar
    {
      case '+':
         answer = no1 + no2;
         break;
      case '-':
         answer = no1 - no2;
         break;
      case '*':
         answer = no1 * no2;
         break;
      case '/':
         answer = no1 / no2;
         break;
      case '^':
         answer = Math.Pow(no1, no2);
         break;
      default:
         Console.WriteLine("\n\tOBS! Felaktig inmatning:"   +
            "\n\tDu får mata in endast +, -, *, / eller ^ " +
               "som operator.\n\n");
         op = '?';
         break;
    }                         // switch slutar

    if (op != '?')
       Console.WriteLine("\n\tResultat:\t" + no1 + "  " +
       op + "  " + no2 + "  =  " + answer + "\n\n");
  }
}
```

Vi matar in först ett tal, sedan ett av tecknen **+, -, *, /** eller **^** och sist ett tal till, där **^** ska utföra operationen *upphöjt till*. Resultatet av resp. räkneoperation skrivs ut om man följer instruktionerna. I **switch**-satsen väljs det inmatade alternativet bland de fem symbolerna för räknesätten och räkneoperationen utförs. Medan talen deklareras som **double** är symbolen för räknesättet en **char**-variabel som kallas **op** och används i **switch**-satsen som väljare och ska stå för *operator*. Valet av det mer beskrivande namnet *operator* var inte möjligt eftersom **operator** är ett reserverat ord i C# (sid 36). En körning med korrekt inmatning ger följande dialog:

```
Mata in no1:   24,5
Mata in en operator +, -, *, / eller ^ :        ^
Mata in no2:   7,8
Resultat:      24,5  ^  7,8  =  68469232237,5913
```

Här har operatorn **^** valts dvs *upphöjt till* (exponentiering). I programmet **Switch** har operationen utförts med metoden **Pow()** som är fördefinierad i klassen **Math** som i sin tur ligger i C#:s namnutrymme **System** – det bibliotek som inkluderas med **using**-direktivet. Metoden **Pow(***a*, *b***)** tar in två **double**-parametrar *a* och *b* och returnerar **double**-värdet *a* upphöjt till *b* enligt en inbyggd matematisk formel.

Matar man in trots instruktion en felaktig operator dvs något annat tecken än **+, -, *, /** eller **^**, får man ut en dialog av typ:

```
Mata in no1:   3
Mata in en operator +, -, *, / eller ^ :        \
Mata in no2:   5
OBS! Felaktig inmatning:
Du får mata in endast +, -, *, / eller ^ som operator.
```

Anledningen är att satsen för den här utskriften är placerad i **switch**-satsens **default**-del som är motsvarigheten till **else** i de andra varianterna av selektion. Om ingen likhet påträffats i någon **case**-sats mellan **op** och tecknen **+, -, *, /, ^** 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 ha den kvar.

case

I varje **case** testas ett villkor på likhet mellan expression och en konstant. Men vad menas med *likhet* i **case**-satserna? Där står ju ingen likhet. Jo, det är därför att den

är gömd, den ingår implicit där. Som vi redan nämnde, görs i själva verket jämförelsen på likhet vilket man ser när man översätter den första **case**-satsen till **if**:

```
if (op == '+')
{
   answer = no1 + no2;
   break;
}
```

break

är ett reserverat ord i C# som bryter programflödet i **switch**-satsen och i loopar och skickar programflödet till den första satsen efter det block där **break** skrivs. Alla satser mellan **break** och blockets avslutande klammer **}** hoppas över. I det här fallet gör alltså **break** att programflödet lämnar **switch**-satsen. Detta garanterar ett entydigt val mellan flera alternativ. **break**-satsen är som sagt obligatorisk.

Variabeln **op** läses in med: **op = Convert.ToChar(Console.ReadLine());** som är ett nästlat anrop av två metoder: Först anropas **ReadLine()** som returnerar tecknet vi matar in som ett **String**-objekt. Sedan anropas **ToChar()** för att omvanbdla strängen till **char** som tilldelas **op**. Sedan jämför **switch**-satsen variabeln **op**:s värde med de fem teckenkonstanterna **'+'**, **'-'**, **'*'**, **'/'** och **'^'**. Hittar den likhet med någon av dem, utförs de satser som följer efter resp. **case** tills **break** bryter **switch**-satsen. På så sätt träffas ett entydigt val mellan de fem alternativen. Hittas ingen likhet, har användaren matat in ett tecken som inte är en räkneoperation. **Default**-satsen skriver ut meddelandet **OBS! Felaktig inmatning:** …

Två frågor är kvar att besvara innan vi lämnar **switch**-satsen:

1. Varför är variabeln **answer** initierad till **0** direkt vid deklarationen före och inte i **switch**-satsen där den används? Vi har redan sett att **switch**-satsen är en strukturerad samling av **if**-satser. Och i **if**-satser är villkorlig initiering inte möjligt (sid 118). När regeln formulerades sades också att förbudet gällde även för andra kontrollstrukturer. Faktiskt skulle en initiering av variabeln **answer** i **switch**-satsen räknas som en villkorlig initiering och leda till kompileringsfel därför att **switch**-satsens väljare är en variabel dvs en sådan initiering vore då beroende av t.ex. villkoret **op** == **'+'** osv. Testa gärna!

2. Varför har variabeln **op** tilldelats **?** i **default**-delen av **switch**-satsen? Det har att göra med den **if**-sats som följer efter **switch**-satsen som har villkoret **op != '?'** och skriver ut resultatet. Denna kombination ska *förhindra* att resultatet dyker upp i fall och *efter* att felmeddelandet har skrivits ut. Resultatet ska endast visas när variabeln **op** verkligen fått ett av värdena **+**, **-**, *****, **/**, **^**. Den ska *inte* visas vid felaktig inmatning. Testa gärna genom att kommentera bort **op = '?';** i **default**-satsen.

6.5 Spelserien Gissa tal

Här introduceras ett litet enkelt spel som i fortsättningen kommer att utvecklas steg för steg över flera kapitel. I varje version av det kommer vi att lära oss ett nytt koncept. Låt oss kalla det för *Gissa tal*: Användaren ska gissa fram ett hemligt tal inom ett visst intervall. Talet är hårdkodat i de första och slumpat i de senare versionerna av spelet. Som hjälp får användaren reda på inom vilket intervall talet ska ligga samt om det gissade talet var mindre än, större än eller lika med det hemliga talet. För att kunna ge den hjälp användaren behöver för sina gissningar, måste programmet vid varje gissning avgöra vilket fall bland dessa tre alternativ föreligger. Därför är *Gissa tal*-spelet programmeringstekniskt ett exempel på ett trevägsval. Tänkbara utvecklingssteg är: Till att börja med kan det hemliga talet vara en hårdkodad konstant. Sedan kan man gå över till att använda C#:s slumptalsgenerator för att förse hemliga talet med slumptal i ett önskat intervall. Så lär vi oss på köpet hanteringen av slumptal i C#. Önskemålet att kunna upprepa gissningarna tills man gissat rätt och genomföra flera spelomgångar leder till att skriva repetitioner (loopar) i C#. För att kontrollera loopars korrekta avslutning behöver man kunskaper i logik som vi ägnar oss åt i nästa kapitel. Slutligen kommer vi att skriva spelet som en klass. Vi börjar med att lösa problemet med:

Nästlad `if-else`

Denna nästlade kontrollstruktur kan tänkas både som alternativ och komplement till **`switch`**-satsen för att koda flervägsval. Vi kommer att lära oss båda möjligheter. Men först ska vi använda den för att få fram *Gissa tal*-spelet, version 1:

```
// GuessIfElse.cs
// Flervägsval med nästlad if-else-sats
using System;

class GuessIfElse
{
   static void Main()
   {
      Console.Write("\n\tGissa ett tal mellan 1 och 20:\t");
      int guessedNo = int.Parse(Console.ReadLine());

      if (guessedNo <= 17)
         if (guessedNo == 17)
            Console.WriteLine("\u0007\n\tGrattis, du har " +
                                          "gissat rätt!\n");
         else
            Console.WriteLine("\n\tFör litet!\n");
      else
         Console.WriteLine("\n\tFör stort!\n");
   }
}
```

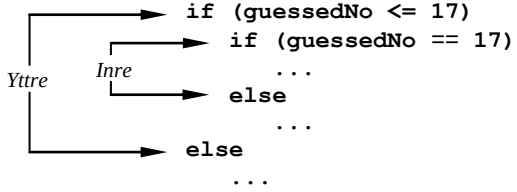
Programmet ovan läser in ett tal, avgör om det är mindre, lika med eller större än 17 och skriver ut det. Så här kan körningar se ut för de tre olika alternativen:

```
Gissa ett tal mellan 1 och 20:  15
För litet!
```

```
Gissa ett tal mellan 1 och 20:  19
För stort!
```

```
Gissa ett tal mellan 1 och 20:  17
Grattis, du har gissat rätt!
```

Samtidigt kommer programmet **GuessIfElse** producera datorljudet när man gissat rätt pga escapesekvensen **\u0007** skickas till konsolen om det inmatade tal som läses in till variabeln **guessedNo**, är **17**. Men låt oss titta hur den nästlade strukturen ser ut:



Vi har en *inre* **if-else**-sats som är nästlad i **if**-delen av en *yttre* **if-else**-sats. Den yttre **if-else**-satsen behandlar de två grupperade alternativen **<= 17** i **if**-delen och alternativet **> 17** i **else**-delen. Den inre **if-else**-satsen tar hand om ”gruppen”, splittrar upp den i sina beståndsdelar **< 17** och == **17**, behandlar == **17** i **if**-delen och **< 17** i **else**-delen. På så sätt återförs trevägsvalet till två tvåvägsval som var och en löses med en **if-else**-sats. Man anar hur komplexiteten växer med större antal alternativ. För att inte råka ut för det s.k. *luriga-***else***-fenomenet* – så kallas det när något **else** paras med ”fel” **if** – låter vi alla **else** hitta ”rätt” **if** genom att skriva **if-else** alltid som *par* och inte hoppa över något **else**. Regeln är att **else** automatiskt paras till närmaste **if**. Man kan jämföra det med parenteser.:Öppnar man en parentes måste man även stänga den.

Kombination av **switch** *och* **if-else**

Även om tomma **case**-satser löser trevägsvalet med olikheter, är det ju inte precis någon elegant lösning att rada upp en massa **case** utan innehåll, särskilt om man

skulle vilja utvidga gissningsintervallets storlek. Alternativt kan problemet lösas med en kombination av **switch** för **if-else**. Då det gäller att skilja mellan de tre alternativen *lika med*, *större än* och *mindre än 17* kan **switch**-satsen testa *likheten* i en **case**-sats. När det är gjort, har man reducerat trevägsvalet till ett tvåvägsval mellan *större än* och *mindre än*. Tvåvägsvalet tar sedan hand om fallen *större än* och *mindre än* i en vanlig **if-else**-sats som kan placeras i **switch**-satsens **default**-del. **case**-satsen behandlar alltså ett fall och **default** de två andra fallen:

```
// GuessSwitch.cs
// Gissa tal-spelet med switch kombinerad med if-else
// Reduktion av trevägsval till tvåvägsval
using System;
class GuessSwitch
{
   static void Main()
   {
      Console.Write("\n\tGissa ett tal mellan 1 och 20:\t");
      int guessedNo = int.Parse(Console.ReadLine());
      Console.Write("\n\t");
      switch (guessedNo)
      {
         case 17:
           Console.Write("\aGrattis, du gissade rätt!\n\n");
           break;
         default:
           if (guessedNo < 17)
              Console.Write("För LITET, försök igen!\n");
           else
              Console.Write("För STORT, försök igen!\n");
           break;
      }
   }
}
```

Programmet ovan ger exakt samma resultat (sid 129) som programmet **GuessIfElse**. Kombinationen av de två kontrollstrukturerna **switch** och **if-else** leder till en avsevärt förenkling och klarhet av koden. Även denna kombination är förstås en slags nästling: **if-else**-satsen är nästlad i **switch**-satsen. Men nästlingen här innebär mindre komplexitet än hos den nästlade **if-else**-satsen. Enkelheten och klarheten i strukturen motiverar användningen av **default**-satsen på ett okonventionellt sätt.

Den egentliga fördelen med den här lösningen är att man tillämpat idén att bryta ned ett stort, svårt problem (trevägsval) till ett mindre, enklare problem (tvåvägsval) vars lösning redan är känd – en metodik som med fördel kan användas även i andra sammanhang. I matematiken är det vanligt att bevisa nya satser (stora, svåra problem) med hjälp av redan kända satser (mindre, enklare problem). I programmeringen används idén om *modularisering* på ett mer genomgripande sätt när man skriver metoder och klasser.

6.6 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*. ”Att låta datorn göra grovjobbet” innebär att låta datorn utföra en repetition. Beroende på hur repetitionens avslutningsvillkor formuleras och var det placeras skiljer man mellan:

Tre typer av repetition*

Efter-testad repetition
För-testad repetition
Bestämd repetition

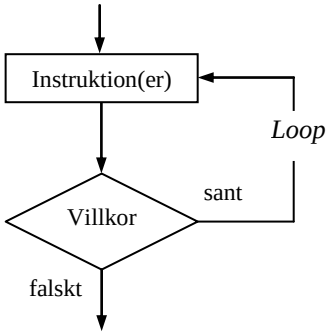
Efter-testad repetition

När avslutningsvillkoret till en upprepningsslinga – även kallad *loop* – testas *efter* loopens instruktioner dvs *efter* det som egentligen ska upprepas, kallas den för **`do`**-satsen. Så här kan den formuleras i pseudokod och som flödesschema:

Pseudokod

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

Flödesschema



I C# inleds den efter-testade repetitionen med det reserverade ordet **`do`**:

```
do
{
    statement(s);
} while (condition);
```

* Ibland används istället för repetition det synonyma begreppet *iteration* som är en fackterm även i andra sammanhang, t.ex. i numerisk analys. Ett besläktat koncept är *rekursion* som är ett alternativ till repetition, men har en annan logisk struktur. Alla rekursiva algoritmer kan skrivas som iterativa.

Första raden är **do**-satsens *huvud* och får inte avslutas med semikolon. Resten är **do**-satsens *kropp* som består av ett block (sid 117). Om kroppen består endast av en sats kan klamrarna **{** och **}** utelämnas. Till skillnad från **if**-satsen kan här kroppens avslutande klammer inte ersätta **do**-satsens avslutande semikolon, då **do**-satsen inte är komplett utan fortsätter med villkoret. Och villkoret kan bara testas när det som vanligt skrivs inom vanliga parenteser som följer det reserverade ordet **while**. Efter villkoret är **do**-satsen komplett vilket bekräftas med det avslutande semikolonet.

Med hjälp av den nya kontrollstrukturen *efter-testad repetition* ska vi nu skriva en **do**-sats. Som applikation tar vi *Gissa tal* för att vidareutveckla det. Den stora nackdelen av alla tre versioner hittills var att man inte kunde gissa flera omgånger utan var tvungen att köra om programmet för att gissa vidare. Följande program använder en **do**-loop för att kunna köra programmet tills man gissat rätt:

```
// GuessDo.cs
// Gissa tal-spelet i dialog med do-loop
using System;

class GuessDo
{
  static void Main()
  {
    int guessedNo;
    do
    {
       Console.Write("\n\tGissa ett tal mellan 1 och 20:\t");
       guessedNo = int.Parse(Console.ReadLine());
       Console.Write("\n\t");
       switch (guessedNo)
       {
         case 17:
           Console.Write("\aGrattis, du gissade rätt!\n\n");
           break;
         default:
           if (guessedNo < 17)
              Console.Write("För LITET, försök igen!\n");
           else
              Console.Write("För STORT, försök igen!\n");
           break;
       }
    } while (guessedNo != 17);
  }
}
```

do-satsen är en lämplig variant av repetition när det gäller att åstadkomma en dialog mellan datorn och användaren. I **GuessDo** inleds dialogen med inläsning av **guessedNo**. Sedan tar **switch**-satsen hand om valet mellan tre alternativ, nämligen om det gissade talet är lika med, mindre än eller större än spelets hemliga tal **17**. I slutet testas om **guessedNo** är skilt från **17**. Om så är fallet, återvänder pro-

gramflödet till början av **do**-blocket och allt upprepas tills **guessedNo** någon gång blir lika med **17**. Nu kan vi göra flera gissningar vid endast en körning. Programmet avslutas först när vi hittat det hemliga talet. En körning av programmet **GuessDo** kan t.ex. ge följande dialog:

```
Gissa ett tal mellan 1 och 20: 5

        För LITET, försök igen!

Gissa ett tal mellan 1 och 20: 15

        För LITET, försök igen!

Gissa ett tal mellan 1 och 20: 19

        För STORT, försök igen!

Gissa ett tal mellan 1 och 20: 17

        Grattis, du gissade rätt!
```

I **do**-satsen utförs satserna första gången oavsett om villkoret är sant eller falskt. Sedan testas villkoret: är det sant upprepas satserna. Sedan testas villkoret igen: är det fortfarande sant, fortsätts repetitionen osv. Är villkoret falskt, stoppas repetitionen. Man kan alltså säga: dörrvakten (villkoret) står vid utgången till lokalen (loopen). Konsekvensen blir att, när villkoret är falskt från början, kommer satserna i alla fall att utföras åtminstone *en* gång. I nästa avsnitt behandlas en annan variant av repetition, den för-testade repetitionen där dörrvakten så att säga står vid ingången till lokalen och inte tillåter att någon sats exekveras när villkoret är falskt från början. Är villkoret sant hela tiden, kommer loopen att snurra i all evighet. Därför kallas den *evighetsloop* (sid 138).

Det är avgörande att skilja mellan *repetition* och *selektion*. I selektionens pseudokod har vi nyckelordet **OM** och i C# det reserverade ordet **if**, vilket innebär ett val *en enda* gång, dvs ingen upprepning alls. I repetitionens pseudokod har vi **SÅ LÄNGE** och i C# det reserverade ordet **while**, vilket innebär att villkoret testas *upprepade gånger*. I selektionens flödesschema går allt flöde endast framåt dvs alla pilar nedåt, se sid 113, 121 och 123. I repetitionens flödesschema ovan går pilen efter instruktionerna *tillbaka* till villkoret för att testa det igen. Orsaken till att programflödet går tillbaka är att det finns en *hoppats* inbyggd i alla repetitioner som skickar programflödet tillbaka till loopens villkor. Ett annat sätt att se på efter-testad repetition är att i pseudokoden (sid 131) använda nyckelordet **TILLS** istället för **SÅ LÄNGE**. Så kan logiken ibland uppfattas enklare:

REPETERA
instruktion(er)
TILLS *villkor* **inte** uppfyllt

Om man väljer samma villkor som i formuleringen med **SÅ LÄNGE**, dvs bibehåller villkorets formulering, måste man negera villkoret när man går över till **TILLS**. Det

beror på skillnaden i den logiska innebörden av **SÅ LÄNGE** och **TILLS**. I flödesschemat av den efter-testade repetitionen blir det ingen strukturell ändring, bara man sätter sant och falskt på de logiskt korrekta utgångarna av villkoret.

Hantering av slumptal

En nackdel av programmet **GuessDo** är att det hemliga talet är hårdkodat som **17**. Det skulle innebära en väsentlig förbättring av *Gissa tal* om programmet kunde generera ett slumptal mellan 1 och 20 som hemligt tal varje gång man körde det. Därför öppnar vi här en liten parentes om slumptal av typ **int** och deras hantering.

Generellt kan man med datorn som en deterministisk maskin som datorn är, inte producera äkta slumptal utan endast *simulera* dvs på något sätt *beräkna* s.k. *pseudo*slumptal enligt en viss algortim. Överallt vi pratar om slumptal menar vi egentligen pseudoslumptal. I C# kan man simulera slumptal på olika sätt, bl.a. med klassen **Random** och dess metod **Next()** som returnerar slumptal av typ **int** mellan **1** och **int.MaxValue**, om den anropas utan parameter. En annan variant av **Next()** returnerar slumptal mellan sina parametrar, närmare bestämt:

```
a <= r.Next(a, b) < b
```

där **r** är ett objekt klassen **Random**. För att skräddarsy metoden **Next(a, b)** till att få slumptal mellan **1** och **20** måste vi anropa **r.Next(1, 21)**. Följande program testar båda varianter av **Next()**:

```
// DoRand.cs
// Skriver ut 5 slumptal mellan 1 och int.MaxValue samt
//            20         mellan 1 och 20
// Anropar två varianter av Random-metoden Next() en gång
// med ingen parameter, en gång med två paramtrar
using System;
class DoRand
{
  static void Main()
  {
    int i = 1, j = 1;
    Random r = new Random(); // Objekt av klassen Random

    Console.WriteLine("Slumptal mellan 1 & int.MaxValue:\n");
    do                       // do-loop
       Console.WriteLine("\t" + r.Next());
    while (i++ < 5);         // i testas först, ökar sedan

    Console.WriteLine("\nSlumptal mellan 1 och 20:\n\t");
    do                       // do-loop
       Console.Write(r.Next(1, 21) + "\t");
    while (j++ < 20);        // j testas först, ökar sedan

    Console.WriteLine('\n');
  }
}
```

En körning av **DoRand** ger följande resultat:

```
Slumptal mellan 1 & int.MaxValue:

        1460841191
        225482400
        1438321568
        1700127070
        1513406452

Slumptal mellan 1 och 20:

7     20    2     12    12    14    3     16    3     15
2     15    12    9     1     10    14    15    1      2
```

För det första ser man att vi får endast heltal vilket beror på att båda metoderna **Next()** och **Next(a, b)** returnerar **int**. Vill man ha decimalslumptal finns det en annan metod i klassen **Random** som heter **NextDouble()**. För det andra har vi fått i intervallet [1, 20] även randvärdena **1** och **20**. Hade vi anropat **r.Next(1, 20)** hade vi fått slumptal mellan **1** och **19** eftersom den andra parametern *inte* ingår i slumptalsgenereringen. Så, anropet **r.Next(1, 21)** ger slumptal mellan **1** och **20**.

När det gäller de båda varianterna av metoden **Next()** ger den ena utan parameter de stora slumptalen i utskriften ovan mellan **1** och **int.MaxValue** och den andra med två parametrar de små slumptalen mellan **1** och **20**. Två olika **do**-satser i **DoRand** tar hand om slumptalen i dessa två olika intervall. I den första **do**-satsen anropas **Next()** utan parameter, i den andra med två parametrar. Vi har här att göra med ett koncept i programmering som kallas *överlagring av metoder* som är en generalisering av överlagringen av operatorer som vi behandlat tidigare (sid 74). Innebörden är att det är *två olika metoder* med *samma namn*, men olika parameterlistor. I anropet avgörs vilken av dem det gäller därför att parameterlistan avslöjar identiteten – både för oss och kompilatorn. C#-biblioteket är fullt med överlagrade metoder. De flesta bibliotheksklasserna har t.o.m. flera överlagrade metoder dvs *flera* olika metoder med samma namn.

Gissa tal med slumptal

Resultatet från **DoRand** kan vi nu använda i nästa version av *Gissa tal* för att slumpa fram det hemliga talet varje gång vi kör och kunna spela tills vi gissat rätt (sid 128).

Spelet har förbättrats i två avseenden: För det första bestäms programmets hemliga tal inte längre redan i koden utan slumpas fram med **Random**-metoden **Next()** enligt ovan. För det andra behöver man inte vänta tills man gissat rätt för att avsluta programkörningen, utan kan avsluta innan man hunnit gissa rätt: Man matar in **0** och får samtidigt reda på spelets hemliga tal som är olika för varje körning pga användningen av slumptal.

Följande program implementerar *Gissa tal*-spelet med slumptal:

```
// GuessDoRand.cs
// Gissa tal-spelet med slumptal i dialog med do-loopen
using System;

class GuessDoRand
{
   static void Main()
   {
      int guessedNo;
      Random r = new Random();
      int secretNo = r.Next(1, 21);

      do
      {
        Console.Write(
        "\n\tGissa ett tal mellan 1 och 20" +
                                    " (Avsluta med 0):\t");
        guessedNo = int.Parse(Console.ReadLine());
        Console.Write("\n\t");
        if (guessedNo == 0)
        {
           Console.WriteLine("Avbrott: Programmets " +
                     " hemliga tal var " + secretNo + '\n');
           break;   // Bryter do-loopen
        }
        if (guessedNo == secretNo)
        {
           Console.Write("\aGrattis, du har gissat " +
                                                "rätt!\n\n");
           break;   // Bryter do-loopen
        }
        if (guessedNo < secretNo)
           Console.Write("För LITET, försök igen!\n");
        else
           Console.Write("För STORT, försök igen!\n");
      } while (guessedNo != secretNo);
   }
}
```

Programmets första **if**-sats bryter då **do**-loopen med hjälp av **break**. När vi behandlade **switch**-satsen sade vi att **break** är ett reserverat ord som bryter programflödet även i loopar (sid 127). Och det är precis vad den gör här. **break** bryter **do**-satsen utan att testa **do**-satsens avslutningsvillkor **(guessedNo != secretNo)** som i regel – dvs när **break** inte utförs – kommer till användning och avslutar dialogen när man gissat rätt. Annars fortsätter dialogen så länge man gissar fel.

Frågan som dyker upp när man tittar på koden i programmet **GuessDoRand**, är: Varför används inte längre **switch** i kombination med **if-else** som i den senaste versionen av *Gissa tal* hade gett bra resultat. Vi hade helst velat göra det. Men övergången till slumptal gör att slumptalet måste lagras i en variabel – i det här fallet **secretNo** – och **switch**-satsen inte tillåter jämförelse med en variabel. I spelets första versioner var programmets hemliga tal hårdkodat som konstanten **17** och **switch** kunde jamföra *expression* **guessedNo** med denna konstant. Men nu lagras det hemliga talet i variabeln **secretNo**. Den allmänna strukturen:

```
switch (expression)
{
    case constant1 :
      .
      .
```

sätter stopp för användningen av **switch** i **GuessDoRand** därför att *expression* kan vara en variabel av typ **int** eller **char** – i vårt fall är **guessedNo** en **int**-variabel (det är ok) – medan *constant1* måste vara ett konstant uttryck, annars kan man inte kompilera. I vårt fall är **secretNo** som skulle skrivas efter **case**, inget konstant uttryck utan en **int**-variabel, vilket inte är ok. Vi stöter här på **switch**-satsens begränsningar. Därför används i **GuessDoRand** en enkel **if**- samt en **if-else**-sats för att avgöra trevägsvalet ”**guessedNo** lika med, mindre eller större än **secretNo**”. En körning ger:

```
Gissa ett tal mellan 1 och 20 (Avsluta med 0):  10
För LITET, försök igen!
Gissa ett tal mellan 1 och 20 (Avsluta med 0):  15
För STORT, försök igen!
Gissa ett tal mellan 1 och 20 (Avsluta med 0):  12
För LITET, försök igen!
Gissa ett tal mellan 1 och 20 (Avsluta med 0):  13
För LITET, försök igen!
Gissa ett tal mellan 1 och 20 (Avsluta med 0):  14
Grattis, du har gissat rätt!
```

Har man efter ett tag ingen lust att gissa vidare och vill avsluta, kan man mata in **0**. Man får då reda på programmets hemliga slumptal vid just den aktuella körningen:

```
Gissa ett tal mellan 1 och 20 (Avsluta med 0):  0
Avbrott: Programmets hemliga tal var 20
```

do-satsen är en lämplig variant av repetition för dialoger mellan dator och användare. Frågan om vilken variant av repetition man ska välja, kan inte besvaras generellt, eftersom det är det konkreta problemet som avgör valet.

Innan vi avslutar **do**-satsen vill vi nämna en företeelse som man kan råka ut för när man jobbar med loopar:

Evighetsloop

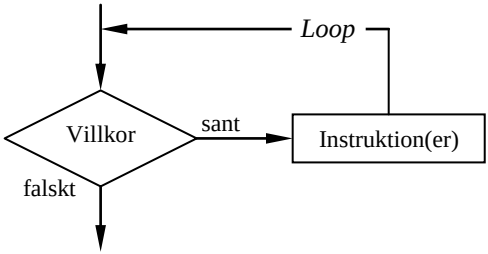
I det inledande exemplet **DoRand** är den första **do**-satsens avslutningsvillkor **i++ < 5**. Detta innebär att det egentliga villkoret är **i < 5** och att efter villkorets test, **i**:s värde ökas med **1**. Ersätts avslutningsvillkor med **i < 5** avslutas loopen och därmed programmet aldrig: Man har råkat ut för en s.k. *evighetsloop*. Orsaken är att villkoret är sant från början – **i** har ju initierats till **1** – och förblir sant hela tiden, så loopen fortsätter i all evighet. Generellt innehåller en loop alltid *möjligheten* till en evighetsloop. För att undvika den, måste villkoret och satserna i kroppen formuleras på ett sådant sätt att villkorets sanningsvärde *ändras*, så att villkoret *blir* falskt efter några varv. Detta krav har realiserats i programmet **DoRand** genom att använda **i++**. Dvs, har man med en lämplig initiering av **i** kommit in i **do**-satsen, kommer **i** att öka med **1** i varje varv så att det någon gång blir = **5**. Då stoppas loopen. Glömmer man ökningen **++** och initierar **i** med ett värde mindre än **5** blir **do**-satsen en evighetsloop. Omvänt: Är **do**-villkoret falskt från början, görs ingenting. Initieras **i** till ett värde **>= 5**, blir villkoret falskt från början och man kommer aldrig in i kroppen (”aldrigloop”). Programflödet fortsätter vid första satsen *efter* **do**-loopen. Testa gärna dessa möjligheter.

6.7 För-testad repetition: `while`-satsen

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

Pseudokod och ritas som Flödesschema

SÅ LÄNGE *villkor* uppfyllt
instruktion(er)



I C# inleds för-testad repetition med det reserverade ordet **`while`** och skrivs så här:

```
while (condition)
{
    statement(s);
}
```

Första raden är *huvudet* och får inte avslutas med semikolon, om man inte vill ha en tom **`while`**-sats. Resten är **`while`**-satsens *kropp* som omsluts av klamrarna **{** och **}**. Om kroppen består endast av en sats kan klamrarna utelämnas.

```
// Ascii.cs
// Skriver ut en del av teckentabellen med en while-loop
using System;

class Ascii
{
   static void Main()
   {
      int code = 33;
      while (code <= 256)
      {
         Console.Write(code + " " + (char) code + '\t');
         if (code % 8 == 0)           // Var 8:e utskrift:
            Console.WriteLine();      // Radbyte
         code++;
      }
      Console.WriteLine("\nEfter while-satsen är code = " +
                                                code + '\n');
   }
}
```

Programmet **Ascii** visar ett exempel på **while**-satsen med tre satser i kroppen som skriver ut följande del av **Ascii**-tabellen till konsolen. Kroppens avslutande klammer kan ersätta det semikolon som skulle avsluta hela **while**-satsen.

```
33 !    34 "    35 #    36 $    37 %    38 &    39 '    40 (
41 )    42 *    43 +    44 ,    45 -    46 .    47 /    48 0
49 1    50 2    51 3    52 4    53 5    54 6    55 7    56 8
57 9    58 :    59 ;    60 <    61 =    62 >    63 ?    64 @
65 A    66 B    67 C    68 D    69 E    70 F    71 G    72 H
73 I    74 J    75 K    76 L    77 M    78 N    79 O    80 P
81 Q    82 R    83 S    84 T    85 U    86 V    87 W    88 X
89 Y    90 Z    91 [    92 \    93 ]    94 ^    95 _    96 `
97 a    98 b    99 c    100 d   101 e   102 f   103 g   104 h
105 i   106 j   107 k   108 l   109 m   110 n   111 o   112 p
113 q   114 r   115 s   116 t   117 u   118 v   119 w   120 x
121 y   122 z   123 {   124 |   125 }   126 ~   127 ⌂   128 ?
129 ?   130 ?   131 ?   132 ?   133 ?   134 ?   135 ?   136 ?
137 ?   138 ?   139 ?   140 ?   141 ?   142 ?   143 ?   144 ?
145 ?   146 ?   147 ?   148 ?   149 ?   150 ?   151 ?   152 ?
153 ?   154 ?   155 ?   156 ?   157 ?   158 ?   159 ?   160
161 ¡   162 ¢   163 £   164 ¤   165 ¥   166 ¦   167 §   168 ¨
169 ©   170 ª   171 «   172 ¬   173 -   174 ®   175 ¯   176 °
177 ±   178 ²   179 ³   180 ´   181 µ   182 ¶   183 ·   184
185 ¹   186 º   187 »   188 ¼   189 ½   190 ¾   191 ¿   192 À
193 Á   194 Â   195 Ã   196 Ä   197 Å   198 Æ   199 Ç   200 È
201 É   202 Ê   203 Ë   204 Ì   205 Í   206 Î   207 Ï   208 Ð
209 Ñ   210 Ò   211 Ó   212 Ô   213 Õ   214 Ö   215 ×   216 Ø
217 Ù   218 Ú   219 Û   220 Ü   221 Ý   222 Þ   223 ß   224 à
225 á   226 â   227 ã   228 ä   229 å   230 æ   231 ç   232 è
233 é   234 ê   235 ë   236 ì   237 í   238 î   239 ï   240 ð
241 ñ   242 ò   243 ó   244 ô   245 õ   246 ö   247 ÷   248 ø
249 ù   250 ú   251 û   252 ü   253 ý   254 þ   255 ÿ   256 Ā

Efter while-satsen är kod = 257
```

ASCII-tabellen med **while**

När vi i kapitel 5 tog upp ASCII-tabellen kunde vi i programmen **Int2char**, **Char2int** och **Unicode** med hjälp av explicit typkonvertering få reda på *enskilda* teckens koder och omvänt. Nu när vi kan hantera loopar kan vi skriva ut delar av teckentabellen i ett sammanhängande kodintervall. Programmet **Ascii** skriver ut i en **for**-loop både tecken och tillhörande kod genom att använda en **int**-variabel **code** som räknare och få ut resp. tecken genom explicit typkonvertering från **int** till **char**: Variabeln **code** initieras till **33**. **while**-satsen börjar med att testa villkoret **code <= 256**. Är det sant utförs kroppens satser. Därför hamnar **33 !** som allra första i utskriften ovan. Sedan testas villkoret igen: Är det fortfarande sant, utförs satserna igen osv. Detta upprepas gång på gång. Sist skrivs ut **256 ?** därför att **256** är variabeln **code**:s sista värde som fortfarande uppfyller villkoret **code <= 256**. I nästa varv då **code** hunnit bli **257** är villkoret inte längre uppfyllt och **while**-loopen stoppas. När vi *efter* **while**-satsen skriver ut **code** får vi **257**.

En jämförelse av **Ascii**-programmets utskrift på förra sidan med ASCII-tabellen på sid 93 visar överensstämmelse i standard ASCII-koderna upp till **127**. Resten är icke standardiserade koder. Men hur åstadkommer **while**-satsen utskriften? Låt oss gå igenom det varv för varv.

Då **code** i första varvet är **33** och därmed mindre än **256**, kommer vi in i **while**-satsen och får utskriften **33 !** följd av en tabultaor. Att det blir **!** och inte **33** beror på att vi i utskriftssatsen med explicit typkonvertering omvandlat **int**-variabeln **code**:s värde till **char** (sid 94). Sedan följer **if**-satsen med villkoret **code % 8 == 0**. Det här villkoret är falskt då **code**:s värde **33** modulo **8** ger **1** dvs inte **0** (sid 75). Därför utförs inte **if**-satsens kropp dvs inget radbyte skrivs ut. Efter **if**-satsen utförs uppdateringen **code++** så att **code** blir **34**.

Efter **while**-satsens första varv går programflödet tillbaka till villkoret **code <= 256**. Då är **code**:s värde **34** som jämförs med **256**. Då **34** är mindre än **255**, kommer vi igen in i **while**-satsens andra varv. **33 "** skrivs ut följt av en tabultaor. Då **if**-satsens villkor fortfarande är falskt – **34** modulo **8** ger **2** som inte är **0** – utförs kroppen inte heller den här gången: inget radbyte. I slutet av loopens andra varv uppdateras **code**:s värde till **35**.

Allt detta upprepas på samma sätt även i **while**-satsens 3:e, 4:e, 5:e, 6:e och 7:e varv. Vi kommer så långt då även **38** och **39** är mindre än **256**. I det 8:e varvet har **code** hunnit bli **40**. Då skrivs ut **40 (** följt av en tabultaor. Men nu är för första gången **if**-satsens villkor **code % 8 == 0** sant, eftersom **code**:s värde, **40** modulo **8** ger **0**. Därför utförs **if**-satsens kropp: **Console.WriteLine()** vilket innebär radbyte. Observera också att utskrifterna *längs en rad* görs med **Console.Write()** vilket innebär utskrift *utan* radbyte. I slutet på loopens 8:e varv uppdateras **code**:s värde till **41**.

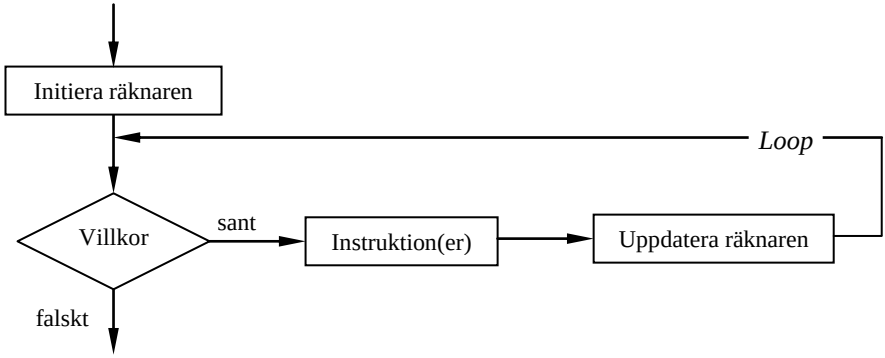
På den nya, andra raden i utskriften – i loopens 9:e varv – skrivs ut **41)** följt av en tabulator. Därefter inget radbyte då, pga **41** modulo **8** är **1**. Utskriften fortsätter på den andra raden utan radbyte tills **code**:s värde nått **48**. För andra gången blir **if**-satsens villkor sant, när **code** är **48** för **48** modulo **8** ger **0**. Mönstret har klarnat: **if**-satsens roll är att producera radbyte när **code**:s värde är jämnt delbart med **8** dvs var åttonde utskrift. Detta är just innebörden i villkoret **code % 8 == 0**. Vi har gjort så för att få en tabellartad utskrift.

while-satsen avslutas när villkoret **code <= 256** blivit falskt dvs när **code** nått **257** som är varken mindre eller lika med utan större än **256**. Därför stoppas repetitionen. Efter det skrivs **code**:s sista värde **257** ut för kontroll.

6.8 Bestämd repetition: `for`-satsen

`for`-satsen är en upprepningsloop där antalet repetitioner är känt i förväg. I de hittills behandlade varianterna – för- och efter-testad repetition – styr endast villkoret antalet repetitioner och man kan få reda på antalet repetitioner efter att ha kört programmet dvs i efterhand. I den bestämda repetitionen kan programmeraren redan vid kodningen *bestämma* antalet repetitioner. Det är användbart i de fall då man vet hur många gånger en sak ska upprepas. Visserligen finns även i den bestämda repetitionen ett villkor som testas i varje varv av loopen, men det finns även en inbyggd möjlighet att styra villkoret och därmed antalet repetitioner med hjälp av en *räknare*, även kallad *styrvariabel*.

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 repetitioner fastlagt, t.ex. till slutvärde minus startvärde om räknaren ökar med 1. Om villkoret är uppfyllt, t.ex. om räknaren är mindre än slutvärdet, utförs ett antal instruktioner. Sedan görs en *uppdatering* av räknaren, ofta en ökning med 1, men det är möjligt att räkna nedåt eller välja ett annat steg än 1. Allt detta händer i varje varv av repetitionen. Så här ser den bestämda repetitionens flödesschema ut:



Flödesschemat åskådliggör den *logiska strukturen* av problemet. Den bestämda repetitionens pseudokod blir enligt flödesschemat ovan:

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

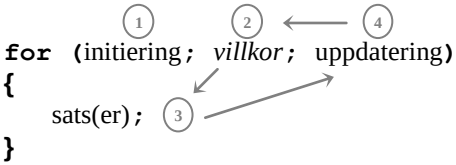
I några äldre programspråk som t.ex. Basic, Fortran och Pascal, finns endast detta specialfall, där villkoret implicit (dvs underförstått) är inbyggt och räknarens uppdatering sker automatiskt. Detta specialfall kan beskrivas med följande pseudokod:

STEGA räknaren **FRÅN** startvärde **TILL** slutvärde (med **STEG**)
instruktion(er)

Då det inbyggda villkoret *räknare* ≤ *slutvärde* inte syns i pseudokoden – därför *implicit* – kan det inte heller ändras. I flödesschemat blir det ingen strukturell ändring om man tar över detta villkor.

Nyckelordet **SÅ LÄNGE** i pseudokoden på förra sidan visar att den bestämda repetitionen alltid kan översättas till en **while**-sats om man själv tar hand om räknaren. Längre fram ska vi ge exempel på översättningar från **do** och **while** till **for**. Precis som i **while**-satsen har man i princip friheten att formulera villkoret hur som helst. Men då räknaren är inbyggd i **for**- till skillnad från **while**-satsen, kan man i villkoret jämföra räknaren med något slutvärde, t.ex. så här: *"räknare är mindre än eller lika med slutvärde"*. Detta ger ett specialfall av den bestämda repetitionen.

I C# inleds bestämd repetition med det reserverade ordet **for** och skrivs så här:



Första raden är **for**-satsens *huvud* och får inte avslutas med semikolon. Resten är **for**-satsens *kropp* som omsluts av klamrarna **{** och **}**. Kroppens avslutande klammer kan ersätta det semikolon som skulle avsluta hela **for**-satsen. Om kroppen endast består av en sats kan klamrarna utelämnas. Jämför man C#-koden med flödesschemat på förra sidan kan man konstatera att koden är lite kryptisk i den bemärkelsen att den inte följer flödesschemats struktur *initiering – villkor – sats(er) – uppdatering*. Därför har vi i koden ovan numrerat **for**-satsens olika delar för att visa i vilken ordning de utförs. Pilarna markerar loopens förlopp. Initieringen görs endast en gång och ingår ej i loopen. Både initieringen och uppdateringen, avser räknaren som är en vanlig variabel och därför måste definieras precis som vilken variabel som helst.

I följande program har vi modifierat programmet **DoRand** genom att skriva om **do**-satserna till **for**-satser (sid 134):

```
// ForRandom.cs
// Skriver ut 4 slumptal mellan 1 och int.MaxValue samt
//           19          mellan 1 och 20
using System;

class ForRandom
{
   static void Main()
   {
      Random r = new Random();

      Console.WriteLine("Slumptal mellan 1 och " +
                                          " int.MaxValue:\n");
      for (int i = 1; i < 5; i++)           // i gäller bara i
          Console.WriteLine("\t" + r.Next());  // denna for-
                                               // sats


      Console.WriteLine("\nSlumptal mellan 1 och 20:\n\t");
      for (int i = 1; i < 20; i++)         // Ny lokal var. i
          Console.Write(r.Next(1, 21) + "\t");

      Console.WriteLine('\n');
   }
}
```

Frågan är nu: Blir det samma resultat som i programmet **DoRand**? Gör de två **do**-saterna där samma sak som motsvarande **for**-satserna här? En körning av programmet ovan producerar följande resultat vilket visar att det finns en viktig skillnad till **DoRand**-utskriften på sid 135 bortsett från de annorlunda slumptalsvärdena:

```
Slumptal mellan 1 och int.MaxValue:

        3107148
        735561933
        153248854
        1692537805

Slumptal mellan 1 och 20:

1     7     10    10    7     4     8     19    6     10
18    20    14    3     7     1     15    17    7
```

Den avgörande skillnaden är att det nu skrivs ut 4 medan då fanns 5 slumptal mellan **1** och **int.MaxValue** och att det nu skrivs ut 19 medan då fanns 20 slumptal mellan **1** och **20**. Låt oss nu jämföra looparnas koder lite närmare med varandra för att få reda på orsaken till denna skillnad. Låt oss ta t.ex. den första **do**-satsen i **DoRand** (sid 134) :

```
do
   Console.WriteLine("\t" + r.Next());
while (i++ < 5);
```

Vi jämför denna **do**-loop med den första **for**-loopen i programmet **ForRandom**:

```
for (int i = 1; i < 5; i++)
    Console.WriteLine("\t" + r.Next());
```

Det står exakt samma sats i deras resp. kroppar: Den skriver ut en tabulator samt ett slumptal genom konkatenering. Men frågan är: hur många gånger sker detta dvs hur många varv har resp. loop? Frågan besvaras inte av kroppen utan av villkoret och programflödet, dvs i vilken ordning villkorets test och räknarens uppdatering genomförs.

I **do**-satsen testas räknaren **i** först och ökar sedan pga **i++** dvs ökningsoperatorns *post*fixvariant, vilket innebär: När räknaren är **4** i loopens 4:e varv, har pga **do**-loopens efter-testade karaktär, redan fyra slumptal skrivs ut innan villkoret **4 < 5** testas. Då fortsätter loopen och räknaren uppdateras till **5**. Loopen kommer in i sitt 5:e varv och utför kroppen. Sedan avslutas loopen då **5 < 5** ger **false**. Alltså genomgår **do**-loopen *fem* varv och skriver ut *fem* slumptal. I **for**-satsen däremot händer pga strukturen *initiering – villkor – sats(er) – uppdatering* (förra sidan) följande: När **for**-loopen inleder sitt 4:e varv med testet **4 < 5** har pga **for**-loopens förtestade karaktär skrivits ut först 3 slumptal, vilket förstås även framgår av **for**-satsens flödesschema (förförra sidan). Då går programflödet först ”ned” till kroppen och skriver ut det 4:e slumptalet, innan räknaren hinner bli 5. Sedan testas villkoret **5 < 5** som ger **false**. Därför avslutas loopen. Alltså genomgår **for**-loopen *fyra* varv och skriver ut *fyra* slumptal. Det är den avgörande skillnaden mellan **do**- och **for**-satserna ovan och därmed mellan programmen **DoRand** och **ForRandom**. Därför producerar det första *fem* medan det andra endast *fyra* slumptal. Vill man att **DoRand** producerar fyra utskrifter behöver man bara byta ut *post*fixvarianterna **i++** och **j++** mot *pre*fixvarianterna **++i** och **++j** i **do**-looparnas avslutningsvillkor. Omvänt: Vill man att **ForRandom** producerar fem utskrifter behöver man bara byta ut **i < 5** mot **i <= 5** i **for**-looparnas avslutningsvillkor. Testa gärna! Ganska liknande resonemang förklarar varför **ForRandom** skriver ut 19 slumptal medan mellan **1** och **20**, medan **DoRand** gör det 20 gånger.

Räckvidden av `for`*-satsens räknare*

En annan företeelse som man kan observera när man jämför **do** i **DoRand** med **for** i **ForRandom** är att vi i första fallet behövde *två olika* variabler **i** och **j** som räknare, en i varje **do**-sats, medan det i andra fallet räckte med *en* variabel **i** som räknare i båda **for**-satserna. Frågan är nu: Är det i fallet **for** verkligen en och samma variabel eller är det bara ett och samma *namn* för två olika variabler? Nästa fråga: Om det är så, blir det inte namnkonflikt? Får man definiera två olika variabler med samma namn? Svaret är: För det första är det faktiskt *ett* namn för *två olika* variabler. För det andra, kan man göra så eftersom båda **for**:s räknare i

programmet **ForRandom** är definierade *inuti* **for**-satserna. Det gäller nämligen i C# följande regel:

> **for**-satsens räknare är odefinierad efter **for**-satsen om den definieras inuti **for**-satsen.

Variabeln **i** är definierad så att säga *lokalt* i **for**-satsen:

```
for (int i = 1; i < 5; i++)
    Console.WriteLine("\t" + r.Next());
```

Variabeln **i** är inte definierad och gäller därför inte i hela programmet utan endast i **for**-satsen, därför *lokalt*. Efter **for**-satsen ”dör” variabeln **i**. Varje försök att referera till den *efter* **for**-satsen kommer att leda till kompileringsfel. Därför är det möjligt att i nästa **for**-sats definiera räknaren med samma namn **i**. **for**-satsens inre variabler är inte synliga utåt i enlighet med C#:s generella regler om lokala variabler. Vill man inte ha det så, måste man definiera räknaren *före* **for**-satsen:

```
int i;
for (i = 1; i < 5; i++)
    Console.WriteLine("\t" + r.Next());
```

Eller:

```
int i = 1;
for (; i < 5; i++)
    Console.WriteLine("\t" + r.Next());
```

Då kommer **i** vara även giltig *efter* **for**-satsen och vi skulle kunna referera till den, t.ex. skriva ut värdet. Då kommer det inte längre vara möjligt att använda namnet på nytt i resten av programmet.

6.9 Nästlade `for`-satser

Nästlade **`for`**-satser är ett viktigt verktyg i alla programmeringsspråk för att bearbeta ordnade tvådimensionella strukturer. Tabeller och rektangulära scheman är exempel på sådana 2D-strukturer. Följande program skriver ut tal i en tabell genom att nästla två **`for`**-satser i varandra:

```
// NestedFor.cs
// Skriver ut en tabell över tal med 6 rader och 8 kolumner
// Nästlad for-sats: En inre for-loop nästlas i en yttre
// Radbyte mellan den yttre och inre loopen
using System;

class NestedFor
{
  static void Main()
  {
    for (int row = 1; row <= 6; row++)     // Yttre loop skri-
    {                                      // ver ut 6 rader
       for (int column=1; column<=8; column++)  // Inre loop
       Console.Write("  " + row);          // skriver ut 8
//         Console.Write("  " + column);    // tal i en rad
       Console.WriteLine();                // Radbyte mellan
    }                                      // yttre och inre
  }
}
```

Utskriften till vänster får man när man kör den *aktuella* koden ovan. Utskriften till höger fås om man kör koden med den *bortkommenterade* raden istället för raden ovanpå.

```
  1  1  1  1  1  1  1  1
  2  2  2  2  2  2  2  2
  3  3  3  3  3  3  3  3
  4  4  4  4  4  4  4  4
  5  5  5  5  5  5  5  5
  6  6  6  6  6  6  6  6
```

```
  1  2  3  4  5  6  7  8
  1  2  3  4  5  6  7  8
  1  2  3  4  5  6  7  8
  1  2  3  4  5  6  7  8
  1  2  3  4  5  6  7  8
  1  2  3  4  5  6  7  8
```

Med andra ord, satsen **`Console.Write("  " + row);`** ger utskriften till vänster. Ersätts däremot variabeln **`row`** med **`column`** så att den bortkommenterade satsen **`Console.Write("  " + column);`** körs istället får man utskriften till höger. För att förstå varför det blir så, låt oss börja med att undersöka hur den vänstra utskriften kommer till: När vi nästlade **`if-else`**-satser i varandra (sid 128) pratade vi om en *inre* **`if-else`**-sats som nästlas i en *yttre*. Samma sak är det i programmet **`NestedFor`** på förra sidan: Den nästlade **`for`**-satsen, även kallad *dubbel* **`for`**-sats, består av två slingor: Vi har en *inre* **`for`**-loop som nästlas i en *yttre*. Den yttre **`for`**-

loopen omfattar två satser, för det första den inre **for**-loopen och för det andra en utskriftssats som gör radbyte. Därför är dessa satser omslutna av klamrar. Den inre **for**-loopen

```
for (int column=1; column<=8; column++)
    Console.Write("  " + row);
```

skriver ut i sina 8 varv den första raden av **1**-orna (med två mellanslag däremellan) som syns i den vänstra utskriften på förra sidan. Variabeln **row** som är den *yttre* **for**-loopens räknare, har nämligen under *alla* dessa 8 inre varv, värdet **1** därför att vi då hela tiden befinner oss i den *yttre* **for**-loopens första varv. När alla 8 inre varv är slutförda och villkoret **column<=8** sätter stopp för den inre loopen fortsätter programflödet till nästa sats som följer i den yttre **for**-satsen. Det är den som lägger till radbytet i utskriften. Sedan uppdateras räknaren **row** till **2** och det hela upprepas: Raden av **2**-orna (med två mellanslag) skrivs ut i den vänstra utskriften på förra sidan osv. Detta pågår tills den yttre **for**-loopens villkor **row<=6** sätter stopp för den. Därför får vi 6 rader utskrivna där i varje rad den yttre räknaren **row**:s värde syns.

Om vi nu tar den högra utskriften på förra sidan och undersöker hur den kommer till kan vi konstatera att den är enklare att förstå, för det är den inre **for**-loopen

```
for (column=1; column<=8; column++)
    Console.Write("  " + column);
```

som är ansvarig för den, efter att vi aktiverat den bortkommenterade raden, gjort den till den inre **for**-loopens kropp och kommenterat bort raden ovanför. Det blir enklare då loopens räknare **column** är samtidigt den variabel vars värde skrivs ut. Så det är inte några variabler från den yttre loopen som är involverade här: Endast räknaren **column**:s värden **1-8** hamnar i den högra utskriften på förra sidan. Rad för rad kommer de ut skilda med radbyte, **6** gånger sammanlagt pga den yttre loopens huvud:

```
for (row=1; row<=6; row++)
```

I båda utskrifternas fall skriver programmet ut tabellen *radvis*. På så sätt uppstår en rektangulär utskrift av tal bestående av 6 rader och 8 kolumner.

Programmet **NestedFor** är ett exempel på följande generell regel:

> Regel för nästlade **for**-satser:
>
> I nästlade **for**-satser måste den inre **for**-loopen slutföras innan den yttre kan varva vidare.

Denna regel är inget principellt nytt utan en direkt konsekvens av **for**-satsens flödesschema när man tillämpar den både på den inre och yttre **for**-satsen (sid 139). Sammanfattningsvis kan vi säga att den yttre loopen enligt denna regel låter den in-

re loopen konkatenera raderna och göra radbyte, medan den inre loopen konkatenerar talen (samt två mellanslag) i varje rad.

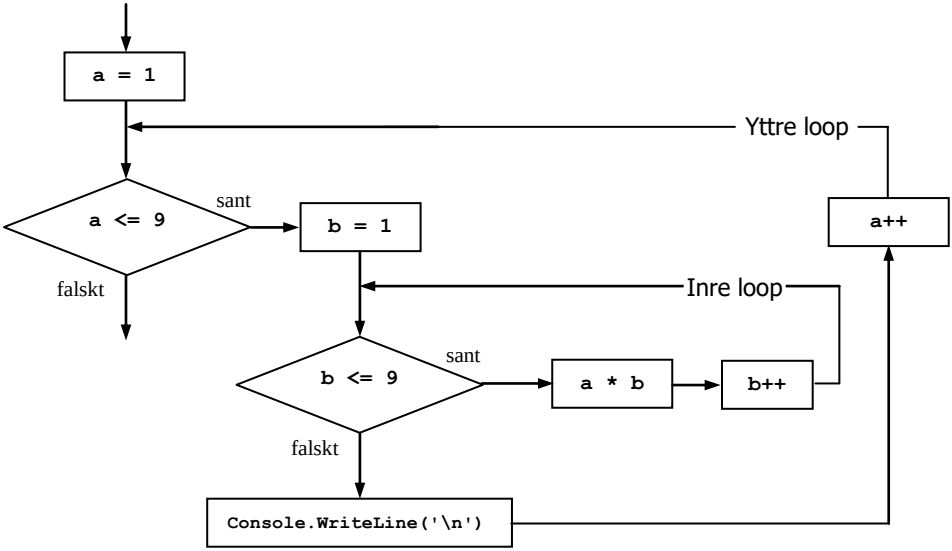
Multiplikationstabellen

Nu när vi lärt känna den nästlade **`for`**-satsen kan vi använda den för en lite nyttig applikation, nämligen att skriva ut multiplikationstabellen. Samtidigt kommer vi genom att rita flödesschemat, att besvara frågan om den nästlade **`for`**-satsen är en ny kontrollstruktur eller en en nästling av den redan kända bestämda repetitionen.

```
// MultipTab.cs
// Skriver ut multiplikationstabellen med nästlad for-sats
using System;

class MultipTab
{
   static void Main()
   {
      Console.WriteLine("\nMultiplikationstabell:\n");
      for (int a = 1; a <= 9; a++)         // Yttre loop skri-
      {                                    // ver ut 9 rader
          for (int b = 1; b <= 9; b++)     // Inre loop
              Console.Write(a*b + "\t");   // skriver ut 9
                                           // tal i en rad
          Console.WriteLine('\n');         // Radbyten mellan
      }                                    // yttre och inre
   }
}
```

Illustrationen nedan visar flödesschemat till den nästlade **`for`**-satsen i **`MultipTab`**:



Pga platsbrist i flödesschemat står i den inre loopens första ruta, **a * b** som en slags symbolisk förkortning för hela den inre loopens kropp i programmet **MultipTab** dvs koden:

```
Console.Write(a*b + "\t");
```

Detta görs i varje varv av den inre **for**-loopen. En jämförelse av den nästlade **for**-satsens flödesschema på förra sidan med den enkla **for**-satsens flödesschema på sid 139 visar att det är en nästling av två enkla **for**-strukturer i varandra: Den yttre **for**-strukturen har efter initiering av sin räknare **a = 1** och efter test av sitt villkor **a <= 9** som ”instruktion(er)” en **for**-struktur till, den inre, följd av en utskriftssats som konkatenerar radbyte. Den inre **for**-strukturen har i sin tur sin egen initiering av räknaren **b = 1** och sitt eget villkor **b <= 9** och som ”instruktion(er)” de två satser ovan som lägger till multiplikationernas resultat på en rad med ett avstånd av 8 mellanslag för alla, dessutom 2 till dvs 10 mellanslag för alla tal mellan 0-9. Vi får följande tabellerad utskrift när vi kör **MultipTab**:

```
Multiplikationstabell:
1       2       3       4       5       6       7       8       9
2       4       6       8       10      12      14      16      18
3       6       9       12      15      18      21      24      27
4       8       12      16      20      24      28      32      36
5       10      15      20      25      30      35      40      45
6       12      18      24      30      36      42      48      54
7       14      21      28      35      42      49      56      63
8       16      24      32      40      48      56      64      72
9       18      27      36      45      54      63      72      81
```

Självklart kan man minska (eller höja) avståndet mellan kolumnerna genom att skicka mindre (eller större) antal mellanslag istället för en tabulator till utskrift. Radavståndet som egentligen motsvarar *två* radbyten pga **Console.WriteLine('\n');** kan minskas om man skriver **Console.WriteLine();** istället.

En kluring:

Byt ut i programmet **MultipTab** raden:

```
                Console.Write(a*b + "\t");      // skriver ut 9
```

Mot följande rad och kör:

```
                Console.Write(a*b + '\t');      // skriver ut 9
```

Utsrkiften blir knasig. Förklara varför?

Övningar till kapitel 6

6.1 Skriv ett program som läser in två *tal* och skriver ut **`OK`** om de matats in i rätt ordning, dvs om det första är mindre än det andra. Vad händer om de är lika stora?

6.2 Modifiera din lösning från övn 6.1 genom att läsa in två *tecken* istället för tal. Skriv ut **`OK`** om de matats in i ordning. Annars ska programmet skriva ut ett *meddelande* om att tecknen matades in i fel ordning.

6.3 Skriv ett program som läser in tre tal, hittar och skriver ut det största av dem. Vilken ändring i koden leder till det minsta talet?

6.4 Skriv ett program som läser in begynnelsebokstaven till en veckodag, med en **`switch`**-sats bestämmer vilken veckodag det är och skriver ut den. Fixa problemet med tisdag/torsdag genom att nästla en **`if-else`**-sats i **`switch`**-satsen för att läsa in och bearbeta den *andra* bokstaven. Ta hand om felaktig inmatning.

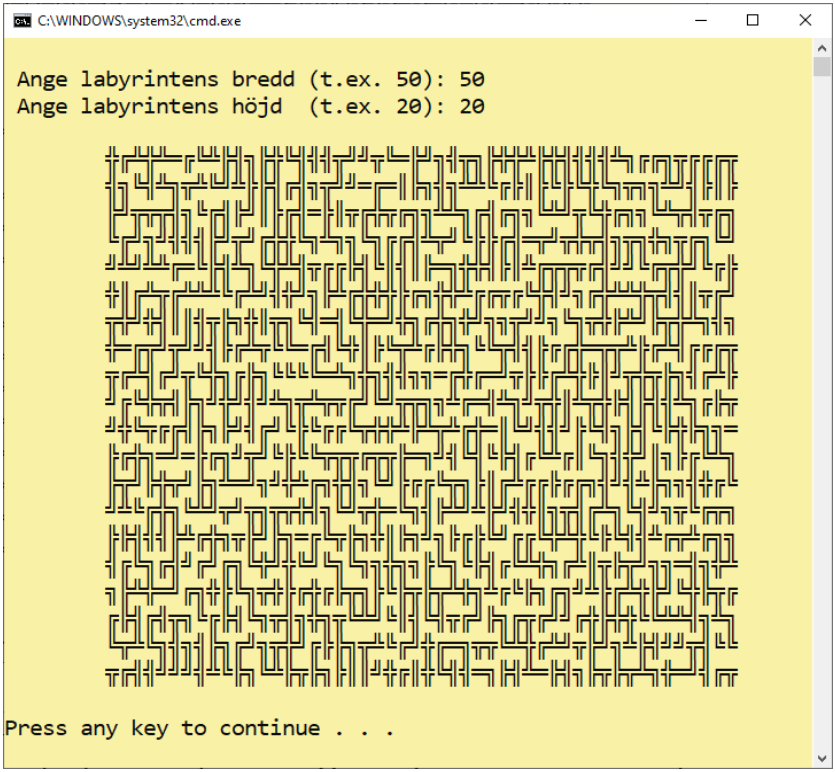
6.5 Vidareutveckla övn 6.2 så att användaren får flera chanser att mata in två tecken i rätt ordning så länge han/hon matar in dem i fel ordning. Du kan göra det genom att bygga in inmatningen, bearbetningen och utmatningen i en **`do`**-loop.

6.6 Skriv ett program som läser in ett heltal och använder det som stegvariabel för att skriva ut talen från 1 till 5000. Om steget är t.ex. 5 skrivs var femte tal ut.

6.7 Skriv ett program som använder **`for`**-satser för att läsa in fem tecken, kryptera dem genom att förskjuta dem med *ett* steg i ASCII-tabellen och skriva ut dem. T.ex. ska inmatningen **`Kalle`** ge utskriften **`Lbmmf`**. Återställ sedan det krypterade ordet: **`Lbmmf`** ska återställas till **`Kalle`**. Vidareutveckla programmet genom att utöka (t.ex. läsa in) antalet steg (krypteringsnyckeln). I övn 5.5 (Kryptering av ord, sid 109) gjordes detta utan **`for`**-satser.

6.8 Skriv ett program som läser in ett stort heltal och utgående från det, skriver ut alla tal baklänges till 1. För att räkna ned i en loop kan du använda minskningsoperatorn **`--`** som fungerar på liknande sätt som ökningsoperatorn: Satsen **`i--;`** gör samma sak som **`i = i - 1;`**

6.9 ***Labyrinten (projekt)*** Visst är det roligt att med ett C# program låta datorn rita en labyrintartad figur på skärmen som kan se ut så här:



Visserligen är detta ingen riktig labyrint. För en sådan skulle det krävas mycket mer. En riktig labyrint skulle kunna vara föremål t.ex. för ett spelprojekt, som underliggande grafik, självklart med lite andra finesser, färg osv. Bilden ovan visar snarare om en labyrintartad *figur* som är slumpmässigt ihopsatt av ett antal tecken som vi kallar för *dubbla linjegrafiktecken* (LGT). De är tagna ur teckentabellen *Unicode* som är den gällande teckenstandarden i hela världen. I figuren ovan är de ordnade som en sorts tabell (50 rader, 20 kolumner). I koden gör man det med en dubbel- eller nästlad **`for`**-loop, som är helt enkelt en (inre) **`for`**-loop i en (yttre) **`for`**-loop. Denna nästlade kontrollstruktur används i alla programmeringsspråk för att åstadkomma en 2D utskrift – typ tabell – där den yttre loopen skriver ut raderna och den inre loopen kolumnerna.

Tecknen i figuren ovan är slumpvis valda. Därför borde varje körning av programmet generera en lite annorlunda labyrintartad figur. Du kan gärna försöka med en egen algoritm att åstadkomma ett program som ritar en laby-

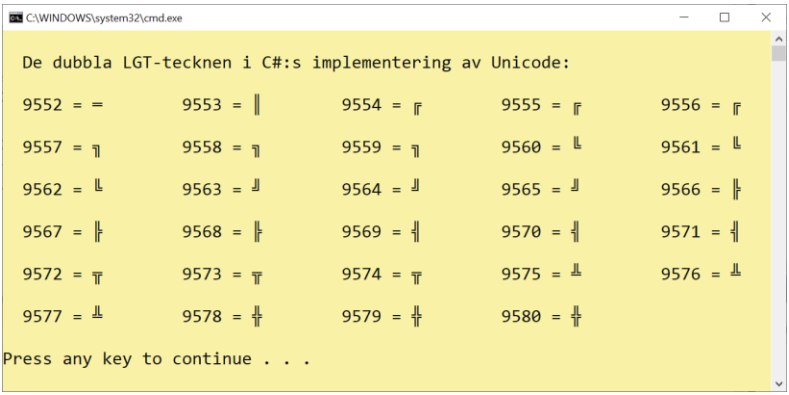
rintartad figur. Men följer du instruktionerna i övningarna har du i alla fall ett förslag till en algoritm som fungerar.

Gör så här för att rita ”labyrinten”:

Steg 1 Repetera hantering av tecken inkl. *explicit typkonvertering* och *Unicode* genom att mata in och köra programmet **`Int2char`** (sid 95) för koderna 9552-9580. För att se alla tecken till dessa koder i en översikt genomför ***Stegen 2-3*** :

Steg 2 Studera programmet **`NestedFor`** (sid 147) som visar hur man nästlar en inre **`for`**-sats i en yttre **`for`**-sats. Jämför den nästlade **`for`**-satsens kod med programmets körexempel på samma sida. Använd idén till nästlade **`for`**-satser för att konstruera en egen sådan, som du kommer att behöva i ***Steg 3*** :

Steg 3 Skriv ett C# program som producerar följande utskrift:



Dessa tecken finns i den standardiserade teckentabellen *Unicode* och används i *text mode* för att rita raka linjer, ramar osv. i konsolen. Vi kallar dem för *linjegrafiktecken* (LGT). Deras koder som är angivna ovan, används i ***Steg 5*** där du ska rita den labyrintliknande figuren på förra sidan med dessa tecken. Den fullständiga Unicode-tabellen som är den gällande teckenstandarden i hela världen, hittar du t.ex. på Internet under adressen: **`unicode.-coeurlumiere.com`**. Jämför gärna koderna ovan med denna tabell som är den gällande teckenstandarden i hela världen, och konstatera de små skillnaderna. C# följer inte exakt Unicode-standarden.

Steg 4 Bekanta dig med hantering av slumptal i bl.a. programmet **`DoRand`** (sid 134),

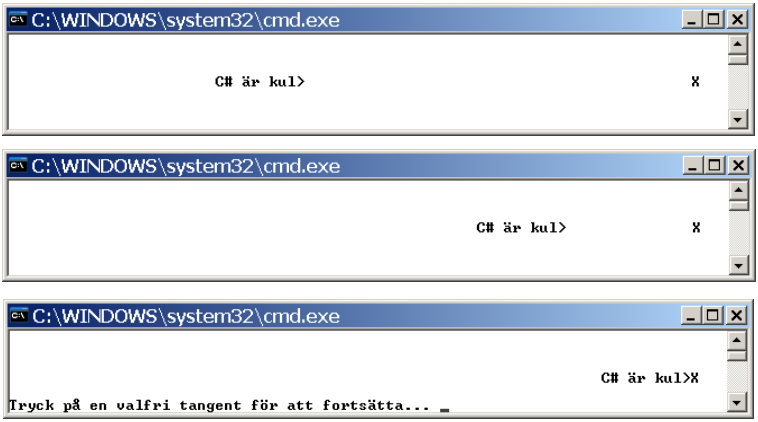
Steg 5 Skriv slutligen det program som med hjälp av de dubbla linjegrafiktecknen från ***Steg 3***, C#:s slumpgenerator och en dubbel- eller nästlad **`for`**-sats ritar en labyrintliknande figur i konsolen som är slumpmässigt ihopsatt av de nämnda LGT-tecknen, se projektets presentation.

6.10 ***Löpande texten (projekt)*** Skriv ett program som visar (simulerar) en löpande text, t.ex.: **`C# är kul>`** som horisontellt rör sig i konsolfönstret tills den ”träffar” på ett hinder, t.ex. ett kryss i form av ett **`X`**. Skriv ut först krysset i slutet av en rad i konsolen, sedan textens initialposition i början av samma rad. Ta exakt reda på hur många mellanslag krysset har avstånd från konsolens vänstra rand. För att gå till början av samma rad (utan radbyte) för att skriva texten initialt, kan escapesekvensen **`\r`** (carriage return) användas. Gör experiment med **`\r`** för att bekanta dig med dess funktion.

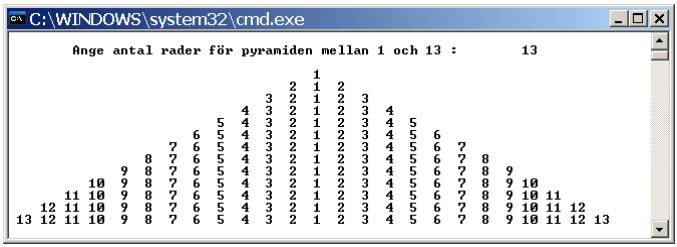
Rörelsen kan sedan simuleras i en loop genom att i varje varv av loopen med 10 styck **`\b`** ta bort den i förra varvet ritade texten (om den var ritad med 10 tecken), stega med (dvs skriva ut) kanske ett (eller flera) mellanslag (rörelsens ”hastighet”) och skriva om texten **`C# är kul>`**. Har loopen lika många varv som krysset **`X`** har avstånd från konsolens vänstra rand minus textens längd – i det föreslagna exemplet 10 – kommer rörelsen att stoppas strax innan texten ”träffar” på **`X`**. Nedan ser du några ögonblicksbilder av den löpande txten.

Även om du gjort allt rätt kommer du inte se att texten rör sig om du inte lägger in en fördröjning i loopen, eftersom allt går så fort och man inte hinner se något förlopp. Det kan du göra genom att i loopen skriva satsen:

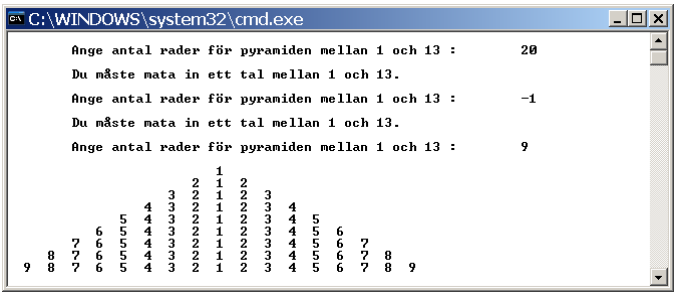
```
System.Threading.Thread.Sleep(100);
```



6.11 ***Pyramiden (projekt)*** Slutmålet med detta projekt är att utveckla ett program som skriver ut en pyramidliknande figur med tal, t.ex. så här:

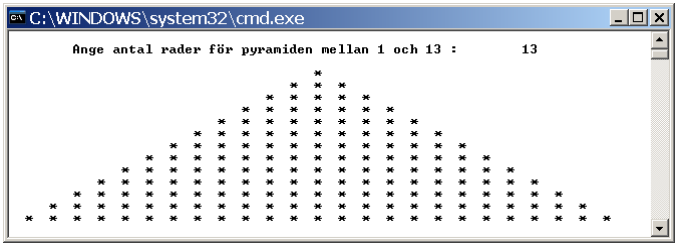


Programmet ska vara så generellt att det skriver ut talpyramider även om man matar in mindre antal rader. Uppmana användaren att hålla sig talintervallet [1, 13]. Anledning till denna restriktion är att talpyramiden inte ryms i konsolen om man överskrider detta intervall . Så här kan det se ut:



Tips till Pyramiden:

För att komma igång börja med ett program som ritar en *stjärn*pyramid:



Strunta till att börja med även på hanteringen av felinmatning av antal rader och jobba med ett fast antal rader. Du kan lägga till det senare.

Använd en nästlad for-sats med en yttre loop och tre inre loopar:

- En för de tomma platserna i pyramiden (mellanslagen)
- En för stjärnorna i pyramidens högra halvan (räknat från den vertikala mittlinjen (symmetriaxeln))
- En för stjärnorna i pyramidens vänstra halvan.

Räkna med att du måste använda i de inre looparna den yttre loopens räknare och slutvärde. T.ex. kan villkoret i den första inre loop som ritar de tomma platserna, se ut så här:

```
column <= numberOfRows - row;
```

Där **`column`** är den inre loopens, **`row`** den yttre loopens räknare och **`numberOfRows`** hela pyramidens antal rader, t.ex. 13 som ovan. Då kan den här första inre loopen skriva ut tre mellanslag i varje varv. I de två andra inre looparna kan två mellanslag och en * skrivas ut.

Observera att alla dessa tips inte ska förhindra att du använder dina egna idéer för att lösa projektuppgiften. Det finns inte endast ett tillvägagångssätt. Uppgiften kan lösas på väldigt många olika sätt.

Kapitel 7

Metoder

7.1 Vad är en metod?

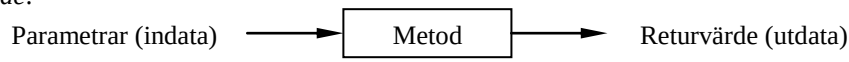
De flesta känner till begreppet *funktion* från matematiken. Man tänker först på en formel som beräknar ett värde utgående från ett annat värde. Även i programmering finns den matematiska synen på funktion som underliggande koncept och historisk utgångspunkt. Men under tiden har den fått en bredare tolkning då den tillämpats på all datoriserad problemlösning.

En *metod* är en funktion som definieras i en klass. I objektorienterade programmeringsspråk är metoder inkapslade i klasser. I C# är det obligatoriskt. Därför finns det i C# inga fristående funktioner utan endast metoder. Bortser man från denna överordnade struktur och ser på det ”inifrån”, är funktioner och metoder identiska.

En metod i C# är en namngiven kodmodul (ett antal satser) i en klass som utförs när metoden anropas. Vid anropet kan den ta emot indata, s.k. parametrar, bearbeta dem och returnera utdata, s.k. returvärde.

Som ”ett antal satser” är en metod en del av en klass som isoleras och skrivs separat som en anropbar modul för att kunna användas även i andra klasser.

Ur praktisk synpunkt kan en metod jämföras med en ”svart låda” i vilken man stoppar in indata och får ut utdata: Indata kallas även *parametrar* och utdata *returvärde*:



En metod kan ha 0, 1 eller flera parametrar. Den kan ha 0 eller 1 returvärde. En metod kan alltså inte ha flera returvärden. Både parametrarna och returvärdet kan vara tal, tecken, strängar, sanningsvärden eller referenser till objekt. Metoden bearbetar de ev. inkommande parametrarna på ett visst sätt och returnerar ev. ett värde. Metoder *med* returvärde behandlas först. Metoder *utan* returvärde tas upp senare.

Vi har hittills använt några av C#-bibliotekets metoder, t.ex. **`Console.Write()`**, **`Console.WriteLine()`**, **`Console.Read()`**, **`Console.ReadLine()`**, **`int.Parse()`**, … utan att behöva veta hur de var kodade, därför: ”svarta lådor”. De var förprogrammerade åt oss och vi använde dem bara för att åstadkomma vissa funktionaliteter. I detta kapitel ska vi nu lära oss att själva skriva metoder. Men en metod som vi redan har skrivit själva – och det har vi gjort i alla våra programexempel – är metoden **`Main()`**, för den är obligatorisk. Så här definierade vi tidigare ett *C# program* (sid 43):

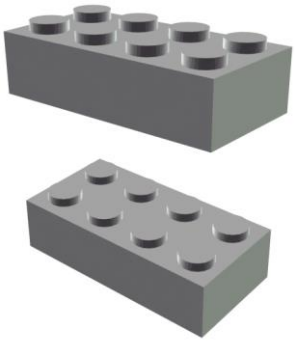
Ett C# program är en samling av klasser, av vilka en och endast en måste innehålla metoden **`Main()`**.

När programmet körs startar exekveringen i **`Main()`**.

Varför metoder?

Frågan är berättigad för nybörjare: Varför ska man krångla till det hela? Kan man inte helt enkelt skriva kod rakt ned i `Main()`? Föreställ dig en verksamhet som dynamiskt växer med tiden, ett expanderande företag eller en organisation med stigande antal medlemmar. Hur organiserar man jobbet? Man genomför arbetsdelning och delegerar uppgifterna. Var och en får en väl definierad specifik arbetsuppgift. Annars skulle man inte kunna klara av jobbets komplexitet. Samma sak gör man med program vars kod växer, vilket händer när man utvecklar program efter behov och behoven bara blir större och större. Man delar upp det stora programmet i mindre moduler för att kunna klara av komplexiteten. Hur det görs ska vi nu diskutera under rubrikerna: *Modularisering*, *återanvändning av kod* och *strukturering av program*.

Modularisering eller Lego-principen

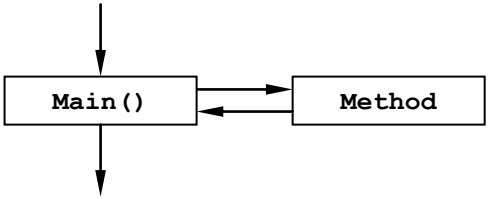


De flesta har väl någon gång som barn, eller tillsammans med sina barn, byggt ett hus, en bil eller liknande med Lego-bitar. Efter ett tag har huset kanske rasat och nya tekniska underverk har konstruerats. Men även de har någon gång plockats isär. Det enda som blivit kvar är själva Lego-bitarna som 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 både modellering och problemlösning. Ett stort komplext problem bryts ned i mindre *moduler* – motsvarande Lego-bitarna – och bearbetas en i taget. I traditionell (procedural) programmering är dessa moduler *funktioner* som blir *metoder* när de bindas in i klasser.

För att kunna göra det 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. De delar av Lego-biten som tillåter denna passning, kan anses som Lego-bitens *gränssnitt* mot andra Lego-bitar. På samma sätt har en metod ett gränssnitt mot andra metoder för att kunna kommunicera med dem. Även detta gränssnitt har två delar: För det första metodens *parametrar* som importerar värden från omgivningen och för det andra metodens *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 metod innebär att *aktivera* metoden. Detta sker genom att ev. skicka till den parametrar, utföra koden som står i metoden och ev. få tillbaka returvärdet. Generellt finns det i ett program flera metoder som anropar varandra. Det enklast tänkbara

exemplet är att **`Main()`** anropar en **`Method`** dvs **`Main()`** är den *anropande* och **`Method`** den *anropade* metoden. Då kan programflödet mellan dem se ut så här:



Återanvändning av kod

är det andra svaret på frågan varför man i programmering sysslar med metoder. Samma idé finns bakom Lego-biten som minsta återanvändbara modul för att bygga i princip vad som helst. Har man i ett program redan beskrivit en funktionalitet 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 den 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 separera koden för denna kategori från det aktuella programmet och skriva den som en klass för att kunna återanvända koden i vilket annat program som helst. Det kräver att den ursprungliga koden som kanske var skräddarsydd för just det speciella programmet, nu som klass måste formuleras på ett mer generellt sätt. Hela C#:s klassbibliotek bygger på idén om återanvändning av kod.

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). Beroende på applikationen kan det finnas många olika möjligheter till modularisering.

Det enklast tänkbara sättet att strukturera vilket program som helst är t.ex. att dela in det i *inmatning – bearbetning – utmatning* vilket diskuterats tidigare (sid 72). Dessa tre delar kan skrivas i var sin metod som sedan kan anropas av **`Main()`**. Denna huvudmetod kan då bestå av ett få antal satser som endast skapar objekt och anropar objektens metoder. På så sätt har man från **`Main()`** en övergripande kontroll över hela programflödet. Dessutom kan metoderna placeras i klasser som lagras i separata filer och importeras som innehåller **`Main()`**. Så kan man så småningom bygga upp sitt eget bibliotek av egendefinierade klasser och metoder – skräddarsydda för det egna behovet.

7.2 Metoder med returvärde

Modularisering och strukturering är inga självändamål. Man genomför dem för att underlätta och effektivisera programutveckling. Endast logiskt sammanhängande uppgifter som på ett naturligt och meningsfullt sätt kan avgränsas från andra, ska modulariseras. Som exempel tar vi upp programexemplet **`Operator`** (sid 71) där vi redan hade strukturerat koden enligt mönstret *inmatning – bearbetning – utmatning*. Nu modulariserar vi programmet genom att skriva bearbetningen i en metod som vi döper till **`TotalDays()`**, placerar den utanför och anropar den i **`Main()`**. Fortfarande finns all kod i *en* fil kallad **`ReturnMethod.cs`**.

Den bit av kod som formulerar metoden – längst ned vitmarkerad utanför **`Main()`** – kallas för metodens *definition*. Metoden **`TotalDays()`** har tre parametrar av typ **`int`** och ett returvärde av typ **`int`**. Denna bit av kod har hamnat på nästa sida. Kvar blir i **`Main()`** in- och utmatningen samt metodens *anrop*. Anropet är den bit av kod som kallar funktionens namn med parametrarna i parentesen – även den vitmarkerad, men i **`Main()`**. Den här biten av kod finns på denna sida.

```
// ReturnMethod.cs
// Gör samma sak som progr. Operator (sid 71). In- och utmat-
// ning görs i Main(), bearbetningen utanför Main()
using System;

class ReturnMethod
{
   static void Main()
   {
      int  year, months, weeks, days;

      /* I n m a t n i n g */
      Console.Write("\n\tAnge antal år:\t\t");   // Ledtext
      year = int.Parse(Console.ReadLine());      // Inläsning

      Console.Write("\n\tAnge antal månader:\t");
      months = int.Parse(Console.ReadLine());

      Console.Write("\n\tAnge antal veckor:\t");
      weeks = int.Parse(Console.ReadLine());

      Console.Write("\n\tAnge antal dagar:\t");
      days = int.Parse(Console.ReadLine());

      /* U t m a t n i n g */
      Console.WriteLine("\n     " + year + " år, " +
         " månader, " + weeks + " veckor och " + months +
         days + " dagar är " +
         TotalDays(year, months, weeks, days) +
                                    " dagar totalt.\n");
   }                  // OBS! Här slutar Main()
```

```
    static int TotalDays(int y, int m, int w, int d)// Metoden
    {
        /* B e a r b e t n i n g */
        return 365*y + 30*m + 7*w + d;
    }
}                       // OBS! Här slutar programmet (klassen)
```

En körning av programmet **ReturnMethod** ger följande utskrift:

```
        Ange antal år:          2
        Ange antal månader:     11
        Ange antal veckor:      3
        Ange antal dagar:       6
 2 år, 11 månader, 3 veckor och 6 dagar är 1087 dagar totalt.
```

Självfallet blir det samma utskrift som för programmet **Operator** (sid 72).

Programmet **ReturnMethod** består av en klass innehållande *två* metoder: Först skrivs **Main()**, sedan metoden **TotalDays()**, men det går lika bra att placera dem i omvänd ordning. Det enda som inte får göras är att placera dem *i* varandra. Båda skrivs (just nu) i samma klass och även i samma fil. Metoden **TotalDays()** har fyra parametrar **y**, **m**, **w** och **d**, alla av typ **int** samt ett returvärde av typ **int**. Exekveringen startar i **Main()** som i sin tur anropar **TotalDays()**. För att *båda* metoderna ska kunna utföras när programmet körs, har anropet av **TotalDays()** skrivits i **Main()**. När **Main()** anropar **TotalDays()** överförs de inlästa värdena till variablerna **year**, **months**, **weeks**, **days** från **Main()** till variablerna **y**, **m**, **w**, **d** i den ordning som de förekommer i parentesen av metodnamnet. Metoden beräknar antalet totaldagar och returnerar resultatet till **Main()** där det skrivs ut tillsammans med lite användarvänlig text.

Definition av metoder med returvärde

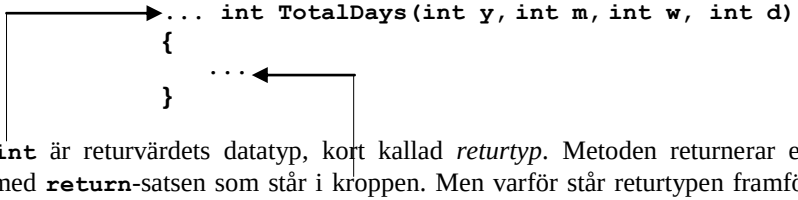
Generellt ser definitionen av metoder med returvärde ut så här:

```
modifierare returtyp metodnamn (datatyp fpar1, datatyp fpar2, ...)
{
    statement(s);
    return uttryck;
}
```

Första raden kallas metodens *huvud* inklusive parentesen **(** ... **)** som innehåller listan över alla parametrar, därför kallad *parameterlistan*. Resten, det som står inom klamrarna **{ ... }**, är metodens *kropp*. Huvudet kan inledas med en eller flera modifierare vilket vi kommer att återkomma till. Med returtyp menas datatypen till *returvärdet*. Ordningen är av betydelse: Modifierarna kan utelämnas, men får inte

skrivas efter returtypen. fpar står för *formell parameter*. Så kallas parametrar som förekommer i metodens definition inom parameterlistan. Den kan innehålla en eller flera parametrar, men kan även vara tom. Oavsett antalet parametrar inkluderar man alltid, när man i beskrivande text nämner en metod, parentesen **()** och lägger den till metodnamnet vilket man gör för att skilja mellan metoder och variabler. Parentesen är alltså kännetecknet för en metod.

Observera att metodhuvudet inte avslutas med semikolon. Det är ju inte en sats utan bara huvudet till en metod vars kropp följer. I programexemplet **ReturnMethod** är metoden **TotalDays()** definierad på följande sätt:



Detta **int** är returvärdets datatyp, kort kallad *returtyp*. Metoden returnerar ett heltal med **return**-satsen som står i kroppen. Men varför står returtypen framför metodens namn? Det verkar – om man för ett ögonblick bortser från parameterlistan – som om **int TotalDays** vore en deklaration för ”variabeln” **TotalDays** till datatypen **int**. Denna tolkning är korrekt om man vet att **TotalDays** är både metodens namn *och* returvärdets ”variabel”. Denna variabel kan få sitt värde endast från **return**-satsen när metoden anropas. Medan returvärdet är metodens output (utdata) är parametrarna metodens input (indata). **TotalDays()** har fyra parametrar av typ **int**. De är definierade i parameterlistan: **(int y, int m, int w, int d)**. Observera att det inte går att skriva **(int y, m, w, d)** vilket man kan göra vid deklaration av vanliga variabler. Visserligen är parametrar också variabler, men när de definieraras i parameterlistan, måste man upprepa deras datatyper även om de är av samma typ. Det är den enda syntaktiska skillnaden mellan parametrar och vanliga variabler. I definitionen heter de *formella* parametrar därför att de definieras i parameterlistan som ”tomma” minnesceller i väntan på att bli initierade när metoden anropas. Deras namn saknar betydelse – bara man använder konsekvent samma namn i metodens kropp. De initieras när metoden anropas med **TotalDays(year, months, weeks, days)**. Då *importeras* de s.k *aktuella* parametrarna **year, months, weeks, days** från **Main()** via de formella parametrarna **y, m, w, d** in i metoden **TotalDays()** där de bearbetas. Sedan *exporteras* returvärdet från **TotalDays()** via metodnamnet till **Main()**.

I *kroppen* till en metod kan ett antal satser stå avgränsade med klammerparet **{ ... }**. Klamrarnas uppgift är att gruppera satserna under metodhuvudet till ett s.k. *block* (sid 117). I metoden **TotalDays()** består detta block av **return**-*satsen* som returnerar ett heltalsvärde till metodnamnet. Men **return**-satsen gör en sak till: Den avslutar även metoden. Eventuell kod efter den kommer att inte utföras. Därför ska **return**-satsen alltid vara metodens sista sats. Den logiska slutsatsen är att det får exekveras endast en **return**-sats i en metod. Då den returnerar ett värde till metodnamnet, måste namnet ha beredskapen at ta emot det. Detta innebär att **Total-**

Days samtidigt som det är metodnamnet, också är en variabel av typ **int** – med den begränsningen att den endast kan få sitt värde från **return**-satsen. **TotalDays** kan inte lagra returvärdet som är ett haltal, om det inte är via returtypen deklarerat till datatypen **int**.

Placering av metoder

När vi pratar om metoders placering menar vi placeringen av deras *definition*. Kan man definiera metoder var som helst i ett C# program? För det första måste metodens definition alltid placeras i en klass. I programexemplet **ReturnMethod** placerade vi metoden **TotalDays()** i en och samma klass som **Main()**. Har man dem i samma klass som **Main()** spelar inbördesordningen ingen roll. Detta är o.k. så länge man håller på med att utveckla och testa metoden. Men när man är klar med utvecklingsstadiet, ska man helst ha den i en separat klass som då kommer även att placeras i en separat fil vilket vi kommer att ta upp i nästa avsnitt. Men innan vi går vidare ska vi konstatera följande absolut förbud när det gäller placering av metoder:

> Metoder får inte definieras inuti en annan metod.

Observera att detta förbud endast gäller för definition av metoder, inte för anropet. Anropen kan vara nästlade i varandra, men inte definitionen.

Anrop av metoder med returvärde

Metodens definition är endast är en *mall*, en *föreskrift* om vad som *skulle* hända om metoden anropades, jämförbar med ett matrecept som man skriver ned och stoppar i kökskåpets låda i väntan på att någon gång ta fram det och laga mat. Först när beredskapen till matlagning finns – alla ingredienser är handlade och finns på plats – kan matreceptet komma till användning som en algoritm. Samma sak är det med metodens definition: den är endast en potentiell eller formell kod. Aktuell blir den först när vi anropar metoden. Då börjar saker och ting att hända. I programmet **ReturnMethod** anropas metoden **TotalDays()** från metoden **Main()** med:

```
TotalDays(year, months, weeks, days)
```

Anropet består av att kalla metoden vid namn och i parameterlistan skriva *lika många* parametrar som definitionen föreskriver – i det här fallet fyra. Parametrar som förekommer i metodens anrop – här **year, months, weeks, days** – kallas *aktuella*. Antalet aktuella parametrar *måste* vara lika med antalet formella parametrar – de som förekommer i metodens definition. Annars blir det kompileringsfel. Samtidigt *borde* de aktuella parametrarna ha samma datatyp som de formella. Annars försöker kompilatorn att göra automatisk typkonvertering till måldatatypen, vilket kan vara problematiskt. Dessutom måste vi se till att parametrarnas *ordning* stämmer dvs vi måste kontrollera, för exakt i den ordning vi skriver dem i anropet, kommer deras värden att överföras till de formella parametrarna i metodens definition.

Att anropet kan läggas i utskriftssatsen beror på att **`TotalDays()`** är en metod *med returvärde* – dvs har en **`return`**-sats. Därmed bär metodnamnet samtidigt returvärdet i sig. I exemplet bakar vi in anropet i utskriftssatsen för att konkatenera det med förklarande, användarvänlig text:

```
Console.WriteLine(metodnamn(apar1, apar2, ...));
```

Ett alternativ är att lägga anropet i en tilldelningssats:

```
variabel = metodnamn(apar1, apar2, ...);
```

där **`apar`** står för *aktuell parameter* och **`metodnamn`** är den anropade metoden. Självklart måste **`variabel`** och även alla aktuella parametrar vara definierade *före* anropet på vanligt sätt i den anropande metoden – i vårt exempel i **`Main()`**. Parametrar som skrivs i en metods *anrop* kallas *aktuella parametrar* [*], en beteckning som ska framhäva deras skillnad till de *formella parametrar* som skrivs i metodens *definition*. Med *aktuell* menas att de har aktuella värden som gäller vid anropet för att skickas till metodens formella parametrar. Därför måste de vara konstanter eller definierade och initierade variabler. I programmet **`ReturnMethod`** är de variabler: **`year`**, **`months`**, **`weeks`**, **`days`**. De formella parametrarna däremot – i vårt exempel **`y`**, **`m`**, **`w`**, **`d`** – *måste* alltid vara variabler som definieras i metodens parameterlista när denna definieras. Sina värden får de *första gången* inte tilldelade i metodens kropp utan från de aktuella parametrarna vid metodens anrop.

Vad händer vid anropet av en metod?

Tre saker händer när en metod anropas från en annan metod. Låt oss som exempel ta programmet **`ReturnMethod`** där metoden **`TotalDays()`** anropas från **`Main()`**:

1. **Parameteröverföring** Då överförs de aktuella parametrarna **`year`**, **`months`**, **`weeks`**, **`days`** till de formella parametrarna **`y`**, **`m`**, **`w`**, **`d`**. Observera att korresponderande parametrar bestäms av ordningen i parameterlistan och borde vara av samma datatyp. Anropet av metoden vidarebefordrar värdena till metodens formella parametrar. Så hamnar de i kroppen till metoden **`TotalDays()`**.

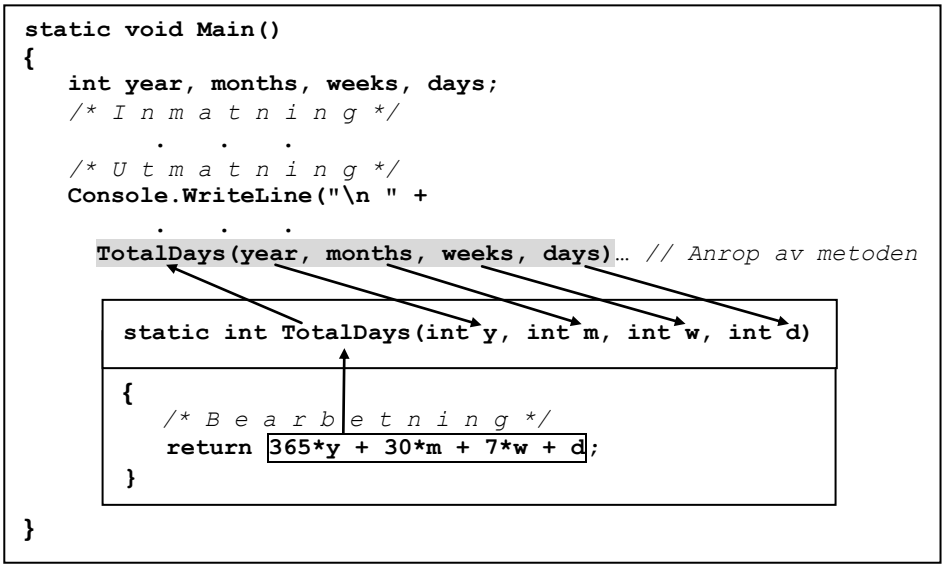
2. **Exekvering av koden i metodens kropp** vilket i vårt fall innebär att utföra den **`return`**-sats som står där dvs beräkna uttrycket **`365*y + 30*m + 7*w + d`** och returnera resultatet till metodnamnet **`TotalDays`**. Med värdena från parameteröverföringen (punkt **1** ovan) blir det **`365*years + 30*months + 7*weeks + days`**.

[*] Andra beteckningar som förekommer i litteraturen är *anropsparametrar* eller *argument*. Speciellt *argument* används ofta då det är en inkörd matematisk term: T.ex. är $\sqrt{3}$ ett anrop av metoden $\sqrt{x}$ där x – i matematiska termer – är ”variabeln” och 3 ”argumentet”. I programmeringstermer skulle x kallas för den *formella* och 3 den *aktuella* parametern.

3. **Överföring av returvärdet** sker i omvänd riktning jämfört med parameteröverföringen, nämligen från den anropade metoden **`TotalDays()`** till den anropande metoden **`Main()`**. Vi får returvärdet från metoden, i exemplet är det heltalsvärdet till det uttryck som står efter **`return`**. Att returvärdet hamnar i **`Main()`** beror på att anropet görs med metodnamnet **`TotalDays`** som fått returvärdet.

Det som händer vid anrop av en metod är alltså att data byts ut mellan den anropande och den anropade metoden, i vårt exempel mellan **`Main()`** och **`TotalDays()`**, att koden i den anropade metoden utförs när anropet inträffar, samt att returvärdet till sist hamnar i den anropande metoden. En översikt över dataflödet mellan dessa två metoder (se pilarna nedan) och framför allt i *vilken ordning* deras koder utförs ges i följande bild som illustrerar punkterna **1-3** ovan.

Dataflödet mellan Main() och TotalDays()



Observera att bilden visar det som *händer* när programmet **`ReturnMethod`** exekveras, inte hur det skrivs i koden. Det är nämligen en skillnad mellan den ordning i vilken koden skrivs och den när koden körs. Bilden visar själva *aktionen* som startar vid exekveringen i **`Main()`**. När den den kommit till anropet av metoden **`TotalDays()`** klistras metodens kod in just på detta ställe och utförs enligt punkterna **1-3** på förra sidan. De båda metodernas *aktioner* är nästlade i varandra, men det är absolut inte koden som är skriven i programmet **`ReturnMethod`**. Den har en helt annan struktur: De båda metodernas koder är inte alls nästlade i varandra, snarare tvärtom, de är isolerade från varandra. Och så måste det vara: Koden till metoden **`TotalDays()`** får inte under några omständligheter skrivas *i* **`Main()`**, den måste stå *utanför* **`Main()`**, antingen *före* eller *efter* **`Main()`**, ja den

kan t.o.m. ligga i en *separat* fil. Men när vi kör programmet i sin helhelt händer saker och ting i den ordning som visas på bilden på förra sidan.

Bilden ovan visar dataflödet som går från de aktuella parametrarna **`1, 6`** till de formella parametrarna **`a, b`**. Dessa bearbetas i metoden **`TotalDays()`** dvs formeln efter **`return`** beräknas. Sedan skickas resultatet som returvärde via metodnamnet till **`Main()`**. Metodens **`TotalDays()`**:s kropp utförs precis på det stället där anropet i **`Main()`** står. I den här processen spelar metodhuvudet **`static int TotalDays(int y, int m, int w, int d)`** rollen av ett *gränssnitt* mellan **`Main()`** och **`TotalDays()`**. Det är där kommunikationen mellan dessa separat skrivna moduler äger rum. Därför har vi satt metodhuvudet i en extra ruta i bilden för att understryka rollen som gränssnitt. Huvudet är nämligen åtkomligt både från **`Main()`** och **`TotalDays()`**. De skulle annars inte kunna kommunicera med varandra eftersom de ligger i olika block. Bilden ska nämligen även visa att metodanropet resulterar i en *blockstruktur* som återges av ramarna i bilden. Motsvarande kod till denna blockstruktur utgörs av klamrarna **`{ }`** till både **`Main()`** och **`TotalDays()`**. Klamrarna bildar dessa modulers fasta gränser för kodens giltighet eller räckvidd. För att överskrida dem måste vissa regler beaktas vilket vi kommer att precisera senare. Blockstrukturen är den egentliga orsaken till att metoden **`TotalDays()`** inte kan kommunicera med **`Main()`** annat än via gränssnittet **`static int TotalDays(int y, int m, int w, int d)`**. Det motiverar dessutom varför metodernas koder inte får nästlas i varandra när de skrivs. Om block läs på sid 117.

En viktig ingrediens av metoden **`TotalDays()`**:s huvud har vi inte gått in på hittills, nämligen att det står **`static`** framför namnet **`TotalDays()`**. Vi nöjer oss med att konstatera att **`TotalDays()`** inte kan anropas i **`Main()`** utan att definiera den med **`static`**. Vi ska avsluta behandlingen av *metoder med returvärde* med en företeelse som liknar villkorlig initiering av variabler (sid 118):

Villkorlig **`return`***-sats*

För det första måste man konstatera att **`return`**-satsen är en obligatorisk del av en *metod med returvärde* dvs en metod vars huvud innehåller en returtyp. T.ex. har metodhuvudet **`static int TotalDays(int y, int m, int w, int d)`** returtypen **`int`**. Därför *måste* metodens kropp ha en **`return`**-sats, annars blir det kompileringsfel. För det andra får **`return`**-satsen inte stå i en **`if`**-sats (utan **`else`**) eller i andra kontrollstruktur där villkor är inblandade. Precis som variablers initiering får inte heller **`return`**-satsen vara beroende av villkor utan alternativ.

Följande regel gäller för villkorliga **`return`**-satser:

> Metoder vars **`return`**-sats är beroende av villkor utan alternativ leder i C# till kompileringsfel.

En villkorlig **`return`**-sats ger kompileringsfel oavsett villkorets sanningsvärde. Förbudet gäller alltså även för sanna villkor, ja t.o.m. för sådana som i **`if(1 == 1)`** eller **`if(true)`**. Dvs till skillnad från villkorligt initierade variabler gäller förbudet även för konstanta villkor, dvs sådana som inte involverar variabler.

Avslutningsvis ska vi nämna att **`return`**-satsen samtidigt *avslutar* en metod med returvärde. Den är alltså en slags terminator efter vilken ingen kod mer utförs. All kod efter den kommer att ignoreras. Därför borde den stå sist i en metods kropp.

7.3 Externlagrade metoder

Programmet **ReturnMethod** som behandlades i förra avsnitt (sid 161) hade två moduler: Metoden **Main()** som organiserar inläsningen samt utskriften och metoden **TotalDays()** som sköter beräkningen. Men nackdelen var att dessa moduler inte var oberoende av varandra: Båda fanns i en och samma klass. Tanken med modularisering är ju att kunna t.ex. använda metoden **TotalDays()** i helt andra sammanhang än att bara skriva ut slumptal i tabellform, t.ex. som rådata till sökning och sortering eller som statistiskt material eller för att överföra dem till en databas, … . Då är det nödvändigt att renodla modulariseringstekniken och inte bara separera modulerna på metodnivå, utan även på klassnivå. Detta åstadkommer man genom att separera metoden **TotalDays()** från allt annat och skriva den i en klass för sig och lagra klassen i en fil för sig. Det menar vi med *externlagrade metoder* som garanterar att man kan anropa dem från vilket program som helst utan att i det nya programmet ha med sig kod som inte har att göra med den nya applikationen. Följande klass ger nu vår gamla metod **TotalDays()** en ny ram och gör den till en renodlad klassmodul:

```
// Total.cs
// Egen modul som kan användas i vilket program som helst
// Metoden TotalDays() är inkapslad i klassen Total
// public för att kunna anropas utifrån klassen Total

class Total
{
   public static int TotalDays(int y, int m, int w, int d)
   {
      return 365*y + 30*m + 7*w + d;
   }
}
```

I programmet nedan sker anrop av metoden **TotalDays()** från **Main()** som nu finns i en annan klass, klassen **TotalTest**. Förutom den annorlunda syntaxen till anropet av **TotalDays()**-metoden skiljer sig koden inte från programmet **ReturnMethod**. Men nu tillhör metoderna **Main()** och **TotalDays()** två *olika* klasser, lagrade i två olika filer, varför **Main()** måste anropa den externlagrade metoden **TotalDays()** med koden **Total.TotalDays()**. De två klasserna kan hitta varandra om de lagras i samma mapp och i Visual Studio i samma projekt.

```
// TotalTest.cs

using System;

class TotalTest
{
```

```
   static void Main()
   {
      int  year, months, weeks, days;



      /* I n m a t n i n g */
      Console.Write("\n\tAnge antal år:\t\t");   // Ledtext
      year = int.Parse(Console.ReadLine());      // Inläsning

      Console.Write("\n\tAnge antal månader:\t");
      months = int.Parse(Console.ReadLine());

      Console.Write("\n\tAnge antal veckor:\t");
      weeks = int.Parse(Console.ReadLine());

      Console.Write("\n\tAnge antal dagar:\t");
      days = int.Parse(Console.ReadLine());

      /* U t m a t n i n g */
      Console.WriteLine("\n     "     +
            year + " år, "         + months + " månader, " +
            weeks + " veckor och " + days   + " dagar är " +
            Total.TotalDays(year, months, weeks, days)     +
                                       " dagar totalt.\n");
   }
}
```

Programmet ger samma utskrift som programmet **`ReturnMethod`** (sid 162).

Pga den externa placeringen av metoden **`TotalDays()`** i klassen **`Total`** är syntaxen till metodanropet annorlunda än i förra avsnittet där både den anropande metoden **`Main()`** och den anropade metoden **`TotalDays()`** fanns i samma klass. Klassen **`TotalTest`** har ingen direkt tillgång till metoden **`TotalDays()`** och kan därför inte anropa den endast med metodnamnet utan måste gå via klassen **`Total`** för att göra det med punktnotation: **`Total.TotalDays()`**.

Det är anmärkningsvärt att vi för detta anrop varken behöver inkludera klassen **`Total`** till det nya programmet eller någon annan speciell åtgärd i koden. Det räcker med att placera filerna **`TotalTest.cs`** och **`Total.cs`** som lagrar de resp. klasserna i en och samma mapp på datorn. När C#-kompilatorn stöter på anropet med det okända klassnamnet **`Total`** söker den automatiskt efter filen **`Total.cs`** i den aktuella C#-filens mapp och inkluderar den i kompileringen. I Visual Studio måste båda filerna ingå i det aktuella projektet.

7.4 Metoder utan returvärde

De flesta av de metoder vi hittills använt i våra programexempel hade returvärden. Det var *en* typ av metod, en ganska viktig sådan. Metoder med returvärde kan returnera endast ett värde, ett tal, ett tecken, ett sanningsvärde, en sträng eller en referens till ett objekt. Med *returvärde* menar vi alltid det som returneras av **return**-satsen via metodnamnet. En annan kategori av metoder är sådana som inte har något returvärde alls. *Metoder utan returvärde* kallas även för **void**-*metoder*.

> En **void**-metod är en metod som inte returnerar något värde. I metodhuvudet ersätter **void** returtypen.
> En **void**-metod har antingen ingen **return**-sats alls eller en *tom* **return**-sats: **return;**

void är i C# ett reserverat ord som kan tolkas som ”ingenting”. När **void** i metodhuvudet ersätter datatypen till returvärdet (returtypen) och skrivs framför metodnamnet, eliminerar det returvärdet och definierar en **void**-metod. Detta medför att **return**-satsen i kroppen antingen måste strykas eller ersättas av en tom **return**-sats dvs utan värde: **return;** vilket verkar vara onödigt: I våra exempel föredrar vi att stryka den helt.

Metoden **Test()** i klassen **Password** var ett exempel på en **void**-metod utan parametrar (sid 181). Här följer ett exempel med parametrar. Båda utelämnar **return**-satsen och lagras externt i en klass (i en separat fil) samt anrpas av **Main()** i en testklass:

```
// VoidMethod.cs
// Klass med en void-metod: två parametrar utan returvärde
using System;

class VoidMethod
{
  public static void divSafe(int numerator, int denominator)
  {
    if (denominator != 0)      // Förhindrar division med 0
       Console.WriteLine("\n\tSäker heltalsdivision:\n\n" +
            "\t\t" + numerator + " heltalsdividerad med " +
       denominator + " är " + numerator/denominator + '\n');
    else
       Console.WriteLine("\n\tOBS! Division med 0:\n\n\t\t" +
           "Du har matat in 0 för det tal som ska \n\t\t"   +
           "delas med. Det går inte att dela med 0.\n\t\t"  +
                        "Division med 0 är odefinierad.\n");
  }
}
```

I exemplet ovan står **void** exakt på samma plats som returtypen eller istället för den för att indikera att metoden **divSafe()** inte ger något värde när den anropas. Speciellt returnerar den *inte* divisionens värde. Allt den gör är att exekvera kroppens kod när den anropas. Då skrivs ut all relevant information från metodens kropp: Endast om den andra parametern **denominator** är skilt ifrån **0**, utförs divisionen **numerator/denominator** och divisionens värde skrivs ut. På så sätt undviks division med **0**. Pga datatypen **int** som både **numerator** och **denominator** är deklarerade till (i parameterlistan), blir det inte vanlig utan heltalsdivision (sid 75). Att vi valt här **int** och inte **float** eller **double** beror på att C# beter sig olika vid olika datatyper när det gäller division med **0**. Medan vanlig decimaltalsdivision med **0** inte leder till programavbrott utan producerar symbolen INF som står för infinity (oändligheten) och körningen avslutas regulärt, leder heltalsdivision med **0** till abrupt programavbrott (DivideByZeroException) som uppfattas som krasch. All kod som följer exekveras inte längre. Det är den situation som klassen **VoidMethod**, närmare bestämt metoden **divSafe()** ska förebygga. Pga det modulariserade upplägget kan den användas i vilket program som helst. I längre program kommer även all kod som följer efter en felaktig division med **0** att utföras som vanligt och körningen avslutas regulärt. Däremot kommer ett egenkonstruerat felmeddelande att hänvisa till felet utan att avbryta exekveringen.

Fast **void** kan tolkas som ”ingenting” får det inte stå överallt i koden där man misstänker ”ingenting”. T.ex. får **void** inte stå i parameterlistan till en metod för att antyda att den inte har några parametrar: **void Test(void)** som metodhuvud ger kompileringsfel. **void** får inte heller ersätta datatypen i definitionssatser till vanliga variabler. C# som är strikt typbestämt tillåter inte att en variabel definieras till ”ingen datatyp”. I följande program anropas nu **void**-metoden **divSafe()**:

```
// VoidMethodTest.cs
// Testar klassen VoidMethod genom att från Main() anropa
// dess void-metod divSafe()
// Anropet står som en fristående sats eftersom divSafe()
// inte returnerar något värde
using System;

class VoidMethodTest
{
 static void Main()
 {
   int t, n;
   Console.Write("\n\tAnge ett heltal som ska delas:\t\t");
   t = int.Parse(Console.ReadLine());
   Console.Write("\n\tAnge ett heltal som ska delas med:\t");
   n = int.Parse(Console.ReadLine());

   VoidMethod.divSafe(t, n);          // Anrop av void-metod
 }
}
```

Matar man in ett heltal skilt från `0` för det andra heltalet ger programmet ovan följande utskrift:

```
Ange ett heltal som ska delas:          27
Ange ett heltal som ska delas med:      4
Säker heltalsdivision:
        27 heltalsdividerad med 4 är 6
```

Inmatning av `0` till det andra heltalet producerar däremot följande dialog:

```
Ange ett heltal som ska delas:          27
Ange ett heltal som ska delas med:      0
OBS! Division med 0:
        Du har matat in 0 för det tal som ska
        delas med. Det går inte att dela med 0.
        Division med 0 är odefinierad.
```

En viktig praktisk konsekvens av **`void`**-metoder är att anropet till skillnad från metoder med returvärde inte behöver – ja inte får – inbäddas i en utskrifts- eller tilldelningssats. **`void`**-metoder anropas helt enkelt med namn och väl definierad parameterlista, om sådan finns. I vårt exempel finns två parametrar:

```
VoidMethod.divSafe(t, n);
```

Eftersom själva metodnamnet inte längre bär något värde, kan det varken skrivas ut eller tilldelas någon variabel. Därför behöver vi inte längre någon variabel som tar hand om returvärdet. Det enda anropet gör är att exekvera den kod som står i metodens kropp, efter att ha överfört parametrarnas värde till metoden. Från metoden kommer inget tillbaka.

Övningar till kapitel 7

7.1 Modularisera lösningen till övn 4.4 (sid 83) som läser in två heltal, gör beräkningar med dem och skriver ut resultaten. Separera en av beräkningarna, t.ex. multiplikationen från kodens andra delar *inmatning* och *utmatning*.

a) Flytta multiplikationen till en metod med returvärde med huvudet **`static int Mult(int a, int b)`** i samma klass som **`Main()`**. Anropa metoden **`Mult()`** från **`Main()`**. Bibehåll alla andra beräkningar. Se upp med att inte placera den nya metoden i **`Main()`**, utan före eller efter.

b) Fortsätt med att flytta metoden **`Mult()`** till en annan klass i samma fil. Anropet ska fortfarande göras från **`Main()`**. Även här: Se upp med att inte placera den nya klassen i den gamla, utan före eller efter.

c) Flytta den nya klassen samt metoden **`Mult()`** till en separat fil.

d) Gör samma sak med alla andra beräkningssätt. Lagra var och en klass med resp. metod i en separat fil. Anropa alla metoder från **`Main()`**.

7.2 Modularisera programmet **`Operator`** (sid 71) genom att skriva dess bearbetningsdel som en ny metod i samma klass. Bibehåll in- och utmatnigsdelen i **`Main()`** och anropa den nya metoden från **`Main()`**. Avgör själv om den nya metoden ska returnera ett värde och om den ska vara statisk. Ge den ett beskrivande namn.

7.3 Vänd om problemet från övn 9.2: Modularisera programmet **`OverloadOp`** (sid 76) genom att flytta bearbetnings- och utmatnigsdelen till en **`void`**-metod. Dvs skriv ett program som läser in tiden i ett antal dagar, anropar **`void`**-metoden som omvandlar tiden till antal år, månader, veckor och restdagar och skriver ut resultaten. Använd för omvandlingen den algoritm som är implementerad i programmet **`OverloadOp`**. Varför är det inte lämpligt här att använda en metod med returvärde?

7.4 Skriv först ett program med endast **`Main()`**-metoden som läser in **`side`** till en kub samt beräknar och skriver ut kubens volym **`side`**$^{\mathbf{3}}$ och dess yta **`6 x side`**$^{\mathbf{2}}$. Flytta sedan dessa beräkningar till två metoder, en för volymen, en för ytan, båda i en separat klass **`Cube`**. Definiera **`side`** som en datamedlem i klassen **`Cube`**. Avgör om metoderna **`Volume()`** och **`Area()`** ska returnera eller vara av **`void`**-typ. Anropa dem från **`Main()`**. Skriv först en variant med statiska metoder, byt sedan till icke-statiska metoder. Testa båda varianter. Avgör slutligen själv vilken variant som ska föredras om lösningen ska vara objektorienterad.

7.5 Varför ger följande program kompileringsfel? Åtgärda felet genom att flytta på kod, utan att ta bort någon klammer och utan att ha tomma klamrar:

```
using System;
class Ovn_7_5
{
   static void Main()
   {
      {
         int t = 30;
      }
      Console.WriteLine("t = " + t);
   }
}
```

7.6 Modularisera programmet **`MiniSort`** (kap 6, sid 116) efter eget godtycke.

7.7 Modularisera programmet **`OverloadOp`** (kap 4, sid 76) efter eget godtycke.

7.8 ***Collatz problemet (projekt)*** * Skriv följande pseudokod till ett C# program:

```
Läs in ett positivt heltal (startvärde)
Så länge talet ≠ 1 REPETERA:
   OM talet är udda
      multiplicera med 3, addera 1
   ANNARS
      dividera talet med 2
   Skriv ut talet
```

För ≠ (inte lika med) kan du skriva C# koden **`!=`** och för att avgöra om **`tal`** är udda, koden **`if (tal % 2 == 1)`**. Testa programmet för startvärdena 3, 6, 7, 13 och 50. Testa även gärna större startvärden.

Studera de talföljder som uppstår. Fundera på varför alla startvärden slutar med 1 oavsett hur stora de är. Förmodan är att det är sant generellt, vilket dock hittills inte har kunnat bevisas matematiskt.

7.9 ***Tillägg till Pyramiden (projekt)*** Modularisera projektet *Pyramiden* från övn. 6.11 (sid 155) genom att flytta koden som bestämmer det tillåtna antalet rader 1-13 till en metod som definieras i en separat klass och anropas från **`Main()`** innan pyramiden ”ritas”.

* Känt även som *Collatz förmodan* eller *(3n+1)-problemet* och kallat efter den tyske matematikern *Lothar Collatz* (1910-1990) som ställde upp det när han var student. Collatz var Professor för Tillämpad Matematik vid Hamburgs Universitet på 60-talet.

Att talföljderna i Collatz problemet slutar med 1 för *alla* startvärden, är matematiskt hittills obevisat. Men man kan testa: Kör koden till denna algoritm i appen **app.mattekollen.se → En mobil pythonmiljö**. Där kan du själv övertyga dig om att körningen slutar med 1 för vilket startvärde som helst.

7.10 ***Kalkylatorn (projekt)*** I detta projekt ska du komplettera kalkylatorn som presenterades i klassen **`Switch`** i kap 6.4, sid 125, med ytterligare funktionalitet. Du ska skriva en klass **`Calculator`** som stödjer funktionalitet för addition, subtraktion, multiplikation, division och potensiering av två tal precis som den ursprungliga kalkylatorn i **`Switch`**-klassen. Funktionaliteten som du ska lägga till är att kalkylatorn skall kunna ange det största och minsta av två inmatade tal. Dessutom ska din kalkylator vara igång kontinuerligt tills användaren väljer att stänga av den, vilket innebär att du måste lägga in hela **`switch`**-satsen i en loop. Dessutom ska de olika räkneoperationerna definieras i separata metoder och anropas i **`switch`**-satsen.

Följande metoder ska definieras i klassen **`Calculator:`**

```
public double Add(double operand1, double operand2)
{
      // Additon av operand1 och operand2
}

public double Sub(double operand1, double operand2)
{
      // operand1 - operand2
      // Även subtraktion av negativa tal ska vara möjligt
}

public double Mult(double operand1, double operand2)
{
      // Multiplikation av parametrarna
}

public double Div(double operand1, double operand2)
{
      // operand1 / operand2
      // Division med 0 får ej förekomma (operand2 != 0)
}

public double Potens(double operand1, double operand2)
{
      // Beräkning av potens: operand1 upphöjt till operand2

}

public double max(double operand1, double operand2)
{
      // Returnera det större värdet av operand1 och operand2
      // Här kan du använda dig av den födefinierade metoden
      // Math.Max(double a, double b) för att snabbt
      // avgöra vilken av operanderna som är större
}

public double Min(double operand1, double operand2)
{
      // Returnera det mindre värdet av operand1 och operand2
      // Math.Min(double a, double b) kan användas
}
```

Programmet skall exekvera kontinuerligt tills användaren väljer att avsluta körningen. För att åstadkomma detta kan du exempelvis använda dig av en **`do`**-sats. Kalkylatorn kan avslutas genom att användaren matar in t.ex. tecknet ’q’ (Quit) istället för en operator.

Du får själv bestämma om du vill placera all kod i en fil eller om du hellre skapar en separat fil för klassen **`Calculator`** med alla ovannämnda metoder och en klass med **`Main()`** i en annan fil som testar klassen **`Calculator`**. Det senare är att föredra.

Det är upp till dig om du lägger in kod för att kunna hantera fel inmatning av operator eller andra felaktiga inmatningar.

Kapitel 8

Klasser, objekt och referenser

8.1 Vad är en klass?

I förra avsnitt besvarades frågan *Vad är en klass?* i allmänna termer av modellering och design. Nu ska vi behandla samma fråga i mer konkreta termer av implementering dvs C# programmering. Den allmänna definitionen *Program = Modell av verkligheten* som introducerades där kommer att preciseras här. Vi börjar med klassbegreppet:

> En klass är kod som på ett generellt och modulärt sätt beskriver en kategori av verkliga eller virtuella saker och ting. Den består av ett antal datamedlemmar och ett antal metoder och används som en mall för att skapa objekt (instanser, exemplar) av denna klass.

En klass är en del av ett program som isoleras och skrivs separat som en namngiven modul för att kunna användas även av andra program. I denna bemärkelse är en klass *modulär*. De saker och ting som den beskriver är objekt i den reala världen som är föremål för datorisering. Varje klass är en abstrakt idé, en definition av alla saker och ting (objekt) som tillhör en viss kategori. I denna bemärkelse är en klass *generell*. Klassens centrala roll för programmering framgår redan av definitionen för *C# program* (sid 43):

> Ett C# program är en samling av klasser, av vilka en och endast en måste innehålla metoden **`Main()`**.
> När programmet körs startar exekveringen i **`Main()`**.

Alla C# program består av klasser som minsta beståndsdel. I alla procedurala programmeringsspråk bildar funktioner och procedurer programmets byggstenar. C# som är objektorienterat har klasser som minsta komponenter som i sin tur kan innehålla funktioner och procedurer vilka då kallas metoder.

Varför klasser?

Frågan är berättigad för nybörjare: Varför ska man krångla till det hela? Kan man inte helt enkelt skriva kod rakt ned i **`Main()`**? Det som i programmeringshistorien gjorde att man behövde klasser var den växande komplexiteten hos program under 70-talet. Programmens storlek var avgörande för den växande komplexiteten. Man förstod att det inte längre räckte till att skriva och testa program som fungerade just då. Man insåg nödvändigheten att med rimliga kostnader även kunna *underhålla* stora program, *förnya* och *vidareutveckla* dem så att de fungerade även i flera år och att de framför allt kunde anpassas till nyuppkomna situationer utan oöverkomliga svårigheter. Men varför måste man använda sig av klasser för att uppnå detta mål? Föreställ dig en verksamhet som dynamiskt växer med tiden, ett expanderande företag eller en organisation med stigande antal medlemmar. Hur organiserar man jobbet? Man genomför arbetsdelning och delegerar uppgifterna. Var och en får en väl definierad specifik arbetsuppgift. Samma sak gör man med program vars kod växer, vilket händer när man utvecklar program efter behov och

behoven bara blir större och större. Man delar upp det stora programmet i mindre moduler för att kunna klara av komplexiteten. I objektorienterad programmering är modulerna *klasser*. Program bryts ned i ett antal klasser. Varje klass beskriver endast *en* kategori av saker och ting som är oberoende av andra och antagligen enklare att koda än det stora programmet. Sedan gäller det att sätta ihop modulerna till det stora programmet. Detta kallas för modularisering på klassnivå.

Testa lösenord som klass

Vi kan i denna lärobok inte komma upp till att kunna presentera sådana komplexa program som motiverade användningen av klasser i programmeringshistorien. Men idén bakom klasser kan även illustreras med de små program som vi brukar visa för att exemplifiera programmeringens koncept. Låt oss realisera klasskonceptet genom att skriva vår första klass: I alla våra program hittills finns all kod rakt nedskriven i **`Main()`** vilket inte ett dugg är objektorienterat, även om C#:s klassbibliotek används flitigt. Men man kan skriva om alla dessa program till objektorienterade varianter. Vi ska demonstrera, hur man gör det, med det program vi avslutade förra kapitlet med. Metodiken är viktigare än resultatet. Samtidigt blir det en avslutning på programserien *Testa lösenord*. Här är klassvarianten:

```
// Password.cs
// Beskriver klassen Password med 2 datamedlemmar & en metod
// Innehåller endast kod som är relaterad till ett lösenord:
// En sträng för inmatning och ett sanningsvärde för test
// Att verifiera sig och skriva ut resultatet (metod)
// Kan kompileras men inte exekveras eftersom Main() saknas
using System;

class Password
{
   String input;                              // Datamedlemmar
   bool wrongPasswd;

   public void Test()                        // Metoden Test()
   {
      do
      {
         Console.Write("\n\tSkriv ditt lösenord:\t");
         input = Console.ReadLine();
         wrongPasswd = !input.Equals("hemligt") &&
                       !input.Equals("HEMLIGT");
         if (wrongPasswd)
            Console.WriteLine("\n\tFel lösenord. " +
                              "Försök igen!");
      } while (wrongPasswd);

      Console.WriteLine("\n\tOK, nu är du inloggad!\n");
   }
}
```

För första gången i våra exempel består ett C# program av *två* klasser. I en separat fil skrivs följande klass där **Main()** skapar ett objekt av den första klassen för att testa den:

```
// PasswordTest.cs
// Testar klassen Password genom att skapa ett objekt av den
// Anropar metoden Test() som är definierad i Password
// Kan både kompileras och exekveras: Utgör med klassen Pass-
// word ett program som gör samma sak som PasswdCapsLock
using System;

class PasswordTest
{
   static void Main()
   {
      Password myPasswd = new Password();     // Objekt skapas
      myPasswd.Test();                        // Metod anropas
   }
}
```

En körning av **PasswordTest** ger följande dialog:

```
        Skriv ditt lösenord:    HEMLIGT

        OK, nu är du inloggad!
```

Med inmatningen **HEMLIGT** i versaler lyckas inloggningen. Inmatningen **hemligt** i gemener skulle ge samma resultat. Alla andra inmatningar kommer att misslyckas.

Observera att klassen **Password** som tillhör programmet och lagras i filen **Password.cs**, endast kan kompileras – antingen separat eller när **PasswordTest** kompileras – men inte köras, för exekveringen ska starta i **Main()** när programmet körs. Och någon **Main()** finns ju inte i **Password**, den finns i **PasswordTest** istället. Men varför är det så? Jo, för det första kan *ett* program ha endast *en* **Main()**-metod. **Password** och **PasswordTest** utgör *ett* program, så **Main()** får skrivas antingen i den ena eller i den andra. Sedan hänger det ihop med hur vi skrivit om progarmmet **PasswdCapsLock** till klassvarianten bestående av två klasser. Nedan följer förfarandet att skriva om ett icke-objektorienterat program där all kod är rakt nedskriven i **Main()**, till en objektorienterad version med klasser [*]:

[*] Generellt rekommenderas inte att skriva om ett icke-objektorienterat program till en objektorienterad version, utan att helst från början tänka och skriva objektorienterat. Men det är kanske lättare sagt än gjort. Vi måste ju först lära oss att modellera objektorienterat. För många kan det vara enklare att (fortsätta) tänka i icke-objektorienterade banor. Men strävan borde vara att lämna denna vana och modellera verkligheten så som den är, nämligen som en samling av objekt – reella eller virtuella – som kommunicerar med varandra genom att anropa varandras metoder. Att vi här ändå beskriver övergången från ett icke-objektorienterat program till ett objektorienterat sådant har endast pedagogiska skäl.

1. Hitta på ett namn för den klass som du ska skriva. Namnet ska vara beskrivande för de objekt som programmet är tänkt för att skapa enligt klassmallen. I vårt fall var det naturligt att kalla klassen för **`Password`**. Här följer vi förstås de vanliga namngivningsreglerna för identifierare som ställdes upp tidigare (sid 59) samt att inleda klassnamn med versaler för att skilja dem från andra identifierare som variabler, metoder osv. Därmed är även namnet på filen som du ska skriva klassen i fastlagd: **`Password.cs`**.

2. Ta de lokala variablerna från den rakt nedskrivna **`Main()`**-koden och deklarera dem som klassens *datamedlemmar*. Med *lokala variabler* menas de som är definierade i **`Main()`**-blocket och är därför giltiga endast *inom* **`Main()`**. Klassen kring **`Main()`** känner inte till dem. I vårt fall är det **`String`**-variabeln **`input`** och **`bool`**-variabeln **`wrongPasswd`** som var lokala variabler i **`Main()`** och blir nu datamedlemmar när vi flyttar dem till klassen **`Password`**. Med denna flyttning sker samtidigt en ändring av dessa variablers status: De får ett större giltighetsområde och blir nu giltiga i hela klassen och därmed i klassens alla metoder. Dessutom blir de automatiskt initierade till vissa default-värden. Så här inleds vår klass:

   ```
   class Password
   {
      String input;
      bool wrongPasswd;
      ...
   }
   ```

   Vi gör så därför att varje lösenord som testas måste ha en strängvariabel för lagring av användarens inmatning och en logisk variabel för lagring av sanningsvärdet som testar loopen och utskriften av testresultatet. Detta är typiskt för alla lösenord.

3. Ta hela koden i den rakt nedskrivna **`Main()`**-metoden och skriv den till kroppen av en eller flera *metoder* i den nya klassen. Hur många och vilka metoder det blir beror på de logiskt sammanhängande funktionaliteter dessa koder har. I vårt fall är det hela **`do`**-loopen och utskriftssatsen efter **`do`** som utgör en enda funktionalitet, nämligen att testa lösenordet och skriva ut testresultatet. Därför gör vi en enda metod av det hela och döper den till **`Test()`**. Vi flyttar alltså hela koden från **`PasswdCapsLock`**:s **`Main()`** till **`Password`**-klassens **`Test()`**-metod som då blir en s.k. **`void`**-metod och får returtypen **`void`** därför att den inte returnerar något värde alls. Förutom **`void`** måste metoden **`Test()`** skrivas som **`public`** för att den ska vara åtkomlig för och ska kunna anropas av klassen **`PasswordTest`** som lagras i en annan fil. Därmed är **`Main()`**-kroppen tömd från all sin kod. Samtidigt är den nya klassen **`Password`** färdigskriven.

4. I den tomma **`Main()`**-kroppen skriver vi nu följande kod istället:

   ```
   Password myPasswd = new Password();
   myPasswd.Test();
   ```

Samtidigt döper vi om den gamla **`PasswdCapsLock`**-klassen till **`PasswordTest`**. Visserligen är frågan om beteckningen inte avgörande, men vi följer en viss konvention som också ger en bättre förståelse om de två klassers samverkan i programmet: **`PasswordTest`** är en slags *test* för klassen **`Password`**, för i dess **`Main()`**-metod – med den första satsen i koden ovan – testas klassen **`Password`** i den bemärkelsen att det skapas ett objekt av denna klass. Alla C#-klasser som skrivs i separata filer och inte själva kan exekveras behöver ha ett program som testar – eller *instansierar* – dem. Man säger så eftersom *instans* är bara ett annat ord för objekt. Annars liknar klassen ett matrecept som ligger i kökslådan och aldrig används. Man kan också jämföra **`PasswordTest`** med en slags drivrutin för klassen **`Password`**. Det är drivrutinen som sätter igång och gör något med klassen, skapar ett objekt av den och – med den andra satsen i koden ovan – anropar det nyss skapade objektets metod **`Test()`**.

Klassen **`Password`** beskriver *kategorin* lösenord som en abstrakt idé utan att skapa ett verkligt lösenord (sid 181). Den är en *mall* för att skapa verkliga lösenord, en föreskrift om hur ett verkligt lösenord med en viss inmatning och ett testvärde *skulle* se ut och hur det *skulle* verifieras *om det skapades*. Ett verkligt, konkret lösenord kallas för *objekt*. Det är objektet som behöver minnesutrymme för att lagras. Klassen definierar inga objekt utan ställer bara till förfogande modellen för framtida objektdefinitioner. Om man byter ut lösenord mot pepparkakor kan man säga att pepparkaks*formen* är klassen och själva pepparkakorna är objekten. Formen behöver ingen pepparkaksdeg – motsvarigheten till minne – den framställs bara en gång medan kakorna kan bakas i tusentals. Även klassen skrivs endast en gång, objekt däremot kan skapas hur många som helst. I exemplet **`PasswordTest`** skapas bara ett **`Password`**-objekt. *Hur* man gör det med det reserverade ordet **`new`** och hur man sedan kan komma åt objektet behandlas i nästa avsnitt. Sedan får vi också reda på varför **`myPasswd`** kan deklareras till *klassen* **`Password`**.

Slutligen kan man undra om det hade varit möjligt resp. rimligt att lagra båda klasser **`Password`** och **`PasswordTest`** i en och samma fil. Svaret är: Möjligt ja, men inte rimligt. Varför? Jo, därför att det går både att kompilera och köra programmet när båda klasser lagras i en fil, följer man bara regeln att döpa filen efter den klass som innehåller **`Main()`**. Men rimligt är det inte, för då går man miste om hela idén med klasser, nämligen modularisering och återanvändning av kod. Meningen med att skriva separata klasser var ju att kunna återanvända koden i andra program. Det kan man inte längre om man stoppar allt i en fil. Då kan man ju lika bra köra med den ursprungliga icke-klassvarianten **`PasswdCapsLock`**.

8.2 Klass som egendefinierad datatyp

Det är inte alls fel att jämföra klasser även med enkla datatyper som t.ex. **int**. Vad är det som gör **int** till en datatyp? Definitionen säger att *datatyp* är en föreskrift om hur en viss typ av data ska lagras i datorn, hur mycket minne den tar och vilka operationer man får utföra med den. För **int** är det **+**, **-**, *****, **/** och **%**. Även explicit typkonvertering av en **float** till en **int** eller någon annan enkel datatyp, är en operation som är implementerad i datatypen. Minnesstorleken – ett fast värde i antal bytes – som måste lagras som ett konstant värde i datatypen, kan jämföras med *datamedlem*. Vad som får *göras* med värden av en viss datatyp och *hur* allt detta ska göras är *metoder* som är definierade för datatypen. Att samla data och metoder som är relaterade till dessa data, i en enhet, är samma koncept som ligger bakom klassbegreppet.

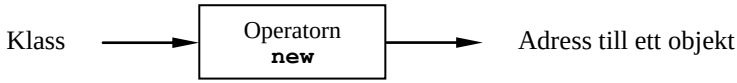
Den första satsen i **PasswordTest**-klassens **Main()** är:

```
Password myPasswd = new Password();
```

För att förstå vad den gör förenklar vi först läsligheten genom dela upp den i sina beståndsdelar:

```
Password myPasswd;
myPasswd = new Password();
```

Vi börjar med att förklara det reserverade ordet **new** som förekommer i den andra satsen ovan. **new** är en fördefinierad minnesallokeringsoperator i C#. Generellt utför en *operator* en operation och returnerar ett värde. Den tar in en parameter (eller flera), gör något med den (dem) och ger tillbaka ett värde. **new** tar in en klass som parameter, allokerar (reserverar) minne av den storlek som klassen föreskriver, kopierar ett exemplar av klassens datamedlemmars värden till det nyss skapade minnet och returnerar det allokerade minnesutrymmets *adress*:



I exemplet ovan skapar **new** genom att sättas framför klassen **Password** ett **Password**-objekt med datamedlemmarna **input** och **wrongPasswd** och initierar dem automatiskt. Skillnaden till traditionell allokering med vanlig definition av variabler av enkel datatyp är att **new** allokerar minne under programmets körning – *at run time* medan vanlig definition av variabler av enkel datatyp allokerar minne under kompileringen – *at compile time*. Denna teknik kallas *dynamisk minnesallokering*. På så sätt skapar **new** ett objekt enligt klassen som mall. När objektets metod anropas utökas det allokerade minnesutrymmet så att även metodens data (parametrar, returvärdet och lokala variabler) får plats vilket är möjligt pga att minnesallokeringen sker dynamiskt.

Eftersom operatorn **new** returnerar en adress måste den tilldelas en variabel som kan ta emot och lagra adresser, för att sedan kunna referera till data som lagras vid denna adress. Med data menas det objekt som **new** skapar. Ingen av våra hittills kända datatyper kan lagra adresser. Därför finns det i C# en helt ny typ av variabler som är konstruerade endast för att lagra adresser: *Referensvariabler* eller kort *referenser*.

Vad är en referens?

En *referens* är en ny typ av variabel, en förkortning för *referensvariabel*:

> Referens är en variabel vars datatyp är en klass.

Ett exempel är variabeln **myPasswd** som ovan deklarerats till datatypen **Password**. Lyckligtvis behöver vi varken bry oss om hur adresshantering sköts internt i C# eller lära oss hexadecimala tal – formatet för lagring av adresser i datorn. Vi behöver inte heller som i C++ lära oss pekare, för i C# finns inga pekare. Adresshanteringen sköts automatiskt. Det enda som vi måste känna till är användningen av referensvariabler för att via dem kunna komma åt våra objekt, för det är det enda sättet att göra det i C#: Objekt kan bl.a. skapas med operatorn **new**. De har inga direkta namn som vanliga variabler av enkel datatyp utan kan endast refereras indirekt med referensvariabler. Det är jämförbart med tyglar till en häst eller fjärrkontrollen till en TV, båda är lätthanterade men avgörande verktyg för styrning av tunga objekt. Även en referens är jämfört med stora objekt, minnesekonomisk och tar så litet minne som en **int**: 4 bytes.

Med satsen **Password myPasswd;** skapas inget objekt av typ **Password** utan endast en referens till ett sådant objekt. Så länge man inte initierar denna variabel till ett värde behandlas den som vilken oinitierad variabel som helst, leder t.ex. till kompileringsfel när den används. En referensvariabels värde däremot kan per definition bara vara en adress. En sådan levereras av operatorn **new**:s returvärde efter att med koden **new Password()** har allokerat minne för och därmed skapat objektet. Först kopplingen av referensdefinitionen till skapandet av objekt med hjälp av tilldelning ger resultat:

```
Password myPasswd = new Password();
```

Följande generell struktur är formen till kanske de mest förekommande satserna i objektorienterade C# program:

> Klass referensvariabel **=** **new** Klass()**;**

Själva **new**:s syntax står höger om tilldelningstecknet. Till vänster definieras en referensvariabel som tar hand om **new**:s returvärde. Här ser man också att operatorn **new** inte behöver parenteser kring sin parameter Klass(). Observera att klasserna på tilldelningens båda sidor måste vara identiska: **new Password()** allokerar minne

för lagring av ett **`Password`** och returnerar en adress till ett **`Password`**. Därför måste den också tilldelas en variabel av typ referens-till-**`Password`**. Tilldelning till en referens-till-annan klass ger kompileringsfel. Man kan, om man inte har behov av att komma åt objektet senare, även skapa *anonyma* objekt direkt när man behöver dem. T.ex. kan anropet **`myPasswd.Test();`** i **`PasswordTest`** även ersättas av **`new Password().Test();`** vilket än en gång visar att det är inget annat än **`new`** som *skapar* objektet. Testa gärna!

När det gäller vanliga variabler av enkel datatyp hänvisar vi till minnescellerna direkt med *variabelnamn*. När det gäller objekt gör vi det indirekt med deras *adresser* i form av objektens referenser. Vilken metod som är ”direkt” och vilken ”indirekt” kan man ha olika åsikter om. När man vant sig vid att använda referenser kan man t.o.m. tycka att hanteringen av data via adresser är det naturliga sättet, vilket inte är någon dum idé med tanke på att variabelnamn ändå är en slags mjukvarulänk till hårdvarans minnesadress.

Det enda som vi inte förklarat ovan är de tomma parenteserna efter klassnamnet: I den allmänna formen Klass() och i exemplet **`Password()`**. De får absolut inte utelämnas även om de är tomma. De anropar *klassens konstruktor*, ett koncept som har att göra med objektets initiering.

8.3 Gissa tal som klass

Nu kan vi sammanföra våra kunskaper om klasser, objekt och referenser, för att skriva en klassvariant till spelserien *Gissa tal* som introducerades på sid 128 och vidareutvecklades sedan i ett antal steg. Samtidigt tillämpar vi idéerna om modularisering och återanvändning av kod: Klassen **Random** samt dess metod **Next()** används – från biblioteket **System** – för att initiera programmets hemliga tal med ett slumpvärde mellan **1** och **20**. Följande klass anropar **Next()** för att förse varje objekt av den med ett nytt slumptal:

```
// GuessNo.cs
// Klass som implementerar Gissa tal-spelet med två datamed-
// lemmar för det gissade och hemliga talet och en metod med
// spelets regler som hjälper användaren att gissa rätt
using System;

class GuessNo
{
  int guessedNo;
  int secretNo = new Random().Next(1, 21); // Anrop av metod
                                           // i anonymt objekt
  public void Play()
  {
    do
    {
      Console.Write("\n\tGissa ett tal mellan 1 och 20 " +
                                    "(Avsluta med 0):\t");
      guessedNo = int.Parse(Console.ReadLine());
      Console.Write("\n\t");
      if (guessedNo == 0)
      {
         Console.WriteLine("Avbrott: Programmets hemliga " +
                           "tal var " + secretNo + '\n');
         break;                          // Bryter do-loopen
      }
      if (guessedNo < secretNo)
         Console.Write("För LITET, försök igen!\n");
      if (guessedNo > secretNo)
         Console.Write("För STORT, försök igen!\n");
    } while (guessedNo != secretNo);

   if (guessedNo == secretNo)
      Console.Write("\aGrattis, du har gissat rätt!\n\n");
  }
}
```

I klassen **GuessNo** anropas metoden **Next()** för att initiera datamedlemmen **secretNo** med ett slumptal i intervallet [1, 20]. Anropet görs med ett s.k. *anonymt* objekt **new Random()** dvs ett objekt som skapas med **new** utan att man tilldelar minnesadressen till en referens. Dessutom är det inbakad i en tilldelningssats därför

att **Next()** är en metod med returvärde. Att initiera en datamedlem direkt i klassen är inte så vanligt. I det här fallet är det motiverat eftersom variationen i slumpen ger olika värden för de olika objekt som kommer att skapas av klassen **GuessNo**. Annars hade det lika bra varit möjligt att initiera **secretNo** i metoden **Play()**. Dessutom ska varje spelomgång – och därmed varje objekt – ha endast ett hemligt tal som användaren ska gissa fram sig till. Att initieringen görs i klassen betyder inte att datamedlemmen **secretNo** är en klassvariabel. Då borde den vara deklarerad som **static** vilket den inte är. Utan den är en instansvariabel precis som den andra datamedlemmen **gissat** som initieras först i metoden **Play()**.

Klassen ovan kan kompileras för sig, men inte exekveras eftersom **Main()** saknas. Programmet i sin helhet består av *två* klasser (moduler) i två filer som lagras i en mapp: Klassen **GuessNo** och följande klass som ”testar” den i den bemärkelse att **Main()** är placerad i den och ett **GuessNo**-objekt skapas samt dess metod **Play()** anropas där. Man kan också säga att ”test”klassen (nedan) är en slags ”drivrutin” eller ”drivklass” för klassen **GuessNo**. Den sätter igång (”driver”) hela programmet. Utan den fungerar ingenting. Men huvudjobbet görs ändå av klassen **GuessNo** eftersom den innehåller spelets egentliga kod i form av metoden **Play()**.

```
// GuessNoTest.cs
// Testar klassen GuessNo genom att skapa ett objekt av den
// Anropar objektets metod Play()
// Utgör med klassen GuessNo ett program som gör samma sak
// som Gissa tal-spelets tidigare versioner
using System;

class GuessNoTest
{
   static void Main()
   {
      GuessNo g = new GuessNo();            // Objekt skapas
      g.Play();                             // Metod anropas
   }
}
```

Allt går ut på att anropa **void**-metoden **Play()** som implementerar spelets regler och genomför det i ordande former dvs låter användaren – med lite hjälp – gissa flera gånger tills den gissat rätt eller på begäran avslöjar spelets hemliga tal. Den innehåller i huvudsak en loop som håller igång, styr, kontrollerar och avslutar en spelomgång. Vi har tagit över koden från *Gissa tal*-spelets senaste version **GuessNEG** och skrivit den i klassen **GuessNo** på förra sidan efter att ha eliminerat allt som har att göra med den logiska operatorn NEGATION för att ha fokus på programmets objektorienterade aspekter.

Men metoder i C# och därmed även metoden **Play()** kan endast anropas utifrån ett objekt eller en klass. När det sker från en klass måste de vara statiska vilket **Play()** inte är (se klassen **GuessNo**). Därför behöver det ett objekt för att anropas.

Ett sådant skapas i klassen **GuessNoTest** med **new GuessNo()** och tilldelas referensvariabeln **g**. Sedan anropas metoden med denna objektreferens: **g.Play()**, dvs metoden anropas *i* det objekt som **g** refererar till. Man skulle kunna anropa **Play()** även med ett anonymt objekt i en enda sats: **new GuessNo().Play();** och därmed spara undan referensvariabeln vilket inte just förbättrar kodens läslighet.

Mer om metoder kommer vi att lära oss i nästa kapitel (sid 157). Då kommer vi även fullt förstå modifierarna **public** och **void** i metoden **Play()**:s huvud.

En körning av **GuessNoTest** kan se ut så här:

```
Gissa ett tal mellan 1 och 20 (Avsluta med 0):  12
För STORT, försök igen!
Gissa ett tal mellan 1 och 20 (Avsluta med 0):  6
För LITET, försök igen!
Gissa ett tal mellan 1 och 20 (Avsluta med 0):  9
För LITET, försök igen!
Gissa ett tal mellan 1 och 20 (Avsluta med 0):  10
För LITET, försök igen!
Gissa ett tal mellan 1 och 20 (Avsluta med 0):  11
Grattis, du har gissat rätt!
```

Har man efter ett tag ingen lust att gissa vidare och vill avsluta, kan man mata in **0**. Man får då reda på programmets hemliga slumptal vid just den aktuella körningen:

```
Gissa ett tal mellan 1 och 20 (Avsluta med 0):  0
Avbrott: Programmets hemliga tal var 16
```

Körresultaten ovan är – bortsett från slumpens variation – identiska med dem från prorammet **GuessDoRand** på sid 137, därför att **GuessNoTest** är en klassvariant av samma program.

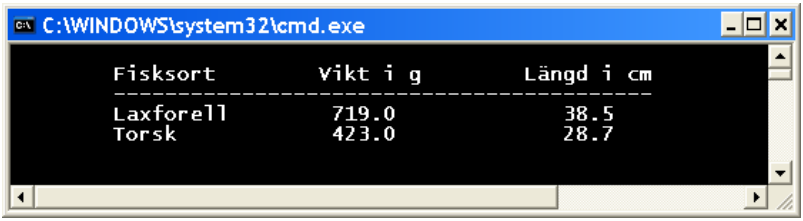
Övningar till kapitel 8

8.1 Skriv ett program som består endast av klassen **`All_in_Main`** som i sin tur innehåller endast **`Main()`**-metoden. Läs in radien **`r`** till en cirkel och beräkna samt skriv ut cirkelns area π **`r`**^{2} och dess omkrets 2π **`r`**, där $\pi = 3.14159$. Du kan använda konstanten **`Math.PI`** från C#:s klassbibliotek för π. Programmet ska inte vara objektorienterat eftersom du inte skapar några objekt, utan endast lokala variabler (radie, area, omkrets). Programmet ska inte heller vara modulariserat eller proceduralt eftersom all kod (inmatning-bearbetning- utmatning) finns i en enda metod **`Main()`** som definieras i en klass. Dessa steg ska tas i de efterföljande två övningarna. Deklarera alla variabler till **`double`**.

8.2 Modularisera programmet **`All_in_Main`** från övn 8.1 på metodnivå, dvs: Flytta bearbetningsdelen dvs beräkningen av area och omkrets ur **`Main()`** till separata metoder **`Area()`** och **`Circumference()`**, men stanna i samma klass. Döp om klassnamnet till **`Procedural`**. I **`Main()`** ska finnas kvar variabeln för radien, inmatning, utmatning och anropen av **`Area()`** och **`Circumference()`**. Förse de nya metoderna med en parameter som överför radiens värde från **`Main()`** till dem. Välj olika namn för den aktuella än för den formella parametern. Dessutom ska **`Area()`** och **`Circumference()`** returnera ett **`double`**-värde och vara statiska. För att testa, mata in **`1`** för radien. Då ska arean bli π pga π **`r`**$^2 = \pi$ och omkretsen bli 2π pga 2π **`r`** $= 2\pi$.

8.3 Modularisera programmet **`All_in_Main`** från övn 8.1 på klassnivå, dvs: Dela upp programmet i två klasser, lagrade i två separata filer. Kalla den ena klassen för **`Circle`**, den andra för **`CircleTest`**. Samla all information om *begreppet cirkel* i klassen **`Circle`**, dvs: Deklarera radien **`r`** som datamedlem samt **`Area()`** och **`Circumference()`** som metoder. Ta bort från metoderna både **`static`** och parametern för radien. Den andra klassen **`CircleTest`** ska endast innehålla metoden **`Main()`**. Skapa i den ett objekt av klassen **`Circle`**. Läs in ett värde till objektets datamedlem **`r`** och anropa samt skriv ut returvärdena till objektets metoder **`Area()`** och **`Circumference()`**. Båda klassfiler borde ligga i samma projekt.

8.4 Skriv en klass **`Fish`** som beskriver en fisk med datamedlemmarna **`sort`**, **`weight`** och **`size`**. Testa din klass i en annan klass **`FishTest`** i en separat fil som endast innehåller metoden **`Main()`** där två objekt av klassen **`Fish`** skapas. Tilldela det första objektets datamedlemmar värdena *Laxforell*, 719 (gram) och 38,5 (cm). Enheterna gram och cm behöver inte anges. Välj själv andra värden till det andra objektets datamedlemmar. Skriv ut dessa värden till konsolen i en tabell av typ:



8.5 Ta klassen **`Fish`** från övn 8.4. Förse den med en metod som beräknar priset på fisken oberoende av sort, t.ex. 7,25 kr per hekto. Lägg till även en metod som beräknar och returnerar frakten utifrån fiskens vikt och längd genom att t.ex. multiplicera en viss kostnadsfaktor, säg 0,02, med vikten, en annan, säg 0,1, med längden och addera dem. Metoderna ska returnera priset och frakten i hela kronor utan ören. Anropa metoderna från klassen **`FishTest`**:s **`Main()`**-metod för de två **`Fish`**-objekten. Lägg till nya rubriker *Pris* och *Frakt* i tabellen ovan och skriv ut deras värden till tabellens två rader.

8.6 Modifiera programmet från övn 8.5 så att datamedlemmarnas värden inte hårdkodas utan läses in. Utskriften ska skickas till konsolen och läggas till tabellen ovan. Skriv din kod så att den lätt kan generaliseras så att man kan mata in flera fisksorter med hjälp av en loop och en array av referenser till **`Fish`**-objekt som vi kommer att lära oss senare. Dessutom ska programmet kunna modifieras till att skriva ut till en tabell i en fil eller en databas istället för att skriva till konsolen.

8.7 Deklarera en klass **`Triangle`** med datamedlemmarna **`side_a`**, **`side_b`**, **`side_c`**, **`height_b`** av typ **`int`** och metoderna **`Area()`**, **`Circumference()`**. Skapa i en annan klass som innehåller **`Main()`**, ett objekt av klassen **`Triangle`** och tilldela datamedlemmarna värden. Anropa metoderna och skriv ut denna triangels area och omkrets. Skapa en andra referens som pekar på samma objekt och anropa metoderna samt skriv ut deras returvärden med denna referens. Du borde få samma resultat som med den första referensen. Anropa sedan metoderna **`Area()`** och **`Circumference()`** med två anonyma objekt (utan referenser). Kolla om du får de förväntade resultaten som är baserade på objektens default-initiering. Sist, peka om **`Triangle`**-objektets första referens till **`null`** och försök att anropa metoderna med denna referens. Vad händer?

8.8 ***Automaten (projekt)*** Skriv ett program som simulerar interaktionen med en automat. Följande klass beskriver i stora drag det som pågår i automaten efter att användaren lagt in pengar och valt en vara:

```
class Automat
{
    string productName;
    double price;
```

```
    double payment;
    double change;

    void Automat_action(double money,
                                    char product)
    {
       switch(product)
       {
          . . .
       }
       payment = money;
       change = payment - price;
    }

    void Change_in_coins()
    {
          . . .
    }
}
```

switch-satsen i metoden **Automat_action()** ska tilldela datamedlemmarna **productName** och **price** värden beroende på valet av vara och skriva ut ett meddelande om inlagt belopp samt varan som ska levereras. Testa klassen **Automat** i en annan klass (testklassen) i en separat fil som endast innehåller metoden **Main()**.

Börja i **Main()** med att skriva ut en meny över alla varor samt priserna. För enkelhets skull kan du t.ex. börja med en meny för en kaffeautomat. Låt sedan användaren lägga in pengar. Läs in beloppet till en **double**-variabel. Låt användaren även välja en vara. Sedan kan ett objekt av den ovan deklarerade klassen **Automat** skapas och metoden **Automat_action()** anropas. Vid detta anrop skickas de inlästa värdena till det inlagda beloppet och den valda varan som aktuella parametrar till **Automat_action()**. Efter att objektet skapats och datamedlemmarna initierats, kan metoden **Change_in_coins()** anropas.

Komplettera programmet med att ta hand om en eventuellt felaktig eller otillräcklig betalning från användarens sida.

Metoden **Change_in_coins()** är till för att dela upp växeln i ”tillåtna” myntslag (endast 10-kr, 5-kr, 1-kr och 50-öringar [*]) och skriva ut hur många

[*] Inkluderingen av myntslaget 50-öring som inte längre finns i det svenska myntsystemet, beror inte på nostalgi utan på internationalisering. Vi vill hålla möjligheten öppen för en översättning till andra språk resp.för en övergång till Euro eller andra valutor. Behandlingen av en halv enhet (50-öring) vid omvandling av växelbeloppet till automatens tillåtna myntsystem inkluderar en programmeringsteknisk finess som är värd att lära sig. Så kan t.ex. våra program även användas för Euron där 50 cent ersätter 50-öringen.

av varje ”tillåtet” myntslag som ska ges tillbaka. Växelbeloppet måste omvandlas till detta mynt”system”. För att åstadkomma det, använd följande algoritm:

Algoritm för omvandling av ett belopp till olika myntslag

Eftersom denna algoritm endast fungerar för heltal, måste **`change`** som är ett belopp i kronor och ören av typ **`double`**, först räknas om till ett rent öre-belopp av typ **`int`**, vilket kan göras genom att multiplicera det först med **`100`** och sedan avrunda resultatet till heltal:

```
int total = (int) Math.Round(change * 100);
```

I fortsättningen kommer alltså den givna växeln att stå som ett *öre*belopp i **`int`**-variabeln **`total`**. Anledningen till konverteringen till **`int`** i satsen ovan är att den fördefinierade metoden **`Round()`** som avrundar till närmaste heltal, ändå returnerar ett värde av typ **`double`**.

1. För att få antalet 10-kronor heltalsdivideras **`total`** med **`1000`** eftersom 10-kronor är **`1000`** ören:

```
int ten = total / 1000;
```

Hur många gånger ryms **`1000`** – eller 10-kronor – i **`total`**? Det antalet tilldelas till **`ten`**. Eller med andra ord: **`1000`** dras av från **`total`** så många gånger tills resten blivit mindre än **`total`**. Det antalet som tilldelas till **`ten`** blir antalet 10-kronor. Divisionen ovan är inte vanlig division utan heltalsdivision (sid 75) eftersom både **`total`** och **`1000`** är heltal. Dvs **`total`** divideras med **`1000`**, resultatet tas, resten ignoreras, t.ex. **`6975/1000`** ger **`6`**. Resten **`975`** ignoreras här, men används i fortsättningen.

2. För att få antalet 5-kronor divideras just *resten* som blev kvar från punkt **1** med **`500`** eftersom 5-kronor är **`500`** ören:

```
int five = (total % 1000) / 500;
```

”Resten som blev kvar från punkt **1**” är just **`(total % 1000)`**. Här används modulooperatorn **`%`** (sid 75). T.ex. **`6975 % 1000`** ger **`975`**. Efter att ha dragit av alla 10-kronor från **`total`** divideras resten med **`500`** för att få reda på hur många 5-kronor som finns i **`total`**. T.ex. **`975/500`** ger **`1`**. Resultatet av denna division ges till **`five`**, resten ignoreras och används i fortsättningen.

I ytterligare tre steg kan följande satser för beräkning av antalet 1-kr (**`one`**), 50-öringar (**`half`**) och resten i öre (**`rest`**) skrivas, när mönstret i algoritmen (förhoppningsvis) har trätt fram. Om 50-öringar läs fotnoten på sid 193.

```
int one  =  ((total % 1000) % 500) / 100;
int half = (((total % 1000) % 500) % 100) / 50;
int rest = (((total % 1000) % 500) % 100) % 50;
```

Man tar förra stegets formel, ersätter `/` med `%` och lägger till en heltalsdivision med den nya enhetens örebelopp. I det allra sista steget däremot, där man är ute efter allra sista resten i öre, måste `%` användas hela vägen. Självklart är restörebeloppet inte av praktiskt intresse när automaten inte kan spotta ut det.

Kapitel 9

Array

9.1 Vad är en array?

Ordet *array* betyder i engelskan *ordnad skara* eller *ordnad uppställning* (*battle array* = stridsordning). Som datalogisk term hittar man i litteraturen begreppen *fält*, *vektor*, *matris*, *lista*, … . Ibland används även *härledd datatyp* som syftar åt att den är baserad på en *annan* datatyp. Vi kommer dock att använda endast termen *array*.

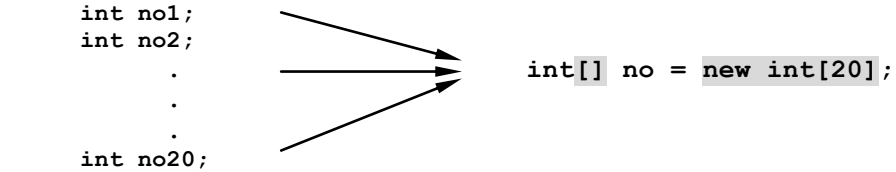
En **array** är en sammansatt datatyp – en ordnad mängd av variabler av *samma* datatyp grupperade under *samma* namn.

En array består av ett antal **element**. Elementens position kallas för **index**.

Index är synonym till nummer och specificerar varje elements position i arrayen för att ”adressera” elementet. Elementen kan i sin tur vara av enkel, sammansatt eller av referenstyp. Så man kan även – med hjälp av referenser – gruppera objekt till en array. En array är den enklast tänkbara sammansatta datatypen. Som exempel tar vi en array som är sammansatt av den enkla datatypen **`int`**. Varje element i en sådan array kan betraktas som en indexerad dvs numrerad variabel av typ **`int`**.

Anta att vi vill definiera 20 heltalsvariabler. Hittills behövde vi skriva 20 satser för att göra det. Men array ger oss möjligheten att göra samma sak med endast *en* sats:

Hittills: enkel datatyp **`int`**: **Nu:** **`int`**-*array* med *referens*:



Vi definierar *en* variabel **`no`** av datatypen **`int[]`**, använder **`new`** och lägger till informationen om antalet element inom hakparentes: **`[20]`**. Men vad är **`int[]`** för datatyp? Det reserverade ordet **`new`** avslöjar att det är ett *objekt*. **`new`** allokerar minnesutrymme för ett objekt bestående av 20 **`int`**-värden och returnerar den sammanhängande ”minneskedjans” adress – närmare bestämt adressen till dess första cell – till referensvariabeln **`no`**. Därmed har vi att göra med en *referenstyp*: Datatypen **`int[]`** är en *referens till en* **`int`**-*array* som i själva verket är ett *objekt*. För att göra det ännu tydligare kan man skriva den nya koden även i två separata satser:

```
int[] no;
no = new int[20];
```

Det är inte den första utan den andra satsen, närmare bestämt koden **`new int[20]`** som *skapar* själva arrayen. Därför står också storleken **`20`** där det behövs, nämligen

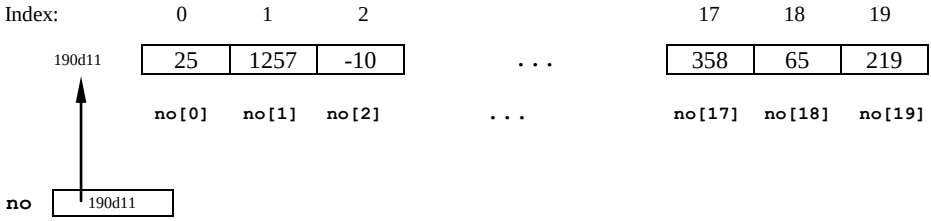
i satsen där **new** allokerar minne. Typiskt för array är *hakparenteserna* **[ ]**, på eng. *brackets*. I satserna ovan har **[ ]** två olika betydelser: I den första satsen specificerar **int[]** variabeln **no**:s *datatyp* som en referens till en **int**-array, i den andra satsen innehåller **[20]** arrayens *storlek*. Referensvariabeln **no** ersätter de 20 vanliga **int**-variablerna **no1**, **no2**, …, **no20**, vilket medför en stor effektivitet i koden. Tänk dig att det är inte **20** utan fler data vi vill jobba med. **no** pekar fysiskt på det *första* elementet av arrayen som allokeras i ett sammanhängande minnesutrymme. Därför kan man komma åt de *andra* elementen via *indexering* som är bara ett annat namn för numrering.

Indexering i en array

Låt oss anknyta till exemplet ovan där både arrayen och dess referens **no** definieras:

```
int[] no = new int[20];
```

Låt oss ytterligare anta att vissa *värden* – de som visas i bilden nedan – har tilldelats arrayens element efter satsen ovan. Eftersom elementen lagras i ett sammanhängande minnesområde uppstår följande minnesbild av arrayen i datorns RAM:



Medan själva arrayens allokering (den övre delen) görs av **new int[20]**, allokeras minnescellen **no** (den undre delen) av **int[] no**. Kopplingen mellan dem görs av tilldelningsoperatorn, vilket gör att arrayens adress (t.ex. 190d11 – ett hexadecimalt tal) som **new** har genererat, hamnar i minnescellen **no**. Den så uppkomna situationen innebär att **no** *pekar på* eller *refererar till* arrayen. Under arrayens minnesceller står varje elements värde: **no[0]** ger den första minnescellens värde 25 som har index **0**, **no[1]** ger den andra minnescellens värde 1257 som har index **1** osv. **no[0]** lagras vid adressen till arrayens första minnescell. **no[1]** lagras vid adressen till den andra minnescellen. **no[2]** lagras vid adressen som ligger **2** x 4 bytes längre bort från **no** osv. Adressering i RAM sker byte-vis. Avgörande för denna indexeringsteknik är att en array alltid allokeras i ett *sammanhängande* minnesområde. Indexnumreringen börjar med **0** och inte med **1**. Det gäller:

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

Med *position* menas numret som *människan* använder för att numrera elementen. Människor är vana vid att påbörja numreringen av saker och ting med 1. Med *index* menas numret som *datorn* använder för samma sak. C# och de flesta andra pro-

grammeringsspråken börjar numreringen av index i en array med **0**. Tillämpad på exemplet: Det 1:a elementet i den array som **no** refererar till har *värdet* 25 och *index* 0: Positionen är 1 medan indexet är **0**. Det 2:a elementet (värdet 1257) har index **1** och koden **no[1]**, det 3:e elementet (värdet –10) har index **2** och koden **no[2]** osv. Det n:e elementet har alltid index n-1. Därför har också det 20:e elementet (värdet 219) index **19**.

Det är avgörande när man arbetar med array och är samtidigt felkälla nr 1 – om man glömmer det – att hålla isär det mänskliga sättet att numrera som börjar med 1 från C#-kodens sätt som börjar med 0. I exemplet ovan har vi definierat en array av 20 heltalselement med referenserna **no[0], ..., no[19]**. Antalet element är 20. Indexen däremot går från **0** till **19**. Felkälla nr 2 är att förväxla en arrayelements *index* med dess *värde*: Det sista elementet i exemplet ovan har index **19**, men värdet 219. Man har alltid med *två* tal att göra, index (position) och värde (innehåll). Det gäller att hålla isär positionen från innehållet.

Tre egenskaper skiljer objekt från array:

- Indexering
- Allokering i ett sammanhängande minnesområde
- Alla arrayelement har samma datatyp.

Annars behandlas array i C# som objekt: Båda måste skapas med **new** och man kan komma åt båda endast med referensvariabler. Båda initieras till default-värden även om de kan förekomma som lokala variabler i metoder.

Deklaration och initiering av en array

Här testas allt vi sagt hittills om array speciellt indexregeln. Utöver det visas ytterligare en egenskap hos array som relaterar den till objekt, nämligen en egenskap **Length** som lagrar arrayens storlek när den skapas. Programmet demonstrerar också vad som händer om man överskrider arrayens maximala index: Man kan kompilera, men inte exekvera: arrayens allokering sker vid *run time*.

```
// Array.cs
// Definierar en array, skriver ut arrayens storlek,
// initieringsvärdena 0 och de nya tilldelade värdena
// Överskridning av arrayens index leder till exekveringsfel
using System;

class Array
{
   static void Main()
   {
      int[] no;                  // Deklarerar referensen no
                                 // utan att skapa arrayen
      no = new int[4];           // Skapar arrayen vars adress
                                 // tilldelas referensen no
//    int[] no = new int[4];     // Alternativt i EN sats
```

```
      Console.Write("\n\tArrayens storlek:\t\t");
      Console.WriteLine(no.Length);
      Console.Write("\n\tArrayens default-initiering:\t");
      foreach (int element in no)
         Console.Write(element + "\t");
      no[0] = 64;              // Tilldelar 1:a elementet
      no[1] = 86;              // värdet 64 osv. Överskriver
      no[2] = 34;              // default-initieringen
      no[3] = -6;
      Console.Write("\n\n\tArrayen efter tilldelning:\t");
      foreach (int element in no)
         Console.Write(element + "\t");
      Console.WriteLine(
        "\n\n\tÖverskridning av arrayens index leder till " +
        "programavbrott:\n\n\t\tno[4] inte definierad\n\t"  +
        "\tIndex 4 överskrider gränsen: Exekveringsfel!")   ;
      no[4] = 1;               // no[4] kan kompileras, men
   }                           // leder till exekveringsfel
}
```

Inte alla satser i programmet **Array** exekveras. Det blir avbrott när den kompilerade koden **no[4]** i allra sista satsen ska exekveras där index **4** överstiger arrayens tillåtna maximala indexgräns som är **3** därför att **new** i början av programmet allokerar endast **4** minnesceller åt arrayen, nämligen de med index **0**, **1**, **2** och **3**. Någon minnescell med index **4** är inte allokerad. Därför kan vi inte heller referera till den med **no[4]**. Men eftersom arrayens allokering sker med **new** och därmed under exekveringstid (eng. *run time*) leder detta till exekveringsfel, medan kompilatorn godtar den syntaxmässigt korrekta koden **no[4]**. Programmet **Array** ger följande utskrift när man kör det:

```
   Arrayens storlek:               4

   Arrayens default-initiering:    0       0       0       0

   Arrayen efter tilldelning:      64      86      34      -6

   Överskridning av arrayens index leder till programavbrott:

            no[4] inte definierad
            Index 4 överskrider gränsen: Exekveringsfel!

Unhandled Exception: System.IndexOutOfRangeException: Index
was outside t bounds of the array.
   at Array.Main() in C:\Programmering\Programmering 2\2
OOP\Array.cs:line 32
```

Vi drar slutsatsen:

> Att referera till icke-definierade element i en array leder till exekveringsfel.

Man kan även säga att C#-interpretatorn (VM) kontrollerar indexgränserna och inte tillåter åtkomsten till icke-allokerade minnesplatser, vilket ur allmän datasäkerhetssynpunkt är en fördel. Programmen blir stabilare. Andra programmeringsspråk som C++ har i detta avseende en mer liberal attityd. Där ligger ansvaret för kontroll av indexgränserna helt och hållet hos programmeraren.

foreach-satsen

Denna sats som används i programmet **Array** (sid 199) är en ny kontrollstruktur som inte kunde tas upp i kapitlet om kontrollstrukturer *(Progr1)* därför att den förutsätter array-begreppet eller liknande sammansatta datatyper, som vi inte hade hunnit gå igenom då.

foreach-satsen är idealisk för att skriva ut sammansatta datatypers värden. Den gör samma sak som **for**-satsen, men har en lite annorlunda – ja t.o.m. lite enklare syntax, om man är förtrogen med arrays. I programmet **Array** (sid 199) ser satsen ut så här:

```
foreach (int element in no)
    Console.Write(element + "\t");
```

Översatt till svenska:

För varje **element** av arrayen **no**
Skriv ut elementet följt av en tabulator.

element – ett namn som är valt av oss – kallas för **foreach**-satsens *iterationsvariabel*. Den definieras till **int** och motsvarar **for**-satsens räknare. **element** pekar på värdet (innehållet) som står i arrayen. Iteration betyder upprepning och innebär här att satsens kropp upprepas: Programflödet fortskrider från element till element tills alla element är genomgångna. Det reserverade ordet **in** betyder *av* eller *element av*. **no** pekar på arrayen som ska loopas igenom. Därför: ” För varje **element** *av* arrayen **no**”.

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

foreach-satsens iterationsvariabel måste ha samma datatyp som arrayelementen eller en sådan datatyp som arrayelementens datatyp automatiskt kan konverteras till. I vårt exempel har vi **int**. Det är t.o.m. möjligt att ha egendefinierade datatyper dvs klasser.

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

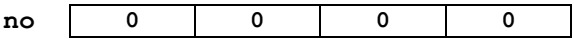
Hakparentesernas tre olika betydelser

Man kan ju undra varför **no[4]** inte är definierat – som vi hävdar ovan – fast talet **4** ”förekommer” i definitionssatsen **new int[4]**. Detta beror på att hakparenteserna **[]** i **no[4]** inte har samma betydelse som i **new int[4]**. Den korrekta tolkningen av **[]** beror på sammanhanget. Man kan också säga att **[]** är symbolen för tre olika operatorer som överlagrar varandra dvs betyder olika i olika sammanhang:

1. **[] som storleksoperator** omsluter i definitioner med **new** *antalet* element i arrayen specificerar därmed arrayens *storlek*. T.ex. innebär koden

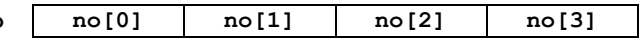
   ```
   new int[4]
   ```

   i programmet **Array** att **new** skapar en array av **int** med **4** element dvs att **4** minnesceller reserveras för lagring av **int**-värden. Det gemensamma för dessa element är att de lagras en efter den andra vid adressen eller referensen **no**:

   Här är frågan om ”Hur många element?”. I matematiken kallas detta *kardinaltal*.

2. **[] som indexeringsoperator** omslutar *indexet* till varje element av en array. Här handlar det om ett elements *position* i arrayen. Man anger index inom hakparenteser för att referera till elementet när man vill hämta eller tilldela det ett värde. Indexregeln (sid 198) tillämpas enligt vilken indexeringen börjar med **0**. Därför är **no[4]** i arrayen ovan inte definierat:

   Här är frågan om ”Vilket element?”. I matematiken kallas detta *ordinaltal*.

3. **[] som en del av datatypen** ”referens till array” omsluter ingenting utan är tom och skrivs direkt efter en datatyp för att definiera en ny referenstyp. T.ex. innebär satsen

   ```
   int[] no;
   ```

   i programmet **Array** att en minnescell allokeras (en referensvariabel med namnet **no** definieras) för lagring av en adress till en **int**-array. Vi kan i fortsättningen använda namnet **no** för att komma åt arrayen vid denna adress. I satsen ovan har referensen **no** inte initierats. Det sker inte heller automatiskt,

för **no** är en lokal variabel i **Main()**. Det sker först med tilldelningen **no = new int[4];** som initierar referensen explicit.

Default-initiering av en array

Det anmärkningsvärda är nu att det som gäller för referensen **no** – att den är oinitierad när den skapas – inte gäller för själva arrayen. Referensen **no** är oinitierad och måste initieras explicit eftersom den är en lokal variabel i **Main()**. Men trots att även arrayen är lokal i **Main()** initieras den till datamedlemmars default-värden, vilket är ett tecken på att array även i detta avseende behandlas som objekt. Programmet **Array** skriver ut arrayelementens värden en gång *innan* och en andra gång *efter* att de har fått värdena 64, 86, 34 och -6. Utskriften på förra sidan visar för arrayens alla element initialvärdet **0** som är den föreskrivna default-initieringen för variabler av typ **int** vilket även gäller för element i en **int**-array. Generellt gäller:

> Alla element i en array initieras automatiskt till default-värden (precis som datamedlemmar i ett objekt) även om arrayen skapas lokalt i en metod.

9.2 Arrayens initieringslista

Man kan effektivisera hanteringen av arrays inte bara med **foreach**-satser utan även genom att använda sig av en s.k. *initieringslista* som slår ihop definitionen med initieringen – en kortform som ersätter koden **new**r:

```
// ArrayInit.cs
// Initieringslista: Kortform för definition och initiering
// av en array i en och samma sats utan new
// Utskrift av arrayens element med foreach-satsen
using System;

class ArrayInit
{
  static void Main()
  {
    int[] no = { 64, 86, 34, -6 };     // Initieringslista:
                                       // Deklaration OCH ini-
                                       // tiering av en array
    //int[] no = new int[4] { 64, 86, 34, -6 };  // Samma sak

    Console.Write("\nVärdena från arrayen skrivs ut med" +
                                         " referensen:\n\n\t");
    foreach (int element in no)
       Console.Write(element + "\t");
    int[] copy = no;                  // Ny referens till no
                                       // samma array
    Console.Write("\n\n\tArrayens värden skrivs ut" +
                    " med den nya referensen copy:\n\n\t");
    foreach (int element in no)
       Console.Write(element + "\t");
    Console.WriteLine("\n\n\tEndast referensen kopieras,
                                           inte arrayen.\n");
   }
}
```

En körning visar att värdena i initieringslistan som först tillelas arrayen **no** verkligen kopierats över till arrayen **copy**, för det är de som skrivs ut:

```
      Arrayens värden skrivs ut med referensen no:

      64      86      34      -6

      Arrayens värden skrivs ut med den nya referensen copy:

      64      86      34      -6

      Endast referensen kopieras, inte arrayen.
```

Både definitionssatsen och initieringssatserna i programet **Array** (sid 199) – det är de 5 första satserna i **Main()** – kan slås ihop till en enda sats:

```
int[] no = { 64, 86, 34, -6 };
```

Satsen ovan är bara en förkortning på:

```
int[] no = new int[4] { 64, 86, 34, -6 };
```

Dvs initieringslistan kan skrivas efter **new int[4]** som egentligen skapar eller definierar arrayen. Men **new int[4]** får utelämnas. Detta visar att den förkortade versionen gör två saker: Först, fram till tilldelningstecknet definieras referensen **no** (*utan* någon uppgift om arrayens storlek). Sedan, från och med tilldelningstecknet tilldelas arrayen **no**:s element fyra värden som står i en kommaseparerad lista grupperad inom klamrarna **{ }** som kallas arrayens *initieringslista*. Kortformen gör precis samma sak som satsen med **new**. Kompilatorn får informationen om arrayens storlek genom att i initieringslistan räkna antalet element inom klamrarna **{ }**. Det är inte ens tillåtet att explicit ange det korrekta antalet element inom hakparenteserna **[ ]**. Det blir kompileringsfel om man gör det, därför att **no** endast är en referens till en array, inte arrayen själv. Observera även att man inte får använda initieringslistan separat utan endast i samma sats som definitionen.

Valet av variabelnamnet **copy** kan vara missledande i följande sats av programmet **ArrayInit** om man inte beaktar skillnaden mellan referens och array:

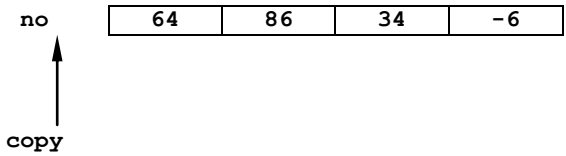
```
int[] copy = no;
```

copy blir nämligen en kopia av referensen **no** i satsen ovan, inte av arrayen – en ny referens som kommer att peka på samma array som den gamla referensen **no** pekar på. Det skapas ingen ny array eftersom det varken finns någon **new** eller någon initieringslista som skulle ersätta **new**. Anledningen till detta är – som vi konstaterat tidigare – följande viktigt faktum:

> En array i C# är alltid ett objekt som behöver en referens.

För att skapa ett objekt måste en **new**-sats skrivas. En referens definieras utan **new**.

Minnesmässigt lagras arrayen på *en och samma* adress som från programmet kan nås med referenserna **no** eller **copy**:



9.3 Texthantering med array av `char`

I början av detta kapitel motiverade vi behandlingen av arrayer med att kunna lagra och bearbeta stora datamängder. Ett exempel på det är *texthantering* där stora textmängder snarare är regel än undantag. Textbehandling är ett klassiskt område för datortillämpning just pga datorns förmåga att effektivt och snabbt kunna hantera stora datamängder. Vanligtvis kan text framställas med datatypen **`String`** som är en klass. En direkt ändring av en text som skapats med datatypen **`String`** är inte möjlig. Har man däremot en array av datatypen **`char`** har man obegränsade möjligheter till manipulation i och med man kan gå ned till elementnivå. Följande program demonstrerar ett enkelt fall av texthantering med en array av **`char`**:

```
// ArrayChar.cs
// Deklaration och initiering av en array av char
// I C# är en array av char inte samma sak som String
// char-array tillåter ändring av innehållet, men inte String
using System;

class ArrayChar
{
  static void Main()
  {
    String str  = "Russell";
    char[] text = str.ToCharArray();      // Omvandling av en
                                  // sträng till en array av char
    // char[] tex = "Russell";     // Kompileringsfel: sträng
                                   // är INTE array av char
    Console.WriteLine("\n\tArrayen har längden\t" +
                                              " text.Length);
    Console.Write("\n\tInnehållet är\t\t");
    foreach (char element in text)
       Console.Write(element);      // Elementvis utskrift
    Console.WriteLine();            // FÖRE ändringen
    text[0] = 'r';                  // Ändring av char-array
    text[1] = 'i';
    Console.Write("\tsom görs om till\t");
    foreach (char element in text)
       Console.Write(element);      // EFTER ändringen
    Console.WriteLine('\n');
  }
}
```

Ett körresultat av programmet ovan ger följande utskrift:

```
        Arrayen har längden     5

        Innehållet är           Russell
        som görs om till        rissell
```

Utskriften visar arrayens innehåll före och efter ändringen av två bokstäver i strängen **Russell**. För att kunna göra denna ändring var det nödvändigt att omvandla **String**-objektet **Russell** till en array av **char** med **String**-metoden **ToCharArray()**. Resultatet används för att initiera den med **char[] text** definierade referensen till en **char**-array. Närmare bestämt är satsen **char[] text = str.ToCharArray();** som gör detta, en kortform för satserna:

```
char[] text;                // Referens till en char-array
text = new char[5];         // Deklaration av char-array
text = str.ToCharArray(); // Initiering av char-array
```

Här måste arrayens storlek anges medan den i kortformen inte får anges. Programmet tar själv reda på storleken från strängen **Russell**. Vi kan sedan få tag i storleken med **String**-egenskapen **Length** som enligt utskriften ovan ger **5**. En gång omvandlad till **char**-array kan vi gå in på elementnivå och ändra strängens innehåll med hjälp av arrayens index.

Slumplösenord

En mer intressant tillämpning av textbehandling med hjälp av **char**-array visas i följande program som genererar ett slumplösenord:

```
// RandPasswd.cs
// Skriver ut ETT slumpvis genererat lösenord med policyn:
// 8 tecken = 3 små bokstäver: ASCII-intervall (97, 122) +
//            2 siffror                        (48,  57) +
//            3 stora bokstäver                (65,  90)
using System;

class RandPasswd
{
  static void Main()
  {
    Random r = new Random();
    char[] password = new char[8];

    for (int i=0; i < 3; i++)
        password[i] = (char) r.Next(97, (122+1)); // 3 små
                                                  // bokstäver
    for (int i=3; i < 5; i++)
        password[i] = (char) r.Next(48, (57+1)); // 2 siffror

    for (int i=5; i < 8; i++)
        password[i] = (char) r.Next(65, (90+1)); // 3 stora
                                                  // bokstäver
      Console.Write("\n\t");
      Console.WriteLine(password);
      Console.WriteLine();
  }
}
```

Ett antal körningar av programmet **RandPasswd** ger följande slumplösenord:

```
gpf49ZLC
```

```
lrn13VSZ
```

```
ztk27CRC
```

```
rmq53QMY
```

```
rxg53JIM
```

Övningar till kapitel 9

9.1 Skriv ett program som läser in 10 heltal från konsolen, lagrar dem i en array och skriver ut dem i omvänd ordning.

9.2 Skriv ett program som slumpar fram 1000 heltal mellan 60 och 140 (tänkbara hastigheter på en motorväg), lagrar dem i en array kallad **`hastighet`**, beräknar och skriver ut deras medelvärde med förklarande text.

9.3 Skriv ett program som läser in en sträng, lagrar den i en array av **`char`** och skriver ut den baklänges. Använd tekniken i programmet **`EncryptChar-Test`** för att omvandla den inlästa strängen i en array av **`char`**.

9.4 Skriv ett program som läser in text i gemener, lagrar den i en array av **`char`** och skriver ut den framhävd i versaler och med mellanslag mellan varje tecken.

9.5 Skriv ett program som frågar efter användarens för- *och* efternamn, hälsar sedan användaren i en utskrift med fullständiga namnet, förnamnets längd samt efternamnets första och sista bokstav. Lös uppgiften generellt utan att använda information om något speciellt för- och efternamn.

9.6 Skriv ett program där **`Main()`** läser in en persons fullständiga namn och hälsar tillbaka med namnets initialer. Dessa ska bestämmas och skrivas ut i en annan metod – med huvudet **`static void Initials(char[] name)`** – som anropas i **`Main()`**.

9.7 ***Master Mind (projekt)*** är ett litet spel som låter användaren gissa ett slumpmässigt genererat fyrsiffrigt heltal genom att leda spelaren med en inbyggd hjälpprocedur vars regler är beskrivna nedan. Även här gäller det att försöka hitta egna lösningar. Följande ska anses som ett förslag till lösning:

Börja med att behandla *fyr*siffriga heltal som en serie av *fyra en*siffriga tal dvs som en array av heltal med fyra element.

Skriv först en metod med huvudet **`void Create(int[] secretNo)`** som ska generera det hemliga fyrsiffriga talet och lagra det i en **`int`**-array, säg **`secretNo`**, med 4 element. Varje element i arrayen **`secretNo`** kan genereras som ett slumptal mellan 0 och 9. Dessutom ska metoden **`Create()`** kontrollera spelets regel enligt vilken alla fyra siffror måste vara olika.

Skriv sedan en metod med huvudet **`void Help(int[] guessedNo, int[] secretNo)`** som ska bearbeta spelarens gissning enligt följande regler:

För varje rätt siffra på rätt plats från vänster till höger skrivs ut ett **R**
För varje rätt siffra på fel plats från vänster till höger skrivs ut ett **S**
För varje fel siffra från vänster till höger skrivs ut ett mellanslag **?**

Om t.ex. det hemliga talet är 4693 och spelaren gissar 7498, så erhålls hjälpen:

? S R ?

När hjälpen skriver ut **RRRR** har spelaren lyckats och programmet avslutas med att skriva ut ett lämpligt meddelande. Skriv ett program som tillåter flera spelomgångar.

Appendix

Vad är objektorienterad programmering?

En ny syn på programmering

En given definition på programmering är problemlösning med hjälp av datorn. Om man då beskriver problemets lösning i form av en *algoritm* kan man säga *Program = algoritm + data*. Denna definition ställdes upp av Niklaus Wirth på 60-talet och återspeglar den procedurala synen på programmering. Fokuset ligger på *algoritmen* dvs att inte bara hitta utan även *beskriva* tillvägagångssättet (proceduren) för att lösa ett problem. Sedan återstår bara att koda denna beskrivning. En annan definition som kom upp på 80-talet och återspeglar den objektorienterade synen på programmering är:

Program = Modell av verkligheten

Om man i formeln *Program = algoritm + data* lägger om betoningen på *data* istället för på algoritmen och inte längre betraktar data som ett slags bihang till algoritmen utan som *objekt* kommer man till *objektorienterad programmering*. Denna nya programmeringsfilosofi kommer att genomsyra hela boken, eftersom C# är ett objektorienterat språk.

Paradigmskifte

Det som i programmeringshistorien gjorde att man behövde objektorienterad programmering var den växande komplexiteten hos program under 70-talet. Programmens storlek var avgörande för den växande komplexiteten. Man insåg att det inte längre räckte till att skriva och testa program som fungerade just då. Det var nödvändigt att med rimliga kostnader kunna även *underhålla* stora program, *förnya* och *vidareutveckla* dem så att de fungerade även i flera år och att de framför allt kunde anpassas till nyuppkomna situationer utan oöverkomliga svårigheter. Det i sin tur krävde att man redan i designstadiet behövde ett annorlunda upplägg. Fokuset förskjöts från problemlösning till modellering av verkligheten. Objektorienterad design kom in i bilden. Allt detta var endast med procedural programmering inte längre möjligt. Ett s.k. *paradigmskifte* hade blivit nödvändigt, dvs en ändring av helhetssynen på programmering.

Objektorienterad programmering syftar åt att efterlikna verkligheten. Man vill avbilda den reala världen – åtminstone den del som tillåter datorisering – och konstruera en modell av den i sina datorprogram för att kunna simulera verkligheten genom att testa modellen. För att undvika filosofiska diskussioner kan vi anta att den reala världen består kort sagt av *objekt*. Världen kring oss är full med sådana objekt: Människor, byggnader, bilar, tåg, flygplan, träd, möbler, böcker, butiker, skolor, bibliotek, kontor, anställda, kunder, varor, fakturor, order, bokningar, kurser osv. Objekten kan vara verkliga eller virtuella. Ett datorprogram försöker att beskriva dessa objekt. Låt oss precisera detta:

Objekt, klass och metod

Ett *objekt* har vissa egenskaper. Generellt kan man säga att ett objekt är summan av alla sina egenskaper. Ett annat ord för egenskap är *attribut*. Ett objekt består av alla sina attribut. Attributen tillhör objektet. T.ex. har objektet bil som attribut fabrikat, modell, färg, årsmodell, antal körda mil, antal hästkrafter, maximala hastigheten, antal och storlek på cylindrar i motorn osv. Alla dessa data ger svar på frågan ”Vad är det för bil?”. Men bilden vore ofullständig om vi nöjde oss med dessa intressanta, men statiska data. Vi vill också veta vad man kan *göra* med bilen. Ett objekt kan i regel även utföra vissa aktioner eller operationer. I den objektorienterade programmeringens terminologi kallas de för *metoder*. Typiska metoder för en bil är t.ex. att köra fram, att backa, att accelerera, att bromsa, att parkera, att byta olja osv. Den fullständiga definitionen på en bil som objekt vore alltså att ange *både* dess attribut *och* metoder. Bilfabrikanten måste förse bilen med alla dessa färdigheter för att kunna sälja den. Därför går man i bilfabriken efter en plan när man tillverkar bilen. I den objektorienterade programmeringens terminologi kallas denna plan för bilens *klass*. När vi skriver ett program måste vi först formulera *klassen Bil* för att sedan kunna skapa objekt av den. Klassen skrivs bara en gång, medan objekt kan skapas enligt klassens beskrivning i obegränsat antal. I klassen måste vi ta upp alla attribut och metoder som är relevanta eller av någon anledning önskvärda för en bil. Den praktiska användningen avgör från fall till fall vad som är relevant eller önskvärt.

Vad är skillnaden mellan *objekt* och *klass*? Om vi byter ut bilar mot pepparkakor kan man säga att pepparkaks*formen* är klassen och själva pepparkakorna är objekten. Klassen är alltså en slags mall, en förskrift för produktion av objekt: En enda pepparkaksform kan producera tusentals pepparkaksgubbar. Gubbarna kan skiljas från varandra i vissa detaljer, t.ex. materialet, smaken osv. Man kan t.o.m. måla dem i olika färger eller modifiera på annat sätt efteråt. De förblir pepparkaksgubbar av den ursprungliga formen. I formen ingår det som är gemensamt hos alla pepparkaksgubbar. Man har, när man byggde formen, bortsett från oväsentliga skillnader och tagit hänsyn endast till det väsentliga, det gemensamma hos alla pepparkakor.

Att *bortse* från skillnader och att bibehålla det gemensamma hos olika verkliga objekt, är en *abstraktion* (*abstrahera* betyder på latin: att ta bort, att dra av). Man tar bort allt som skiljer saker och ting av samma kategori eller typ och kommer på det viset till själva kategorin. Abstraktion leder till *begrepp*sbildning, till *klass*ificering eller *kategori*sering av den reala världen. Ett växande barn går igenom samma abstraktionsprocess, ser först sina föräldrar (objekt), abstraherar sedan via erfarenhet så småningom till begreppet *människa* (klassen) och inser att sina föräldrar är två konkreta exemplar av den abstrakta klassen människa. Så gör barnet med alla saker och ting omkring sig och lär sig vuxenvärldens begreppsapparat. Det abstrakta begreppet *penna* (klassen) t.ex. bildas efter att man sett hundratals verkliga pennor (objekt). Objektorienterad programmering återspeglar denna naturliga tankeprocess från det konkreta till det abstrakta, från objekt till klass.

Metoder

En *metod* är en funktionalitet som definieras i en klass. Den talar om vad ett objekt av denna klass kan *göra*. Det finns två steg i hantering av metoder: Först definierar man dem dvs skapar man deras kod i en klass. Sedan *anropar* dvs aktiverar man dem i ett objekt av denna klass. Ofta är det första steget redan genomfört av andra, så vi behöver bara anropa en redan *fördefinierad* metod. I klassen *Bil* t.ex. är metoderna att köra fram, att backa, att accelerera, att bromsa osv. definierade i huvuden på bilkonstruktörerna och i deras konstruktionsritningar och dokumentationer. Sedan har man tillverkat massor med objekt av klassen Bil i fabriken och byggt in dessa metoder i alla bilar. Vi behöver bara anropa dem i den bil vi kör. Den bil vi kör är ett specifikt objekt av klassen Bil. Låt oss kalla det för **`minVolvo`**. Objektet **`minVolvo`** har ett antal attribut som t.ex. fabrikat, modell, färg, årsmodell osv., men också ett antal metoder, bl.a. metoden **`Kör()`**. Parenteserna i metodens namn brukar man skriva för att karakterisera **`Kör()`** som en *metod* och skilja den från klassens attribut. I C# skriver man ett anrop av metoden **`Kör()`** så här:

```
minVolvo.Kör();
```

Observera att *före* punkten står ett objekt, inte klassen. Det är ju den specifika bil som jag använder just nu som ska köras. Först *efter* punkten står själva anropet av metoden **`kör()`**. Det här sättet att skriva kallas *punktnotation*. Metoder måste alltid anropas med punktnotation, vilket har sin grund i att de endast är deklarerade i klasser, så att de endast existerar i objekt av en klass. Till skillnad från fristående *funktioner* kan metoder varken definieras utanför klasser eller anropas utanför objekt. I C# finns endast metoder, inga funktioner. Om vi bortser från bilexemplet kan det i andra sammanhang även förekomma en klass (istället för objekt) före punkten i anropet av en metod. I så fall är metoden definierad i klassen på ett speciellt sätt nämligen som en *statisk* metod, vilket tas upp senare när vi behandlar metoder i detalj.

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

```
minVolvo.Kör(40);
```

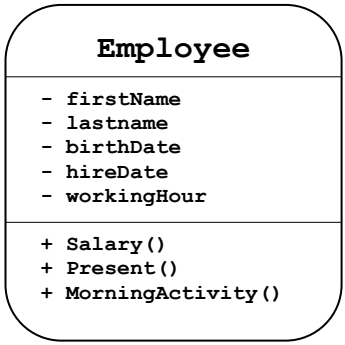
Det kan t.ex. betyda: Kör bilen med hastigheten **`40`** km/h. Värdet **`40`** kallas då en *parameter* som skickas till metoden när den anropas. I så fall måste även metoden **`Kör()`** vara definierad så att den har beredskapen att ta emot denna parameter. Så det kan inte vara samma metod som anropades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

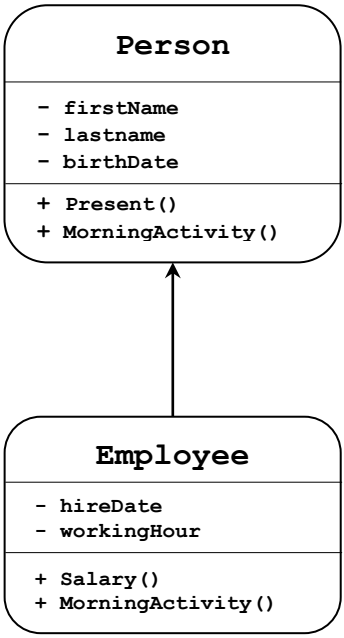
Klassdiagram

Låt oss ta som exempel en algoritm som beskriver hur man går upp, duschar, tar på sig kläderna och åker till jobbet. Låt oss kalla algoritmen för *Morgonsyssla*, vilket är ett typiskt fall av problemlösning: Det löser problemet *hur* man tar sig till jobbet.

Tillvägagångssättet och framför allt hur vi beskriver det, är föremål för algoritmer. Men *vem* eller *vilka* gör det, dvs vilka *objekt* som är involverade i algoritmen och hur man beskriver dessa objekt, är en annan aspekt på saken. Objektorienterad programmering prioriterar objektaspekten framför algoritmaspekten. Den primära frågan är inte längre vad man *gör* utan vem man *är* dvs *hur kan personen beskrivas?* Hur man gör för att ta sig till jobbet kommer att ingå som en del i denna beskrivning. Algoritmen Morgonsyssla blir en metod i *objektet* Person. Det är objektet som utför metodens instruktioner för att ta sig till jobbet.

Personen kan t.ex. vara en anställd vilket förresten skulle förklara varför han tar sig till jobbet. I så fall är personen ett objekt av kategorin eller klassen *Employee*. Därför definieras en klass som beskriver alla anställda. Personen i fråga görs till ett objekt, ett exemplar av denna klass. På så sätt kan koden återanvändas även för andra anställda. Återanvändning av kod gör utvecklingsarbetet av programvara effektivare och är en av den objektorienterade synens fördelar. I klassen *Employee* ingår all typ av information som är relevant för en anställd, det vi kallar för attribut, t.ex. för- och efternamn, födelse- och anställningsdatum, arbetstid osv. Dessutom tar vi upp allt som en anställd kan göra, det vi kallar för metoder, t.ex. att få lön, att presentera sig eller också att ta sig till jobbet. På så sätt blir algoritmen Morgonsyssla i den objektorienterade programmeringens terminologi en metod i klassen *Employee*. Ett verktyg speciellt för objektorienterade modelleringar är UML *(Unified Modeling Language)*. Enligt det här modelleringsspråket skulle klassen *Employee* beskrivas med diagrammet till vänster som kallas för *klassdiagram*. Där står tecknet **–** för attribut och **+** för metoder. Andra beteckningar för attribut är *datamedlem* eller *egenskap*. Dessa termer är synonymer. En klass består av datamedlemmar och metoder. Klassen **Employee** t.ex. har fem datamedlemmar och tre metoder.





Klassens konstruktor

Eftersom klassens datamedlemmar i regel är inkapslade (privata) och inte åtkomliga utifrån klassen – detta gör man bl.a. ur datasäkerhetssynpunkt – måste programmeraren använda sig av ett verktyg för att på ett kodat sätt ändå kunna komma åt dem, läsa och ändra dem osv. Detta verktyg kallas klassens *konstruktor* och är en speciell metod vars namn är identiskt med klassens

namn. Den initierar automatiskt klassens privata datamedlemmar när ett objekt skapas. För enkelhetens skull har vi inte tagit upp den i klassdiagrammet ovan bland klassens metoder. Konstruktorn har ju endast programmeringsteknisk karaktär och behandlas i *Programmering 2*.

Arv

I den reala världen som vi vill efterlikna, finns inga isolerade objekt. Alla objekt är mer eller mindre relaterade till andra objekt. En klok modellering måste dra nytta av de befintliga relationer mellan objekt för att effektivisera och optimera utvecklingsarbetet. En sådan relation är arvrelationen.

Man kan alltid etablera en arvrelation mellan två begrepp om de står i en ”*är*”-relation till varandra. I exemplet ovan kan vi konstatera ett en anställd *är* en person. Därför kan klassen **`Employee`** ärva klassen **`Person`**, närmare bestämt ärver klassen **`Employee`** klassen **`Person`**:s alla datamedlemmar och metoder. Klassen **`Person`** kallas *bas-* eller *superklass*. Klassen **`Employee`** kallas *härledd* eller *subklass*. Subklassen ärver superklassens alla datamedlemmar och metoder, vilket i praktiken innebär att klassen **`Employee`** tar över all kod som redan finns i klassen **`Person`** och lägger till ny kod som närmare specificerar en anställd. På så sätt slipper man skriva om kod som redan finns. T.ex. har en person ett för- och efternamn samt ett födelsedatum. Vid modellering av en anställd ärvs dessa attribut, och man lägger till de nya attributen hireDate och workingHour som är speciella för en anställd. Klassdiagrammet ovan (till vänster) visar modellen där arvrelationen ritats med en pil riktad mot superklassen. Följer man pilens riktning underifrån kan man avläsa att det är klassen **`Employee`** som ärver klassen **`Person`**.

Objektorienterad programmering bygger på tre hörnstenar:

- Inkapsling
- Arv
- Polymorfism

De två sista har vi försökt att introducera här utan att behöva skriva en enda rad kod. För den första behöver vi lite mer detaljerade kunskaper om programmering.

Några konventioner

När vi ritade UML-diagrammen ovan och skrev text i dem tillämpade vi några konventioner som man i regel brukar använda inom programmering.:

1. Klasser och metoder inleds med versaler

För att bättre kunna känna igen klasser och metoder i objektorienterad kod, inleds de med versaler. Så därför skriver vi klasserna **`Employee`** och **`Person`** samt metoderna **`Salary()`**, **`MorningActivity()`**, så här. Följer man konsekvent *alla* konventioner kan man lätt avgöra att ett ord i koden som har en stor begynnelsebokstav, är en klass eller en metod.

2. Datamedlemmar och objekt inleds med gemener

Till skillnad från klasser och metoder inleds alla datamedlemmar och objekt med gemener, t.ex. **`firstName`**, **`surname`**, **`workingHours`**, … . Även den här konventionen höjer kodens läslighet när den följs i kombination med punkt 1.

3. Metoder skrivs med parentes

Till skillnad från en datamedlem som kan ha ett värde, beskriver en metod en funktionalitet som kan anropas från en annan metod vilket medför att man vid anropet kan skicka vissa parametrar till den anropade metoden. Dessa parametrar skrivs inom en parentes direkt efter metodens namn. På så sätt är parentesen kännetecknet för en metod. Även om en viss metod inte har någon parameter alls som i våra exempel med **`Salary()`**, **`MorningActivity()`**, … brukar man ändå skriva den tomma parentesen med. Detta är en *regel* för att skriva kod, men har blivit en *konvention* som man även följer utanför programkod.

Fullständiga lösningar till alla övningar (Facit) *

Kapitel 1 C# programmeringens miljö, sid 30:

Ovn_1_2.cs
Modifiera programmet MessageBox (sid 29) så att meddelanderutan får rubriken ”Övning 1.1”. Hur man skriver ut en MessageBox med egenvald rubrik har vi lärt oss i programmet Interaction (sid 26).

Form1.cs
```
using System;
using System.Windows.Forms;

namespace Messagebox
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }

      private void Form1_Load(object sender, EventArgs e)
      {
         MessageBox.Show("Hälsningar från Windows MessageBox " +
                         "som visas när formen laddas.", "Övning 3.9");
      }
   }
}
```

****
Ovn_1_2, Ovn_1_3 *och* **Ovn_1_4**

Dessa övningar innehåller ingen kod utan löses med visuell programmering. Använd och modifiera anvisningarna på sid 23.

****

Kapitel 3 Att komma igång med C#, sid 53:

Ovn_3_1.cs
Mata in koden till programmet First (sid 42), kompilera och kör.
a) Skriv om programmet First genom att ta bort using-direktivet och modifiera istället utskriftssatsen så att den kan kompileras och ger samma resultat som programmet First.
b) Undersök skillnaderna mellan apostrof,citationstecken & accent.

a)
```
class Ovn_3_1
```

---

* För att uppmuntra till egna lösningar ges inga lösningsförslag till projektuppgifterna *Löpande texten, Pyramiden, Automaten*, *Kalkylatorn, Mater Mind* eller uppgifter som är relaterade till ett projekt. Istället finns det i projektenas lydelse alltid en mer eller mindre utförlig ledning (algoritm) till lösningen.

```
{
   static void Main()
   {
      System.Console.WriteLine("\n\tMitt första C# program!\n");
   }
}
```

b)

Apostrof är ' (Tillsammans med ***** *tangenten)*
Citationstecken är " (Tillsammans med **2:***ans tangent)*
Två typer av accent är ´ och ` (Till höger om **+** *tangenten)*
OBS! Beakta skillnaden mellan apostrof & accent. De förväxlas lätt.

****

Ovn_3_2.cs
Sätt in följande kod i ett C# program för att testa vad den ger för utskrift:

```
Console.Write  ("**\n");
Console.WriteLine("***");
Console.WriteLine("****");
Console.Write  ("*****\n");
Console.WriteLine("******");
Console.WriteLine("*****");
Console.Write  ("****\n");
Console.WriteLine("***");
Console.WriteLine("**\n");
```

a) Ersätt alla anrop av Console.Write() med Console.WriteLine() och ändra lite i koden utan att utskriften ändras.

b) Lägg till lite kod i varje sats så att hela den utskrivna figuren hamnar lite längre bort från konsolfönstrets vänstra och övre rand.

a)

```
using System;
class Ovn_3_2a
{
   static void Main()
   {
      Console.WriteLine("**");
      Console.WriteLine("***");
      Console.WriteLine("****");
      Console.WriteLine("*****");
      Console.WriteLine("******");
      Console.WriteLine("*****");
      Console.WriteLine("****");
      Console.WriteLine("***");
      Console.WriteLine("**\n");
   }
}
```

----

b)

```
using System;
class Ovn_3_2b
{
   static void Main()
   {
      Console.WriteLine("\n\t**");
```

```
      Console.WriteLine("\t***");
      Console.WriteLine("\t****");
      Console.WriteLine("\t*****");
      Console.WriteLine("\t******");
      Console.WriteLine("\t*****");
      Console.WriteLine("\t****");
      Console.WriteLine("\t***");
      Console.WriteLine("\t**\n");
   }
}
```

*****

Ovn_3_3.cs

Skriv ett program och testa vilken utskrift följande satser ger:

```
            Console.Write    ("Jag");
            Console.Write    ("heter");
            Console.WriteLine("K.\n Vad heter du?\n");
```

Lägg till resp. ta bort mellanslag, radbyte och tabulator för att få en snygg utskrift utan att slå ihop de tre satserna till en.

```
using System;
class Ovn_3_3
{
   static void Main()
   {
      Console.Write("\n\tJag ");
      Console.Write("heter ");
      Console.WriteLine("K.\n\n\tVad heter du?\n");
   }
}
```

*****

Ovn_3_4.cs

Skriv ett program som först skickar följande kod till WriteLine():

```
                          "Resultatet är " +  8 + 3
```

och sedan:

```
                          "Resultatet är " + (8 + 3)
```

Förklara skillnaden i de två utskrifterna. Hur måste + tolkas?

```
using System;
class Ovn_3_4
{
   static void Main()
   {
      Console.WriteLine("\n\t Resultatet är " +  8 + 3);
      Console.WriteLine("\n\t Resultatet är " + (8 + 3) + '\n');
   }
}
```

Utskriften blir:

```
                          Resultatet är 83

                          Resultatet är 11
```

I den första utskriftssatsen tolkas båda + som konkateneringar. I den andra utskriftssatsen tolkas det första + tecknet som konkatenerings- och det andra + tecknet som addititonsoperator, eftersom parentesen exekveras först och i parentesen tolkas + som addition.

*****

Ovn_3_5.cs

Vilka utskrifter ger följande satser? Sätt in dem i ett program.

```
    Console.WriteLine("*\n**\n***\n****\n*****");
    Console.WriteLine("*****\n****\n***\n**\n*");
```

Skriv om koden så att du får samma utskrift med EN utskriftssats.

```
using System;
class Ovn_3_5
{
   static void Main()
   {
      Console.WriteLine("*\n**\n***\n****\n*****"   +
                        "\n*****\n****\n***\n**\n*" );
   }
}
```

****

Ovn_3_6.cs

Skriv in koden till programmet Concat (sid 51), kompilera och kör. Modifiera det till att skriva ut en oval byggd av stjärnor (*).

```
using System;
class Ovn_3_6
{
   static void Main()
   {
      Console.Write("\n       * * * * * * * * * *       " +
                    "\n     *                   *     " +
                    "\n    *                     *    " +
                    "\n   *                       *   " +
                    "\n   *                       *   " +
                    "\n   *                       *   " +
                    "\n    *                     *    " +
                    "\n     *                   *     " +
                    "\n       * * * * * * * * * *       " +
                    "\n\n");
   }
}
```

****

Ovn_3_7.cs

Skriv ett program som skriver ut en triangel byggd av stjärnor (*).

```
using System;
class Ovn_3_7
{
   static void Main()
   {
      Console.Write("\n                *          " +
                    "\n               * *         " +
                    "\n              *   *        " +
                    "\n             *     *       " +
                    "\n            *       *      " +
                    "\n           *         *     " +
                    "\n          *           *    " +
                    "\n         *             *   " +
                    "\n        *               *  " +
                    "\n       * * * * * * * * * * " +
                    "\n\n");
   }
```

```
}
*********************************************************************
```

Ovn_3_8.cs

Rita följande figur i konsolen med en enda utskriftssats. Se upp för skillnaden mellan slash / och backslash \. Använd två backslash \\ i koden – som en escapesekvens inbakad i den konkatenerade strängen – för att åstadkomma en backslash \ i utskriften. (Läs om escapesekvenser på sid 97).

```
using System;
class Ovn_3_8
{
   static void Main()
   {
      Console.Write("\n                 ______                " +
                    "\n               /_____/\\               " +
                    "\n              /_____ \\\\ \\\\            " +
                    "\n             /_____\\\\ \\\\\\ /          " +
                    "\n            /_____/ \\\\/ / /           " +
                    "\n           /_____/ /   \\\\//\\\\         " +
                    "\n           \\_____\\\\//\\\\   / /        " +
                    "\n            \\_____/ / /\\\\ /           " +
                    "\n             \\_____/ \\\\\\\\ \\\\        " +
                    "\n              \\_____\\\\ \\\\\\\\         " +
                    "\n               \\_____\\\\/              " +
                    "\n\n\n");
    }
}
*********************************************************************
```

Kapitel 4 Grundbegrepp i programmering, sid 83:

Ovn_4_1.cs

Satsen Console.WriteLine(a); ger kompileringsfel till skillnad från Console.WriteLine('a'); Sätt in båda i ett C# program och testa. Ger även Console.WriteLine(6); kompileringsfel? Testa vilka utskrifter följande satser ger:

```
      Console.WriteLine(6 + 6);
      Console.WriteLine('6' + '6');
      Console.WriteLine("6" + "6");
      Console.WriteLine(6.6 + 6.6);
      Console.WriteLine("6.6" + "6.6");
```

Förklara resultaten.

```
using System;
class Ovn_4_1
{
   static void Main()
   {
      Console.WriteLine(a);          // Ger kompileringsfel
      Console.WriteLine('a');        // Teckenkonstant:              a
      Console.WriteLine(6 + 6);      // Addition av 6 med 6:        12
      Console.WriteLine('6' + '6');  // Addition av 54 med 54:     108
      Console.WriteLine("6" + "6");  // Konkatenering:              66
```

```
        Console.WriteLine(6.6 + 6.6);  // Addition av 6.6 med 6.6: 13,2
        Console.WriteLine("6.6" + "6.6"); // Konkatenering:      6.66.6
    }
}
```

Förklaringar:
Console.WriteLine(a); ger kompileringsfel därför att a är en odefinierad variabel.
Console.WriteLine('a'); ger a därför att 'a' är ett tecken: bokstaven a.
Console.WriteLine(6 + 6); ger 12 därför att 6 är tal och + adderar talen.
Console.WriteLine('6' + '6'); ger 108 därför att '6' är tecknet med ASCII-koden 54, dvs ASCII-koderna adderas: 54 + 54 = 108.
Console.WriteLine("6" + "6"); ger 66 därför att "6" är en sträng och + konkatenerar strängarna "6" och "6".
Console.WriteLine(6.6 + 6.6); ger 13,2 därför att 6.6 är decimaltal och + adderar talen.
Console.WriteLine("6.6" + "6.6"); ger 6.66.6 därför att "6.6" är en sträng och + konkatenerar strängarna "6.6" och "6.6".

****

Ovn_4_2.cs
*Komplettera programmet **Variable** (sid 60): Definiera ytterligare variabler, säg **diff, prod, div, mod,** tilldela till dem uttryck bildade med de andra räknesätten **-, *, /** och **%**. Skriv ut resultaten med meningsfulla utskrifter. Bibehåll ändringen av variabeln **no1**:s värde mellan de två utskrifterna.*

```
using System;
class Ovn_4_2
{
   static void Main()
   {
      int no1, no2, sum, diff,            // Deklaration av variabler
          prod, div, mod;
      no1 = 9;                            // Initiering av variabler
      no2 = 3;
      sum  = no1 + no2;                   // Addition
      diff = no1 - no2;                   // Subtraktion
      Console.WriteLine("\n\tAddition        definierad för int:\t" +
                        no1 + " + " + no2 + " ger " + sum +
                        "\n\tSubtraktion    definierad för int:\t" +
                        no1 + " - " + no2 + " ger " + diff);
      no1 = 11;                           // Variabelns värde ändras
      prod = no1 * no2;                   // Multiplikation
      div  = no1 / no2;                   // Heltalsdivision
      mod  = no1 % no2;                   // Modulo: resten vid div.
      Console.WriteLine("\n\tMultiplikation  definierad för int:\t" +
                        no1 + " * " + no2 + " ger " + prod +
                        "\n\tHeltalsDivision definierad för int:\t" +
                        no1 + " / " + no2 + " ger " + div +
                        "\n\tModulo          definierad för int:\t" +
                        no1 + " % " + no2 + " ger " + mod + '\n');
   }
}
```

****

Ovn_4_3.cs
Vidareutveckla din lösning till övn 4.2 genom att ersätta den hårdkodade initieringen av variablerna no1 och no2 med en initiering genom inläsning som t.ex. kan göras med en ReadLine()-sats samt

```
ledtext. Stryk ändringen av variabeln no1:s värde.
using System;
class Ovn_4_3
{
   static void Main()
   {
      int no1, no2, sum, diff, prod, div, mod;
      string no1AsText, no2AsText;
      Console.Write("\n\tMata in ett heltal:\t\t");   // Ledtext
      no1AsText = Console.ReadLine();                 // Inläsning
      no1 = int.Parse(no1AsText);                     // Omvandling
      Console.Write("\n\tMata in ett heltal till:\t"); // Ledtext
      no2AsText = Console.ReadLine();                  // Inläsning
      no2 = int.Parse(no2AsText);                     // Omvandling

      sum  = no1 + no2;                    // Addition
      diff = no1 - no2;                    // Subtraktion
      prod = no1 * no2;                    // Multiplikation
      div  = no1 / no2;                    // Heltalsdivision
      mod  = no1 % no2;                    // Modulo: resten vid div.

      Console.WriteLine("\n\tAddition        definierad för int:\t" +
                        no1 + " + " + no2 + " ger " + sum +
                        "\n\tSubtraktion     definierad för int:\t" +
                        no1 + " - " + no2 + " ger " + diff +
                        "\n\tMultiplikation  definierad för int:\t" +
                        no1 + " * " + no2 + " ger " + prod +
                        "\n\tHeltalsDivision definierad för int:\t" +
                        no1 + " / " + no2 + " ger " + div +
                        "\n\tModulo          definierad för int:\t" +
                        no1 + " % " + no2 + " ger " + mod + '\n');
    }
}
*********************************************************************

Ovn_4_4.cs
Skriv ett program som läser in två heltal, multiplicerar dem
med varandra och skriver ut resultatet blandat med förklarande
text. Om du t.ex. matar in 3 till det första och 4 till det
andra heltalet, ska programmet skriva ut: 3 gånger 4 är 12.
Utveckla programmet vidare med ytterligare räkneoperationer,
kanske så småningom till en liten kalkylator.
using System;
class Ovn_4_4
{
   static void Main()
   {
      Console.Write("\n\tMata in ett heltal:\t\t");      // Ledtext
      int no1 = int.Parse(Console.ReadLine());          // Inläsning
      Console.Write("\n\tMata in ett heltal till:\t");
      int no2 = int.Parse(Console.ReadLine());

      Console.WriteLine("\n\n\t"                                    +
         no1 + " plus "   + no2 + " är " + (no1 + no2) + "\n\t" +
         no1 + " minus " + no2 + " är "  + (no1 - no2) + "\n\t" +
         no1 + " gånger " + no2 + " är " + (no1 * no2) + "\n\t" +
         no1 + " heltalsdividerad med "   +
                           no2 + " är " + (no1 / no2) + "\n\t" +
```

```
         no1 + " modulo " + no2 + " är " + (no1 % no2) + "\n\t");
   }
}
*********************************************************************
```

Ovn_4_5.cs

*Ersätt i programmet **DefInit** (sid 63) de två satser som definierar & initierar variablerna **no1, no2** med satsen **int no1 = 9, no2 = 2;***

```
using System;
class Ovn_4_5
{
   static void Main()
   {
      int no1 = 9, no2 = 2;              // Deklaration och initiering
                                         // i en och samma sats
      Console.WriteLine("\n\t" +
             "Summan av " + no1 + " och " +
                            no2 + " är " + (no1 + no2) + '\n');
   }
}
*********************************************************************
```

Ovn_4_6.cs

Modifiera progr. Overwrite (sid 68) så att variabeln x:s gamla värde skrivs ut, medan dess nya ökade värde visas senare.
Ersätt i din lösning satsen x = x + 1; med x++;
*Blir det samma resultat om du ersätter den med **x + 1;** istället?*

```
using System;
class Ovn_4_6
{
   static void Main()
   {
      Console.Write("\n\tMata in ett heltal:\t\t");
      int x = int.Parse(Console.ReadLine());
      Console.Write("\n\tTalet " + x);
      x = x + 1;
//    x++;                             // Går: Samma som x = x + 1;
//    x + 1;                           // Går ej! Ger kompilerings -
                                       // fel: Tilldelning saknas
      Console.WriteLine(" har ökats med 1 och är nu " + x + '\n');
    }
}
*********************************************************************
```

Ovn_4_7.cs

Skriv ett program som läser in tre heltal till timmar, minuter och sekunder. Beräkna och skriv ut sedan hur många sekunder det blir totalt. Gör utskriften användarvänlig.

```
using System;
class Ovn_4_7
{
   static void Main()
   {
      Console.Write("\n\t Ange antal timmar:\t\t");
      int hour = int.Parse(Console.ReadLine());
      Console.Write("\n\t Ange antal minuter:\t\t");
      int min = int.Parse(Console.ReadLine());
      Console.Write("\n\t Ange antal sekunder:\t\t");
```

```
      int sec = int.Parse(Console.ReadLine());

      int totalsek = 3600*hour + 60*min + sec;

      Console.WriteLine("\n\t" +
           hour + " timmar, " + min + " minuter och " +
           sec + " sekunder är " + totalsek + " sekunder totalt.\n");
   }
}
*********************************************************************
Ovn_4_8.cs
Varför ger följande program kompileringsfel? Åtgärda felet!
using System;
class Ovn_4_8
{
   static void Main()
   {
      int a, sum = 9;     // sum måste initieras innan den används
                          // I frågeställningen var sum inte initierad
      Console.Write("\n\tMata in ett heltal:\t");
      a = int.Parse(Console.ReadLine());
      sum += a;           // Samma som: sum = sum + a; Här används
                          // sum på höger sidan av tilldelningen
      Console.WriteLine("\n\tsum = "  + sum + '\n');
   }
}
*********************************************************************
```

Övningarna **4.9** och **4.10** är projektuppgifter, se fotnoten på sidan 219.

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

Kapitel 5 Enkla datatyper, sid 109:

```
Ovn_5_1.cs
Skriv ett program som läser in tre tecken och skriver ut dem
i omvänd ordning.
using System;
class Ovn_5_1
{
   static void Main()
   {
      char ch1, ch2, ch3;
      Console.Write("\nMata in tre tecken skilda med mellanslag:\t");
      string text = Console.ReadLine();
      ch1 = text[0];
      ch2 = text[2];
      ch3 = text[4];
      Console.WriteLine("\nOmvänd ordning:\t\t\t\t\t" +
                        ch3 + " " + ch2 + " " + ch1 + " " + "\n");
   }
}
*********************************************************************

Ovn_5_2.cs
Skriv ett program som läser in en gemen och skriver ut dess
```

versal och sedan läser in en versal och skriver ut dess gemen.

```
using System;
class Ovn_5_2
{
  static void Main()
  {
    Console.Write("\nMata in en gemen:\t");
    String str = Console.ReadLine();
    Console.WriteLine("\nDess versal är:\t\t" + str.ToUpper());

    Console.Write("\nMata in en versal:\t");
    str = Console.ReadLine();
    Console.WriteLine("\nDess gemen är:\t\t" + str.ToLower() + '\n');
   }
}
```

****

Ovn_5_3.cs

Experimentera med programmet Int2char (sid 95) för att ta reda på ASCII-koden till datorns ljudsignal. Dvs kör så länge tills du vid inmatning av ett heltal hör ett pip från datorn. Ändra datatypen till variabeln ***kod*** *från int till* ***char****. Åtgärda kompileringsfelet. Hör du fortfarande pipet när du matar in ASCII-koden för ljudsignal? Förklara.*

```
using System;
class Ovn_5_3
{
   static void Main()
   {
      Console.Write("\nMata in en siffra: ");
      char kod = Convert.ToChar(Console.ReadLine());

      Console.WriteLine("\nDet inmatade talet " + kod   +
                     " är ASCII-koden till tecknet " + kod + '\n');
   }
}
```

ASCII-koden till datorns ljudsignal är 7 som man får med programmet Int2char (sid 95). Programmet ovan genererar inte ljudsignalen utan läser in och skriver bara ut det inmatade tecknet 7. Bara om man läser in ASCII-koden som int och explicit omvandlar den till char får man ut ljudsignalen.

****

Ovn_5_4.cs

Kryptering av tecken: Skriv ett program som läser in ett tecken och förskjuter det i teckentabellen med ett visst antal steg som en slags krypteringsnyckel. Skriv ut både det inlästa och det förskjutna tecknet på ett användarvänligt sätt. Börja med att hårdkoda krypteringsnyckeln och fortsätt med att läsa in den.

```
using System;
class Ovn_5_4
{
   static void Main()
   {
      Console.Write("\nMata in ett tecken: ");
      char letter = Convert.ToChar(Console.ReadLine());
```

```
      Console.Write("\nMata in ett heltal (krypteringsnyckel): ");
      int steg = int.Parse(Console.ReadLine());

      String output = "\nDet inlästa tecknet " + letter;

      letter += (char) steg;                  // Kryptering av letter

      Console.WriteLine(output+ " har förskjutits med " +
                  steg  + " steg och är nu:     " + letter + '\n');
   }
}
*********************************************************************
```

Ovn_5_5.cs

Kryptering av ord: Skriv ett program som läser in fem tecken och skriver ut dem förskjutna med ett steg i ASCII-tabellen så att t.ex. inmatningen Kalle ger utskriften Lbmmf. Återställ sedan det krypterade ordet. Vidareutveckla programmet genom att utöka och läsa in antalet steg (krypteringsnyckeln).

```
using System;
class Ovn_5_5
{
  static void Main()
  {
    String encrypt = "";

    Console.Write("\nMata in 5 tecken utan mellanslag:        ");
    String word = Console.ReadLine();

    Console.Write("\nMata in ett heltal (krypteringsnyckel): ");
    int key = int.Parse(Console.ReadLine());

    encrypt += (char)(Convert.ToChar(word.Substring(0, 1)) + key);
    encrypt += (char)(Convert.ToChar(word.Substring(1, 1)) + key);
    encrypt += (char)(Convert.ToChar(word.Substring(2, 1)) + key);
    encrypt += (char)(Convert.ToChar(word.Substring(3, 1)) + key);
    encrypt += (char)(Convert.ToChar(word.Substring(4, 1)) + key);
    Console.WriteLine("\nDet krypterade ordet:\t\t\t" + encrypt);

    word = "";
    word += (char) (Convert.ToChar(encrypt.Substring(0, 1)) - key);
    word += (char) (Convert.ToChar(encrypt.Substring(1, 1)) - key);
    word += (char) (Convert.ToChar(encrypt.Substring(2, 1)) - key);
    word += (char) (Convert.ToChar(encrypt.Substring(3, 1)) - key);
    word += (char) (Convert.ToChar(encrypt.Substring(4, 1)) - key);
    Console.WriteLine("\nDet återställda ordet:\t\t\t" +word + '\n');
   }
}
*********************************************************************
```

Ovn_5_6.cs

Ersätt i följande program satsen letter++; med letter = letter + 1; Varför får du kompileringsfel? Försök att åtgärda felet utan att använda letter++;

```
using System;
```

```
class Ovn_5_6
{
   static void Main()
   {
      char letter = 'Y';
      String output =
             "Tecknet " + letter + " har koden " + (int) letter;

      letter = (char) (letter + 1);

      Console.WriteLine(output +
           "\nTecknet " + letter + " har koden " + (int) letter);
   }
}
```

letter = letter + 1; ger kompileringsfelet: "possible loss of precision" eftersom letter + 1 automatiskt konverteras till int, men vid tilldelningen (=) inte automatiskt kan omvandlas till vänstersidans char eftersom char har mindre plats (2 bytes) än int (4 bytes). Lösningen är explicit typkonvertering.

****

Övning **5.7** är en projektuppgift, se fotnoten på sidan 219.

****

Kapitel 6 Kontrollstrukturer, sid 151:

Ovn_6_1.cs

Skriv ett program som läser in två tal och skriver ut OK om de matats in i rätt ordning, dvs om det första är mindre än det andra. Vad händer om de är lika stora?

```
using System;
class Ovn_6_1
{
   static void Main()
   {
      Console.Write("\n\tMata in no1:\t");
      int no1 = int.Parse(Console.ReadLine());
      Console.Write("\n\tMata in no2:\t");
      int no2 = int.Parse(Console.ReadLine());
      if (no1 < no2)
         Console.WriteLine("\n\tOK. Talen matades in i rätt " +
                                                  "ordning.\n");
   }
}
```

Både om talen är lika stora eller det första är större än det andra, avslutas programmet utan utskrift eftersom alternativ (else) saknas.

****

Ovn_6_2.cs

*Modifiera din lösnig från övn 6.1 genom att läsa in två tecken istället för tal. Skriv ut **OK** om de matats in i ordning. Annars ska programmet skriva ut ett meddelande om att tecknen matades in i fel ordning*

```
using System;
class Ovn_6_2
{
   static void Main()
   {
      Console.Write("\n\tMata in char1:\t");
      String temp = Console.ReadLine();
      char char1 = Convert.ToChar(temp.Substring(0, 1));
      Console.Write("\n\tMata in char2:\t");
      temp = Console.ReadLine();
      char char2 = Convert.ToChar(temp.Substring(0, 1));

      if (char1 < char2)
         Console.WriteLine("\n\tOK. Du matade in i rätt ordning.\n");
      else
         Console.WriteLine("\n\tDu matade in i fel ordning.\n");
   }
}
*********************************************************************
```

Ovn_6_3.cs

Skriv ett program som läser in tre tal, hittar och skriver ut det största av dem. Vilken ändring i koden leder till det minsta talet?

```
using System;
class Ovn_6_3
{
   static void Main()
   {
      int max;

      Console.Write("\n\tMata in no1:\t");
      int no1 = int.Parse(Console.ReadLine());
      Console.Write("\n\tMata in no2:\t");
      int no2 = int.Parse(Console.ReadLine());
      Console.Write("\n\tMata in tal3:\t");
      int tal3 = int.Parse(Console.ReadLine());

      max = no1
      if (no2 > max)
         max = no2;
      if (no3 > max)
         max = no3;

      Console.WriteLine("\n\tDet största talet är " +  max + '\n');
   }
}
Genom att byta ut alla > mot < får man det minsta talet.
*********************************************************************
```

Ovn_6_4.cs

Skriv ett program som läser in begynnelsebokstaven till en veckodag, med en switch-sats bestämmer vilken veckodag det är och skriver ut den. Fixa problemet med tisdag/torsdag genom att nästla en if-else-sats i switch-satsen för att läsa in och bearbeta den andra bokstaven. Ta hand om felaktig inmatning.

```
using System;
```

```
class Ovn_6_4
{
  static void Main()
  {
    char bokstav1, bokstav2;
    String temp, veckodag;

    Console.Write("\n\tMata in en veckodags begynnelsebokstav:\t");
    temp = Console.ReadLine();
    bokstav1 = Convert.ToChar(temp.Substring(0, 1));

    switch (bokstav1)
    {
      case 'm':
         veckodag = "måndag.";
         break;
      case 't':
         Console.Write("\n\tMata in veckodagens andra bokstav:\t");
         temp = Console.ReadLine();
         bokstav2 = Convert.ToChar(temp.Substring(0, 1));
         if (bokstav2 == 'i')
            veckodag = "tisdag.";
         else
            veckodag = "torsdag.";
         break;
      case 'o':
         veckodag = "onsdag.";
         break;
      case 'f':
         veckodag = "fredag.";
         break;
         case 'l':
         veckodag = "lördag.";
         break;
      case 's':
         veckodag = "söndag.";
         break;
      default:
         veckodag = "?";
         break;
    }

    if (veckodag != "?")
       Console.WriteLine("\n\tDet är " + veckodag + '\n');
    else
       Console.WriteLine("\n\tDetta är ingen veckodag! + '\n'");
  }
}
*********************************************************************
```

Ovn_6_5.cs

Vidareutveckla övn 6.2 så att användaren får flera chanser att mata in två tecken i rätt ordning så länge han/hon matar in dem i fel ordning. Du kan göra det genom att bygga in inmatningen, bearbetningen och utmatningen i en do-loop.

```
using System;
class Ovn_6_5
{
```

```
   static void Main()
   {
      char char1, char2;
      String temp;

      do
      {
         Console.Write("\n\tMata in char1:\t");
         temp = Console.ReadLine();
         char1 = Convert.ToChar(temp.Substring(0, 1));
         Console.Write("\n\tMata in char2:\t");
         temp = Console.ReadLine();
         char2 = Convert.ToChar(temp.Substring(0, 1));

         if (char1 < char2)
            Console.WriteLine("\n\tOK. Du matade in i rätt " +
                                                   "ordning.\n");
         else
            Console.WriteLine(
                      "\n\tDu matade in i fel ordning. " +
                                                "Försök igen!\n");
      } while (char1 >= char2);
   }
}
*********************************************************************
```

Ovn_6_6.cs

Skriv ett program som läser in ett heltal och använder det som stegvariabel för att skriva ut talen från 1 till 5000. Om steget är t.ex. 5 skrivs var femte tal ut.

```
using System;
class Ovn_6_6
{
   static void Main()
   {
      Console.Write("\n\tMata in ett heltal för steget:\t");
      int steg = int.Parse(Console.ReadLine());

      Console.WriteLine();

      for (int i = 0; i <= 5000; i += steg)
          Console.Write(i + "\t");

      Console.WriteLine();
   }
}
*********************************************************************
```

Ovn_6_7.cs

Förbättra (effektivisera) lösningen till övn 5.5 (sid 109) med for-satser.

```
using System;
class Ovn_6_7
{
  static void Main()
  {
    String encrypt = "";
```

```
    Console.Write("\nMata in 5 tecken utan mellanslag:        ");
    String word = Console.ReadLine();

    Console.Write("\nMata in ett heltal (krypteringsnyckel): ");
    int nyckel = int.Parse(Console.ReadLine());

    for (int i = 0; i < 5; i++)
      encrypt += (char) (Convert.ToChar(word.Substring(i, 1)) + key);
    Console.WriteLine("\nDet krypterade ordet:\t\t\t" + encrypt);

    word = "";
    for (int i = 0; i < 5; i++)
      word += (char) (Convert.ToChar(encrypt.Substring(i, 1)) - key);
    Console.WriteLine("\nDet återställda ordet:\t\t\t" + word+ '\n');
  }
}
**********************************************************************
```

Kapitel 7 Metoder, sid 174:

Ovn_7_1a.cs

Modularisera lösningen (sid 225) till övn 4.4 (sid 83) som läser in två heltal, gör beräkningar med dem och skriver ut resultaten. Separera en av beräkningarna, t.ex. multiplikationen från kodens andra delar inmatning och utmatning.

a) Flytta multiplikationen till en metod med returvärde med huvudet static int Mult(int a, int b) i samma klass som Main(). Anropa metoden Mult() från Main(). Bibehåll alla andra beräkningar. Se upp med att inte placera den nya metoden i Main(), utan före eller efter.

```
using System;
class Ovn_7_1a
{
   static void Main()
   {
      Console.Write("\n\tMata in ett heltal:\t\t");     // Ledtext
      int no1 = int.Parse(Console.ReadLine());    // Inläsning
      Console.Write("\n\tMata in ett heltal till:\t");
      int no2 = int.Parse(Console.ReadLine());
      Console.WriteLine("\n\n\t"                                        +
         no1 + " plus "   + no2 + " är " + (no1 + no2) + "\n\t"    +
         no1 + " minus " + no2 + " är "  + (no1 - no2) + "\n\t"    +
                                          // Anropet:
         no1 + " gånger " + no2 + " är " + Mult(no1, no2) + "\n\t" +
         no1 + " heltalsdividerad med "   +
                          no2 + " är " + (no1 / no2) + "\n\t"   +
         no1 + " modulo " + no2 + " är " + (no1 % no2) + "\n\t");
   }
----------------------------------------------------------------------
Metoden Mult() som tar in två heltal via sina parametrar a och b och
returnerar ett heltal som är a * b:
   static int Mult(int a, int b)                  // Metoden Mult()
   {
      return a * b;
   }
}
```

```
**********************************************************************
Ovn_7_1b.cs
Modularisera lösningen (sid 225) till övn 4.4 (sid 83) som läser in
två heltal, gör beräkningar med dem och skriver ut resultaten.
Separera en av beräkningarna, t.ex. multiplikationen från kodens
andra delar inmatning och utmatning.
b) Fortsätt med att flytta metoden Mult() till en annan klass i samma
   fil. Anropet ska fortfarande göras från Main(). Även här: Se upp
   med att inte placera den nya klassen i den gamla, utan före eller
   efter.

using System;
class Ovn_7_1b
{
  static void Main()
  {
    Console.Write("\n\tMata in ett heltal:\t\t");     // Ledtext
    int no1 = int.Parse(Console.ReadLine());    // Inläsning
    Console.Write("\n\tMata in ett heltal till:\t");
    int no2 = int.Parse(Console.ReadLine());
    Console.WriteLine("\n\n\t"                                          +
       no1 + " plus "   + no2 + " är " + (no1 + no2) + "\n\t"      +
       no1 + " minus " + no2 + " är "  + (no1 - no2) + "\n\t"      +
       no1 + " gånger " + no2 + " är " +      // Anropet:
                                  Multip.Mult(no1, no2) + "\n\t" +
       no1 + " heltalsdividerad med "   +
                           no2 + " är " + (no1 / no2) + "\n\t"     +
       no1 + " modulo " + no2 + " är " + (no1 % no2) + "\n\t");
  }
}

Ny klass Multip i samma fil som Ovn_7_1b.cs
class Multip                                        // Klassen Multip()
{
   public static int Mult(int a, int b)           // Metoden Mult()
   {
      return a * b;
   }
}
**********************************************************************

Ovn_7_1c.cs
Modularisera lösningen (sid 225) till övn 4.4 (sid 83) som läser in
två heltal, gör beräkningar med dem och skriver ut resultaten.
Separera en av beräkningarna, t.ex. multiplikationen från kodens
andra delar inmatning och utmatning.
c) Flytta den nya klassen samt metoden Mult() till en separat fil.

using System;
class Ovn_7_1c
{
  static void Main()
  {
    Console.Write("\n\tMata in ett heltal:\t\t");
    int no1 = int.Parse(Console.ReadLine());
    Console.Write("\n\tMata in ett heltal till:\t");
    int no2 = int.Parse(Console.ReadLine());
    Console.WriteLine("\n\n\t"                                          +
```

```
        no1 + " plus "   + no2 + " är " + (no1 + no2) + "\n\t"      +
        no1 + " minus " + no2 + " är "  + (no1 - no2) + "\n\t"      +
        no1 + " gånger " + no2 + " är " +      // Anropet:
                                    Multip.Mult(no1, no2) + "\n\t" +
        no1 + " heltalsdividerad med "   +
                            no2 + " är " + (no1 / no2) + "\n\t"     +
        no1 + " modulo " + no2 + " är "  + (no1 % no2) + "\n\t");
  }
}
```

Ovn_7_1cd.cs

Separat fil som borde ligga i samma projekt som filen Ovn_7_1c.cs och när programmet Ovn_7_1d körs, i samma projekt som filen Ovn_7_1d.cs

```
class Multip                                          // Klassen Multip
{
   public static int Mult(int a, int b)              // Metoden Mult()
   {
      return a * b;
   }
}
**********************************************************************
```

Ovn_7_1d.cs

Modularisera lösningen (sid 225) till övn 4.4 (sid 83) som läser in två heltal, gör beräkningar med dem och skriver ut resultaten. Separera en av beräkningarna, t.ex. multiplikationen från kodens andra delar inmatning och utmatning.
d) Gör samma sak med alla andra beräkningssätt. Lagra var och en klass med resp. metod i en separat fil. Anropa alla metoder från Main().

```
using System;
class Ovn_7_1d
{
  static void Main()
  {
    Console.Write("\n\tMata in ett heltal:\t\t");
    int no1 = int.Parse(Console.ReadLine());
    Console.Write("\n\tMata in ett heltal till:\t");
    int no2 = int.Parse(Console.ReadLine());
    Console.WriteLine("\n\n\t" +                  // Anropen:
    no1 + " plus "   + no2 + " är " + Addit.Add(no1, no2) + "\n\t"  +
    no1 + " minus " + no2 + " är "  + Subtr.Sub(no1, no2) + "\n\t"  +
    no1 + " gånger " + no2 + " är " +          // Anropet:
                                    Multip.Mult(no1, no2) + "\n\t"  +
    no1 + " heltalsdividerad med "  +
                  no2 + " är "  +  Div.IntDiv(no1, no2) + "\n\t"   +
    no1 + " modulo " + no2 + " är " + Modu.Mod(no1, no2) + "\n\t");
  }
}
```

Ovn_7_1dA.cs

Separat fil i samma projekt som filen Ovn_7_1d.cs

```
class Addit                                           // Klassen Addit
{
   public static int Add(int a, int b)               // Metoden Add()
   {
```

```
      return a + b;
   }
}
```

Ovn_7_1dS.cs

Separat fil i samma projekt som filen Ovn_7_1d.cs

```
class Subtr                                          // Klassen Subtr
{
   public static int Sub(int a, int b)               // Metoden Sub()
   {
      return a - b;
   }
}
```

Ovn_7_1dD.cs

Separat fil i samma projekt som filen Ovn_7_1d.cs

```
class Div                                           // Klassen Div
{
   public static int IntDiv(int a, int b)          // Metoden IntDiv()
   {
      return a / b;
   }
}
```

Ovn_7_1dM.cs

Separat fil i samma projekt som filen Ovn_7_1d.cs

```
class Modu                                           // Klassen Modu
{
   public static int Mod(int a, int b)              // Metoden Mod()
   {
      return a % b;
   }
}
```

****

Ovn_7_2.cs

Modularisera programmet Operator (sid 71) genom att skriva dess bearbetningsdel som en ny metod i samma klass. Bibehåll in- och utmatnigsdelen i Main() och anropa den nya metoden från Main(). Avgör själv om den nya metoden ska returnera ett värde och om den ska vara statisk. Ge den ett beskrivande namn.

```
using System;
class Ovn_7_2
{
   static void Main()
   {
      /* I n m a t n i n g */
      Console.Write("\n\tAnge antal år:\t\t");
      int year = int.Parse(Console.ReadLine());   // Nästlat anrop
      Console.Write("\n\tAnge antal månader:\t");
      int months = int.Parse(Console.ReadLine());
      Console.Write("\n\tAnge antal veckor:\t");
      int weeks = int.Parse(Console.ReadLine());
      Console.Write("\n\tAnge antal dagar:\t");
      int days = int.Parse(Console.ReadLine());
```

```
      /* U t m a t n i n g */
      Console.WriteLine("\n\t" + year + " år, "                        +
           months + " månader, " + weeks + " veckor och "              +
           days   + " dagar är " + Total(year, months, weeks, days) +
           " dagar totalt." + '\n');
   }

   static int Total(int y, int m, int w, int d)  // Ny metod Total()
   {                                              // med returvärde
      /* B e a r b e t n i n g */
      return 365*y + 30*m + 7*w + d;
   }
}
*********************************************************************
```

Ovn_7_3.cs

Vänd om problemet från övn 9.2: Modularisera programmet OverloadOp (sid 76) genom att flytta bearbetnings- och utmatnigsdelen till en void-metod. Dvs skriv ett program som läser in tiden i ett antal dagar, anropar void-metoden som omvandlar tiden till antal år, månader, veckor och restdagar och skriver ut resultaten. Använd för omvandlingen den algoritm som är implementerad i programmet OverloadOp. Varför är det inte lämpligt här att använda en metod med returvärde?

```
using System;
class Ovn_7_3
{
   static void Main()
   {
      /* I n m a t n i n g */
      Console.Write("\n\tAnge antal dagar:\t");
      int totalDays = int.Parse(Console.ReadLine());
      Conversion(totalDays);                  // Anropet av void-metod
   }

   static void Conversion(int total)          // void-metod
   {
      int year, months, weeks, restDays;

      /* B e a r b e t n i n g */
      year     =   total / 365;
      months   =  (total % 365) / 30;
      weeks    = ((total % 365) % 30) / 7;
      restDays = ((total % 365) % 30) % 7;

      /* U t m a t n i n g */
      Console.WriteLine("\n\t" + total                                  +
          " dagar är " + year + " år, "   + months   + " månader, " +
                weeks + " veckor och " + restDays + " dagar.\n");
   }
}
```

Det är inte lämpligt att använda en metod med returvärde, därför att en metod med returvärde endast kan returnera ETT värde. Här behövs FYRA värden som ska skrivas ut. Void-metoden beräknar OCH skriver ut dem.

****

Ovn_7_4_Test.cs
Skriv först ett program med endast Main()-metoden som läser in sida till en kub samt beräknar och skriver ut kubens volym sida ^{3} och dess yta 6 x sida ^{2} . Flytta sedan dessa beräkningar till två metoder, en för volymen, en för ytan, båda i en separat klass Cube. Definiera side som en datamedlem i klassen Cube. Avgör om metoderna Volume() och Area() ska returnera eller vara av void-typ. Anropa dem från Main(). Skriv först en variant med statiska metoder, byt sedan till icke-statiska metoder. Testa båda varianter. Avgör slutligen själv vilken variant som ska föredras om lösningen ska vara objektorienterad. OBS! Följande lösningsförslag visar endast den optimala varianten.

```
using System;
class CubeTest
{
  static void Main()
  {
    Cube myCube;            // Definierar en referensvariabel
                            // av typ Cube utan att skapa objektet
    myCube = new Cube();    // Skapar ett objekt av typ Cube och
                            // tilldelar objektets adress till re-
                            // ferensen. By default: side = 0.0
                            // Sedan tilldelas side ett nytt värde:
    Console.Write("\n\tAnge sidan till en kub:\t");
    myCube.side = Convert.ToDouble(Console.ReadLine());

    Console.WriteLine("\n\tEn kub med sidan\t"  + myCube.side   +
                      "\n\thar volymen\t\t" + myCube.Volume()   +
                    "\n\toch ytan\t\t"    + myCube.Area()   + '\n');
  }
}
```

Ovn_7_4_Class.cs
Separat fil i samma projekt som filen Ovn_9_4_Test.cs

```
class Cube
{
   public double side;

   public double Volume()
   {
      return side * side * side;
   }

   public double Area()
   {
      return 6 * side * side;
   }
}
```

****

Ovn_7_5.cs
Varför ger följande program kompileringsfel? Åtgärda felet genom att flytta på kod, utan att ta bort någon klammer

och utan att ha tomma klamrar:

```
class Ovn_9_5
{
   static void Main()
   {
      {
         int t = 30;
      }
      Console.WriteLine("t = " + t);
   }
}
```

```
using System;
class Ovn_7_5
{
   static void Main()
   {
      int t;
      {
         t = 30;
      }
      Console.WriteLine("\n\tt = " + t + '\n');
   }
}
```

Kompileringsfelet i programmets första variant berodde på att variabeln t var definierad i ett inre block och att programmet refererade till den utanför det inre blocket där t inte längre var giltig.

****

Ovn_7_6_Test.cs
Modularisera programmet MiniSort från kap 6 (sid 116).

```
using System;
class MiniSortTest
{
  static void Main()
  {
    Console.Write("\n\tTvå osorterade tecken:\n\n\t" +
            "Mata in två tecken skilda med mellanslag:\t");
    string str = Console.ReadLine();

    MiniSort m = new MiniSort();                    // Objekt skapas
    m.char1 = Convert.ToChar(str.Substring(0, 1));  // Objektets data
    m.char2 = Convert.ToChar(str.Substring(2, 1));  // initieras
    m.sortera();                           // Objektets metod anropas

    Console.WriteLine("\n\tDe två tecknen sorterade:\t\t\t" +
                           m.char1 + ' ' + m.char2 + "\n\n");
   }
}
```

Ovn_7_6_Class.cs
Separat fil i samma projekt som filen Ovn_9_6_Test.cs

```
class MiniSort
```

```
{
   public char char1, char2;

   public void sortera()
   {
      char temp;
      if (char1 > char2)                // Här tolkas tecknen som tal
      {
         temp    = char1;               // Algoritm för platsbyte
         char1 = char2;                 // av de två teckenvärdena
         char2 = temp;                  // char1, char2
      }
   }
}
*********************************************************************
```

Ovn_7_7_Test.cs

Modularisera programmet OverloadOp från kap 4 (sid 76).

```
using System;
class TidTest
{
  static void Main()
  {
    /* I n m a t n i n g */
    Console.Write("\n\tAnge antal dagar:\t");
    int totalDays = int.Parse(Console.ReadLine());

    Tidomvandling t = new Tidomvandling(); // Objekt skapas
    t.Conversion(totalDays);               // Objektets metod anropas

    /* U t m a t n i n g */
    Console.WriteLine("\n\t" + totalDays + " dagar är " +
           t.year + " år, " + t.months + " månader, " +
           t.weeks + " veckor och "      + t.restDays  +" dagar.\n");
  }
}
```

Ovn_7_7_Class.cs

Separat fil i samma projekt som filen Ovn_9_7_Test.cs

```
class Tidomvandling
{
   public int year, months, weeks, restDays;

   public void Conversion(int total)
   {
      /* B e a r b e t n i n g */
      year        =   total / 365;
      months   =  (total % 365) / 30;
      weeks    = ((total % 365) % 30) / 7;
      restDays = ((total % 365) % 30) % 7;
   }
}
*********************************************************************
```

Kapitel 8 Klasser, objekt och referenser, sid 191:

Ovn_8_1.cs

*Skriv ett program som består endast av klassen All_in_Main som i sin tur innehåller endast Main()-metoden. Läs in radien r till en cirkel och beräkna samt skriv ut cirkelns area pi*r*r och dess omkrets 2*pi*r, där pi=3.14159. Du kan använda konstanten Math.PI från C#:s klassbibliotek för pi. Programmet ska inte vara objektorienterat eftersom du inte skapar några objekt, utan endast lokala variabler (radie, area, omkrets). Programmet ska inte heller vara modulariserat eller proceduralt eftersom all kod (Input-Bearbetning-Output) finns i en enda metod Main() som definieras i en klass. Dessa steg ska tas i de efterföljande två övningarna. Deklarera alla variabler till double.*

```
using System;
class All_in_Main
{
   static void Main()
   {
      double radius, area, circumference;        // Lokala variabler

      Console.Write("\n\tAnge radien till en cirkel:\t");
      radius = Convert.ToDouble(Console.ReadLine());  // Input

      area = Math.PI * radius * radius;               // Bearbetning
      circumference = 2 * Math.PI * radius;

      Console.WriteLine(                              // Output
              "\n\tEn cirkel med radien " + radius +
              "\n\thar arean           "  + area  +
              "\n\toch omkretsen       "  + circumference + '\n');
   }
}
```

*****

Ovn_8_2.cs

*Modularisera programmet All_in_Main från övn 8.1 på metodnivå, dvs: Flytta bearbetningsdelen dvs beräkningen av area och omkrets ur Main() till separata metoder Area() och Circumference(), men stanna i samma klass. Döp om klassnamnet till Procedural. I Main() ska finnas kvar variabeln för radien, inmatning, utmatning och anropet av Area() och Circumference(). Förse de nya metoderna med en parameter som överför radiens värde från Main() till dem. Välj olika namn för den aktuella än för den formella parametern.Dessutom ska Area() och Circumference() returnera ett double-värde och vara statiska. För att testa mata in 1 för radien.Då ska arean bli pi pga pi*r*r = pi och omkretsen bli 2*pi.*

```
using System;
class Procedural
{
  static void Main()
  {
    double radius;                              // Lokal variabel

    Console.Write("\n\tAnge radien till en cirkel:\t");
    radius = Convert.ToDouble(Console.ReadLine());  // Input
```

```
    Console.WriteLine(                                  // Output
        "\n\tEn cirkel med radien " + radius                +
        "\n\thar arean           "  + Area(radius)          +
        "\n\toch omkretsen       "  + Circumference(radius) + '\n');
  }                                                  // aktuell parameter
-------------------------------------------------------------------------
   static double Area(double r)          // Metoden Area() med formell
   {                                     // parameter r som tar emot
      return Math.PI * r * r;            // aktuell parameter radius
   }
-------------------------------------------------------------------------
   static double Circumference(double r)  // Metoden Circumference()
   {
      return 2 * Math.PI * r;
   }
}
*************************************************************************
```

Ovn_8_3_Class.cs

Modularisera programmet All_in_Main från övn 8.1 på klassnivå, dvs: Dela upp programmet i två klasser, lagrade i två separata filer. Kalla den ena klassen för Circle, den andra för CircleTest. Samla all information om begreppet cirkel i klassen Circle, dvs: Deklarera radien r som datamedlem samt Area() och Circumference() som metoder. Ta bort från metoderna både static och parametern för radien.

```
using System;                                  // Krävs för Math
class Circle
{
   public double radius;                       // Publik datamedlem

   public double Area()                        // Publik metod
   {
      return Math.PI * radius * radius;
   }

   public double Circumference()               // Publik metod
   {
      return 2 * Math.PI * radius;
   }
}
```

Datamedlemmen radius och metoderna Area() och Circumference() måste vara publika för att den externa klassen CircleTest ska kunna komma åt dem.

Ovn_8_3_Test.cs

*Den andra klassen CircleTest ska endast innehålla metoden Main(). Ska-
pa i den ett objekt av klassen Circle. Läs in ett värde till objektets
datamedlem r och anropa samt skriv ut returvärdena till objektets me-
toder Area() och Circumference(). Båda klassfiler borde ligga i samma*

projekt.

```
using System;
class CircleTest
{
   static void Main()
   {
      Circle  myCircle;                    // Definirerar endast en
referensvariabel
                                 //  av  typ  Circle  utan  att  skapa
objekt
      myCircle = new Circle();  // Skapar ett objekt av typ Circle
                                 //  och  tilldelar  objektets  adress
till
                                 // referensvariabeln.
      Console.Write("\n\tAnge radien till en cirkel:\t");
      myCircle.radius  =  Convert.ToDouble(Console.ReadLine());     //
Input

      Console.WriteLine(                                          //
Output
          "\n\tEn cirkel med radien " + myCircle.radius          +
          "\n\thar arean           "  + myCircle.Area()          +
          "\n\toch omkretsen       "  + myCircle.Circumference() +
'\n');
   }
}
**********************************************************************
```

Ovn_8_4_Class.cs
Skriv en klass Fish som beskriver en fisk med datamedlemmarna sort, weight och size. Borde ligga i samma projekt som filen Ovn_8_4_Test.

```
class Fish
{
   public string sort;
   public double weight, size;
}
```

Ovn_8_4_Test.cs
Testa din klass i en annan klass FishTest i en separat fil som endast innehåller metoden Main() där två objekt av klassen Fish skapas. Tilldela det första objektets datamedlemmar värdena Laxforell, 719 (gram) och 38.5 (cm). Enheterna gram och cm behöver inte anges. Välj själv andra värden till det andra objektets datamedlemmar. Skriv ut dessa värden till konsolen i en tabell av typ:

Fisksort	*Vikt i g*	*Längd i cm*
Laxforell	*719.0*	*38.5*
Torsk	*423.0*	*28.7*

```
using System;
```

```
class FishTest
{
   static void Main()
   {
      Fish f1 = new Fish();            // Objekt skapas (definieras)
                                       // och initieras by default

      f1.sort  = "Laxforell";          // Objekt tilldelas värden
      f1.weight  = 719;
      f1.size = 38.5;

      Fish f2 = new Fish();            // 2:a objekt skapas
      f2.sort  = "Torsk\t";            // \t för layoutens skull
      f2.weight  = 423;
      f2.size = 28.7;

      Console.WriteLine("\n\tFisksort\tVikt i g\tLängd i cm"     +
           "\n\t-----------------------------------------\n\t"  +
      f1.sort + "\t " + f1.weight + "\t\t   " + f1.size + "\n\t" +
      f2.sort + "\t " + f2.weight + "\t\t   " + f2.size + "\n\n");
   }
}
```

*****

Ovn_8_5_Class.cs

Ta klassen Fish från övn 8.4. Förse den med en metod som beräknar priset på fisken oberoende av sort, t.ex. 7.25 kr per hekto. Lägg till även en metod som beräknar och returnerar frakten utifrån fiskens vikt och genom att t.ex. multiplicera en viss kostnadsfaktor, säg 0.02, med vikten, en annan, säg 0.1, med längden och addera dem. Metoderna ska returnera priset och frakten i hela kronor utan ören.

```
using System;
class Fish
{
   public string sort;
   public float weight, size;

   public int Price()
   {
      return (int) Math.Round(weight * 7.25 / 100);
   }

   public int shipping()
   {
      return (int) Math.Round(weight * 0.02 + size * 0.1);
   }
}
```

Ovn_8_5_Test.cs

Anropa metoderna från klassen FishTest:s Main()-metod för de två Fish-objekten. Lägg till nya rubriker Pris och Frakt i tabellen ovan och skriv ut deras värden till tabellens två rader

```
using System;
```

```
class FishTest
{
  static void Main()
  {
    Fish f1 = new Fish();          // 1:a objekt skapas (definieras)
                                   // och initieras by default

    f1.sort  = "Laxforell";        // 1:a objekt tilldelas värden
    f1.weight  = 719;
    f1.size = 38.5f;

    Fish f2 = new Fish();          // 2:a objekt skapas

    f2.sort  = "Torsk\t";          // \t för layoutens skull
    f2.weight  = 423;              // 2:a objekt tilldelas värden
    f2.size = 28.7f;
                                   // Metoderna anropas i utskriften:
    Console.WriteLine("\nFisksort\tVikt i g\tLängd i cm"            +
                                              "\tPris\tFrakt\n"   +
  "-------------------------------------------------------------\n" +
  f1.sort    + "\t " + f1.weight     + "\t\t   "+ f1.size + "\t\t " +
  f1.Price() + "\t " + f1.shipping() + "\n"                         +
  f2.sort    +"\t "  + f2.weight     + "\t\t   " + f2.size + "\t\t "+
  f2.Price() + "\t " + f2.shipping() + "\n\n");
  }
}
*********************************************************************
```

Ovn_8_6_Test.cs

Modifiera programmet från övn 8.5 så att datamedlemmarnas värden inte hårdkodas utan läses in. Utskriften ska skickas till konsolen och läggas till tabellen ovan. Skriv din kod så att den lätt kan generaliseras så att man kan mata in flera fisksorter med hjälp av en loop och en array av referenser till Fish-objekt som vi kommer att lära oss senare. Dessutom ska programmet kunna modifieras till att skriva ut till en tabell i en fil eller en databas istället för att skriva till konsolen.

```
using System;
class FishTest
{
  static void Main()
  {
    Fish f1 = new Fish();                        // 1:a objekt skapas
    Fish f2 = new Fish();                        // 2:a objekt skapas

    Console.Write("\n\tMata in sorten till fisk1:\t");
    f1.sort = Console.ReadLine();                              // Inputs:
    if (f1.sort. Length < 6) f2.sort += '\t';
    Console.Write("\tMata in vikten till fisk1:\t");
    f1.weight = (float) Convert.ToDecimal(Console.ReadLine());
    Console.Write("\tMata in längden till fisk1:\t");
    f1.size = (float) Convert.ToDecimal(Console.ReadLine());

    Console.Write("\n\tMata in sorten till fisk2:\t");
    f2.sort = Console.ReadLine();                     // Input
    if (f2.sort.Length < 6) f2.sort += '\t';
    Console.Write("\tMata in vikten till fisk2:\t");
    f2.weight = (float) Convert.ToDecimal(Console.ReadLine());
```

```
    Console.Write("\tMata in en till fisk2:\t");
    f2.size = (float) Convert.ToDecimal(Console.ReadLine());

    Console.WriteLine("\n\nFisksort\tVikt i g\tLängd i cm"          +
                                              "\tPris\tFrakt\n"    +
 "-------------------------------------------------------------\n"  +
 f1.sort    + "\t " + f1.weight     + "\t\t   " + f1.size + "\t\t " +
 f1.Price() + "\t " + f1.shipping() + "\n"                          +
 f2.sort    + "\t " + f2.weight     + "\t\t   " + f2. + "\t\t "     +
 f2.Price() + "\t " + f2.shipping() + "\n\n");
  }
}
*********************************************************************
```

Ovn_8_7_Class.cs

Deklarera en klass Triangle med datamedlemmarna side_a, side_b, side_c, height_b av typ int och metoderna Area(), Circumference().

```
class Triangle
{
   public int side_a, side_b, side_c, height_b;

   public int Area()
   {
      return side_b * height_b/2;
   }

   public int Circumference()
   {
      return side_a + side_b + side_c;
   }
}
```

Ovn_8_7_Test.cs

Skapa i en annan klass som innehåller Main(), ett objekt av klassen Triangle och tilldela datamedlemmarna värden. Anropa metoderna och skriv ut denna triangels area och omkrets. Skapa en andra referens som pekar på samma objekt och anropa metoderna samt skriv ut deras returvärden med denna referens. Du borde få samma resultat som med den första referensen. Anropa sedan metoderna Area() och Circumference() med två anonyma objekt (utan referenser). Kolla om du får de förväntade resultaten som är baserade på objektens default-initiering. Sist, peka om Triangle-objektets första referens till null och försök att anropa metoderna med denna referens. Vad händer?

```
using System;
class TriangleTest
{
  static void Main()
  {
    Triangle tri = new Triangle();   // Skapar ett objekt med en
                                     // första referens tri
    tri.side_a = 4;
    tri.side_b = 6;
    tri.side_c = 5;
    tri.height_b = 3;
```

```
    Console.WriteLine("\n\tMed den första referensen:\n"          +
                      "\tArea    = " + tri.Area()          + '\n' +
                      "\tOmkrets = " + tri.Circumference() + '\n');
    Triangle t = tri;                // Ny referens till samma objekt

    Console.WriteLine("\n\tMed den andra referensen:\n"  +
                      "\tArea    = " + t.Area()          + '\n' +
                      "\tOmkrets = " + t.Circumference() + '\n');

    Console.WriteLine
            ("\n\tAndra, anonyma objekt som default-initieras:\n"   +
             "\tArea    = " + new Triangle().Area()   +    '\n'     +
             "\tOmkrets = " + new Triangle().Circumference() + '\n');

    tri = null;                      // Ompekning till null: tri
                                     // pekar på inget objekt längre
    Console.WriteLine("Användning av null-referens ger " +
                                                 "exekveringsfel:\n");
    Console.WriteLine(tri.side_a);
   }
}
```

Det som händer, är att ett objekt skapas med referensen tri som överförs till en ny referens t, så att både tri och t pekar på samma objekt. Men sedan görs en ompekning av tri till null, dvs tri kopplas bort från objektet. Programmets sista sats försöker att med tri referera till objektet vilket leder till ett s.k. NullReferenceException.

****

Kapitel 9 Array, sid 209:

Ovn_9_1.cs

Skriv ett program som läser in 10 heltal från konsolen, lagrar dem i en array och skriver ut dem i omvänd ordning.

```
using System;
class Ovn_9_1
{
   static void Main()
   {
      int[] no = new int[10];

      Console.WriteLine("\n\tSkriv in 10 heltal:\n");
      for (int i = 0; i <= 9; i++)
      {
          Console.Write("\tTal nr " + (i+1) + ":\t");
          no[i] = int.Parse(Console.ReadLine());
      }

      Console.WriteLine("\nDina tal i omvänd ordning:\n");
      for (int i = 9; i >= 0; i--)
          Console.Write(no[i] + "\t");

      Console.WriteLine();
   }
```

```
}
**********************************************************************
```

Ovn_9_2.cs

Skriv ett program som slumpar fram 1000 heltal mellan 60 och 140 (tänkbara hastigheter på en motorväg), lagrar dem i en array kallad hastighet, beräknar och skriver ut deras medelvärde med förklarande text.

```
using System;
class Ovn_9_2
{
  static void Main()
  {
    Random r = new Random();
    int[] hastighet = new int[1000];
    RandArray.Rand(r, hastighet, 60, 140);
    int sum = 0;
    for (int i = 0; i <= 999; i++)
        sum += hastighet[i];
    Console.WriteLine("\tMedelvärdet av 1000 möjliga hastigheter " +
                   "mellan 60 och 140 är:    "  +  sum/1000 + '\n');
  }
}
```

RandArray.cs

Separat fil i samma projekt som filen Ovn_9_2.cs
Ny metod Rand() slumpar fram en array av heltal mellan
a och b, lagrar dem i arrayen no och skriver ut dem
Anropar biblioteksmetoden Next() i en loop
för att få ETT slumptal mellan a och b i varje varv

```
using System;
class RandArray
{
   public static void Rand(Random r, int[] no, int a, int b)
   {
      Console.Write("\n\t" + no.Length + " heltal mellan " +
                      a + " och " + b + " slumpas fram:\n\n\t");
      for (int i=0; i < no.Length; i++)
      {
          no[i] = r.Next(a, b);
          Console.Write(no[i] + " ");
          if ((i % 16 == 0) && (i != 0))
             Console.Write("\n\t");
      }
      Console.WriteLine("\n\n");
   }
}
```

Ovn_9_3.cs

Skriv ett program som läser in en sträng, lagrar den i en array av char och skriver ut den baklänges.
Använd tekniken i programmet EncryptCharTest för att omvandla

*den inlästa strängen i en array av **char**.*

```
using System;
class Ovn_9_3
{
   static void Main()
   {
      Console.Write("\n\tSkriv in text:\t\t");
      char[] text = Console.ReadLine().ToCharArray();

      Console.Write("\n\tTexten baklänges:\t");
      for (int i = text.Length-1; i >= 0; i--)
          Console.Write(text[i]);
      Console.WriteLine('\n');
   }
}
```

****

Ovn_9_4.cs

Skriv ett program som läser in text i gemener, lagrar den i en array av char och skriver ut den framhävd i versaler och med mellanslag mellan varje tecken.

```
using System;
class Ovn_9_4
{
   static void Main()
   {
      Console.Write("\n\tSkriv in text:\t\t");
      char[] text = Console.ReadLine().ToCharArray();

      Console.Write("\n\tTexten framhävd:\t");
      for (int i = 0; i < text.Length; i++)
          Console.Write("" + (char) (text[i] - 32) + ' ');
      Console.WriteLine('\n');
   }
}
```

****

Ovn_9_5.cs

Skriv ett program som frågar efter användarens för- och efternamn, hälsar sedan användaren i en utskrift med fullständiga namnet, förnamnets längd samt efternamnets första och sista bokstav. Lös uppgiften generellt utan att använda information om något speciellt för- och efternamn.

```
using System;
class Ovn_9_5
{
   static void Main()
   {
      char surname0 = '0';        // Undviker villkorlig initiering
      Console.Write("\n\tSkriv in ditt för- och efternamn:\t");
      string input = Console.ReadLine();
      char[] name = input.ToCharArray();

      int i = 0;
      while (name[i] != ' ')      // Går igenom endast förnamnet
      {
         i++;
```

```
         if (name[i] == ' ')       // Hittar namnens avskiljare
            surname0 = name[i+1]; // Hittar efternamnets 1:a bokstav
      }

      Console.WriteLine("\n\tHej, "   + input     +
        "\n\tDitt förnamns längd är " + i         +
        "\n\tDitt efternamns första bokstav är "  + surname0 +
        "\n\tDitt efternamns sista  bokstav är "  +
                                        name[name.Length-1] + '\n');
   }
}
*********************************************************************
```

Ovn_9_6.cs

Skriv ett program där Main() läser in en persons fullständiga namn och hälsar tillbaka med namnets initialer. Dessa ska bestämmas och skrivas ut i en annan metod - med huvudet static void Initials(char[] name) som anropas i Main().

```
using System;
class Ovn_9_6
{
   static void Main()
   {
      Console.Write("\n\tSkriv in ditt för- och efternamn:\t");
      string input = Console.ReadLine();
      char[] dittNamn = input.ToCharArray();

      Console.Write("\n\tHej, " + input +
                    "\n\n\tDina initialer är\t\t\t" );

      Initials(dittNamn);              // Anropet

      Console.WriteLine('\n');
   }

   static void Initials(char[] name)  // Metoden
   {
      int i = 0;
      Console.Write(name[i]);          // Första initialen
      while (name[i] != ' ')           // Går igenom endast förnamnet
      {
         i++;
         if (name[i] == ' ')           // Hittar för- och efternamnets
                                       //                  avskiljare
            Console.Write(name[i+1]); // Andra initialen
      }
   }
}
*********************************************************************
```

Programförteckning

Kapitel 9 Array

Register

J

K

L

M

N

O

P

R

S

T

U

V,W

Ö