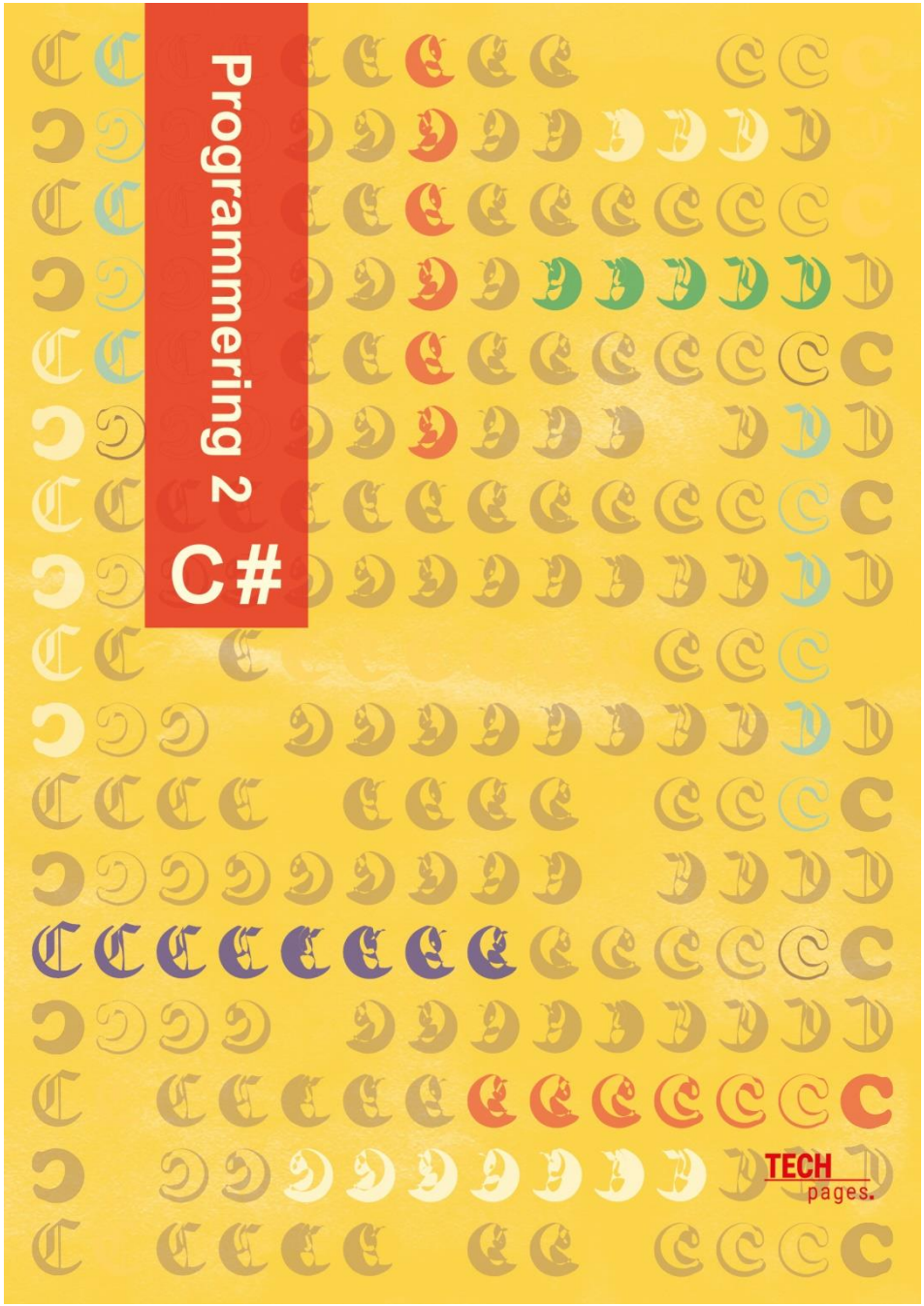

Programmering 2
C#
TECH
pages.

Med hjälp av *Programmering 2 med C#* kan du nu skriva intressanta applikationer i mycket större utsträckning än tidigare. Du lär dig Windowsprogrammering samt objektorienterad modellering och implementation – avgörande för professionell programutveckling, speciellt för webben. Bl.a. visar boken hur du själv kan utveckla en egen webbläsare, se bokens baksida. Kombinationen av interaktiva grafiska användargränssnitt *(GUI)* och webben gör dig till en professionell programmerare.

Programmering 2 med C# är en fortsättning på nybörjarböckerna:

Programmering 1 med C#
Programmering 1+ med C#

De hjälper nybörjaren att komma över den tröskel som är avgörande för att det ska bli roligt att programmera. För deras innehåll se bokens inre och yttre baksida.

Valet av programmeringsspråket är av underordnad betydelse. C# är ett medel, ett verktyg för att presentera programmering. 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.

Printed in Sweden
Published by TechPages Förlag AB
www.techpages.se
ISBN 978-9-197-42043-3

Programmering 2

med **C#**

Fortsättning på *Programmering 1 med C#*

Täcker Skolverkets kursplan för Programmering 2

Med övningar,
fullständiga lösningar
&
projektuppgifter

www.techpages.se TechPages Förlag

Titel: Programmering 2 med C#
ISBN: 978-9-197-42043-3

Tel: 08-792 36 28
www.techpages.se

Tryckeri: Eprint, Stockholm
Augusti 2021

Välkommen till Programmering 2

Efter att ha lärt sig grunderna i programmering öppnas helt nya möjligheter att skriva intressanta applikationer i mycket större utsträckning än tidigare. Boken innehåller bl.a.:

- Windowsprogrammering
- Interaktiva grafiska applikationer *(GUI)*
- En egen webbläsare
- Objektorienterad programmering & modellering
- Language Integrating Query *(LINQ)*
- Sökning och sortering
- Kryptering med slumptal
- Rekursion
- Generics

För att ha det lite roligt i början startar vi med Windowsprogrammering – små grafiska applikationer med möjligheter till interaktion och menyval, bl.a. en egen webbläsare. Boken fortsätter sedan med de teoretiskt tyngre bitarna: objektorienterad programmering och modellering, filhantering, kryptering osv.

Med dessa verktyg i handen kommer det inte längre finnas några gränser för din kreativitet, uppfinningsrikedom och fantasi. Hemligheterna bakom IT kommer att avslöjas för dig en efter den andra. Kombinationen av grafiska gränssnitt och webben gör dig till en professionell programmerare.

Denna bok är en fortsättning på *Programmering 1 med C#* och *Programmering 1+ med C#*. Vissa delar kan ha repetitiv karaktär för att underlätta förståelsen. Innehållet täcker Skolverkets kursplan för ***Programmering 2***. Men precis som nybörjarböckerna innehåller denna bok – utöver Skolverkets kursplan – en hel del extra material för att förmedla relevant kunskap, befästa samt fördjupa kunskapen och göra pliktlektyren mer intressant.

Programmering 2 med C# utvecklas och uppdateras permanent. Därför tas all form av kritik, korrekturanmärkningar såväl som förslag till förbättringar av både form och innehåll tacksamt emot på adressen **`info@techpages.se`**.

Anmärkningar

1. Denna upplaga av boken är förnyad och uppdaterad i många avseenden gentemot tidigare versioner. Den har anpassats till de nya kursplanerna. En del av innehållet har flyttats till *Programmering 3*. Andra delar har integrerats från *Programmering 1*.

2. Alla programexempel inkl. övningarnas fullständiga lösningsförslag är utvecklade och testade i Visual Studio 2019. Några av bokens grafiska programexempel kan dock innehålla layout (fönster, dialogrutor osv.) som härstammar från äldre versioner.

3. Instruktioner för installation, konfiguation och användning av Visual Studio 2019 kan hittas i bokens ***Appendix***, sid 265.

4. Denna bok är en fortsättning på *Programmering 1 med C#* och *Programmering 1+ med C#*. Alla hänvisningar följer mönstret i följande exempel:

   Progr1+, 5 hänvisar till *Programmering 1+ med C#,* kapitel *5*
   Progr1+, 4.3 kapitel *4*, avsnitt *3*

   På liknande sätt hänvisas till *Programmering 1 med C#* som är en light version av *Programmering 1+ med C#*.

Innehållsförteckning

Kapitel 1

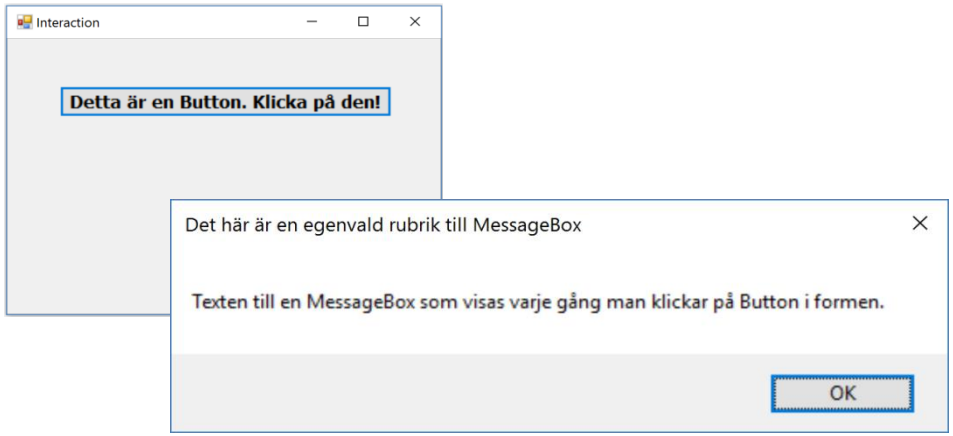
Windowsprogrammering

1.1 Interaktiva grafiska gränssnitt

Windowsprogrammering handlar om att utveckla program som involverar både text och grafik samt producerar fönster och dialogrutor av olika slag – samma grafiska komponenter som även används i operativsystemet *Windows*. Dessutom ska användaren kunna *interagera* med sådana program via grafiska gränssnitt, s.k. *Graphical User Interfaces (GUI)* som byggs både med förprogrammerade komponenter i Visual Studio och med egenskriven C#-kod. Det här kapitlet är en fortsättning samt fördjupning på *Windows Forms Applications* som introducerades kort i *Progr1, 1.3 – 1.5*.

Ett grafiskt gränssnitt är en yta som vi kan använda 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*. Även andra typer av händelser är tänkbara som påverkar både programförloppet och avslutningen i en mycket större utsträckning än det är fallet med rena textbaserade program. 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 reagerar på musklickning och 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

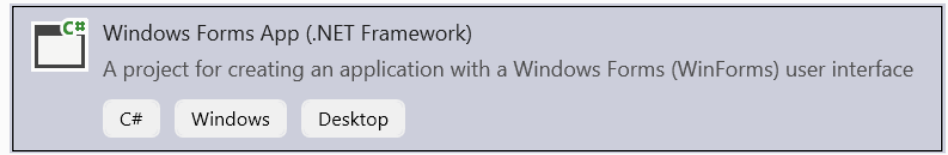
Förprogrammerade grafiska komponenter i Visual Studio kallas för Controls. Ex.: TextBox, Label, Button, … . Man kan dra dem med musen från verktygslådan Toolbox och placera dem i formfönstret. För att få funktionalitet i dem skrivs kod ”bakom” dem.

Hur man bygger applikationen ovan ska vi gå igenom nu. Läs om *projekt* på sid 268.

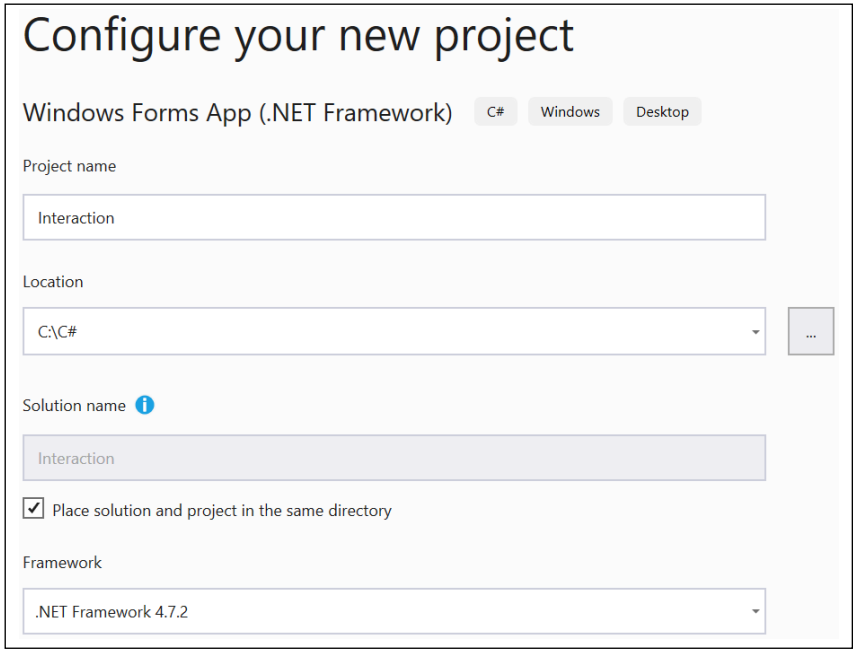
Windows Forms Application

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:

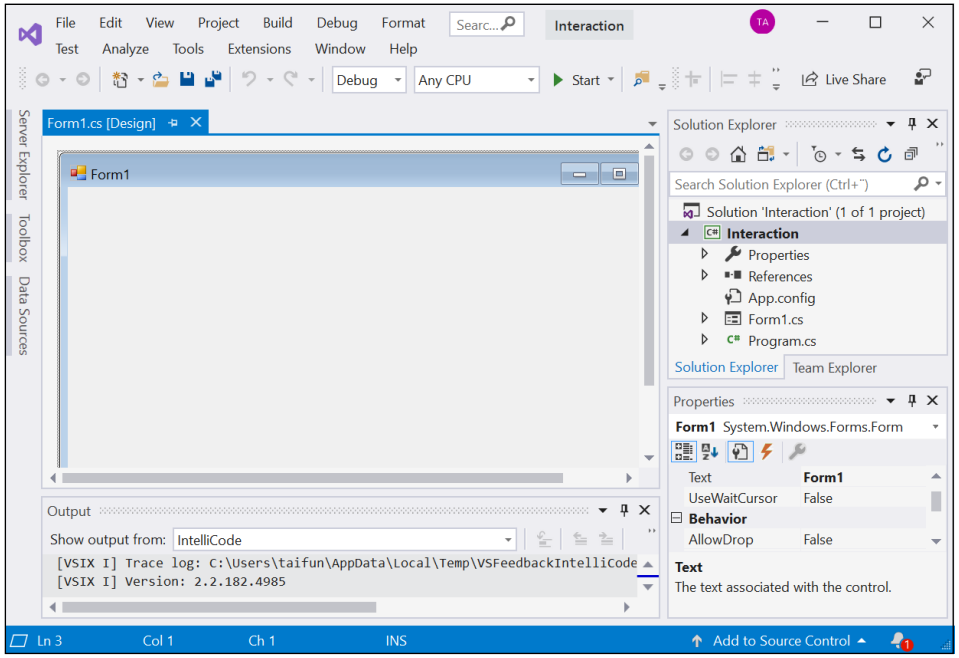


Markera rutan ovan. Klicka sedan i dialogrutan Create a new project som omfattar denna ruta, på knappen Next längst ned till höger. En ny dialogruta dyker upp med rubriken Configure your new project. Fyll i den uppgifterna enligt följande:



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 knappen 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

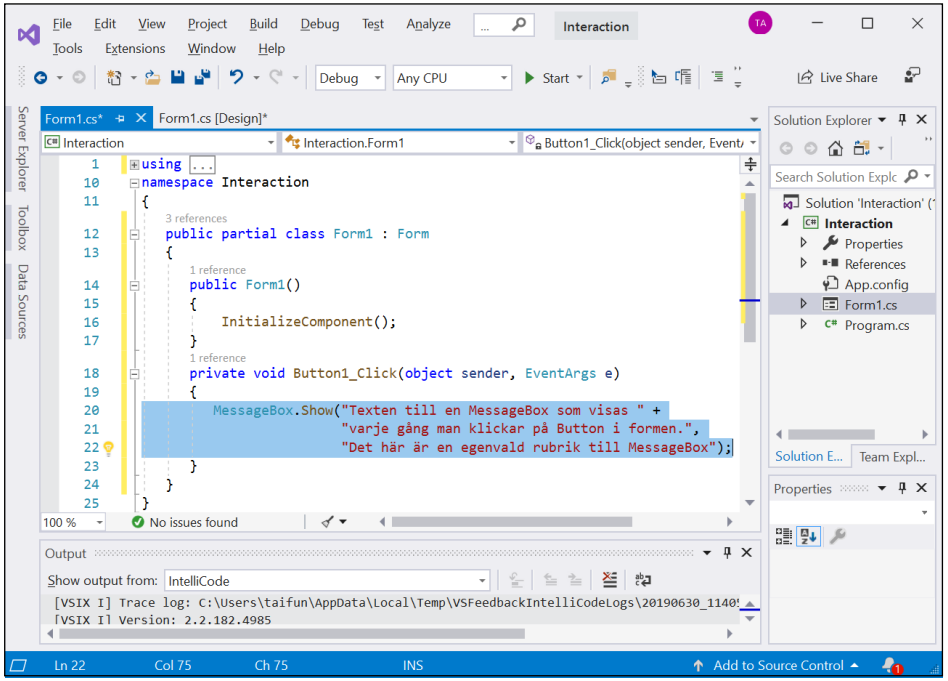
Expandera Common Controls och dubbelklicka på kontrollen Button, så att den hamnar i formfönstret. När du flyttar markören till formen stängs Toolbox-fönstret. Markera den nya kontrollen button1 på din form för att få fram dess egenskaper i Properties-fönstret.

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:

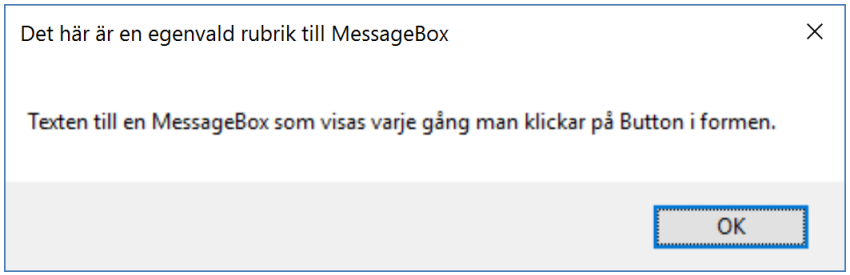
button1:

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. Namnutrymmen är ett bra och – i vissa fall – nödvändigt skydd mot namnkonflikter.

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 **`using`**-direktiv till som vi tagit bort, för de visar sig vara onödiga.

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 biblioteksklassen **`Form`**. I C# är **`:`** koden för arv[*]. Klassen **`Form`** i sin tur är deklarerad i namnutrymmet **`System.Windows.Forms`**. Där 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 deklareras först och anropas sedan. Både deklarationen och anropet sker med kod. En *händelsemetod* (eng.: *event handler*) deklareras 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 15), 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()`

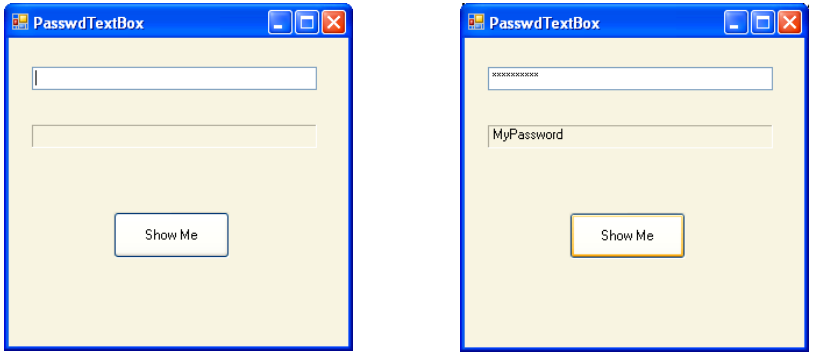
Till skillnad från **`button1_Click()`** är metoden **`Show()`** ingen händelsemetod, utan en vanlig metod. Därför anropas den med kod, inte med en händelse (musklickning). Den anropande koden står i händelsemetoden **`button1_Click()`**. Musklick på knappen med texten Detta är en Button. Klicka på den! (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.

---

[*] Läs om *arv* och *konstruktorn* på sid 69 och om *metoder* på sid 67.

1.2 TextBoxar, Buttons & Labels

Kontrollen TextBox ger oss möjligheten att från ett grafiskt gränssnitt mata in text i en ruta som vidareförs till programmet och kan bearbetas där. Denna kontroll demonstreras i ett program som följer och som kommer att ha följande output när det körs:



Först kommer det upp formen till vänster som innehåller tre olika kontroller, en TextBox, en Label och en Button. Den sista hade vi redan använt i projektet Interaction (sid 12). Skriver man en text i TextBoxen kommer den att maskeras av stjärnor, men klickar man på knappen Show Me kommer texten att visas i labeln under textrutan. Här vidareförs alltså den inmatade texten till programmet som ser till att den för det första syns som stjärnor i TextBoxen. För det andra visas den i klartext i Label-kontrollen och detta endast när man klickar på Show Me som är en kontoll av typ Button. Texten kan ju tänkas vara t.ex. ett lösenord eller något annat hemligt meddelande. Alla dessa kontroller med sina respektive funktionaliteter byggs i ett litet program som vi kallar för PasswdTextBox.

Gör så här för att skapa applikationen:

- Skapa en *Windows Forms Application* och döp den till PasswdTextBox. Hur man gör har vi lärt oss i projektet Interaction (sid 12).

  Sätt följande värde på egenskapen Text till formen Form1 så att formens rubrik bär programmets (projektets) namn. Låt alla andra värden vara oförändrade.

  Form1:

Egenskap	Värde
Text	PasswdTextBox
Size	310;420

- Hämta från Visual Studios Toolbox en TextBox-kontroll till formen och ändra värden till några av dess egenskaper enligt följande:

textBox1:

Egenskap	Värde
(Name)	tbPasswd
PasswordChar	*
Location.X	20
Location.Y	25
Size.Width	245
Size.Height	26

- Hämta från Toolbox en Label-kontroll till formen och sätt följande värden:

label1:

Egenskap	Värde
(Name)	lblShowPasswd
Text	
Location.X	20
Location.Y	75
BorderStyle	Fixed3D
Autosize	False
Size.Width	245
Size.Height	20

- Hämta från Toolbox en Button-kontroll till formen och gör samma sak här:

button1:

Egenskap	Värde
(Name)	btnShowMe
Text	Show Me
Location.X	90
Location.Y	150
Size.Width	100
Size.Height	40

Kod bakom Show Me-knappen

- Dubbelklicka på Show Me-knappen för att få upp formens kod, klassen **`Form1`** med den nya händelsemetoden **`btnShowMe_Click()`**.

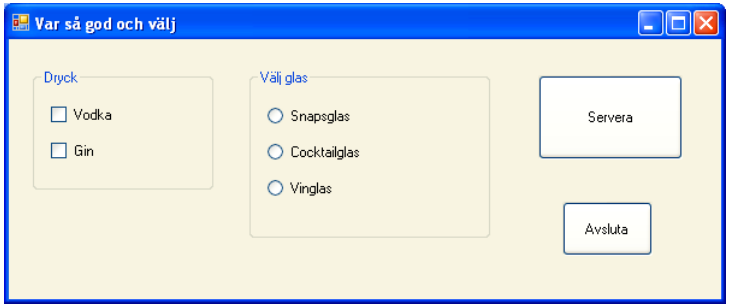
- Lägg in i den nya händelsemetoden **`btnShowMe_Click()`** följande kod:

```
lblShowPasswd.Text = tbPasswd.Text;
```

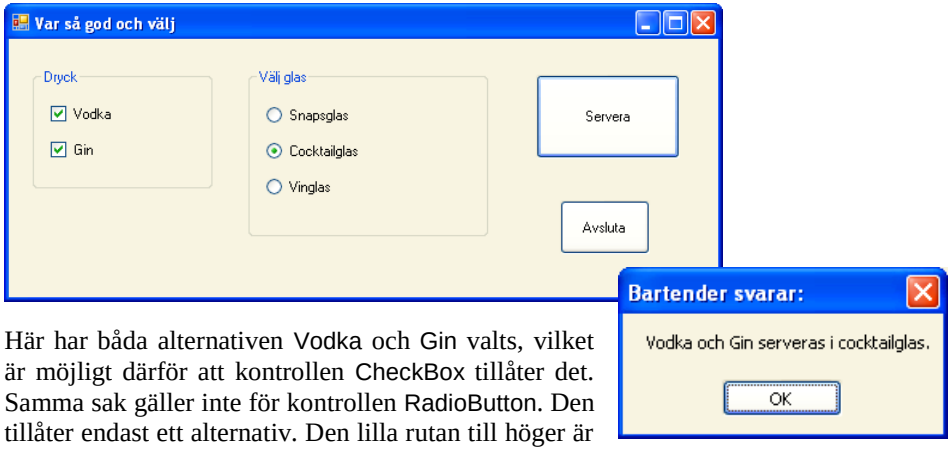
- Kompilera och kör. Skriv något i textboxen. Det visas bara stjärnor. Klickar du på Show Me-knappen visas texten i labeln.

1.3 CheckBoxar och radioknappar

I detta avsnitt vill vi bygga ett grafiskt gränssnitt som har ett antal alternativ som man kan välja mellan. Två sorters val kan förekomma i detta sammanhang: Ett- och flervalsalternativ. Ettvalsalternativ visas i grafiska gränssnitt ofta med små ringar, s.k. *radioknappar* som man markerar eller avmarkerar. Flervalsalternativ däremot visas med små rutor, s.k. *CheckBoxar* som man sätter en bock på eller bockar av. Både radioknappar och checkboxar är kontroller i Visual Studio och heter RadioButton resp. CheckBox. Programmet Bartender som vi ska bygga och vars grafiska gränssnitt visas nedan, använder båda kontroller grupperade under rubrikerna Dryck och Välj glas:



Rubriken Dryck grupperar två checkboxar, medan Välj glas grupperar tre radioknappar. Även själva grupperingen görs med en kontroll som heter GroupBox. Rutan ovan visas inledningsvis när programmet Bartender körs, innan någon interaktion gjorts. Sedan kan man välja dryck och glas samt klicka på knappen Servera för att få de valda alternativen ”serverade” i en MessageBox. Så här kan en sådan dialog se ut:



Här har båda alternativen Vodka och Gin valts, vilket är möjligt därför att kontrollen CheckBox tillåter det. Samma sak gäller inte för kontrollen RadioButton. Den tillåter endast ett alternativ. Den lilla rutan till höger är en MessageBox som kommer upp först när man klickar på knappen Servera. Knappen Avsluta är ett alternativ till det röda krysset i rutans högra övra hörn. Båda avslutar körningen. Innan man avslutar kan man efter att klickat på OK-knappen i MessageBoxen, göra andra val och få fram det nya resultatet i MessageBoxen osv.
Gör så här för att skapa programmet Bartender:

1. Skapa en Windows Forms Application och döp den till Bartender.

Form1:

Egenskap	Värde
Text	Var så god och välj
Size.Width	600
Size.Height	250

2. Hämta från Toolbox (All Windows Forms) en GroupBox-kontroll till formen:

groupBox1:

Egenskap	Värde
(Name)	grbDrink
Text	Dryck
Location.X	20
Location.Y	25
Size.Width	150
Size.Height	100

3. Hämta två CheckBox-kontroller till formen, placera dem i Dryck-gruppboxen och ändra följande värden:

checkBox1:

Egenskap	Värde
(Name)	chkVodka
Text	Vodka
Location.X	15
Location.Y	30

checkBox2:

Egenskap	Värde
(Name)	chkGin
Text	Gin
Location.X	15
Location.Y	60

4. Hämta en till GroupBox-kontroll till formen:

groupBox2:

Egenskap	Värde
(Name)	grbGlass
Text	Välj glas
Location.X	200
Location.Y	25
Size.Width	200
Size.Height	140

5. Hämta tre RadioButton-kontroller till formen, placera dem i Glas-gruppboxen och ändra följande värden:

radioButton1:

Egenskap	Värde
(Name)	optShotGlass
Text	Snapsglas
Location.X	15
Location.Y	30

radioButton2:

Egenskap	Värde
(Name)	optCocktailGlass
Text	Cocktailglas
Location.X	15
Location.Y	60

radioButton3:

Egenskap	Värde
(Name)	optVineGlass
Text	Vinglas
Location.X	15
Location.Y	90

6. Hämta en Button-kontroll till formen:

button1:

Egenskap	Värde
(Name)	btnServ
Text	Servera
Location.X	440
Location.Y	30
Size.Width	120
Size.Height	70

7. Hämta en till Button-kontroll till formen:

button2:

Egenskap	Värde
(Name)	btnFinish
Text	Avsluta
Location.X	460
Location.Y	135
Size.Width	75
Size.Height	45

Kod bakom Servera- och Avsluta-knappen

8. Dubbelklicka på Servera-knappen för att få upp Formens kod, klassen **`Form1`** med den nya händelsemetoden **`btnServ_Click()`**.

9. Lägg in i den nya händelsemetoden **`btnServ_Click()`** följande kod:

```
string output = "";
if (chkVodka.Checked && !chkGin.Checked)
   output = "Vodka serveras ";
if (chkGin.Checked && !chkVodka.Checked)
    output = "Gin serveras ";
if (chkVodka.Checked && chkGin.Checked)
    output = "Vodka och Gin serveras ";
if (optShotGlass.Checked)
    output += "i snapsglas.";
if (optCocktailGlass.Checked)
    output += "i cocktailglas.";
if (optVineGlass.Checked)
    output += "i vinglas.";
MessageBox.Show(output, "Bartender svarar:");
```

10. Dubbelklicka på Avsluta-knappen för att få upp Formens kod, klassen **`Form1`** med den nya händelsemetoden **`btnFinish_Click()`**.

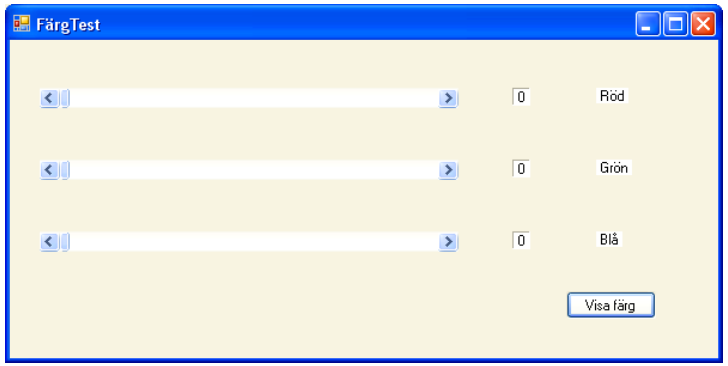
11. Lägg in i den nya händelsemetoden **`btnFinish_Click()`** följande kod:

```
Application.Exit();
```

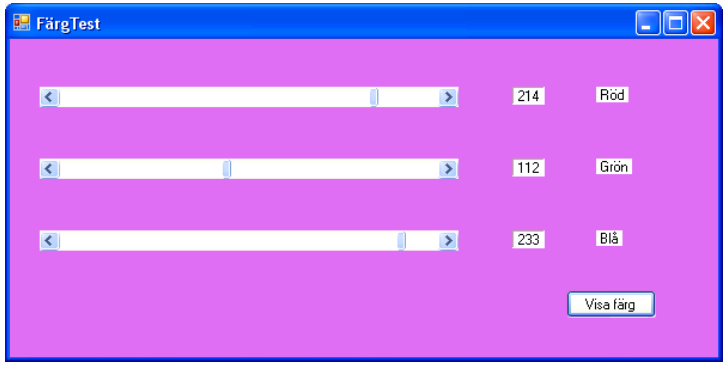
12. Kompilera och kör. Välj dryck, glas och klicka på Servera-knappen.

1.4 Färgtest med kontrollen HScrollBar

Här kommer vi att bekanta oss med Visual Studios kontroll HscrollBar där H står för *horizontal*. Programmet ColorTest demonstrerar denna kontroll. Låt oss först titta på dess grafiska gränssnitt:



Till vänster ser man tre HscrollBar-kontroller. Ordagant betyder scrollbar på svenska *rullningslist*, men vi föredrar det engelska originalet. Till höger om dem finns sex labels, två till varje scrollbar, dessutom en button. De första tre labels som på bilden står 0 på, visar resp. scrollbars värde som kan ändras när man kör programmet. Det gör man genom att med nedtryckt mus dra på scrollbarens reglage och ställa in ett önskat värde genom att släppa musen. Detta värde kommer då att visas på labeln (istället för 0) så snart man klickat på knappan Visa färg. Då kan det hela se ut så här:



De andra tre labels som det står Röd, Grön och Blå på, visar de färgkomponenter som bidrar till bakgrundsfärgen enligt RGB-färgsystemet. I exemplet ovan är bakgrundsfärgen en lilanuans – som dessvärre inte kan ses i svart-vit trycket – och sammansatt av 214 röda, 112 gröna och 233 blå andelar. Varje grundfärg i RGB-systemet kan bidra med 0-255 andelar till den sammansatta färgen. Genom en kombination av olika inställningar kan man få sammanlagt $256^3 = 16\,777\,216$ olika färgnuanser som allihopa är blandningar (i olika doser) av de grundfärgerna röd, grön och blå, därav namnet RGB.

Programmet ColorTest demonstrerar hur man med enkla medel – några kontroller bl.a. HScrollBar och lite kod – kan mixa, få fram och se alla möjliga RGB-färgerna.

Gör så här för att skapa programmet ColorTest:

1. Skapa en Windows Forms Application och döp det till ColorTest.

Form1:

Egenskap	Värde
Text	ColorTest
Size.Width	600
Size.Height	320

2. Hämta tre HScrollBar-kontroller från Toolbox (All Windows Forms) till formen:

hScrollBar1, ...2, ...3:

Egenskap	Värde
Location.X	25
Location.Y	40
Size.Width	350
Size.Height	17
Maximum	255
LargeChange	1

Egenskap	Värde
Location.X	25
Location.Y	100
Size.Width	350
Size.Height	17
Maximum	255
LargeChange	1

Egenskap	Värde
Location.X	25
Location.Y	160
Size.Width	350
Size.Height	17
Maximum	255
LargeChange	1

Kontrollen HScrollBar har två egenskaper Minimum och Maximum som representerar scrollbarens minsta och största värde. Minimum:s defaultvärde är 0. Vi ändrar inte det, eftersom vi vill ha intervallet [0, 255]. Däremot sätter vi värdet på Maximum i alla tre scrollbarer till 255, se tabellen nedan. En annan egenskap av kontrollen HScrollBar är LargeChange som är steget som scrollbarens värde ändras med när man klickar på de små pilarna på båda sidorna av scrollbaren. Vi sätter detta steg till 1.

3. Hämta tre Label-kontroller till formen och placera dem höger om HScrollBar-kontrollerna:

label1, ...2, ...3:

Egenskap	Värde
Text	0
Location.X	420
Location.Y	40
BorderStyle	Fixed3D
TextAlign	MiddleCenter
BackColor	White

Egenskap	Värde
Text	0
Location.X	420
Location.Y	100
BorderStyle	Fixed3D
TextAlign	MiddleCenter
BackColor	White

Egenskap	Värde
Text	0
Location.X	420
Location.Y	160
BorderStyle	Fixed3D
TextAlign	MiddleCenter
BackColor	White

4. Hämta ytterligare tre Label-kontroller till formen och placera längst till höger:

label4, ...5:

Egenskap	Värde
Text	Röd
Location.X	490
Location.Y	40
BackColor	White

Egenskap	Värde
Text	Grön
Location.X	490
Location.Y	100
BackColor	White

label6:

Egenskap	Värde
Text	Blå
Location.X	490
Location.Y	160
BackColor	White

5. Hämta en Button-kontroll till formen som vi i beskrivningen refererar till som Visa-knappen:

button1:

Egenskap	Värde
(Name)	btnShow
Text	Visa färg
Size.X	90
Size.Y	40
Location.X	465
Location.Y	210

Kod bakom Visa färg-knappen

6. Dubbelklicka på Visa-knappen för att få upp Formens kod, klassen **`Form1`** med den nya händelsemetoden **`btnShow_Click()`**.

7. Lägg in i den nya händelsemetoden **`btnShow_Click()`** följande kod:

```
BackColor = Color.FromArgb(
            hScrollBar1.Value, hScrollBar2.Value,
                               hScrollBar3.Value);
label1.Text = Convert.ToString(hScrollBar1.Value);
label2.Text = Convert.ToString(hScrollBar2.Value);
label3.Text = Convert.ToString(hScrollBar3.Value);
```

8. Kompilera och kör. Dra scrollarna och klicka på Visa-knappen.

1.5 Undantagshantering

Övningarna 1.5 och 1.6 (sid 63) kräver undantagshantering. *Undantag* (eng. *exception*) betyder i programmering fel, närmare bestämt *exekveringsfel* som uppstår när datorns processor inte kan utföra programmets instruktioner även om syntaxen är korrekt. Exekveringsfel syns inte vid kompilering. De leder ”bara” till att i bästa fall programmet och i sämsta fall datorn kraschar. I regel är orsaken okänd och inte lätt att spåra just vid exekveringstillfället. Däremot borde man kunna förutse dem när man skriver kod. Till god programmeringsstil hör att man tar hand om ”farlig kod” redan när man programmerar. Det gäller förstås att förutse i vilka situationer fel *kan* inträffa vid exekveringen. I så fall borde man bygga in en felhantering i koden. Alla moderna programmeringsspråk ställer verktyg till förfogande för felhantering som kallas *undantagshantering* (eng. *exception handling*). Detta avsnitt introducerar bara de mest elementära begreppen och metoderna för undantagshantering i C#.

Automatiskt genererade undantag

I ett av våra program **`SimpleIf`** *(Progr1, 5.2)* hade vi redan skrivit en egen felhantering. Programmet läste in två tal och dividerade dem med varandra. Men koden tillät division endast om det andra talet inte var `0`. Detta för att förhindra den matematiskt odefinierade divisionen med `0`. Inmatning av `0` till det andra talet genererade ”felmeddelandet”:

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

Programmeringstekniskt löste vi problemet då med två enkla **`if`**-satser. Nu ska vi försöka att göra det med de verktyg för undantagshantering som är inbyggda i C#.

```
// TryCatchTest.cs
// Förhindrar programavbrott med ett try catch-block
using System;

class TryCatchTest
{
  static void Main()
  {
    int no1 = 8, no2 = 0, div;

    try                          // try catch-blocket
    {
    div = no1 / no2;
    }
    catch
    {
      Console.WriteLine("\n\t OBS! Du försökte dividera med 0.” +
                        "\n\t Det går inte att dividera med 0.");
    }
      Console.WriteLine("\n\t Här fortsätter programmet! \n");
   }
}
```

Det reserverade ordet **try** säger: ”Försök att …” dvs försök att utföra det block av satser som följer, i det här fallet försök att utföra satsen **div = no1 / no2;** som innebär att dela **8** med **0**. Om det uppstår något fel (undantag) ”fånga upp” – i koden **catch** – felet genom att utföra det block av satser som följer efter **catch**. Man har friheten att skriva i **catch**-blocket allt man önskar att det ska ske om **try**-blocket ”kastar ett undantag” dvs ger upphov till ett fel. Därför kallas hela konstruktionen **try catch**-blocket och är undantagshanteringens grundkoncept.

För att förenkla testet har vi i programmet ovan tilldelat **0** direkt till **no2** för att provocera fram undantaget **DivideByZeroException** som är ett fördefinierat undantag och samtidigt en subklass med samma namn i klassen **Exception**. Detta undantag genereras automatiskt av koden **no1/no2** när **no2** har värdet **0**. Hade vi inte hanterat detta undantag genom att placera koden i **try**-blocket och fånga upp det i **catch**-blocket, hade vi fått följande felmeddelande vid exekvering av programmet **TryCatchTest**:

```
Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.
   at TryCatchTest.Main() in c:\C#\MyProject\TryCatchTest.cs:line 14
```

Testa gärna detta genom att kommentera bort hela **try catch**-blocket men behålla satserna **div = no1/no2;** och **Console.WriteLine("\n\tHär fortsätter programmet! \n");**. Samtidigt med felmeddelandet ovan avbryts programkörningen abrupt. Resten av programmet exekveras inte. Hade den kod som kastar undantaget stått i början av ett längre program hade stora mängder kod inte exekverats.

Om vi däremot hanterar undantaget som i **TryCatchTest** sker inget oväntat programavbrott. Istället exekveras koden i **catch**-blocket. Sedan fortsätter programflödet efter **catch**-blocket, resten av koden exekveras och programmet slutförs på ett regulärt sätt. Körresultatet av programmet **TryCatchTest** visar detta:

```
OBS! Du försökte dividera med 0.
Det går inte att dividera med 0.

Här fortsätter programmet!
```

Observera att programflödet inte återgår till den punkt tillbaka där undantaget kastades i **try**-blocket utan fortsätter linjärt, dvs efter **catch**-blocket. Så kod som står efter den ”farliga koden” i **try**-blocket exekveras endast om inget undantag inträffar. Testa gärna själv genom att i programmet **TryCatchTest** lägga in någon utskriftssats i slutet av **try**-blocket. Den sats kommer inte att utföras eftersom **no1/no2** genererar undantag.

Det finns en uppsjö av automatiskt genererade undantag i C# som är fördefinierade i subklasser till klassen **Exception** som finns i namnutrymmet **System**. Varje gång ett undantag inträffar skapas ett objekt av en sådan klass där all information om undantaget lagras. Andra exempel på automatiskt genererade undantag är **IndexOutOfRangeException** som inträffar när man överskrider en arrays gränser och **NullReferenceException** som uppstår när man använder en referens som har värdet **null** dvs inte pekar på något objekt.

Egengenererade undantag

Undantaget **`DivideByZeroException`** var i programmet **`TryCatchTest`** (sid 28) automatiskt genererad, förorsakat av koden **`no1/no2`** och av att variabeln **`no2`** hade värdet **`0`**. Men det finns i C# också möjligheten att programmeraren själv genererar ett undantag vilket ger oss friheten att kontrollera våra program med avseende på tillförlitlighet och stabilitet av kod. Detta kan man göra bl.a. med det reserverade ordet **`throw`** (eng. att kasta). Att kasta ett undantag betyder att generera ett sådant, vilket man kan göra genom att sätta **`throw`** framför ett objekt av någon undantagsklass. Följande program demonstrerar detta:

```
// ThrowTest.cs
// Kastar ett undantag med throw och hanterar det med try catch
using System;

class ThrowTest
{
   static double SafeDiv(double no1, double no2)        // Metod
   {
      if (no2 == 0)
         throw new DivideByZeroException(); // Undantag kastas
      else                                  // Objekt skapas
         return no1 / no2;
   }

   static void Main()
   {
      try                                   // Undantag hanteras
      {
         Console.WriteLine(SafeDiv(8, 0));             // Anrop
      }
      catch(DivideByZeroException e)        // catch + parameter
      {
         Console.WriteLine(e.ToString());   // Undantag skrivs ut
      }
   }
}
```

`throw`-*satsen*

`new DivideByZeroException()` är ett objekt av typ **`DivideByZeroException`**. Genom att sätta **`throw`** framför det genereras (kastas) ett sådant undantag:

```
throw new DivideByZeroException();
```

Denna sats ersätter koden **`no1/no2`** som i förra avsnitt förorsakade det automatiskt genererade undantaget. Därför är denna kod flyttad efter **`else`** och utförs därmed endast om **`no2`** inte är lika med **`0`**. Satsen är inbyggd i metoden **`SafeDiv()`** som anropas i

try-blocket. Därmed genereras undantaget där, vilket länkar programflödet till **catch**-blocket. Huvudet till **catch**-blocket ser här annorlunda ut:

```
catch(DivideByZeroException e)
```

Det ser ut som en metod med en parameterlista i vilken en referens **e** definieras till det ovan skapade undantagsobjektet av typ **DivideByZeroException**. Vi har alltså att göra med en annan variant av **catch** jämfört med programmet **TryCatchTest** (sid 28) där **catch** saknade parameterlista. Med hjälp av referensen **e** som pekar på det kastade undantagsobjektet kan vi nu i **catch**-blocket anropa objektets **ToString()**-metod:

```
Console.WriteLine(e.ToString());
```

Detta anrop resulterar i följande utskrift av programmet **ThrowTest** :

```
System.DivideByZeroException: Attempted to divide by zero.
   at ThrowTest.SafeDiv(Double no1, Double no2) in
c:\C#\MyProject\ThrowTest.cs:line 10
   at ThrowTest.Main() in c:\C#\MyProject\ThrowTest.cs:line 19
```

Observera att detta inte är ett felmeddelande, därför att vi har ju hanterat undantaget **DivideByZeroException** i **try catch**-blocket och skrivit ut dess **ToString()**-metod. **ToString()** är en strängrepresentationsmetod definierad i en superklass som ärvs av alla fördefinierade klasser, så även av klassen **DivideByZeroException**. Därför kan vi använda den med referensen **e** som pekar på det kastade undantagsobjektet av denna klass. Metoden **ToString()** innehåller objektets fullständiga information i strängform. Genom att anropa den i utskriftssatsen ser vi denna information. Den anger först sin källa: **System.DivideByZeroException**. Sedan talar den om vilken typ av undantag det rör sig om: **Attempted to divide by zero**. Resten av informationen handlar om var exakt i programmet undantaget inträffade.

Samma information som vi får med **e.ToString()** ges vidare till det felmeddelande som automatiskt skrivs ut om vi inte hanterar undantaget. Den enda skillnaden är att det hela inleds då med att det är ett ohanterat undantag:

```
Unhandled Exception: ...
```

Då hade detta varit ett verkligt felmeddelande.

Man kan ju undra vilken praktisk relevans programmet **ThrowTest** har och varför och i vilka situationer man använder **throw**-satsen. När ska man låta C# upptäcka möjliga fel och generera undantag automatiskt och när ska vi skriva kod för att själva kasta och hantera undantag? Programmet **ThrowTest** har endast pedagogisk relevans, dvs att ge en första introduktion till de elementära grundbegreppen och metoderna inom undantagshantering. I övningarna 1.5 och 1.6 på nästa sida hittar du ytterligare en användning av undantagshantering.

1.6 ListBoxar

1. Skapa en Windows Forms Application och döp det till ListBoxes.

Form1:

Egenskap	Värde
Text	ListboxTest
Size.Width	600
Size.Height	375

2. Hämta en ListBox-kontroll till formen.

listBox1:

Egenskap	Värde
Location.X	40
Location.Y	40
Size.Width	175
Size.Height	224

3. Högerklicka på listBox1, kopiera och klistra in i formen, för att få en till ListBox-kontroll i samma storlek. Ändra Location:

listBox2:

Egenskap	Värde
Location.X	370
Location.Y	40

4. Markera listBox1, klicka på *Smart Tag* (lilla pilen), välj Edit Items, skriv in följande texter i dialogrutan String Collection Editor som dyker upp – en rad i taget och klicka på OK:

Stockholm
London
Paris
Amsterdam
New York
Wien
Moskva

5. Hämta en Button-kontroll till formen.

button1:

Egenskap	Värde
Location.X	255
Location.Y	110
Size	75; 35
Text	----->

6. Högerklicka på I button1, kopiera och klistra in i formen, för att få en till button i samma storlek. Ändra Text och Location:

button2:

Egenskap	Värde
Location.X	255
Location.Y	160
Text	<-----

Projektets kod

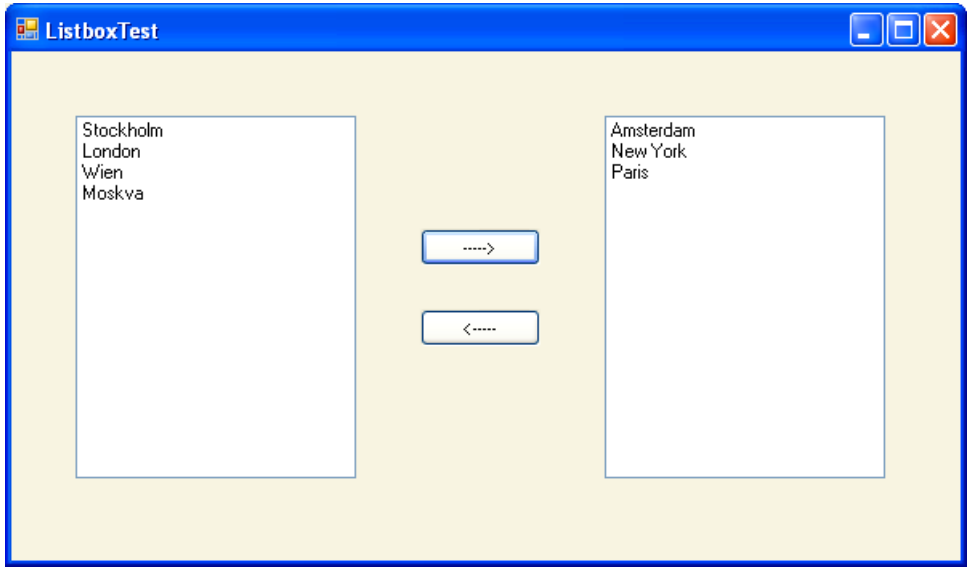
7. Dubbelklicka på button1-kontrollen och skriv in följande:

```
private void Button1_Click(object sender, EventArgs e)
{
    listBox2.Items.Add(listBox1.Text);
    listBox1.Items.Remove(listBox1.Text);
}
```

8. Dubbelklicka på button2-kontrollen och skriv in följande:

```
private void Button2_Click(object sender, EventArgs e)
{
    listBox1.Items.Add(listBox2.Text);
    listBox2.Items.Remove(listBox2.Text);
}
```

9. Kompilera och kör. Så här kan det se ut när man kör programmet **`ListBoxes`**:



1.7 Gränssnitt mot kalendern

Ett grafiskt gränssnitt ska låta användaren välja ett beställningsdatum och skriva ut ett leveransdatum enligt följande regler:

- Leveransdatum får inte vara före beställningsdatum.
- Leveransdatum ska i regel ligga 2 dagar efter beställningsdatum.
- Det ska tas hänsyn till att söndagar inte kan levereras, dvs:
- Ligger en söndag mellan beställnigs- och leveransdatum, blir leveranstiden 3 dagar.

1. Skapa en Windows Forms Application och döp det till DeliveryDate.

Form1:

Egenskap	Värde
Text	Leveransdatum
Size	410; 430

2. Hämta en Label-kontroll till formen och döp den till orderLabel.

label1:

Egenskap	Värde
(Name)	orderLabel
Text	Beställningsdatum:
Location	45; 45

3. Hämta en DateTimePricker-kontroll till formen

dateTimePicker1:

Egenskap	Värde
Location	45; 90
Size	300; 26

4. Hämta en andra Label-kontroll till formen och döp den till outputLabel.

label2:

Egenskap	Värde
(Name)	outputLabel
AutoSize	False
Size	300; 45
Location	45; 230
BorderStyle	FixedSingle
Text	
TextAlign	MiddleCenter

5. Hämta en tredje Label-kontroll till formen och döp den till delivLabel.

label3:

Egenskap	Värde
(Name)	delivLabel
Text	Leveransdatum:
Location	45; 185

Projektets kod

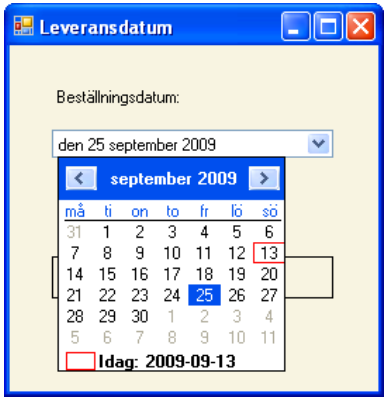
6. Dubbelklicka på dateTimePicker1-kontrollen och skriv in följande:

```
private void dateTimePicker1_ValueChanged
                              (object sender, EventArgs e)
{
   DateTime orderDate = dateTimePicker1.Value;

   if (orderDate.DayOfWeek == DayOfWeek.Friday   ||
       orderDate.DayOfWeek == DayOfWeek.Saturday ||
       orderDate.DayOfWeek == DayOfWeek.Sunday)

       outputLabel.Text =
              orderDate.AddDays(3).ToLongDateString();
   else
       outputLabel.Text =
              orderDate.AddDays(2).ToLongDateString();
}
```

7. Dubbelklicka på formen Form1 och skriv in följande:

```
private void Form1_Load(object sender, EventArgs e)
{
   dateTimePicker1.MinDate = DateTime.Today;
   dateTimePicker1.MaxDate = DateTime.Today.AddYears(1);
}
```

8. Kompilera. Så här ser det ut när man kör programmet **DeliveryDate**:



1.8 En räntekalkylator med multiline TextBox

1. Skapa en Windows Forms Application och döp det till TaxCalculator.

Form1:

Egenskap	Värde
Text	RänteKalkylator
Size.Width	430
Size.Height	430

2. Hämta en Label-kontroll till formen och ändra värden:

label1:

Egenskap	Värde
Text	Kapital:
Location.X	17
Location.Y	30

3. Hämta en TextBox-kontroll till formen ... :

textBox1:

Egenskap	Värde
(Name)	tbCapital
Location.X	120
Location.Y	27
Size.Width	160
Size.Height	26
TextAlign	Right

4. Hämta en Label-kontroll till formen:

label2:

Egenskap	Värde
Text	Räntesats:
Location.X	17
Location.Y	80

5. Hämta en TextBox-kontroll till formen:

textBox2:

Egenskap	Värde
(Name)	tbTaxRate
Location.X	120
Location.Y	77
Size.Width	160
Size.Height	26
TextAlign	Right

6. Hämta en Button-kontroll till formen:

button1:

Egenskap	Värde
(Name)	btnCompute
Text	Beräkna
Location.X	300
Location.Y	25
Size.Width	90
Size.Height	30

7. Hämta en Label-kontroll till formen:

label3:

Egenskap	Värde
Text	Antal år:
Location.X	17
Location.Y	130

8. Hämta en NumericUpDown-kontroll till formen:

numericUpDown1:

Egenskap	Värde
(Name)	numUpDownYear
Location.X	120
Location.Y	127
Size.Width	160
Size.Height	26
Minimum	1
Maximum	20
ReadOnly	True
TextAlign	Right

9. Hämta en Label-kontroll till formen:

label4:

Egenskap	Värde
Text	Årliga saldon:
Location.X	17
Location.Y	175

10. Hämta en TextBox-kontroll till formen:

textBox3:

Egenskap	Värde
(Name)	tbDisplay
MultiLine	True
Location.X	20

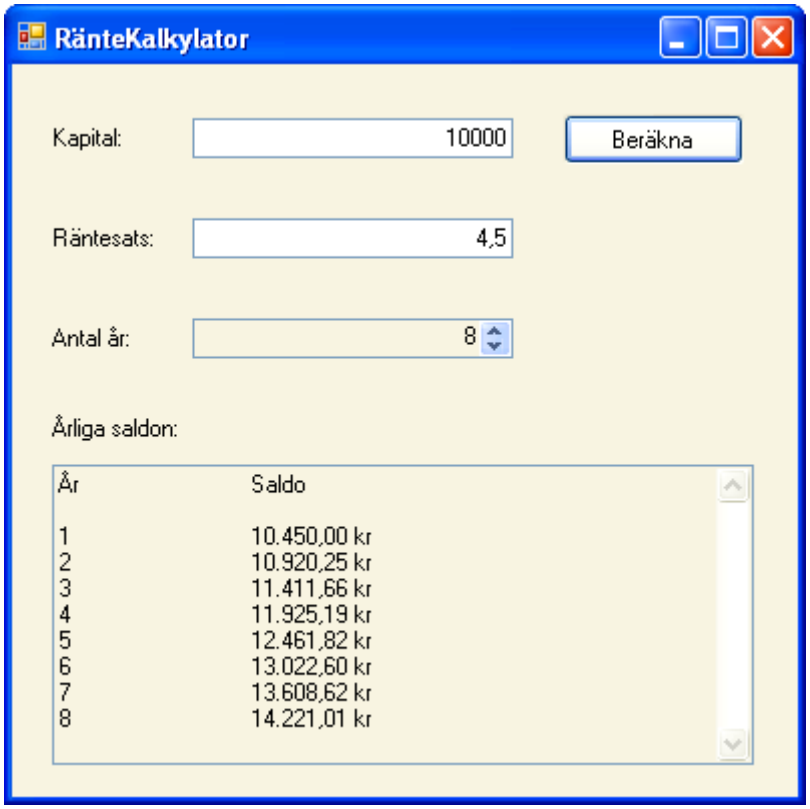
Location.Y	200
Size.Width	350
Size.Height	150
ReadOnly	True
Scrollbars	Vertical

11. Dubbelklicka på Beräkna-knappen för att få upp Formens kod, klassen **`Form1`** med den nya händelsemetoden **`btnCompute_Click()`**. Lägg in kod enligt följande:

```
// Form1.cs
// Beräknar räntan av kapital efter n år enligt formeln:
// saldo = kapital * FF^n där FF = (1 + räntesats/100)
// Demonstrerar kontrollerna NumericUpDown och TextBox (MultiLine)
// samt formaterad utskrift av decimalttal: Valutaformat
using System;
using System.Windows.Forms;

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

    private void btnCompute_Click(object sender, EventArgs e)
    {
      double balance;                    // Inläsning:
      double capital = Convert.ToDouble(tbCapital.Text);
      double taxRate = Convert.ToDouble(tbTaxRate.Text);
      int years      = Convert.ToInt32(numUpDownYear.Value);

      double FF = 1 + taxRate / 100;     // Förändringsfaktorn
      string output = "År\t\tSaldo\r\n\r\n";
                                         // Utskriftsvariabel
      for (int n = 1; n <= years; n++)
      {
        balance = capital * (Math.Pow(FF, n));
        output += n + "\t\t" +
                  string.Format("{0:C}", balance) + "\r\n";
      }                                  // Valutaformat:
                                         // C = Currency

      tbDisplay.Text = output;           // Akkumulerad utskrift
    }                                    // dumpas till multi-
  }                                      // line textbox
}
```

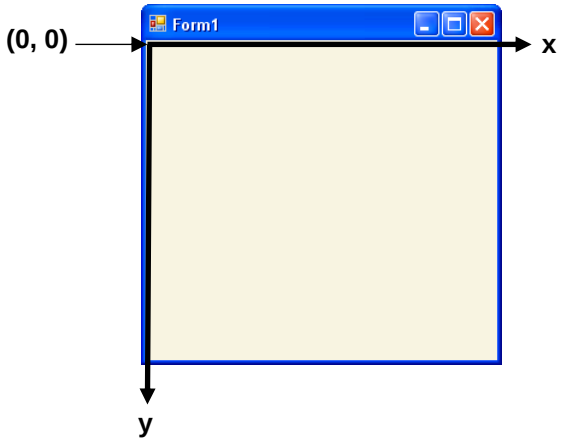
12. Kompilera och kör.

Så här kan det se ut när man kör programmet TaxCalculator:



1.9 Geometriska figurer

För att kunna rita geometriska figurer och placera dem behöver vi ange bl.a. deras storlek och position, vilket förutsätter ett koordinatsystem på den grafiska ritytan. Ett sådant koordinatsystem är automatiskt definierat i alla fönster vi får fram i Visual Studio, där origo dvs positionen (0, 0) är placerad i fönstrets vänstra övre hörn. OBS! formens rubrik ligger utanför. x-koordinaten växer i horisontell led åt höger och y-koordinaten i vertikal led nedåt. Tillämpar vi detta default koordinatsystem t.ex. på formfönstret, kan vi föreställa oss följande situation:



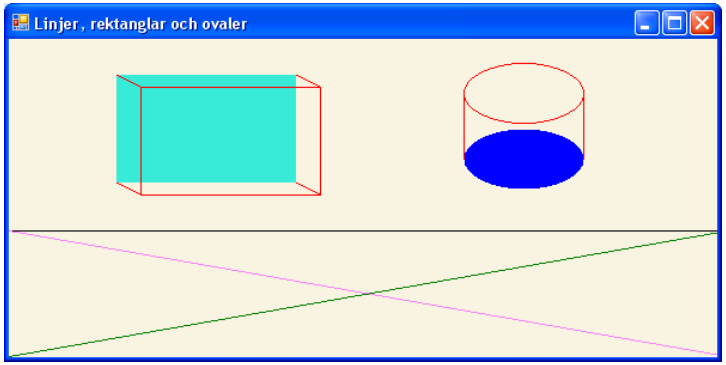
Denna bild borde man ha i minnet när man arbetar med koordinater i en C# Windows Application. Som man ser befinner sig alla positioner på formens rena rityta som är grå.

Man kan undra vad koordinatsystemets *enhet* är. Vi har ju inte satt någon skala på axlarna – och detta av goda skäl: Enheten på en grafisk yta är alltid automatiskt en s.k. *pixel* som står för *picture element*. En pixel är en digital bilds minsta komponent – datorgrafikens atom så att säga. Som en bildpunkt med en viss färg och en placering är storleken beroende av den aktuella tekniska utrustningen som visar bilden – hos oss bildskärmen och dess upplösning. Vill vi placera en punkt i det default koordinatsystemet ovan anges punktens x-koordinat som antalet pixlar som den är borta från formens vänstra kant. Punktens y-koordinat anges som antalet pixlar som den är borta från formens övre kant.

Självklart kan man, om man vill, även skapa sitt eget koordinatsystem som man är van vid från matematiken, med origo i mitten osv. Men vi kommer i våra program att anpassa oss till detta grafiska koordinatsystem som är standard i all datorgrafik. Därmed slipper vi besväret att skriva kod som räknar om uppgifterna i vårt koordinatsystem till default koordinatsystemet. Priset vi måste betala för denna förenkling är: Det vi då måste tänka på när vi skriver kod är att enheten är pixlar, att det därför inte kan finnas några negativa koordinater och att y-koordinaten växer nedåt och inte uppåt. Man vänjer sig ganska fort till detta nya tankesätt. Ytterligare ett starkt skäl till att anpassa sig till det befintliga och inte införa ett nytt eget koordinatsystem, är att alla ritmetoder i C# biblio-

teket är formulerade i termer av default koordinatsystemet. Skriver man ett program där man blandar egen kod med anrop av biblioteksmetoder – och det gör ju nästan alla program – är det en stor fördel att tillämpa samma system.

Programmet Draw använder sig av ett antal biblioteksmetoder för att rita linjer, rektanglar och ovaler. Vi vill t.ex. åstadkomma följande bild:



Gör så här för att skapa programmet Draw:

1. Skapa en Windows Forms Application och döp det till Draw.

   Form1:

Egenskap	Värde
Text	Linjer, rektanglar och ovaler
Size.Width	920
Size.Height	465

2. Gå till Solution Explorer, högerklicka på Form1.cs och välj View Code. Ersätt hela koden i filen Form1.cs med följande:

```
// Projekt Draw, filen Form1.cs
// Demonstrerar ritning av linjer, rektanglar och ovaler
// Metoden OnPaint() ärvs från basklassen Form och överskuggas
using System.Drawing;
using System.Windows.Forms;

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

```
      protected override void OnPaint(PaintEventArgs e)
      {
         Graphics g = e.Graphics;

         Pen pen = new Pen(Color.Black);
         g.DrawLine(pen, 0, 160, 600, 160);
         pen = new Pen(Color.Green);
         g.DrawLine(pen, 0, 265, 600, 160);
         pen = new Pen(Color.Violet);
         g.DrawLine(pen, 0, 160, 600, 265);

         SolidBrush brush = new SolidBrush(Color.Turquoise);
         g.FillRectangle(brush, 90, 30, 150, 90);

         pen = new Pen(Color.Red);
         g.DrawLine(pen,  90,  30, 110,  40);
         g.DrawLine(pen,  90, 120, 110, 130);
         g.DrawLine(pen, 240,  30, 260,  40);
         g.DrawLine(pen, 240, 120, 260, 130);

         g.DrawRectangle(pen, 110, 40, 150, 90);

         brush.Color = Color.Blue;
         g.FillEllipse(brush, 380, 75, 100, 50);

         g.DrawLine(pen, 380, 45, 380, 100);
         g.DrawLine(pen, 480, 45, 480, 100);

         g.DrawEllipse(pen, 380, 20, 100, 50);
      }
   }
}
```

3. Kompilera och kör.

Metoden `OnPaint()`

Nästan hela koden till detta program står i metoden **`OnPaint()`**. Ordet **`override`** i metodens huvud betyder att vi definierar om metoden **`OnPaint()`** och att denna omdefinition *överskuggar* (eng. *override*) den ursprungliga definitionen av metoden i klassen **`Form`** – en klass som vi ärver genom att i klasshuvudet skriva **`public partial class Form1 : Form`**. Dvs all kod som finns fördefinierad i klassen **`Form`** finns till vårt förfogande i klassen **`Form1`** som vi skriver, bl.a. metoden **`OnPaint()`**. Vi tar över metodens huvud och skriver vår egen kropp till den. Observera att all ritning av geometriska figurer i metoden **`OnPaint()`** endast är möjlig om det i början av metoden skapas ett **`Graphics`**-*objekt* med *referensen* **`g`** som i fortsättninngen refererar till objektet: **`Graphics g = e.Graphics;`**. Detta gäller även för nästa programs **`OnPaint()`**-metod i nästa avsnitt där vi fortsätter att rita. Alla dessa begrepp *överskuggning*, *arv*, *objekt*, *referens*, **`override`** och andra, är objektorienterade programmeringens termer som kommer att i detalj behandlas i bokens kapitel 2 (sid 69).

1.10 Bågar och vinklar

1. Skapa en Windows Forms Application och döp det till Arcs.

Form1:

Egenskap	Värde
Text	Bågar och vinklar
Size	460; 465

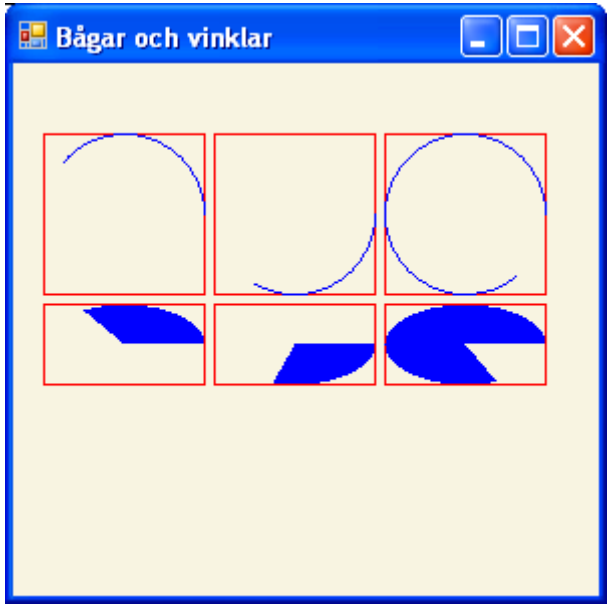
2. Gå till Solution Explorer, högerklicka på Form1.cs och välj View Code. Ersätt hela koden i filen Form1.cs med följande:

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

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

      protected override void OnPaint(PaintEventArgs e)
      {
         Graphics g = e.Graphics;
         Rectangle r1 = new Rectangle(15, 35, 80, 80);

         SolidBrush brush1 = new SolidBrush(Color.Red);
         Pen        pen1   = new Pen(brush1, 1);

         SolidBrush brush2 = new SolidBrush(Color.Blue);
         Pen        pen2   = new Pen(brush2, 1);

         g.DrawRectangle(pen1, r1);
         g.DrawArc(pen2, r1, 0, -140);

         r1.Location = new Point(100, 35);
         g.DrawRectangle(pen1, r1);
         g.DrawArc(pen2, r1, 0, 120);

         r1.Location = new Point(185, 35);
         g.DrawRectangle(pen1, r1);
         g.DrawArc(pen2, r1, 0, -310);
        r1.Location = new Point(15, 120);
         r1.Size = new Size(80, 40);
```

```
            g.DrawRectangle(pen1, r1);
            g.FillPie(brush2, r1, 0, -140);

            r1.Location = new Point(100, 120);
            g.DrawRectangle(pen1, r1);
            g.FillPie(brush2, r1, 0, 120);

            r1.Location = new Point(185, 120);
            g.DrawRectangle(pen1, r1);
            g.FillPie(brush2, r1, 0, -310);
        }
    }
}
```

3. Kompilera och kör.

Så här ser det ut när man kör programmet **`Arcs`**:



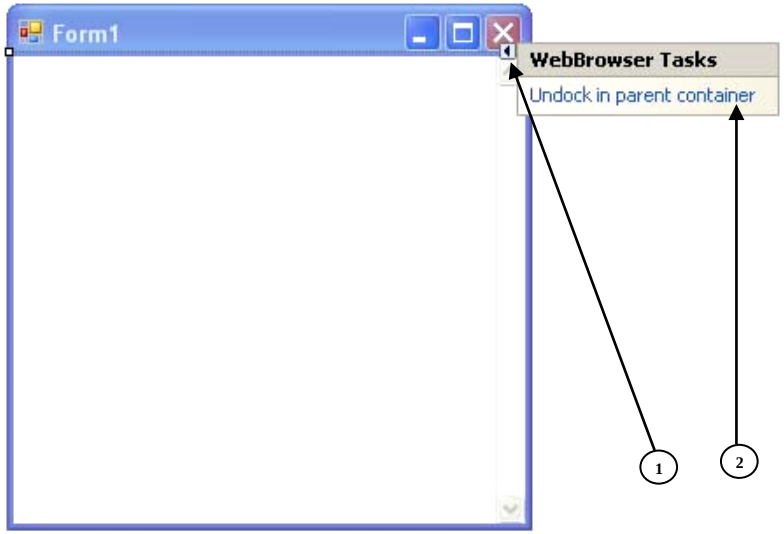
1.11 En egen webbläsare

I detta avsnitt vill vi utveckla en enkel webbläsare med möjligheten att koppla upp sig till Internet och visa en webbsida samt navigera på den – en typisk grafisk applikation, som vi i nästa avsnitt ska utvidga med ytterligare funktionaliteter som t.ex. menyer, undermenyer osv. Men just nu ska den räcka till att kunna skriva en webbadress (URL) i ett textfält och klicka på en knapp för att komma ut på nätet till den angivna adressen. Själva webbsidan behöver ett lite större fönster för att kunna visas. Om vi till en början nöjer oss med dessa få ingredienser borde vi klara oss med följande kontroller som vi ska placera på vårt formfönster:

1. En WebBrowser som visar webbsidan
2. En TextBox för att skriva webbadressen i
3. En Button som vi klickar på för att köra igång

Faktiskt finns det i Visual Studios Toolbox en kontroll som heter WebBrowser och som bildar grunden till denna applikation – ett gränssnitt mot Internet.

Vi skapar först ett nytt projekt av typ Windows Forms Application – så som vi gick igenom i de föregående avsnitten – och döpar det till, säg MyFirstBrowser. Sedan hämtar vi kontrollen WebBrowser till formfönstret genom att dubbelklicka på den. Denna finns i Toolbox under Common Controls som allra sist. OBS! Till skillnad från andra kontroller kommer denna kontroll *inte* att lägga sig i formfönstrets övre vänstra hörn, utan den kommer att sträcka sig över formens hela lediga utrymme, så att man i början inte ens märker att den kommit till formen. Tittar man däremot noga, kan man se att det ligger ett vitt skikt över formens ljusgrå yta och täcker hela formen (utom rubriken). Skillnaden mellan ljusgrå (förr) och vit (nu) är en indikation på förändringen. Det vita skiktet är den nya WebBrowser-kontroll som vi just hämtade och la i formen. En annan indikation är den lilla



triangelformiga pil som (på bilden ovan) pil nr 1 pekar på – kallad *Smart Tag*. Klicka på denna *Smart Tag* för att få fram textrutorna till höger. Klicka sedan på länken Undock in parent container (pil nr 2). Detta kommer att lösa WebBrowser-kontrollen från formen. Då kan du för det första identifiera kontrollen bättre och för det andra placera den i formen var du vill. Självklart kan man även ändra storleken på den osv. Vi måste faktiskt förstora den, om vi vill visa webbsidor i den. Men för att förstora kontrollen måste vi först förstora dess behållare (container), formen. Vid det tillfället passar vi på att även få en lämpligare text på formens rubrik. Därför: Ändra egenskapernas värden hos formen Form1 enligt följande:

Form1:

Egenskap	Värde
Text	Min första webbläsare
Size.Width	1190
Size.Height	760

Observera att formen fortfarande har default *namnet* Form1. Den kommer att endast visa *texten* Min första webbläsare på sin rubrik när vi kör applikationen.

Ändra egenskapernas värden hos WebBrowser-kontrollen som by default har namnet webBrowser1 enligt följande:

webBrowser1:

Egenskap	Värde
(Name)	browserWindow
Size.Width	1150
Size.Height	620
Location.X	12
Location.Y	12

Här ändrar vi verkligen Name-egenskapen och dessutom storleken samt positionen av WebBrowser-kontrollen relativ till formen. Självklart är alla dessa värden – inklusive formens storlek i förra tabellen – relaterade till varandra med syftet att få en någorlunda bra layout på vår webbläsares grafiska utseende. Väljer du andra värden, får du anpassa dem till varandra layoutmässigt.

Markera formen, skapa en ny TextBox-kontroll och ändra dess värden enligt följande:

textBox1:

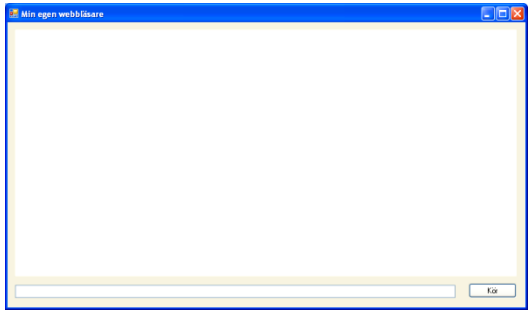
Egenskap	Värde
(Name)	tbURL
Location.X	12
Location.Y	650
Size.Width	1020
Size.Height	26

Slutligen behöver vi även en Button. Markera formen, hämta den från Toolbox och ändra de nedanstående egenskaperna till följande värden:

button1:

Egenskap	Värde
(Name)	btnGo
Location.X	1060
Location.Y	650
Size.width	80
Size.height	30
Text	Kör

Har du genomfört alla ovan beskrivna åtgärder, kommer din form i stort sett att ha följande utseende, här i körläge:



Det stora fönstret är WebBrowser-kontrollen som vi kallat browserWindow och som ska visa webbsidors innehåll. Det avlånga lilla fönstret nedan till vänster är TextBox-kontrollen tbURL, där man ska skriva en webbadress. Den är beredd att ta emot inmatning av text. Kör-knappen nedan till höger är Button-kontrollen btnGo som ska skicka förfrågan till den i tbURL angivna webbplatsen på Internet.

En första webbläsare

Att det inte händer något om du kompilerar och kör programmet och klickar på Kör-knappen – och inte heller om du först skriver en giltig webbadress i textboxen och sedan klickar på Kör-knappen – beror på att vi inte ännu lagt någon kod bakom knappen. Dvs vi har inte än skrivit någon händelsemetod som skulle anropas när händelsen ”Klicka på Kör-knappen” inträffar. Man kan också säga att det inte finns någon funktionalitet bakom Button-kontrollen btnGo. Och så är det med alla kontroller som skapas: De har ett antal egenskaper (datamedlemmar) med vissa defaultvärden som vi kan ändra. De har också ett antal händelsemetoder. Men av dessa metoder är endast huvudet fördefinierat (signaturen, sid 172, polymorfism, sid 113). Kroppen är tom, varför det inget händer, när de anropas vid en händelse, t.ex. en musklickning. Det är vi som måste skriva kod i dessa metoders kropp för att förse våra kontroller med den funktionalitet som är lämplig just för den aktuella applikationen. För att ge liv åt Kör-knappen i vår webbläsare, måste den (endast med huvudet) fördefinierade händelsemetoden

```
private void btnGo_Click(object sender, EventArgs e)
```

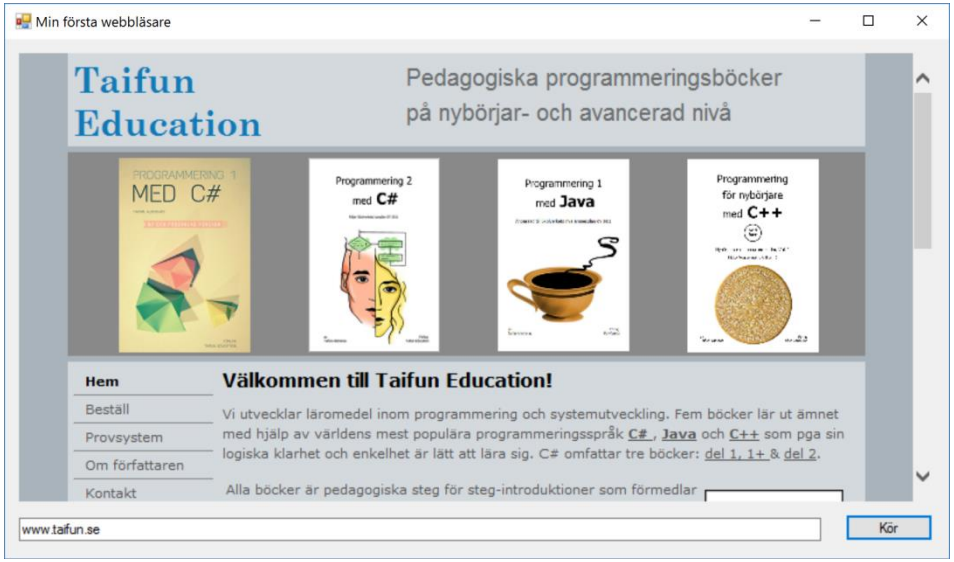
förses med kod i kroppen. För att åstadkomma detta, gör precis som i förra avsnitt:

Markera i designläge Kör-knappen btnGo och dubbelklicka på den. Du får denna kontrolls kod presenterad i editfönstret. Den består av en del av klassen **`Form1`**:s deklaration och lagras i filen Form1.cs. Markören står och blinkar i den tomma kroppen till händelsemetoden **`btnGo_Click()`**, redo att ta emot kod. Skriv där endast följande:

```
browserWindow.Navigate(tbURL.Text);
```

Satsen är ett anrop av metoden **`Navigate()`** tillhörande objektet **`browserWindow`** – vår WebBrowser-kontroll. I så fall måste **`Navigate()`** vara en fördefinierad metod i den klass som **`browserWindow`** är ett objekt av. Och den här gången är det en metod vars både huvud *och* kropp är förprogrammerade i klassen WebBrowser. Här ser man att kontrollerna i Visual Studios Toolbox är helt enkelt klasser som är skrivna och integrerade i miljön för att underlätta utvecklingsarbetet, för att vi inte skulle behöva att återuppfinna hjulet. Metoden **`Navigate()`** gör det egentliga jobbet i denna applikation, nämligen att gå ut på Internet och navigera oss fram till den server på nätet vars adress vi angivit, hämta HTML-filen som genererar webbsidan från servern och exekvera den på vår klientdator. För att kunna förse **`Navigate()`** med adressinformationen skickar vi i anropet ovan den aktuella parametern **`tbURL.Text`**, dvs datamedlemmen (**`Text`**-egenskapen) av **`TextBox`**-objektet **`tbURL`**. Dvs den sträng som vi skriver i textfältet, när programmet körs, blir värdet av **`tbURL`**:s **`Text`**-egenskap. Den har nämligen i designläge inget värde, vilket man kan övertyga sig av genom att titta i **`tbURL`**:s egenskapsfönster. Platsen där värdet ska stå är tom. Variabeln **`tbURL.Text`** blir tilldelad ett värde först när man exekverar. Värdet tas från textfältet vid inmatning och skickas, när Kör-knappen klickas, till metoden **`Navigate()`**.

När vår metods förfrågan har besvarats av servern på Internet, visas resultatet i applikationens browserWindow som fortfarande är en del av och integrerad i formfönstret. Storleken vi valt bibehålls under körningen. Är webbsiddan större än den förvalda storleken får WebBrowser-fönstret horisontella resp. vertikala scrollbars. Även om man maximerar formfönstret, blir det samma sak: Den valda storleken vid designläge kan inte ändras i körläge. Följande resultat visas när vi kompilerar och kör vår första webbläsare MyFirstBrowser, skriver en webbadress i textfältet nedan och klickar på Kör-knappen:



1.12 En mer utvecklad webbläsare

1. Skapa en Windows Forms Application och döp den till DevBrowser.

Form1:

Egenskap	Värde
Text	Utvecklad webbläsare
Size.Width	1500
Size.Height	1000

2. Hämta från Toolbox (Common Controls) en WebBrowser-kontroll till formen. Klicka på den lilla triangelformiga pilen (*Smart Tag*) i det högra övre hörnet av WebBrowser-kontrollen. Välj Undock in Parent container för att förminska den och lösa den från formen. Låt browserWindow vara Dock in Parent container.

webBrowser1:

Egenskap	Värde
(Name)	browserWindow
Size.Width	1500
Size.Height	1000
Location.X	0
Location.Y	40

Dialogrutan About Box

Vi vill nu för första gången vid sidan av Form1 skapa en ny, andra *form* i vårt projekt:

3. Högerklicka på projektnamnet DevBrowser i Solution Explorer. Välj Add → New Item... . Dialogrutan Add New Item dyker upp. Markera i den mellersta kolumnen About Box (Windows Forms) – den nya formens *typ*. Döp den nya formens fil i textfältet Name till AboutBox.cs och klicka på knappen Add längst ner till höger. Den nya formen About Box skapas.

4. Återgå till fliken bredvid: vår ursprungliga form Form1. Hämta från Toolbox (All Windows Forms) den nya kontrollen MenuStrip till formen. En tom menyradplats läggs till formen och täcker delvis över browserWindow. Samtidigt dyker upp en komponent av den längst ner till vänster i Visual Studio (inte i formen) som bär den nya kontrollens namn menuStrip1.

5. Markera MenuStrip-kontrollen, klicka på dess *Smart Tag*, en liten triangelförmig pil som syns invid det lilla röda krysset i det högra övre hörnet av formfönstret – men tillhörande MenuStrip-kontrollen. En pop up-ruta med rubriken MenuStrip Tasks kommer upp. Klicka på Insert Standard Items: En typisk Windows menyrad med menyer, undermenyer osv. läggs till MenuStrip.

6. Ta bort alla menyer utom Help-menyn genom att högerklicka på dem och välja Delete.

7. Markera Help-menyn. Välj undermenyn About... och dubbelklicka på den: Formens kod dyker upp med den nya händelsemetoden **`AboutToolStripMenuItem_Click()`**.

8. Ta bort alla onödiga **`using`**-satser från formens kod, dvs alla utom **`using System;`** och **`using System.Windows.Forms;`**. Testkör för att se att allt är ok. Stäng körningen.

9. Skriv i klassen **`Form1`**, ovanför raden **`public Form1()`**, koden:

```
AboutBox myAboutBox = new AboutBox();
```

   Därmed skapar du ett objekt av typ **`AboutBox`** och döper det till **`myAboutBox`**.

10. Lägg in i den nya händelsemetoden **`aboutToolStripMenuItem_Click()`** från punkt 8 följande anrop av det nya objektets metod **`ShowDialog()`**:

```
myAboutBox.ShowDialog();
```

11. Kompilera och kör. Klicka på Help-menyn samt på undermenyn About... för att se den nya AboutBox-formen om visar rubriken About DevBrowser. Klicka på OK och stäng körningen.

Dialogrutan Navigate

Här ska vi ersätta textfältet för webbadressen och knappen Kör som fanns i förra projektet MyFirstBrowser, med en dialogruta dvs en ny, *tredje form* av typ Windows Form.

12. Högerklicka på projektnamnet DevBrowser i Solution Explorer. Välj Add → New Item... . Välj i dialogrutan Add New Item... typen Form (Windows Forms). Döp den nya formens fil i textfältet Name till Navigate.cs och klicka på knappen Add längst ner till höger. Den nya formen Navigate skapas.

13. Sätt följande värden till den nya formen Navigate:s egenskaper:

Navigate:

Egenskap	Värde
FormBorderStyle	FixedDialog
MaximizeBox	False
MinimizeBox	False
ShowIcon	False
ShowInTaskbar	False
Size.Width	800
Size.Height	250
StartPosition	CenterParent

14. Hämta från Toolbox (All Windows Forms) en kontroll av typ TableLayoutPanel till den nya formen Navigate. Använd kontrollens *Smart Tag* (lilla pilen) och välj Remove Last Row för att få en rad och två kolumner i den nya kontrollen:

tableLayoutPanel1:

Egenskap	Värde
Size.Width	200
Size.Height	40
Location.X	570
Location.Y	130
Anchor	Bottom, Right

15. Lägg in en Button-kontroll i tableLayoutPanel1:s första kolumn:

button1:

Egenskap	Värde
(Name)	btnOK
Text	OK
Size.Width	75
Size.Height	35
Dialogresult	OK

16. Lägg in en Button-kontroll i tableLayoutPanel1:s andra kolumn:

button1:

Egenskap	Värde
(Name)	btnCancel
Text	Cancel
Size.Width	75
Size.Height	35
Dialogresult	Cancel

17. Återgå till Navigate-formen och lägg till följande två värden till egenskaperna:

Navigate:

Egenskap	Värde
AcceptButton	btnOK
CancelButton	btnCancel

Det kan vi göra först nu *efter* att knapparna skapats.

18. Hämta en Label-kontroll till Navigate-formen:

label1:

Egenskap	Värde
(Name)	lblURL
Location.X	30
Location.Y	20
Text	Mata in en Internet adress:

19. Hämta en TextBox-kontroll till Navigate-formen:

textBox1:

Egenskap	Värde
(Name)	txtURL
Location.X	30
Location.Y	50
Size.width	720
Size.height	26
AutoCompleteSource	AllUrl
AutoCompleteMode	SuggestAppend
Modifiers	Public

De Auto-egenskaperna gör att textfältet beter sig liknande adressfältet i Internet Explorer, t.ex. att den kommer ihåg och kompletterar adresser som man använt tidigare. Public gör att txtURL som finns i Navigate-formen (en klass för sig), är åtkomlig från formen Form1 (en annan klass) där Navigate-menyn kommer att läggas.

Menyn Navigate

Här ska vi koppla Navigate-formen till projektet DevBrowser.

20. Återgå till formen Form1 med rubriken Utvecklad webbläsare. Där finns redan en Help-meny. Klicka till höger om Help-menyn så att hela menyraden syns. Klicka i det lilla textfält som dyker upp och skriv &Navigate. En ny meny skapas med texten Navigate.

21. Flytta med musen den nya Navigate-menyn till vänster om Help-menyn.

22. Dubbelklicka på Navigate-menyn för att få upp Formens kod, klassen **`Form1`** med den nya händelsemetoden **`NavigateToolStripMenuItem_Click()`**.

23. Skriv i klassen **`Form1`**, ovanför raden **`AboutBox myAboutBox = ...`** , koden:

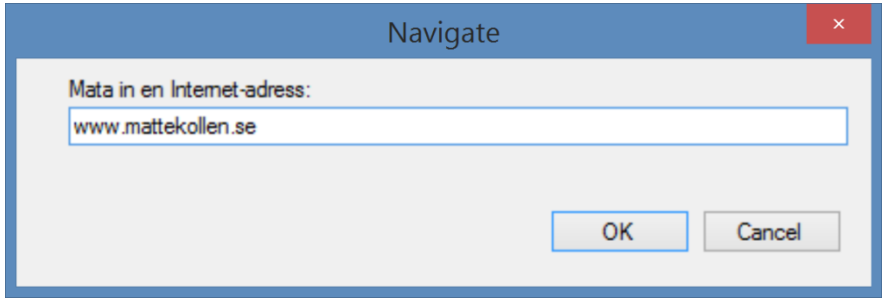
```
Navigate myNavigateBox = new Navigate();
```

Därmed skapar du ett objekt av typ **`Navigate`** och döper det till **`myNavigateBox`**.

24. Lägg in i den nya händelsemetoden **`navigateToolStripMenuItem_Click()`** följande kod:

```
if (myNavigateBox.ShowDialog() == DialogResult.OK)
{
    browserWindow.Navigate(myNavigateBox.txtURL.Text);
}
```

25. Kompilera och kör. Klickar du på Navigate-menyn visas Navigate-formen:



Skriv in en webbadress i textfältet till Navigate-formen ovan och klicka på OK för att få upp sidan i browserWindow. Så här kan det se ut:

Här följer sammanfattat all kod till detta projekt i filen Form1.cs:

```
// Form1.cs till projektet DevBrowser
// Utvecklad webbläsare med en Navigate- och en About-meny
// Tar in webbadressen via en dialogruta och
// kopplar upp sig till Internet
using System;
using System.Windows.Forms;

namespace DevBrowser
{
   public partial class Form1 : Form
   {
      Navigate myNavigateBox = new Navigate();   // Objekt av typ
                                                 // Navigate
      AboutBox myAboutBox    = new AboutBox();   // Objekt av typ
                                                 // AboutBox
      public Form1()
      {
         InitializeComponent();
      }

      private void aboutToolStripMenuItem_Click(
                                      object sender, EventArgs e)
      {
         myAboutBox.ShowDialog();
      }

      private void navigateToolStripMenuItem_Click(
                                      object sender, EventArgs e)
      {
         if (myNavigateBox.ShowDialog() == DialogResult.OK)
         {
            browserWindow.Navigate(myNavigateBox.txtURL.Text);
         }
      }
   }
}
```

1.13 Grafiskt gränssnitt med menyval

1. Skapa en Windows Forms Application och döp det till Menus.

Form1:

Egenskap	Värde
Text	Menyer
Size.Width	610
Size.Height	360

2. Hämta från Toolbox, All Windows Forms, en MenuStrip-kontroll till formen: En tom menyradplats lägger sig till formen direkt under rubriken. Längst ner till vänster i *component tray* dyker upp en annan del av den nya kontrollen med namnet menuStrip1. Den kan användas för att synliggöra MenuStrip-kontrollen om den försvinner, t.ex. när man (av misstag) markerat formen

3. En annan metod att skapa menyer än den vi lärde oss i projektet DevBrowser, är följande: Gå in med musen till textfältet Type Here som dyker upp på den nya kontrollen menuStrip1. Om du inte ser den, klicka på den tomma menyradplatsen längst till höger. Klicka i textfältet Type Here och skriv där File och tryck på Enter för att skapa en File-meny.

4. Markera menyradplatsen och skriv i textfältet Type Here som dyker upp till höger om File-menyn, Format. Tryck på Enter för att skapa en Format -meny.

Att skapa under- och under-undermenyer

5. Markera File-menyn och skriv i textfältet Type Here som dyker upp under den (OBS! inte bredvid den) About och tryck på Enter. Skriv i det textfält som dyker upp direkt under About-textfältet, Exit och tryck på Enter (upprepas kanske inte alltid explicit i fortsättningen). Så har du skapat två undermenyer under File-menyn.

6. Gör samma sak under Format-menyn. Markera den och skriv i textfältet Type Here som dyker upp under den (OBS! inte bredvid den) Color. Skriv i det textfält som dyker upp direkt under Color-textfältet, Font.

7. Gå till Format-menyn och markera undermenyn Color i den. Skriv i textfältet Type Here som dyker upp till höger om den, Black. Skriv i det textfält som dyker upp direkt under Black-textfältet, Blue. Skapa på samma sätt ytterligare två under-undermenyer under Format-Color-menyn, nämligen Red och Green.

8. Gå till Format-menyn och markera undermenyn Font i den. Skriv i textfältet Type Here som dyker upp till höger om den, Times New Roman. Skriv i det textfält som dyker upp direkt under det, Courier och under det, Comic Sans.

9. Efter du skrivit Comic Sans och tryckt på Enter, klicka på den lilla pilen till höger om textfältet Type Here under Comic Sans och välj Separator. Så har du skapat under-undermenyn Separator i undermenyn Font.

10. Fortsätt med att skapa två under-undermenyer till under Separatorn, nämligen Bold och Italic.

11. Kompilera och kör. Testa dina menyer samt undermenyer.

12. Hämta en Label-kontroll till formen och döp den till displayLabel.

label1:

Egenskap	Värde
(Name)	displayLabel
Text	Använd Format-menyn för att ändra denna texts utseende.
Autosize	False
Size	435; 135
Font	Times New Roman; 14pt
Location	80; 120

Projektets kod

Lägg in koderna i filen Form1.cs, klassen **`Form1`** i den här ordningen:

13. Gå till formen, menyn File och undermenyn About. Dubbelklicka på About-undermenyn och skriv in kroppen till följande händelsemetod:

```
private void AboutToolStripMenuItem_Click(
                                object sender, EventArgs e)
{
   MessageBox.Show("Detta program demonstrerar\n" +
                   "användningen av menyer.",
                   "About", MessageBoxButtons.OK,
                     MessageBoxIcon.Information);
}
```

14. Gå till formen (fliken bredvid), menyn File och undermenyn Exit. Dubbelklicka på Exit och skriv in kroppen till följande händelsemetod (OBS! endast en rad):

```
private void ExitToolStripMenuItem_Click(
                               object sender, EventArgs e)
{
   Application.Exit();
}
```

15. Skriv in följande i filen **`Form1.cs`**:

```
private void ClearColor()
{
   blackToolStripMenuItem.Checked = false;
```

```
   blueToolStripMenuItem.Checked  = false;
   redToolStripMenuItem.Checked   = false;
   greenToolStripMenuItem.Checked = false;
}
```

16. Gå till undermenyn Format-Color, dubbelklicka på Black och skriv in följande:

```
private void BlackToolStripMenuItem_Click(
                                object sender, EventArgs e)
{
   ClearColor();
   displayLabel.ForeColor = Color.Black;
   blackToolStripMenuItem.Checked = true;
}
```

17. Gå till undermenyn Format-Color, dubbelklicka på Blue och skriv in följande:

```
private void BlueToolStripMenuItem_Click(
                               object sender, EventArgs e)
{
   ClearColor();
   displayLabel.ForeColor = Color.Blue;
   blueToolStripMenuItem.Checked = true;
}
```

18. Skriv motsvarande händelsemetoder till de andra färgerna Red och Green.

    OBS! Händelsemetoder kommer inte att fungera om du bara klipper, klistrar och ändrar i koden. Du måste dubbelklicka på undermenyerna från formen och koda sedan, för att få en automatisk koppling mellan grafiken och koden.

19. Skriv in följande i filen **`Form1.cs`**:

```
private void ClearFont()
{
   timesNewRomanToolStripMenuItem.Checked = false;
   courierToolStripMenuItem.Checked       = false;
   comicSansToolStripMenuItem.Checked     = false;
}
```

20. Gå tillundermenyn Format-Font, dubbelklicka på Times New Roman och skriv in följande:

```
private void TimesNewRomanToolStripMenuItem_Click(
                                object sender, EventArgs e)
{
   ClearFont();
   timesNewRomanToolStripMenuItem.Checked = true;
   displayLabel.Font = new Font("Times New Roman", 14,
                                displayLabel.Font.Style);
}
```

21. Skriv motsvarande händelsemetoder till de andra fonterna Courier och Comic Sans.

    OBS! Ändra i koden det fysiska namnet på fonten Comic Sans till **`"Comic Sans MS"`**. Så heter fontens namn i den nya versionen av Visual Studio.

    Samma sak här: Händelsemetoder kommer inte att fungera om du bara klipper, klistrar och ändrar i koden. Du måste dubbelklicka på undermenyerna från formen och koda sedan.

22. Gå tillundermenyn Format-Font, dubbelklicka på Bold-undermenyn i och skriv in följande:

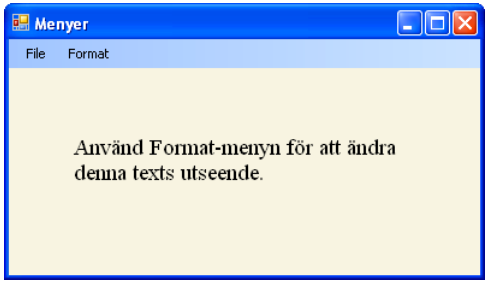
```
private void BoldToolStripMenuItem_Click(
                                object sender, EventArgs e)
{
   boldToolStripMenuItem.Checked =
                           !boldToolStripMenuItem.Checked;
   displayLabel.Font = new Font(displayLabel.Font,
              displayLabel.Font.Style ^ FontStyle.Bold);
}
```

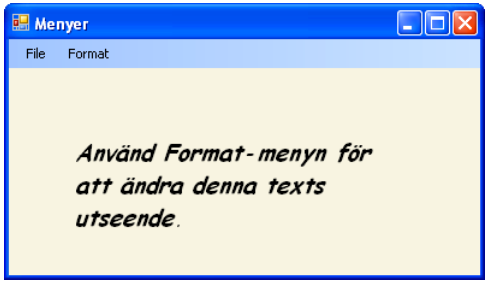
23. Dubbelklicka på Italic-undermenyn i Format-Format-menyn och skriv in följande:

```
private void ItalicToolStripMenuItem_Click(
                              object sender, EventArgs e)
{
   italicToolStripMenuItem.Checked =
                         !italicToolStripMenuItem.Checked;
   displayLabel.Font = new Font(displayLabel.Font,
              displayLabel.Font.Style ^ FontStyle.Italic);
}
```

24. Kompilera och kör. Testa menyerna.

Så här kan det se ut när man kör programmet **`Menus`**:





1.14 Multiple Document Interface (MDI)

1. Skapa en Windows Forms Application och döp den till MDI.

Form1:

Egenskap	Värde
Text	Multiple Document Interface
Size	1100; 770
IsMdiContainer	True

När egenskapen IsMdiContainer sätts till True blir denna form en *Container* eller en s.k. *förälderform*. Lägg märke till att formens bakgrundsfärg ändras till grå. Så länge IsMdiContainer är False har vi en ”vanlig” form, även kallad *barnform*. By default är IsMdiContainer alltid False.

Skapa en barnform

2. Skapa en ny, andra form i projektet, så här: Högerklicka på projektnamnet MDI i Solution Explorer: → Add → New Item... . Välj i dialogrutan Add New Item... den nya formens typ: Form (Windows Forms). Döp den till ChildForm genom att skriva i textfältet Name: ChildForm.cs. Klicka på knappen Add.

Barnformens design

3. Markera barnformen ChildForm och ändra dess storlek:

ChildForm:

Egenskap	Värde
Size	650; 340

4. Hämta från Toolbox, All Windows Forms, en PictureBox-kontroll till ChildForm:

pictureBox1:

Egenskap	Värde
BackColor	(Web) White
Dock	Fill (mellersta rutan)
SizeMode	StrechImage

Lägg märke till att PictureBox-kontrollen lägger sig över hela barnformen.

Barnformens kod

5. Markera ChildForm.cs i Solution Explorer, högerklicka och välj View Code för att se barnformens kod. Byt ut hela koden i childForm.cs till följande:

```
using System.Drawing;
using System.Windows.Forms;
using System.IO;
```

```
namespace MDI
{
   public partial class ChildForm : Form
   {
      public ChildForm(string title, string fileName)
      {
         InitializeComponent();
         Text = title;
         pictureBox1.Image =
            Image.FromFile(Directory.GetCurrentDirectory()
                                                + fileName);
      }
   }
}
```

Skapa menyer i Form1

6. Lämna ChildForm och återgå till Form1 (fliken bredvid). Hämta från Toolbox, All Windows Forms, en MenuStrip-kontroll till Form1: En tom menyradplats lägger sig till formen direkt under rubriken. Längst ner till vänster i *component tray* dyker upp en annan del av den nya kontrollen med namnet menuStrip1. Den kan användas för att markera MenuStrip-kontrollen.

7. Markera MenuStrip-kontrollen. Gå in med musen till textfältet Type Here på den nya kontrollen menuStrip1 och skriv där File. Tryck på Enter.

8. Markera menyradplatsen och skriv i textfältet Type Here till höger om File-menyn, Window. Tryck på Enter (upprepas inte längre i beskrivningen).

9. Markera File-menyn och skriv i textfältet Type Here som dyker upp under den (OBS! inte bredvid den) New. Klicka i textfältet som dyker upp direkt under New-textfältet och skriv Exit.

10. Markera New-menyn och skriv i textfältet Type Here som dyker upp till höger om den, Child1. Skriv i det textfält som dyker upp direkt under Child1-text-fältet, Child2. Skriv i det textfält som dyker upp direkt under Child2-textfältet, Child3.

11. Skapa på samma sätt även undermenyer under Window-menyn: Markera den och skriv i textfältet Type Here som dyker upp under den (OBS! inte bredvid den) Cascade. Skriv i det textfält som dyker upp direkt under Cascade, Tile Horizontal. Skriv i det textfält som dyker upp direkt under Tile Horizontal, Tile Vertical.

Form1:s kod

12. Gå till File-menyn och markera Exit-undermenyn. Dubbelklicka på den och skriv in kroppen (endast en rad) till händelsemetoden:

```
private void exitToolStripMenuItem_Click(
                                  object sender, EventArgs e)
{
   Application.Exit();
}
```

13. Gå tillbaka till formen Form1, där till File-menyn och undermenyn New. Klicka på den och markera Child1-undermenyn. Dubbelklicka på den och skriv in:

```
private void child1ToolStripMenuItem_Click(
                                object sender, EventArgs e)
{
   ChildForm child = new ChildForm("Första bilden",
                                   "\\Valkomst.gif");
   child.MdiParent = this;
   child.Show();
}
```

OBS! Koden kan kompileras, men inte exekveras just nu, därför att filen **`Valkomst.-gif`** som anges i koden ovan, inte finns i projektet. Vi kommer att fixa det senare.

14. Gör samma som i punkten ovan med Child2-undermenyn. Markera den, dubbelklika på den och skriv in följande.

```
private void child2ToolStripMenuItem_Click(
                                object sender, EventArgs e)
{
   ChildForm child = new ChildForm("Andra bilden",
                                   "\\Valkomst.gif");
   child.MdiParent = this;
   child.Show();
}
```

15. Gör motsvarande med Child3-undermenyn. Glöm inte **`"Tredje bilden"`**.

OBS! Det går inte att klippa, klistra och ändra i koden. Du måste dubbelklicka på undermenyn från formen och koda sedan, för att få en koppling mellan grafiken och koden.

16. Gå tillbaka till formen Form1, där till Window-menyn och undermenyn Cascade. Dubbelklicka på den och skriv in följande:

```
private void cascadeToolStripMenuItem_Click(
                                object sender, EventArgs e)
{
   this.LayoutMdi(MdiLayout.Cascade);
}
```

17. Dubbelklicka på Tile Horizontal-undermenyn i Window-menyn och skriv in följande:

```
private void tileHorizontalToolStripMenuItem_Click(
                                object sender, EventArgs e)
{
    this.LayoutMdi(MdiLayout.TileHorizontal);
}
```

18. Dubbelklicka på Tile Vertictal-undermenyn i Window-menyn och skriv in följande:

```
private void tileVerticalToolStripMenuItem_Click(
                                    object sender, EventArgs e)
{
    this.LayoutMdi(MdiLayout.TileVertical);
}
```

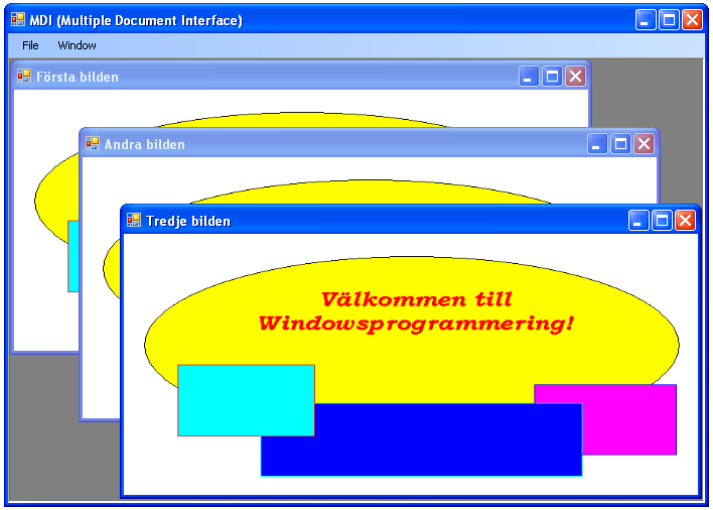
Infoga bildfilen i projektet

19. Gå till webbsidan www.taifun.se. Klicka där på bokens bild *Programmering 2 med C#*, scrolla ned och klicka på länken Valkomst.gif. En zip-fil laddas ned som innehåller filen Valkomst.gif. Klicka på zip-filen och extrahera den på din dator.

20. Återgå till Visual Studio, projektet MDI. Markera projektnamnet MDI i Solution Explorer, högerklicka på det och välj Open Folder in File Explorer. MDI:s projektmapp på din dator öppnas. Navigera till den plats på din dator där du sparat bildfilen **`Valkomst.gif`**. Kopiera filen **`Valkomst.gif`**.

21. Återgå till MDI:s projektmapp du öppnade ovan, undermappen bin → Debug. Klistra in bildfilen **`Valkomst.gif`** i den. Nu finns bildfilen i projektet.

22. Kompilera och kör. Så här kan det se ut när man från File-menyn väljer alla tre barnformer samt Cascade från Window-menyn:



Övningar till kapitel 1

1.1 Skapa en Console Application och kalla den för AdditionC. Den ska definiera och initiera två heltalsvariabler och producera t.ex. följande utskrift till konsolen:

```
Summan av 9 och 2 är 11
```

9 och **2** ska vara de värden som variablerna blivit inirierade till i programmet.

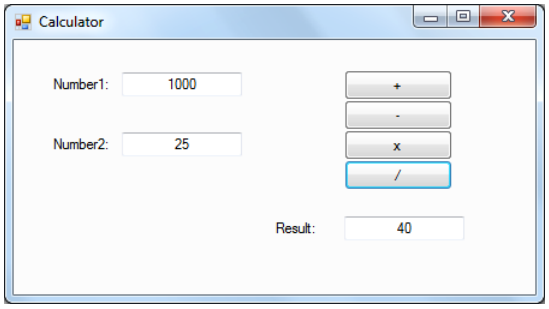
1.2 Skapa en Windows Forms Application och kalla den för AdditionW. Den ska göra samma sak som lösningen i övning 1.1, bara att utskriften inte hamnar i konsolen utan i en MessageBox och visas när man klickar på en knapp (med texten *Visa MessageBox*) i formfönstret. Förse MessageBoxen med rubriken *Windows Addition*.

1.3 I både övn 1.1 och 1.2 är heltalsvärdena 9 och 2 hårdkodade. Vidareutveckla dessa övningar genom att skapa ett användarvänligt, interaktivt grafiskt gränssnitt där man kan mata in vilka tal som helst och få summan utskriven i en MessageBox när man klickar på en knapp med texten *Addera*. Välj lämpliga rubriker för formen och MessageBoxen. Kalla projektet för Addition.

1.4 Skapa en Windows Forms Application och kalla den Division. Modifiera lösningen i övn 1.3 så att beräkningens resultat inte skrivs ut till en MessageBox utan placeras i ett textfält som läggs till i formen. Välj den här gången division som räkneoperation.

1.5 Skapa en Windows Forms Application och kalla den för SafeDivision. Skapa samma grafiska gränssnitt som i projektet Division (övn 1.4). Applikationen ska genomföra säker division, dvs ta hand om en eventuell division med 0. Modifiera koden i Form1.cs genom att införa ett *egengenererat undantag* för fallet att användaren matar in 0 i det andra textfältet. Styr meddelandena från undantagshanteringen till en MessageBox.

1.6 Vidareutveckla övningsserien 1.1-1.5 till en komplett kalkylator som inkluderar de fyra räknesätten. Det grafiska gränssnittet kan se ut som bilden till höger. Förse divisionen med en undantagshantering (sid 28) som vid division med 0 skriver ut ett felmeddelande till en MessageBox.



1.7 ***Grafiska applikationer (projekt)*** Gå igenom dina konsolapplikationer som du skrivit hittills. Undersök vilka av dem som är lämpliga för att skriva om dem till grafiska applikationer. Integrera all inläsning från och utskrift till konsolen helt och hållet i en grafisk miljö. OBS! En befintlig konsolapplikation kan inte laddas i Visual Studio och göras om till en Windows Forms Application. Man måste skapa en ny Windows Forms Application och förse den både med grafik och kod som gör samma sak som den ursprungliga konsolapplikationen. Skillnaden är bara att användaren kommunicerar med programmet via ett grafiskt gränssnitt istället för via konsolen. Har du inga konsolapplikationer fortsätt här.

De projektuppgifter som nu följer är konsolapplikationer. Repetera hur man skapar en C# Console Application i ***Appendix***, sid 265.

1.8 *Gissa tal – ett spel (projekt)*

Skriv en Console Application som slumpmässigt genererar ett heltal mellan 1 och 100. Låt användaren i flera försök gissa detta hemliga tal. För att stödja gissningen, låt programmet efter varje gissningsförsök skriva ut, om det gissade talet var mindre eller större än programmets hemliga slumptal. Låt användaren försöka igen. Gissningen ska pågå tills man gissat rätt. Vid rätt gissning skriv ut ett ”Grattis!”-meddelande följt av ett datorljud, t.ex. med `\a` . Förse programmet med ytterligare två funktionaliteter:

a) Vid rätt gissning skriv även ut ett meddelande om antalet gissningsförsök.
b) Ge möjligheten att avsluta spelet och få reda på programmets hemliga tal, vilket kan ske genom att mata in t.ex. `0` .

Ett exempel på en omgång av Gissa tal-spelet kan se ut så här:

```
Gissa ett heltal mellan 1 och 100 (Avsluta med 0):      50

                För LITET, försök igen!                 75

                För LITET, försök igen!                 87

                För STORT, försök igen!                 81

                För LITET, försök igen!                 84

                Grattis, du har gissat rätt efter 4 försök.
```

Eller om spelaren blivit trött och vill avluta genom att mata in t.ex. `0` :

```
Gissa ett heltal mellan 1 och 100 (Avsluta med 0):      0

                Avbrott: Programmets hemliga tal var    66
```

Ledning:

a) Hantering av slumptal i C#:
För att slumpmässigt generera ett heltal mellan 1 och 100 kan man skriva:

```
Random r = new Random();
int secret = r.Next(1, 101);
```

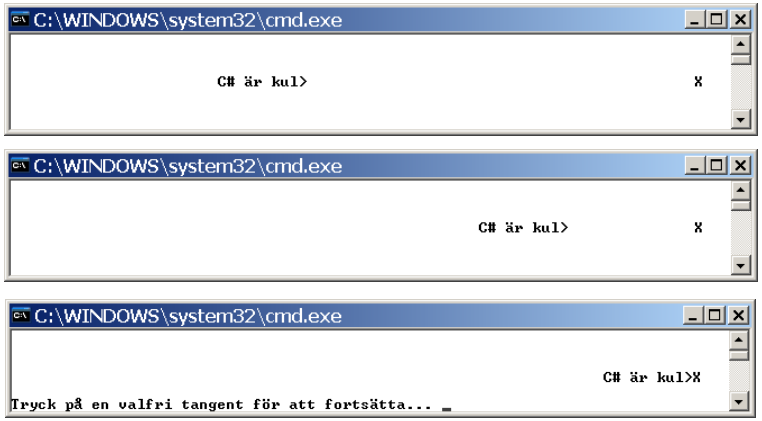
Första raden skapar ett objekt av klassen **`Random`**. Variabeln **`r`** av typ **`Random`** refererar till detta objekt. I den andra raden anropas metoden **`Next()`** som är definierad i klassen **`Random`**. Därför måste anropet ske med referensvariabeln **`r`** via punktnotation. Parametrarna **`1`** och **`101`** bestämmer att metoden returnerar ett heltal mellan 1 och 100. Returvärdet tildelas heltalsvariabeln **`secret`**, programmets hemliga slumptal.

b) Resten består huvudsakligen av en loop som tillåter spelaren gissa upprepade gånger. I loopen kan en kontrollstruktur användas som är lämplig för flervägsval, för att skilja mellan de olika alternativen. Du kan styra loopens förlopp samt avslutning t.ex. med en logisk variabel av typ **`bool`**.

Extrauppgift:

Fundera och testa på en *spelstrategi* som kan minimera antalet gissningsförsök. I körexemplet ovan med bara 4 försök har en sådan strategi använts. Hur skulle du beskriva den?

1.9 ***Löpande texten (projekt)*** Skriv ett program som 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`**. Så här kan ett körresultat se ut:



Ledning:

a) Skriv ut med hjälp av ett antal mellanslag krysset i slutet av en rad i konsolen utan radbyte. Anteckna antalet mellanslag krysset har avstånd från konsolens vänstra rand. Stanna på samma rad, gå med hjälp av escapesekvensen **`\r`** (car-

riage return) till början av raden och skriv ut texten **`C# är kul>`**. Gör experiment med **`\r`** för att bekanta dig med dess funktion. Så här borde ett körresultat se ut:

b) Skriv en **`for`**-loop. Ta bort (dvs skriv ut) i varje varv av loopen med 10 styck **`\b`** texten **`C# är kul>`** som ritats i förra varvet (initialt texten ovan), följt av ett eller flera mellanslag (vilket påverkar rörelsens ”hastighet”). Skriv sedan om texten **`C# är kul>`**. Välj som antal varv i loopen kryssets avstånd från konsolens vänstra rand (antecknat i a) minus textens längd – i det föreslagna exemplet 10. Då kommer rörelsen att stoppas strax innan texten ”träffar” på **`X`**.

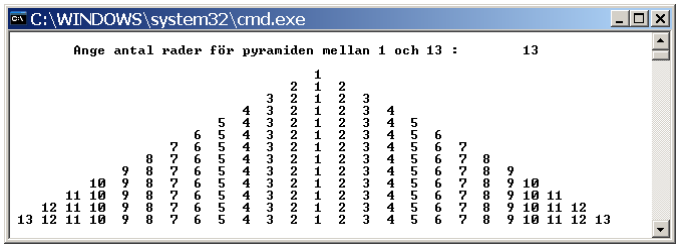
Även om du gjort allt rätt kommer du inte se texten att röra sig om du inte lägger in en fördröjning i loopen, eftersom allt går så fort och ögat inte hinner se något förlopp. Fördröjningen kan du åstadkomma genom att lägga in i loopen satsen:

```
System.Threading.Thread.Sleep(100);
```

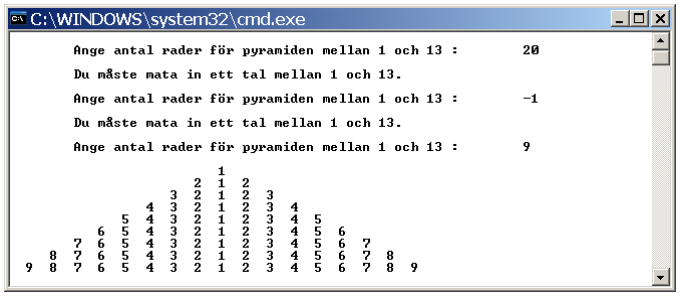
Detta ger en fördröjning på 100 milisekunder i varje varv av loopen.

1.10 *Pyramiden (projekt)*

Slutmålet med detta projekt är att utveckla ett program som skriver ut en pyramidliknande figur med tal, t.ex. så här:

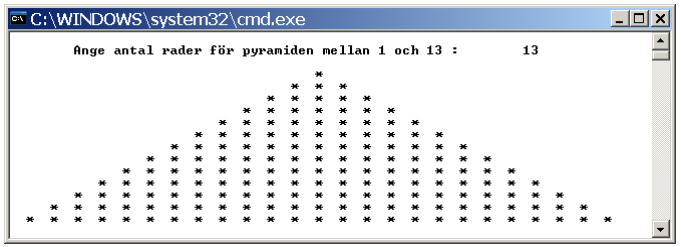


Programmet ska vara så generellt att det skriver ut talpyramider även om man matar in mindre antal rader. Men om användaren inte följer ledtextens instruktion att mata in tal mellan 1 och 13 ska programmet inte skriva ut talpyramiden utan uppmana användaren att hålla sig till det föreskrivna talintervallet [1, 13]. Anledning till denna restriktion är att talpyramiden inte ryms i konsolen om man överskrider detta intervall . Så här kan då en dialog t.ex. se ut:



Tips till Pyramiden:

För att komma igång med talpyramiden, börja med att skriva ett program som ritar en stjärnpyramid:



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 2

Objektorienterad Programmering

2.1 Vad är objektorienterad 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 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 genomsyr alla våra program, eftersom C# med alla sina fördefinierade biblioteksprogram är i högsta grad objektorienterade.

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 be-

höver 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 fäljande sätt:

```
minVolvo.Kör(40);
```

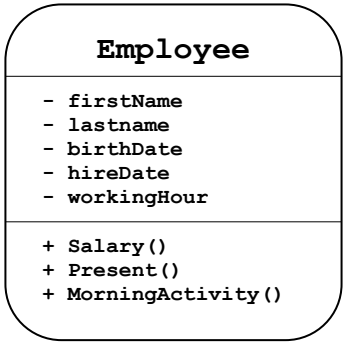
Det kan t.ex. betyda: Kör bilen med hastigheten **`40`** km/h. Värdet **`40`** kallas då en *parameter* som skickas till metoden när den anropas. I så fall måste även metoden **`Kör()`** vara definierad så att den har beredskapen att ta emot denna parameter. Så det kan inte vara samma metod som anropades *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 (algoritmen *Morgonsyssla* i *Progr1+, 1.4*). Detta 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.

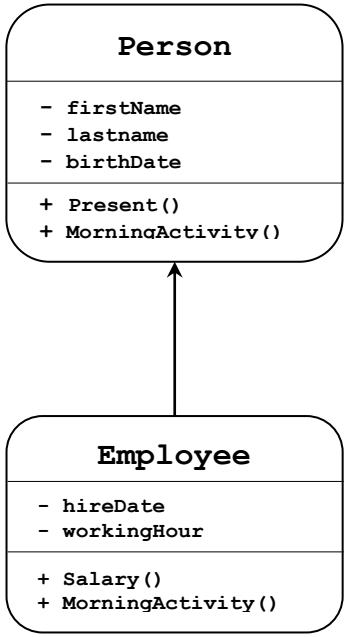
Personen kan t.ex. vara en anställd vilket förresten skulle förklara varför han tar sig till jobbet. I så fall är personen ett objekt av kategorin eller klassen *Employee*. Därför definieras en klass som beskriver alla anställda. Personen i fråga görs till ett objekt, ett exemplar av denna klass. På 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 höger 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 detalj på sid 93.



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`**.

Observera att klassen **`Employee`** inte har två utan fem attribut därför att den via arvrelationen även har **`Person`**-klassens tre attribut. Samma gäller för metoderna: **`Employee`**-klassen ärver metoden **`Present()`** från klassen **`Person`**. Modellen ovan går utifrån att personer presenterar sig på samma sätt som anställda. Sedan har anställda en löneberäkningsmetod som icke-anställda personer saknar. Men varför står metoden **`MorningActivity()`** i båda klasser? Närmare bestämt: Varför förekommer den i **`Employee`**-klassen fast den ärver den från superklassen? Svaret ges av ett annat koncept inom objektorienterad programmering:

Polymorfism

Modellen ovan går utifrån att icke-anställda personer har en annan form av morgonsyssla än anställda. De kanske inte tar sig till jobbet, i alla fall inte alla, utan har en annan morgonsyssla. Så vi har här att göra med två olika morgonsysslor tillhörande två olika klasser, men med samma namn. För objekt av typ **`Person`** kommer den ena och för objekt av typ **`Employee`** kommer den andra att gälla. Men varför har de samma namn? Vore det inte bättre, för att undvika namnkonflikt, att ge dem olika namn, när de ändå är olika metoder? Faktiskt inte!

Anledningen till att de har samma namn är följande: För det första blir det ingen namnkonflikt därför att de tillhör olika typer av objekt. De är inte fristående utan inkapslade i var sitt objekt som skiljer åt dem. För det andra ska vi inte i onödan göra utvecklingsarbetet komplicerat genom att hitta på nya namn på metoder som skiljer sig från varandra endast i detaljer. Ingen människa skulle kunna komma ihåg så många namn. För det tredje vill vi efterlikna verkligheten där det bara kryllar av beteckningar som är identiska, men har olika innebörd i olika sammanhang. Inte heller det vanliga språket har olika namn på dem. Ta följande exempel: Att bromsa en lastbil görs på ett annat sätt än att bromsa en båt. Det finns ingen anledning att hitta på ett annat namn för funktionaliteten "att bromsa" hos olika typer av fordon. Tvärtom, det vore förvirrande att använda olika namn. Man vill ju helst slippa att tänka på de tekniska skillnaderna mellan olika typer av fordon när man pratar om bromsning. En och samma funktionalitet är realiserad på olika sätt. Med andra ord, man gör "samma sak", fast på annorlunda sätt. Objektorienterad programmering tar över detta koncept genom att välja ett och samma namn för olika metoder. När metoderna dessutom finns i klasser som ärver varandra kallas konceptet för *polymorfism*.

Polymorfism modifierar helt eller delvis funktionaliteten hos metoder med samma namn som förekommer i en arvhierarki.

”Poly” betyder *många* och ”morf” är *form* eller *gestalt* på latin och antik grekiska. Polymorfism handlar om en sak som har många olika gestalter, t.ex. ett ord som har många olika betydelser. En metod beskriver alltid någon funktionalitet. Polymorfism förändrar denna funktionalitet genom att definiera en metod i superklassen och definiera om innehållet, men behålla namnet i subklassen.

Objektorienterad programmering har kommit till för att förverkliga programmeringens gamla önskedrömmar om *modularisering*, *återanvändning av kod* och *strukturering av program*. Allt för att kunna underhålla stora program, förnya och vidareutveckla dem, så att de fungerar över längre tid och snabbt kan anpassas till nyuppkomna situationer.

Objektorienterad programmering bygger på tre hörnstenar:

- Inkapsling
- Arv
- Polymorfism

De sista två har vi försökt att introducera här utan att behöva skriva kod. För att förstå *inkapsling* behöver vi mer detaljerade kunskaper om objektorientering samt skriva lite kod, vilket vi gör i de kommande avsnitten. Sedan ska vi återkomma till *arv* och *polymorfism*, för att förse aven dem med kod.

2.2 Klassbegreppet

Ett första C#-program:

```
using System;

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

Hela programmet är en klass som inleds med det reserverade ordet `class` om vi bortser från `using`-direktivet. Den innehåller `Main()`. En funktion som definieras i en klass kallas för *metod*. Det som står här är `Main()`-metodens *definition*. Den *anropas* automatiskt av C#-interpretatorn, den s.k. *Virtual Machine*, när vi exekverar programmet efter att vi kompilerat koden. Kompilatorn översätter källkoden till maskinkod. Interpretatorn tolkar maskinkoden till ettor och nollor och skickar dem till datorns processor för exekvering. Klassens centrala roll framgår av följande definition för C#-program.

Vad är ett C#-program?

Ett C#-program är en samling av klasser, av vilka en och endast en måste innehålla metoden **`Main()`** som är exekveringens startpunkt.

Alla C#-program måste innehålla metoden `Main()` för att kunna exekveras, annars har exekveringen ingen startpunkt. För att exekveringen ska kunna starta i `Main()` måste metodens *huvud* skrivas så här: `static void Main()` för att kunna kännas igen av C#-interpretatorn. Metodens *kropp* (innehåll) däremot kan vi helt och hållet programmera själva. Klassen **`First`** är det enklast tänkbara C#-program därför att det endast består av en klass med metoden `Main()`. Denna metod – och inte heller någon annan – kan definieras fristående, utanför en klass. En metod måste alltid inbäddas i en klass. Det beror på att C#-programmens primära byggstenar är klasser, medan metoder är *delar* av en klass. I andra programmeringsspråk som C++ finns även *funktioner* som kan definieras fristående. I C# finns inga funktioner utan endast metoder.

Vad är en klass?

En klass är kod som på ett generellt och modulärt sätt beskriver en kategori av verkliga eller virtuella saker och ting. Den består av *datamedlemmar* samt *metoder* och används som en mall för att skapa *objekt* av klassen.

Generell är en klass därför att den beskriver en kategori av saker och ting som är föremål för datorisering. Enligt klassens mall skapas sedan *objekt* av denna kategori. Medan klassen är ett abstrakt begrepp, en abstrakt idé, är objekten verkliga eller virtuella saker och ting i den reala världen.

Modulär är en klass därför att den kodas som en namngiven modul så att den kan användas av vilka andra program som helst. Programmen byggs med dessa moduler som minsta beståndsdelar som sedan kan användas för att konstruera andra program – liknande Lego-principen (sid 81).

Vår första klass utan `Main()`

Låt oss realisera klasskonceptet genom att skapa en egen klass utan **`Main()`**: I alla våra program hittills finns all kod rakt nedskriven i **`Main()`** vilket inte är objektorienterat, även om C#:s objektorienterade klassbibliotek används flitigt. Här är vårt första program som inte innehåller **`Main()`**. Vi kallar den för **`Password`**:

```
// Password.cs
// Deklarerar klassen Password med metoden Ok() som returnerar
// true eller false

using System;

class Password
{
   public bool Ok(string passwd)          // Metoden Ok():s huvud,
   {
      return passwd == "hemligt" || passwd == "HEMLIGT"; // kropp
   }
}
```

Klassen **`Password`** skrivs i en fil som vi döper till **`Password.cs`**. I en separat fil som döps till **`PasswordUse.cs`** skriver vi klassen **`PasswordUse`** som endast innehåller metoden **`Main()`**. Den i sin tur skapar ett objekt av klassen **`Password`**.

```
// PasswordUse.cs
// Använder klassen Password, skapar ett objekt av den och
// anropar metoden Ok() som är definierad i klassen Password
// Utgör tillsammans med klassen Password ETT program

using System;

class PasswordUse
{
   static void Main()
   {
      string input;
      Password p = new Password();         // Objekt skapas
```

```
      do                                    // do-loop
      {
         Console.Write("\n\tSkriv ditt lösenord:\t");
         input = Console.ReadLine();
         if (!p.Ok(input))                  // Metoden Ok() anropas
            Console.WriteLine("\n\tFel lösenord. Försök igen!");
      } while (!p.Ok(input));               // Metoden Ok() anropas

      Console.WriteLine("\n\tOK, nu är du inloggad!\n");
   }
}
```

Båda klasser laddas i ett Visual Studio-projekt av typ *Console Application*. En körning av programmet ger t.ex.:

```
       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.

Klassen **`Password`** kan endast kompileras men inte exekveras, för exekveringen startar i **`Main()`**. Och någon **`Main()`** finns ju inte i **`Password`**. Så **`Main()`** får skrivas endast i en av dem – i vårt fall finns den i **`PasswordUse`**. Paret **`Password/PasswordUse`** utgör nu *ett* program bestående av *två* klasser:

1. Klassen Password

Namnet **`Password`** är beskrivande för det som programmet är tänkt för. Här följer vi både de vanliga namngivningsreglerna för identifierare som gäller i C# *(Progr1+ 4.3)* och konventionen att inleda klassnamn med versaler för att skilja dem från andra identifierare som variabler osv. Valet av filnamnet **`Password.cs`** som klassen **`Password`** lagras i är inte obligatoriskt utan en konvention vi följer.

Klassen **`Password`** innehåller metoden **`Ok()`**. Av metodens huvud framgår att den returnerar ett värde av typ **`bool`**. I C# finns möjligheten att definiera inte bara logiska variabler utan även metoders returvärde med datatypen **`bool`** som är en enkel datatyp och representerar sanningsvärdena sant (**`true`**) och falskt (**`false`**). Metoden **`Ok()`** är en sådan och har den *formella parametern* **`passwd`** av typ **`string`** som tar emot strängen **`input`** från klassen **`PasswordUse`** när metoden **`Ok()`** anropas. Metodens **`return`**-sats returnerar följande logiska uttryckets sanningsvärde:

```
passwd == "hemligt" || passwd == "HEMLIGT"
```

Koden **`||`** står i C# för den logiska operatorn ELLER. Logiska uttrycket ovan har värdet **`true`** om strängen **`passwd`** är identisk med programmets hårdkodade lösenord **`hemligt`**

eller **HEMLIGT**. Är däremot **passwd** varken lika med **hemligt** eller med **HEMLIGT** blir uttryckets värde **false**. Det är den logiska innebörden av ELLER.

Klassen **Password** beskriver *begreppet* lösenord som en abstrakt idé utan att skapa ett verkligt lösenord. Den är en *mall* för att testa verkliga lösenord, en föreskrift om hur ett verkligt lösenord med en viss inmatning och ett testvärde *skulle* verifieras *om det skapades*. Den typiska operationen för verifiering av lösenord definieras i metoden **Ok()**. Klassen **Password** har inga datamedlemmar.

2. Klassen PasswordUse

Även denna klass består endast av en enda metod – nämligen **Main()**. I den deklareras först variabeln **input** av typ **string**. Sedan skapas ett objekt av klassen **Password**:

```
Password p = new Password();
```

Koden som skapar själva objektet är **new Password()** som sedan tilldelas variabeln **p** av typ **Password**. Dvs klassen **Password** spelar här rollen av en datatyp och används för att deklarera variabeln **p**. Pga att variabelns datatyp är en klass och dess värde ett objekt skiljer den sig från vanliga variabler. **p** kallas för en *referensvariabel* – kort *referensen* till objektet **new Password()**. I C++ kallas **p** för *pekaren* som pekar på objektet, i hårdvarumässiga termer *adressen* till objektets minnesutrymme. Man kan uppfatta **p** även som objektets *namn*. Vi kommer i fortsättningen att använda termen *referens*.

Ett verkligt, konkret lösenord är ett *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 **PasswordUse** skapas bara ett **Password**-objekt. *Hur* man gör det med det reserverade ordet **new** och hur man sedan kan komma åt objektet samt vad parentesen i **new Password()** betyder, kommer att behandlas i de kommande avsnitten.

Resten av koden i **PasswordUse**-klassens **Main()** består av en **do**-loop och en utskrift. Loopen börjar med att läsa in strängen **input** som användaren vill logga in med. Detta inloggningsförsök skickas till metoden **Ok()** för verifiering. Dvs metoden **Ok()** anropas med strängen **input** i parameterlistan: **Ok(input)**. Men eftersom **Ok()** är definierad i klassen **Password** måste anropet göras med referensen **p** till objektet av typ **Password**, därför: **p.Ok(input)**. Detta anrop står i villkoret till en **if**-sats. Och dessutom är anropet som pga av meoden **Ok()**:s returtyp **bool** ger ett sanningsvärde, negerat. Dvs det föregås av den logiska negationen **!** : **if (!p.Ok(input))** ... . Detta för att själva meoden **Ok()** returnerar **true** om **input** är identisk med programmets hårdkodade lösenord och **false** om det inte är fallet. Samma logiska uttryck används i **do**-loopens avslutningsvillkor: **while (!p.Ok(input))** och styr logiken i både **do**-loopen och **if**-satsen som ingår i den. Loopen ser till att dialogen mellan program och användare fort-

sätter så länge `p.Ok(input)` returnerar `true`. Och det är samma sak som att säga: när `!p.Ok(input)` blir `false`, dvs så länge man matar in felaktigt lösenord, någon sträng som varken är `hemligt` eller `HEMLIGT`.

Två filer eller en fil ?

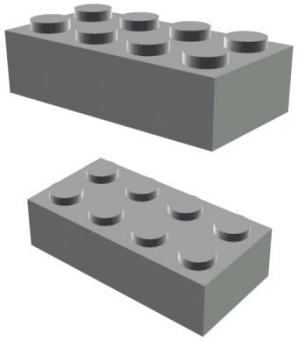
Slutligen kan man undra om det hade varit möjligt resp. rimligt att lagra båda klasser `Password` och `PasswordUse` i en och samma fil. Svaret är: Möjligt ja, men inte rimligt ur den objektorienterade programmeringens synpunkt. Därför att det går både att kompilera och köra programmet när båda klasser lagras i en fil. Men fullt objektorienterat är det inte längre, för då går man miste om hela idén med 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.

Varför klasser?

Frågan är berättigad: Varför ska man krångla till programmeringen med klasser? Är det inte enklare att skriva kod rakt ned i `Main()`? Så länge man skriver små program kan frågan bejakas. 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. 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. På vilket sätt ska vi nu diskutera i termer av *Modularisering* och *återanvändning av kod*.

Vi kan i denna lärobok aldrig komma upp till att kunna presentera sådana komplexa program som motiverade användningen av klasser i programmeringshistorien. Men idén bakom klasser och principerna i objektorienterad programmering kan även illustreras med de små program som vi brukar använda för att förklara programmeringens koncept.

2.3 Modularisering



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.

Lego-principen

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. Princien 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 objektorienterad programmering är dessa moduler *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.

Återanvändning av kod

är det andra svaret på frågan varför man i programmering sysslar med klasser. 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 kategori av saker och ting 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.

Utan modularisering

I förra avsnitt *2.1 Klassbegreppet* presenterades *ett* program bestående av två klasser som redan var objektorienterat. Men hur kommer man dit om man börjat koda icke-objektorientrat, vilket de flesta nybörjare gör? Här ska vi visa *vägen* från `"vanlig"` till objektorientrad programmering (OOP). Om det inte hade varit för pedagogikens skull – nämligen att med enkla små program illustrera principerna i OOP – hade vi kanske inte skrivit programparet **`Password/PasswordUse`** objektorienterat. Vi hade nöjt oss med att skriva all kod rakt ned i **`Main()`** i ett enda program, vilket i alla fall hade varit enkla-

re. Vi ska göra det nu och sedan modularisera upp till klassnivå steg för steg. Så hade det sett ut om vi hade struntat i all modularisering:

```
// Password_All_in_Main.cs
// Verifierar lösenord inmatat i versaler eller gemener
// Ingen modularisering: All kod skriven rakt ned i Main()
using System;

class Password_All_in_Main
{
   static void Main()
   {
      String input;                  // Lokala variabler i Main()
      bool ok;

      do
      {
         Console.Write("\n\tSkriv ditt lösenord:\t");
         input = Console.ReadLine();
         ok = (input == "hemligt" || input == "HEMLIGT");
         if (!ok)
            Console.WriteLine("\n\tFel lösenord. Försök igen!");
      } while (!ok);

      Console.WriteLine("\n\tOK, nu är du inloggad!\n");
   }
}
```

Programmet ovan är inte ett dugg objektorienterat, men har exakt samma funktionalitet och ger exakt samma utskrift som det objektorienterade programparet **`Password/-PasswordUse`** (sid 77). I nästa steg ska vi modularisera programmet genom att lyfta en del av det, skriva den som en namngiven modul – närmare bestämt en metod – utanför **`Main()`** och anropa den i **`Main()`**. Denna del är framhävd i **`do`**-loopen med vit bakgrund i koden ovan och utgör det logiska uttryck som styr både **`if`**-satsen och **`do`**-loopens avslutning.

Modularisering på metodnivå

```
// Password_Method_Module.cs
// Verifierar lösenord inmatat i versaler eller gemener
// Modulariserad på metodnivå: Inte objektorienterad
using System;

class Password_Method_Module
{
   static bool Ok(string passwd)          // Metodens definition
   {
      return passwd == "hemligt" || passwd == "HEMLIGT";
   }
```

```
   static void Main()
   {
      string input;

      do
      {
         Console.Write("\n\tSkriv ditt lösenord:\t");
         input = Console.ReadLine();
         if (!Ok(input))                  // Metodens anrop
            Console.WriteLine("\n\tFel lösenord. Försök igen!");
      } while (!Ok(input));               // Metodens anrop

      Console.WriteLine("\n\tOK, nu är du inloggad!\n");
   }
}
```

Klassen **Password_Method_Module** innehåller två metoder: **Ok()** och **Main()**. Metoden **Main()** anropar metoden **Ok()** två gånger. Vid anropet skickas den *aktuella parametern* **input**:s värde som är en sträng till den *formella parametern* **passwd**. Där jämförs den med programmets hårdkodade lösenord **hemligt** resp. **HEMLIGT**. Sedan returnerar metoden **Ok()** ett sanningsvärde **true** eller **false**, vilket i **Main()** används för att skriva ut om inloggningen lyckats eller ej. Programmet är inte objektorienterat än, därför att det inte skapats något objekt av de befintliga klasserna. Men programmet har tagit ett första steg mot OOP genom att separera en bit kod och skriva den som en namngiven modul – en metod – utanför **Main()**, men fortfarande i samma klass. Nästa steg:

Modularisering på klassnivå

```
// Password.cs
// Deklarerar klassen Password med 2 datamedlemmar och en metod
// Klassen Password med metoden Ok(): returnerar true eller false
// Kan kompileras men inte exekveras eftersom Main() saknas

using System;

class Password
{
   public bool Ok(string passwd)                  // Metoden Ok()
   {
      return passwd == "hemligt" || passwd == "HEMLIGT";
   }
}
```

Klassen **Password** är förstås samma som på sid 77 och skrivs i en separat fil. Som man ser innehåller den samma metod **Ok()** som vi vid modularisering på metodnivå hade flyttat ut ur **Main()**. I en annan fil skrivs den klass som endast innehåller metoden **Main()** där objekt av klassen **Password** skapas och som är identisk med **Password-Use** på sid 77:

```
// Password_Class_Module.cs
// Verifierar lösenord inmatat i versaler eller gemener
// Modulariserad på klassnivå: Objektorienterad
// Använder klassen Password, skapar ett objekt av den och
// anropar metoden Ok() som är definierad i klassen Password
// Utgör med klassen Password ETT program bestående av 2 klasser
using System;

class Password_Class_Module
{
   static void Main()
   {
      string input;
      Password p = new Password();              // Objekt skapas

      do
      {
         Console.Write("\n\tSkriv ditt lösenord:\t");
         input = Console.ReadLine();
         if (!p.Ok(input))                      // Metod anropas
            Console.WriteLine("\n\tFel lösenord. Försök igen!");
      } while (!p.Ok(input));                   // Metod anropas

      Console.WriteLine("\n\tOK, nu är du inloggad!\n");
   }
}
```

Den uppmärksamme läsaren har väl konstaterat att vi vid övergången från modularisering på metodnivå (sid 82) till klassnivå (sid 83) har ändrat i metoden **`Ok()`**:s huvud modifieraren från **`static`** till **`public`**. Här följer en förklaring:

Anledningen varför vi vid övergången från modularisering på metodnivå till klassnivå ändrade metoden **`Ok()`**:s modifierare från **`static`** till **`public`** är att vi i klassen **`Password_Class_Module`** (sid 84) redan har ett objekt av denna typ. Därför kan vi anropa metoden **`Ok()`** med hjälp av detta objekts referens: **`p.Ok(input)`**. Pga metodens **`non-static`** egenskap tillhör metoden objektet och inte klassen. Dessutom måste metoden **`Ok()`** ha egenskapen **`public`** i sin definition i klassen **`Password`** för att kunna kommas åt från en annan klass, nämligen **`Password_Class_Module`**.

Anledningen varför metoden **`Ok()`** i sin definition i klassen **`Password_Method_Module`** (sid 82) har egenskapen **`static`** är att vi vill slippa skapa ett objekt när vi anropar den. Därför kan vi anropa metoden **`Ok()`** direkt: **`Ok(input)`**. Egenskapen **`static`** gör att metoden tillhör klassen och inte ett objekt av den. Vi vill ju i detta program demonstrera modularisering på metodnivå och medvetet inte koda objektorienterat. Annars är det fullt möjligt att slippa **`static`** och istället skapa ett objekt av klassen **`Password_Method_Module`** direkt efter deklarationen av variabeln **`input`**. Testa gärna!

2.4 Användning av klasser

På sid 76 ställdes upp en defnition för klassbegreppet. Här följer en annan:

> En klass är en ny, egendefinierad och sammansatt datatyp som skapas med det reserverade ordet **`class`**.

Kan ett begrepp ha flera definitioner? Ja, om de inte motsäger varandra och belyser olika aspekter av begreppet. Vilken som är relevant i en viss situation avgörs av sammanhanget begreppet används i. Det finns ingen begränsning på vilka, hur många eller vilka kategorier av saker och ting man kan involvera i sin klass, inkl. andra klasser (Komposition). Allt beror på den verkliga miljön man vill modellera i sitt program.

Följande steg måste tas när man använder **`class`** för att skapa nya, egna datatyper:

1. *Deklaration av en klass*
2. *Definition av ett objekt*
3. *Åtkomst till objektets medlemmar*

1. Deklaration av en klass

Med deklaration av en klass menas själva *koden* man skriver för klassen. Denna kod allokerar (reserverar) inget minnesutrymme utan introducerar endast ett nytt ord, en ny identifierare i programmet, nämligen en *ny datatyp*: Deklarationen av en klass kan med hjälp av det reserverade ordet **`class`** generellt skrivas så här:

`class` *KlassNamn*
{
    Deklaration av *datamedlemmar*
    Deklaration av *metoder*
}

KlassNamn är ett namn som vi kan välja fritt med hänsyn till de kända regler och rekommendationer som gäller för all namngivning *(Progr1, 2.2)* samt konventionen att inleda klassnamn med en versal. Sedan kan vi använda namnet som *datatyp* i programmet för att – och endast för att – definiera nya typer av variabler som kallas *referensvariabler*. Det är variabler som kan lagra adresser till objekt av klassens typ. Med koden ovan skapas den nya datatypen. Pga den speciella styrkan att kunna beskriva vad som helst betecknas **`class`** själv inte längre som datatyp utan som datastruktur, abstrakt datatyp eller kort som *klass* då den fungerar på en kvalitativt högre nivå än vanliga datatyper. Här har vi ett exempel på en klass som beskriver kategorin *Anställd*:

```
// Emp.cs
// Deklarerar klassen Emp med 4 datamedlemmar och 2 metoder
// Båda metoder returnerar strängar med return-satsen
// AsString() är klassens strängrepresentationsmetod
```

```
using System;

class Emp
{
   public int empNo;                    // Datamedlemmar
   public String firstName, lastname;
   public float salary;

   public String Email()                // Metoden Email()
   {
      return (firstName.Substring(0, 1) + lastname).ToLower();
   }

   public String AsString()             // Metoden AsString()
   {
      return "\t" + firstName + " " + lastname + "\n" +
             "\tLön: "              + salary  + "\n" +
             "\tE-mail: "           + Email() + "\n" +
             "\tAnställningsnr: "   + empNo   + "\n" ;
   }
}
```

I filen **Emp.cs** ovan deklareras den nya klassen **Emp** som har fyra datamedlemmar **empNo**, **firstName**, **lastname** och **salary**. Observera att syntaxen för deklarationen av datamedlemmarna är som i vanliga deklarationssatser för variabler – med skillnaden att de dessutom måste deklareras som **public** för att en annan klass ska kunna komma åt dem. Annars hade de by default varit **private**. Man ser också att man i en klass kan blanda data av helt olika typer, här: **int**, **String** och **float**. Man kan ha datamedlemmar inte bara av fördefinierade klasser som **String**, utan även av egendefinierade klasser. I regel är datamedlemmar i en klass endast deklarerade, men inte initierade än, det görs först när ett objekt skapas. Skälet är att klassen enligt definition ska vara generell. Skulle ett värde till någon datamedlem vara hårdkodad i klassen, skulle alla objekt få detta värde, vilket just i exemplet ovan inte vore önskvärt. Det finns däremot situationer där man uttryckligen vill initiera vissa datamedlemmar i klassen. Möjligheten till det finns vilket vi kommer att återkomma senare till.

Metoden **Email()** konstruerar en sträng bestående av förnamnets initial och hela efternamnet – en ganska vanlig policy för e-mailadresser – genom att anropa metoden **Substring()** som tar ut den första bokstaven från **firstName**. Den konkateneras med **lastname**. Låt oss anta att man vill ha hela e-mailsträngen i gemener. Då kan en annan **String**-metod vara till hjälp, nämligen **ToLower()**. På så sätt uppstår foljande kod:

```
firstName.Substring(0, 1) + lastname).ToLower()
```

Den konstruerar en sträng, t.ex. strängen *acarlsson* om **firstName** refererar till *Anders* och **lastname** till *Carlsson*. Denna sträng returneras av metoden **Email()** när den anorpas – tack vare reserverade ordet **return** som står framför hela koden ovan. Därför kallas **Email()** för en metod *med returvärde*, till skillnad från **void**-metoder som inte returnerar något värde *(Progr1, 6.6)*.

Även **Emp**-klassens andra metod **AsString()** har en **return**-sats – snarare består endast av den – och är därför en metod med returvärde. Returvärdet är en konkatenerad sträng bestående av **Emp**-klassens datamedlemmar med lite förklarande text och layout (radbyten). Denna metod är till för att ge en strängrepresentation av objektet, dvs för att få en sträng med en anställds alla uppgifter, kort för att ”skriva ut” en anställd, när den skapas som ett **Emp**-objekt. Det finns även möjligheten att använda C#:s fördefinierade metod **ToString()**.

2. Definition av ett objekt

När en klass definierar en ny datatyp kan objekt av denna klass anses som *variabler* av denna nya datatyp. Att definiera ett objekt är således samma sak som att definiera variabler av den datatyp som definieras av klassen. Därför kan man anse ett objekt som en ny, mer sofistikerad (sammansatt) typ av variabel. Programmet **EmpTest** nedan demonstrerar definition av objekt av klassen **Emp**, tilldelning och utskrift av de skapade objektens datamedlemmar samt anrop av deras metoder.

Att en klass är en ny sammansatt datatyp som skapas med det reserverade ordet **class** ser man i klassen **Emp** (sid 85) som är sammansatt av klassen **String** och datatyperna **int** och **float**. I programmet **EmpTest** (sid 87) ser man dessutom att satsen **Emp a;** att klassen **Emp** används som en datatyp för att definiera variabeln **a** som en referens till ett **Emp**-objekt. Förutsättning är förstås att klassen **Emp** är definierad innan.

```
// EmpTest.cs
// Använder klassen Emp för att skapa ett objekt av klassen Emp
// Tilldelar objektets datamedlemmar värden och skriver ut dem
// Ändrar lönen och skriver ut löneskillnaden samt nya data
using System;

class EmpTest
{
   public static void Main()
   {
      float oldSalary, procent = 15;     // Lokala variabler
      String output;

      Emp a;                             // Referens skapas
      a = new Emp();                     // Objekt definieras
                                         // och tilldelas a
      a.empNo     = 123;                 // Tilldelning av
      a.firstName = "Anders";            // datamedlemmar
      a.lastname   = "Carlsson";
      oldSalary   = a.salary = 21450;

      output = a.AsString();             // Lagring av gamla data

      a.salary = a.salary * (1 + procent/100); // Löneförhöjning
```

```
                                          // Utskrift av data:
      Console.WriteLine("\n\tAnställden\n" + output          +
              "\n\tfår en löneförhöjning på " + procent      +
              "%.\n\tVåra lönekostnader kommer att öka med " +
              (a.salary - oldSalary) + ".\n\n\t"             +
              "Uppdaterad anställd:\n" + a.AsString());
   }
}
```

Själva objektet som skapas av klassen **Emp** är den med vit bakgrund framhävda koden **new Emp()**, medan **a** är en referens till objektet som i sin tur definieras med koden **Emp a**. I programmet ovan kopplas de med varandra med tilldelning vilket gör att **a** får det nya objektets adress och kan därför i fortsättningen användas för att referera till objektet. Parenteserna i **Emp()** måste vara med, annars kan koden inte kompileras vilket förklaras senare. Ett objekt kan ha flera referenser. T.ex. skulle **Emp b = a;** ge ytterligare en referens till samma objekt eftersom den får samma adress till objektet. Programmet ovan använder bara en referens och producerar följande resultat när det exekveras:

```
Anställden
Anders Carlsson
Lön: 21450
E-mail: acarlsson
Anställningsnr: 123

får en löneförhöjning på 15%.
Våra lönekostnader kommer att öka med 3217,5.

Uppdaterad anställd:
Anders Carlsson
Lön: 24667,5
E-mail: acarlsson
Anställningsnr: 123
```

Efter att ha skapat en anställd dvs ett objekt av typ **Emp** och tilldelat till den vissa värden, anropas objektets strängrepresentationsmetod **AsString()** för att lagra dem i strängen **output**:

```
output = a.AsString();
```

Denna sträng skrivs ut i slutet av programmet. Vi får den anställdas oförändrade uppgifter. Vid tilldelning av den anställdes lön lagras värdet dessutom i variabeln **oldSalary** för att kunna beräkna skillnaden efter löneförhöjningen:

```
a.salary = a.salary * (1 + procent/100);
```

a.salary - oldSalary ger sedan löneskillnaden som skrivs ut. Den sista utskriften av den anställdas nya uppgifter med den ändrade lönen görs med anropet av **AsString()** direkt i utskriftssatsen.

3. Åtkomst till objektets medlemmar

Efter att ha skapat ett objekt vill man kunna arbeta med objektets medlemmar. När man generellt pratar om medlemmar måste man skilja mellan två typer av medlemmar, *datamedlemmar* och *metoder*. Dvs: Att komma åt objektets datamedlemmar och *att anropa* objektets metoder. För båda ändamål används samma teknik som redan nämnts i olika sammanhang och som vi tar upp nu i detalj:

Punktnotation

Som redan tidigare nämnts betyder *notation* sättet att skriva. Sättet att skriva kod för att komma åt både ett objekts datamedlemmar och metoder kallar vi för *punktnotation*. Om vi tar vårt exempel i programmet **`EmpTest`** med objektet av typ **`Emp`** som **`a`** refererar till, har vi redan sett att koden **`new Emp()`** skapar objektet genom att allokera minne åt det och fylla det med default-initialvärden. Vill vi sedan tilldela objektets datamedlemmar våra egna värden, kan vi skriva:

```
a.empNo     = 123;
a.firstName = "Anders";
a.lastname   = "Carlsson";
a.salary    = 21450;
```

`empNo` till den anställd som **`a`** refererar till, ska vara **`123`** osv. **`empNo`** är en datamedlem i objektet och inte en fritt tillgänglig variabel. Objektet kan jämföras med en behållare som innehåller medlemmar bl.a. medlemmen **`empNo`**. För att komma åt **`empNo`** måste vi först öppna behållaren. Sättet i koden att komma åt datamedlemmen **`empNo`** är att *först* skriva objektets referens, sedan en punkt och sist medlemmens namn. Samma sak gäller för de andra datamedlemmarna **`firstName`**, **`lastname`** och **`salary`**. Punktnotation förutsätter förstås objektets och referensens existens dvs kan endast användas efter att objektet skapats med **`new`**:

Klass referens **`= new`** *Klass`();`*

Då ser punktnotation ut så här för åtkomst till objektets datamedlem:

referens`.`*datamedlem*

Till vänster om punkten måste alltid finnas namnet på en referens till ett objekt och till höger någon datamedlem tillhörande *detta* objekt. Punktnotation skrivs för att *referera* till just detta objekts datamedlem och kan därför användas antingen för att tilldela den ett värde (skriva till minnescellen) eller för att hämta värdet (läsa från minnescellen). Satserna ovan kan även ersättas av en enda:

`new` *Klass*`().`*datamedlem*

Då har man skapat ett anonym objekt utan referens och kan därför inte heller referera till det efteråt.

Anrop av metoder med punktnotation

Samma teknik används i princip på ett objekts metoder. När ett objekt av typ **`Emp`** skapats kan vi anropa dess metod **`AsString()`** även med punktnotation. Skillnaden är bara att efter punkten skrivs ett vanligt anrop av metoden istället för datamedlemmen:

```
a.AsString()
```

Metoden **`AsString()`** anropas i objektet som **`a`** refererar till enligt deklarationen i klassen **`Emp`**. Eftersom **`AsString()`** är en metod inkapslad i en klass och inte fritt tillgänglig, måste man först (*före* punkten) referera till objektet för att sedan (*efter* punkten) kunna anropa metoden i detta objekt. Generellt har anropet av en metod i ett objekt som redan skapats, en syntax som liknar den för åtkomst av datamedlemmar:

referens`.`*metodanrop*

Till skillnad från datamedlemmar allokerar metoder inte minnesutrymme i objektet. När objektet skapas allokeras minne endast för datamedlemmar, inte för metoder. De är bara *deklarerade* i klassen och deklarationen skapar inget minne. Först när metoden anropas, allokeras minne åt de parametrar och variabler som är involverade i metoden. Men detta sker inte i objektet utan i det program som anropar metoden. En närmare titt på metoden **`AsString()`** i klassen **`Emp`** (sid 85) visar att den varken har parametrar eller lokala variabler. Men den involverar klassens alla datamedlemmar som vid anropet tas från objektet. Därför säger vi att metoden **`AsString()`** anropas *i* objektet som **`a`** refererar till och har därmed direkt tillgång till datamedlemmarna. Det är också därför de får skrivas i metoden **`AsString()`**:s kropp utan punktnotation. Båda befinner sig inuti objektet och har tillgång till varandra direkt. De är medlemmar i samma klubb – ”insiders” så att säga – och kan därför hälsa varandra utan att ange klubbens namn. Även om de hade förekommit i parameterlistan hade de angetts utan punktnotation. Punktnotation måste och får användas endast utanför objektet.

I programmet **`EmpTest`** anropas metoden **`AsString()`** två gånger, första gången i tilldelningssatsen **`output = a.AsString();`** andra gången sist i programmet inbakad i en utskriftsstas (sid 87). Anledning till dessa anropsmiljöer är att **`AsString()`** är en metod *med* returvärde. Därför måste ett meningsfullt anrop bakas in antingen i en tilldelnings- eller utskriftssats. Anropsmiljön måste ta hand om returvärdet *(Progr1, 6.2)*.

2.5 Klassens konstruktor

Objektorienterad programmering bygger på tre hörnstenar:

- Inkapsling
- Arv
- Polymorfism

Nu när vi inte bara lärt känna utan även sysslat en hel del med klasser och objekt kan vi gå vidare och börja med att stifta bekantskap med koncepten ovan. I detta avsnitt kommer vi att gå igenom det första: *Inkapsling*, medan *Arv* och *Polymorfism* kommer att i detalj behandlas senare (sid 108/113). Följande klass demonstrerar inkapsling genom att introducera privata datamedlemmar och klassens konstruktor:

```
// Circle.cs
// Deklarerar klassen Circle som inkapslar den privata data-
// medlemmen radius och kommer åt den via publika metoder
// En av dem är klassens konstruktor med parametern r
using System;

class Circle
{
   private double radius;          // Privat datamedlem

   public Circle(double r)         // Klassens konstruktor
   {                               // Publik metod med r som
      radius = r;                  // formell parameter
   }                               // Initierar datamedlemmen

   public double Area()
   {
      return Math.PI * radius * radius;
   }

   public double Circumference()
   {
      return 2 * Math.PI * radius;
   }
}
```

Åtkomstmodifieraren `private`

I objektorienterad programmering brukar man deklarera klassens datamedlemmar som **`private`** och klassens metoder som **`public`**. Tanken är att via klassens offentliga metoder kunna komma åt och styra de privata datamedlemmarna. På så sätt kan gradvis inkapsling uppnås. I klassen **`Circle`** deklareras datamedlemmen **`radius`** som **`private`** dvs kan endast nås från klassen. **`private`** är en åtkomstmodifierare i C# som spärrar åtkomsten till klassens medlemmar utifrån klassen. Den gäller endast för datamedlem-

mar och metoder, inte för klasser (förutom s.k. inre klasser), inte heller för lokala variabler. **`private`** spärrar strikt åtkomsten till datamedlemmar och metoder från andra klasser, vare sig dessa deklareras i samma fil eller ej, vare sig de ärver varandra eller ej. För att ändå kunna initiera den privatdeklarerade datamedlemmen **`radius`** utifrån med ett explicit värde, definieras en s.k. *konstruktor*, se nästa sida.

Det man vill åstadkomma med denna teknik är att kunna efterlikna verkligheten i sina datorprogram så mycket som möjligt. I verkligheten är det självklart att vissa egenskaper hos objekt är eller ska vara ”hemliga”. T.ex. vem känner till en persons religion eller politiska inställning när man ser personen? Allt man kan se, är personens offentliga egenskaper, utseendet, hårfärgen, storleken, klädseln osv. Allt annat är okänt – så länge man inte ställer frågor. Och även då är det upp till personen att svara, inte svara eller svara delvis, tala sanning eller ljuga. Egenskaperna kan jämföras med klassens datamedlemmar. Att ”ställa frågor” kan jämföras med att anropa klassens metoder. Offentliga metoder används för att via dem kunna efterfråga de privata datamedlemmarna.

Inkapsling innebär att deklarera klassens datamedlemmar som **`private`**
för att spärra åtkomsten till dem från andra klasser.
I objektorienterad programmering brukar man deklarera datamedlemmarna som **`private`** och metoderna som **`public`**.
Både **`private`** och **`public`** kallas för åtkomstmodifierare.

Observera att detta inte är någon absolut regel utan en attityd att jobba med klasser i alla objektorienterade språk. Det finns säkert i många specialfall skäl nog att använda inkapsling även på andra sätt. Men gör man det som beskrivet ovan, bildar datamedlemmarna klassens *kärna* som är skyddad mot direkta oönskade tillgrepp vare sig från andra program eller även andra programmerare. Metoderna däremot kan tänkas som ett skal kring kärnan som är till för att hantera klassens datamedlemmar. Man pratar om att metoderna bildar klassens *gränssnitt* mot användaren. Det är via dessa metoder man ska kunna kommunicera med den inkapslade kärnan. I så fall måste gränssnittet vara offentligt. Självklart kan man tänka sig även olika grader av inkapsling. Inte alla datamedlemmar måste vara privata. Lika bra kan det finnas skäl att även deklarera några metoder som privata. Vissa applikationer kräver kanske mer, andra mindre inkapsling. Detta är av betydelse med tanke på att inkapsling alltid innebär en viss overhead dvs mer programmeringsarbete. På vilket sätt ska vi diskutera nu:

Ett problem som generellt uppstår när man arbetar med klasser som har privata datamedlemmar är: Hur ska dessa datamedlemmar initieras när de är oåtkomliga? Svaret ligger i det offentliga gränssnittet. Man utnyttjar publika metoder för att initiera klassens privata datamedlemmar. Initieringsproblematiken är redan viktig för enkla variabler och därför ännu viktigare för objekt. Dessutom är den så generell – den dyker upp i alla objektorienterade program – att man i C# har konstruerat ett automatiskt verktyg som kallas klassens *konsruktor*. ”Automatiskt” därför att den alltid finns med i varje C#-klass, vare sig vi definierar den själva eller kompilatorn gör det åt oss by default.

Konstruktorns egenskaper

Som namnet antyder är konstruktorn en byggare, närmare bestämt en *objektbyggare*. I klassens deklaration kan man definiera en egen konstruktor som en av klassens publika metoder. Den anropas automatiskt när man skapar ett objekt av klassen. Konstruktorns uppgift är att initiera objektets datamedlemmar. Det speciella som skiljer konstruktorn från klassens alla andra metoder kan beskrivas med följande tre egenskaper:

1. **Namnet** är inte fritt väljbart. Konstruktorn och klassen måste ha samma namn. Om man själv definierar konstruktorn har man inget val än att ge konstruktorn samma namn som klassen.
2. **Returtypen** saknas. Konstruktorns definition får inte börja som hos alla andra metoder med en returtyp. För det första *kan* en konstruktor inte returnera ett värde. För det andra *får* den inte ens ha returtypen **`void`** framför sitt namn som alla andra **`void`**-metoder.
3. **Anropet** av konstruktorn sker i samma sats som objektet skapas. För att initiera objektets datamedlemmar anropas konstruktorn samtidigt som objektet skapas. Man kan varken skapa ett objekt utan att anropa konstruktorn eller anropa konstruktorn utan att skapa ett objekt.

De två första egenskaperna måste beaktas när man definierar en konstruktor i klassen. Den tredje egenskapen måste tillämpas när man utanför klassen anropar konstruktorn och samtidigt skapar ett objekt. Med konstruktorn erbjuds en bekväm möjlighet att förhindra oinitierade datamedlemmar dvs allokera minne åt dem utan att tilldela dem värden. Därmed minskas risken för icke-väl definierade objekt.

Klassen **`Circle`** har en egendefinierad konstruktor som är framhävd med vit bakgrund (sid 91). Med den vill vi testa egenskaperna **1-3** ovan. Som man ser är de två första egenskaperna givna: konstruktornamnet **`Circle`** = klassnamnet och konstruktorn har ingen returtyp, inte ens **`void`**, vilket gör att både kompilatorn och vi kan känna igen **`Circle()`** som konstruktor och kan skilja den från klassens andra metoder **`Area()`** och **`Circumference()`**. Varje försök att sätta en datatyp eller **`void`** framför metodnamnet kommer att leda till kompileringsfel. Den tredje egenskapen kan vi se när ett objekt av typ **`Circle`** skapas vilket görs i programmet **`Encapsulation`** på nästa sida.

Konstruktorn **`Circle()`** har en formell parameter **`r`** av typ **`double`**. Den gör i kroppen inget annat än att vidarebefordra parametern **`r`**:s värde till klassens privata datamedlem **`radius`**. Vid anrop av konstruktorn i programmet **`Encapsulation`**:s metod **`Main()`** på nästa sida överförs (kopieras) värdet av den aktuella parametern **`input`** till den formella parametern **`r`**.

Observera att både konstruktorn **`Circle()`** och metoderna **`Area()`** och **`Circumference()`** refererar till datamedlemmen **`radius`** *utan* punktnotation. Orsaken till att de inte behöver punktnotation är att de refererar inifrån klassen där det inte kan råda någon tvivel om att vilken datamedlem som är menad. Alla involverade variabler och metoder är medlemmar i en och samma klass och kan referera till varandra utan punktnotation. Refererar man däremot till ett speciellt *objekts* medlemmar vare sig i eller *utanför* klassen måste punktnotation användas.

```
// Encapsulation.cs
// Skapar ett objekt av typ Circle och anropar konstruktorn med
// en parameter vars värde läses in för att via konstruktorn
// initiera Circle-objektets privata datamedlem
using System ;

class Encapsulation
{
   static void Main()
   {
     Console.Write("\n\tMata in radien till en cirkel: ");
     double input = Convert.ToDouble(Console.ReadLine());

     Circle c;                          // Referensvariabel
     c = new Circle(input);             // Ett objekt skapas och
                                        // konstruktorn anropas
                                        // som initierar radius
                                        // till inputs värde
     // c.radius = input;               // Ger kompileringsfel
                                        // pga radius privat
     Console.WriteLine("\n\tCirkeln med radien "  + input    +
                 " har\n\n\tarean\t\t" + c.Area() + "\n\n\t" +
                 "och omkretsen\t" + c.Circumference() + '\n');
   }
}
```

En körning ger följande utskrift:

```
       Mata in radien till en cirkel: 1

       Cirkeln med radien 1 har

       arean           3,14159265358979

       och omkretsen   6,28318530717959
```

Programmet **`Encapsulation`** testar konstruktorns tredje egenskap (sid 93) genom att anropa konstruktorn i samma sats som ett objekt skapas:

```
Circle c;
c = new Circle(input);
```

Den första satsen skapar en referensvariabel **`c`** till av typ **`Circle`**.

Den andra satsen skapar ett objekt av typ **Circle** och anropar konstruktorn **Circle()** med den aktuella parametern **input**. Det nyskapade objektet initieras till värdet av den inlästa variabeln **input**. Definition av objektet och anrop av konstruktorn kan inte separeras utan måste ske i *en och samma* sats – allt enligt konstruktorns tredje egenskap. Sist tilldelas referensvariabeln **c** det nya objektet.

I sin struktur liknar satsen ovan det som vi alltid gör för enkla datatyper, nämligen:

```
int number = 5;
```

Denna sats definierar **number** som en variabel av typ **int** *och* initierar den samtidigt. Så gör vi också nu: Vi definierar ett objekt av typ **Circle** och *initierar den samtidigt*. Skillnaden är bara att objektet inte har något namn utan en referens som också måste skapas eftersom datatypen inte längre är en fördefinierad enkel datatyp – som i fallet av **int** – utan en egendefinierad klass. Jämförelsen visar än en gång att objektorienterad programmering är en naturlig och logisk fortsättning på traditionell programmering.

När konstruktorn i satsen ovan anropas med **Circle(input)** skickar den den aktuella parametern **input** till den formella parametern **r** i objektet där den tilldelas till objektets datamedlem **radius**, se konstruktorns definition i klassen **Circle** (sid 91). På så sätt blir **radius** initierad fast den är **private**. Konstruktorn gör det möjligt att *indirekt* initiera den privata datamedlemmen. Varje försök att initiera den *direkt* – ja överhuvudtaget att referera till den med punktnotation – kommer att leda till kompileringsfel. Detta försök finns som kommentar i programmet **Encapsulation**. Testa!

Programmet **Encapsulation** har vissa *begränsningar*: I utskriftssatsen har vi hämtat endast värdena till area och omkrets från objektet genom att anropa de resp. metoderna med punktnotation. Det var möjligt eftersom de var offentliga. Värdet till **radius** kunde vi inte hämta från objektet utan från inmatningen med hjälp av den lokala variabeln **input**, eftersom **radius** är privat. Därför kan vi inte komma åt den i **Main()**. Konstruktorn tillåter bara *initiering*, den skickar endast en första gång ett initialvärde till objektet. Vad som händer *efteråt* har konstruktorn ingen möjlighet att påverka. Det behövs andra offentliga metoder som tar hand om att hämta ut *(exportera)* privata datamedlemmar. Även om vi vill *ändra* privata datamedlemmarnas värden *efter* initieringen behöver vi speciella offentliga metoder. Med en *exportmetod* hade vi kunnat hämta ut värdet till **radius** från **Circle**-objektet efter att ha initierat den med konstruktorn. Frågan om hur sådana problem löses diskuteras i kapitlets kommande avsnitt.

Default konstruktorn

Experiment:

1. Kommentera bort hela konstruktorns definition i klassen **Circle** (koden som är inramad och framhävd med vit bakgrund, sid 91).
2. Ersätt i klassen **Encapsulation** konstruktorns anrop:

```
c = new Circle(input);
```

   med:

```
c = new Circle();
```

Detta ersätter den egendefinierade konstruktorn med parametern **`input`** med en *annan* konstruktor nämligen **`Circle()`** *utan* parameter. Men har vi definierat en sådan i klassen **`Circle`**? Självklart inte! Slutsats: Den är automatiskt definierad – by default – vilket bekräftas av en testkörning av programmet **`Encapsulation`** efter ändringen ovan:

```
Mata in radien till en cirkel: 1
Cirkeln med radien 1 har
arean           0,000000
och omkretsen   0,000000
```

För att förstå detta resultat måste vi förstå *default konstruktorn*:

> En default konstruktor är en konstruktor utan parametrar som automatiskt definieras när man skapar en klass. Default konstruktorn initierar klassens alla datamedlemmar till defaultvärden.

Därför finns den alltid där i bakgrunden, I vårt exempel ser den ut så här:

```
Circle()
{
   radius = 0;
}
```

Default-värdet till en **`float`** är **`0`**. Definierar man ingen egen konstruktor i sin klass, blir den ”osynliga” default konstruktorn automatiskt klassens konstruktor. Skriver man däremot sin egen konstruktor sätts default konstruktorn ur funktion. I klassen **`Circle`**:s ursprungliga version har vi definierat en egen konstruktor. Om vi nu aktiverar den och försöker samtidigt att skapa ett objekt av den med **`c = new Circle();`** kommer vi att få kompileringsfelet *There is no argument given that corresponds to the required formal parameter 'r' of 'Circle.Circle(double)'*. Det som sker är att vi samtidigt som vi definierat en egen konstruktor, anropar default konstruktorn. Men kompilatorn hittar den inte eftersom vi har satt den ur spel genom att explicit definiera vår egen konstruktor. Kommenterar vi däremot bort den egendefinierade konstruktorn i klassen **`Circle`**, går det alldeles utmärkt att kompilera och köra. Men att resultatet blir som ovan dvs med värdet **`0`** för arean och omkretsen, beror på att default konstruktorn – som nu aktiveras automatiskt – nollsätter cirkelns **`radius`** så att inmatningen **`1`** till **`input`** inte förs vidare via **`r`** till **`radius`**. Man kan säga att vi i alla program hittills, före behandlingen av konstruktorn, har anropat default konstruktorn varje gång vi skapat ett objekt. Självklart kan man, om man vill, definiera i sina klasser även en egen konstruktor utan parameter som i dess kropp initierar datamedlemmarna till andra än defaultvärden. Lika bra kan man definiera en konstruktor utan parameter som initierar datamedlemmarna till defaultvärden – en slags simulering av default konstruktorn för att testa dess egenskaper.

Flera konstruktorer

En klass kan ha flera konstruktorer som kan användas för att skapa objekt med olika initieringar. Följande klass innehåller tre datamedlemmar och två konstruktorer, en av dem utan parameter med en kropp som simulerar default konstruktorn, den andra med lika många parametrar som klassen har datamedlemmar. Denna kan användas för att initiera ett objekts datamedlemmar med vilka värden som helst som skickas vid anrop:

```
// AccountD.cs
// Klass med två konstruktorer, en av dem en simulerad default-
// konstruktor, den andra med tre parametrar
using System;

class AccountD
{
   int    accountNo;
   String accountName;
   double balance;

   public AccountD()           // Simulerar default konstruktorn
   {                           // Så här skulle den se ut
      accountNo   = 0;         // Den är gömd men kan även skrivas
      accountName = "";
      balance     = 0.0;
   }

   public AccountD(int aNo, String aName, double b)
   {
      accountNo   = aNo;       // En andra konstruktor
      accountName = aName;
      balance     = b;
   }

   public String AsString()  // Strängrepresentation av AccountD-
   {                                                    // objekt
      return "\tKontonr     " + accountNo   + '\n' +
             "\tKontonamn   " + accountName + '\n' +
             "\tSaldo       " + balance     + '\n' ;
   }
}
```

Det är ganska vanligt med flera konstruktorer. Anledningen är att man vill ha möjligheten att initiera sina objekt på olika sätt i olika sammanhang. Man vill inte begränsa sig på endast *ett* sätt att konstruera objekt. Men pga konstruktoregenskapen ”konstruktornamn = klassnamn” måste alla konstruktorer i en klass ha samma namn. Eftersom konstruktorer är speciella metoder, blir det flera metoder med samma namn. Det programmeringstekniska koncept som gör detta möjligt, är *överlagring av metoder* som vi kommer att behandla i detalj senare (sid 173). Kort sagt, innebär överlagring av metoder att ha samma namn på *olika* metoder i en och samma klass, men skilja dem genom olika parameterlistor. Därför kan vi ha flera konstruktorer i en klass, bara vi förser dem med olika parameterlistor. I klassen **`AccountD`** har den första konstruktorn ingen parameter, den andra har tre parametrar. De skulle kunna ha även *lika många* parametrar, men då

måste datatypen till minst en av parametrarna vara olika. Flera konstruktorer är en av de viktigaste tillämpningarna av överlagring. Klassen **AccountD** testas i följande program:

```
// CreateAccountD.cs
// Anropar en simulerad default konstruktor i samma sats som
// ett objekt skapas och skriver ut defaultvärdena
// Skapar nytt objekt med annan konstruktor och skriver ut de
// nya värdena. Ompekning av referensvariabeln till nya objektet
// Garbage collector dödar automatiskt det orefererade objektet
using System;

class CreateAccountD
{
   static void Main()
   {
     AccountD myAccount = new AccountD();  // Anrop av simulerad
                                           // default konstruktor
     Console.WriteLine("\n\tDefaultvärden:\n" +
                       myAccount.AsString());

     myAccount =                           // Ompekning till
                                           // nytt objekt
     new AccountD(123456, "Kalle", 100);  // Anrop av den andra
                                           // konstruktorn
     Console.WriteLine("\tNya värden:\n" + myAccount.AsString());
   }
}
```

Programmet ovan vars körexempel kan beskådas på nästa sida, demonstrerar användningen av flera konstruktorer i en klass. Två objekt av klassen **AccountD** skapas där: Det första initieras till defaultvärden genom anrop av den simulerade default konstruktorn utan parameter (framhävd med vit bakgrund). Det andra objektet initieras till nya värden som skickas vid anropet av den andra konstruktorn som har tre parametrar (även det framhävd med vit bakgrund). Utskriften bekräftar detta. Vi vet att datamedlemmen **accountName**:s datatyp är **String**. C# tolkar dock **accountName** inte som en sträng utan som en *referens* till ett **String**-objekt, därför att **String** är en klass. Referensernas defaultvärde är **null**, se sid 102.

```
        Defaultvärden:
        Kontonr     0
        Kontonamn
        Saldo       0

        Nya värden:
        Kontonr     123456
        Kontonamn   Kalle
        Saldo       100
```

En annan intressant observation som dock tyvärr utskriften inte visar, är att objekten i programmet **CreateAccountD** lagras vid två *olika* adresser fast vi refererar till dem i

båda fall med en och samma referensvariabel **`myAccount`**. Att de två objekten lagras vid två olika adresser, är inte konstigt därför att var och ett skapas med en **`new`**-sats och varje **`new`** genererar en annan adress. Men vi har använt för båda objekt samma referensvariabel **`myAccount`**. Dvs det har skett en ompekning: Först lagrar **`myAccount`** det första objektets adress, men sedan överskrivs den av det andra objektets adress. En konsekvens av denna ompekning är att det första objektet tappat sin referens, dvs den kan inte längre nås. Det som sker i sådana fall är att C#:s s.k. *garbage collector* automatiskt rensar den från minnet. Så det behövs ingen speciell åtgärd från programmet (som *destructor* i C++) att ta bort oanvända eller orefererade objekt.

2.6 Referensvariabler

En *referensvariabel* – kort *referens* – är en ny typ av variabel:

> Referens är en variabel vars datatyp är en klass.

Ett exempel är variabeln **a** i programmet **EmpTest** (sid 87) där **a** deklareras till **Emp** som är en klass: **Emp a;** Med satsen **Emp a;** skapas inget objekt utan endast en referens till ett objekt. Självaste objektet skapas med koden **new Emp()**.

Den allmänna formen *hur* referensvariabler kopplas till objekt, kan beskrivas så här:

> Klass referensvariabel = **new** Klass();

Exempel med klassen **Emp**: **Emp a = new Emp();**

Själva objektet skapas med **new** höger om tilldelningstecknet. Till vänster deklareras referensvariabeln **a**, som tilldelas objekteta adress. Här ser man också att operatorn **new** inte följs av parenteser. Klasserna på tilldelningens båda sidor måste vara identiska: **new Emp()** allokerar minne för lagring av ett **Emp**-objekt och returnerar minnets adress till referensen **a** av samma datatyp (klass) **Emp**. Tilldelning till en referens av en annan typ (klass) skulle ge kompileringsfel.

Referensvariabler är ett verktyg för att kunna komma åt objekt. Objekten själva kan endast skapas med **new** som är en *minnesallokeringsoperator*. Allokering betyder reservering av minnesutrymme. Objektens namn är inga vanliga variabler av enkel datatyp, utan referensvariabler. Man kan jämföra detta med tyglar till en häst, där tyglar är referensen och hästen objektet. Eller fjärrkontrollen (referens) till en TV (objekt). Båda är lätthanterade verktyg för styrning av tunga objekt. Andra jämförelser länkar till webbsidor eller namnskyltar: Den lilla skylten *Gamla Stan* (referens) pekar på den stora ön *Gamla Stan* (objekt). En referens lagrar nämligen minnesadressen till ett objekt och tar jämfört med det tunga objektet så litet minne som en vanlig **int**: 4 bytes, vilket minnesekonomiskt innebär en stor effektivitet vid exekvering av minneskrävande program.

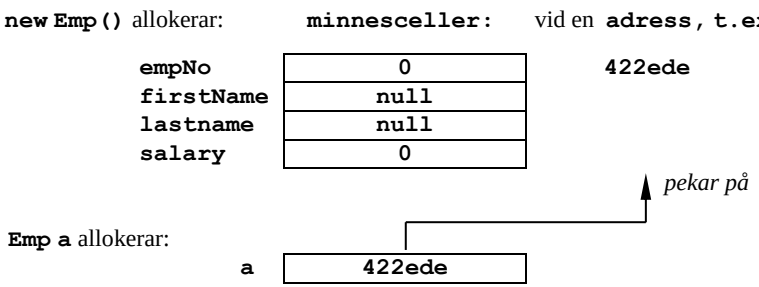
När det gäller vanliga variabler av enkel datatyp hänvisar vi till minnescellerna med *variabelnamn*. När det gäller objekt gör vi det med deras *adresser* i form av objektens referenser. När man vant sig vid att använda referenser kan man t.o.m. tycka att hanteringen av data via adresser är det naturliga sättet, vilket inte är någon dum idé med tanke på att variabelnamn ändå är en slags mjukvarulänk till hårdvarans minnesadress.

Man kan, om man inte har behov av att komma åt objektet senare, även skapa s.k. *anonyma objekt* direkt, när man behöver dem, t.ex.: **new Emp().AsString()** där metoden **AsString()** anropas i objektet **new Emp()**. Testa gärna själv att i programmet **EmpTest** (sid 87) byta ut **a.AsString()** mot **new Emp().AsString()**.

De tomma parenteserna efter klassnamnet i exemplet **`Emp()`** och även i den allmänna formuleringen Klass() får absolut inte utelämnas även om de är tomma. De anropar klassens konstruktor (sid 91), närmare bestämt default konstruktorn (sid 95).

Referensen ”pekar” på objektet

Vad händer exakt när satsen **`Emp a = new Emp();`** exekveras i programmet **`EmpTest`**? För det första definieras referensen **`a`**, för det andra skapar **`new`** ett **`Emp`**-objekt dvs allokerar minnesutrymme för objektet. För det tredje tilldelas minnesadressen till referensen **`a`**. Adressen tilldelas referensen **`a`** vilket gör att **`a`** nu *pekar på* den av **`new`** allokerade minnescellen, så att minnesbilden i datorns RAM efter satsen ovan ser ut så här:



Tilldelningsoperatorn mellan **`Emp a`** och **`new Emp()`** gör att objektets adress hamnar i referensvariabeln **`a`**:s minnescell vilket resulterar i att **`a`** *pekar på* objektet och vi därför kan och måste referera till objektet genom att använda refrensen **`a`**. Alla objekt i C# kan endast hanteras med referenser. Det är avgörande att inte förväxla *objekt* med *referens*: Det är två helt olika typer av saker och ting med två olika minnesplatser och olika egenskaper som relateras till varandra på det beskrivna sättet. Men varför hamnar **`0`** och **`null`** i objektets minnesceller? Och vad är överhuvudtaget **`null`**? Att de hamnar där beror på anropet av default konstruktorn **`Emp()`** som automatiskt initierar datamedlemmarna **`empNo`**, **`firstName`**, **`lastname`** och **`salary`** till s.k. defaultvärden.

Automatisk initiering av datamedlemmar

Samtidigt som **`new`** allokerar minne för objektet anropar koddelen **`Emp()`** en metod som initierar objektets datamedlemmar med vissa defaultvärden som beror på deras datatyper. Denna metod som heter klassens *konstruktor* har samma namn som klassen samt den viktiga uppgiften att initiera objektets datamedlemmar. En s.k. *default konstruktor* skapas alltid automatiskt med när man deklarerar en klass utan att själv skriva en konstruktor. Vi kommer att behandla konstruktorer senare i detalj. Default konstruktorn initierar datamedlemmarna, när ett objekt skapas, automatiskt till följande defaultvärden:

`0`	om de är tal,
`null`	om de är referenser,
nolltecknet	om de är tecken,
`false`	om de är av typ **`bool`**.

Exempel på `0` och `null` har vi i programmet `EmpTest`: datamedlemmen `a.firstName` initieras till `null` eftersom den är av typ referens, `a.empNo` initieras till `0` eftersom den är av typ `int` och `a.salary` som är en `float` får `0` som initieralvärde. Utskriften i konsolfönstret ovan visar dessa värden. Observera att vi får dessa defaultvärden utskrivna eftersom vi i programmet `EmpTest` placerade `Console.WriteLine()`-satsen direkt efter objektets definition av och *före* den nya tilldelningen av datamedlemmarna. Källan till datatypinformationen är förstås klassen `Emp` där dessa datamedlemmar är deklarerade till sina resp. datatyper (sid 85). Alla objekt är ju kopior av klassen. Vi hittar där bl.a. deklarationen `String firstName;` men hävdar ändå att datamedlemmen `a.firstName` inte är en sträng utan en *referens* till strängar. Anledningen är att `String` är en klass och inte en enkel datatyp som `int` och `float`. Men har vi inte lärt oss tidigare att `String` är en datatyp och skrivs så här: `string`? Jo, det har vi och det gäller fortfarande: `string` är en sammansatt datatyp och samtidigt en klass, närmare bestämt ett alias till klassen `String`, precis som de enkla datatyperna är alias till sina resp. klasser. Att den tomma platsen efter `Referens:` i körexemplet av programmet `EmpTest` (förförra sidan) är det osynliga tecknet `null` och inte den tomma strängen, kan testas om man lägger in följande rader i programmet mellan utskriften av defaultvärdena och tilldelningen av nya värden till objektet `a`:s datamedlemmar:

```
if (a.firstName == null) Console.WriteLine("null");
if (a.firstName == "")   Console.WriteLine("tom sträng");
```

Referensen `null`

`null` i C# betyder *inget objekt*, dvs en referens som inte pekar på något objekt. Själva ordet `null` hittar man bland språkets reserverade ord *(Progr1+, 2.3)*. `null` är ett värde som kan tilldelas referensvariabler, nämligen när referensen inte lagrar någon adress till ett objekt. Just därför kallas `null` för referensernas *defaultvärde*. Även om `null` representeras i datorn med `0`, får det inte förväxlas varken med *talet* `0` eller med *tecknet* `'0'`. `null`:s datatyp är varken `int` eller `char`, utan `null` är av referenstyp, dvs ett värde som endast kan tilldelas referenser. Och vi vet ju att referensvariablernas datatyper alltid är klasser. En variabel av referenstyp kan endast lagra minnesadresser.

> En referens med värdet `null` pekar på inget objekt.
> I C# är `null` referensvariablernas defaultvärde.

`null`-referenser kan jämföras med *parkerade* bilar, om bilar *i fart* jämförs med referenser som pekar på objekt. Man kan ”sätta igång” `null`-referenser när som helst genom att tilldela objektadresser till dem. Omvänt kan man ”parkera” dem igen genom att tilldela `null` till dem. Observera att `null`-referenser inte alls är samma sak som oinitierade referenser. Till skillnad från `null`-referenser leder oinitierade referenser precis som alla andra oinitierade variabler till kompileringsfel när de används. Till skillnad från oinitierade referenser *har* `null`-referenser ett värde, bara att deras värde `null` inte är en adress till ett befintligt objekt utan bara en symbol som betyder ”referens i väntan på att få en objektadress” precis som en parkerad bil i väntan på att sättas i fart. Man använder `null`-referenser i C#-program för att initiera referensvariabler direkt efter deklarationen

när det vid deklarationens tidpunkt inte kan avgöras vilket objekt de ska bindas till. På så sätt vill man förhindra oinitierade referenser som alltid bär risken med sig att de används av misstag innan de binds till ett objekt. Det är rekommenderad att alltid initiera sina lokala referensvariabler till **`null`** om de inte kan tilldelas ett objekt när de skapas. Förväxla inte **`null`** med nolltecknet:

Nolltecknet

I listan över defaultvärden till de olika datatyperna på förra sidan dyker upp *nolltecknet* som defaultvärdet för teckenvariabler dvs till datatypen **`char`**. Man stöter på det när man försöker skriva ut datatypen **`char`**:s undre gräns och använder sig av explicit typkonvertering för att omvandla den till ASCII-koden 0. Sedan kan man framställa det oskrivbara och osynliga nolltecknet med hjälp av escapesekvensen **`'\0'`**. Då känns det naturligt att det allra första tecknet i ASCII-tabellen med koden 0 används som defaultvärde för teckenvariabler. Fysiskt består det alltså av av 2 bytes dvs 16 bitar fyllda med endast nollor. På den fysiska bitnivån representeras även **`null`** av nollor. Däremot skiljer de sig på den logiska programnivån via sina datatyper: Medan nolltecknet är av typ **`char`** är **`null`** av typ referens.

Nolltecknet är i C# **`char`**-variablernas defaultvärde med ASCII-koden 0.

2.7 Komposition

Komposition betyder sammansättning och är relaterad till modularisering, Lego-principen och den diskussion vi hade om att bygga program med hjälp av redan skrivna och testade moduler dvs *klasser* som kan ingå som komponenter i andra klasser. Den övergripande strukturen av ett C#-program är fortfarande en samling av klasser som i sin tur innehåller datamedlemmar och metoder. Objektorienterade program har för det mesta bara **`Main()`**-metoden kvar och resten är klasser i vilka man definierar och anropar sina metoder. Komposition är sammansättning av ett objekt med ett annat objekt. Tänk på en bil som har en motor. Man sätter ihop bilen som ett objekt av klassen *Bil* genom att bygga in i den bl.a. en motor som i sin tur är ett objekt av en annan klass, klassen *Motor*.

En bil *har* en motor. En anställd *har* arbetstider. En sådan relation mellan två begrepp kallas i objektorienterad design för en ”*har*”-relation och är den grundläggande förutsättningen för komposition av klasser. Om två begrepp står i en ”*har*”-relation till varandra kan man bygga det ena (stora) med hjälp av det andra (lilla). Ett hus har en dörr och andra komponenter. En cykel har hjul. En bil har en motor, en motor har i sin tur cylindrar osv. I praktiken bygger man också alla dessa enkla komponenter separat först och sammansätter dem sedan till det mer komplexa objektet.

En annan viktig relation mellan objekt i den reala världen kallas i objektorienterad design för ”*är*”-relation och måste begreppsmässigt noggrant skiljas från ”*har*”-relationen. Båda är relevanta klassificeringsverktyg vid modellering och design av en verklig miljö. ”*Är*”-relationen är den grundläggande förutsättningen för *arv* hos klasser som efter *inkapsling* är den andra hörnstenen i objektorienterad programmering. Klasser kan ärva varandra om de står i en ”*är*”-relation till varandra. En lastbil *är* en bil, därför kan klassen lastbil ärva klassen bil dvs ta över bilens delar och metoder, modifiera och anpassa dem till lastbilen. *Komposition* är vid modellering ibland ett alternativ och ibland en konkurrent till *arv*. Vi behandlar först komposition. I nästa avsnitt tas upp arv.

Komposition av klasser

För att bättre kunna förstå skillnaden mellan komposition och arv vill vi i båda avsnitt behandla samma exempel, nämligen en anställd som förutom för- och efternamn, också har ett födelse- och ett anställningsdatum. Medan för- och efternamn är strängar och kan deklareras som sådana, har födelse- och anställningsdatum inte några fördefinierade typer. De är båda av typ *datum*, så vi måste först deklarera en sådan klass. Observera att datum och tid inte är samma sak, så vi kan inte använda klassen **`Time`** från tidigare. Medan tid är en varaktighet bestående av ett antal tidsenheter, ett intervall med en början och ett slut, är datum en viss tid*punkt*. En tid består av många tidpunkter. I praktiskt sammanhang är det i regel tillräckligt att modellera datum som en klass med datamedlemmarna dag, månad och år. I koden använder vi engelska beteckningar:

```
// Date.cs
// Deklarerar klassen Date med två konstruktorer (överlagring),
// en allmän konstruktor och en simulerad default konstruktor
// Metoden AsString() formaterar datum till en sträng
using System;

class Date
{
   int day, month, year;

   public Date(int d, int m, int y)
   {
      day   = d;                          // Allmän konstruktor
      month = m;
      year  = y;
   }

   public String AsString()               // Strängrepresentation
   {
      return year + "-" + month +         // Svenskt datumformat
                  "-" + day;
   }
}
```

Datamedlemmarna i klassen **`Date`** är privata. Därför finns det en konstruktor som vi kommer att anropa t.ex. i klassen **`Composition`** med **`new Date(12, 10, 1969)`** för att skapa ett **`Date`**-objekt och initiera det till **`1969-10-12`**, födelsedatumet till en antälld.

Set-metoder behöver vi inte i **`Date`** därför att vi i vårt exempel inte kommer att ha något behov för ändringar av varken födelse- eller anställningsdatum. För andra ändamål där det behövs kan man lätt komplettera klassen med Set-metoder. Vad gäller Get-metoder ersätter metoden **`AsString()`** alla sådana när den konkataenerar alla tre datamedlemmar och representerar datum som en sträng, dessutom i svenskt datumformat.

Nu, när vi har klassen **`Date`** till förfogande, kan vi använda den i följande klass för att deklarera en anställds födelse- och anställningsdatum med den nya datatypen **`Date`**:

```
// Employ.cs
// Komposition av klasser: Klassen Employ sätts ihop (komponeras)
// bl.a. med hjälp av klassen Date
// Mellan klasserna Employ och Date finns en "har"-relation:
// Employ "har" två Dates som datamedlemmar
using System;

class Employ
{
   String firstName, lastname;
   Date birthDate;                       // Komposition
   Date hireDate;
```

```
   public Employ(String f, String l, Date b, Date h)
   {
      firstName = f;                    // Konstruktorn
      lastname  = l;
      birthDate = b;
      hireDate  = h;
   }

   public String AsString()             // Objektens
   {                                    // strängrepre-
      return "\n\tDen anställde " +     // sentation
             firstName + " " + lastname;
   }
}
```

I klassen **Employ** har en anställd ett för- och efternamn som båda är av typ **String**. Faktiskt är även **String** en klass, även om en fördefinierad sådan, så att vi redan här har att göra med komposition. Sedan kommer den självgjorda kompositionen med klassen **Date**. En anställd har också ett födelse-och ett anställningsdatum, båda av typ **Date**. I koden utgörs denna ”*har*”-relation av deklarationen av datamedlemmarna **birthDate** och **hireDate** som **Date**-objekt (framhävd med vit bakgrund). Metoden **AsString()** returnerar en sträng som konkateneras med operatorn **+** .

Komposition av objekt

Nu har vi två klasser till förfogande – **Employ** och **Date** – där den ena är en komponent i den andra. Därmed kommer varje objekt av typ **Employ** att vara ett sammansatt objekt, sammansatt av två objekt av typ **String** och två objekt av typ **Date**. Som en konsekvens har även konstruktorn två **String**-objekt och två **Date**-objekt som parametrar:

```
public Employ(String f, String l, Date b, Date h)
```

Parametrarna *skapas* inte här som objekt utan *deklareras* endast här. Som komponenter (delobjekt) skapas de först när ett helt **Employ**-objekt skapas i följande testprogram:

```
// Composition.cs
// Komposition av objekt: Ett Employ-objekt byggs upp med hjälp
// av 2 Date-objekt: För att kunna skapa Employ-objektet måste
// först komponenterna av typ Date skapas och initieras med resp.
// konstruktor. Date-objektens referenser kan sedan skickas till
// konstruktorn Employ() för att initiera Employ-objektet
using System;

class Composition
{
   static void Main()
   {
```

```
      Date birthday = new Date(12, 10, 1969);
      Date hireday  = new Date(15, 11, 2001);

      Employ emp =
      new Employ("Kalle", "Karlsson", birthday, hireday);

      Console.WriteLine(emp.AsString() + " är född " +
          birthday.AsString() + "\n\n\toch har jobbat sedan " +
           hireday.AsString() + '\n' );
   }
}
```

Objekten **birthDate** och **hireDate** ingår som komponenter i objektet **Employ**. Därför måste de skapas *först*. Det gör vi genom att initiera dem med vissa datum och anropa **Date**-klassens allmänna konstruktor med 3 parametrar. Sedan skickas de som parametrar till **Employ**-konstruktorn när objektet **emp** skapas. Observera att **Date**-klassens default konstruktor inte anropas här. Den behövs bara i fall man i någon annan applikation vill skapa ett **Date**-objekt med default-initiering och tilldela det nya värden senare, t.ex. genom att läsa in vissa datum.

Slutligen får vi följande utskrift när vi kör **Composition**:

```
       Den anställde Kalle Karlsson är född 1969-10-12

       och har jobbat sedan 2001-11-15
```

Namnet skrivs ut med anrop av **emp.AsString()** som returnerar den konkatenerade strängen med för- och efternamn med mellanslag däremellan. Därför kan den integreras i utskiftssatsen. Datumen däremot skrivs ut med **Date**-klassens strängrepresentationsmetod som har samma namn. Det är alltså två olika metoder med sammma namn **AsString()**, definierade i två olika klasser.

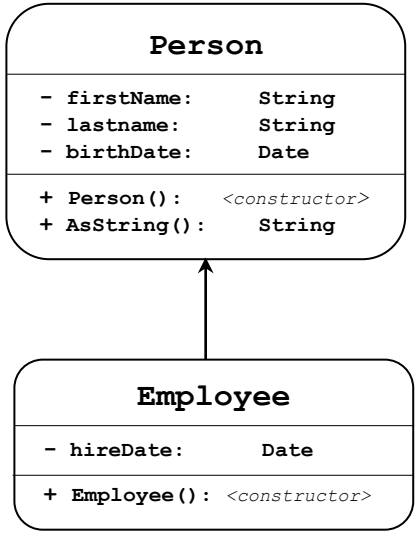
I nästa avsnitt kommer vi att vidareutveckla exemplet med anställda genom att flytta en del av koden från klassen **Employ** till en överordnad klass och etablera en arvrelation mellan dem. Komposition med klassen **Date** bibehålls så att resultatet blir en kombination av arv och komposition.

2.8 Arv

Arv är efter inkapsling den andra hörnstenen i objektorienterad programmering. Medan inkapsling har att göra med dataskydd och dataintegritet, är arv ett koncept som förverkligar *modularisering*, *återanvändning av kod* och *strukturering av program* – mål som är svårt att uppnå och som i praktiken ofta uppnås endast delvis. Man skapar en ny klass som en underkategori av en annan, redan befintlig klass. Man återanvänder den befintliga klassens kod i den nya klassen. Den nya klassen *ärver* den befintliga klassen.

Arv är en relation mellan två klasser. Ex.: Klassen **`Employee`** ärver klassen **`Person`**, eftersom en anställd är en person.

Subklassen **`Employee`** tar över all kod från *superklassen* **`Person`** och lägger till ny kod som specificerar en anställd.



I klassdiagrammet, till höger, ritas arvrelationen med en pil från subklassen **`Employee`** riktad uppåt mot superklassen **`Person`**.

”Är”-relationen

Man kan etablera en arvrelation mellan två begrepp, om de står i en en ”*är*”-relation till varandra: En anställd *är* en person. Därför kan en ny klass **`Employee`** ärva klassen **`Person`**. Då tar den över all kod som redan finns där och lägger till ny kod som är speciell för 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 datamedlemmar, och man lägger till den nya datamedlemmen **`hireDate`** som är speciell för en anställd. *Subklassen* **`Employee`** ärver *superklassen* **`Persons`** alla datamedlemmar och metoder.

Dock behöver en inte alla arvrelationer motiveras av en ”*är*”-relation. T.ex. kan en cylinder ärva en cirkel genom att *utvidga* den med en höjd, utan att behöva vara en cirkel.
Det som har hänt med **`Employee`**-exemplet jämfört med förra avsnitt är att en del av koden har flyttats från klassen **`Employee`** till klassen **`Person`**. Det som är specifik för en anställd, datamedlemmen **`hireDate`**, är kvar i **`Employee`**. Allt som är relevant för alla personer har flyttats till klassen **`Person`**. Arvrelationen garanterar att dessa data-

medlemmar och metoder kan nås även från ett **Employee**-objekt – självklart upp till åtkomstreglerna. Arv upphäver inte åtkomstmodifierarnas giltighet: En privat medlem är absolut oåtkomlig utifrån klassen, även från en subklass.

Observera att klassen **Date** är helt oberörd av denna omplacering av kod (arvkonstruktionen). Fortfarande ”har” en anställd ett anställningsdatum. En person ”har” ett födelsedatum. Detta är oberoende av att en anställd ”är” en person. Båda relationer förekommer parallellt. Därför har vi nu att göra med en kombination av komposition och arv. Komposition är något helt naturligt och ställer inga speciella krav på syntaxen, medan arv introducerar ny kod i C#. Frågan är: Hur ser syntaxen ut för pilen i klassdiagrammet ovan? Och hur påverkar arvrelationen konstruktorns kod speciellt i subklassen? För att få svar implementerar vi modellen ovan genom att börja med klassen **Person**. Observera att denna klass förutsätter att klassen **Date** från förra avsnitt (sid 105) redan är deklarerad innan och infogat i samma projekt i Visual Studio.

```
// Person.cs
// Deklarerar klassen Person som en bas- eller superklass till
// alla subklasser av Person, bl.a. Employee
using System;

class Person
{
   String firstName, lastname;
   Date   birthDate;

   public Person(String f, String l, Date b)  // Konstruktorn
   {
      firstName = f;
      lastname  = l;
      birthDate = b;
   }

   public String AsString()                   // Person som
   {                                          // sträng
      return "\n\t" + firstName + " " + lastname;
   }
}
```

Klassen **Person** kan användas som en överordnad kategori, kallad superklass, till klassen **Employee** då varje anställd ”är” en Person. Därför kan klassen **Employee** ärva klassen **Person** och bli subklass till den. **Person** kan även användas som superklass till andra subklasser, t.ex. **Elev**, ... Dessutom används komposition för att definiera klassen **Person** med bl.a. en datamedlem **birthDate** av typ **Date**: Varje **Person** ”har” ett födelsedatum.

Som man ser är klassen **Person** exakt samma som klassen **Employ** minus datamedlemmen **hireDate** (sid 105). Bara att den även fungerar nu som en superklass i den ovan beskrivna UML-modellen. Men av denna roll finns det inget spår i koden. Arvrelationen skrivs alltid in i subklassen, inte i superklassen. Klassen **Person** måste vara så generell

att den även kan användas i andra program som behöver en sådan klass. Det är ju just meningen med *återanvändning av kod*. Självklart kan man tänka sig en ännu mer generell version av klassen **Person** med fler medlemmar som t.ex. personnr, postadress, mailadress, telnr osv. Vi nöjer oss dock för enkelhetens skull med versionen ovan.

Arvrelationen

Som ett resultat av återanvändning av kod blir nu subklassen **Employee** mycket kort för det mesta är redan kodad i superklassen **Person**:

```
// Employee.cs
// Deklarerar Employee som en subklass till superklassen Person
// Alla datamedlemmar och metoder ärvs automatiskt från Person
// utom konstruktorn. En ny datamedlem hireDate tillkommer.
// Konstruktorn måste explicit ärva och anropa superklassens kon-
// struktor samt lägga till initieringen av den nya datamedlemmen
using System;

class Employee : Person            // Employee ärver Person
{
   Date hireDate;                  // Ny datamedlem

   public Employee(String f, String l, Date b, Date h)
                  : base(f, l, b) // Arv & anrop
                                  // Konstruktorn ärver och an-
                                  // ropar superklassens kon-
                                  // struktor explicit
                                  // base = referens till super-
                                  // eller basklassen Person
   {
      hireDate = h;               // Initiering av ny datamedlem
   }
}
```

Employee ärver **Person**: **first-** och **lastname** samt **birthDate** tas över från **Person**. En anställd ”är” en Person som dessutom har ett anställningsdatum. Därför lägger vi till den nya datamedlemmen **hireDate** till de ärvda datamedlemmarna. Konstruktorn måste nu initiera inte bara denna nya datamedlem utan även de som är ärvda. Men eftersom **Employee** ärver **Person**:s metoder, utom konstruktorn, måste vi explicit ärva och anropa **Person**-klassens konstruktor för initiering av de ärvda datamedlemmarna med tillägget **: base(firstName, lastname, birthDate)** i konstruktorns huvud där **base** är en *referens* till super- eller basklassen. Alltså är **base** en referens till **Person**. För att koppla ihop klasserna **Employee** och **Person** och etablera en arvrelation mellan dem måste alltså två saker göras:

1. **I klasshuvudet** måste tilläggas information om att en arvrelation ska etableras utifrån den här klassen (subklassen). Namnet på den klass som relationen ska kopplas till (superklassen) måste anges. Så här ser den allmänna syntaxen ut:

   class *subklass* **:** *superklass*

Detta innebär att *subklass* ärver *superklass*. **:** är i C# symbolen för ”arv”. Att *subklass* ärver *superklass* innebär att subklassen fortsätter att koda superklassen, fast i subklassen.

2. **I konstruktorns huvud** måste så många parametrar tas upp i parameterlistan som det finns privata datamedlemmar *både* i super- *och* subklassen. Det räcker inte bara med subklassens privata datamedlem. Man måste nämligen från ett **Employee**-objekt kunna initiera inte bara **hireDate**, utan även **firstName**, **lastname** och **birthDate**. Man måste kunna initiera ett *fullständigt* **Employee**-objekt med konstruktorn. Därför måste denna ha fyra parametrar:

```
public Employee(String f, String l, Date b, Date h)
```

De tre första parametrarna vidarebefordras till superklassens konstruktor med tillägget **: base (f, l, b)** till koden ovan vilket innebär både explicit arv och anrop av **Person**-konstruktorn. Slutligen initieras den fjärde parametern, datamedlemmen **hireDate**, i **Employee**-konstruktorns kropp.

Så här kan subklassen **Employee** testas. Observera att det inte finns ett spår kvar av superklassen **Person** fast hela dess kod används i programmet:

```
// Inheritance.cs
// Testar klassen Employee
// För att kunna skapa ett Employee-objekt skapas först två
// Date-objekt, ett födelse- och ett anställningsdatum vars
// referenser skickas till konstruktorn Employee() med 4 para-
// metrar, varav 3 vidarebefordras till superklassen Person
using System;

class Inheritance
{
   static void Main()
   {
      Date     birth = new Date(16, 6, 1978);
      Date     hire  = new Date(12, 3, 2001);
      Employee emp   = new Employee("Anders", "Larsson",
                                    birth, hire);
      Console.WriteLine(
             emp.AsString() + " är född "  + birth.AsString() +
             "\n\n\toch har jobbat sedan " +  hire.AsString() +
                                                    '\n');
   }
}
```

Programmet ovan testar klassen **Employee** och är, när det gäller koden, nästan identiskt med **Composition** (sid 106). Enda skillnaden är att klassen **Employ** (sid 105) har byts ut mot **Employee** som är helt annorlunda nu i och med att den tillämpar arv. T.ex. har den ingen metod **AsString()**. Ändå kan vi i utskriftssatsen anropa denna metod i objektet **emp** som är av typ **Employee**:

```
emp.AsString()
```

Kompilatorn tittar i klassen **`Employee`** och hittar där ingen metod **`AsString`**. Men eftersom **`Employee`** tillämpar arv och är subklass till **`Person`**, går kompilatorn ”upp” till superklassen och hittar där **`AsString`** som en metod i klassen **`Person`**. För första gången anropar vi en metod i ett objekt som inte är definierad i objektets klass utan i *superklassen* till objektets klass.

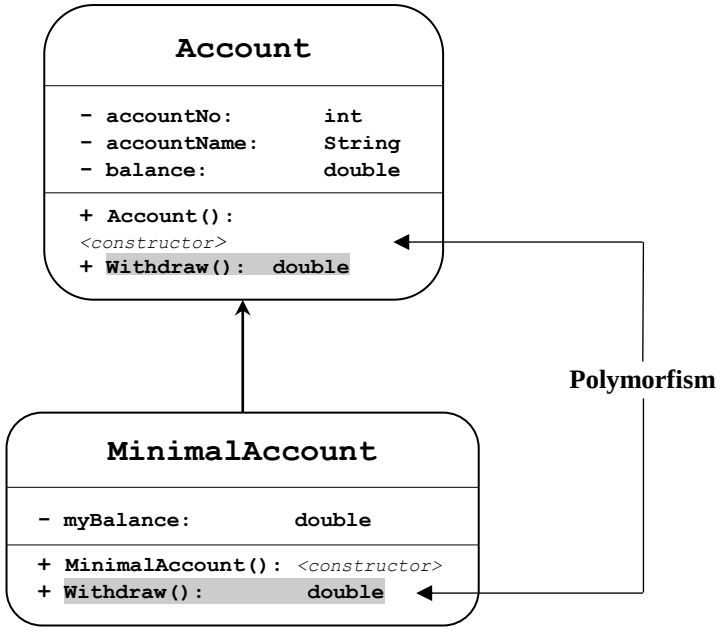
Att kompilatorn går ”upp” i klasshierarkin kan man ta som en analogi till att pilen i klassdiagrammet är riktad *uppåt* från subklassen till superklassen.

Programmet **`Inheritance`** producerar följande utskrift:

```
Anders Larsson är född 1978-6-16

och har jobbat sedan 2001-3-12
```

2.9 Polymorfism

I följande exempel ärver klassen **`MinimalAccount`** klassen **`Account`**:



Man har två olika typer av konto i en bank, ett vanligt konto och ett konto med begränsad uttagsrätt. Operationen ”att ta ut pengar” definieras på olika sätt i dessa två kontotyper, men operationens namn ska alltid vara **`Withdraw()`**. Metoden **`Withdraw()`** är definierad både i superklassen **`Account`** och i subklassen **`MinimalAccount`**. Utan arvrelation skulle detta varit ett exempel på vanlig *överlagring av metoder* (avsn. 3.7, sid 172). Men eftersom **`MinimalAccount`** ”är” ett speciellt **`Account`** kan **`MinimalAccount`** ärva klassen **`Account`**. Etablerar vi arvrelationen blir det *polymorfism*.

”Poly” betyder *många* och ”morf” betyder *form* på gammal grekiska. Polymorf är något som har många former, t.ex. ett ord som har olika betydelser. Det vanliga språket är fullt med sådana ord: Ta bara ordet *köra*. Man kan köra bil, köra tåg, köra program osv. Det är sammanhanget som avgör den aktuella betydelsen.

En metod beskriver en funktionalitet. Polymorfism definierar en metod i superklassen, definierar om funktionaliteten i subklassen, men behåller namnet (eng. *overriding*).

Polymorfism modifierar helt eller delvis funktionaliteten hos metoder med samma namn som förekommer i en arvhierarki.

Subklassens metod kommer då att *överskugga* superklassens metod.

Överlagring av metoder som togs upp tidigare (3.7, sid 172) innebär samma namn för olika metoder. Skillnaden mellan polymorfism och överlagring av metoder är följande:

I vanlig överlagring (eng. *overloading*) definieras *i samma klass* metoder med samma namn, som skiljs åt genom olika antal eller olika typer av parametrar. Polymorfa metoder definieras *i olika klasser* som dessutom ärver varandra. Dvs polymorfa metoder förekommer endast i klasser som står i *arvrelation* till varandra. De har samma namn och samma parameterlista, medan deras kroppar (innehåll) är olika.

Vi implementerar modellen ovan i C#. Så här kan t.ex. klassen **`Account`** se ut:

```
// Account.cs
// Super- eller basklass till subklassen Minimalkonto
// Definierar bl.a. metoden Withdraw() med en vanlig uttags-
// policy: Uttag medges ej om uttagsbloppet är större än saldo
// Definieras om i subklassen som har en striktare uttagspolicy
using System;
class Account
{
   protected int    accountNo;
   protected String name;
   protected double balance;

   public Account(int no, String n, double b)
   {
      accountNo = no;                          // Konstruktorn
      name      = n;
      balance   = b;
   }

   public String Withdraw(double amount)       // Metod som defi-
   {                                           // nieras om i
      if (balance - amount < 0)                // subklassen
         return "\n\tIngen täckning\n\tför uttag på " +
             amount.ToString("c") + " på " + name + "s konto\n";
      else
      {
         balance = balance - amount;
         return "\n\tUttag på "  + amount.ToString("c") +
                " genomfört på " + name + "s konto\n"   ;
      }
   }

   public String AsString()
   {
      return "\tKontonr   " + accountNo             + '\n' +
             "\tNamn      " + name                  + '\n' +
             "\tSaldo     " + balance.ToString("c") + '\n' +
             "**************************\n\n" ;
   }
}
```

Överskuggning av metoder (eng. overriding)

Klassen **`Account`** beskriver ett vanligt bankkonto med en **`Withdraw()`**-metod som inte tillåter uttag av pengar om uttagsbeloppet överstiger saldot. En bank har däremot många olika typer av konton. Tänkbart är t.ex. ett konto som alltid behåller ett visst minimalbelopp på kontot och inte tillåter uttag av pengar om saldot efter uttag understiger detta minimalbelopp. Ett sådant specialkonto beskrivs nedan i klassen **`MinimalAccount`** som ett **`Account`** med en **`Withdraw()`**-metod som implementerar denna affärslogik.

```
// MinimalAccount.cs
// Subklass som ärver superklassen Account, men definierar om
// den ärvda metoden uttag() med en striktare uttagspolicy
// Uttag medges ej om saldo efter uttag är mindre än minimalSaldo
// Withdraw() har samma huvud, men en annan kropp än superklassen
using System;

class MinimalAccount : Account           // Ärver klassen Account
{
   double myBalance;                     // Ny datamedlem

   public MinimalAccount(int no, String n, double b, double minB)
                         : base(no, n, b)  // Superklassens kon-
   {                                       // struktor
      myBalance = minB;
   }

   public String Withdraw(double amount) // Definierar om super-
   {                                     // klassen Kontos me-
      if (balance - amount < myBalance) // tod: Inte längre < 0
          return "\n\tIngen täckning\n\tför uttag på " +
              amount.ToString("c") + " på " + name + "s konto\n";
      else
      {
          balance = balance - amount;
          return "\n\tUttag på " + amount.ToString("c") +
                   " genomfört på " + name + "s konto\n";
      }
   }
}
```

`MinimalAccount` ärver klassen **`Account`** genom att lägga till den nya datamedlemmen **`myBalance`** och definiera om **`Account`**-klassens **`Withdraw()`**-metod. Vi har med två olika metoder **`Withdraw()`** att göra. I alla objekt av typ **`Account`** kommer den ena – den ursprungliga – att gälla, i alla objekt av typ **`MinimalAccount`** kommer den andra att gälla. Man säger: Den nya, modifierade metoden **`Withdraw()`** *överskuggar* den gamla. Dvs en metod i en subklass överskuggar (slår ut temporärt) metoden med samma namn i sin superklass. Överskuggning (eng. *overriding*) är ett koncept som vi redan lärt känna och använt när vi diskuterade lokala variabler. Men då handlade det om *överskuggning av variabler* medan nu har vi att göra med *överskuggning av metoder*.

En konsekvens av att metoderna **`Withdraw()`** inte längre befinner sig i samma klass, är att de inte längre behöver skiljas åt genom olika parameterlistor. De är redan skilda genom sin placering i olika klasser och kommer därför att anropas i objekt av *olika* klasser. De måste tvärtom ha t.o.m. *samma* parameterlista. För att subklassens metod ska kunna överskugga (slå ut temporärt) superklassens metod, *måste* metodhuvuden vara exakt identiska. Därför har **`Withdraw()`** i **`MinimalAccount`** samma huvud som **`Withdraw()`** i **`Account`** (framhävd med vit bakgrund). De skiljer sig endast genom kroppen, närmare bestämt i **`if`**-satsens villkor: I superklassen implementeras den vanliga policyn för uttag av pengar med **`if (balance - amount < 0)`**, medan i subklassen ska den speciella uttagpolicyn gälla: **`if (balance - amount < myBalance)`**. Överskuggning av metoder är en konsekvens och en väsentlig ingrediens av polymorfism.

Åtkomstmodifieraren `protected`

När vi diskuterade inkapsling lärde vi känna åtkomstmodifieraren **`private`**. Innan dess hade vi använt åtkomstmodifieraren **`public`**. Det finns ytterligare en åtkomstmodifierare i C# som heter **`protected`**. De reglerar åtkomsten till medlemmarna i en klass utifrån klassen. Ställer man upp dem i en rangordning från *restriktiv* till *liberal* får man följande lista:

- **`private`**
- **`protected`**
- **`public`**

`private` är den mest restriktiva modfieraren och spärrar åtkomsten absolut. Inte ens en subklass har tillgång till superklassens privata medlemmar fast den ärver allt ovanifrån. **`public`** är den mest liberala modfieraren och friger åtkomsten åt alla utifrån. **`protected`** är en kompromiss som friger åtkomsten till klassens medlemmar från en subklass och spärrar åtkomsten från alla andra klasser. Subklassen kan finnas i samma eller i en annan fil.

I klassen **`Account`** är det ganska naturligt att deklarera datamedlemmarna som **`protected`**. På så sätt skyddas uppgifterna om **`accountNo`**, **`accountName`** och **`balance`** från all kod som inte har att göra med klassen **`Account`**. Samtidigt är de tillgängliga från alla klasser som ärver klassen **`Account`** dvs är också konton, fast mer specialiserade. Alla dessa specialkonton kommer att ha åtminstone dessa tre grund-datamedlemmar. Med **`protected`** slipper man skriva Set- och Get-metoder i subklassen **`MinimalAccount`**, vilket underlättar programmeringen. Subklassen **`MinimalAccount`** kan t.ex. i sin **`Withdraw()`**-metod komma åt superklassens datamedlemmar **`name`** och **`balance`** tack vare **`protected`**. Annars, om **`name`** och **`balance`** hade varit **`private`**, hade vi behövt definiera och anropa Get-metoder.

Nu när vi testar både **`Account`**- och **`MinimalAccount`**-klassen i en väldigt enkel applikation kan vi konstatera att kompilatorn automatiskt väljer rätt **`Withdraw()`**-metod vid anrop – trots samma namn och samma parameterlista:

```
// PolymorphTest.cs
// Demonstrerar anrop av den polymorfa metoden uttag()
// En gång anropas uttag() i ett Account-objekt (kalle)
// den andra gången i ett MinimalAccount-objekt (pelle)
using System;

class PolymorphTest
{
   static void Main()
   {
      Account kalle = new Account(12345, "Kalle", 200);
      MinimalAccount pelle =
                     new MinimalAccount(67890, "Pelle", 100, 50);

      Console.Write("\nKalles konto före uttag:\n" +
                                          kalle.AsString() +
                    "\tTa ut ett belopp från Kalles konto: ");
      double out = Convert.ToDouble(Console.ReadLine());

      Console.Write(kalle.Withdraw(out) +  // Här anropas super-
                                           // klassens Withdraw()
             "\nKalles konto efter uttag:\n" + kalle.AsString() +
             "Pelles konto före uttag:\n"    + pelle.AsString() +
             "\tTa ut ett belopp från Pelles konto: ");
      out = Convert.ToDouble(Console.ReadLine());

      Console.Write(pelle.Withdraw(out)+   // Här anropas sub-
                                           // klassens Withdraw()
             "\nPelles konto efter uttag:\n" + pelle.AsString());
   }
}
```

På den första raden i **`Main()`** skapas objektet **`kalle`** av typ **`Account`**, på den andra raden objektet **`pelle`** av typ **`MinimalAccount`**. De initieras med var sin konstruktor. **`kalle`** får **`200`** kr insatt på sitt konto, **`pelle`** **`100`**. Eftersom **`pelle`** har ett **`Minimal-Account`** måste hans konto initieras även med ett värde till den nya datamedlemmen **`myBalance`**. Därför skickas som sista parameter till **`pelle`**-konstruktorn värdet **`50`** som enligt affärslogiken alltid ska vara kvar på ett **`MinimalAccount`**. Så, **`pelle`** får maximalt ta ut 50 kr från sitt konto. Försöker han ta ut t.ex. **`100`** kr – vilket vi gör i kör-exemplet på nästa sida – godtas inte uttaget och han får meddelandet **`Ingen täckning på Pelles konto`** som har sitt ursprung i anropet **`pelle.Withdraw(out)`**. Efter det misslyckade uttagsförsöket är **`pelle`**:s saldo fortfarande **`100`**.

I programmet **`PolymorphTest`** förekommer två anrop av den polymorfa metoden **`Withdraw()`**, en gång superklassens och en gång subklassens **`Withdraw()`**-metod:

`kalle.Withdraw(out)` och **`pelle.Withdraw(out)`**

Att vi kallar metoden för polymorf beror på att det är två *olika* metoder med två olika funktionaliteter (två olika kroppar) med samma namn och samma huvud. Det är de två

olika objekten **kalle** och **pelle** som gör att kompilatorn väljer rätt metod. Men det finns i programmet även två gånger två anrop av metoden **AsString()**:

kalle.AsString() och **pelle.AsString()**

Är det här också två olika metoder? Är även metoden **AsString()** polymorf? Svaret är nej, därför att det endast finns *en* metod **AsString()** som är definierad i superklassen **Account**. Hur kommer det sig då att vi kan anropa den även i **pelle**-objektet som inte är av typ **Account**? Det kan vi göra därför att **MinimalAccount** som **pelle** är ett objekt av, *ärver* **Account** och därmed även den publika metoden **AsString()**. Därför är metoden **Withdraw()** polymorf, men inte metoden **AsString()**. Så här kan en körning av **PolymorphTest** se ut:

```
Kalles konto före uttag:
        Kontonr   12345
        Namn      Kalle
        Saldo     200,00 kr
**************************

        Ta ut ett belopp från Kalles konto: 200

        Uttag på 200,00 kr genomfört på Kalles konto

Kalles konto efter uttag:
        Kontonr   12345
        Namn      Kalle
        Saldo     0,00 kr
**************************

Pelles konto före uttag:
        Kontonr   67890
        Namn      Pelle
        Saldo     100,00 kr
**************************

        Ta ut ett belopp från Pelles konto: 100

        Ingen täckning
        för uttag på 100,00 kr på Pelles konto

Pelles konto efter uttag:
        Kontonr   67890
        Namn      Pelle
        Saldo     100,00 kr
**************************
```

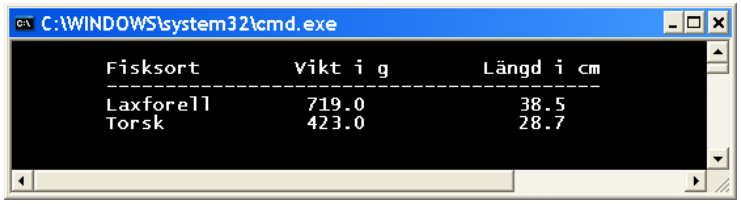
Övningar till kapitel 2

2.1 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 π = 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`**.

2.2 Modularisera programmet **`All_in_Main`** från övn 2.1 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} = π och omkretsen bli 2π pga 2π **`r`** = 2π.

2.3 Modularisera programmet **`All_in_Main`** från övn 2.1 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()`**. Klassfilerna borde ligga i samma projekt.

2.4 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:



2.5 Ta klassen **Fish** från övn 2.4. 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.

2.6 Modifiera programmet från övn 2.5 så att datamedlemmarnas värden inte hårdkodas utan läses in. Utskriften ska skickas till konsolen och läggas till tabellen från övn 2.4. 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 databas istället för att skriva till konsolen.

2.7 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?

2.8 Skriv en klass **Rectangle** med datamedlemmarna **width**, **height** samt metoderna **Area()** och **Circumference()**. Deklarera datamedlemmarna en gång som **private** och en annan gång med ingen åtkomstmodifierare alls. Deklarera metoderna som **public**. Förse klassen med en konstruktor och välj andra namn för konstruktorns parametrar än för datamedlemmarna. Testa din klass i en annan klass genom att i **Main()** skapa ett **Rectangle**-objekt vars datamedlemmar initieras till konstanta värden. Skriv ut dess area och omkrets.

2.9 Modifiera klassen **Rectangle** från övn 2.8 genom att lägga till Get- och Set-metoder i klassen. Testa den nya klassen i **Main()** genom att *läsa in* värden till datamedlemmarna. Efter utskriften av area och omkrets, fördubbla rektangelns längd och bredd med anrop av Get- och Set-metoderna. Skriv ut en gång till rektangelns area och omkrets. Med vilken faktor växer arean resp. omkretsen?

2.10 Modellera en klass **Cylinder** som subklass till klassen **Circle**. Förse superklassen **Circle** med en privat datamedlem **radius**, en konstruktor, en Get-metod och med beräkningsmetoderna **Area()** och **Circumference()**. Betrakta **Cylinder**n som en ”utvidgad” **Circle** som ärver **Circle** och lägger till den en

privat datamedlem **`height`**. Förse även subklassen med en konstruktor och en Get-metod. **`Cylinder`**n ska dessutom ha metoderna **`Volume()`** och **`Surface()`**. Implementera din objektorienterade modell så att du vid beräkning av **`Cylinder`**ns **`Volume()`** och **`Surface()`** kan återanvända koden till – dvs anropa – cirkelns metoder **`Area()`** och **`Circumference()`**. Testa dina klasser i **`Main()`** genom att läsa in **`radius`** och **`height`** samt skriva ut **`Volume()`** och **`Surface()`**.

2.11 ***Employee – en arvhierarki (projekt)*** Modellera en arvhierarki över olika typer av anställda och använd den polymorfa metoden **`Salary()`** i alla klasser för att beräkna lönen för de olika anställdtyper. Skriv en superklass **`Employee`** som ärvs av subklasserna **`PermEmployee`**, **`Seller`** och **`Employee`**. Varje subklass ska ha privata datamedlemmar, en konstruktor, Properties till varje ny datamedlem, en **`AsString()`**-metod som skriver ut en anställds typ, namn och anställningsnummer samt metoden **`Salary()`** som i varje subklass definierar om superklassens metod **`Salary()`**. Introducera privata datamedlemmar till klasserna **`PermEmployee`**, **`Seller`** och **`Employee`**. Skriv dessutom en subklass **`PermSeller`** som ärver klassen **`Seller`** och har den nya privata datamedlemmen **`permSalary`**. Testa dina subklasser genom att skapa och initiera en instans av varje subklass och ändra lönen till en av dem samt skriva ut deras gamla och nya data.

2.12 ***Kaffeautomaten (projekt)*** Du får i uppdrag att programmera en kaffeautomat. Uppdragsgivaren förväntar sig ett professionellt program som lätt kan uppdateras, om man skulle byta till en nyare automatmodell om något år. Därför anlitar man en objektorienterad programmerare. Skriv koden så generellt som möjligt så att programmet även kan modifieras för vilken varuautomat som helst, dessutom enkelt kan översättas till vilket programmeringsspråk som helst.

Programmet ska inte simulera själva automaten utan en *aktion* i automaten, dvs snarare det man *gör* med den. I händelsernas centrum ska finnas en *klass* som beskriver det som pågår i automaten *efter* att användaren stopppat in pengar i den och valt en dryck. Deklarationen till denna klass kan – i stora drag – se ut så här:

```
class Automat_action
{
   string productName;
   double price;
   double payment;
   double change;

   public Automat_action(double money, char product)
   {
      switch(product)
      {
         . . .
      }
```

```
            payment = money;
            change = payment - price;
        }

        public void Change_in_coins()
        {
                . . .
        }
    }
```

Konstruktorn **`Automat_action()`** ska tilldela de by default privata datamedlemmarna **`productName`** och **`price`** värden beroende på valet av dryck och skriva ut ett meddelande om inlagt belopp samt drycken som ska levereras. Detta kan kodas med hjälp av en **`switch`**-sats (ovan). Men istället för **`switch`** kan man lika bra använda nästlade **`if-else`**-satser. Skapa objekt av klassen **`Automat_action`** i en annan klass i en separat fil som endast innehåller **`Main()`**.

Börja i **`Main()`** med att skriva ut en meny över alla varor samt priserna, t.ex.:

K(affe)	8.00 kr
E(spresso)	9.50 kr
C(hoklad)	7.50 kr
L(Kaffe Latte)	9.00 kr
P(Cappuccino)	9.50 kr

Låt sedan användaren lägga in pengar. Läs in beloppet till en **`double`**-variabel. Låt användaren även välja en dryck genom att läsa in begynnelsebokstaven till varorna ovan med en **`char`**-variabel. Sedan kan ett objekt av den ovan deklarerade klassen **`Automat_action`** skapas in inkl. anrop av konstruktorn **`Automat_action()`**. 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 automatens ”tillåtna” myntslag (10-kr, 5-kr, 1-kr och 50-öringar) och skriva ut hur många av varje ”tillåtet” myntslag som ska ges tillbaka. Växelbeloppet måste omvandlas till detta mynt”system”. För att åstadkomma det, kan följande algoritm användas:

---

[*] Myntbetalningen inkl. behandlingen av 50-öringen beror inte på nostalgi utan på internationalisering. Vi vill hålla möjligheten öppen för en överföring av programmet till andra länder där automater med myntbetalning fortfarande finns. Även ett ev. byte till Euro eller andra valutor där den halva valutaenheten finns kvar, ska vara möjligt. Omvandlingen av växelbeloppet till automatens myntsystem inkluderar en programmeringsteknisk finess som kan vara värd att lära sig. Logiken inkl. användningen av modulooperatorn ligger till grund även för en generell omvandling av det decimala talsystemet till andra system. Läs mer om det på nästa sida: Modulooperatorn % .

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 örebelopp 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 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. Om heltalsdivision läs på nästa sida: Modulooperatorn **%** .
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;
```

Här används modulooperatorn **%**. Läs om den nedan. ”Resten som blev kvar från punkt **1**” är just **(total % 1000)**. 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 de övriga formlerna för beräkning av antalet 1-kronor (**one**), 50-öringar (**half**) och resten i öre (**rest**) skrivas, när mönstret i algoritmen (förhoppningsvis) har trätt fram:

```
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. Mer om modulooperatorn och heltalsdivision kan du läsa här:

Modulooperatorn %

% har i C# ingenting med procenträkning att göra utan är symbolen för ett räknesätt som kallas *modulo* och innebär *resten vid heltalsdivision*. Exempel:

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

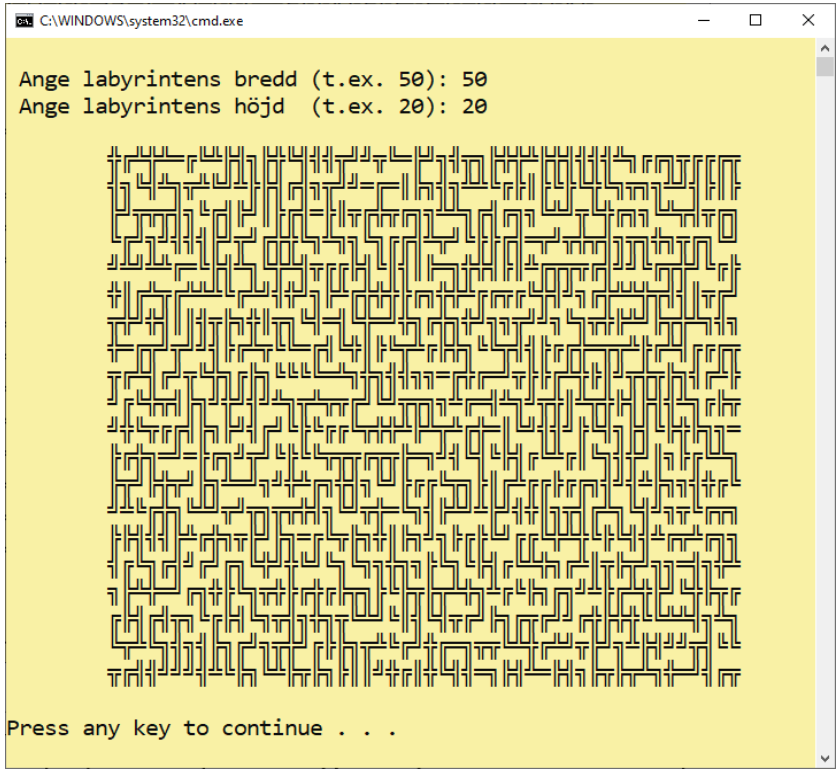
Dvs veckodagen i frågan är tisdag. Man lägger till aktuell veckodag **5**, antalet dagar **11** vilket ger **16**, men räknar modulo **7** dvs **16 % 7 = 2**, som är veckodag nr. **2**: tisdag.

I själva verket handlar det om en omvandling av det decimala talsystemet med basen 10 och siffrorna 0-9 – det talsystem vi är vana vid att räkna med – till veckodagarnas system dvs till talsystemet med basen **7** och siffrorna 0-6.

Modulo 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** att man bortser från alla multipler av heltalet **a** och behåller resten.

Räknesättet modulo har många tillämpningar, speciellt vid övergång mellan två system, t.ex. mellan talsystem med olika baser som det decimala talsystemet med basen 10 och det binära med basen 2. Man kan användsa modulo för att omvandla ett antal sekunder till antal timmar, minuter och sekunder.

2.13 ***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 labyrintartad 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”:

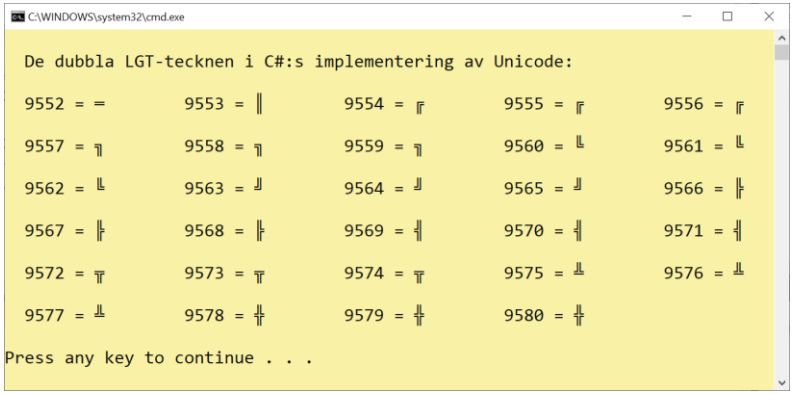
Steg 1 Bekanta dig med hantering av tecken i C# inkl. *explicit typkonvertering* och *Unicode*, genom att mata in och testa följande program:

```
// Int2char.cs
// Ger tecknet till en inmatad Unicode genom explicit
// typkonvertering från int till char
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 " +
         "Unicode till tecknet " + (char) code +
         "\n\n");        // Explicit typkonvertering
   }
}
```

Kör programmet **`Int2char`** (ovan) för koderna 9552-9580. För att se alla tecken till dessa koder i en översikt genomför ***Stegen 2-3*** :

Steg 2 Studera programmet **`RandTest`** i kursboken, sid 137-138, som visar hur man nästlar en inre **`for`**-sats i en yttre **`for`**-sats. Jämför den nästlade **`for`**-satsens kod med programmets körexempel på sid 138. 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 med klassen **`Random`** och metoden **`Next()`**, bl.a. i programmet **`RandTest`** i kursboken, sid 137-138.

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.

2.14 ***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.

Kapitel 3

Metoder i OOP

3.1 Accessmetoder

Här återanknyter vi till vår diskussion kring inkapsling och klassens konstruktor (sid 91). Accessmetoder är nämligen direkta konsekvenser av inkapsling dvs att man vill ha privata datamedlemmar. Konstruktorn kan ju lösa problemet med de privata datamedlemmarnas oåtkomlighet endast i initialfasen, dvs när objekten skapas. Men man vill ju även i fortsättningen kunna komma åt de inkapslade datamedlemmarna. För detta ändamål har man i C# accessmetoder till förfogande.

Det finns tre sorters accessmetoder: *Get-metoder* för att hämta (läsa), *Set-metoder* för att ändra (skriva) värden till privata datamedlemmar och s.k. *strängrepresentationsmetoder* för att få ut och kunna visa de privata datamedlemmarna i läsbar textform. Följande program visar exempel på alla tre typer av accessmetoder:

```
// Empl.cs
// Deklarerar klassen Empl med 3 privata datamedlemmar, en
// konstruktor, en Get- och Set-metod till datamedlemmen salary
// och metoden AsString() som ger ett Empl-objekt i strängform
using System;

class Empl
{
   String name;
   int empNo;
   double salary;

   public Empl(String n, int e, float s)
   {
      name   = n;                          // Konstruktorn
      empNo  = e;
      salary = s;
   }

   public double GetSalary()               // Get-metod
   {
      return salary;
   }

   public void SetSalary(double newSalary) // Set-metod
   {
      salary = newSalary;
   }

   public String AsString()               // Strängrepresentation
   {
      return "\tNamn       " + name   + '\n' +
             "\tAnst nr    " + empNo  + '\n' +
             "\tLön        " + salary + '\n' ;
   }
}
```

Som man ser handlar det hos den sista metoden **`AsString()`** konkret om att skriva ut klassens data som en konkatenerad sträng, en slags representation av klassens objekt i strängform.

Förfarandet som visas här kan generaliseras, ja t.o.m. automatiseras: Till varje privat datamedlem kan en Get- och en Set-metod definieras, medan en utskriftsmetod räcker för hela klassen. Om man sedan faktiskt utnyttjar alla dessa verktyg i varje program, måste avvägas från fall till fall. Get-metoder ska ha ett returvärde med samma returtyp som den privata datamedlemmen, inga parametrar och endast en **`return`**-sats som returnerar den privata datamedlemmens värde. Alla Get-metoder har detta utssende. Man kan t.o.m. standardisera namngivningen genom att döpa Get-metoden till GetX, där X är den privata datamedlemmens namn som man inleder med en versal. Set-metoden däremot är en **`void`**-metod med en parameter som har samma datatyp som den privata datamedlemmen och innehåller endast en tilldelningssats som tilldelar parametern till den privata datamedlemmen. Namnet ska vara SetY där Y är den privata datamedlemmens versala initial. Utskriftsmetoden är av **`void`**-typ utan parametrar och skriver ut alla privata datamedlemmar i en användarvänlig layout. Självklart behöver man inte i alla fall genomföra det ”fulla” förfarandet ovan. I klassen **`Empl`** som vi testar i följande program har vi nöjt oss med att definiera en Get- och Set-metod endast för den privata datamedlemmen **`salary`**:

```
// GetSet.cs
// Använder klassen Empl för att skapa en anställd, ändra dess
// salary (som är privat) med Get- och Set-metoden samt skriva ut
// data, bl.a. den gamla och nya lönen, med AsString()
using System;

class GetSet
{
   static void Main()
   {
      Empl emp = new Empl("Kalle Karlsson", 349, 22500);

      Console.WriteLine("\n\tFöre löneförhöjning:\n" +
                                       emp.AsString());

      emp.SetSalary(emp.GetSalary()*1.25);       // Ändrar lön

      Console.WriteLine("\tEfter löneförhöjning:\n" +
                                      emp.AsString());
   }
}
```

Ändringen av **`salary`** görs via anrop av Set-metoden **`SetSalary()`**. Som parameter skickas den gamla lönen höjd med 25%. För att få tag i den gamla lönen hämtas den privata datamedlemmen **`salary`** med ett anrop av Get-mtoden **`GetSalary()`**.

Programmet **`GetSet`**:s körresultat blir:

```
Före löneförhöjning:
Namn       Kalle Karlsson
Anst nr    349
Lön        22500

Efter löneförhöjning:
Namn       Kalle Karlsson
Anst nr    349
Lön        28125
```

3.2 Property i C#

Property i C# är inte längre en vanlig egenskap (attribut) eller datamedlem i den betydelse vi använt begreppet hittills, utan ett koncept i C# som automatiserar Get- och Set-metoder i klasser med privata datamedlemmar för att underlätta programutvecklingen och minska den overhead av kod som behövs för att hantera inkapsling. Property i C# är motsvarigheten till *Beans* i Java. Istället för att till varje privat datamedlem skriva en Get- och en Set-metod, kan man definiera en Property till den. Vi tar upp klassen **`Empl`** från förra avsnitt, döper om den till **`EmplP`** och ersätter dess Get- och Set-metoder till datamelemmen **`salary`** med en Property:

```
// EmplP.cs
// Klassen Empl med Property som automatiserar den privata data-
// medlemmen salary:s Get- och Set-metoder
using System;

class EmplP
{
   String name;
   int empNo;
   double salary;

   public EmplP(String n, int e, float s)
   {
      name   = n;               // Konstruktor samma som för Empl
      empNo  = e;
      salary = s;
   }

   public double Salary         // Property Salary av typ double
   {                            // till privat datamedlem salary
      get                       // Kan skrivas till alla privata
      {                         // datamedlemmar
         return salary;
      }
      set
      {
         salary = value;
      }
   }

   public String AsString()     // Strängrepresentation
   {
      return "\tNamn       " + name   + '\n' +
             "\tAnst nr    " + empNo  + '\n' +
             "\tLön        " + salary + '\n' ;
   }
}
```

Propertyn är framhävd med vit bakgrund. **get**, **set** och **value** är reserverade ord. Propertys namn **Salary** däremot kan man välja fritt. Inga parenteser förekommer, för Property är inte någon metod. Den liknar mycket mer en datamedlem, ja man kan säga, det är en slags *generaliserad datamedlem*, dessutom en publik sådan. Den är till för att utanför klassen kunna läsa värdet av och skriva ett nytt värde till en privat datamedlem. Operationerna *läsa* och *skriva* är implementerade i Propertyns delar **get** och **set**. Deras innehåll inom **{ }** *liknar* metoder, fast även de skrivs utan parenteser och därmed inte kan vara metoder. **get**:s kod är identisk med kroppen till vår ”manuella” Get-metod i klassen **Empl** dvs består endast av en **return**-sats. Även **set**:s kod är nästan identisk med kroppen till vår gamla Set-metod, bara att parametern som skickar det nya värdet, har ersatts av **value**. Den stora programmeringstekniska fördelen av Property kan först ses när man använder den, t.ex. i följande program som testar klassen **EmplP**:

```
// Property.cs
// Använder klassen EmplP, skapar en anställd, ändrar dess
// salary (som är privat) med Propertyn Salary och skriver ut
// data, bl.a. den gamla och nya lönen, med AsString()
// Propertyn Salary anropar automatiskt Get- resp. Set-metod
using System;

class Property
{
   static void Main()
   {
      EmplP emp = new EmplP("Kalle Karlsson", 349, 22500);

      Console.WriteLine("\n\tFöre löneförhöjning:\n" +
                                      emp.AsString());

      emp.Salary = emp.Salary*1.25;      // Ändrar privat salary
                                         // med Propertyn Salary
      Console.WriteLine("\tEfter löneförhöjning:\n" +
                                     emp.AsString());
   }
}
```

Ändringen av den anställdas lön görs inte längre via anrop av någon metod, utan med hjälp av Propertyn **Salary**. Om man inte kände till klassen **EmplP**:s kod, skulle man kunna misstänka att **Salary** vore en vanlig datamedlem. Det enda som ”stör” bilden är det stora **S**. Man är van vid små initialer till datamedlemmar. Det stora **S** avslöjar **Salary** som en Property, för en metod kan den ju inte vara pga avsaknaden av parenteser.

Det intressanta är nu att **emp.Salary** till *höger* om tilldelningstecknet automatiskt *läser* den privata datamedlemmen **salary**:s värde dvs exekverar Propertyns **Get-**del, medan **emp.Salary** till *vänster* om tilldelningstecknet automatiskt *skriver* det nya värdet som bildats till höger, till den privata datamedlemmen salary dvs exekverar Propertyns **Set-**del. På så sätt ändras lönen till den anställd som får en 25%-ig löneförhöjning. En körning av programmet **Property** ger samma utskrift som programmet **EmpTest**.

3.3 Statiska datamedlemmar och metoder

När vårt allra första C# program **First** introducerades i *Progr1* sades så här om metoden **Main()**:

> ”Det är Virtual Machine (VM) som exekverar vårt program **First** genom att anropa metoden **Main()**. För att detta anrop ska kunna utföras behövs de s.k. *modifierarna* **public** och **static** i metodens huvud.” *(Progr1, 1.1)*

Sedan dess har vi använt **static** tillsammans med **void** i alla våra program i huvudet till metoden **Main()**:

```
static void Main()
```

En allra första förklaring gavs i *Progr1* så här:

public innebär	att man kan anropa denna metod utifrån klassen **First**,
static	att den kan anropas utan att skapa objekt av klassen **First**,
void	att metoden **Main()** inte returnerar något värde.

Men nu när vi har mer kunskap om klasser och objekt ska vi precisera dessa förklaringar. **void** behandlas i detalj när vi i nästa kapitel närmare går in på metoder. **public** kommer att tas upp i samband med **private** och andra s.k. *åtkomstmodifierare*. Men **static** kommer vi att behöva snart när vi vill skriva egna metoder som ska anropas från **Main()** utan att skapa ett objekt. Se även *Åtkomstmodifieraren* **static** på sid 137.

Klass- och instansvariabler

En icke-statisk datamedlem kallas för *instansvariabel*. Följande klass visar ett exempel:

```
// StatDemo.cs
// Klass med två datamedlemmar, en statisk och en icke-statisk

class StatDemo
{
   public static int klassVar; // Klassvariabel allokeras här
                               // i klassen. Behöver inget objekt.
   public int instVar;  // Föreskrift om att en instansvariabel
                        // med namnet instVar skall allokeras i
                        // varje objekt som skapas av denna klass
}
```

Programmet nedan demonstrerar skillnaden mellan klass- och instansvariabler i en loop:

```
// StatDemoTest.cs
using System;

class StatDemoTest
{
```

```
   static void Main()
   {
      int i = 0;
      Console.WriteLine("\nKlassvariabeln skapas och initieras" +
                "  i klassen till: " + StatDemo.klassVar + '\n');

      do
      {
         StatDemo obj = new StatDemo();          // Nytt objekt i
                                                 // varje varv
         Console.WriteLine(
               "Samma klassvariabel ökar i varje varv:\t\t    " +
                                    StatDemo.klassVar + '\n'  +
               "Ny instansvariabel skapas i varje objekt:     " +
                                               obj.instVar + '\n');
         StatDemo.klassVar++;     // Ökar löpande
         obj.instVar++;           // Ökar i varje objekt från 0-1
      } while (i++ < 4);           // Fem varv
   }
}
```

Programmet ovan producerar följande resultat när det exekveras:

```
Klassvariabeln skapas och initieras i klassen till: 0

Samma klassvariabel ökar i varje varv:              0
Ny instansvariabel skapas i varje objekt:     0

Samma klassvariabel ökar i varje varv:              1
Ny instansvariabel skapas i varje objekt:     0

Samma klassvariabel ökar i varje varv:              2
Ny instansvariabel skapas i varje objekt:     0

Samma klassvariabel ökar i varje varv:              3
Ny instansvariabel skapas i varje objekt:     0

Samma klassvariabel ökar i varje varv:              4
Ny instansvariabel skapas i varje objekt:     0
```

På första raden skrivs ut klassvariabeln **`klassVar`**:s initialvärde **`0`** genom att i programmets första utskriftssats före loopen referera till **`StatDemo.klassVar`** dvs klassen **`StatDemo`**:s statiska datamedlem **`klassVar`**. Modifieraren **`static`** framför deklarationen av **`klassVar`** i klassen **`StatDemo`** gör att vi kan referera till denna variabel med klassnamnet utan att behöva att skapa ett objekt.

I utskriften ovan visas klassen **`StatDemo`**:s både statiska och icke-statiska datamedlemmar. Man ser att klassvariabelns värde löpande ökar, medan instansvariabeln visar värdet **`0`**. Utskrifterna är resultat av en loop i programmet som har fem varv motsvarande

räknaren `i`:s fem värden `0-4`. I varje varv skapas ett objekt av typ `StatDemo`. Både objektets (dvs instansens – instans är bara ett annat ord för objekt) och klassens variabel ökar sina värden med `1` i varje varv av loopen. Detta sker i satserna `StatDemo.klassVar++;` och `obj. instVar++;` som står sist i loopen. Klassvariabeln kommer ihåg sina gamla värden från loopens gångna varv eftersom dess ökning sker i en och samma minnescell genom hela programmet och därför löpande. Instansvariablerna däremot skapas i varje varv av loopen i sina objekt på nytt, initieras till `0` och skrivs ut, ökar sedan till `1`, men ”dör” i början av nästa varv när ett nytt objekt skapas. Närmare bestämt, överskrivs deras adress i referensen `obj` med det nya objektets adress. Därför får de bara värdena `0` och `1` tilldelade av vilka endast `0` skrivs ut eftersom utskriftssatsen står före ökningen. Det handlar om olika objekt med olika instansvariabler i varje varv lagrade vid olika adresser.

I programmet ovan behandlades `static` för datamedlemmar. Nu ska vi undersöka `static` för metoder. Låt oss först sammanfatta vad vi vet om `static`.

Allokeringsmodifieraren `static`

Allokering betyder reservering av minnesutrymme i datorns RAM-minne under programkörningen. Med *modifierare* menas en egenskap som ges till en datamedlem, en metod eller en klass, genom att skriva modifieraren framför namnet. Medan `public` är en *åtkomstmodifierare*, därför att den ger en status angående *åtkomsten*, är `static` en *allokeringsmodifierare*, därför att den talar om på vilket sätt minnesutrymme ska *allokeras* åt en datamedlem, en metod eller en klass. Regler för `static`:

> Klasser, datamedlemmar & metoder kan deklareras som `static`, men inte lokala variabler. I en statisk klass är alla medlemmar statiska. Av en statisk klass kan man inte skapa objekt. Statiska medlemmar allokeras i klassen och inte i objekten. Därför används de med klassnamnet.
> Statiska datamedlemmar kallas även för *klassvariabler.* Ett annat namn

Statiska datamedlemmar kallas för klassvariabler, därför att de tillhör klassen och inte något specifikt objekt och instansvariabler används för icke-statiska datamedlemmar, därför att de tillhör objekten och inte klassen. Instans är synonym till objekt. Alla objekt som skapas av en icke-statisk klass kommer att *dela* minnesutrymme vid en och samma adress för sina statiska medlemmar. Ändrar man något objekt via sin referens kommer alla objekt i fortsättningen att ha detta ändrade värde. Kommer inget objekt att skapas av denna klass kan man nå datamedlemmen direkt via klassnamnet, därför att minnesallokeringen pga `static` har gjorts i klassen och inte i objekten. Vid punktnotation skrivs klassnamnet före punkten. Ett exempel är konstanten `π` som kodas med `Math.PI`. I klassen `Math` är `PI` en statisk datamedlem. Därför behöver vi inte skapa objekt av klassen `Math` för att kunna komma åt `PI`.

Den mest förekommande användningen av **`static`** är framför metoden **`Main()`**. Som en konsekvens kan man anropa statiska metoder i **`Main()`**. T.ex. är biblioteksmetoden **`Console.Write()`** som ofta anropas i **`Main()`**, statisk. Bland våra egendefinierade metoder är **`Encrypt()`** deklarerad som **`static`** (sid 140).

Statiska metoder

Precis som hos datamedlemmar allokerar **`static`** minne för en metod en gång för alla i *klassen* och inte i varje enskilt objekt som skapas. Därför används också ibland beteckningen *klassmetoder* för statiska metoder, parallellt till klassvariabler. Att allokera minne för en metod innebär att allokera minne för alla dess parametrar och lokala variabler. Av det följer att även en statisk metods parametrar och lokala variabler är statiska. Självklart kan man inte längre anropa en icke-statisk metod med klassnamnet. Hela **`Main()`** är ett statiskt kodområde i vilket man inte kan referera till icke-statiska vare sig metoder eller datamedlemmar. För att slippa deklarera en metod som **`static`** är det därför nödvändigt att skapa ett objekt av referens istället för med klassnamnet. Regler:

> Statiska medlemmar kan användas med klassnamnet.
>
> Icke-statiska medlemmar kan bara användas genom att först skapa ett objekt. De kan sedan användas med obkjetreferenser.

Dessa regler är logiska ur minnesallokeringssynpunkt: En metod är en modul som i sin helhet kan allokeras *antingen* i klassen *eller* i enskilda objekt. En blandning är omöjligt. Följande program visar att en icke-statisk metod måste i den statiska miljön **`Main()`** anropas med en objektreferens:

```
// RandTest.cs
// Simulerar 150 tärningskast: slumpar heltal mellan 1 och 6
// Icke-statisk metod Next() behöver ett objekt för att anropas
// Nästlad for-sats ordnar utskrifterna i en (10 x 15)-tabell
using System;
uclass RandTest
{
  public static void Main()
  {
    Random r = new Random();                // Random-objekt
    Console.WriteLine("\n\t150 tärningskast:");
    for (int rad = 1; rad <= 10; rad++)     // 10 rader
    {
       Console.Write("\n\t");               // Radbyte i tabellen
       for (int kol = 1; kol <= 15; kol++)  // 15 kolumner
           Console.Write(r.Next(1, 7) + "  "); // Anrop av icke-
    }                                          // statisk metod
    Console.WriteLine("\n");
    }
}
```

Programmet **RandTest**:s körresultat blir t.ex.:

```
150 tärningskast:

5  2  1  2  3  1  5  4  4  3  4  3  1  3  2
2  4  6  2  6  3  6  4  1  2  2  3  4  1  2
2  2  5  5  1  4  4  5  4  3  4  3  5  2  5
6  2  3  4  2  4  2  2  5  2  6  5  1  1  3
6  2  4  3  3  3  6  3  4  5  5  1  4  1  4
1  2  4  3  3  1  2  1  1  6  1  2  2  5  2
1  4  5  3  5  3  1  3  1  4  2  6  1  6  4
5  5  3  3  3  5  3  6  5  3  5  3  5  5  3
3  5  6  3  6  3  3  3  3  2  1  1  4  6  3
3  2  3  4  3  3  6  1  4  4  5  5  1  1  1
```

När ska man deklarera en metod som statisk?

Behovet av statiska metoder uppstår när en metod från modelleringssynpunkt inte kan relateras till ett specifikt objekt med specifika datamedlemmar utan är en allmän rutin som kan utföras helt fristående i objekt av alla möjliga slag. Ett exempel är metoder i klasser som antingen inte har några datamedlemmar alls eller vars datamedlemmar inte är relaterade till och därför inte förekommer i metoderna. En stor grupp statiska metoder är rena beräkningsmetoder. T.ex. är alla metoder i klassen **Math**, bland dem alla matematiska metoder, statiska därför att de är generella och inte bundna till andra data än sina egna parametrar (argument). Det vore slöseri med minnesutrymme och datortid om man behövde skapa först ett objekt av någon klass för att bara beräkna t.ex. sinusmetoden för ett visst värde, när denna beräkning inte har att göra med objektet: **Math.Sin(x)** exekverar endast kod som är nödvändigt för beräkning av sinusmetoden och kan anropas i både statiska och icke-statiska metoder.

Statiska metoder i C# ersätter de fristående funktioner som man har i andra, mindre objektorientorienterade programmeringsspråk som C++. Ur den objektorientorienterade programmeringens synpunkt är statiska metoder förstås av mindre intresse. Men deras flitiga förekomst i C#:s klassbibliotek visar att det finns behov för dem.

3.4 Referens i metoder

Kan en metod ha endast *enkla datatyper* som parametrar och returvärde, eller kan man skicka också *objekt* som indata (parametrar) till en metod och få även tillbaka *objekt* som utdata (returvärde)? Även om frågan kan bejakas utan vidare, måste svaret preciseras i den bemärkelse att det inte är själva objekten som skickas och fås tillbaka utan snarare deras referenser. Vi känner också till att man i C# hanterar objekten med hjälp av deras adresser och inte direkt. Det vore slöseri med datorns resurser (minnesutrymme) om man kommunicerade tunga objekt (TV-apparaten) istället för lätthanterade referenser till objekt (fjärrkontrollen). Så, det är inget nytt utan snarare det normala att använda referenser som företrädare för objekt – ett slags namn, precis som man använder vanliga namn för variabler av enkel datatyp.

Följande klass visar ytterligare ett exempel på en metod som har en referens **`t`** till ett **`String`**-objekt som parameter, men även en **`String`**-referens som returvärde. Dessutom har den också en vanlig **`int`**-parameter. Krypteringsmetoden **`Encrypt()`** skrivs i denna klass och anropas från **`Main()`** i klassen **`EncryptStrTest`** på nästa sida. Den är väldigt enkel, men kan lätt ersättas av mer sofistikerade krypteringsalgoritmer.

```
// EncryptStr.cs
// Metoden Encrypt() tar emot en sträng t och krypterar den
// genom att förskjuta alla tecken med n steg i teckentabellen
// Den krypterade strängen skrivs teckenvis till platsen temp
// Sedan returneras den krypterade strängen från metoden
using System;

class EncryptStr
{
   public static String Encrypt(String t, int n)
   {
      char ch;
      String temp = null;                // null-referensen
      for (int i=0; i < t.Length; i++)
      {                                  // Läser tecknen från t
          ch = Convert.ToChar(t.Substring(i, 1));
          ch = (char) (ch + n);          // Ändrar tecknen
          temp += ch;                    // Lagrar tecknen i temp
      }
      return temp;                       // Skriver till Encrypt
    }
}
```

Med den första parametern **`t`** får metoden **`Encrypt()`** tillgång till det **`String`**-objekt som skapas i den anropande metoden **`Main()`**. Adressen till detta objekt kopieras över till referensvariabeln **`t`** när **`Encrypt()`** anropas. Samma sak sker med krypteringsnyckeln vars värde kopieras till den andra parametern **`n`**. Sedan har vi i kroppen av metoden två lokala variabler **`ch`** och **`temp`**. Den första som är av typ **`char`** initieras i **`for`**-loopen och lagrar varje tecken från den inkommande okrypterade strängen **`t`**, men även det

krypterade tecknet för att slutligen överföra det via konkatenering till strängen **temp**. **for**-satsen går igenom alla tecken i **t** genom att initiera sin räknare **i** till **0** och avsluta loopen när räknaren har nått strängens sista tecken. Att man börjar med **0** beror på att C# räknar strängens första tecken med index **0**, det andra med index **1** osv. så att det sista tecknet får t.ex. index **25** om strängen innehåller **26** tecken. **Length** är en **String**-egenskap som ger antalet tecken i strängen, här **t**. Därför har vi i **for**-loopen avslutningsvillkoret **i < t.Length**. I varje varv av den läggs det uttagna tecknet från **t** i den lokala **char**-variabeln **ch** och görs om till ett nytt tecken med satsen **ch = (char) (ch + n);** där tecknet **ch**:s Unicode adderas med heltalet **n** (se teckenaritmetik, *Progr1, 3.2*). Resultatet omvandlas med explicit typkonvertering till **char** för att sedan tilldelas **ch** och överskriva dess gamla värde. Utan explicit typkonvertering skulle vi få kompileringsfel pga C#:s vägran att automatiskt typomvandla nedåt från **int** till **char** (*Progr1+, 5.7*). **for**-loopens sista sats bygger den krypterade strängen **temp** som efter **for** returneras när **Encrypt()** anropas i följande klass:

```
// EncryptStrTest.cs
// Krypterar strängen text med en slumpad krypteringsnyckel
// Återställer sedan den krypterade texten med den negativa
// krypteringsnyckeln, båda med samma metod Encrypt()
using System;

class EncryptStrTest
{
   public static void Main()
   {
      String text = "abcdefghijklmnopqrstuvwxyz";
      Random r = new Random();
      int nyckel = r.Next(40, 200);             // Krypterings-
                                                // nyckeln
      Console.WriteLine("\n\tKryptering av text:   ");
      Console.Write("\n\tOkrypterad text:    " + text);

      text = EncryptStr.Encrypt(text, nyckel);  // 1:a anropet
                                                // krypterar
      Console.Write("\n\n\tKrypterad text:     " +
                    text + "\n\n\tKrypteringsnyckeln: " + nyckel);

      text = EncryptStr.Encrypt(text, -nyckel); // 2:a anropet
                                                // återställer
      Console.WriteLine("\n\n\tÅterställd text:    " +
                                                  text + "\n");
   }
}
```

Ett körresultat visar följande utskrift:

```
Kryptering av text:

Okrypterad text:    abcdefghijklmnopqrstuvwxyz

Krypterad text:     ¥¦§¨©ª«¬-®¯°±²³´µ¶·¸¹°»¼½¾

Krypteringsnyckeln: 68

Återställd text:    abcdefghijklmnopqrstuvwxyz
```

Det engelska alfabet som använts som teststräng har krypterats med slumpnyckeln **68** och återställts med **-68**. Båda operationer utförs i programmet ovan med anrop av metoden **Encrypt()**, definierad i klassen **EncryptStr** (sid 140). Det första anropet sker med den **key** som anropet **r.Next(40, 200)** genererar, dvs ett heltalsslumpvärde mellan **40** och **199**.

Initieringen av datamedlemmen **temp** till **null** är nödvändig därför att den sedan används i satsen **temp += ch;** som pga den sammansatta tilldelningsoperatorn += är identisk med **temp = temp + ch;** . Därför måste den vara initierad när den initialt konkateneras med **char**-variabeln **ch** som av + automatiskt typkonverteras till **String**. Även här är det avgörande att skilja mellan *referensen* **temp** och den tomma strängen som ett **String**-*objekt*.

3.5 Abstrakta klasser och metoder

När du får instruktionen ”Rita en geometrisk figur!” kommer du antagligen att ställa frågan ”*Vilken* geometrisk figur ska jag rita?”. För, utan en närmare specificering är det inte klart, om du ska rita en cirkel, en kvadrat, en triangel eller … . Vilken geometrisk figur du än ritar, kommer den att vara ett exemplar av någon *under*kategori (cirkel, kvadrat, triangel, …) av huvudkategorin *geometrisk figur*. Detta beror på att geometrisk figur är en *abstrakt* kategori som inte kan exemplifieras direkt utan endast via sina underkategorier. Endast cirkel, kvadrat, triangel, … kan exemplifieras. Ytterligare exempel på en abstrakt kategori är levande väsen, fordon, biljett eller transportmedel.

Världen är full med abstrakta kategorier. Man programmerar dem som abstrakta klasser. Det vi sa ovan innebär i programmeringstermer att man inte kan skapa objekt av abstrakta klasser. Endast deras subklasser kan instansieras. Ändå är abstrakta klasser av intresse, därför att de tillåter att förverkliga några av den objektorienterade programmeringens viktigaste mål, nämligen att återanvända kod, att modularisera och strukturera program. Abstrakta klasser förenar alla egenskaper och metoder som är gemensamma för alla objekt av denna typ – och med objekt menar vi objekt av de subklasser som ärver den abstrakta superklassen. T.ex. har alla geometriska figurer de gemensamma metoderna att rita, att beräkna arean och att beräkna omkretsen. Men även dessa metoder kommer att vara abstrakta, eftersom man inte kan utföra dem förrän någon subklass har specificerats. Abstrakta klasser ger alltså automatiskt upphov till abstrakta metoder, varför vi behandlar dem tillsammans i följande enkelt exempel:

```
// Super.cs
// Abstrakt superklass som deklarerar en abstrakt metod
// Endast metodens huvud (signatur) skrivs i klassen
// Överskuggning sker med abstrakt metod i superklassen OCH
//                        override       i subklasserna

abstract class Super
{
    public int number;     // Datamedlem: Initieras autom. till 0
    public abstract void Method();         // Abstrakt metodhuvud
}
```

Det är det reserverade ordet **`abstract`** i klasshuvudet som gör att man t.ex. i **`Main()`** eller i någon annan metod inte kan skriva **`Super a = new Super();`** Dvs försöket att skapa ett objekt av klassen **`Super`** som är deklarerad som **`abstract`** kommer att leda till kompileringsfel. Klassen **`Super`** har en datamedlem som deklareras, men inte explicit initieras. Till skillnad från lokala variabler i metoder som måste initieras explicit, blir datamedlemmar automatiskt initierade till **`0`** (om de är number). Sedan har klassen **`Super`** en klass som vi själva valt att deklarera som **`abstract`**. Abstrakta metoder är sådana som inte har sin kropp i samma klass som huvudet. Huvudet (signaturen) skrivs här och avslutas med semikolon. Kroppen kommer att definieras i subklasser till klassen **`Super`**. En mycket strikt regel för abstrakta metoder är att de *måste* implementeras (få en kropp) i *alla* subklasser som ärver superklassen. Självklart kan en abstrakt klass även

innehålla icke-abstrakta metoder (vilket inte förekommer i exemplet ovan). Men abstrakta metoder kan endast skrivas i abstrakta klasser. En annan regel för abstrakta metoder är att de inte får deklareras som privata.

Implementation av abstrakt metod

För att testa och bättre förstå de ovannämnda reglerna skapar vi följande subklass som ärver klassen **`Super`** och implementerar den abstrakta metoden **`Method()`** i den:

```
// Sub1.cs
// Subklass till klassen Super som implementerar den abstrakta
// metoden Method(): number ökar med 1
// Method() överskuggar (override) klassen Super:s metod Method()
using System;

class Sub1 : Super                    // Sub1 ärver Super
{
   public override void Method()     // override ersätter abstract
   {
      Console.WriteLine("\n\tSub1:s Method(): " +
                        "Initialvärde = " + number);
      number++;                       // number ökar med 1
      Console.WriteLine("\tSub1:s Method(): " +
                        "Uppdaterat värde = " + number);
   }
}
```

Implementeringen består av en ökning av datamedlemmen **`number`**:s värde med **`1`**, inramad av två utskriftssatser som skriver ut värdet, en gång före och sedan efter ökningen. I huvudet av metoden **`Method()`** ersätts **`abstract`** (i klassen **`Super`**) av det reserverade ordet **`override`**, vilket innebär att subklassens **`Method()`** ska överskugga dvs åsidosätta superklassens **`Method()`** och utföra den kod som vi skriver i denna kropp. På liknande sätt skapar vi en andra subklass till klassen **`Super`** och implementerar den abstrakta metoden **`Method()`** i den på ett lite annorlunda sätt:

```
// Sub2.cs
// En andra subklass till klassen Super som implementerar den
// abstrakta metoden Method(): number minskar med 1
using System;
class Sub2 : Super                    // Sub2 ärver Super
{
   public override void Method()     // override ersätter abstract
   {
      Console.WriteLine("\n\tSub2:s Method(): " +
                        "Initialvärde = " + number);
      number--;                         // number minskar med 1
      Console.WriteLine("\tSub2:s Method(): " +
                        "Uppdaterat värde = " + number + "\n\n");
   }
}
```

Även här överskuggar **Method()** klassen **Super**:s metod **Method()**. Den enda skillnaden till den första subklassen är att datamedlemmen **number**:s värde nu minskar med **1**.

Test av abstrakt metod

Vi ska nu testa om den rätta metoden anropas när vi med samma namn **Method()** en gång anropar metoden med ett objekt av den ena, en annan gång med ett objekt av den andra subklassen. Detta gör vi genom att i **Main()** skapa objekt av den ena och den andra subklassen. Klassen **Override** ser endast subklasserna **Sub1** och **Sub2**, inte superklassen **Super**:

```
// Override.cs
// Testar de överskuggade metoderna i subklasserna Sub1 och Sub2
// new Super() kan inte skrivas pga abstract class Super
// Objekten avgör vilken av metoderna Method() som ska anropas

class Override
{
   static void Main()
   {
      Sub1 a = new Sub1();
      a.Method();                    // Anrop av Sub1:s Method()
      Sub2 b = new Sub2();
      b.Method();                    // Anrop av Sub2:s Method()
   }
}
```

Så här ser en körning av **Override** ut:

```
      Sub1:s Method(): Initialvärde = 0
      Sub1:s Method(): Uppdaterat värde = 1

      Sub2:s Method(): Initialvärde = 0
      Sub2:s Method(): Uppdaterat värde = -1
```

De två första raderna kommer från anropet av **a.Method()** dvs från **Method()** tillhörande objektet **a** av den första subklassen **Sub1**. De två sista raderna kommer från anropet av **b.Method()** dvs från **Method()** tillhörande objektet **b** av den andra subklassen **Sub2**. Att datamedlemmen **number** uppdateras till **1** först och till **-1** sedan visar att **Sub1**:s **Method()** med **number++** i kroppen har anropats först och **Sub2**:s **Method()** med **number--** i kroppen sedan. Så båda metoder i de två subklasserna har verkligen överskuggat den abstrakta metoden med samma namn **Method()** i superklassen.

3.6 Virtuella metoder

Reglerna kring **abstract** är ganska strikta, speciellt regeln att man *måste* implementera superklassens abstrakt metoder i *alla* subklasser. I vissa applikationer vill man inte göra det. Man vill kanske implementera superklassens metoder i *några*, men inte i alla subklasser. Då kan man inte använda abstrakta metoder. Eller kanske vill man ha en del av metodkroppen i superklassen och en annan del i en subklass. För att kunna använda objektorientering grad- eller delvis, finns det i C# en lite svagare variant av **abstract** som kallas **virtual**. Med virtuella metoder *kan* man överskugga superklassens metod, man *måste inte* göra det. Som vi vet innebär polymorfism att man även kan *delvis* modifiera superklassens metod. Här är definitionen från *Progr1, Appendix A*:

> Polymorfism modifierar helt eller delvis funktionaliteten hos metoder med samma namn som förekommer i en arvhierarki.

Vi ska nu ge ett exempel på en virtuell metod. Eftersom den inte är abstrakt, utan ”bara” virtuell, behöver den inte heller definieras i en abstrakt klass. Därför skriver vi den i följande icke-abstrakt klass:

```
// SuperV.cs
// Icke-abstrakt superklass som definierar en virtuell metod som
// sedan ska överskuggas i en subklass
// Överskuggning sker med virtuell metod här i superklassen OCH
//                    med override           i subklasserna
using System;

class Super
{
   public int number;                 // Datamedlem: Initieras
                                      // automatiskt till 0
   public virtual void Method()       // Virtuell metod med kropp
   {
      Console.WriteLine("\n\tSuper:s Method(): Initialvärde = " +
                                                        number);
      number++;
      Console.WriteLine("\tSuper:s Method(): Uppdaterat värde = "
                                                      + number);
   }
}
```

Metoden **Method()** är vituell och behöver – till skillnad från en abstrakt metod – inte vara i en abstrakt klass. Därför är klassen **Super** inte abstrakt. En annan skillnad är att **Method()**:s kropp definieras i klassen **Super** och inte utanför. Men precis som abstrakta metoder får även **Method()** inte vara privat. Så långt till skillnaderna och de gemensamma egenskaperna hos abstrakta och virtuella metoder. Förresten, någon virtuell klass finns inte. Det finns bara virtuella metoder.

Överskuggning av virtuell metod

Nu ska klassen **Super** få en subklass där vi definierar om den virtuella metoden **Method()**:

```
// Sub.cs
// Subklassen Sub ärver Super och modifierar dess metod Method()
// Överskuggning sker med virtual      i superklassen OCH
//                    med override här i subklassen
// Testa gärna:   Ta bort virtual  från Super:s Method() och
//                        override från Sub:s   Method()
using System;

class Sub : Super                       // Sub ärver Super
{
   public override void Method()        // override nödvändigt för
   {                                    // överskuggning
      Console.WriteLine("\n\tSub:s Method(): Initialvärde = " +
                                                        number);
      number--;
      Console.WriteLine("\tSub:s Method(): Uppdaterat värde = " +
                                                  number + "\n");
   }
}
```

Metoden **Method()** får i subklassen **Sub** en ny kropp som skiljer sig från superklassens **Method()** i och med att datamedlemmen **number** inte uppdateras till **1** utan till **-1**. För att **Sub**-klassens **Method()** ska kunna överskugga **Super**-klassens **Method()**, när den anropas, måste vi förse metodhuvudet med det reserverade ordet **override**.

Test av virtuell metod

Nu ska vi testa om överskuggning verkligen sker, dvs om vi med samma namnet **Method()** anropar två olika metoder, en gång **Method()** med ett **Super**-objekt, en annan gång med ett **Sub**-objekt:

```
// TestVirtual.cs
// Testar överskuggning av den virtuella metoden Method()
using System;

class TestVirtual
{
   static void Main()
   {
      Super a = new Super();
      a.Method();               // Anrop av Super:s Method()

      Sub b = new Sub();
      b.Method();               // Anrop av Sub:s Method()
      a = new Sub();            // a pekas om till ett Sub-objekt
```

```
        a.Method();                 // Anrop av Sub:s Method()
    }                               // Men: Anrop av Super:s Method()
}                                   // om Method() är icke-virtuell
```

Följande körning av **TestVirtual** visar att anropet **a.Method()** uppdaterar datamedlemmen **number**:s värde till **1** och att anropet **b.Method()** ändrar värdet till **-1**:

```
Super:s Method(): Initialvärde = 0
Super:s Method(): Uppdaterat värde = 1

Sub:s Method(): Initialvärde = 0
Sub:s Method(): Uppdaterat värde = -1

Sub:s Method(): Initialvärde = 0
Sub:s Method(): Uppdaterat värde = -1
```

Detta visar att det första anropet **a.Method()** anropar **Super**-klassens **Method()**, eftersom **a** är en referens till ett **Super**-objekt, medan det andra anropet **b.Method()** anropar **Sub**-klassens **Method()**, eftersom **b** är en referens till ett **Sub**-objekt. Det andra anropet är ett exempel på överskuggning. Därför är resultaten olika.

I det tredje och sista anropet **a.Method()** har **a** genom ompekningen **a = new Sub()** strax innan blivit en referens till ett **Sub**-objekt. Även här sker som väntat en överskuggning dvs **a.Method()** anropar **Sub**-klassens **Method()**, eftersom **a** pekar på ett **Sub**-objekt.

Men skulle man ta bort **virtual** från **Super**:s **Method()** i filen **SuperV.cs** (sid 146) och **override** från **Sub**:s **Method()** i filen **Sub.cs** (sid 147) blir det tredje anropets körresultat annorlunda:

```
Super:s Method(): Initialvärde = 0
Super:s Method(): Uppdaterat värde = 1

Sub:s Method(): Initialvärde = 0
Sub:s Method(): Uppdaterat värde = -1


Super:s Method(): Initialvärde = 0
Super:s Method(): Uppdaterat värde = 1
```

I det tredje anropet **a.Method()** blir **Super**:s **Method()** anropad, fast **a** pekar på **Sub**. Det uppdaterade värdet **1** visar att det inte sker någon överskuggning. Detta beror på att vi inte explicit skriver **virtual** i **Super**:s **Method()** och inte heller **override** i **Sub**:s **Method()**. Och referensen a pekar pga sin **Super**-typ på ett **Super**-objekt och inte på ett **Sub**-objekt. Testa detta gärna själv.

Övningar till kapitel 3

3.1 Följande program är inte modulariserat:

```
// Non_modularized_1.cs
// Läser in två heltal, gör beräkningar med dem och skriver ut
// resultaten 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 osv.
// Innehåller ytterligare räkneoperationer
// Kan så småningom vidareutvecklas till en liten kalkylator

using System;
class Non_modularized_1
{
   static void Main()
   {
      Console.Write("\n\tMata in ett heltal:\t\t");     // Ledtext
      int no1 = Convert.ToInt32(Console.ReadLine());    // Input
      Console.Write("\n\tMata in ett heltal till:\t");
      int no2 = Convert.ToInt32(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");
   }
}
```

Modularisera programmet **`Non_modularized_1`** för att vidareutveckla det till en liten kalkylator (fast i konsolen): Separera beräkningarna, t.ex. multiplikationen från kodens andra delar dvs från *input* och *output*.

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()`**.

3.2 Modularisera programmet **Non_modularized_2** 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 metoden ett beskrivande namn.

```
// Non_modularized_2.cs
// Läser in tiden i antal år, månader och veckor, omvandlar den
// till antal dagar och skriver ut resultatet.
// Använder ett aritmetiskt uttryck för beräkning av antal dagar.
// Inmatning - bearbetning - utmatning. Nästlat anrop av metoder.
using System;

class Non_modularized_2
{
   static void Main()
   {
      int  years, months, weeks, days, totalDays;

      /* I n m a t n i n g */
      Console.Write("\n\tAnge antal år:\t\t");     // Ledtext
      years = Convert.ToInt32(Console.ReadLine()); // Nästlat anrop

      Console.Write("\n\tAnge antal månader:\t");
      months = Convert.ToInt32(Console.ReadLine());

      Console.Write("\n\tAnge antal veckor:\t");
      weeks = Convert.ToInt32(Console.ReadLine());

      Console.Write("\n\tAnge antal dagar:\t");
      days = Convert.ToInt32(Console.ReadLine());

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

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

3.3 a) Vänd om problemet från övn 3.2. Dvs Omvandla en tid som är angiven i dagar till år, månader, veckor samt resterande dagar. Skriv ett icke-modulariserat program **Non_modularized_3**, 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.

b) Modularisera programmet **Non_modularized_3** (lösningen från a) 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 **`Non_modularized_3`**. Varför är det inte lämpligt här att använda en metod med returvärde?

3.4 Skriv först ett program med endast **`Main()`**-metoden som läser in **`side`** till en kub samt beräknar och skriver ut kubens volym **`side`**^{3} och dess yta **`6`** x **`side`**^{2}. Flytta sedan dessa beräkningar till två metoder, en för volymen, en för ytan, båda i en separat klass **`Cube`**. Deklarera **`side`** som en datamedlem i klassen **`Cube`**. Avgör om metoderna **`Volume()`** och **`Surface()`** 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.

3.5 Modularisera programmet **`Non_modularized_3`** efter eget godtycke.

Kapitel 4

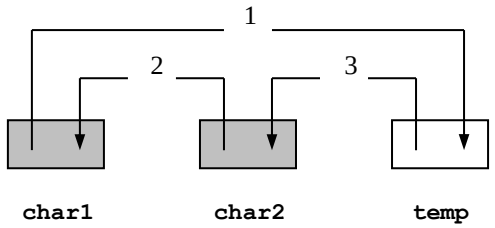
Mer om metoder

4.1 Algoritm för platsbyte

Hur kan man byta plats på två objekt när de står i ”fel” ordning? Detta är kärnfrågan i alla försök att *sortera* data. Och *sortering* är en av de mest efterfrågade uppgifterna i programmering som är dessutom besläktad med *sökning*. Vi vill här lägga grunden till en sök- och sorteringsalgoritm som vi kan använda på alla möjliga objekt. Men vi börjar med teckentabellen för att lära känna principen och begränsar oss till *två* tecken – till att börja med. I följande program formulerar vi algoritmen för platsbyte av två tecken först utan metoder och kommer i nästa avsnitt att modularisera koden, för att ha algoritmen som en metod som kan anropas av andra program.

Algoritmen

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.

Programmet

```
// 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 str = Console.ReadLine();
```

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

      if (char1 > char2)          // tecknen tolkas som tal
      {
          temp  = char1;          // Algoritm för platsbyte
          char1 = char2;          // av två tecken char1, char2
          char2 = temp;
      }

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

Själva sorteringsalgoritmen finns i **`if`**-satsen av programmet **`MiniSort`**. Om de två tecknen blir inmatade i rätt ordning, ska de inte byta plats utan skrivas ut i oförändrad ordning. Därför tas upp i **`if`**-satsens villkor endast fallet **`char1 > char2`** dvs när tecknen är inmatade i fel ordning. Följande körexempel sorterar de inmatade tecknen **`Z`** och **`A`** i rätt dvs i Unicode-tabellens 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.

4.2 Värde- och referensanrop

I det här avsnittet ska vi lära oss *på vilket sätt* parametrar överförs mellan metoder. Det finns nämligen i C# olika typer för parameteröverföring, en av dem är *värdeanrop* (*Call by Value*) som demonstreras i följande program där **`Main()`** anropar en metod där man kan studera parametereröverföringen.

```
// CallByVal.cs
// Demonstrerar Värdeanrop: Vid metodanrop överförs VÄRDENA
// De formella parametrarna (kopior) ändras i metoden
// Men ändringen påverkar inte aktuella parametrarna (originalen)
using System;

class CallByVal
{
  static void Main()
  {
    int hour = 5, min = 35, sec = 49;

    Console.WriteLine("\nI Main() FÖRE anrop av metod:\ttim="  +
                        hour + ", min=" + min + ", sec=" + sec);

    int total = totalsek(hour, min, sec); // Anrop av metoden: De
                                          // aktuella parametrar-
                                          // nas VÄRDEN skickas
    Console.WriteLine("\nI Main() EFTER anrop av metod:\ttim="  +
      hour + ", min=" + min + ", sec=" + sec + "\n\t\t\t\tger " +
       total + " sekunder totalt.\nVÄRDEANROP:\n\nÄndringen av" +
       "  de formella parametrarna (kopior)\npåverkar inte de " +
                      "aktuella parametrarna (originalen).\n") ;
  }
/*************************************************************/
  static int totalsek(int t, int m, int s)
  {
    Console.WriteLine("\n\tI metoden FÖRE ändringen:\n\tt="  +
                                t + ", m=" + m + ", s=" + s);
    int resultat = 3600 * t + 60 * m + s;
    t = m = s = 0;                        // Ändring av formella
                                          // parametrar
    Console.WriteLine("\n\tI metoden EFTER ändringen:\n\tt=" +
                                t + ", m=" + m + ", s=" + s);
    return resultat;
  }
/*************************************************************/
}
```

Varför har vi valt andra namn för de aktuella **`hour, min, sec`** än för de formella parametrarna **`t, m, s`** fast de lagrar samma värden? Båda representerar timmar, minuter och sekunder. Frågan är: Lagras dessa värden i 3 eller 6 minnesceller? Om det är 3 vore valet av samma namn motiverat, därför att de lagrar samma värden. Men om det är 6 vore

det bättre att återspegla verkligheten även i koden genom att välja olika namn för de aktuella än för de formella parametrarna.

Parametrar som skrivs i en metods *anrop* – i vårt exempel **`hour, min, sec`** – kallade vi för *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 väl definierade variabler eller konstanter. I exemplet ovan läses in de i **`Main()`**. De formella parametrarna – i vårt exempel **`t, m, s`** – måste alltid vara variabler som definieras i metoden **`totalsek()`**:s parameterlista när denna skapas. Sina värden får de *första gången* inte tilldelade i metodens kropp utan från de aktuella parametrarna vid metodens anrop. Sedan ändras deras värden i metoden: De sätts allihop till 0 för att testa vilken påverkan denna ändring har på de formella parametrarna. Men för att ändå kunna få resultatet med de ursprungliga värdena beräknas antalet totalsekunder och sparas undan i variabeln **`resultat`** som slutligen returneras från metoden. Innan dess skrivs ut värden som ändrats till 0.

I **`Main()`** skriver vi ut de aktuella parametrarnas värden före och efter anropet av metoden för att se om de formella parametrarnas ändring i metoden påverkar de aktuella parametrarna. Följande körexempel visar att detta inte är fallet:

```
I Main() FÖRE anrop av metod:   hour=5, min=35, sec=49

        I metoden FÖRE ändringen:
        t=5, m=35, s=49

        I metoden EFTER ändringen:
        t=0, m=0, s=0

I Main() EFTER anrop av metod:  hour=5, min=35, sec=49
                                ger 20149 sekunder totalt.
VÄRDEANROP:

Ändringen av  de formella parametrarna (kopior)
påverkar inte de aktuella parametrarna (originalen).
```

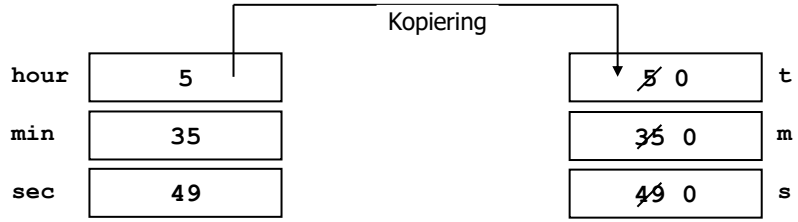
Körexemplet visar att de formella och aktuella parametrarna har var sitt eget liv. Det enda som relaterar dem till varandra är att de tar över värdena från varandra. Ändringen av de formella parametrarna påverkar inte alls de aktuella parametrarna. Av detta kan man dra slutsatsen att **`hour, min, sec`** och **`t, m, s`** är två olika uppsättnigar variabler. De lagras i 6 olika minnesceller. Även om vi skulle välja samma namn för dem – vilket vore tillåtet då de ligger i två olika metoder och därmed i två olika block – kommer

[*] Andra beteckningar som förekommer i litteraturen är *anropsparametrar* eller *argument*. Speciellt *argument* används ofta då det är en inkörd matematisk term: T.ex. är √3 ett anrop av funktionen √x där x – i matematiska termer – är ”variabeln” och 3 ”argumentet”. I programmeringstermer skulle x kallas för den *formella* och 3 den *aktuella* parametern.

namnen fortfarande beteckna 6 olika minnesceller. Även om beteckning är av sekundär betydelse vill vi i fortsättningen välja andra namn för de aktuella än för de formella parametrarna för att återspegla denna verklighet. Kodens läsare ska inte luras som om de vore samma variabler pga namnvalet.

En annan slutsats av körningen ovan är: Parameteröverföringen mellan metoderna **`totalsek()`** och **`Main()`** realiseras genom kopiering av värdena från de aktuella till de formella parametrarna. Denna parameteröverföringsmetod kallas *värdeanrop* därför att det är själva *värden* som kopieras över när metoden aropas. Minnesbilden av värdeanrop ser ut så här:

Värdeanrop:



Ändring av kopiorna, de formella parametrarna **`t, m, s`**, påverkar inte originalen, de aktuella parametrarna **`hour, min, sec`**.

Vid denna parameteröverföringsmetod skapas alltid en *dubbel* uppsättning av minnesceller: 6 om vi har 3 parametrar. Därför leder värdeanrop oundvikligen till fördubblad minnesåtgång. Datatypen till respektive parameter är avgörande för den automatiska tillämpningen av värdeanrop. Det gäller följande regel:

> I C# väljs automatiskt värdeanrop (Call by Value) för parameteröverföring vid metodanrop, om parametern är av enkel datatyp.

Fördubblingen av minnesåtgången anses inte som ett stort problem eftersom enkla datatyper i alla fall tar upp relativt litet minnesutrymme. För datatyper som kräver större minnesutrymme används en annan teknik som undviker denna fördubbling och som heter *referensanrop*.

Ur minnessynpunkt är förstås fördubblingen av minnesåtgången en nackdel. Men värdeanrop har även fördelen att just pga minnesbilden ovan de formella och de aktuella parametrarna har var sitt liv och inte påverkar varandra. I vissa sammenhang är detta önskvärt, i andra inte. Så, beroende på applikationen kan man välja bland de två parameteröverföringsmetoderna värde- och referensanrop genom att välja rätt datatyp till sina parametrar. Enkel datatyp leder automatiskt till värdeanrop. Vilken datatyp som automatiskt leder till referensanrop ska vi ta upp på de följande sidorna.

Referensanrop (Call by reference)

Värdeanrop använder sig av kopiering av parametervärdena till nya minnesceller och tillämpas när parametrarna är enkla datatyper. Nackdelen med värdeanrop är att den medför fördubbling av minnesåtgången. Alternativet till det är *referensanrop* som överför minnesadressen istället för värdet och där man slipper denna nackdel. Referensanrop är relaterad till datatypen referens som behandlades tidigare varifrån också namnet härstammar. Anledningen är att parametrarnas datatyp automatiskt styr valet av överföringsmetoden. Det gäller nämligen:

> I C# väljs automatiskt referensanrop (Call by reference) för parameteröverföring vid metodanrop, om parametern är av datatypen referens.

Samtidigt kommer vi att se att det för vissa problem t.o.m. är nödvändigt att använda referensanrop då det inte går att modularisersa dem med värdeanrop. Man vill t.ex. skicka vissa parametrar till en metod där de ändras och man vill få tillbaka ändringen till huvudprogrammet. Ta följande exempel: Vi vill skicka två parametrar till en metod som ska sortera dem. Skickar vi dem i fel ordning ska metoden ställa dem i rätt ordning och skicka tillbaka dem i den rätta ordningen – grunden till alla sorteringsalgoritmer. Ett exempel på ett sådant problem som vi ska ta upp här, är modulariseringen av program **`MiniSort`** (sid 154) som presenterade en algoritm för platsbyte mellan två tecken. Vi ska nu separera själva algoritmen och skriva den som en metod med tanke på att den kommer att utvecklas till en allmän sorteringsalgoritm för större datamängder senare. Det programmeringstekniska verktyget för att få parametrar av typen referens är det reserverade ordet **`ref`** som sätts framför parameterdeklarationerna och fungerar som en slags adressoperator: **`ref char t1`** blir *adressen* till **`char`**-parametern **`t1`**:

```
// Swapping.cs
// Klass med metoden Swap() som tar in 2 tecken och byter plats
// på dem om de kommer in i fel ordning enligt Unicode-tabellen
// De ombytta parametrarna i Swap() blir även ombytta i den
// anropande metoden pga parametrarna är deklarerade som
// referenser med det reserverade ordet ref: Referensanrop

class Swapping
{
   public static void Swap(ref char t1, ref char t2)
   {
      char temp;
      if (t1 > t2)
      {
         temp = t1;                 // Algoritm för platsbyte
         t1   = t2;                 // av de två teckenvärdena
         t2   = temp;               // t1 och t2
      }
   }
}
```

Bearbetningsdelen av **MiniSort** (sid 154) har flyttats till en **void**-metod. Parametrarna **t1** och **t2** är definierade som referenser. De tar inte emot några teckenvärden från **char1** och **char2** (se nedan) utan endast deras adresser. **t1** och **ref char1** är två olika referenser till samma värde **char1**. Samma sak är det med **t2** och **ref char2**. När värdena ändras i metoden med hjälp av referenserna **t1** och **t2** kan ändringen ses i **Main()** med **char1** och **char2**:

```
// CallByRef.cs
// Läser in 2 tecken, skickar dem till metoden Swap() i klassen
// Swapping som sorterar dem i teckentabellens ordning
// Ändringen är synlig i Main() pga referensanrop som påtvingas
// med ref så att adresserna skickas vid anrop, inte värdena
using System;

class CallByRef
{
   static void Main()
   {
      char char1, char2;
      Console.Write("\n\tTvå osorterade tecken:\n\n\t"      +
              "Mata in två tecken skilda med mellanslag:\t");
      string str = Console.ReadLine();

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

      Swapping.Swap(ref char1, ref char2);      // Metodanrop

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

Metoden **Swap()** ställer i rätt ordning tecken som är inmatade i fel ordning vilket en körning av ovanstående program visar:

```
       Två osorterade tecken:

       Mata in två tecken skilda med mellanslag:       Z A

       De två tecknen sorterade:                       A Z
```

Gör gärna följande test: Ta bort **ref** från definitionen av båda parametrarna i parameterlistan av metoden **Swap()**, så att **t1** och **t2** blir vanliga **char**-variabler. Ta även bort **ref** från de aktuella parametrarna i anropet av metoden **Swap()** i **Main()** så att värdena skickas och inte adresserna. Du kommer inte få tecknen sorterade i rätt ordning om du matar in dem i fel ordning. Anledningen är att genom borttagningen av **ref** blir **t1** och **t2** variabler av *enkel* datatyp så att värdeanrop tillämpas automatiskt. Ändringen av **t1** och **t2** i metoden kommer inte att påverka **char1** och **char2** i **Main()**.

4.3 In- och utparametrar

Nu har vi lärt oss en hel del om metoder, med och utan returvärde, med en, flera eller inga parametrar, värde- och referensanrop osv. Ändå kan vi inte returnera *flera* värden från en metod. Det beror på att alla metoder i C# returnerar endast ett eller inget värde. Men för att vara mer noggrant, borde vi lägga till *med* **`return`**-*satsen*. Begreppet *returvärde* används i programmeringsterminologin endast för värden som skickas med **`return`**-satsen via metodnamnet. I denna bemärkelse finns det inga metoder med *flera* returvärden. Men metodens gränssnitt mot omgivningen dvs mot andra metoder är inte begränsad till metodnamnet. Även parameterlistan tillhör gränssnittet och kan användas för kommunikation med andra metoder. Hittills har denna kommunikation varit *enkelriktad*: Våra parametrar importerade data bara in i metoden. Frågan är: Kan man inte använda dem även för export av data ut ur metoden? I så fall skulle vi kunna få tillbaka även *flera* värden från en metod genom att använda *flera* parametrar. Detta är möjligt fast man kallar sådana data inte längre för returvärden då de inte skickas med **`return`**-satsen via metodnamnet, utan via parametrarna. De kallas för *utparametrar*. Hittills har vi använt bara *in*parametrar. I detta avsnitt ska vi lära känna *ut*parametrar. Verktyget som behövs för det är datatypen referens som behandlats tidigare (sid 100). Det enda som behövs för att kännetecknа en parameter som *ut*parameter är nämligen att definiera den i parameterlistan som *referens* vilket kan göras med **`ref`** eller **`out`**.

I följande metod finns det en inparameter som tillför metoden ett värde och fem utparametrar vars värden exporteras ur metoden. De kommer in i metoden oinitierade, initieras där och används sedan i **`Main()`** som anropar metoden. I själva verket är utparametrarna endast referenser till de aktuella parametrarna i **`Main()`**. Där är de endast definierade. I metoden sker initieringen med referenserna.

```
// Outparam.cs
// Tar in växelbeloppet a och delar upp det i antalet t 10-
// kronor, f 5-kronor, o 1-kronor, h 50-öringar och
// resten r i öre. Endast b är en inparameter pga enkel datatyp
// t, f, o, h och r är utparametrar pga referensdatatypen out int

class Outparam
{
   public static void Change(double a, out int t, out int f,
                                       out int o, out int h,
                                                  out int r)
   {
      int total = (int) (a * 100);              // växel som int
      t =    total / 1000;                      // 10-kronor
      f =   (total % 1000) / 500;               // 5-kronor
      o =  ((total % 1000) % 500) / 100;        // 1-kronor
      h = (((total % 1000) % 500) % 100) / 50;  // 50-öringar
      r = (((total % 1000) % 500) % 100) % 50;  // rest i öre
   }
}
```

Den reala bakgrunden till metoden är följande problem: I en automat erbjuds vissa varor. Man väljer en vara och stoppar in en viss summa pengar, i regel mer än varan kostar. Sedan ska automaten ge tillbaka växelpengar vilket endast är möjligt med ett antal myntslag som är föreskrivna i automaten. Låt oss säga det är 10-, 5-, 1-kronor och 50-öringar (Läs fotnot på sid 122). I så fall måste växelbeloppet omvandlas till detta mynt-”system”. Just denna beräkning utförs av **`void`**-metoden **`Change()`** ovan. Men hur genomförs omvandlingen med de uttryck för **`t, f, o, h`** och **`r`** som står i metoden? Följande algoritm som redan nämndes i *Automaten*, övn 8.8 (sid 123), löser problemet:

Algoritm för omvandling av ett belopp till olika myntslag

Eftersom denna algoritm endast fungerar för heltal måste växelbeloppet **`b`** som är en **`double`** först konverteras till **`int`**, vilket görs i metoden **`Change()`**:s första sats explicit eftersom automatisk typkonvertering inte kan omvandla nedåt i datatypshierarkin. Växelbeloppet i kronor och ören konverteras till ett rent örebelopp som lagras i **`int`**-variabeln **`total`**. I fortsättningen står alltså det givna växelbeloppet i variabeln **`total`**.

1. För att få antalet 10-kronor divideras **`total`** med **`1000`** då 10-kronor är **`1000`** ören:

```
t = total / 1000;
```

Hur många gånger ryms **`1000`** – eller 10-kronor – i **`total`**? Det antalet tilldelas till **`t`**. Eller med andra ord: **`1000`** dras av från **`total`** så många gånger tills resten blivit mindre än **`total`**. Det antalet som tilldelas till **`t`** blir antalet 10-kronor. Divisionen ovan är inte vanlig division utan heltalsdivision då både **`total`** och **`1000`** är heltal. Dvs **`total`** divideras med **`1000`**, resultatet tas, resten ignoreras, t.ex. **`6975/1000`** ger **`6`**. Se körexemplet på nästa sida. Resten **`975`** ignoreras här, men används i fortsättningen.

2. För att få antalet 5-kronor divideras resten som blev kvar från punkt **1** med **`500`** då 5-kronor är **`500`** ören:

```
f = (total % 1000) / 500;
```

”Resten som blev kvar från punkt **1**” är just **`(total % 1000)`**. Här används en annan operator som är besläktad med heltalsdivision, nämligen modulooperatorn **`%`** (sid 124). **`%`** har ingenting att göra med procenträkning utan ger *resten* vid heltalsdivision. 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 **`f`**, resten ignoreras och används i fortsättningen.

I ytterligare tre steg skulle man kunna förklara de övriga formlerna för beräkning av **`e`**, **`h`** och **`r`**. Men nu har mönstret i algoritmen kommit fram: 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. Läs om heltalsdivision **`/`** och modulooperatorn **`%`** på sid 124.

För att testa algoritmen ovan anropas metoden **`Change()`** av följande program:

```
// OutparamTest.cs
// Efter inköp av en vara i en automat ska växeln ges tillbaka
// i form av ett antal föreskrivna myntslag:
// 10-kronor, 5-kronor, 1-kronor, 50-öringar (och en rest i öre)
// Main() läser in ett växelbelopp, skickar det till metoden
// Change() i klassen Outparam som omvandlar växeln till mynt
using System;

class OutparamTest
{
   static void Main()
   {
      double amount;
      int ten, five, one, half, rest;  // Ingen initiering behövs
      Console.Write("\nAnge ett växelbelopp i kronor, ören:  ");
      amount = Convert.ToDouble(Console.ReadLine());

      Outparam.Change(amount, out ten, out five,  // Endast ut-
                              out one, out half,  // parametrar-
                                       out rest); // nas adresser
                                                  // skickas
      Console.WriteLine("\n" + amount + " kr =\t"            +
                        ten  + " tio-kronor\n\t\t"           +
                        five + " fem-krona\n\t\t"            +
                        one  + " en-kronor \n\t\t"           +
                        half + " femtio-öring\n\nDet blir\t" +
                        rest + " ören kvar\n");
   }
}
```

Växelbeloppet läses in. Metoden **`Change()`** anropas varvid förutom **`belopp`** de aktuella parametrarna **`ten`**, **`five`**, **`one`**, **`half`** och **`rest`**:s *adresser* skickas. Dessa tas emot i **`Change()`** av **`t`**, **`f`**, **`o`**, **`h`** och **`r`**, dvs referenserna till **`ten`**, **`five`**, **`one`**, **`half`** och **`rest`**. När beräkningen görs där med hjälp av referenserna kan man komma åt resultaten i **`Main()`** därför att **`t`** är en referens till **`ten`**. Samma sak är det med de övriga parametrarna.

Ett körexempel visar att vi verkligen får tillbaka till **`Main()`** de värden som beräknas i metoden pga referensanrop som automatiskt tillämpas vid utprametrar av referenstyp.

```
Ange ett växelbelopp i kronor, ören:  69,75

69,75 kr =      6 tio-kronor
                1 fem-krona
                4 en-kronor
                1 femtio-öring

Det blir        25 ören kvar
```

Vad man sedan gör med det sista restörebeloppet beror på teknikaliteter i automaten.

4.4 Variablers livslängd

Alla program i C# går ut på att ett antal objekt kommunicerar med varandra genom att använda vissa egenskaper, funktionaliteter eller färdigheter de förfogar över – precis som i det verkliga livet. Programmeringstekniskt sett sker denna kommunikation i och med att objekt anropar andra objekts metoder. I större program kan detta jämföras med trafikflödet i en storstad. Ingen storstadstrafik fungerar utan trafikregler. På liknande sätt finns det i C# strikta regler vad gäller samverkan mellan objekt och deras metoder, närmare bestämt mellan objektens och metodernas variabler samt mellan variablerna sinsemellan. Dessa regler tillämpas automatiskt och blir speciellt påtagliga, när en metod anropas *nästlad* i en annan metod, t.ex. i satsen

```
Convert.ToInt32(Console.ReadLine());
```

Detta gjorde vi för att kunna läsa in ett heltal till ett program. Metoden **`ReadLine()`** tar emot en sträng som inmatning varför den måste omvandlas till heltal. Men då handlade det om ett nästlat anrop av *bibliotek*smetoder. Nu ska vi kunna göra samma sak med *egendefinierade* metoder.

I detta avsnitt ska vi studera C#:s regler för livslängden eller räckvidden av variabler i olika block. För att kunna göra det ska vi aktualisera vår förståelse om *block(struktur)*. I detta sammanhang ska vi även belysa skillnaden mellan datamedlemmar och lokala variabler. I nästa avsnitt kommer vi att ta upp ett fall av namnkonflikt mellan variabler som är en konsekvens av blockstruktur och som kallas för *överskuggning av variabler*, inte att förväxla med *överskrivning* av variabler *(Progr1+, 4.5)*.

Blockstruktur

Vad är ett *block* i C# ?

> Ett antal satser som omsluts av klamrarna **{** och **}** kallas för ett *block*. … Klamrarna är *gränser* mellan programmets olika block. De sätter gräns för variablers livslängd. För att överskrida dem måste vissa regler beaktas, se nedan.

Vad händer med en variabel när man överskrider blockgränserna? Hur långt går en variabels *räckvidd* (eng. *scope*)? Man pratar om variablers *livslängd*. Generellt gäller:

Regler för livslängden (scoping) av variabler:

Variablers livslängd börjar med deklarationen och slutar med det block i vilket de är deklarerade. De är giltiga (synliga) i det block de är deklarerade och i alla underblock, men inte i överordnade block.

Variabler "söker" efter sin deklaration uppåt i blockstrukturen.

Låt oss testa dessa regler i ett program som i **`Main()`** skapar ett inre (undre) block. Själva **`Main()`** skulle man kunna beteckna som det yttre (övre) blocket:

```
// Block.cs
// Variablers räckvidd (scoping) och blockstruktur i C#
using System;

class Block
{
   static void Main()
   {
      int x = 10;              // x gäller i hela Main() och
                               // i alla dess underblock
      String output = "\tUtskrift utanför det inre blocket:" +
                      "\n\n\tx före det inre blocket är " + x;
/*************************************************************/
      {                        // Här börjar det inre blocket
         int y = 100;          // y gäller endast i inre blocket
         x++;                  // Men x gäller även här
         Console.WriteLine(
                "Utskrift från det inre blocket:\n\n"   +
                "x är " + x + " och y är " + y + "\n\n");
      }                        // Här slutar det inre blocket
/*************************************************************/
                               // y gäller inte längre f.o.m.här:
      Console.WriteLine(output +
             "\n\tx efter det inre blocket är " + x + "\n\n");
   }
}
```

I det yttre blocket definieras variabeln **`x`** och initieras till **`10`**, i det inre blocket definieras variabeln **`y`** och initieras till **`100`**. I koden markerar den inledande klammern **`{`** strax före **`y`**:s definition början och den avslutande klammern **`}`** slutet på det inre blocket. För bättre läslighetens skull skiljs dessutom blocken åt med kommentarrader fyllda med stjärnor. Utskriften från det inre blocket leds direkt till konsolen medan all utskrift *utanför* det inre blocket samlas i **`String`**-variabeln **`output`** och skrivs ut sist. Resultatet blir:

```
Utskrift från det inre blocket:

x är 11 och y är 100


        Utskrift utanför det inre blocket:

        x före det inre blocket är 10
        x efter det inre blocket är 11
```

Som man ser har i programmet **`Block`** metoden **`Main()`**:s lokala variabel **`x`** med initialvärdet **`10`** ”trängt genom” det inre blocket där dess värde ökats till **`11`** och har *efter* det inre blocket det ökade värdet **`11`**, vilket är ett exempel på regeln: Variabler är giltiga i

det block de är deklarerade och i alla underblock. **x** är i hela **Main()** en och samma variabel, både i och utanför det inre blocket. Vi kallar **x** för *lokal* variabel i **Main()** för att skilja den från begreppet *datamedlem*. Hade **x** varit deklarerad tre rader innan hade den varit klassens datamedlem och inte metoden **Main()**:s lokala variabel. Vi hade då hamnat i ett överordnat block: Klassen kan uppfattas som överordnat block till alla dess metoder. Så det är deklarationens *plats* i koden som avgör skillnaden mellan datamedlemmar och lokala variabler. I andra programmeringsspråk som C++ måste man skilja mellan lokala och *globala* variabler, men:

I C# finns det inga "globala" variabler.

De har blivit ersatta av datamedlemmar. Man kan också säga att i strikt objektorienterade språk ersätter klassens datamedlemmar globala variabler. Klassens variabler är datamedlemmar (”globala”), medan metodernas variabler är lokala. Utanför klassen kan ingen kod skrivas (utom **using**-direktivet) och därmed kan inte heller någon variabel deklareras, vilket däremot är möjligt i C++.

Programmet **Block** har dessutom en variabel **y** som deklareras och initieras till **100** i ett block som är nästlat i **Main()**-blocket och därför kallas för *inre* eller *under*block. Dess livslängd går endast till slutet av detta underblock. *Efter* den avslutande klammern **}** som terminerar det inre blocket, är **y** inte längre giltigt. Varje förekomst av **y** efter klammern kommer att leda till kompileringsfel. Det är ett exempel på regeln: Variabler är inte giltiga i överordnade block: **Main()**-blocket är överordnat det inre blocket där **y** är deklarerad. **y** söker efter sin deklaration uppåt i blockstrukturen och hittar den i det inre blocket och är därmed det inre blockets lokala variabel. Därför kan vi skicka den till utskrift endast från det inre blocket. När vi gör det får vi dess värde **100** i konsolfönstret.

Variabeln **x** gör samma sak, söker t.ex. med satsen **x++;** som utgångspunkt efter sin deklaration uppåt i blockstrukturen, hittar den däremot inte i det inre blocket, söker därför vidare i det överordnade **Main()**-blocket och hittar den där. Därmed måste **x** betraktas som **Main()**-blockets lokala variabel. Den är giltig överallt i **Main()**, även i dess underblock. Därför kan vi skicka den till utskrift från det inre blocket efter uppdateringen av dess ursprungliga värdet **10** med **x++;** och får värdet **11** i konsolfönstret. Men till skillnad från **y** kan vi skriva ut **x** även *efter* det inre blocket i **Main()** och får samma värde **11**. En utskrift av **x** *före* det inre blocket – via **String**-variabeln **output** – hade gett värdet **10** pga att den skedde innan uppdateringen **x++;**

Blockstrukturen i programmet **Block** är konstruerad för att testa C#:s allmänna regler för variablers livslängd med ett så enkelt exempel som möjligt. Man inser inte nödvändigheten av blockbildningen. Men liknande och mycket mer komplicerade situationer kan uppstå om man istället för ett inre block har ett anrop av en metod vars kropp i så fall tar över rollen av det inre blocket när metoden anropas. Program med metoder som anropar andra metoder ger upphov till blockstruktur. Nästa avsnitt tar upp ett sådant fall.

4.5 Överskuggning av variabler

Här ska vi ta upp ett koncept som löser namnkonflikter vilka kan uppstå när lokala variabler och datamedlemmar har samma namn: *överskuggning* (eng. *overriding*) kallas det och förekommer inte bara hos variabler utan även hos metoder som defnieras i klasser som ärver varandra. Det sista tas upp senare när vi gått igenom arvbegreppet som är en hörnsten inom objektorienterad programmering. Som en slags förberedelse på det ska vi här bekanta oss med själva begreppet *överskuggning* genom att tillämpa det på variabler. På köpet kommer vi att lära känna C#:s reserverade ord **`this`**.

Titta på följande klass med tre datamedlemmar och två metoder där den ena, **`Main()`** anropar den andra, **`Inner()`**. Försök sedan med de kunskaper du fått i förra avsnitt om variablers livslängd, att besvara några frågor:

```
class OverrideVar
{
   double salary, bonus;                // Datamedlemmar gäller i
   String mess;                         // både Main() och Inner()

   static void Main()
   {
      OverrideVar y = new OverrideVar();       // Objekt skapas
      y.salary   = 60000;
      y.bonus = y.salary * 0.20;
      y.mess  = " lämplig för bonus";
(1)-->
      y.Inner();                        // Anrop av en annan metod
                                        // bildar inre block
      y.mess = "Den anställde " + y.mess;
(2)-->
   }

   void Inner()
   {
      double salary   = 50000;          // 2 nya lokala variabler
      double bonus = 0;                 // överskuggar datamedlem
      mess = "Andersson inte" + mess;   // Använder datamedlem
      this.bonus = salary * 0.30;       // this pekar på objektet
(3)-->                                  // kommer så åt datamedlem
   }
}
```

Följande frågor kan vara intressanta:

1. Vilka värden har variablerna **`salary`**, **`bonus`** och **`mess`** i position (1)?
2. Vilka värden har samma variabler i position (2)?
3. Vilka värden har **`salary`**, **`bonus`**, **`this.bonus`** och **`mess`** i position (3)?

Referensen `this`

En nyhet i programmet ovan är C#:s reserverade ord **`this`** som alltid är en *referens* till det objekt i vilket den står. Men var står **`this`** här? **`this`** står i metoden **`Inner()`** som anropas i **`Main()`** med **`y.Inner()`**. Därför blir **`this`** en referens till det objekt som **`y`** pekar på. Dvs **`this.bonus`** är en annan beteckning för **`y.bonus`**. Men **`y`** finns inte i **`Inner()`** pga reglerna för variablers livslängd. Därför: **`this.bonus`**. Detta för att skilja datamedlemmen **`bonus`** från **`Inner()`**:s lokala variabel **`bonus`**. Följande program besvarar frågorna ovan genom att skriva ut de efterfrågade värdena i resp. position:

```
// OverrideVar.cs
// Datamedlemmar överskuggas av lokala variabler med samma namn
using System;

class OverrideVar
{
   double salary, bonus;               // Datamedlemmar gäller i
   String mess;                        // både Main() och Inner()

   static void Main()
   {
      OverrideVar y = new OverrideVar();       // Objekt skapas
      y.salary      = 60000;
      y.bonus       = y.salary * 0.20;
      y.mess        = " lämplig för bonus";
      String output = "\tUtskrift från yttre Main()-blocket:"   +
                      "\n\n\tPosition 1 FÖRE anrop av Inner()\n"+
                      "\tlön = "   + y.salary   + '\n'          +
                      "\tbonus = " + y.bonus + '\n'             +
                      "\tmedd = "  + y.mess  + "\n\n"           ;

      y.Inner();                          // Anrop av en annan metod
                                          // bildar inre block
      y.mess = "Den anställde " + y.mess;
      Console.WriteLine(output + "\tPosition 2 "               +
       "EFTER anrop av Inner()\n" + "\tlön = " + y.salary + '\n'+
       "\tbonus = " + y.bonus + "\n\tmedd = " + y.mess + '\n')  ;
   }

   void Inner()
   {
      double salary   = 50000;         // 2 nya lokala variabler
      double bonus = 0;                // överskuggar datamedlem
      mess = "Andersson inte" + mess;  // Använder datamedlem
      this.bonus = salary * 0.30;      // this pekar på objektet
      Console.WriteLine("Utskrift från metoden Inner():"        +
         "\n\nPosition 3\nLokal salary   = " + salary           +
         "\nLokal bonus = " + bonus + "\nbonus = " + this.bonus +
         "\nmedd  = "       + mess  + '\n')                    ;
   }
}
```

Precis som programmet **Block** leder även programmet **OverrideVar** utskriften från det inre blocket – här metoden **Inner()** – direkt till konsolen medan all utskrift *utanför* det inre blocket samlas i **String**-variabeln **output** och skrivs ut sist. Därför får vi följande utskrift när vi kör programmet **OverrideVar**:

```
Utskrift från metoden Inner():

Position 3
Local salary = 50000
Local bonus  = 0
bonus = 15000
mess  = Andersson inte lämplig för bonus

        Utskrift från yttre Main()-blocket:

        Position 1 FÖRE anrop av Inner()
        salary = 60000
        bonus  = 12000
        mess   =  lämplig för bonus

        Position 2 EFTER anrop av Inner()
        salary = 60000
        bonus  = 15000
        mess   = Den anställde Andersson inte lämplig för bonus
```

För att få klarhet över de här resultaten måste vi kartlägga de olika variablerna och följa upp deras värden. Vilka variabler är inblandade i klassen **OverrideVar**? Samma namn behöver nämligen inte betyda samma variabel när olika block är inblandade och dessutom är nästlade. Man ser variablerna **salary**, **bonus**, och **mess** dyka upp på olika ställen i programmet. Men är de hela vägen de *samma* eller är det *olika* variabler med samma namn? Tillämpar vi våra kunskaper om variablers livslängd från förra avsnitt, kan vi konstatera följande: Det finns två *olika* variabler som har namnet **salary** och två *olika* variabler som har namnet **bonus**, en gång som datamedlemmar, en gång som lokala variabler i metoden **Inner()**. De refererar till *olika* minnesceller dvs olika fysiska adresser med samma logiska namn i olika block, vilket är tillåtet. Däremot finns det endast *en* variabel **mess** i hela programmet som är datamedlem och inte förekommer som lokal variabel. Här följer en detaljerad genomgång av programmet **OverrideVar** i den ordning som saker och ting händer när programmet körs:

1. I klassen **OverrideVar** gäller datamedlemmarna **salary**, **bonus**, och **mess** i princip i klassens alla metoder dvs i **Main()** och **Inner()**. Vi säger ”i princip”, därför att det finns undantag, t.ex. när metoden **Inner()** använder samma namn för sina lokala variabler. Då slår ut de lokala variablerna datamedlemmarna i metodens kropp och sätter dem temporärt ur spel. Därför pratar vi om överskuggning, se punkt **2**. När vi i **Main()** skapar ett objekt av typen **OverrideVar**, tilldelas objektets datamedlemmar följande värden:

salary	**60000**
bonus	**12000**
mess	**lämplig för bonus**

Detta visas också i utskriften på förra sidan under **Position 1 FÖRE anropet av Inner()**.

2. Sedan är anropet av metoden **Inner()** på tur som sker i **Main()** med satsen **y.Inner();** dvs **Inner()** anropas i det objekt som **y** pekar på. Metoden **Inner()** har två nya lokala variabler **salary** och **bonus** med samma namn som klassens datamedlemmar. I namnkonflikten mellan lokala variabler och datamedlemmar gäller följande regel i C#:

> Att överskugga betyder att slå ut temporärt. En lokal variabel i en metod överskuggar en datamedlem med samma namn.

Överskuggning (eng. *overriding*) bör inte förväxlas med överskrivning (eng. *overwriting*). Skillnaden är att överskuggning är temporär medan överskrivning är definitiv och oåterkallelig. Överskriva kan man bara variabelns *värde* t.ex. med en ny tilldelning. Då blir det gamla värdet överskrivet för gott, kan aldrig återskapas och det nya värdet gäller i fortsättning *(Progr1, 2.4)*. Överskuggning har inget att göra med variabelns värde utan med variabelns *giltighet*. I en metod kastar den lokala variabeln med samma namn en ”skugga” över datamedlemmen, fast temporärt dvs i metodens kropp. Bilden av skuggan ska betona fenomenets temporära karaktär. Före och även efter metodens anrop har datamedlemmen sin fulla giltighet. I metoden **Inner()** initieras de ”egna” lokala variablerna **salary** och **bonus** till:

Local salary	**50000**
Local bonus	**0**

Men vad gäller variabeln **mess** som används i **Inner()** har ingen ny definition av den skett i metoden. Därför är **mess** här *samma* variabel som gäller i hela klassen, nämligen datamedlemmen **mess**. Den hade redan i objektet som skapades i **Main()**, fått värdet **lämplig för bonus**, se punkt **1**. Men hur vet vi att det är *samma* objekts datamedlem som vi har att göra med i metoden **Inner()**? Där i **Main()** hade vi refererat till objektet med **y** och därmed till datamedlemmen med **y.mess**. Men referensen **y** gäller bara i **Main()** och är inte tillgänglig i **Inner()**. Här i **Inner()** refereras till datamedlemmen med **mess** utan **y**. Svaret är att vi ”här i **Inner()**” hela tiden befinner oss i det objekt som **y** pekar på eftersom anropet i **Main()** har ursprungligen skett med **y.Inner()**. Därför är **mess** här *en och samma* variabel **y.mess** i **Main()** som enligt punkt **1** hade värdet **lämplig för bonus**. Detta

värde överskrivs nu i satsen `mess = "Andersson inte" + mess;` och uppdateras genom konkatenering till:

mess	Andersson inte lämplig för bonus

Efter den här ändringen av `mess` följer i metoden `Inner()` satsen `this.bonus = salary * 0.30;` som ändrar variabeln `bonus`' värde. Men vilken variabel `bonus` är det? Frågan besvaras av `this` som är en referens till det objekt som `y` pekar på, därför att satsen i vilken `this` står, utförs i anropet `y.Inner()` (sid 168). `this.bonus` refererar i kroppen av metoden `Inner()` till `OverrideVar`-objektets datamedlem `bonus` för att skilja den från `Inner()`:s lokala variabel `bonus`. Utan `this` hade det blivit det lokala `bonus`. `this` hämtar det överordnade objektets datamedlem in i den lokala metoden och upphäver på så sätt dess överskuggning, vilket är en teknik som används ofta och som vi kommer att återkomma till senare. Vid beräkning av det nya värdet av `this.bonus` måste nu beaktas att `salary` (till höger om tilldelningstecknet) är den lokala variabel vars värde är `50000`. Därmed blir det följande uppdatering av datamedlemmen `bonus`:

bonus	15000

Alla värden som visades under punkt **2** skrivs ut till konsolen på förförra sidan under Position 3 Utskrift från metoden `Inner()`.

3. Slutligen skrivs ut objektets datamedlemmar *efter* anropet av metoden `Inner()` och efter uppdatereringen `mess` i `Main()`. Datamedlemmen `salary` har inte ändrats sedan före anropet av `Inner()`, inte heller *i* `Inner()` och har därmed samma värde som i `position 1`. Datamedlemmen `bonus` däremot har i `Inner()` fått ett nytt värde som fortsätter att gälla nu i `Main()` efter anropet. Även datamedlemmen `mess` ändrades i `Inner()` med hjälp av referensvariabeln `this` och uppdateras nu i `Main()` i satsen `y.mess = "Den anställde " + y.mess;` innan alla datamedlemmar skrivs ut, så att följande minnesbild visas på förförra sidan som utskrift från `Position 2 EFTER anropet av Inner()`:

salary	60000
bonus	15000
mess	Den anställde Andersson inte lämplig för bonus

Programmet `OverrideVar` använder referensvariablerna `this` och `y` som lagrar identiska adresser: `this` och `y` är två olika referenser till *samma* objekt (sid 168).

4.6 Överlagring av metoder

Överlagring av operatorer har vi nämnt tidigare. Då såg vi att t.ex. symbolen **+** betydde både additions- och konkateneringsoperatorn. Det var sammanhanget där symbolen användes, som avgjorde vilken av dessa betydelser som gällde. Även operatorn **/** är överlagrad: En gång som symbol för heltalsdivision, en gång för vanlig division. På samma sätt kan metoder vara överlagrade, t.ex. metoderna **`Console.WriteLine()`** och **`MessageBox.Show()`**. Även **`Next()`** och dess varianter som genererar slumptal på olika sätt är exempel på *överlagring av metoder* (eng. *overloading*).

> Överlagring av metoder innebär olika metoder med samma namn.
> De bildar en *metodgrupp*. Signaturen skiljer åt deras varianter.

Signaturen

Det som avgör om två metoder är identiska eller olika är metodens *signatur*, dvs:

- Metodens namn
- Antal parametrar
- Parametrarnas datatyper

Signaturen är alltså en metods igenkänningstecken. T.ex. har metoden **`public static String Encrypt(String t, int n)`** som vi använt tidigare, följande signatur:

```
Encrypt(String t, int n)
```

Signaturen ovan består av namnet **`Encrypt`**, antalet *två* (parametrar) och datatyperna **`String`** och **`int`**. OBS! Returtypen och modifierarna ingår *inte* i signaturen. Metoder med samma signatur anses vara *identiska*. Metoder som skiljer sig på *något* av signaturelementen anses vara *olika*. Två eller flera metoder i en och samma klass kan ha samma namn om deras parameterlistor är olika dvs om metoderna antingen har olika *antal* parametrar eller lika antal, men olika *datatyper*. Då *överlagrar* de varandra. En klass däremot med två metoder som har samma signatur kan inte kompileras.

Överlagring är ett koncept inom programmering som används för att koda funktionaliteter som är besläktade med varandra men ändå inte exakt identiska. Verkligheten är full av överlagring. Ta följande exempel: Att bromsa en lastbil görs på ett annat sätt än att bromsa en båt. Det finns ingen anledning att hitta på ett annat namn för funktionaliteten ”att bromsa” hos olika typer av fordon. Tvärtom, det vore t.o.m. förvirrande att använda olika namn. Vem skulle kunna komma ihåg alla dessa namn? Man vill ju helst slippa att tänka på de tekniska skillnaderna mellan olika typer av fordon när man pratar om bromsning. En och samma funktionalitet är *realiserad* på olika sätt. Med andra ord, man gör ”samma sak”, fast ändå lite annorlunda. Programmering tar över detta koncept genom att välja ett och samma namn för olika metoder. C#:s klassbibliotek är fullspäckat med överlagrade metoder. C#-kompilatorn skiljer åt överlagrade metoder genom den annorlunda parameterlistan och skickar automatiskt rätt anrop till rätt metod.

Följande program innehåller ett exempel på överlagring av C#:s biblioteksmetoder och två egendefinierade metoder som överlagrar varandra:

```
// Overload.cs
// Två exempel på överlagring av metoder:
// 1) 2 String-metoder med samma namn för delsträngbildning
// 2) 2 egendef. metoder med samma namn men olika parameterlistor
//    En beräknar potensen "bas upphöjd till int-exponent"
//    Den andra   potensen "bas upphöjd till double-exponent"
using System;

class Overload
{
   static void Main()
   {
      Console.WriteLine(
        "Överlagring av egendefinierad metod:\n\n"       +
        "2 upphöjd till 3   =  " + Power(2, 3)   + '\n'  +
        "2 upphöjd till 3.0 =  " + Power(2, 3.0) + '\n'  +
        "2 upphöjd till 3.5 =  " + Power(2, 3.5) + '\n') ;

      String s = "abcdefghijklmnopqrstuvwxyzåäö";
      Console.WriteLine(
        "\tÖverlagring av biblioteksmetod:\n\n"                 +
        "\tHela stängen:   " + s + "\n\tSubstring(10)   = "     +
        "delsträng från index 10 till strängens slut\n\t\t\t= " +
        s.Substring(10) + "\n\tSubstring(0, 6) = "              +
        "delsträng från index 0 av längden 6\n\t\t\t= "         +
        s.Substring(0, 6) + '\n') ;
   }

   static int Power(int bas, int exponent)       // Potens med en
   {                                             // int-exponent
      int resultat = 1;
      for (int i=1; i <= exponent; i++)          // Loopen bygger
          resultat *= bas;                       // potensen med
      return resultat;                           // upprepad mul-
   }                                             // tiplikation

   static double Power(double bas, double exponent)   // Potens
   {                                                  // med en
      return Math.Exp(exponent*Math.Log(bas));        // double-
   }                                                  // exponent
}
```

För att testa överlagring anropar vi båda metoderna **`Power()`** från **`Main()`**: C#-kompilatorn skiljer åt dem via parameterlistan och skickar automatiskt anrop med **`int`**-parametrar till den första och sådana med **`double`**-parametrar till den andra potensmetoden. Det första anropet **`Power(2, 3)`** går automatiskt till den första potensmetoden med en **`int`** som exponent eftersom den andra parametern, heltalskonstanten **`3`**, är en **`int`**. De två sista anropen går automatiskt till den andra potensmetoden med en **`double`** som ex-

ponent eftersom decimaltalskonstanterna **3.0** och **3.5** tolkas som **double**. En körning av programmet **Overload** ger:

```
Överlagring av egendefinierad metod:

2 upphöjd till 3   =  8
2 upphöjd till 3.0 =  8
2 upphöjd till 3.5 = 11,3137084989848

    Överlagring av biblioteksmetod:

    Hela stängen:   abcdefghijklmnopqrstuvwxyzåääö
    Substring(10)   = delsträng från index 10 till strängens slut
                    = klmnopqrstuvwxyzåääö
    Substring(0, 6) = delsträng från index 0 av längden 6
                    = abcdef
```

I programmet **Overload** har de två metoderna **Power()** samma namn, men olika datatyper till parametrarna. Den ena metoden har **int** som datatyp till parametrarna bas och exponent. Denna metod beräknar ”bas upphöjd till exponent” när exponent är heltal, t.ex. 2 upphöjd till 3, dvs $2 \cdot 2 \cdot 2$, genom enkel upprepad multiplikation i en **for**-sats som gör samma sak som: **resultat = bas * bas * bas** om vi tillämpar exemplet 2 upphöjd till 3. Den andra beräknar **p**otensen när exponent är decimaltal, t.ex. 2 upphöjd till 3.5 genom att använda en avancerad matematisk formel då det är meningslöst att multiplicera 2 med sig själv 3.5 gånger. Man tillämpar två olika metoder för beräkning av potensen beroende på om exponenten är heltal eller decimaltal. Vilken datatyp basen har, är däremot irrelevant för val av metod. Självklart täcker den matematiska formeln även beräkningen av ”bas upphöjd till heltal”. Men varför göra det komplicerat när det går enklare? Den matematiska formeln använder biblioteksklassen **Math** som definierar metoderna **Exp()** och **Log()**. Dessutom kan man minska risken för avrundningsfel när man använder en enklare beräkningsmetod för den enklare uppgiften. Därför är det motiverat att ställa båda metoder till förfogande. Överlagring ger oss dessutom möjligheten att döpa dem till samma namn. Det är två metoder som båda gör ”samma sak” nämligen potensiering, men ändå inte är exakt identiska.

Metoden `Substring()`

Klassen **String** har bl.a. två metoder **Substring()** som överlagrar varandra. Båda tar ut delstängar ur en sträng. Den ena har *en* parameter **n** och tar ut delsträngen från och med index **n** till strängens slut. Därför ger anropet **Substring(10)** delsträngen **klmnopqrstuvwxyzåääö** ur det svenska alfabetet eftersom **k** har index **10**, Den andra har *två* parametrar **a**, **b** och tar ut delsträngen från och med index **a** av längden **b**. Därför ger anropet **Substring(0, 6)** delsträngen **abcdef** från **a** dvs index **0** och av längden **6** – ett exempel på överlagring av biblioteksmetoder. Begreppet *index* förekommer i *array*-sammanhang som behandlas i nästa kapitel. Det som vi behöver veta om det just nu är att det är en numrering som börjar att räkna från **0** och inte från **1**.

4.7 Rekursiva metoder

Rekursiva metoder är sådana som anropar sig själva, ungefär som hundar som bitar sig i svansen. Ordet rekursiv kommer från *recurrere* på latin som på engelska betyder *to run back* eller *to run again* dvs att gå tillbaka och köra igen.

Rekursion är ett koncept som används för att lösa problem genom successiv upprepning av vissa beräkningar (algoritmer). Upprepade beräkningar är datorn bra på. Rekursiva algoritmer genererar kort och elegant kod som är mycket nära matematisk notation. I regel finns det även icke-rekursiva, s.k. *iterativa* lösningar till samma problem.

Ett exempel på problem som kan lösas rekursivt är följande uppgift som den italienske matematikern *Leonardo Pisano Fibonacci* år 1202 formulerade i sin bok *Liber abaci* (Boken om räknekonsten). Den handlar om kaniners fortplantning:

> Ett kaninpar föder från den andra månaden av sin tillvaro ett nytt par varje månad. Samma gäller för de nya paren.
>
> Hur många par kommer det att finnas om ett år?

Fibonacci hade väl knappast kunnat drömma om att hans problem skulle bli föremål för datoriserade lösningar med rekursiva metoder 810 år senare.

Om vi följer uppgiftens lydelse och räknar fram de första månaderna får vi följande:

Antal månader	1	2	3	4	5	6	7	8	…
Antal kaninpar	1	1	2	3	5	8	13	21	…

Det uppstår en talföljd i den andra raden av tabellen som kallas *Fibonaccis talföljd* eller kort *fibonaccitalen*. Så här uppstår de:

De två första månaderna finns det 1 kaninpar. De föder sitt första barnpar först efter 2 månader dvs i månad nr 3, varför det finns 2 kaninpar i månad 3. I månad 4 föder det första paret sitt andra barnpar, varför det finns 3 par i månad 4. I månad 5 föder det första paret sitt tredje barnpar, men även deras första barnpar föder ett nytt par, eftersom det har gått 2 månader sedan deras födelse. Därför finns det 5 par i månad 5. Osv. …

Praktiskt taget blir det allt svårare att hålla reda på antalet kaninpar när antalet månader växer. Man måste kanske rita någon sorts diagram och anteckna allt från månad till månad. En utväg ur dilemmat vore att upptäcka ett mönster, en struktur, t.ex. ett samband mellan antal månader och kaninpar, en slags laglighet i bildandet av fibonaccitalen som kan beskrivas i form av en algoritm för att sedan kunna skrivas som program. Undersöker man tabellen noga kan man se följande enkelt mönster: Summan av två på varandra

följande fibonaccital ger nästa fibonaccital. Kolla själv! Men hur kav vi beskriva detta mönster? Vi inför beteckningarna:

n = Antalet månader
F_n = Antalet kaninpar i månaden n

Mönstret som vi upptäckte ovan kan vi nu beskriva så här:

$F_1 = 1, \quad F_2 = 1$
$F_n = F_{n-1} + F_{n-2}$ för n = 3, 4, 5, …

Den första raden säger att de första två fibonaccitalen är 1 och 1. Den andra raden säger att det n-te fibonaccitalet är summan av de två föregående, vilket är bara en annan formulering av samma mönster vi upptäckte i tabellen. Formeln ovan kallas *Fibonaccis rekursionsformel*. Men vad är det rekursiva i denna formel? I en vanlig, icke-rekursiv formel står den sökta storheten vänster om likhetstecknet och alla givna storheter höger om likhetstecknet. Men här står den sökta storheten, fibonaccitalen, på båda sidor likhetstecknet, fast för olika månader, för olika parametrar så att säga. För att beräkna ett fibonaccital måste man känna till de två föregående. Men eftersom vi har de två första F_1 och F_2, s.k. *startvärden*, kan vi beräkna alla andra successivt utgående från dessa startvärden. Att det sökta står på båda sidor likhetstecknet är alltså det rekursiva, vilket, när vi kodar formeln, resulterar i en metod som anropar sig själv, fast med olika parametrar. Så här ser den rekursiva metoden ut när vi implementerar Fibonaccis rekursionsformel:

```
// Fibonacci.cs
// Rekursiv metod Fib() som för varje n returnerar fibonaccitalet
// Rekursiv därför att metoden anropar sig själv

class Fibonacci
{
   public static long Fib(int n)
   {
      if (n <= 1)
         return n;
      else
         return Fib(n-1) + Fib(n-2);       // 2 rekursiva anrop
   }                                       // i metodens kropp
}
```

Som man ser är koden en ren översättning av Fibonaccis rekursionsformel till C#-kod. Därför är den också väldigt kort. För n=0 eller 1 returneras n själv, dvs 0 eller 1 där 1 är enligt formeln det första fibonaccitalet. För alla andra n returneras summan av de två föregående dvs **`Fib(n-1)+Fib(n-2)`**. Men de i sin tur är var och en, anrop av **`Fib()`**. Men dessa anrop står i själva metoden **`Fib()`**:s kropp, vilket är just det rekursiva. Ett anrop av **`Fib(4)`** t.ex. resulterar i att **`Fib(3)`** och **`Fib(2)`** anropas, **`Fib(3)`** i sin tur resulterar i att **`Fib(2)`** och **`Fib(1)`** anropas, osv. Varje anrop av metdoden resulterar i ett stort antal följdanrop. Växer n leder det till en väldigt stor mängd av beräkningar. För stora fibonaccital är tidsåtgången stor. Låt oss testa metoden **`Fib()`** i följande program:

```
// FibonacciTest.cs
// Testar metoden Fib() genom att anropa den för de första
// 30 fibonaccitalen och skriva ut dem
using System;

class FibonacciTest
{
   static void Main()
   {
      Console.Write("\n\n\tDe första 30 fibonaccitalen:\n\n\t");
      for (int i = 1; i <= 30; i++)
      {
          Console.Write(Fibonacci.Fib(i) + "\t");     // Anropen
          if (i % 6 == 0)
             Console.Write("\n\n\t");
      }
      Console.WriteLine();
   }
}
```

Det är i **`for`**-satsen metoden **`Fib()`** anropas. Räknaren **`i`** blir metodens parameter, vilket genererar de första 30 fibonaccitalen. I var 6:e utskrift läggs in ett radbyte:

```
De första 30 fibonaccitalen:
1       1       2       3       5       8
13      21      34      55      89      144
233     377     610     987     1597    2584
4181    6765    10946   17711   28657   46368
75025   121393  196418  317811  514229  832040
```

Så kan vi besvara den inledande frågan: Det kommer att finnas 144 kaninpar om ett år.

Nackdelen av rekursiva metoder

Rekursiva metoder har en stor beräkningskomplexitet. Man pratar om exponentiellt växande tidskomplexitet av typ 2^n för att beräkna **`Fib(n)`**. Dvs tidsåtgången växer med en faktor 2^n. T.ex. om det tar $2^4 = 16$ nanosekunder för att beräkna **`Fib(4)`**, tar det 2^{40} dvs över 10^{12} nanosekunder (ca. 2½ timmar) för att beräkna **`Fib(40)`**, vilket uppenbart är ineffektivt. I så fall är det effektivare att använda en alternativ icke-rekursiv, t.ex. en iterativ implementering av Fibonaccis rekursionsformel. Därmed är det inte sagt att rekursiva metoder alltid är ineffektiva. Det finns problem som enklast löses med rekrursiv teknik, t.ex. att manipulera datastrukturer som träd och grafer. Det finns t.o.m. problem där rekursiva metoder leder till effektivare lösningar än alternativa icke-rekursiva algoritmer, t.ex. sortering. Ett annat problem är hur svårt det är att beskriva och implementera dessa algoritmer. Man borde alltså avväga från fall till fall om rekursiv eller iterativ metod ska användas.

4.8 Lambdauttryck

Lambdauttryck (eng. *lambda expressions*) är korta funktioner utan namn. De anropas i samma kod som de definieras. Ex.: `(a, b) => a+b` `=>` kallas för *Lambdaoperatorn* och skiljer parameterlistan `(a, b)` från kroppen `a+b`. Detta lambdauttryck är en funktion som adderar a med b. Vid exekveringen ersätts lambdauttrycket av summans värde: a+b.

Följande exempel demonstrerar lambdauttryck:

```
// Lambda.cs
// Lambdauttryck skrivs med Lambdaoperatorn => som separerar
// funktionens parametrar (vänster) från dess kropp (höger)
// => betyder "ska skickas till" (OBS! ingen jämförelseoperator)
using System;
using System.Linq;                          // Krävs för Where(...)
class Lambda
{
   static void Main()
   {
      int[] numbers = { 11, 37, 52, 26, 57, 90, 101 };

      int[] oddNum = numbers.Where(n => n % 2 == 1).ToArray();
      int[] divBy3 = numbers.Where(n => (n % 3) == 0).ToArray();
      int[] square = numbers.Select(n => n * n).ToArray();
      int[] sorted = numbers.OrderBy(n => n).ToArray();

      Console.Write("\n\tAlla heltal:");
      foreach (int element in numbers)
         Console.Write("\t" + element);
      Console.Write("\n\tSorterade:");
      foreach (int element in sorted)
         Console.Write("\t" + element);
      Console.Write("\n\tKvadraterna:");
      foreach (int element in square)
          Console.Write("\t" + element);
      Console.Write("\n\tDe udda talen:");
      foreach (int element in oddNum)
         Console.Write("\t" + element);
      Console.Write("\n\tDelbara med 3:");
      foreach (int element in divBy3)
         Console.Write("\t" + element);
      Console.WriteLine("\n");
   }
}
```

n => n % 2 == 1 är själva lambdauttrycket dvs anonyma funktionen vars definition och anrop sammanfaller i denna kod. **n** är funktionens parameter. Den behöver inte deklareras. **n** skickas med => till kroppen, dvs till **n % 2 == 1**. Detta logiska uttryck evalueras lokalt och returnerar sant eller falskt, beroende på **n % 2 == 1** eller ej. **n % 2** ger resten vid heltalsdivision av **n** med **2** (se modulooperatorn, sid 124). Därmed blir **n % 2 == 1** sant om och endast om **n** är udda. Dvs endast de udda talen selekteras från arrayen **numbers**. Det görs genom att skicka den anonyma funktionens returvärde till **Linq**-metoden **Where()** och skapa den nya arrayen **oddNumbers**. **Linq** är ett bibliotek i C# som bl.a. tillhandahåller metoden **Where()**. Den selekterar enligt returvärdets sanningsvärde element från arrayen **numbers**. Eftersom **Where()** är definierad som en generisk metod måste dess returvärde med metoden **ToArray()** omvandlas till array av **int** för att kunna tilldelas **int**-arrayen **oddNumbers**. De filtrerade talen från arrayen **numbers** skrivs ut när man kör programmet **Lambda**:

```
Alla heltal:    11    37    52    26    57    90      101
Sorterade:      11    26    37    52    57    90      101
Kvadraterna:    121   1369  2704  676   3249  8100    10201
De udda talen:  11      37      57      101
Delbara med 3:  57      90
```

Innan vi går vidare följer en parentes om LINQ som vi använde i programmet **Lambda**.

Vad är LINQ ?

I programmet **Lambda** finns koden: **numbers.Where(n => n % 2 == 1)**
Här ”frågas” arrayen **numbers** om den har element som är udda tal. Metoden **Where()** är definierad för arrays i biblioteket **System.Linq** som är ett tillägg till C#.

I koden ovan har man kombinerat lambdauttryck med språkelement från *LINQ* för att åstadkomma effektiv kod. *LINQ* står för *Language Integrating Query* och är en språkmodul vars syntax liknar frågespråket SQL *(Structured Query Language)*. SQL har funnits sedan länge som standardspråk för kommunikation med databaser. Microsoft har utvecklat och implementerat *LINQ* i versionen 3.5 av sin .NET-plattform som släpptes år 2007. Man har integrerat *LINQ* bl.a. i C# och udvidgat därmed språket. Implementationen finns i biblioteket **System.Linq**. Men ambitionen har varit att gå vidare och presentera *LINQ* som ett nytt sätt att tänka och skriva kod inte bara inom .NET utan inom programmering i största allmänhet – som ett slags nytt paradigm där man försöker dra nytta av databastänkandet i objektorienterad programmering. Men språket används inte bara i samband med databaser. *LINQ* har många olika användningsområden, bl.a:

LINQ to SQL som används för att fråga databaser, *LINQ to XML* för att fråga XML-dokument, *LINQ to Array* för att fråga arrays och *LINQ to Object* för att fråga objekt. *LINQ to Array* har vi använt i programmet **Lambda**.

4.9 Delegater

Ex.: `d = (a, b) => a+b`

Delegaten `d` är en referens till den anonyma funktionen (lambdauttrycket). `d` kan användas för att anropa funktionen eller för att skicka den som parameter till andra metoder – som *representant* för den anonyma funktionen.

OBS! Det här är något helt nytt i programmeringen:

Hittills kunde vi skicka variabler, arrays, ja t.o.m. objekt (med hjälp av referenser) som parametrar till andra metoder. Men vi kunde aldrig skicka metoder som parametrar till andra metoder. Med hjälp av delegater kan vi skriva om våra metoder som anonyma funktioner (lambdauttryck), namnge dem med delegater och skicka dem som parametrar till andra metoder, där de sedan kan anropas. Det kan vi göra genom att tilldela lambdauttrycken till referenser som då kallas för *delegater* – en slags representant för lambdauttrycken.

I programmet **`Delegate`** nedan visas ett vanligt anrop. I **`DelegateParam`** längre fram demonstreras anrop av en delegat som skickats som parameter till en annan metod.

```
// Delegate.cs
// Delagat som referens till en anonym funktion
using System;

class Delegate
{
  delegate void Dtype(string t);            // Deklarerar den nya
                                            // delegattypen Dtype
  static void Main()
  {
    Dtype d;                                // Deklarerar delegat

    d = text => Console.WriteLine(text);    // Delegat pekar på
                                            // anonym funktion
    d("Denna sträng kommer från delegate"); // Anropar funktionen
  }
}
```

En delegat skapas i två steg: Först deklareras en ny datatyp av typen **`delegate`** med ett namn som vi väljer. I exemplet ovan har vi valt namnet **`Dtype`**:

```
delegate void Dtype(string t);
```

Som man ser inleds metodens huvud med det reserverade ordet **`delegate`**. Denna sats skrivs på samma plats som klassens datamedlemmar och på samma sätt som man dekla-

rerar en metod utan kropp. Sedan används den nya datatypen för att i **`Main()`** deklarera en delegat av denna nya datatyp som är en delegattyp:

```
Dtype d;
```

Den nyss deklarerade delegaten tilldelas en anonym funktion som formuleras med ett lambdauttryck:

```
d = text => Console.WriteLine(text);
```

I den här anonyma funktionen (gråmarkerad) ska parametern **`text`** skickas till att skrivas ut. Men **`d`** är en referens av typ **`Dtype`**. En sådan referens kan endast tilldelas ett objekt av typ **`Dtype`**. Därför måste den ovan gråmarkerade anonyma funktionen samtidigt vara ett objekt av typ **`Dtype`**. Vi kan i fortsättningen referera till detta objekt med **`d`**, vilket vi gör i nästa sats:

```
d("Denna sträng kommer från delegate");
```

Här anropas den anonyma funktionen med referensen **`d`**. Därvid skickas strängen i parentesen som aktuell parameter till den formella parametern **`text`**. Där skickas den vidare till utskrift. Därför ser resultatet av en körning av programmet **`Delegate`** ut så här:

```
Denna sträng kommer från delegate
```

Vi ser i programmet **`Delegate`** på vilket sätt en funktion samtidigt kan vara ett objekt. Detta tack vare delegatkonceptet dvs en referens som kan peka på en funktion.

Varför vi förresten säger funktion och inte metod beror på funktionens anonymitet just här i det behandlade programexemplet. Eftersom funktionen inte har något namn kan den inte heller vara medlem i klassen **`Delegate`** och därmed inte en metod. Men generellt kan delegater peka även på metoder. Vi kommer i slutet av det här avsnittet att ta upp ett exempel på delegater som pekar på metoder, ja t.o.m. på s.k. *metodgrupper*. Då kommer det också att avslöjas varför dessa referenser till metoder heter delegater. Men innan dess ska vi gå vidare lite med delegater:

Delegat som parameter i metoder

Vi har lärt oss att skicka vanliga variabler, referenser, arrays, listor och även objekt som parametrar till metoder. Men hittills har det inte varit möjligt att skicka metoder, ja inte ens funktioner, som parametrar till andra metoder. Medan det t.ex. i matematik är ganska vanligt att bilda funktioner av funktioner, s.k. *sammansatta funktioner*, har vi i programmering inte haft denna möjlighet. Men det ska nu bli annorlunda, för delegater öppnar dörren till denna nya värld. Kan man skicka referenser som parametrar till metoder, då borde man även kunna göra det med sådana referenser som pekar på metoder, dvs med delegater. Följande programexempel ska ge oss en insikt i **`delegate`**:s möjligheter att även i programmering använda sammansatta metoder:

```
// DelegateParam.cs
// Räknar ut hur många element i en given array som är nollor,
// hur många som är negativa och hur många som är positiva
// Delegat skickas till metoden MyCount():s parameter
// Där anropas den metod som delegaten pekar på
using System;

class DelegateParam
{
  delegate bool Dtype(int number);// Deklaration av delegattyp
                                  // OCH av metod med returvärde
  static void Main()
  {
    Dtype d0 = a => a == 0;       // Delegater som pekar på anony-
    Dtype d1 = a => a < 0;        // ma funktioner (lambdauttryck)
    Dtype d2 = a => a > 0;


    int[] vector = { -1, 2, -3, 0, 5, 0, -4, 1, 6, 8, -9, 0 };


    Console.WriteLine("\n\tDet finns {0} nollor i vektorn.",
                                      MyCount(vector, d0));
    Console.WriteLine("\n\tDet finns {0} negativa tal i vektorn.",
                                      MyCount(vector, d1));
  Console.WriteLine("\n\tDet finns {0} positiva tal i vektorn.\n",
                                      MyCount(vector, d2));
  }
// ---------------------------------------------------------------

  static int MyCount(int[] v, Dtype d)// Metod med en  delegat
                                      // som parameter
  {                                   // Räknar antal element i v
                                      // som uppfyller det villkor
    int counter = 0;                 // som skickas med delegat-
    foreach (int element in v)       // parametern d
       if (d(element))               // Anrop av delegaten d i
          counter++;                // if-satsens villkor
    return counter;
  }
}
```

Så här ser resultatet av en körning av **DelegateParam** ut:

```
       Det finns 3 nollor i vektorn.

       Det finns 4 negativa tal i vektorn.

       Det finns 5 positiva tal i vektorn.
```

Det som gör att dessa tre rader skrivs ut är tre anrop av den egendefinierade metoden **`MyCount()`** inbyggda i **`System`**-metoden **`Console.WriteLine()`**. Metoden **`MyCount()`** räknar antalet av de **`vector`**-element som uppfyller det villkor som definieras av den delegat som skickas till den andra parametern **`d`** till **`MyCount()`**. I första anropet skickas delegaten **`d0`** som returnerar sant om **`vector`**-elementet är lika med **`0`**: Antalet nollor returneras. I andra anropet skickas delegaten **`d1`** som returnerar sant om **`vector`**-elementet är **`< 0`**: Antalet negativa element returneras. I tredje anropet skickas delegaten **`d2`** som returnerar sant om **`vector`**-elementet är **`> 0`**: Antalet positiva element returneras. Vi menar förstås *...delegaten* ***`dx`*** *vars metod som den pekar på, returnerar sant...*. För att undvika alltför komplexa formuleringar nämner vi ofta bara referensen.

Anropet av delegaten **`d`** som kommer in i metoden **`MyCount()`** via den andra parametern, sker i **`foreach`**-satsens **`if`**-sats. Där bestäms delegatens sanningsvärde, vilket avgör om räknaren ska **`counter`** uppdateras. **`MyCount()`** returnerar detta värde till utskriftssatsen i **`Main()`**.

Överlagrade varianter av `Console.WriteLine()`

Av förekommen anledning ska vi lägga in här en parentes: I förra programexemplet **`DelegateParam`** används i utskriftssatsen en syntax som skiljer sig från våra utskriftssatser hittills. Så här lyder t.ex. den första **`Console.WriteLine()`**-satsen:

```
Console.WriteLine("\n\tDet finns {0} nollor i vektorn.",
                                    MyCount(vector, d0));
```

För det första har metoden **`Console.WriteLine()`** här två parametrar och inte en. Dvs vi har att göra med en överlagrad variant av denna metod. Anledningen är att vi vill undvika manuella konkateneringar av strängar med konverterade variabelvärden i **`Console.WriteLine()`**-satsen, vilket kan komplicera koden. En förenkling är då att använda den tvåparametriga, överlagrade varianten av **`Console.WriteLine()`**-metoden där den andra parametern **`MyCount()`** som returnerar ett värde, automatiskt konverteras till sträng och infogas i den första strängparametern på en plats som anges med syntaxen **`{0}`**. Med detta menar man det första elementet (med index **`0`**) i den serie av parametrar som följer efter den första strängparametern. Man skulle alltså – med en flerparametrig variant av metoden – kunna skriva ytterligare parametrar med värden som infogas i den första strängparametern med t.ex. **`{1}`**, **`{2}`**, **`{3}`** osv. Men hos oss är **`{0}`** ingenting annat än **`MyCount(vector, d0)`**, konverterad till sträng.

Följande program visar med en fyrparametrig variant av **`Console.WriteLine()`** hur smidigt det kan vara att låta de överlagrade varianterna automatisk konvertera variablerna i den 2:a, 3:e och 4:e parametern till strängar och konkatenera dem med dvs infoga dem i den 1:a parametern (utskriftssträngen):

```
// WriteLineOverl.cs
// Console.WriteLine()-metoden med 4 parametrar
// Att infoga variabelvärden i utskriftssträngen
using System;

class WriteLineOverl
{
   static void Main()
   {
      int no1 = 9, no2 = 3, sum;
      sum = no1 + no2;
      Console.WriteLine("\n\t Addition:\t {0} + {1} ger {2} \n",
                                                  no1, no2, sum);
  }
}
```

Programmet **WriteLineOverl** ger följande utskrift:

```
        Addition:        9 + 3 ger 12
```

Lösningen med LINQ

Efter parentesen om de olika varianterna av **Console.WriteLine()** ska vi återvända till delegater, närmare bestämt till delegater som parametrar i metoder. Vi ska titta om vi kan skriva programmet **DelegateParam** (sid 182) lite effektivare. Där använde vi en delegat som parameter i den egendefinierade metoden **MyCount()** som räknade antalet element i en given array som uppfyllde en viss egenskap. Själva egenskapen formulerades i **Main()** med ett lambdauttryck och skickades till **MyCount()** med en delegat. Men vi hade redan i programmet **Lambda** (sid 178) sett att man kunde fråga en array om dess element uppfyllde en viss egenskap, nämligen att vara udda tal. Det gjorde vi med hjälp av **Linq**-metoden **Where()**. Frågan är om det även finns en **Linq**-metod som frågar en array efter antal element som uppfyller en viss egenskap som kan formuleras med ett lambdauttryck. Den här gången för att bestämma hur många element i arrayen som är nollor, hur många som är negativa och hur många som är positiva. I så fall skulle vi kunna byta ut vår egendefinierade metod **MyCount()** mot denna **Linq**-metod och slippa koda själva. Faktiskt finns det en sådan metod som heter **Count()**. Det är anmärkningsvärt att **Linq**-metoden **Count()** kommer att göra användningen av delegat onödigt eftersom den kommer att kunna anropas direkt i **Main()**. Vi slipper att skicka en parameter till en egendefinierad metod. Även här kommer vi att dra nytta av kombinationen av LINQ och lambdauttryck som ger en effektiv och elegant kod.

Följande **Linq**-version av programmet **DelegateParam** ger exakt samma resultat som **DelegateParam**, men utan delegat:

```
// CountLINQ.cs
// Anropar Linq-metoden Count() med ett lambdauttryck
using System;
using System.Linq;              // Krävs för Linq-metoden Count()

class CountLINQ
{
 static void Main()
 {
  int[] vector = { -1, 2, -3, 0, 5, 0, -4, 1, 6, 8, -9, 0 };

  Console.WriteLine("\n\tDet finns {0} nollor i vektorn.",
                                        vector.Count(a => a == 0));

  Console.WriteLine("\n\tDet finns {0} negativa tal i vektorn.",
                                         vector.Count(a => a < 0));

  Console.WriteLine("\n\tDet finns {0} positiva tal i vektorn.\n",
                                         vector.Count(a => a > 0));
 }
}
```

Utskriften blir den samma som utskriften av programmet **DelegateParam** (sid 182): Antal element i arrayen **vector** som är 0, negativa och positiva skrivs ut.

Metodgrupper

En metodgrupp är mängden av samtliga överlagringar av en metod. T.ex. har metoden **Console.WriteLine()** 18 olika överlagringar (varianter) som bildar metodgruppen

Console.WriteLine

För första gången skriver vi **Console.WriteLine** utan parenteserna **( )** vilket innebär att det inte handlar om en metod utan om metod*grupp*. Metodgruppen kan direkt tilldelas en delegat. Först vid anropet av delegaten avgörs vilken av gruppens metoder (varianter) ska exekveras, för då skrivs en parameterlista till delegaten som specificerar den metod som ska anropas. I och med detta får också begreppet *delegat* sin betydelse, nämligen som en företrädare eller representant för gruppen. Följande program demonstrerar detta:

```
// MethodGroup.cs
// Delegat pekar på metodgruppen Console.WriteLine utan parentes
// dvs på Console.WriteLine()-metodens alla överlagrade varianter
// Delegat representerar metodgruppen
// Anrop via delegat avgör vilken av gruppens metoder anropas
using System;

class MethodGroup
{                                  // Delegattypen deklareras:
   delegate void Dtype(string t, string a, string b, string c);
```

```
   static void Main()
   {
      int no1 = 9, no2 = 3, sum;
      sum = no1 + no2;

      Dtype d;                    // En delegat deklareras

      d = Console.WriteLine;      // Delegat tilldelas metodgruppen
                                  // utan parentes: Alla varianter
      d("\n\t Addition:\t {0} + {1} ger {2} \n",
      no1.ToString(), no2.ToString(), sum.ToString());
   }                              // 4 parametrar skickas via
}                                 // delegat till metodgruppen
```

Programmet **MethodGroup** ger samma utskrift som programmet **WriteLineOverl**:

```
        Addition:        9 + 3 ger 12
```

Denna utskrift kommer från metoden **Console.WriteLine()**, närmare bestämt från den variant av den som har *fyra* parametrar. Detta trots att vi endast har tilldelat metodgruppen **Console.WriteLine** till delegaten **d** och inte specificerat vilken av gruppens metoder som ska anropas. Att ändå just den metod av gruppen automatiskt väljs som har fyra parameter, beror på att anropet av delegaten i programmets sista sats sker med just fyra strängar i parameterlistan. På så sätt används delegaten som en länk mellan metodgruppen **Console.WriteLine** och programmet. Samtidigt ser man att delegaten **d** blir en representant för hela metodgruppen.

Övningar till kapitel 4

4.1 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_4_1
{
   static void Main()
   {
      {
         int t = 30;
      }
      Console.WriteLine("t = " + t);
   }
}
```

4.2 Modularisera programmet **`MiniSort`** från (sid 154) efter eget godtycke.

4.3 Skriv en rekursiv metod **`Faculty()`** som implementerar n! = 1· 2 · 3· … · n.
Testa metoden i en klass **`FacTest`** genom att anropa den för n = 1, 2, 3, … , 20.

4.4 ***Tillägg till Pyramiden (projekt)*** Modularisera programmet ***Pyramiden*** från övn. 1.10 (sid 66) genom att flytta koden som bestämmer det tillåtna antalet rader 1-13 till en metod som deklareras i en separat klass och anropas från **`Main()`** innan pyramiden ”ritas”.

4.5 ***Kalkylatorn (projekt)*** I detta projekt ska skapa en klass **`Calculator`** som stödjer följande funktionaliteter: addition, subtraktion, multiplikation, division och potensiering av två tal samt att 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 en loop. De olika räkneoperationerna ska definieras i separata metoder och anropas i **`Main()`**.

Klassen Calculator:

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, se programmet **GuessDo** i kap 6. 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 5

Tillämpning av OOP

5.1 Arrays

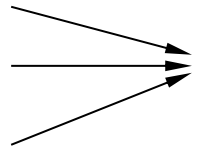
Ordet *array* betyder i engelskan *ordnad samling* eller *ordnad uppställning* (*battle array* = stridsordning). Andra beteckningar som används i litteraturen är *fält*, *vektor*, *lista, …* . Vi kommer att använda *array*.

> En **array** är en ordnad mängd av variabler av *samma* datatyp grupperade under *samma* namn och lagrade i ett *sammanhängande* minnesområde.
>
> En array består av ett antal **element**. Elementens position i arrayen kallas för **index**. Indexnumreringen börjar med 0, inte med 1.

Anta att vi vill definiera 20 variabler av typ **`int`**. Hittills behövde vi skriva 20 satser för att göra det. Men nu ger array oss möjligheten att göra samma sak med endast *en* sats:

Hittills: enkel datatyp **`int`**:

```
int no1;
int no2;
      .
      .
      .
int no20;
```



Nu: **`int`**-*array* med *referens*:

```
int[] no = new int[20];
```

Vi definierar *en* variabel **`no`** av datatypen **`int[]`**, dvs array av **`int`**, använder **`new`** och lägger till informationen om antalet element inom hakparentes: **`[20]`**. 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 variabeln **`no`** som är en *referens*. Dess datatyp **`int[]`** är en *referens till en* **`int`**-*array*. För att göra det tydligare kan man skriva det även i två separata satser:

```
int[] no;
no = new int[20];
```

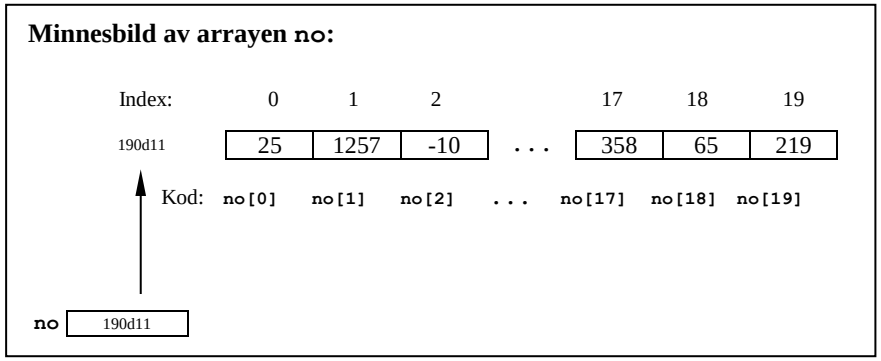
Det är inte den första utan den andra satsen, närmare bestämt koden **`new int[20]`** som *skapar* själva arrayen. Därför står också storleken **`20`** där det behövs, nämligen i satsen där **`new`** allokerar minne. Typiskt för array är *hakparenteserna* **`[ ]`**, på engelska *brackets*. I satserna 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:



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. 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**.

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 har vi skrivit C#-kod som kommer åt 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 som ligger **1** x 4 bytes – storleken för en **int** – längre bort från **no**. **no[2]** lagras vid adressen som ligger **2** x 4 bytes längre bort från **no** osv. Adressering i RAM sker nämligen byte-vis, så att bytes som är grannar till varandra, har adresser som skiljer sig på *en* enhet. Avgörande för denna indexeringsteknik är att en array alltid allokeras i ett *sammanhängande* minnesområde. Ser man på det hela ur hårdvarans synpunkt kan man förstå varför indexnumreringen börjar med **0** och inte med **1**: **no[0]** kan tolkas som den *adress* som ligger **0** x 4 bytes längre bort från **no**, dvs **no[0]**:s adress är identisk med adressen **no**.

Därför gäller:

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

Med *position* menas numret som *människan* använder för att 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 programmeringsspråken 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 defaultvärden även om de kan förekomma som lokala variabler i metoder. Detta visas i följade program:

Definition 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 – ett tecken på att arrayens allokering sker vid *run time*.

```
// Array.cs
// Definierar en array av 4 int-värden, 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 the bounds of the array.
at Array.Main() in C:\Programmering\Programmering 2\2OOP\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.

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 (sid 195):

`foreach`-satsen

Denna sats som används i programmet **`Array`** (sid 192) är en ny kontrollstruktur som inte kunde tas upp i kapitlet om kontrollstrukturer *(Progr1)* därför att den förutsätter arraybegreppet 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 192) 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. Ett exempel på det är programmet **`ArrayOfRef`** (sid 200). Där deklareras iterationsvariabeln i en **`foreach`**-sats till den egendefinierade klassen **`Fish`** (sid 199), för att skriva ut ett **`Fish`**-objekts sort, vikt, längd, pris och frakt.

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

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 alla dessa element är att de lagras en efter den andra vid adressen eller referensen **`no`**:

`no`	**`0`**	**`0`**	**`0`**	**`0`**

   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 192) tillämpas enligt vilken indexeringen börjar med **`0`**. Därför är **`no[4]`** i arrayen ovan inte definierat:

`no`	**`no[0]`**	**`no[1]`**	**`no[2]`**	**`no[3]`**

   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 de defaultvärden vi nämnde för datamedlemmar i objekt (sid 101), 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 defaultvärden (precis som datamedlemmar i ett objekt) även om arrayen skapas lokalt i en metod.

5.2 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**, men bibehåller dess egenskaper:

```
// 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:
                                          // Definition OCH ini-
                                          // tiering av en array
//    int[] no = new int[4] { 64, 86, 34, -6 };  // Gör 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 192) – 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:

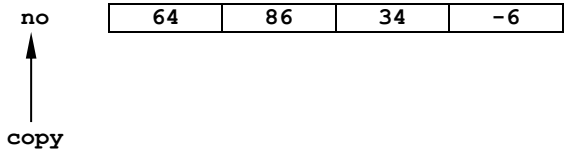
```
int[] copy = no;
```

`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`**:



5.3 Array av referenser

Hittills har vi bildat arrays endast av den fördefinierade datatypen **`int`**. På samma sätt kan man också definiera arrays av alla andra enkla datatyper. Men kan man bilda även arrays av klasser dvs egendefinierade datatyper? Frågan måste preciseras: Menar man arrays av *referenser*, är svaret ja, därför att klasser – referensernas datatyper – har exakt samma ”rättigheter” som vilka andra datatyper som helst och kan därför skrivas överallt i koden där en fördefinierad datatyp kan stå. Precis som referensvariabler kan skrivas överallt, där även en variabel av enkel typ kan stå. Menar man arrays av *objekt*, är svaret nej, vilket vi kommer att förklara i detta avsnitt. Vi kommer att inse att en array av objekt inte är nödvändig, när man har en array av referenser vars element pekar på ett objekt. Array av referenser gör oss samma tjänst som array av objekt.

Vi börjar med att deklarera en klass so§m vi sedan i programmet **`ArrayOfRef`** (nästa sida) kommer att använda för att konstruera en array av referenser som i sin tur ska användas för att peka på objekt av denna klass:

```
// Fish.cs
// Deklarerar klassen Fish med tre datamedlemmar och två metoder
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);
   }
}
```

Klassen **`Fish`** modellerar en fisk med datamedlemmarna **`sort`**, **`weight`** och **`size`**. En laxforell t.ex. med en viss vikt i gram och en viss längd i cm kan vara ett objekt av denna klass, där laxforell är fiskens sort. Metoden **`Price()`** beräknar priset på fisken oberoende av sort, med 7,25 kr per hekto. Metoden **`Shipping()`** beräknar transportkostnaden utifrån fiskens vikt och längd genom att t.ex. multiplicera kostnadsfaktorn 0,02 med vikten och 0,1 med längden och addera dem. Båda Metoder returnerar priset och frakten i hela kronor utan ören. Biblioteksmetoden **`Math.Round()`** avrundar till närmaste heltal. Självklart kan man anmärka att den här modelleringen har vissa brister ur praktisk synpunkt: För det första är fiskpriser i praktiken inte oberoende av sorten. För det andra är både pris och frakt i regel belopp i kronor och ören dvs decimaltal och inte heltal. Men vi gör medvetet båda förenklingar i modellen för att förenkla imple-

menteringen och koncentrera oss på det programmeringstekniska konceptet av *array av referenser*. Vi vill nämligen använda detta koncept, för att på ett effektivt sätt skapa och hantera många objekt av klassen **Fish**. För det här ändamålet är de nämnda bristerna i modelleringen irrelevanta. Följande program skapar en array av referenser till **Fish**-objekt och anropar metoderna **Price()** och **Shipping()** för att sedan registrera (skriva ut) alla uppgifter till varje objekt:

```
// ArrayOfRef.cs
// Skapar först en array av 5 referenser till Fish-objekt, skapar sedan 5
// Fish-objekt på vanligt sätt och tilldelar dem till referenserna.
using System;

class ArrayOfRef
{
   static void Main()
   {
      Fish[] f = new Fish[5];                        // Array av referenser
                                                     // OBS! Inga objekt
      for (int i = 0; i < f.Length; i++)
      {
          f[i] = new Fish();                         // Skapar objekt och
                                                     // tilldelar adressen
                                                     // till en referens
          Console.Write("\n\tMata in sorten till fisk" + (i+1) + ":\t");
          f[i].sort = Console.ReadLine();            // Input
          if (f[i].sort.Length <= 7) f[i].sort += '\t';

          Console.Write("\tMata in vikten till fisk" + (i+1) + ":\t");
          f[i].weight  = (float) Convert.ToDecimal(Console.ReadLine());

          Console.Write("\tMata in längden till fisk" + (i+1) + ":\t");
          f[i].size = (float) Convert.ToDecimal(Console.ReadLine());

      }

      Console.Write("\nFisksort\tVikt i g\tLängd i cm\tPris\tFrakt\n"   +
      "-------------------------------------------------------------\n");
      foreach (Fish element in f)
      {
          Console.WriteLine(element.sort + "\t "  +
             element.weight  + "\t\t   " + element.size       + "\t\t " +
             element.Price() + "\t "     + element.Shipping() + "\n")   ;
      }
   }
}
```

I programmet **ArrayOfRef** skapas en array av 5 *referenser* till **Fish**-objekt med satsen:

```
Fish[] f = new Fish[5];
```

Observera att denna sats inte skapar något objekt alls, för då skulle det behövas koden **new Fish()** – OBS! parentesen – som inte finns med i satsen ovan. Förväntar man sig att en ”array av 5 **Fish**-objekt” skulle skapas med **new Fish()[5]** så är det fel, för den här koden kan inte kompileras – ett tecken på att begreppet ”array av objekt” måste förkastas. Istället måste man gå *två* steg: Först måste en array av rena *referenser*

definieras som i satsen ovan. Initieringsproblematiken löses automatiskt pga att en array alltid initieras till sin datatyps defaultvärden och att datatypen *referens* default-initieras till **`null`** (sid 101). Då spelar det ingen roll om det handlar om referenser till objekt av klassen **`Fish`** eller av någon annan klass. Sedan kan man fundera hur man explicit initierar referenserna så att de pekar på verkliga objekt av typ **`Fish`**. Detta görs i programmet **`ArrayOfRef`** med:

```
f[i] = new Fish();
```

som står i **`for`**-satsen. Först efter den här satsen har vi allokerat minnesutrymme för *ETT objekt* av typ **`Fish`**, *inte* för en *array* av objekt, för i koden ovan finns inget spår av en sådan array. Detta objekts minnesadress tilldelas referensarray-elementet **`f[i]`** där **`i`** tack vare **`for`**-loopen går från **`0`** till **`4`**. Vi har endast att göra med en *array av referenser* till **`Fish`**-objekt, för hakparentesen – arrayens symbol – står efter referensvariabeln **`f`** som pekar på denna referensarray. Varje element i denna referensarray pekar i sin tur på ett separat **`Fish`**-objekt. De två stegen som tas är: Först från **`f`** till referensarrayen och sedan från den till objekten. Det första steget står utanför och det andra steget i **`for`**-loopen. Efter objektens definition initieras varje objekts datamedlemmar **`sort`**, **`weight`** och **`size`** i **`for`**-loopen till värden som läses in från konsolen. Sedan skrivs de fullständiga uppgifterna till varje objekt, dvs även priset samt fraktkostnaden, ut. Anropet av metoderna **`Price()`** och **`Shipping()`** är inbakade i utskriftssatsen. En körning av programmet **`ArrayOfRef`** kan ge följande slutlig dialog:

```
Mata in sorten till fisk1:      Laxforell
Mata in vikten till fisk1:      719
Mata in längden till fisk1:     38,5

Mata in sorten till fisk2:      Torsk
Mata in vikten till fisk2:      423
Mata in längden till fisk2:     28,7

Mata in sorten till fisk3:      Aborre
Mata in vikten till fisk3:      550
Mata in längden till fisk3:     25,5

Mata in sorten till fisk4:      Gädda
Mata in vikten till fisk4:      985
Mata in längden till fisk4:     58

Mata in sorten till fisk5:      Gös
Mata in vikten till fisk5:      395
Mata in längden till fisk5:     14
```

```
Fisksort        Vikt i g        Längd i cm      Pris    Frakt
-------------------------------------------------------------
Laxforell         719               38,5          52      18

Torsk             423               28,7          31      11

Aborre            550               25,5          40      14

Gädda             985               58            71      26

Gös               395               14            29      9
```

”Array av objekt” ?

För att kunna datorisera en verksamhet med fiskar behöver vi objekt av typ **`Fish`**. Självklart skulle man kunna skapa sådana objekt t.ex. med **`Fish f1 = new Fish();`** osv. Men vad gör man om man vill modellera en handel med stora fiskmängder under en längre period? Array skulle då vara den givna lösningen för att effektivisera kodningen. Men funderar man närmare på begreppet ”array av objekt” av typ **`Fish`** dyker upp följande fråga: Vilket defaultvärde ska t.ex. en array av **`Fish`**-objekt få vid initieringen? Till de enkla datatyperna i C# kommer de fördefinierade defaultvärdena **`0`**, tom sträng, **`null`**, nolltecknet och **`false`** (sid 101). Men **`Fish`** är ju ingen fördefinierad datatyp. Det finns ingen begränsning på egendefinierade datatyper (klasser) och det går inte att förutsäga vilka man kan skapa i C#. Och därför går det inte heller att fastslå vilken defaultinitiering en sådan array skulle få. Vi ser att begreppet ”array av objekt” leder till en återvändsgränd. Lösningen är *array av referenser* – referenser till objekt dvs en tvåstegslösning som användes i programmet **`ArrayOfRef`** (sid 200).

5.5 Array som parameter i metoder

Array som bearbetar större datamängder ger upphov till mer komplexa och sofistikerade program. Exempel på det är applikationer som söker, sorterar eller krypterar data. Vi kommer i fortsättningen att behandla enkla varianter av sådana program. Modularisering är metoden för att bryta ned stora komplexa program i mindre och enklare moduler. Helst vill man ha program som består av ett antal enkla, överskådliga metoder där varje metod löser ett specifikt problem. Sedan vill man sätta ihop dem dvs anropa dem med ett antal parametrar från **`Main()`** och kontrollera hela händelseförloppet från denna metod som helst ska ha så lite kod som möjligt. Ju mer avancerade datatyper man använder i sitt program desto större blir behovet av modularisering. Självklart vill man även modularisera program som använder array. I C# är det möjligt att skicka en array som parameter till en metod dvs att definiera en array i parameterlistan. I nästa program definieras en **`void`**-metod **`Method()`** med en array av **`int`** som parameter:

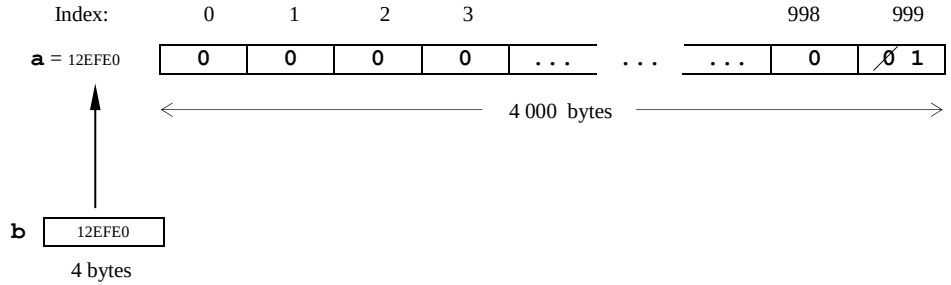
```
// ArrayParam.cs
// Skickar en stor array till en metod, men:
// Array som parameter i en metod behandlas som en referens
// Parameteröverföring sker med referensen: adressen skickas
using System;

class ArrayParam
{
  static void Method(int[] b)       // Array som parameter
  {
    Console.WriteLine("\n\tI metoden\n\tär arrayens sista " +
                      "element före ändringen " + b[999]);
    b[999] = 1;                    // Ändringen
    Console.WriteLine("\n\t\t\t     och efter ändringen "   +
                      b[999] + '\n');
   }

/*********************************************************/

  static void Main()
  {
    int[] a = new int[1000];     // Array med 1000 nollor
    Console.WriteLine("\n\tI Main()\n\tär arrayens sista " +
                      "element FÖRE anropet   " + a[999]);

    Method(a);                   // Referensanrop: arrayens
                                 // adress skickas till metod
    Console.WriteLine("\tI Main()\n\tär arrayens sista " +
               "element EFTER anropet  " + a[999] + '\n');
   }
}
```

Låt oss börja titta på **`Main()`** innan vi går in på hur arrayen **`b`** i metoden **`Method()`** behandlas. I **`Main()`** har vi en **`int`**-array **`a`** med 1000 element, alla initierade till default-

värdet **0**. En körning av **ArrayParam** avslöjar även en del intressanta nyheter för oss. Den viktigaste är att en ändring som görs i en annan metod återspeglas i **Main()**:

```
I Main()
är arrayens sista element FÖRE anropet   0

I metoden
är arrayens sista element före ändringen 0

                        och efter ändringen 1

I Main()
är arrayens sista element EFTER anropet  1
```

Som man ser har arrayen **a**:s sista element **a[999]** – kom ihåg att indexeringen hos arrays börjar med **0** – som hade initialvärdet **0**, EFTER anropet av metoden fått värdet **1**, fast denna ändring inte gjorts i **Main()** utan i metoden **Method()**, dessutom med arrayen **b** och inte med **a**. Detta verkar bryta mot de regler vi lärt oss om lokala variablers livslängd, därför att **a** trots allt är en lokal variabel i **Main()** och därmed inte giltig i **Method()**. Samma sak gäller för **b** som är lokal variabel i **Method()** och därmed inte giltig i **Main()**. Gåtans lösning är att det handlar endast om *en och samma* array till vilken **a** och **b** är bara två olika *referenser*. Därför pratar vi i utskriften ovan inte om arrayen **a** och inte om arrayen **b** utan om *arrayen*, för det finns bara en. För att första detta bättre låt oss titta på följande minnesbild som ska förtydliga vad som händer i programmet **ArrayParam**:



Vi vet att varje **int** tar 4 bytes i minnesutrymme. Därmed tar hela arrayen **a** med 1 000 **int**-element 4 000 bytes. Detta ”stora” minnesutrymme allokeras av satsen:

```
int[] a = new int[1000];
```

a är en referensvariabel som lagrar ett hexadecimalt tal, säg 12EFE0 (decimalt: 1241056) som är arrayens adress. Adresser visas i datavärlden – det är en de facto-standard – som tal i hexadecimalt format. Med *adress* menas alltid en plats i datorns *RAM*-minne (*Random Access Memory*). När en array definieras lagras den vid en adress och arraynamnet blir en länk mellan programmet och denna fysiska adress. När arrayen **a** sedan i metodanropet **Method(a);** skickas som en aktuell parameter, då överförs inte arrayens vär-

den utan arrayens *adress* till metoden **`Method()`**. Denna adress tas emot av den formella parametern **`b`** som är definierad i metodens parameterlista som en array av **`int`**. På så sätt hamnar **`a`**:s adress, det hexadecimala talet 12EFE0 i minnescellen **`b`**. Dvs **`b`** lagrar **`a`**:s adress som tar 4 bytes. Därmed pekar både **`a`** och **`b`** på en och samma array. Någon kopiering av arrayinnehållet på 4 000 bytes till en ny plats förekommer inte. Endast adressen på 4 bytes kopieras till **`b`** vid metodanropet. I **`Main()`** kommer man åt arrayen med **`a`** och i **`Method()`** gör man det med **`b`**. När vi sedan i **`Method()`** ändrar värdet i arrayens sista element med **`b`** från **`0`** till **`1`**, kan ändringen ses i **`Main()`** med **`a`**.

Den ovan beskrivna metoden för överföring av parametrar kallas *referensanrop*. Dvs inte parametrarnas värden utan deras adresser överförs vid metodanropet. När parametrarnas *adresser* överförs och inte deras värden, förekommer ingen fördubbling av minnesåtgång. Alla eventuella ändringar i metoden återspeglas i **`Main()`**. Valet av parameteröverföringsmetod styrs av datatypen:

> I C# väljs automatiskt referensanrop (Call by reference) för parameteröverföring vid metodanrop, om parametern är av datatypen array.

Låt oss nu även gå in på med vilken syntax programmet **`ArrayParam`** använder en array som en parameter i en metod.

1. Att definiera en metod med array som parameter
har gjorts i metoden **`Method()`** genom att definiera den formella parametern som en array av **`int`** dvs samma datatyp som den aktuella parametern har i anropet:

```
int[] b
```

Antalet element inom hakparentesen får inte anges. Att antalet element inte behövs här beror på att en formell parameter får sitt initialvärde från den anropande metoden. Även arraystorleken följer med vid anropet. Detta har i sin tur att göra med att hela definitionen av en metod endast är en mall, en föreskrift om vad som ska hända om metoden anropas, en potentiell kod som blir aktuell först när vi anropar metoden. I metoden **`Method()`** står definitionen av parametern **`b`** till datatypen array av **`int`** som vanligt i parameterlistan och därmed i metodhuvudet:

```
static void Method(int[] b)
```

2. Att anropa en metod med array som parameter
sker genom att skriva den aktuella parametern som array utan hakparenteser i anropet:

```
Method(a);
```

Anmärkningsvärt är att det för första gången dyker upp en array utan hakparenteser. Så, tittar man inte på definitionssatsen några rader ovan kan man inte känna igen **`a`** som array. Anledningen till att hakparentesen inte får stå efter arrayen **`a`** i anropssatsen är just det vi sade ovan om referensanrop: Anropet skickar inte hela arrayen med dess värden till **`Method()`** utan endast referensen **`a`**. En hakparentesens skulle tolkas som kod som anger index som specificerar ett visst element i arrayen. En anropssats av typen

`Method(a[999]);` skulle skicka endast *ett element* av arrayen nämligen det med index **`999`**. Det blir i så fall ett *tal* av typ **`int`** som skickas till metoden. Man kommer att få kompileringsfel i alla fall eftersom metodens formella parameter **`b`** är definierad som en *array* av **`int`** och inte som en vanlig **`int`**. Den enkla datatypen **`int`** kan inte konverteras till den sammansatta datatypen array av **`int`**. De automatiska typkonverteringsreglerna gäller endast för enkla datatyper. Det tänkbara alternativet **`Method(a[]);`** fungerar inte heller av samma anledning: Det handlar om en icke-definitionssats där hakparentesens innehåll tolkas som index. Men index får aldrig utelämnas (se punkt **1**). För att skicka en array som parameter till en metod måste alltså arrayen i metodanropet skrivas endast med arraynamnet *utan hakparentes*. Självklart måste arrayen innan anropet vara definierad i **`Main()`** som vanligt med hakparentes och en uppgift om storleken. Arraynamnet används vid anropet som adressen till arrayen.

5.6 Sökning och sortering

Ett viktigt – numera självklart – användningsområde för datorer är sökning i och sortering av stora datamängder. Programmeringstekniskt sett kan sådana applikationer inte skrivas utan arrays (eller högre datatyper). Därför är sökning och sortering klassiska tillämpningar för sammansatta datatyper. Samtidigt ökar behovet av modularisering ju mer avancerade datatyper man använder i sitt program. Nu när vi lärt oss att skicka arrays som parametrar till metoder, kan vi modularisera program som arbetar med arrays. Detta är nödvändigt för att koncentrera sig på den egentliga uppgiften nämligen sökning, sortering eller andra applikationer som t.ex. kryptering (kommer att tas upp i nästa avsnitt). När man söker i eller sorterar data finns redan ett material i form av databaser, tabeller eller listor osv. som man använder. För att skaffa ett liknande underlag för våra testprogram har vi valt att låta den i C# inbyggda slumptalsgeneratorn producera materialet och lagra det i en array.

Slumptal i en array

Eftersom vi i fortsättningen kommer att jobba med flera program som använder slumptal lagrade i en array vill vi skriva en metod som kan användas av alla dessa program. Vi har valt formen av en **`void`**-metod för att generera ett antal slumpvärden och tilldela dem till elementen i en array:

```
// RandArray.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 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");
   }
}
```

För förståelse av biblioteksmetoden **`Next()`** hänvisas till hantering av slumptal på sid 65. Det nya i koden ovan är att slumptalen lagras i en array som kommer att användas av fler program vilket demonstrerar inte bara modularisering utan även återanvändning

av kod. Filen ovan innehåller inte ett fullständigt program utan endast en klass med **void**-metoden **Rand()** som har fyra parametrar varav den ena är en array av **int**, kallad **no** som lagrar slumptalen. Arrayen deklareras i parameterlistan och tilldelas i kroppen mellan **a** och **b** via satsen:

```
no[i] = r.Next(a, b);
```

som i en **for**-sats anropar den biblioteksmetoden **Next()** som i sin tur i varje varv av loopen slumpar fram *ett* slumptal mellan **a** och **b**. Vi har använt denna metod tidigare i andra program. **for**-satsen som anropar metoden skriver ut slumptalen. Antalet arrayelement bestäms i början av **Main()** i följande program:

```
// SearchTest.cs
// Skapar en array och skickar den till metoden Rand() där den
// tilldelas slumptal. Ändringen fås tillbaka pga referensanrop.
// Den tilldelade arrayen skickas vidare till metoden MySearch()
// som söker efter ett inläst tal bland slumptalen
using System;

class SearchTest
{
  static void Main()
  {
    Random r = new Random();
    int a = 1, b = 1000, searchedNo;
    int[] intArray = new int[200];          // Default-initiering
    RandArray.Rand(r, intArray, a, b);      // Slump-tilldelning
    Console.Write("\tAnge tal som programmet ska söka efter:\t");
    searchedNo = int.Parse(Console.ReadLine()); // Sökt tal
    Search.MySearch(intArray, searchedNo);        // Anrop av
  }                                               // sökmetoden
}
```

Även om vi inte gått igenom programmets alla delar – klassen **Search** med metoden **MySearch()** fattas – ska vi titta på en körning för att bättre förstå vad som händer:

```
200 heltal mellan 1 och 1000 slumpas fram:

237 255 104 898 422 575 712 34 775 299 192 530 442 17 656 344 276
18 929 282 720 967 336 17 934 378 427 667 600 787 581 838 346
525 224 576 710 484 865 211 360 686 858 798 455 501 142 521 138
405 101 747 951 13 889 271 567 88 612 45 796 46 82 989 366
355 832 918 441 728 635 440 801 719 570 35 757 539 563 434 237
907 177 843 334 835 535 981 637 954 657 623 520 468 63 315 252
870 80 101 317 872 728 58 771 662 594 880 444 502 162 676 173
179 809 890 517 887 303 532 468 852 282 488 719 660 568 981 657
256 784 888 460 463 118 13 180 120 73 673 242 303 538 783 793
982 98 342 660 174 446 13 215 549 281 113 591 241 987 759 95
261 224 836 719 922 217 711 709 444 358 398 815 631 938 166 962
147 696 738 563 874 322 484 811 419 674 912 830 653 423 587 781
962 226 982 80 703 712 519

Ange tal som programmet ska söka efter: 519
Det sökta talet 519 är det 200:e elementet bland talen ovan.
```

Anrop av RandArray.Rand()

Anrop av Search.MySearch()

I programmet **SearchTest**:s **Main()**-metod finns bara anrop av två metoder samt definition av deras aktuella parametrar och inläsning av det sökta talet. En array av **int** har definierats med **200** element och tilldelats referensen **intArray**. I anropssatsen **RandArray.Rand(r, intArray, a, b);** skickas arrayen till metoden. Det anmärkningsvärda är följande: När arrayen **intArray** som aktuell parameter i anropet överförs till den formella parametern **no** i metoden **RandArray.Rand()**, är den definierad och default-initierad till **0**-värden. Faktum är att, när parametern är en array, så används referensanrop (sid 199) där den aktuella parametern **intArray**, och den formella parametern **no**, endast är två olika *referenser* till ett och samma minnesområde, till en och samma array. Med **intArray** definierar vi arrayen i **Main()** och anropar **RandArray.Rand()**. Med **no** tilldelar vi samma array i metoden **RandArray.Rand()** slumpvärden som överskriver arrayens default-värden. En sådan ”arbetsdelning” mellan olika metoder kan endast göras med referensanrop.

Efter anropet av slumpmetoden läses in ett värde till variabeln **searchedNo** som tillsammans med arrayen **intArray** skickas till metoden **Search.MySearch()**. När **MySearch()** anropas är arrayen **intArray** både definierad och tilldelad slumpvärden. Sökmetoden får alltså slumptalsvärden som överförs till den formella parametern **t**. Vid sidan om **no** är **t** nu en till minnescell som lagrar arrayen **intArray**:s adress i detta program. Även den här parameteröverföringen sker med referensanrop. Vid anropet skickas inte värdena i arrayelementen till metoden utan endast adressen som lagras i **intArray**. I själva verket är det arrayens adress som överförs till **MySearch()**, tas emot av **t** och används sedan i sökmetoden för att hitta det sökta talet i arrayen:

```
// Search.cs
// Metoden MySearch() tar emot två parametrar:
// arrayen t och heltalet s, det sökta elementet
// Söker efter den första förekomsten av s bland arrayelementen
using System;

class Search
{
   public static void MySearch(int[] t, int s)
   {
      int i;
      for (i = 0; i < t.Length; i++)     // Söker igenom array t
          if (t[i] == s)                 // Sökkriteriet
          {
             Console.WriteLine("\n\tDet sökta talet " + t[i] +
                         " är det " + (i+1) + ":e elementet" +
                         " bland talen ovan.\n\n");
             break;                      // Bryter for-satsen
          }                              // när det sökta hittats
      if (i == t.Length)
         Console.WriteLine("\n\tDet sökta talet finns ej " +
                                    "bland talen ovan.\n\n");
   }
}
```

Det sökta talet skickas med den aktuella parametern **searchedNo** och tas emot av den formella parametern **s**. Nu ska vi titta på vad **void**-metoden **MySearch()** egentligen gör och hur den hittar eller inte hittar det sökta talet. Arrayen och det sökta talet är givna. Frågan är: finns det sökta talet i arrayen? Om ja, på vilken position? Algoritmen är väldigt rak och enkel och kallas för *linjär sökalgoritm*:

1. Gå igenom alla element i arrayen dvs sök igenom arrayen **t** från början till slutet (linjär sökning).
2. Jämför varje element med det sökta talet. Finns likhet med något element, skriv ut ett hittat-meddelande samt elementets position som är lika med index + 1. Har du hittat en likhet avbryt sökningen.
3. Har du gått igenom alla arrayelement utan att hitta någon likhet skriv ut ett ej-hittat-meddelande.

Denna algoritm hittar endast den första förekomsten av det sökta talet i arrayen och tar inte hänsyn till att det ev. kan finnas flera exemplar av det sökta talet i arrayen. Progammeringstekniskt har vi översatt algoritmens punkt **1** till C#-kod genom att i metoden **MySearch()** skriva en **for**-sats som söker igenom arrayen **t** från index **0** till **t.Length-1**. I denna **for**-sats finns en **if**-sats som implementerar algoritmens punkt **2** och i sin tur innehåller två satser: Hittat-meddelandet och **break**-satsen. En **break**-sats avbryter alltid den loop eller den **switch**-sats i vilken den står, här alltså **for**-satsen. Det är den som enligt anvisningen i punkt **2** gör att programmet endast hittar den första förekomsten av det sökta talet i arrayen. I punkt **3**:s implementering – den sista **if**-satsen i **MySearch()** – utnyttjar vi att **for**-satsens räknare **i** är väl definierad även *efter* **for**-satsen och att den har kvar det värde den fick där. Om sökningen gått igenom alla arrayelement utan att hitta något element som är lika med det sökta talet, har **for**-satsens räknare **i** nått värdet **t.Length** eftersom detta är första värdet som inte uppfyller **for**-satsens villkor **i < t.Length**. I detta fall avslutas **for**-satsen utan **break** med värdet **t.Length** för **i** så att villkoret till den efterföljande **if**-satsen blir uppfyllt och skriver ut ett Ej-hittat-meddelande.

Bubbelsortering

Sökning i och sortering av stora datamängder är klassiska tillämpningar för sammansatta datatyper, speciellt för arrays. Medan sökning i förra exemplet baserades på en linjär algoritm, bygger sortering på en ny algoritm, även om den har vissa likheter med sökning. Vi ska fortsätta kapitlet om arrays med en sorteringsalgoritm som är en vidareutveckling av algoritmen för platsbyte av *två* värden. Vi har i programmet **MiniSort** (sid 154) använt denna algoritm på två tecken:

```
if (char1 > char2)
{
   temp = char1;
   char1 = char2;
   char2 = temp;
}
```

Om tecknen står i fel ordning ska de byta plats. För att göra det läggs **char1**:s värde undan i en tredje, temporär variabel **temp**. Sedan tar vi **char2**:s värde och lägger det i **char1**. Till sist läggs värdet i **temp** (som ju har mellanlagrat **char1**:s värde) in i **char2**. Illustrationen på sid 154 bör underlätta förståelsen av denna process. I själva verket beskriver den en algoritm för sortering av *två* värden. För att utvidga algoritmen till *flera* värden kopplar vi den till den linjära sökalgoritmen som vi använde för sökning. Principen där var en **if**-sats inbakad i en **for**-sats. **for**-satsen söker igenom värdena i en array och **if**-satsen innehåller sökkriteriet. När det gäller sortering måste **if**-satsen istället byta plats på två värden om de står i fel ordning. Denna **if**-sats har vi ju redan skrivit för två tecken (se ovan). Det gäller bara att formulera den för två arrayelement och stoppa in den i en **for**-sats:

```
for (i = 0;  i < n-1;  i++)
     if (t[i] > t[i+1])
     {
        temp = t[i];
        t[i] = t[i+1];
        t[i+1] = temp;
     }
```

där **t** är en array som innehåller värdena som ska sorteras och **n** antalet element i arrayen. När två på varandra följande arrayelement **t[i]** och **t[i+1]** står i oönskad ordning ska de byta plats där **i** genomlöper alla index. Man skulle kunna tro att problemet vore löst med detta. Men eftersom **if**-satsen endast testar om *två grannvärden* står i fel ordning och byter sedan plats på *dem*, räcker koden ovan inte till att sortera arrayen fullständigt, även om **for**-satsen söker igenom hela arrayen. Jämförelsen mellan två grannvärden tar inte hänsyn till värden som står längre bort. Ett experiment bekräftar detta: Om man tillämpar koden ovan på en array av 20 heltal som med metoden **RandArray.Rand()** är utvalda ur intervallet [1, 100] får man följande resultat:

```
20 heltal mellan 1 och 100 slumpas fram:

75 2 24 94 30 88 10 42 50 54 27 47 45 83 34 86 67 66 14 14

De 20 slumptalen efter koden ovan:

2 24 75 30 88 10 42 50 54 27 47 45 83 34 86 67 66 14 14 94
```

Resultatet visar att sorteringen inte är klar, men att vi är på rätt väg. Arrayen är *delvis* sorterad. Bara om två grannvärden stod i fel ordning har de bytt plats och detta har gjorts löpande genom hela arrayen. Denna delsortering kallas för ett *pass* i en sorteringsalgoritm som är känd under beteckningen *bubbelsortering*. För att uppnå en fullständig sortering måste detta pass upprepas flera gånger vilket innebär att lägga in ovanstående **for**-sats i en ny **for**-sats som går igenom flera pass. I varje pass kommer en del värden att placera sig i rätt ordning. Metoden kan jämföras med luftbubblor i vattnet som så småningom stiger upp till vattenytan. Därav namnet *bubbel*sortering. Vi har implementerat bubbelsorteringsalgoritmen i följande externlagrade **void**-metod:

```
// Bubble.cs
// Sorterar heltal lagrade i arrayen t med en algoritm
// (bubbelsortering) som baseras på algoritmen för platsbyte av
// två objekt i programmet MiniSort (sid 154)
using System;

class Bubble
{
  public static void sort(int[] t)
  {
    int temp;
    for (int pass=0; pass<t.Length-1; pass++)
        for (int i=0; i<t.Length-1; i++)
            if (t[i] > t[i+1])          // Sortering i stigande
            {                           // ordning
               temp = t[i];             // Algoritm för platsbyte
               t[i] = t[i+1];           // av de två elementen
               t[i+1] = temp;           // t[i] och t[i+1]
            }
    Console.WriteLine("\tDe " + t.Length +
                      " slumptalen efter sortering:");
    Console.Write("\n\t");
    for (int i=0; i < t.Length; i++)    // Sorterad utskrift
        Console.Write(t[i] + " ");
    Console.WriteLine("\n\n");
  }
}
```

Bubbelsorteringsalgoritmen består alltså av en **`if`**-sats inbakad i en nästlad **`for`**-sats där **`if`**-satsen implementerar algoritmen för platsbyte av *två* värden. Den inre **`for`**-satsen söker igenom arrayelementen, utför *ett* sorteringspass och den yttre **`for`**-satsen upprepar sorteringspassen. Metoden **`sort()`** har arrayen **`t`** som ska sorteras som parameter och används i den inre **`for`**-satsen. Den anropas från **`Main()`** i följande program efter definitionen av arrayen **`intArray`** och dess tilldelning i metoden **`RandArray.Rand()`**:

```
// BubbleTest.cs
using System;

class BubbleTest
{
   static void Main()
   {
      Random r = new Random();
      int a = 1, b = 100;
      int[] intArray = new int[17];
      RandArray.Rand(r, intArray, a, b);
      Bubble.sort(intArray);
   }
}
```

En körning av programmet **`BubbleTest`** visar att sorteringen nu genomförts fullständigt:

```
17 heltal mellan 1 och 100 slumpas fram:

23 76 23 31 67 94 79 38 46 10 85 100 87 61 17 71 14


De 17 slumptalen efter sortering:

10 14 17 23 23 31 38 46 61 67 71 76 79 85 87 94 100
```

Andra algoritmer

Som en sista anmärkning till kapitlet sökning och sortering bör påpekas att de algoritmer som avhandlats här, är enkla och elementära. De är däremot inte de mest effektiva när det gäller att minimera antalet operationer och maximera snabbheten. Det finns effektivare (och mer komplicerade) algoritmer både när det gäller sökning och sortering som vi inte tar upp här. Vi nämner bara en algoritm som kallas *binärsökning* som heter så för att den i varje steg halverar arrayen man ska söka i. Den behöver ett mindre antal operationer och är därmed snabbare. När det gäller sortering finns den effektiva algoritmen *Quicksort* som bygger på *rekursion*. Rekursiva metoder är metoder som anropar sig själva – ett alternativ till repetition (loopar) som behandlas på sid 175.

5.7 Generiska metoder

I programmering är variabler platshållare för värden.
I *generiska metoder* kan variabler även användas som platshållare för *datatyper*.

Generiska metoder är metoder vars parametrar har variabla datatyper.
Ex.: I metoden `public static void G_out <T> (T[] t)` är parametern `t` är en array av typ `T` där `T` är en platshållare för datatyper.
Den variabla datatypen `T` (`T`ype) definieras med `<T>` och kan användas istället för vilken datatyp som helst: `int`, `double`, `char`, `string`, … .

I generiska metoder är de involverade datatyperna inte specifierade förrän man utvecklar koden. De bestäms först när metoderna anropas av de aktuella parameternas datatyper. Detta innebär en generalisering som kallas för *Generics* som kan tillämpas även på klasser. Man kan skriva ETT program för många tillämpningar.

Generics

I de flesta programmeringsspråken har man infört *Generics* som ett tillägg till standarden först i de nyare versioner av språket. I C++ t.ex. kom motsvarigheten *Templates* först på 90-talet. I Java introducerades generics 2004. I C# har det funnits stöd för *Generics* sedan 2005.

Genom att använda *Generics* behöver man inte längre skriva olika varianter av ett program som i praktiken löser (nästan) samma problem. Dessa skiljer sig programmeringstekniskt endast i datatypen till de involverade parametrarna. Alla dessa varianter kan förenas i ett och samma – numera *generiskt* – program i vilka datatyperna är variabler. Låt oss säga, vi vill skriva ett program för sortering av olika slags objekt. Det kan handla om sortering av heltal, decimaltal, bokstäver, strängar, eller … . Sorteringsalgoritmen till alls dessa program är den samma oavsett man sorterar heltal, decimaltal, bokstäver eller strängar. Metoden som implementerar algoritmen skrivs då generiskt, dvs med variabla datatyper, så att den kan användas för att sortera olika typer av objekt beroende på i vilket syfte den anropas. Låt oss titta på följande exempel:

```
// G_Output.cs
// Generisk metod G_out <> () skriver ut en array av godtycklig
// variabel datatyp T som kan vara int, double, char eller string
// foreach loopar igenom och skriver ut listans alla element
using System;
using System.Collections.Generic;
class G_Output
{
   public static void G_out <T> (T[] t)
   {
      Console.Write("\t");
```

```
      foreach (T element in t)
         Console.Write(element + "  ");
      Console.WriteLine("\n");
   }
}
```

Metoden **G_out <> ()** i klassen **G_Output** är en generisk variant av den vanliga metoden **Ut()** i klassen **Skriv** som presenterades tidigare när vi behandlade listor (*Progr1, 7.9*). Det som gör att denna metod är generisk är den annorlunda syntaxen i metodhuvudet:

```
public static void G_out <T> (T[] t)
```

Till skillnad från vanliga metoder har denna metod *två* parameterlistor. Den ena är den vanliga med runda parenteser **(T[] t)** som innehåller parametern **t**, bara att dess datatyp är en array av **T**. Den andra är den ”generiska parameterlistan” **<T>** där **T** definieras som en formell parameter för en datatyp som bestäms när metoden anropas, t.ex. så här: **G_Output.G_out(hel);** **T** får den datatyp som i det anropande programmet har tilldelats variabeln **hel**. Har vi t.ex. definierat **hel** som en **int**, så antar den formella parametern **T** den aktuella parametern **int**. I generiska metoder finns det alltid en sådan *typparameter*. I det program där vi testar generiska metoder, anropas **G_out <> ()** fyra gånger, varje gång med en annan datatyp, närmare bestämt med **int**, **double**, **char** och **string**. Med hjälp av dessa bildas sedan med koden **T[]** arrays av **int**, **double**, **char** och **string**. Den vanliga parametern **t** definieras då med koden **T[] t** till sådana arrays. Här följer nu det program som testar och anropar två generiska metoder:

```
// GenericTest.cs
// Testar de generiska metoderna G_out <> () och G_sort <> ()
// Skapar 4 arrays av olika typer: int, double, char och string
// och skickar dem till G_out <> () för utskrift och till
//                      G_sort <> () för sortering
// Generiska metoderna anropas som vanliga metoder
// Utskrift sker före och efter sortering
using System;
class GenericTest
{
   public static void  Main()
   {
     int[] hel  =  { 9,   7,   2,   1,   8,   5,   4,   3,   6 };
     double[] deci =
                 { 9.9, 7.7, 2.2, 1.1, 8.8, 5.5, 4.4, 3.3, 6.6 };
     char[] boks = {'h', 'c', 'f', 'a', 'e', 'i', 'b', 'd', 'g'};
     string[] text = { "zeta", "beta", "gamma", "psi", "alpha" };
     Console.WriteLine(
      "\n\tOlika datatyper skrivs ut med samma generiska metod" +
      "\n\tFÖRE SORTERING:\n");            // Osorterad utskrift
     G_Output.G_out(hel);                  // Anrop av generisk
     G_Output.G_out(deci);                 // metod G_out <> ()
     G_Output.G_out(boks);
```

```
      G_Output.G_out(text);
      Console.WriteLine(
        "\tDe olika typerna sorteras med samma generisk metod");
      G_Bubble.G_sort(hel);                  // Sortering: Anrop
      G_Bubble.G_sort(deci);                 // av generisk metod
      G_Bubble.G_sort(boks);                 // G_sort <> ()
      G_Bubble.G_sort(text);
      Console.WriteLine("\toch skrivs ut EFTER SORTERING:\n");
      G_Output.G_out(hel);                   // Sorterad utskrift
      G_Output.G_out(deci);
      G_Output.G_out(boks);
      G_Output.G_out(text);
    }
}
```

Den vitmarkerade koden visar fyra anrop av den generiska metoden **`G_out <> ()`**. Det anmärkningsvärda är att dessa anrop inte skiljer sig alls från anrop av vanliga metoder. De aktuella parametrarna **`hel`**, **`deci`**, **`boks`** och **`text`** är definierade som arrays av **`int`**, **`double`**, **`char`** resp. **`string`** och skickar, när de anropas, inte bara sina vanliga värden – heltalen, decimaltalen, bokstäverna och strängarna – till de anropade metoderna, utan även sina datatyper. Medan de vanliga värdena i resp. array går till den formella parametern **`t`** i resp. metods runda parameterlista, går datatyperna arrays av **`int`**, **`double`**, **`char`** och **`string`** till parametern **`T`** i resp. metods ”generiska” parameterlista **`<T>`**. Därmed blir varje datatyp specificerad och insatt på alla ställen där **`T`** står i den generiska metoden, vare sig i huvudet eller i kroppen. Så här blir resultatet av en körning av programmet **`GenericTest`**:

```
Olika datatyper skrivs ut med samma generiska metod
FÖRE SORTERING:

9  7  2  1  8  5  4  3  6

9,9  7,7  2,2  1,1  8,8  5,5  4,4  3,3  6,6

h  c  f  a  e  i  b  d  g

zeta  beta  gamma  psi  alpha

De olika typerna sorteras med samma generiska metod
och skrivs ut EFTER SORTERING:

1  2  3  4  5  6  7  8  9

1,1  2,2  3,3  4,4  5,5  6,6  7,7  8,8  9,9

a  b  c  d  e  f  g  h  i

alpha  beta  gamma  psi  zeta
```

Som man ser har heltalen, decimaltalen, bokstäverna och strängarna dvs värdena i de fyra olika arrays skrivits ut som ett resultat av de vitmarkerade anropen i programmet **GenericTest** på förra sidan. Alla fyra anrop har gått till en och samma generisk metod **G_out <> ()** (sid 214) som skriver ut dem. Visserligen behöver man skriva fyra olika anrop i programmet **GenericTest**. Men man behöver definiera och koda själva metoden bara en gång, vilket innebär en stor effektivitet i utvecklingsarbetet.

Generisk bubbelsortering

Men körresultatet ovan har också andra delar, precis som själva programmet **GenericTest**. Efter att värdena skrivits ut skickas de till en annan generisk metod som sorterar dem. Detta görs i **GenericTest** med anropen:

```
G_Bubble.G_sort(hel);
G_Bubble.G_sort(deci);
G_Bubble.G_sort(boks);
G_Bubble.G_sort(text);
```

Även dessa anrop kan man inte skilja från anrop till vanliga metoder, fast metoden **G_sort <> ()** är generisk. Efter sorteringen skickas arrayvärdena igen till utskrift, så att vi ser dem sorterade i utskriften ovan – och detta sker inte bara för hel- och decimaltalen samt bokstäverna utan även för strängarna. Även här använder vi oss av en enda generisk metod som vi nu ska titta närmare på:

```
// G_Bubble.cs
// Genersik metod G_sort <> () sorterar en array av godtycklig
// variabel datatyp T som kan vara int, double, char eller string
using System;
using System.Collections.Generic;

class G_Bubble
{
  public static void G_sort <T> (T[] t) where T : IComparable<T>
  {                                       // Krävs för CompareTo()
    T temp;
    for (int pass=0; pass<t.Length-1; pass++)
        for (int i=0; i<t.Length-1; i++)
            if (t[i].CompareTo(t[i + 1]) > 0) // Om t[i] > t[i+1]
            {                                 // Sortering i sti-
               temp   = t[i];                 // gande ordning
               t[i]   = t[i + 1];             // Algoritm för
               t[i+1] = temp;                 // platsbyte
            }
  }
}
```

Metoden **G_sort <> ()** i klassen **G_Bubble** är en generisk variant av den vanliga metoden **sort()** i klassen **G_Bubble** som presenterades när vi behandlade sökning och sortering (*Progr1, 7.7*). Här gäller samma som vi sa om den första generiska metoden **G_out <> ()**: Den generiska formella parametern **T** står för datatyper som är kopplade

till den aktuella anropsparametern som skickas till den vanliga formella parametern **t**, dvs för datatyperna till de objekt som ska sorteras.

Constraints

Till skillnad från **G_out <> ()** har vi i den generiska metoden **G_sort <> ()** ett tillägg i metodhuvudet:

```
public static void G_sort <T> (T[] t) where T : IComparable<T>
```

Tillägget **where T : IComparable<T>** är en s.k. *constraint*, dvs en *restriktion* som läggs på **T**. Den är nödvändig eftersom vi i metodens kropp använder oss av ett villkor i **if**-satsens huvud som ska jämföra två på varandra följande element i arrayen:

```
if (t[i].CompareTo(t[i + 1]) > 0)
```

Motsvarigheten till detta i den vanliga icke-generiska metoden **sort()** är:

```
if (t[i] > t[i + 1])
```

Anledningen till att denna kod inte fungerar i den generiska metoden är att vi inte längre har att göra med en array av **int** vars element ska jämföras med varandra, utan med en generaliserad datatyp **T** som kan vara vilken datatyp som helst. Hur ska koden avgöra sanningsvärdet till ett sådant villkor om **T** är t.ex. en sträng? Självfallet måste den ta strängarnas begynnelsebokstäver och jämföra deras ASCII-koder med varandra för att avgöra vilken som är större. Men en sådan ”intelligens” finns inte automatikst inlagd i den generaliserade datatypen **T**, utan den är förprogrammerad i metoden **CompareTo()**. För att kunna åt denna kod måste **T** ärva denna metod som i sin tur finns i Interfacet **IComparable<>**. Det är därför vi måste skriva tillägget **where T : IComparable<T>** i huvudet till metoden **G_sort <> ()**. Annars kan vi inte kompilera **if**-villkoret

```
t[i].CompareTo(t[i + 1]) > 0
```

Det enklare alternativet **t[i] > t[i + 1]** som betyder samma sak, fungerar inte heller när vi arbetar med den generaliserade datatypen **T** istället för med **int** eller en annan specifik datatyp.

I generisk programmering kallas konstruktionen **where T : IComparable<T>** en *constraint* dvs en *restriktion* som man lägger på **T**. Just denna constraint innebär att data av typ **T** ska vara jämförbara. Man ska kunna använda jämförelseoperatorerna **>**, **<**, **==** osv. på dem. Interfacet **IComparable<>** innehåller ett antal fördefinierade metoder som implementerar denna möjlighet.

5.8 Kryptering av text

Vi ska nu dra lite praktisk nytta av våra samlade kunskaper om bl.a. slumptal, ASCII-koder, array, stränghantering, metoder och referensanrop, för att med ganska enkla medel skriva en liten applikation om kryptering av text. Egentligen har vi redan skrivit en sådan, nämligen klassen **`EncryptStr`** med **`return`**-metoden **`Encrypt()`** (sid 140). Men då löstes problemet med bibliotekslkassen **`String`**. Nu ska vi göra det med en egen array av **`char`** och en **`void`**-metod istället. Följande program läser in text som en **`char`**-array, skickar den till **`void`**-metoden **`Encrypt()`** där den krypteras resp. återställs teckenvis med ett slumptal som krypteringsnyckel. Tekniken som används för kryptering är samma som i **`EncryptStr`**-metoden, fast ännu enklare i och med man arbetar på **`char`**-nivå. Ett **`String`**-objekt kan inte manipuleras på **`char`**-nivå. Nu behöver strängen själv inte kopieras till en annan plats utan kan pga referensanrop krypteras på samma ställe, varför **`char`**-programmet behöver hälften av det minnesutrymme som det gamla **`String`**-programmet behövde.

```
// EncryptCharTest.cs
// Läser in text som en char-array och skickar den med en slump-
// krypteringsnyckel till metoden Encrypt() där den krypteras
// Referensanrop gör den krypterade texten tillgänglig i Main()
// Encrypt() anropas en andra gång med den krypterade texten och
// inverterad (negativ) krypteringsnyckel för att återställa den
using System;

class EncryptCharTest
{
   static void Main()
   {
      Random r = new Random();
      int key = r.Next(1, 501);                 // Slump-krypte-
                                                // ringsnyckeln
      Console.Write("\nSkriv text som ska krypteras:\t");
      char[] text = Console.ReadLine().ToCharArray();
      Console.Write("\n\tOkrypterad text:\t");
      Output(text);

      EncryptChar.Encrypt(text, key);           // 1:a anropet
                                                // krypterar
      Console.Write("\n\n\tKrypterad text:\t\t");
      Output(text);                             // text är ändrad

      EncryptChar.Encrypt(text, -key);          // 2:a anropet
                                                // återställer
      Console.Write("\n\n\tÅterställd text:\t");
      Output(text);                             // text är ändrad
      Console.WriteLine("\n\nKrypteringsnyckeln:\t\t" +
                                               key + '\n');
   }
```

```
   static void Output(char[] a)                  // Metod som
   {                                             // skriver ut
      for (int i = 0; i < a.Length; i++)         // en array
          Console.Write(a[i]);
   }
}
```

Med en array av **char** allokeras minne för texten med en maximal längd som är föreskriven av metoden **Console.ReadLine()**, något antal tecken som ryms på en rad, kanske **80** eller lite fler. Sedan överförs parametern **text** med ett första anrop av metoden **Encrypt()**:

```
EncryptChar.Encrypt(text, key);
```

som är definierad i klassen **EncryptChar** (se nedan), till metoden **Encrypt()**. I detta anrop används automatiskt referensanrop eftersom **text** är definierad som array. Därför är ändringarna som görs med **text** i metoden **Encrypt()**, tillgängliga efter anropet. Texten är okrypterad före och krypterad efter anropet både i **Encrypt()** och i **Main()**. Den andra parametern **key** däremot överförs med vanligt värdeanrop – dvs med kopiering av värdena – eftersom denna parameter är definierad till den enkla datatypen **int**. Efter **Encrypt()**:s första anrop skrivs den krypterade texten ut. Sedan anropas **Encrypt()** andra gången med **-key**, det negativa värdet av **key**, för att återställa texten som sedan skrivs ut för kontroll. Hur krypteringsmetoden fungerar, förstår man bäst om man samtidigt tittar på metoden **Encrypt()**:

```
// EncryptChar.cs
// Tar emot en text via arrayen t och krypterar den genom att
// förskjuta alla tecken med n steg i teckentabellen
// Kontrollerar textens slut med arrayegenskapen Length

class EncryptChar
{
   public static void Encrypt(char[] t, int n)
   {
      for (int i = 0; i < t.Length; i++)
          t[i] = (char) (t[i] + n);
   }
}
```

Krypteringsmetoden är väldigt enkel: tecknens ASCII-värden ökas med **n** i satsen **t[i] = (char) (t[i] + n);** genom vanlig addition. Att det verkligen adderas **n** till *ASCII-koden* till **t[i]** beror på att **t[i]** är av typ **char** och att en teckenvariabel i aritmetiska uttryck tolkas som sin ASCII-kod – ett tal man kan räkna med. **for**-satsen som går igenom hela strängen genom att koppla loopens räknare till arrayens index, gör att hela texten förskjuts med **n** steg i ASCII-tabellen. **n** får sitt värde genom kopiering (värdeanrop) från **key** vid första och från **-key** vid andra anropet. **key**:s värde i sin tur slumpas fram i **Main()** med hjälp av **Random**-metoden **Next()**. Dess anrop med para-

metrarna **1** och **501** gör att vi får ett slumpvärde som är ett heltal mellan **1** och **500** som sedan skickas som krypteringsnyckel till **Encrypt()** via dess andra parameter. Vid andra anropet av **Encrypt()** skickas **-key** för att återställa texten. Genom att ersätta **t[i] + n** med mer sofistikerade formler kan man utveckla mer avancerade krypteringsalgoritmer.

Programmet **EncryptCharTest** kan köras på olika sätt. Varje körning ger en annan slumpmässig krypteringsnyckel. Här ett exempel på en körning:

```
Skriv text som ska krypteras:   abcdef

        Okrypterad text:        abcdef

        Krypterad text:         åæçèéê

        Återställd text:        abcdef

Krypteringsnyckeln:             132
```

Man kan kontrollera krypteringen för hand: Man ser att bokstaven **a** förskjutits till **å**. Krypteringsnyckeln har vid denna körning varit **132**. ASCII-koden till **a** som är 97, har förskjutits **132** steg vidare till 97 + 132 = 229 som är koden till tecknet **å**. Testa gärna med programmet **Int2Char** (sid 126). Därför har **a** förskjutits till **å** med krypteringsnyckeln **132**. På samma sätt görs det med de andra tecknen i texten **abcdef**.

Självklart borde i en skarp applikation krypteringsnyckeln inte skrivas ut utan endast sparas i variabeln **key** för att använda den vid återställningen. Vi gör det här endast för experimentens skul.

Lägger man till filhantering i programmet **EncryptCharTest** kan samma metod **Encrypt()** användas för kryptering av filer.

Vi ska avsluta detta kapitel med ett sista avnsnitt som behandlar en utvidgning av arraybegreppet: en array vars element i sin tur är arrays.

5.9 2D Array

Med array kunde vi bearbeta större mängder av data. Men ibland är inte kvantiteten avgörande utan strukturen av data. Följande problem illustrerar detta:

”Sex elever i en klass har skrivit fyra olika prov och fått poäng i dem. Skriv ett program som lagrar elevernas poäng i alla prov och skriver ut dem. Sedan ska man kunna ändra ett provresultat till en elev samt skriva ut den uppdaterade elevens poäng.”

Elevernas poäng i olika prov kan lämpligast lagras i en tabell. Tvådimensionell array som används i följande program, är den naturliga datastrukturen för att lagra tabeller:

```
// DoubleArray.cs
// Elevernas poäng i olika prov lagras i en 2D-array (table)
// En elevs poäng i ett prov uppdateras och visas
// Både den ursprungliga och uppdaterade poängtabellen skrivs ut
// Tvådimensionell array modellerar tabellen och kommer åt
// arrayens element dvs tabellvärdena
using System;

class DoubleArray
{
   static void Main()
   {
      int[ , ] table =  { {67, 78, 84, 56},    // (6 x 4)-array
                          {49, 37, 59, 74},
                          {89, 54, 68, 34},
                          {72, 51, 85, 63},
                          {39, 41, 52, 27},
                          {98, 69, 79, 80}  };

      Console.Write("\n\t6 elevers provresultat i 4 prov:\n\n");

      PrintTable(table);

      Console.Write("Elev 4 har förbättrat poäng i prov 2. " +
                                        "Mata in ny poäng:\t");
      table[3, 1] = int.Parse(Console.ReadLine());

      Console.Write("\nElev 4:s nya poäng:\t");
      for (int k=0; k<4; k++)           // 4:e uppdaterade raden
          Console.Write(table[3, k] + "   ");

      Console.WriteLine("\n\n\tUppdaterad poängtabell:\n");

      PrintTable(table);
   }
```

```
   static void PrintTable(int[,] t)
   {
      for (int r=0; r<6; r++)
      {
          Console.Write("Elev " + (r+1) + ":\t\t");
          for (int k=0; k<4; k++)
              Console.Write(t[r, k] + "   ");
          Console.WriteLine();
      }
      Console.WriteLine();
   }
}
```

En tabell är en tvådimensionell struktur som vi redan stött på i olika sammanhang, t.ex. i nästlade **`for`**-satser med vars hjälp vi *skrivit ut* tabeller. Men vi har aldrig kunnat *lagra* tabeller i våra program för att sedan kunna referera till, komma åt och hantera tabellenvärdena. Tvådimensionell array är i C# och andra programspråk den datastruktur som kan löse detta problem. I programmet **`DoubleArray`** definieras och initieras den tvådimensionella arrayen **`table`** med koden:

```
int[ , ] table = { {...} ... {...} }
```

som är en array vars element i sin tur är arrays, dvs en dubbel eller nästlad array av **`int`** av storleken 6 x 4, dvs en stor array bestående av 6 små arrays, var bestående av 4 **`int`**-element, även kallad en (6 x 4)-array. Strukturen i arrayen kan jämföras med en tabell av 6 rader och 4 kolumner, se den nästlade initieringslistan på förra sidan. Storleken får inte anges explicit i hakparentesen, om man fortsätter initiera arrayen med initieringslistan. Storleken avläses automatiskt från initieringslistan på höger sidan. I själva verket är det inget annat än en 6-array av 4-arrays av **`int`**, om vi tillåter att elementen i en array i sin tur kan vara arrays. Eller varför inte prata om *nästlade arrays?* På så sätt kan man föreställa sig arrays av ännu högre dimension än två. I C# finns det ingen begränsning för att bilda flerdimensionella arrays: Man bara ökar antalet nivåer och i koden antalet komma i hakparentesen i definitionen ovan. Vi nöjer oss dock med två dimensioner där vi har den enkla tabellanalogin.

Deklarationen av den tvådimensionella arrayen **`table`** använder sig av initieringslistan som introducerades för endimensionella arrays på sid 197. Observera att det vid initieringen – närmare bestämt strax efter tilldelningsoperatorn – står *två inledande* klamrar efter varandra **`{ {67, ...`** och även i slutet av initieringssatsen *två avslutande* klamrar **`... , 80} };`** Klamrarna är nästlade i varandra, vilket är ett kännetecken för en tvådimensionell array. Ett annat är kommat i annars tomma hakparentesen vid definitionen. Vi har alltså att göra med en array på *första nivå* – representerad av de yttre klamrarna. I denna första nivå-array finns det 6 element som i sin tur är arrays. Därför är 6 storleken på denna *första* nivå-array. Dess element som i sin tur är arrays, befinner sig på en djupare *andra nivå* – representerade av de inre klamrarna – och har 4 element som är vanliga **`int`**-värden. Därför är 4 storleken på dessa *andra* nivå-arrays. Man kan också säga, vi har en *yttre* stor array som innehåller 6 *inre* små arrays med 4 **`int`**-element var,

som är nästlade i den stora arrayen. Därför är det hela en tvådimensionell (6 x 4)-array. Med hjälp av kodens layout har vi försökt att anknyta till tabellform. Tabellen har 6 rader och 4 kolumner. Varje *rad* representerar en *elev* med sina poäng i olika prov. Varje *kolumn* visar ett *prov* med poäng tillhörande olika elever. Därmed har vi bilden av en (6 x 4)-tabell till en (6 x 4)-array. Generellt kan tvådimensionella (m x n)-strukturer kodas med (m x n)-arrays där m och n är positiva heltal.

Åtkomst till element i en tvådimensionell array

Här hänvisas till diskussionen på sid 195: En arrays hakparenteser **[]** har inte samma betydelse i programmets alla satser. I definitionssatser omsluter hakparenteserna *antalet* element i arrayen dvs arrayens *storlek*. I alla andra satser omslutar hakparenteserna *index* till varje element av en array. Detta gäller förstås även för tvådimensionella arrays. Hakparenteserna i koden **table[3, 1]** i programmet **DoubleArray**, innehåller *indexen* till ett element i arrayen **table**. Självklart är det ett *dubbelindex* som refererar till ett **int**-värde. Man vill komma åt en tabellplats och ändra dess värde genom att läsa in ett nytt värde till den som kommer att skriva över det gamla. Låt oss säga, en elev har gjort omprov i ett ämne, förbättrat sina poäng, och man vill läsa in det nya värdet och föra in det i tabellen. Men vilken elev och vilket prov är det, vilket element i arrayen **table** är det? Även här måste vi hänvisa till indexregeln som även gäller för tvådimensionella arrays: Numreringen av index börjar alltid med 0 (sid 192). Det gäller: elementets position = index + 1, där med position menas numret som *människan* använder för att numrera elementen, medan index är det som skrivs i *koden*. Därför betyder dubbelindexet **[3, 1]** i satsen ovan *inte* elev 3, prov 1, utan enligt indexregeln: elev 4, prov 2. De hårdkodade värdena till arrayen **table** i programmet **DoubleArray** visar att det är värdet **51** som står i korset mellan rad 4 och kolumn 2 (sid 222). Alltså refererar koden **table [3, 1]** till värdet **51**. Man tar det *första* indexet **3** och räknar – genom att börja med 0 – *raderna* i den stora arrayen **table**. Så kommer man till tabellens rad 4 eller elev 4. Det innebär att söka igenom arrayen **table** på första nivå. Sedan tar man det andra indexet **1** och räknar – genom att börja med 0 – *kolumnerna* i den redan hittade raden 4. Så kommer man till tabellens kolumn 2 eller prov 2 och hittar där värdet **51**. Det är samma som att söka igenom arrayen **table** på andra, djupare nivå. Dubbelindexets första index refererar till arrayens första och det andra index till arrayens andra nivå. Denna generella regel tillämpas även i den nästlade **for**-satsen som lägger hela poängtabellen i **String**-variabeln **box** för att senare skriva ut **box** på skärmen:

```
for (int r=0; r<6; r++)            // Lägger table i box
{
    for (int k=0; k<4; k++)
        box += table[r, k] + "   ";
    box += '\n';
}
```

Den inre **for**-slingan skriver ut *en* rad, närmare bestämt den **r**:te raden genom att hålla fast det första indexet **r** och låta det *andra* indexet **k** gå igenom kolumnindexen **0**, **1**, **2**, **3**. Dessutom skickas mellan kolumnerna en tabulator till utskrift. Den yttre **for**-slingan låter den inre slingan att skriva ut raderna 6 gånger genom att låta det *första* indexet **r**

gå igenom radindexen **0**, **1**, **2**, **3**, **4**, **5**. Dessutom skickas ett radbyte mellan raderna till utskrift. På liknande sätt hade vi med nästlad **for**-sats skrivit ut en tabell över tal och multiplikationstabellen.

Efter uppdateringen av elev 4:s poäng i prov 2 vill vi verifiera ändringen genom att skriva ut just denna elevs poäng i alla prov dvs ta ut hela raden 4 ur tabellen med:

```
for (int k=0; k<4; k++)            // Lägger den 4:e uppdate-
    box += table[3, k] + "   ";    // rade raden i box
```

Som man ser är detta en kopia av den inre slingan från den nästlade **for**-satsen ovan med **r = 3**. Raden 4 har enligt indexregeln index **3**. Observera att **poäng**:s första index hålls fast och det andra indexet räknas upp. Varje enskild rad kan skrivas ut på det här sättet. Att ta ut en enskild kolumn ur tabellen och skriva ut alla elevers poäng från ett prov, t.ex. prov 2, borde gå med följande sats:

```
for (int r=0; r<6; r++)
    box += table[r, 1] + '\n';
```

Här har vi tagit den yttre slingan från den nästlade **for**-satsen ovan, eliminerat den inre slingan och ersatt den med utskrift av ett enda värde per rad. Till skillnad från radutskrift hålls **table**:s andra index fast och det första indexet räknas upp. Dessutom har radbytet lyfts in i satsen då blanksteg inte behövs när man skriver ut endast en kolumn. Prova gärna! Slutligen ger ett körresultat av programmet **DoubleArray**:

```
        6 elevers provresultat i 4 prov:

Elev 1:         67   78   84   56
Elev 2:         49   37   59   74
Elev 3:         89   54   68   34
Elev 4:         72   51   85   63
Elev 5:         39   41   52   27
Elev 6:         98   69   79   80

Elev 4 har förbättrat poäng i prov 2. Mata in ny poäng: 99

Elev 4:s nya poäng:     72   99   85   63

        Uppdaterad poängtabell:

Elev 1:         67   78   84   56
Elev 2:         49   37   59   74
Elev 3:         89   54   68   34
Elev 4:         72   99   85   63
Elev 5:         39   41   52   27
Elev 6:         98   69   79   80
```

5.10 Dynamiska arrays: Listor

Array har många fördelar när det gäller hantering av stora datamängder, men också en stor nackdel, nämligen att man i förväg måste ange storleken på arrayen utan att ha möjligheten att ändra den vid behov senare. Anta att vi vill ha ett program som läser data, t.ex. laddar ned text, bild eller ljud – från någon källa, säg en fil, och vi vet inte hur mycket data filen innehåller, när vi skriver kod. Det här problemet kan inte lösas med en vanlig array eftersom den tillämpar s.k. *statisk minnesallokering*, dvs minnesutrymmets storlek bestäms när man definierar arrayen. När koden kompileras reserveras minne av den angivna storleken som inte längre kan ändras under exekveringen. Därför kan en array inte klara av den här uppgiften. När man läser data från en fil ska minnesallokeringen helst göras samtidigt som filen läses under programmets körning. I det enklaste fallet ska man kunna läsa in data till ett C#-program utan att på förhand behöva ange dess storlek. Lösningen vore *dynamisk minnesallokering*, dvs minnesutrymmet kan utökas efter behov under programmets exekvering. En slags dynamisk array behövs. Och just en sådan dynamisk array är den nya datastrukturen **`List`** som vi ska stifta bekantskap med i detta avsnitt. **`List`** är inte bara dynamisk utan har även en mängd fördefinierade kraftfulla metoder som sorterar, söker i eller på annat sätt manipulerar listor, så att man själv inte behöver koda så mycket. I denna bemärkelse är listor bättre arrays.

Följande program visar ett exempel på denna nya datastruktur:

```
// List.cs
// Skapar en lista och skickar den till metoden Rand() där den
// fylls med slumptal. Listan skickas vidare till List-metoden
// Sort() där den sorteras. Utskrift sker före + efter sortering.
using System;
using System.Collections.Generic;          // Krävs för List

class List
{
   static void Main()
   {
      List<int> intList = new List<int>(); // List-objekt av int
      Random r = new Random();
      int a = 1, b = 1000;
      Console.WriteLine(
              "\n\t100 heltal mellan " + a + " och " + b +
              " slumpas till ett List-objekt:\n");
      RandList.RandL(r, intList, a, b);    // Slump-tilldelning
      Print.Out(intList);                  // Osorterad utskrift
      intList.Sort();                      // List-sortering
      Console.WriteLine(
              "\tHeltalen sorteras med List-metoden Sort():\n");
      Print.Out(intList);                  // Sorterad utskrift
   }
}
```

Klassen List

Klassen **List** är fördefinierad i C#-biblioteket **System.Collectins.Genetric**. För att använda listor måste vi skapa ett objekt av denna klass. Det gör man med satsen:

```
List<int> intList = new List<int>();
```

Variabeln som refererar till det nya objektet kallar vi **intList**. Det speciella med klassen **List** är att den måste kopplas till en datatyp. Här är den kopplad till **int**, dvs klassen heter egentligen **List<int>**. Vi har skapat en lista av **int**, ganska liknande en array av **int**, bara att vi nu inte behöver ange antal element. Det är just det dynamiska i listor till skillnad från arrays. Som en konsekvens får vi tilldela till en lista av **int** också bara heltal av typ **int**. Varje försök att tilldela till den andra än **int**-värden kommer att leda till kompileringsfel. Man kan förstås skapa även objekt av listor av alla andra datatyper inkl. andra klasser. Har man t.ex. definierat en klass **Person** kan man med **List<Person> p = new List<Person>();** skapa en lista över personer. **p** refererar då till ett objekt av typ **List<Person>**. Varje element i denna lista är i sin tur ett objekt av typ **Person**.

Listan **intList** vi skapat ovan är just nu tom. Den blir inte heller tilldelad i koden på förra sidan. För att fylla den med värden skickar vi den som parameter till metoden **RandL()** som vi definierar i klassen **RandList**:

```
// RandList.cs
// Metod Rand() slumpar fram heltal mellan a och b och
// lagrar dem i ett List-objekt med List-metoden Add()
using System;
using System.Collections.Generic;

class RandList
{
   public static void RandL(Random r, List<int> no, int a,
                                                    int b)
   {
      for (int i=0; i < 100; i++)            // Här fylls listan
          no.Add(r.Next(a, b));   // med slumptal
   }
}
```

Deklarationen av parametern i metoden **RandL()**:s parameterlista sker med koden **List<int> no**. Namnet **no** på den formella parametern är oväsentligt. Eftersom referensanrop tillämpas, pekar **no** i alla fall på samma objekt som **intList** dvs den lista som skapades i **Main()**. Så fyller vi den i **for**-satsen med 100 slumptal genererade av den gamla **Rand()**-metod som vi använt tidigare och som i varje varv skapar *ett* slumptal mellan a och b (**1** och **1000**). För att placera dem i listan använder vi oss av metoden **Add()** som är definierad i klassen **List**, därför anropet **no.Add()**. Varje anrop infogar *ett* slumptal i listan. Vi behöver inte ange i förväg hur lång listan ska vara. Den är öppen och växer vid behov. Det är fördelen med dynamiska arrays som tillhandahålls i klassen

List. Slumptalsgenereringsmetoden **Next()** anropas i **Add()**-metodens parameterlista med **r.Next(a, b)** som är definierad i bibliotekssklassen **Random** (sid 65) .

Vi har även modulariserat utskriftsproceduren med all layout som tillhör den, i metoden **Out()** i den externa klassen **Print** som ser ut så här:

```
// Print.cs
// Metoden Out() skriver ut en lista med en foreach-sats som
// loopar igenom listans ALLA element
using System;
using System.Collections.Generic;

class Print
{
   public static void Out(List<int> t)
   {
      Console.Write("\t");
      int i = 1;
      foreach (int element in t)
      {
         Console.Write(element + " ");
         if (i % 14 == 0)                          // Radbyte var
            Console.Write("\n\t");                  // 14:e utskrift
         i++;
      }
      Console.WriteLine("\n");
   }
}
```

I metodens huvud väljs namnet **t** för den formella parametern. Eftersom metodens anrop i **Main()** sker med den aktuella parametern **intList**, pekar **t** på samma lista som **intList**. Därför skrivs ut listans innehåll – de 100 slumptalen – när **Out()** anropas första gången direkt efter att listan blivit tilldelad i **Rand()**-metoden. Andra gången sker anropet efter sorteringen. All utskrift i **Out()** sker med hjälp av en kontrollstruktur som är typisk för listor och arrays och som inleds med det reserverade ordet **foreach**.

foreach-satsen i listor

Det är en kontrollstruktur som behandlades tidigare i detta kapitel (sid 194), fast då var det i samband med array. Nu används **foreach** med listor. Skillnaden är dock obetydlig. I klassen **Print** (ovan) ser huvudet till **foreach**-satsen ut så här:

```
foreach (int element in t)
```

Översatt till svenska:

För varje **element** av listan **t** gör:

Iterationsvariabeln **element** definieras till **int**. Men till skillnad från **for**-satsens räknare är **element** inget index (nr) i listan utan en variabel som pekar på själva värdet (innehållet) som står i listan. **t** är en referens till listan som ska loopas igenom. **for-**

each-satsen går igenom listans *alla* element, från det första till det sista. Variabeln **element** som i varje varv pekar på resp. listelementets värde, används sedan i loopens kropp för att göra det man önskar. I vårt exempel sätts den i följande anrop för att skriva ut listans element följt av ett mellanslag:

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

Mellanslaget samt resten av koden i metoden **Out()** är till för att få en snygg layout i utskriften. Räknaren **i** som vi själva definierar, håller reda på loopens varv och ger oss möjligheten att i följande **if**-sats infoga ett radbyte samt tabulator var 14:e utskrift utom i den allra första:

```
if (i % 14 == 0)
   Console.Write("\n\t");
```

Äntligen kan vi testa programmet **List** som *kan* resultera i följande utskrift:

```
100 heltal mellan 1 och 1000 slumpas till ett List-objekt:

378 297 220 134 803 115 218 227 346 300 508 559 845 872 417
829 559 105 477 869 602 493 117 713 541 92 572 988 796
982 184 431 259 39 566 724 465 722 14 817 235 751 446
256 650 231 413 914 907 297 464 943 557 957 999 533 181
155 594 359 191 231 79 365 764 725 948 454 307 341 12
485 739 661 635 852 695 862 711 958 680 659 729 147 166
242 522 303 688 681 544 958 129 656 274 652 320 82 493
573

Heltalen sorteras med List-metoden Sort():

12 14 39 79 82 92 105 115 117 129 134 147 155 166 181
184 191 218 220 227 231 231 235 242 256 259 274 297 297
300 303 307 320 341 346 359 365 378 413 417 431 446 454
464 465 477 485 493 493 508 522 533 541 544 557 559 559
566 572 573 594 602 635 650 652 656 659 661 680 681 688
695 711 713 722 724 725 729 739 751 764 796 803 817 829
845 852 862 869 872 907 914 943 948 957 958 958 982 988
999
```

”*Kan* resultera”, därför att det blir andra siffror i varje körning pga at det är slumptal som genereras och som är olika varje gång man kör programmet. Sorteringen görs i programmet **List**:s anrop (sid 226) av metoden **Sort()** som är fördefinierad i klassen **List**.

Övningar till kapitel 5

5.1 Skriv ett program som läser in 10 heltal från konsolen, lagrar dem i en array och skriver ut dem i omvänd ordning.

5.2 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. Använd klasserna **`RandArray`** (sid 207) och **`RandomNo`** som externa moduler.

5.3 Skriv ett program som läser in en sträng, lagrar den i en array av **`char`** och skriver ut den baklänges. Använd tekniken i programmet **`EncryptCharTest`** (sid 219) för att omvandla den inlästa strängen i en array av **`char`**.

5.4 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.

5.5 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.

5.6 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()`**.

5.7 Modifiera programmet **`List`** (sid 226) så att sorteringen av slumptalen görs med vår egen bubbelsorteringsmetod **`sort()`** (sid 212) istället för med den fördefinierade **`List`**-metoden **`Sort()`**. Testa först med array-notationen som **`sort()`** är skriven i. Försök sedan att skriva om **`sort()`** till en **`List`**-version.

5.8 Modifiera programmet **`ArrayOfRef`** (sid 200). Deklarera klassen **`Fish`**:s datamedlemmar som **`private`** och metoderna som **`public`**. Förse klassen med en konstruktor och en strängrepresentationsmetod **`AsString()`**. I övrigt ska det modifierade programmet göra samma sak som det ursprungliga.

Fullständiga lösningar till övningar (Facit)

I programmering finns alltid flera möjliga lösningar till en uppgift. Därför är det, som slarvigt kallas för lösningar, i själva verket endast *lösningsförslag*. Dessutom ges inga lösningsförslag till *projektuppgifterna* eller uppgifter som är relaterade till ett projekt, för att uppmuntra till egna lösningar. Istället finns det i projektens lydelse en mer eller mindre utförlig ledning resp. algoritm till lösningen.

Kapitel 1 Windowsprogrammering, sid 63:

Övning 1.1
Skapa en Console Application och kalla den för AdditionC. Den ska definiera och initiera två heltalsvariabler och producera t.ex. följande utskrift till konsolen:

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

`9` och `2` ska vara de värden som heltalsvariablerna blivit inirierade till i programmet.

Lösningen: (→ betyder musklickning, vänster- eller högermusklick)
→ New Project → Console Application, Name: AdditionC, Location: … → OK.
SOLUTION EXPLORER: → Program.cs → Exclude From Project → AdditionC → Add
→ New Item… → Code File → Name: AdditionC.cs → Add
Skriv följande kod i filen AdditionC.cs:

```
// AdditionC.cs
// Adderar talen 9 och 2 samt skriver ut resultatet
// från en konsolapplikation till konsolen
using System;

class AdditionC
{
   static void Main()
   {
      int number1 = 9;                    // Definition och initiering
      int number2 = 2;

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

Övning 1.2
Skapa en Windows Forms Application och kalla den AdditionW. Den ska göra samma sak som lösningen i övning 1.1, bara att MessageBoxen ska visas när man klickar på en knapp (med texten Visa MessageBox) i formfönstret. Förse MessageBoxen med rubriken Windows Addition.

Lösningen:
→ New Project → Windows Forms Application, Name: AdditionW, Location: … → OK.
HUVUDMENYN: → View → Toolbox

TOOLBOX: → Common Controls: Dubbelklicka på kontrollen Button.
PROPERTIES: Sätt **button1**-egenskaperna till följande värden:

Egenskap	Värde
AutoSize	True
Font	Tahoma; 12pt; style=Bold
Location	56; 45
Text	Visa MessageBox

Dubbelklicka på knappen i formfönstret och skriv kod i händelsemetoden button1_Click() så att filen Form1.cs får följande utseende:

```
// Form1.cs
// Adderar talen 9 och 2 samt skriver ut resultatet
// från en Windows applikation till en MessageBox
using System;
using System.Windows.Forms;

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

      private void button1_Click(object sender, EventArgs e)
      {
         int number1 = 9;
         int number2 = 2;

         MessageBox.Show("Summan av " + number1 + " och " + number2 +
              " är "  + (number1 + number2), "Windows Addition");
      }
   }
}
```

Övning 1.3
I både övn 1.1 och 1.2 är heltalsvärdena 9 och 2 hårdkodade. Vidareutveckla dessa övningar genom att skapa ett användarvänligt, interaktivt grafiskt gränssnitt där man kan mata in vilka heltal som helst och få summan utskriven i en MessageBox när man klickar på en knapp med texten Addera. Välj lämpliga rubriker för formen och MessageBoxen. Kalla projektet för Addition.

Lösningen:
→ New Project → Windows Forms Application, Name: Addition, Location: … → OK.
PROPERTIES: Sätt **Form1**-egenskaperna till följande värden:

Egenskap	Värde
Text	Addition
Size	400; 200

HUVUDMENYN: → View → Toolbox
TOOLBOX: → Common Controls: Dubbelklicka på kontrollen Label.

PROPERTIES: Sätt **label1**-egenskaperna till följande värden:

Egenskap	Värde
Location	30; 30
Text	Tal 1:

TOOLBOX: Dubbelklicka på kontrollen TextBox.
PROPERTIES: Sätt **textBox1**-egenskaperna till följande värden:

Egenskap	Värde
Location	90; 27
Size	100; 20
TextAlign	Center

TOOLBOX: Dubbelklicka på kontrollen Label.
PROPERTIES: Sätt **label2**-egenskaperna till följande värden:

Egenskap	Värde
Location	30; 80
Text	Tal 2:

TOOLBOX: Dubbelklicka på kontrollen TextBox.
PROPERTIES: Sätt **textBox2**-egenskaperna till följande värden:

Egenskap	Värde
Location	90; 77
Size	100; 20
TextAlign	Center

TOOLBOX: Dubbelklicka på kontrollen Button.
PROPERTIES: Sätt **button1**-egenskaperna till följande värden:

Egenskap	Värde
Location	275; 25
Size	90; 25
Text	Addera

Dubbelklicka på knappen Addera i formfönstret och skriv kod i händelsemetoden button1_Click() så att filen Form1.cs får följande utseende:

```
// Form1.cs
// Läser av två tal från två textfält i formfönstret och adderar dem
// Skriver ut resultatet till en MessageBox
using System;
using System.Windows.Forms;

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

```
      private void button1_Click(object sender, EventArgs e)
      {
         double no1 = Convert.ToDouble(textBox1.Text);
         double no2 = Convert.ToDouble(textBox2.Text);

         MessageBox.Show("Summan av " + no1 + " och " + no2 + " är " +
                                              (no1 + no2), "Resultat");
      }
   }
}
```

Övning 1.4
Skapa en Windows Forms Application och kalla den Division. Modifiera lösningen i övn 1.3 så att beräkningens resultat hamnar i ett textfält i formen istället för i en MessageBox. Välj den här gången division som räkneoperation.

Lösningen:
→ New Project → Windows Forms Application, Name: Division, Location: … → OK.
PROPERTIES: Sätt **Form1**-egenskaperna till följande värden:

Egenskap	Värde
Text	Division
Size	450; 250

Den grafiska designen av de två första labels och textfälten är identisk med övn 1.3. Så ta över därifrån. En tredje label och ett tredje textfält kommer till:

TOOLBOX: Dubbelklicka på kontrollen Label.
PROPERTIES: Sätt **label3**-egenskaperna till följande värden:

Egenskap	Värde
Location	215; 150
Text	Resultat:

TOOLBOX: Dubbelklicka på kontrollen TextBox.
PROPERTIES: Sätt **textBox3**-egenskaperna till följande värden:

Egenskap	Värde
Location	275; 147
Size	100; 20
TextAlign	Center

TOOLBOX: Dubbelklicka på kontrollen Button.
PROPERTIES: Sätt **button1**-egenskaperna till följande värden:

Egenskap	Värde
Location	275; 25
Size	90; 25
Text	Dividera

Dubbelklicka på knappen Dividera i formfönstret och skriv kod i händelsemetoden button1-_Click() så att filen Form1.cs får följande utseende:

```
// Form1.cs
// Läser av två tal från två textfält i formfönstret och dividerar dem
// Skriver ut resultatet till ett tredje textfält
using System;
using System.Windows.Forms;

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

      private void button1_Click(object sender, EventArgs e)
      {
         double no1 = Convert.ToDouble(textBox1.Text);
         double no2 = Convert.ToDouble(textBox2.Text);

         textBox3.Text = (no1 / no2).ToString();
      }
   }
}
```

Övning 1.5

Skapa en Windows Forms Application och kalla den SafeDivision. Ta bort filerna Form1.cs och Form1.Designer.cs från projektet. Infoga istället filerna med samma namn från projektet Division (övn 1.4) i projektet SafeDivision. Döp om i båda filerna raderna namespace Division till namespace SafeDivision. Modifiera koden i Form1.cs genom att införa ett egengenererat undantag *(Progr1, 8.2)* i programmet för fallet att användaren matar in 0 i det andra textfältet, dvs när division med 0 uppstår. Styyr meddelandena från undantagshanteringen till en MessageBox.

Lösningen:
→ New Project → Windows Forms Application, Name: SafeDivision, Location: … → OK.
SOLUTION EXPLORER: → Form1.cs → Delete → Form1.Design.cs → Delete → SafeDivision → Add → Existing Item… → Form1.cs (från projektmappen Division) → Add → SafeDivision → Add → Existing Item… → Form1.Design.cs (från projektmappen Division) → Add
→ Form1.cs: Ersätt namespace Division med namespace SafeDivision.
→ Form1.Design.cs: Ersätt namespace Division med namespace SafeDivision.

Modifiera filen Form1.cs så att den får följande utseende:

```
// Form1.cs
// Läser av två tal från två textfält i formfönstret och dividerar dem
// Skriver ut resultatet till ett tredje textfält
// Kastar och hanterar undantag om det matas in 0 i det andra textfältet
using System;
using System.Windows.Forms;

namespace SafeDivision
{
```

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

      private void button1_Click(object sender, EventArgs e)
      {
         double no1 = Convert.ToDouble(textBox1.Text);
         double no2 = Convert.ToDouble(textBox2.Text);

         try
         {
            if (no2 == 0)
               throw new DivideByZeroException();  // Undantag kastas
            else
               textBox3.Text = (no1 / no2).ToString();
         }
         catch (DivideByZeroException exception)   // Undantag fångas upp
         {
            MessageBox.Show("\tOBS! Du försökte dividera med 0.\n\t"    +
                            "Det går inte att dividera med 0.\n\n\t"    +
                            "C# undantagsmeddelande:\n\n"               +
                                  exception.ToString(), "Felmeddelande");
         }                                            // Undantag skrivs ut
      }
   }
}
```

Övning 1.6
Vidareutveckla övningsserien 1.1-1.5 till en komplett kalkylator med interaktivt grafiskt gränssnitt och undantagshantering som inkluderar de fyra räknesätten.

Lösningen:
→ New Project → Windows Forms Application, Name: Calculator, Location: … → OK.
PROPERTIES: Sätt **Form1**-egenskaperna till följande värden:

Egenskap	Värde
Text	Calculator
Size	450; 250

Den grafiska designen av de tre labels och textfälten är identisk med övn 1.3 resp. 1.4. Så ta över därifrån. Döp om de tre labels Text-egenskaper till Number1:, Number2: och Result. Däremot bygger vi här fyra nya knappar för de fyra räknesätten:

TOOLBOX: Dubbelklicka på kontrollen Button.
PROPERTIES: Sätt **button1**-egenskaperna till följande värden:

Egenskap	Värde
Location	275; 25
Size	90; 25
Text	+

Dubbelklicka på knappen **+** i formfönstret. Återgå till formfönstret.

TOOLBOX: Dubbelklicka på kontrollen Button.
PROPERTIES: Sätt **button2**-egenskaperna till följande värden:

Egenskap	Värde
Location	275; 50
Size	90; 25
Text	-

Dubbelklicka på knappen **-** i formfönstret. Återgå till formfönstret.

TOOLBOX: Dubbelklicka på kontrollen Button.
PROPERTIES: Sätt **button3**-egenskaperna till följande värden:

Egenskap	Värde
Location	275; 75
Size	90; 25
Text	x

Dubbelklicka på knappen **x** i formfönstret. Återgå till formfönstret.

TOOLBOX: Dubbelklicka på kontrollen Button.
PROPERTIES: Sätt **button4**-egenskaperna till följande värden:

Egenskap	Värde
Location	275; 100
Size	90; 25
Text	/

Dubbelklicka på knappen / i formfönstret. Återgå till formfönstret.

Skriv kod i knapparnas händelsemetoder så att filen Form1.cs får följande utseende:

```
// Form1.cs
// Kalkylator för de fyra räknesätten
// Läser av två tal från två textfält och beräknar deras
// summa, differens, produkt eller kvot
// Skriver ut resultatet till ett tredje textfält
// Kastar och hanterar undantag vid division med 0
using System;
using System.Windows.Forms;

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

      private void button1_Click(object sender, EventArgs e)
      {
         double no1 = Convert.ToDouble(textBox1.Text);
```

```
         double no2 = Convert.ToDouble(textBox2.Text);
         textBox3.Text = (no1 + no2).ToString();
      }

      private void button2_Click(object sender, EventArgs e)
      {
         double no1 = Convert.ToDouble(textBox1.Text);
         double no2 = Convert.ToDouble(textBox2.Text);

         textBox3.Text = (no1 - no2).ToString();
      }

      private void button3_Click(object sender, EventArgs e)
      {
         double no1 = Convert.ToDouble(textBox1.Text);
         double no2 = Convert.ToDouble(textBox2.Text);

         textBox3.Text = (no1 * no2).ToString();
      }

      private void button4_Click(object sender, EventArgs e)
      {
         double no1 = Convert.ToDouble(textBox1.Text);
         double no2 = Convert.ToDouble(textBox2.Text);

         try
         {
            if (no2 == 0)
               throw new DivideByZeroException();  // Undantag kastas
            else
               textBox3.Text = (no1 / no2).ToString();
         }
         catch (DivideByZeroException exception)   // Undantag fångas upp
         {
            MessageBox.Show("\tOBS! Du försökte dividera med 0.\n\t"  +
                            "Det går inte att dividera med 0.\n\n\t"  +
                            "C# undantagsmeddelande:\n\n"             +
                                exception.ToString(), "Felmeddelande");
         }                                          // Undantag skrivs ut
      }
   }
}
```

Kapitel 2 Objektorienterad programmering (OOP), sid 119:

Ovn_2_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 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_2_2

*Modularisera programmet All_in_Main från övn 2.1 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()                                 // Metoden Main()
   {                                                  // med
      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_2_3_Class

Modularisera programmet All_in_Main från övn 2.1 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_2_3_Test

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.

```
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_2_4_Class
Skriv en klass Fish som beskriver en fisk med datamedlemmarna sort, weight och size. Borde ligga i samma projekt som filen Ovn_2_4_Test.

```
class Fish
{
   public string sort;
   public double weight, size;
}
```

Ovn_2_4_Test
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_2_5_Class
Ta klassen Fish från övn 2.4. 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_2_5_Test

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_2_6_Test

Modifiera programmet från övn 2.5 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();                                 // Input
    if (f1.sort.Length < 6) f2.sort += '\t';
    Console.Write("\tMata in vikten till fisk1:\t");
    f1.weight = (float) Convert.ToDecimal(Console.ReadLine()); // Input
    Console.Write("\tMata in längden till fisk1:\t");
    f1.size = (float) Convert.ToDecimal(Console.ReadLine());   // Input

    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()); // Input
    Console.Write("\tMata in en till fisk2:\t");
    f2.size = (float) Convert.ToDecimal(Console.ReadLine());   // Input

    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_2_7_Class
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_2_7_Test

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.

Ovn_2_8_Class

*Skriv en klass **Rectangle** med datamedlemmarna **width, height** samt metoderna **Area()** och **Circumference()**. Deklarera datamedlemmarna en gång som **private** och en annan gång med ingen åtkomstmodifierare alls. Deklarera metoderna*

*som **public**. Förse klassen med en konstruktor och välj andra namn för konstruktorns parametrar än för datamedlemmarna.*

```
class Rectangle
{
   private int length, width;

   public Rectangle(int l, int w)             // Konstruktorn
   {
      length = l;
      width  = w;
   }

   public int Area()
   {
      return length * width;
   }

   public int Circumference()
   {
      return 2 * (length + width);
   }
}
```

Ovn_2_8_Test
*Testa din klass i en annan klass genom att i **Main()** skapa ett **Rectangle**-objekt vars datamedlemmar initieras till konstanta värden. Skriv ut dess area och omkrets.*

```
using System;
class RectangleTest
{
   static void Main()
   {
      Rectangle rekt = new Rectangle(8, 4);   // Objekt rekt av typ Rek-
                                              // tangel skapas och klas-
                                              // sens konstruktor anropas
      Console.WriteLine("\n\tArea    = " + rekt.Area()    + '\n' +
                        "\n\tOmkrets = " + rekt.Circumference()  + '\n');
   }
}
```

Ovn_2_9_Class
Modifiera klassen Rectangle från övn 2.8 genom att lägga till Get- och Set-metoder i klassen.

```
class Rectangle_new
{
   private int length, width;

   public Rectangle_new(int l, int w)
   {
      length = l;
      width  = w;
   }
```

```
   public int GetLength()                          // Get-metod för length
   {
      return length;
   }

   public void SetLength(int newLength)            // Set-metod för length
   {
      length = newLength;
   }

   public int GetWidth()                           // Get-metod för width
   {
      return width;
   }

   public void SetWidth(int newWidth)              // Set-metod för width
   {
      width = newWidth;
   }

   public int Area()
   {
      return length * width;
   }

   public int Circumference()
   {
      return 2 * (length + width);
   }
}
```

Ovn_2_9_Test

Testa den nya klassen i Main() genom att läsa in värden till datamedlemmarna. Efter utskriften av area och omkrets, fördubbla rektangelns längd och bredd med anrop av Get- och Set-metoderna. Skriv ut en gång till rektangelns area och omkrets. Med vilken faktor växer arean resp. omkretsen?

```
using System;
class Rectangle_newTest
{
   static void Main()
   {
      Console.Write("\n\tMata in längd:\t");
      int no1 = Convert.ToInt32(Console.ReadLine());

      Console.Write("\n\tMata in bredd:\t");
      int no2 = Convert.ToInt32(Console.ReadLine());

      Rectangle_new rekt = new Rectangle_new(no1, no2);

      Console.WriteLine("\nFöre fördubblingen:\n"                +
                        "\n\tArea    = " + rekt.Area()    + '\n' +
                        "\n\tOmkrets = " + rekt.Circumference()  + '\n');

      rekt.SetLength(2 * rekt.GetLength());
      rekt.SetWidth(2 * rekt.GetWidth());
```

```
      Console.WriteLine("\nEfter fördubblingen:\n"              +
                        "\n\tArea    = " + rekt.Area()    + '\n' +
                        "\n\tOmkrets = " + rekt.Circumference()  + '\n');
   }
}
```

Arean växer med faktor 4 när rektangelns sidor fördubblas, medan omkretsen växer med faktor 2, eftersom arean är en kvadratisk funktion av sidorna, medan omkretsen är en linjär funktion av dem.

Ovn_2_10_Class

Modellera en klass Cylinder som subklass till klassen Circle. Denna modellering ser Cylindern som en Circle som dessutom har en höjd. Betrakta därför Cylindern som en "utvidgad" Cirkel som ärver Circle och lägger till den en privat datamedlem height. Förse även subklassen med en konstruktor och en Get-metod. Cylindern ska dessutom ha metoderna Volume() och Surface(). Vid beräkning av Cylinderns Volume() och Surface() ska koden kunna återanvända cirkelns metoder genom att anropa dem.

```
class Cylinder : Circle                 // Cylinder ärver klassen Circle
{
   private double height;               // Den nya datamedlemmen

   public Cylinder(double radius, double height) : base(radius)
                                        // Cylinderns konstruktor
   {                                    // Anrop av Circle:s konstruktor
      this.height = height;
   }

   public double GetHeight()
   {
      return height;
   }

   public double CylinderVolume()       // Cylinderns metod för volym
   {                                    // återanvänder Circle:s metod
      return CircleArea() * height;     // för area genom att anropa den
   }

   public double CylinderSurface()      // Cylinderns metod för yta åter-
   {                                    // återanvänder Circle:s metod för
      return CircleCircumference() * (GetRadius() + height);   // omkrets
   }
}
```

Ovn_2_10_SuperClass

Förse superklassen Circle med en privat datamedlem radius, en konstruktor, en getmetod och med beräkningsmetoderna Area() och Circumference().

```
using System;
class Circle
{
   private double radius;

   public Circle(double radius)
```

```
   {
      this.radius = radius;
   }

   public double GetRadius()      // Get-metod för att exportera radius
   {                              // bl.a. till klassen Cylinder som
      return radius;              // behöver den för Surface()
   }

   public double CircleArea()
   {
      return (Math.PI * radius * radius);
   }

   public double CircleCircumference()
   {
      return (2 * Math.PI * radius);
   }
}
```

Ovn_2_10_Test
Testa dina klasser i main() genom att läsa in en cylinders radie och höjd samt skriva ut Volume() och Surface().

```
using System;
class CylinderTest
{
   static void Main()
   {
     Console.Write("\n\tMata in radie:\t");
     int no1 = Convert.ToInt32(Console.ReadLine());

     Console.Write("\n\tMata in höjd:\t");
     int no2 = Convert.ToInt32(Console.ReadLine());

     Cylinder c = new Cylinder(no1, no2); // Ett objekt skapas och
                                          // konstruktorn anropas som
                                          // initierar radius och height
     Console.WriteLine("\nEn cylinder med radien " + c.GetRadius()     +
                       " och höjden " + c.GetHeight() + " har volymen " +
                        c.CylinderVolume()  + "\n\t\t\t\t\t och ytan "  +
                        c.CylinderSurface() + "\n");
   }
}
```

Kapitel 3 Metoder i OOP, sid 149:

Ovn_3_1a
Modularisera programmet Non_modularized_1 (sid 149) för att vidareutveckla det till en liten kalkylator (fast i konsolen):Separera beräkningarna, t.ex. multiplikationen från kodens andra delar dvs från input och output.
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_3_1a
{
   static void Main()
   {
      Console.Write("\n\tMata in ett heltal:\t\t");          // Ledtext
      int no1 = Convert.ToInt32(Console.ReadLine());         // Input
      Console.Write("\n\tMata in ett heltal till:\t");
      int no2 = Convert.ToInt32(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_3_1b

Modularisera programmet Non_modularized_1 (sid 149) för att vidareutveckla det till en liten kalkylator (fast i konsolen):Separera beräkningarna, t.ex. multiplikationen från kodens andra delar dvs från input och output.
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_3_1b
{
   static void Main()
   {
      Console.Write("\n\tMata in ett heltal:\t\t");     // Ledtext
      int no1 = Convert.ToInt32(Console.ReadLine());    // Inläsning
      Console.Write("\n\tMata in ett heltal till:\t");
      int no2 = Convert.ToInt32(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_3_1b
class Multip                                          // Klassen Multip()
{
   public static int Mult(int a, int b)              // Metoden Mult()
   {
      return a * b;
   }
}
```

Ovn_3_1c

Modularisera programmet Non_modularized_1 (sid 149) för att vidareutveckla det till en liten kalkylator (fast i konsolen):Separera beräkningarna, t.ex. multiplikationen från kodens andra delar dvs från input och output. c) Flytta den nya klassen samt metoden Mult() till en separat fil.

```
using System;
class Ovn_3_1c
{
   static void Main()
   {
      Console.Write("\n\tMata in ett heltal:\t\t");
      int no1 = Convert.ToInt32(Console.ReadLine());
      Console.Write("\n\tMata in ett heltal till:\t");
      int no2 = Convert.ToInt32(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_3_1cd

Separat fil som borde ligga i samma projekt som filen Ovn_3_1c och när programmet Ovn_3_1d körs, i samma projekt som filen Ovn_3_1d

```
class Multip                                          // Klassen Multip
{
   public static int Mult(int a, int b)              // Metoden Mult()
   {
      return a * b;
   }
}
```

Ovn_3_1d

Modularisera programmet Non_modularized_1 (sid 149) för att vidareutveckla det till en liten kalkylator (fast i konsolen):Separera beräkningarna, t.ex. multiplikationen från kodens andra delar dvs från input och output. 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_3_1d
{
   static void Main()
   {
      Console.Write("\n\tMata in ett heltal:\t\t");
      int no1 = Convert.ToInt32(Console.ReadLine());
      Console.Write("\n\tMata in ett heltal till:\t");
      int no2 = Convert.ToInt32(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_3_1dA
Separat fil i samma projekt som filen Ovn_3_1d

```
class Addit                                          // Klassen Addit
{
   public static int Add(int a, int b)              // Metoden Add()
   {
      return a + b;
   }
}
```

Ovn_3_1dS
Separat fil i samma projekt som filen Ovn_3_1d

```
class Subtr                                          // Klassen Subtr
{
   public static int Sub(int a, int b)               // Metoden Sub()
   {
      return a - b;
   }
}
```

Ovn_3_1dD
Separat fil i samma projekt som filen Ovn_3_1d

```
class Div                                            // Klassen Div
{
   public static int IntDiv(int a, int b)            // Metoden IntDiv()
   {
      return a / b;
   }
}
```

Ovn_3_1dM
Separat fil i samma projekt som filen Ovn_3_1d

```
class Modu                                                  // Klassen Modu
{
   public static int Mod(int a, int b)                     // Metoden Mod()
   {
      return a % b;
   }
}
```

Ovn_3_2
Modularisera programmet Non_modularized_2 (sid 150) 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 metoden ett beskrivande namn.

```
using System;
class Ovn_3_2
{
   static void Main()
   {
      /* I n m a t n i n g */
      Console.Write("\n\tAnge antal år:\t\t");
      int years = Convert.ToInt32(Console.ReadLine());

      Console.Write("\n\tAnge antal månader:\t");
      int months = Convert.ToInt32(Console.ReadLine());

      Console.Write("\n\tAnge antal veckor:\t");
      int weeks = Convert.ToInt32(Console.ReadLine());

      Console.Write("\n\tAnge antal dagar:\t");
      int dag = Convert.ToInt32(Console.ReadLine());

      /* U t m a t n i n g */
      Console.WriteLine("\n\t"   + years + " år, "                 +
           months + " månader, " + weeks + " veckor och "          +
           dag    + " dagar är " + total(years, months, weeks, dag)
                                 + " 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_3_3a
a) Vänd om problemet från övn 3.2: Dvs Omvandla en tid som är angiven i dagar till år, månader, veckor samt resterande dagar. Skriv ett icke-modulariserat program Non_modularized_3, som frågar efter en tid i antal

dagar, läser in den, och sedan beräknar samt skriver ut resul tatet i antal år, månader, veckor samt resterande dagar.

```
// Non_modularized_3
// Omvandlar antal dagar till år, månader, veckor och restdagar
// Överlagring av operatorn / som heltalsdivision
// Modulooperatorn % (Progr1, 2, Övningar)

using System;
class Non_modularized_3
{
   static void Main()
   {
      int years, months, weeks, restDays, totalDays;

      /* I n m a t n i n g */
      Console.Write("\n\tAnge antal dagar:\t\t");
      totalDays = Convert.ToInt32(Console.ReadLine());

      /* B e a r b e t n i n g */
      years    =   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 " +
                  years + " år, " + months + " månader, " +
                  weeks + " veckor och "   + restDays + " dagar.\n");
   }
}
```

Ovn_3_3b

b) Modularisera programmet Non_modularized_3 (Ovn_3_3a) 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 Non_modularized_3. Varför är det inte lämpligt här att använda en metod med returvärde?

```
using System;
class Ovn_3_3b
{
   static void Main()
   {
      /* I n m a t n i n g */
      Console.Write("\n\tAnge antal dagar:\t");
      int totalDays = Convert.ToInt32(Console.ReadLine());
      ConvertTime(totalDays);                    // Anropet av void-metod
   }

   static void ConvertTime(int total)            // void-metod
   {
      int years, months, weeks, restDays;

      /* B e a r b e t n i n g */
```

```
      years    =   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 " + years + " 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 **4** värden som ska skrivas ut. Void-metoden beräknar OCH skriver ut dem.

Ovn_3_4_Test

Skriv först ett program med endast Main()-metoden som läser in side till en kub samt beräknar och skriver ut kubens volym side ^{3} och dess yta 6 x side ^{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 surface() 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.Surface() +
                                                                 '\n');
   }
}
```

Ovn_3_4_Class

Separat fil i samma projekt som filen Ovn_3_4_Test

```
class Cube
{
   public double side;

   public double Volume()
```

```
   {
      return side * side * side;
   }

   public double Surface()
   {
      return 6 * side * side;
   }
}
```

Ovn_3_5_Test
Modularisera programmet Non_modularized_3 (sid 253) efter eget godtycke.

```
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 = Convert.ToInt32(Console.ReadLine());

      TimeConversion t = new TimeConversion(); // Objekt skapas
      t.ConvertTime(totalDays);                // Objektets metod anropas

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

Ovn_3_5_Class
Separat fil i samma projekt som filen Ovn_3_5_Test

```
class TimeConversion
{
   public int years, months, weeks, restDays;

   public void ConvertTime(int total)
   {
      /* B e a r b e t n i n g */
      years    =   total / 365;
      months   =  (total % 365) / 30;
      weeks    = ((total % 365) % 30) / 7;
      restDays = ((total % 365) % 30) % 7;
   }
}
```

Kapitel 4 Mer om metoder, sid 187:

// Ovn_4_1.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_4_1
//           {
//              static void Main()
//              {
//                 {
//                    int t = 30;
//                 }
//                 Console.WriteLine("t = " + t);
//              }
//           }

using System;
class Ovn_4_1
{
   static void Main()
   {
      int t;
      {
         t = 30;
      }
      Console.WriteLine("\n\tt = " + t + '\n');
   }
}
/* Kompileringsfelet i programmets första variant berodde på att varia-
   beln t var definierad i ett inre block och att programmet refererade
   till den utanför det inre blocket där t inte längre var giltig.
*/
*************************************************************************

// Ovn_4_2_Test.cs
// Modularisera programmet MiniSort från kap 4 (sid 154)
// efter eget godtycke.

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_4_2_Class.cs
// Separat fil i samma projekt som filen Ovn_4_2_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
      }
   }
}
*************************************************************************
```

Kapitel 5 Tillämpning av OOP, sid 230:

```
// Ovn_5_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_5_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_5_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. Använd klasserna RandArray (sid 207) och RandomNo som externa
// moduler.

using System;
class Ovn_5_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 (sid 207)
// Separat fil i samma projekt som filen Ovn_5_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 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_5_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 (sid 219) för att
omvandla
// den inlästa strängen i en array av char.

using System;
class Ovn_5_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_5_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_5_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_5_5.cs
// Skriv ett program som frågar efter användarens för- och efternamn,
// hälsar sedan användaren i en utskrift med fullständiga namnet, för-
// namnets längd samt efternamnets första och sista bokstav. Lös upp-
// giften generellt utan att använda information om något speciellt för-
// och efternamn.
using System;
class Ovn_5_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 för- och efternamnets avskiljare
            surname0 = name[i+1]; // Hittar efternamnets första 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_5_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_5_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
      }
   }
}
*************************************************************************

// Ovn_5_7.cs
// Modifiera programmet List (sid 226) så att sorteringen av slumptalen
// görs med vår egen bubbelsorteringsmetod sort() (sid 212) istället för
// med den fördefinierade List-metoden Sort(). Testa först med array-
// notationen som sort() är skriven i. Försök sedan att skriva om sort()
// till en List-version.
using System;
using System.Collections.Generic;              // Krävs för List
class List
{
   static void Main()
   {
      List<int> intList = new List<int>();     // List-objekt av int
      Random r = new Random();
      int a = 1, b = 1000;
      Console.WriteLine(
              "\n\t100 heltal mellan " + a + " och " + b +
              " slumpas till ett List-objekt:\n");
      RandList.Rand(r, intList, a, b);         // Slump-tilldelning
      Print.Out(intList);                      // Osorterad utskrift
      Bubble.sort(intList);                    // List-sortering
      Console.WriteLine(
```

```
            "\tHeltalen sorteras med List-metoden Sort():\n");
      Print.Out(intList);                      // Sorterad utskrift
   }
}

// ---------------------------------------------------------------------

// BubbleList.cs (List-versionen av Bubble.cs sid 212)
// Separat fil i samma projekt som filen Ovn_5_7.cs
// Sorterar heltal lagrade i arrayen t med en bubbelsorteringsalgoritm
using System;
using System.Collections.Generic;
class Bubble
{
   public static void sort(List<int> t)
   {
      int temp;
      for (int pass=0; pass<t.Count-1; pass++)
          for (int i=0; i<t.Count-1; i++)
              if (t[i] > t[i+1])               // Sortering i stigande
              {                                // ordning
                 temp = t[i];                  // Algoritm för platsbyte
                 t[i] = t[i+1];                // av de två elementen
                 t[i+1] = temp;                // t[i] och t[i+1]
              }
   }
}

// ---------------------------------------------------------------------
// Print.cs (sid 228)
// Separat fil i samma projekt som filen Ovn_5_7.cs
// Metoden Out() skriver ut en lista med en foreach-sats som
// loopar igenom listans ALLA element

using System;
using System.Collections.Generic;
class Print
{
   public static void Out(List<int> t)
   {
      Console.Write("\t");
      int i = 0;
      foreach (int element in t)                // Loop
      {
         Console.Write(element + " ");
         if ((i % 14 == 0) && (i != 0))          // Radbyte var
            Console.Write("\n\t");               // 14:e utskrift
         i++;
      }
      Console.WriteLine("\n");
   }
}

// ---------------------------------------------------------------------

// RandList.cs (sid 227)
// Separat fil i samma projekt som filen Ovn_5_7.cs
// Metod Next() slumpar fram heltal mellan a och b och
// lagrar dem i ett List-objekt med List-metoden Add()
```

```
using System;
using System.Collections.Generic;
class RandList
{
   public static void Rand(Random r, List<int> no, int a, int b)
   {
      for (int i=0; i < 100; i++)             // Här fylls listan
          no.Add(r.Next(a, b));               // med slumptal
   }
}
```

****

Ovn_5_8_Test

Modifiera programmet ArrayOfRef (sid 200). ...(se klassen **Fish_priv***)*
Det modifierade programmet ska göra samma sak som det ursprungliga.

```
using System;
class ArrayOfRef_ny
{
   static void Main()
   {
      string fiskSort;
      float  fiskVikt, fiskLängd;
      Fish_priv[] f = new Fish_priv[5];            // Array av referenser

      for (int i = 0; i < f.Length; i++)
      {
          Console.Write("\n\tMata in sorten till fisk" + (i+1) + ":\t");
          fiskSort = Console.ReadLine(); // Input
          if (fiskSort.Length <= 7) fiskSort += '\t';
          Console.Write("\tMata in vikten till fisk" + (i+1) + ":\t");
          fiskVikt = (float) Convert.ToDecimal(Console.ReadLine());
          Console.Write("\tMata in längden till fisk" + (i+1) + ":\t");
          fiskLängd = (float) Convert.ToDecimal(Console.ReadLine());

          f[i] = new Fish_priv(fiskSort, fiskVikt, fiskLängd);
      }
      Console.Write("\nFisksort\tVikt i g\tLängd i cm\tPris\tFrakt\n" +
      "-----------------------------------------------------------\n");
      for (int i = 0; i < f.Length; i++)
          Console.WriteLine(f[i].toString());
   }
}
```

Ovn_5_8_Class

Deklarera klassen Fish:s datamedlemmar som private och metoderna som public. Förse klassen med en konstruktor och en strängrepresentationsmetod.

```
using System;
class Fish_priv
{
   private string sort;
   private float weight, size;

   public Fish_priv(string S, float w, float s)
```

```
   {
      sort   = S;
      weight = w;
      size   = s;
   }

   public int Price()
   {
      return (int) Math.Round(weight * 7.25f / 100);
   }

   public int Shipping()
   {
      return (int) Math.Round(weight * 0.02f + size * 0.1f);
   }

   public String AsString()
   {
      return sort    + "\t "     +
             weight  + "\t\t   " + size       + "\t\t " +
             Price() + "\t "     + Shipping() + "\n"    ;
   }
}
```

Appendix

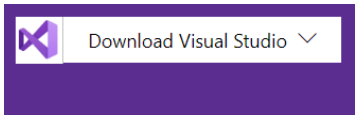
Visual Studio

Installation av Visual Studio

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

   Webbsidan Visual Studio 2019 visas. Gå med musen över knappen Download Visual Studio → Välj i dropplistan som dyker upp:

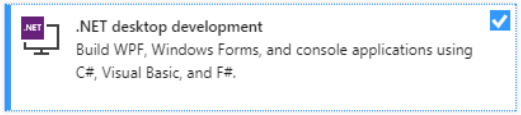
   Community 2019



2) Installationsfilen vs_community_….exe laddas ner. 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 när det dyker upp rutan Visual Studio Installer.

3) Visual Studio Installer öppnar ett stort vitt fönster 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 (3:e till vänster):

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 nedre högra hörnet av det stora fönstret på knappen **Install**. Detta kan ta ett tag – beroende på din dators prestation.

6) Visual Studio 2019 är en gratis programvara vars licens är tidsbegränsad. Du behöver skapa ett Microsoft-konto med din e-mailadress som användarnamn. När du gör det glöm inte att anteckna och spara ditt lösenord. Du kommer att behöva det när du efter ett tag måste uppdatera licensen. Följ instruktionerna som kan involvera verifiering via din e-mailadress. Det är gratis, går fort och medför inga komplikationer.

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

8) När du lyckats med installationen startas Visual Studio antingen automatiskt eller du kan göra det själv. Stäng rutan Visual Studio Installer.

9) Beroende på vilken typ av applikation du vill skapa fortsätt enligt instruktionerna på sid 268 för Console Application eller sid 273 för Windows Forms Application.

Konfiguration och användning

av Visual Studio

Efter lyckad installation av Visual Studio enligt anvisningarna i förra avsnitt kan du här läsa nu hur man *använder* programvaran. För att kunna göra det 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. Vi vill i denna beskrivning hålla oss till det absolut minimala vad gäller miljön för att kunna koncentrera oss på själva *språket C#*. Beskrivningens viktigaste moment är:

- Att välja rätt typ av applikation
- Att skapa ett *projekt*
- Att lägga till en C#-källkodsfil till projektet
- Att kompilera och exekvera C#-koden i projektet

Det finns olika typer av C#-program. Ett annat ord för program är applikation.

Två olika typer av applikation

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

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 boken *Programmering 1 med C#* 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. Windows Forms Applications introduceras i denna bok på sid 13.

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.

Console Application

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

Start → Visual Studio 2019

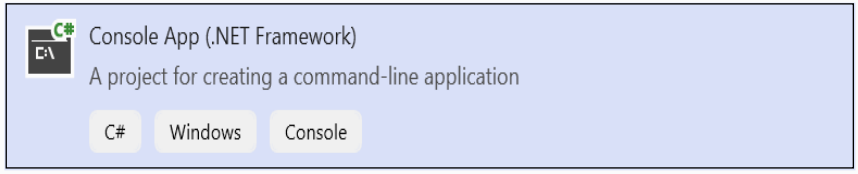
Ett vitt fönster öppnas med rubriken Visual Studio 2019. I kolumnen till höger under rubriken Get started finns ett antal rutor.

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

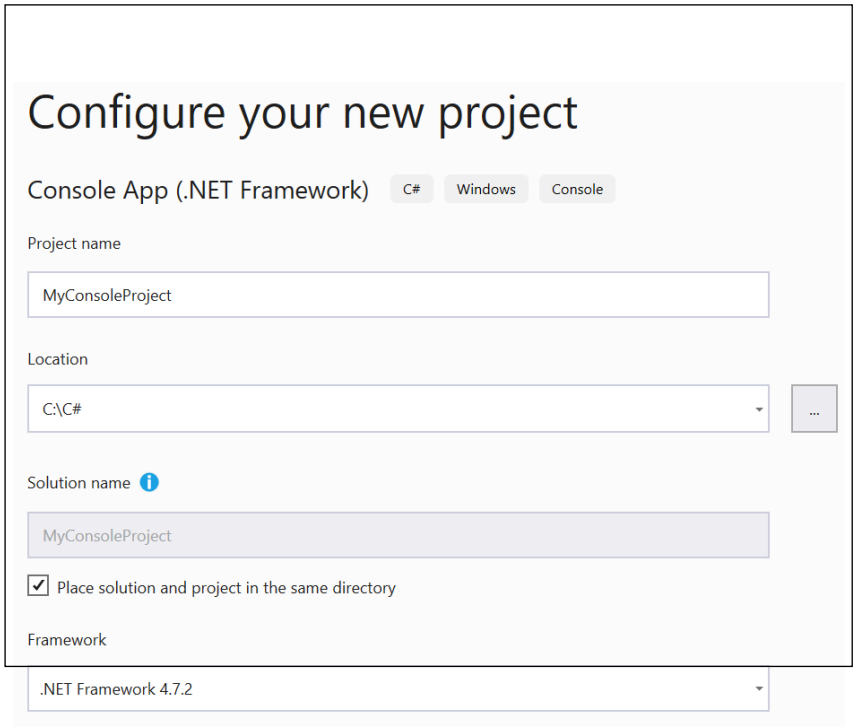
   Create a new project

   En ny dialogruta dyker upp med rubriken Create a new project. Scrolla ned den (på höger sidan) och leta efter rutan som ser ut så här och har rubriken Console App (.NET Framework):

   Markera rutan ovan. Klicka i dialogrutan Create a new project som omfattar denna ruta, på knappen Next längst ned till höger.

   En ny dialogruta dyker upp med rubriken Configure your new project. Fyll i den uppgifterna enligt följande:



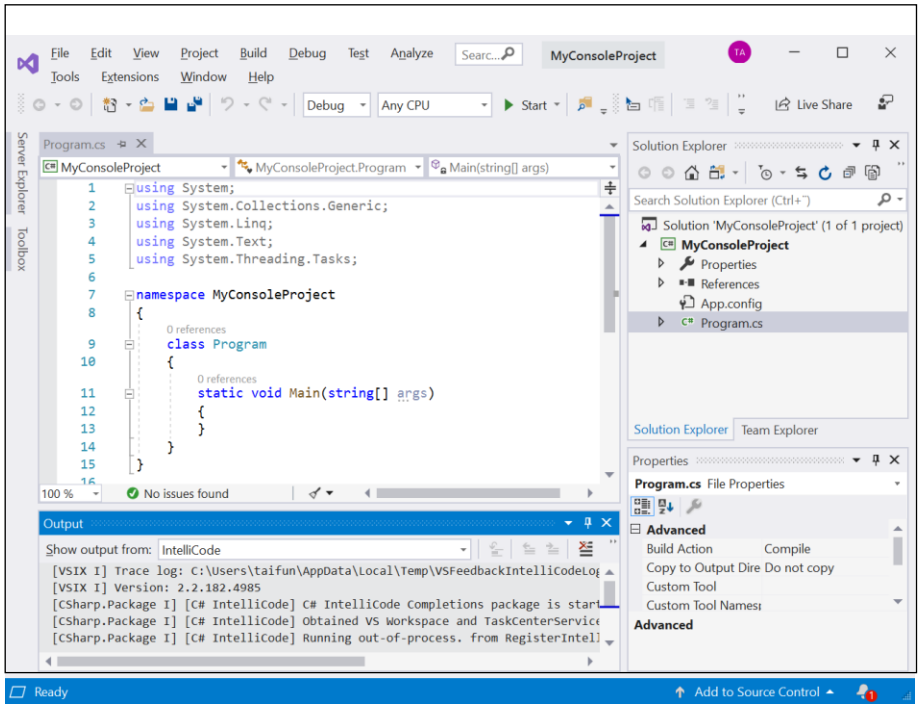
I den övre delen av dialogrutan döper vi vårt projekt till MyConsoleProject. I textrutan Location anger vi den fullständiga sökvägen till den mapp vi vill placera vårt projekt i. Låt oss säga vi vill samla våra C#-program i en mapp som vi kallar C# i enheten C:\ på vår dator. I så fall anger vi som Location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

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.

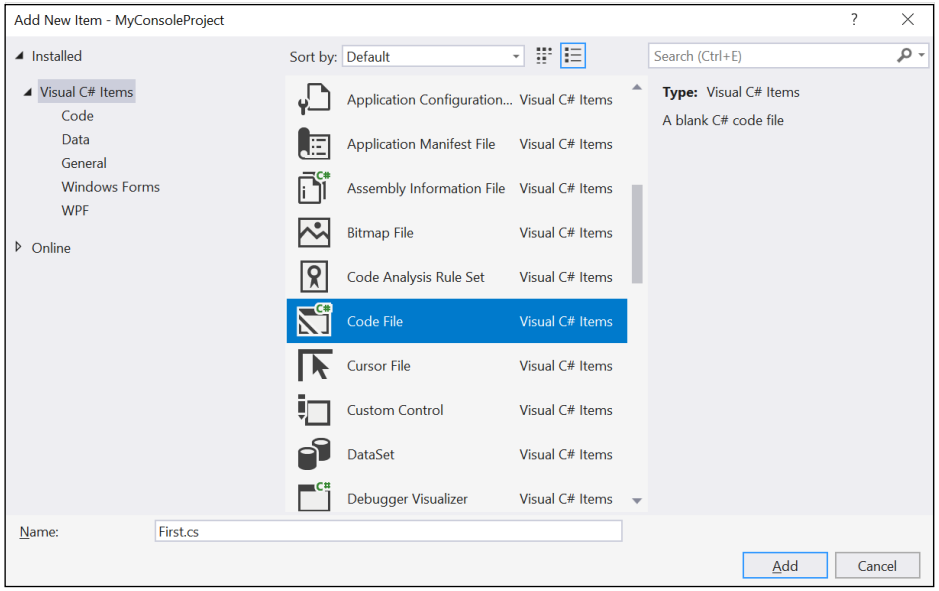
Därmed har vi avlägsnat denna fil från projektet för att kunna infoga vårt eget C#-program i projektet. Det finns två alternativ att göra det: Antingen vill vi skapa ett helt nytt program, skriva in koden, spara den i en fil och infoga den i projektet eller vi vill lägga till en redan befintlig fil som 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 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 t.ex. *Progr1+, 4.1*. 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 270.

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 271 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 270-271. *Ett* projekt räcker för alla konsolapplikationer. Så det var värt mödan.

Windows Forms Application

En fullständig genomgång av denna typ av applikation finns utförligt på sid 13-17 där bokens första Windowsapplikation Interaction behandlas i detalj. Vi hänvisar till den.

Programförteckning

Kapitel 3 Metoder i OOP

Kapitel 4 Mer om metoder

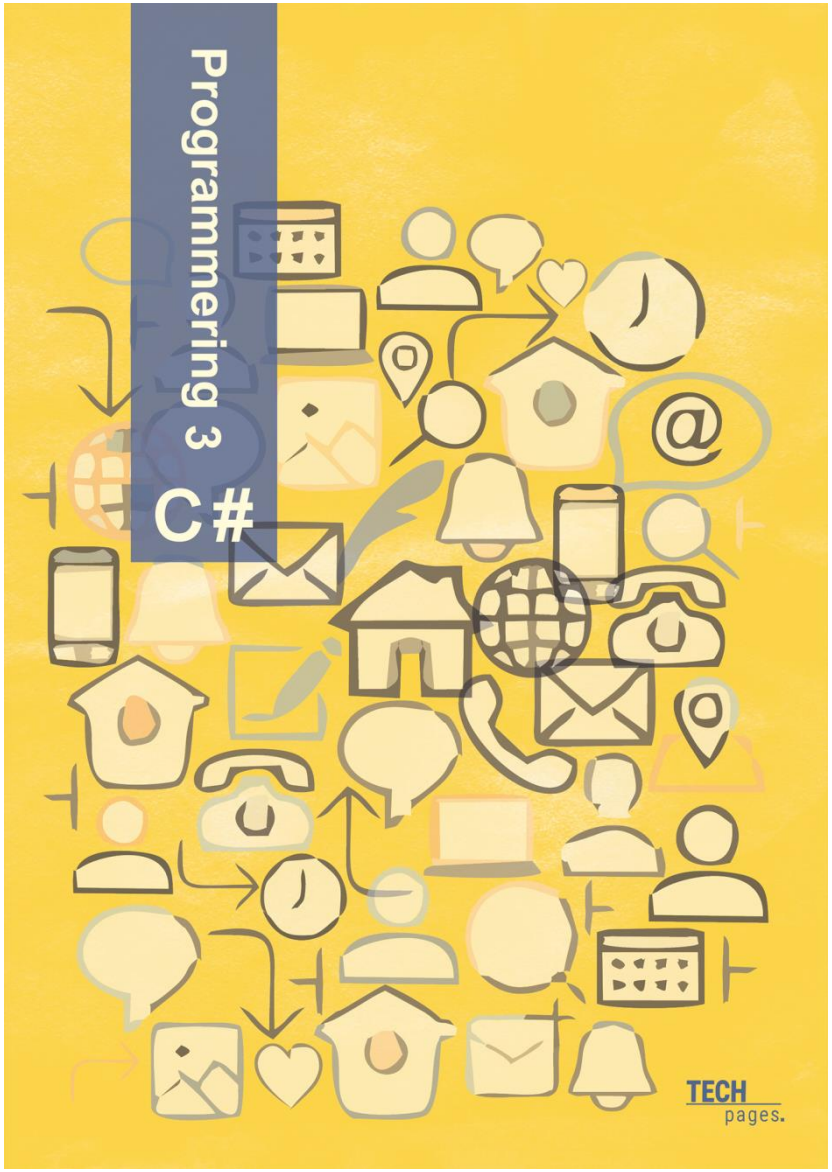
Kapitel 5 Tillämpning av OOP

Register

V,W

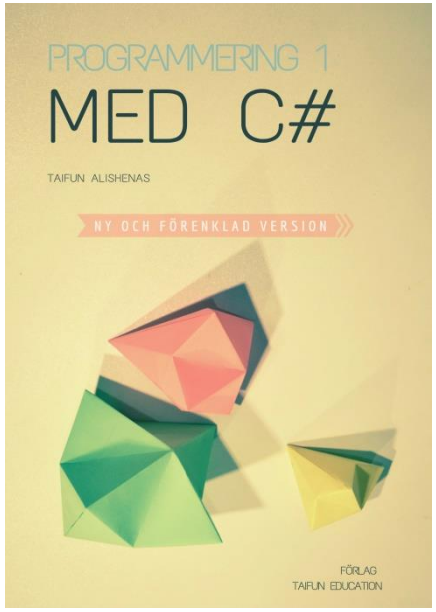
Å

Ö



Programmering 3 med C# är en fortsättning på denna bok och behandlar programmeringens mer avancerade koncept samt tillämpningar, bl.a. filhantering och databaser, speciellt relationsdatabaser. Databasers kommunikationsspråk SQL introduceras som ett inbäddat språk i C#.

Programmering 1 med C#



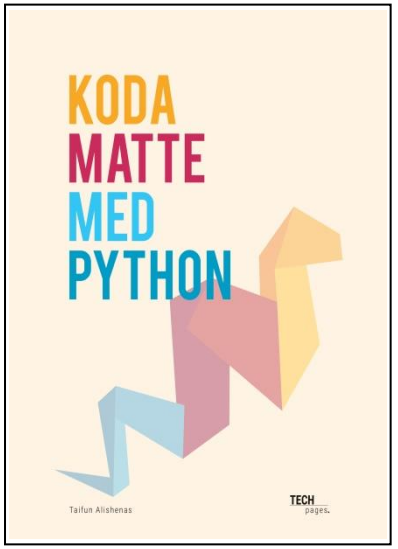
Ur innehållet:

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

Koda matte med Python

Programmering i matematik

En enkel, pedagogisk lärobok som kompletterar matematikundervisningen med inslag av programmering. Den vägleder både lärare och elever genom att kombinera teori med praktiska övningar och fullständiga lösningar. Boken presenterar ett pedagogiskt koncept om hur programmering kan integreras i kurserna Matematik åk 7-9 och Matematik 1 (a,b,c). Ett övningshäfte för elever med lektionsupplägg planeras till vårterminen 2020.



www.kodamatte.se

Programmering 2 med C#

Ur innehållet

Windowsprogrammering
Grafiskt gränssnitt mot Internet
Egen webbläsare
Grafiskt gränssnitt med menyval
Multiple Document Interface
Objektorienterad programmering
Obj. modellering & implementation
Metoder i OOP
Arv och polymorfism
Lambdauttryck
Delegater
LINQ
Abstrakta klasser & metoder
Sökning och sortering
Kryptering med slumptal
Rekursion
Generiska metoder
2D Array
Virtuella metoder
Metodgrupper
Visual Studios C# miljö
Windows Forms Applications
Övningar & projektuppgifter
Fullständiga lösningar till övningar

Utveckla en egen webbläsare (ex. ur boken, sid 45-54):