Objektorienterad programmering och design

Med övningar,
fullständiga lösningar
&
projektuppgifter

www.techpages.se TechPages Förlag

Innehåll

Ämne	Sida	Program
Kapitel 1 Objektorienterad programmering (OOP)	**5**	
1.1 Vad är objektorienterad programmering?	6	
- Objekt, klass, datamedlem och metod	6	
- OOP:s tre hörnstenar	9	
- Klassdiagram	11	
1.2 Klassbegreppet	13	
- Definition av klass	13	
- Två klasser som ett program	14	**Password**
- Klass 2 skapar objekt av klass 1	16	**PasswordUse**
1.3 Modulariseringsprocessen	18	**All_in_Main**
- Modularisering på metodnivå	19	**Method**
- Modularisering på klassnivå	20	**Obj_Oriented**
1.4 Användning av klasser	22	
- Deklaration av en klass	22	**Emp**
- Definition av ett objekt	24	**EmpTest**
- Åtkomst till objektets medlemmar	26	
1.5 Klassens konstruktor	28	
- Åtkomstmodifieraren **private**	28	**Circle**
- Konstruktorns egenskaper	30	**Encapsulation**
- Default konstruktorn	33	**AccountD**
- Flera konstruktorer	34	**AccountDTest**
1.6 Referensvariabler	37	
- Datamedlemmarnas defaultvärden	38	
- Referensen null och nolltecknet '\0'	39/40	
- Anonyma objekt	40	
1.7 Komposition	41	**Date/Employ**
- Komposition av klasser och objekt	43	**Composition**
1.8 Arv	45	**Person**
- Arvrelationen	47	**Employee**
	48	**Inheritance**
1.9 Polymorfism	50	**Account**
- Överskuggning av metoder	52	**MinAccount**
- Åtkomstmodifieraren protected	53	**PolymorphTest**
Övningar till kapitel 1	57	
Kapitel 2 Objektorienterad design	**63**	
2.1 Modellering med UML	64	
- Projekt Lönespecifikation	64	
- Kundens kravspecifikation	64	

Ämne	Sida	Program
- UML design och modellering i fyra steg	64	
2.2 Implementation av modellen	68	
- Klassen Time	68	**`Time`**
- Klassen Employee	69	**`Employee`**
- Array av referenser	70	**`EmploTest`**
- Skrivning till fil	73	**`WriteFile`**
Övningar till kapitel 2	74	
Kapitel 3 Metoder i OOP	**75**	
3.1 Accessmetoder	76	**`Empl/GetSet`**
3.2 Property	79	**`Property`**
3.3 Statiska datamedlemmar och metoder	82	**`StatDemo`**
- Klass- och instansvariabler	82	**`StatDemoTest`**
- Allokeringsmodifieraren **`static`**	84	**`RandTest`**
3.4 Överlagring av metoder	88	**`Overload`**
3.5 Variablers livslängd	92	**`Block`**
3.6 Överskuggning av variabler	96	**`OverrideVar`**
- Referensen **`this`**	97	
3.7 Lambdauttryck	102	**`Lambda`**
- Vad är LINQ ?	109	**`CountLINQ`**
3.8 Delegater	105	**`Delegate`**
- Delegat som parameter i metoder	106	**`DelegParam`**
- Varianter av **`Console.WriteLine()`**	108	**`WriteLOverl`**
- Lösningen med LINQ	109	**`CountLINQ`**
- Metodgrupper	110	**`MethodGroup`**
Övningar till kapitel 3	112	
Kapitel 4 Tillämpningar	**115**	
4.1 Tvådimensionella arrays	116	**`DoubleArray`**
4.2 Tabellhantering i filer	121	
- 2D array som parameter i metoder	122	**`SetTable`**
- Att skriva tabeller till filer	125	**`WriteTable`**
- Att uppdatera tabeller i filer	127	**`UpdateTable`**
4.3 Primtalsfaktorisering	129	**`Prime`**
- Rekursiv algoritm för primtalsfaktorisering	131	**`PrimeFactors`**
4.4 Rekursion	135	**`Fibonacci`**
- Collatz algoritmen, rekursiv	138	**`Collatz_reku`**
Övningar till kapitel 4	141	
Kapitel 5 Windowsprogrammering	**142**	
5.1 Interaktiva grafiska gränssnitt	143	**`Interaction`**

Ämne	Sida	Program
- Controls	144	
- Windows Forms Applications	144	
- Händelsemetoder	148	
5.2 TextBoxar, Buttons & Labels	150	**PassWTextBox**
5.3 Checkboxar och radioknappar	152	**Bartender**
5.4 En egen webbläsare	156	
- En första webbläsare	158	**FirstBrowser**
5.5 En mer utvecklad webbläsare	160	**DevBrowser**
- Dialogrutan Navigate	161	
Övningar till kapitel 5	167	
Fullständiga lösningar till alla övningar (Facit)	**169**	

Kapitel 1

Objektorienterad Programmering

Ämne	Sida	Program
1.1 Vad är objektorienterad programmering?	6	
- Objekt, klass, datamedlem och metod	6	
- OOP:s tre hörnstenar	9	
- Klassdiagram	11	
1.2 Klassbegreppet	13	
- Definition av klass	13	
- Två klasser som ett program	14	**Password**
- Klass 2 skapar objekt av klass 1	16	**PasswordUse**
1.3 Modulariseringsprocessen	18	**All_in_Main**
- Modularisering på metodnivå	19	**Method**
- Modularisering på klassnivå	20	**Obj_Oriented**
1.4 Användning av klasser	22	
- Deklaration av en klass	22	**Emp**
- Definition av ett objekt	24	**EmpTest**
- Åtkomst till objektets medlemmar	26	
1.5 Klassens konstruktor	28	
- Åtkomstmodifieraren **private**	28	**Circle**
- Konstruktorns egenskaper	30	**Encapsulation**
- Default konstruktorn	33	**AccountD**
- Flera konstruktorer	34	**AccountDTest**
1.6 Referensvariabler	37	
- Datamedlemmarnas defaultvärden	38	
- Referensen null och nolltecknet '\0'	39/40	
- Anonyma objekt	40	
1.7 Komposition	41	**Date/Employ**
- Komposition av klasser och objekt	43	**Composition**
1.8 Arv	45	**Person**
- Arvrelationen	47	**Employee**
1.9 Polymorfism	50	**Account**
- Överskuggning av metoder	52	**MinAccount**
- Åtkomstmodifieraren protected	53	**PolymorphTest**
Övningar till kapitel 1	57	

1.1 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.

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.

Objekt, klass, datamedlem och metod

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.

Ett *objekt*, t.ex. en bil, har vissa egenskaper. Man kan t.o.m. säga att bilen är summan av alla sina egenskaper. Ett annat ord för egenskap är *attribut*. Summan av alla attribut utgör objektet. Bilen har som attribut: fabrikat, modell, färg, årsmodell, antal

körda mil, antal hästkrafter, maximala hastigheten, antal och storlek på cylindrar i motorn osv. Alla dessa data utgör objektet bil och ger svar på frågan ”Vad är det för bil?”. Alla bilar har sådana attribut. Därför abstraherar man – dvs bortser från bilarnas olikheter – och samlar bilarnas gemensamma egenskaper (attribut) i något som man kallar för *klassen Bil*. När man programmerar, deklarerar man klassen *Bil* och skriver upp alla dessa bilattribut som klassens *datamedlemmar*.

Metoder

Men bilden vore ofullständig om vi nöjde oss med dessa intressanta, men statiska datamedlemmar. Vi vill också veta vad man kan *göra* med bilen. Ett objekt med alla sina attribut kan i regel även utföra vissa aktioner eller operationer. I den objektorienterade programmeringens terminologi kallas dessa aktioner 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 vore alltså att ange *både* dess attribut *och* metoder. Bilfabrikanten måste förse bilen med alla dessa färdigheter för att kunna sälja den som en bil. Därför går man i bilfabriken efter en plan när man tillverkar bilen. Denna plan för konstruktion av bilen är *klassen Bil*. Konstruktörerna, mest ingenjörer, måste skapa denna plan, innan bilen kan byggas. 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.

En *metod* är en funktionalitet som definieras i en klass. Den talar om vad ett objekt av denna klass kan *göra*. Det finns två steg i hantering av metoder: Först definierar man dem dvs skapar man deras kod i en klass. Sedan *anropar* dvs aktiverar man dem i ett objekt av denna klass. Ofta är det första steget redan genomfört av andra, så vi behöver bara anropa en redan *fördefinierad* metod. I klassen *Bil* t.ex. är metoderna att köra fram, att backa, att accelerera, att bromsa osv. definierade i huvuden på bilkonstruktörerna och i deras konstruktionsritningar och dokumentationer. Sedan har man tillverkat massor med objekt av klassen Bil i fabriken och byggt in dessa metoder i alla bilar. Vi behöver bara anropa dem i den bil vi kör. Den bil vi kör är ett specifikt objekt av klassen Bil. Låt oss kalla det för **`minVolvo`**. Objektet **`minVolvo`** har ett antal attribut som t.ex. fabrikat, modell, färg, årsmodell osv., men också ett antal metoder, bl.a. metoden **`Kör()`**. Parenteserna i metodens namn brukar man skriva för att karakterisera **`Kör()`** som en *metod* och skilja den från klassens attribut. I C# skriver man ett anrop av metoden **`Kör()`** så här:

```
minVolvo.Kör();
```

Observera att *före* punkten står ett objekt, inte klassen. Det är ju den specifika bil som jag använder just nu som ska köras. Först *efter* punkten står själva anropet av metoden **`kör()`**. Det här sättet att skriva kallas *punktnotation*. Metoder måste alltid anropas med punktnotation, vilket har sin grund i att de endast är deklarerade i klasser, så att de endast existerar i objekt av en klass. Till skillnad från fristående

funktioner kan metoder varken definieras utanför klasser eller anropas utanför objekt. I C# finns endast metoder, inga funktioner. Om vi bortser från bilexemplet kan det i andra sammanhang även förekomma en klass (istället för objekt) före punkten i anropet av en metod. I så fall är metoden definierad i klassen på ett speciellt sätt nämligen som en *statisk* metod, vilket tas upp senare när vi behandlar metoder i detalj.

En annan variant av metoden **`Kör()`** kan anropas på 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.

Abstraktion

Det är avgörande att skilja mellan *objekt* och *klass*. Vi tar ett annnat exempel: Pepparkakor är objekt vars klass är pepparkaks*formen*. 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 pepparkaksformen 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 – samma abstraktionsprocess som vi kunde observera hos bilar.

Att *bortse* från skillnader och att bibehålla det gemensamma hos olika verkliga objekt, kallas för *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 *begreppsbildning*, till *klassificering* eller *kategorisering* 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. Därför kallas även en klass för en *abstrakt datatyp* i programmering.

OOP:s tre hörnstenar

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

Vi ska nu få en första inblick i dessa begrepp utan att skriva kod. För att förstå dem bättre behöver vi sedan i alla fall skriva kod och på så sätt få mer detaljerade kunskaper om objektorientering.

Inkapsling och klassens konstruktor

Att låta klassens datamedlemmar vara *privata* och inte åtkomliga från andra klasser kallas för *inkapsling*. Detta gör man bl.a. ur datasäkerhetssynpunkt – den första av tre hörnstenar i objektorienterad programmering. Ändå måste programmeraren kunna komma åt dem för att läsa, ändra och hantera dem i koden. För att kunna göra det måste programmeraren använda sig av ett verktyg som kallas för klassens *konstruktor*. Konstruktorn en speciell metod vars namn är identiskt med klassens namn. Den initierar automatiskt klassens privata datamedlemmar när ett objekt skapas. Konstruktorn är ett programmeringstekniskt koncept för att realisera inkapsling och kommer att behandlas i detalj senare.

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!

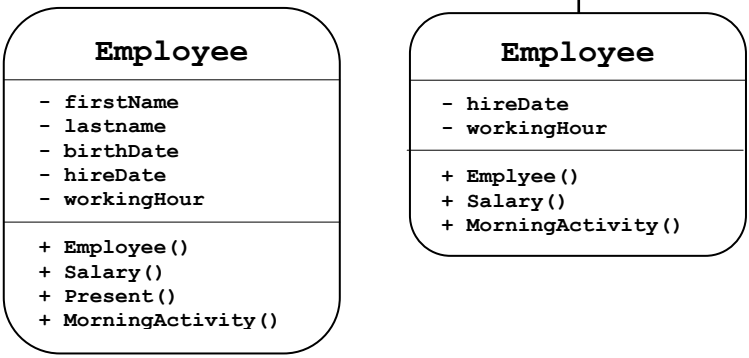
Anledningen till att de har samma namn är följande: För det första blir det ingen namnkonflikt därför att de tillhör olika typer av objekt. De är inte fristående utan inkapslade i var sitt objekt som skiljer åt dem. För det andra ska vi inte i onödan göra utvecklingsarbetet komplicerat genom att hitta 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.

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*). 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.

1.2 Klassbegreppet

En klass som ett C# program:

```
using System;

class MyFirst
{
    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. Klassens centrala roll framgår av följande definition för 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 **MyFirst** 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 (en sanning med modifikationer, se sid 102).

Definition av klass

Här följer *en* definition på *klass*. Läs på sid 22 en *annan* definition.

> En klass är kod som på ett generellt och modulärt sätt beskriver en kategori av verkliga eller virtuella saker och ting. Den består av *data-medlemmar* 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 18).

Två klasser som ett C# program

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 returne-
// rar ett sanningsvärde: true eller false

using System;

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

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 deklareras

Namnet **`Password`** är beskrivande för det som programmet är tänkt för. Här följer vi både de vanliga namngivningsreglerna för identifierare som gäller i C# 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 skapar objekt av klass Password

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 fortsätter 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?

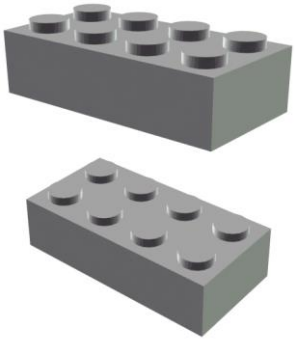
Slutligen kan man undra om det hade varit möjligt resp. rimligt att lagra båda klasser **Password** och **PasswordUse** i en och samma fil. Svaret är: Möjligt ja, men inte rimligt ur den objektorienterade programmeringens synpunkt. Därför att det går både att kompilera och köra programmet när båda klasser 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*.

1.3 Modulariseringsprocessen

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. Principen heter *modularisering* och kan användas vid både modellering och problemlösning. Ett stort komplext problem bryts ned i mindre *moduler* – motsvarande Lego-bitarna – och bearbetas en i taget. I 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 enklare. 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:

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

class 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 14). I nästa steg ska vi modularisera programmet genom att lyfta en del av det 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

```
// Method.cs
// Verifierar lösenord inmatat i versaler eller gemener
// Modulariserad på metodnivå: Inte objektorienterad
using System;
class Method
{
   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 **Method** 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 Ok() som 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 14 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 **PasswordUse** på sid 14:

```
// Obj_Oriented.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 ETT program bestående av 2 klasser
using System;

class Obj_Oriented
{
   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 19) till klassnivå (sid 20) 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 **`Obj_Oriented`** (sid 21) 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 **`Obj_Oriented`**.

Anledningen varför metoden **`Ok()`** i sin definition i klassen **`Method`** (sid 19) 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 **`Method`** direkt efter deklarationen av variabeln **`input`**. Testa gärna!

1.4 Användning av klasser

På sid 13 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
using System;

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

   public String Email()                    // Metoden Email()
   {
     return (firstName[0] + lastname).ToLower() +
                                                "@domain.se";
   }

   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 använda arraynotation för att ta 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[0] + lastname).ToLower() + "@domain.se";
```

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.

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 23) som är sammansatt av klassen **String** och datatyperna **int** och **float**. I programmet **EmpTest** (sid 24) 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 objekt av klassen Emp
// Tilldelar objektets datamedlemmar värden & 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 anonymt objekt utan referens och kan därför inte heller referera till det efteråt, se sid 40.

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 23) 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 24). 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.

1.5 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 45/50). 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 datamedlemmar 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 gene-

rell – 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 28). 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 konstruk
// torn 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 30) 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 28). 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 28).
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`**:

```
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 simulerad default konstruktor.

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 89). 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:

```
// AccountDTest.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 ob-
// jektet Garbage collector dödar autom.det orefererade obj.
using System;

class AccountDTest
{
   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 39.

```
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 **`AccountDTest`** 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.

1.6 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 24) där **a** deklareras till **Emp** som är en klass: `Emp a;` Med satsen `Emp a;` skapas inget objekt utan endast en referensvariabel. Självaste objektet skapas med koden `new Emp()`.

Den allmänna formen *hur* referensvariabler kopplas till objekt, ser ut så här:

```
Klass referensvariabel = new Klass();
```

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

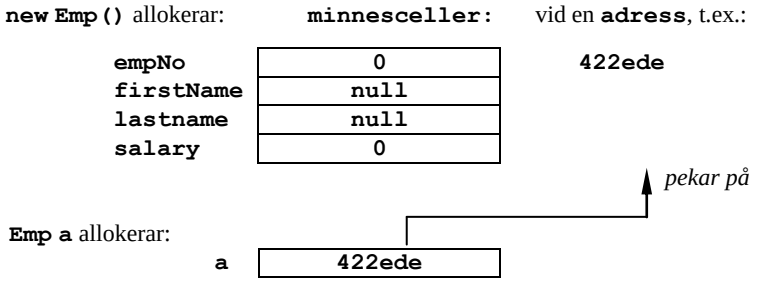
new kallas för *minnesallokeringsoperatorn* och allokerar (reserverar) minnesutrymme för **Emp**-objektet. Storleken bestäms av klassen **Emp**. Minnesadressen tilldelas referensvariabeln **a** som deklareras i samma sats. Operatorn **new** följs *inte* av parenteser. Klasserna på tilldelningens båda sidor måste vara identiska: **new Emp()** allokerar minne för lagring av ett **Emp**-objekt och returnerar minnets adress till referensen **a** vars datatyp är klassen **Emp**. Tilldelning till en referens av en annan klass skulle ge kompileringsfel.

Referensvariabler är ett verktyg för att kunna komma åt objekt. De ersätter objektens namn. 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 är länkar till webbsidor eller namnskyltar: Den lilla skylten *Gamla Stan* (referens) pekar på den stora ön *Gamla Stan* (objekt). En referens lagrar den ”lilla” minnesadressen till ett stort objekt och tar jämfört med det tunga objektet så litet minne som en vanlig **int**: 4 bytes. Detta innebär minnesekonomiskt en stor effektivitet vid exekvering av minneskrävande program.

Vid vanliga variabler av enkel datatyp hänvisar vi till minnescellerna med *variabelnamn*. Vid 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. Det är exakt samma sak som C++ gör med *pekare*.

Referensen ”pekar” på objektet

Vad händer exakt när satsen **`Emp a = new Emp();`** exekveras i programmet **`EmpTest`**? För det första deklareras 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 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 ovan 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.

Datamedlemmarnas defaultvärden

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 *default konstruktor* skapas alltid automatiskt med när man deklarerar en klass utan att själv skriva en konstruktor. Default konstruktorn initierar datamedlemmarna, när ett objekt skapas, automatiskt till följande defaultvärden:

`0`	om de är tal,
`null`	om de är referenser,
`'\0'`	om de är av typ **`char`**,
`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 initialvärde. Källan till datatypinformationen är förstås klassen **Emp** där dessa datamedlemmar är deklarerade till sina resp. datatyper (sid 23). Alla objekt är ju kopior av klassen. Vi hittar i klassen **Emp** deklarationen **String firstName;** men hävdar ändå att datamedlemmen **firstName** inte är en sträng utan en *referens* till strängar. Anledningen är att **String** är en *klass* och inte en enkel datatyp. Variabler vars datatyper är klasser är referensvariabler. Därför är **firstName** en referens och inte en sträng. Man kan bevisa det genom att t.ex. lägga in följande rader i programmet **EmpTest** (sid 24) mellan objektets definition **a = new Emp();** och tilldelningen av nya värden till objektets datamedlemmar (som börjar med **a.empNo = 123;**) dvs när datamedlemmarna fortfarande har defaultvärden:

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

Man kommer att få **null** och inte **tom sträng**, därför att det är **null** som är defaultvärdet för referenser, vilket visar att **a.firstName** är en referens och inte en 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. **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.

> **null** är i C# referensvariablernas defaultvärde med ASCII-koden 0. **null** är av referenstyp.
> En referens med värdet **null** pekar på inget objekt.

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#-pro-

gram 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 `'\0'`

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 **`'\0'`** är i C# **`char`**-variablernas defaultvärde med ASCII-koden 0.
> Nolltecknets datatyp är **`char`**.

Anonyma objekt

Man kan även skapa s.k. *anonyma objekt*, dvs objekt utan referens. Anonyma objekt skapas direkt när man behöver dem. T.ex. om metoden **`AsString()`** är definierad i klassen **`Emp`** kan man anropa den så här: **`new Emp().AsString()`**. Då är **`new Emp()`** ett *anonymt* objekt av klassen **`Emp`**, därför att det skrivs utan referens (namn). Nackdelen med anonyma objekt är att man inte längre kan komma åt dem efteråt. Testa gärna själv: Skriv i programmet **`EmpTest`** (sid 24) **`new Emp().AsString()`** istället för **`a.AsString()`** där **`a`** är en referens till ett **`Emp`**-objekt.

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 28), närmare bestämt default konstruktorn (sid 33). Allt detta under förutsättningen att man inte har definierat en egen konstruktor som i så fall skulle slå ut default konstruktorn. Har man däremot definierat en egen konstruktor måste parerentesen matcha den egna konstruktorns parameterlista.

1.7 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 (överlag
// ring): 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 konkatenerar alla tre datamedlemmar och representerar datum som en sträng, dessutom i svenskt datumformat.

Nu, när vi har klassen **`Date`**, 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 (kompo
// neras) 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.

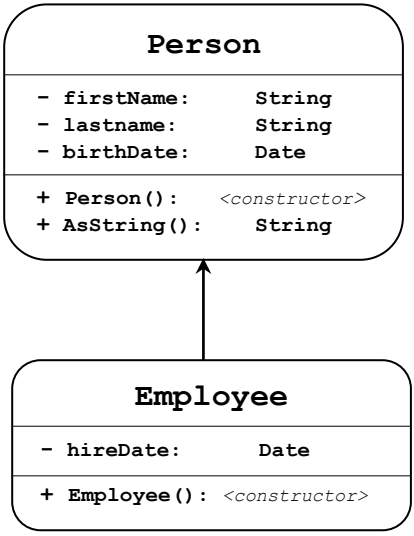
I nästa avsnitt kommer vi att vidareutveckla exemplet med anställda genom att flytta en del av koden från klassen **Employ** till en överordnad klass och etablera en arvrelation mellan dem. Komposition med klassen **Date** bibehålls så att resultatet blir en kombination av arv och komposition.

1.8 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 datamedlemmar 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 42) 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 42). 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 konstruktor 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 av base()
                                  // Konstruktorn ärver och an-
                                  // ropar superklassens kon-
                                  // struktor med:
                                  // 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()
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 43). Enda skillnaden är att klassen **Employ** (sid 42) 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.

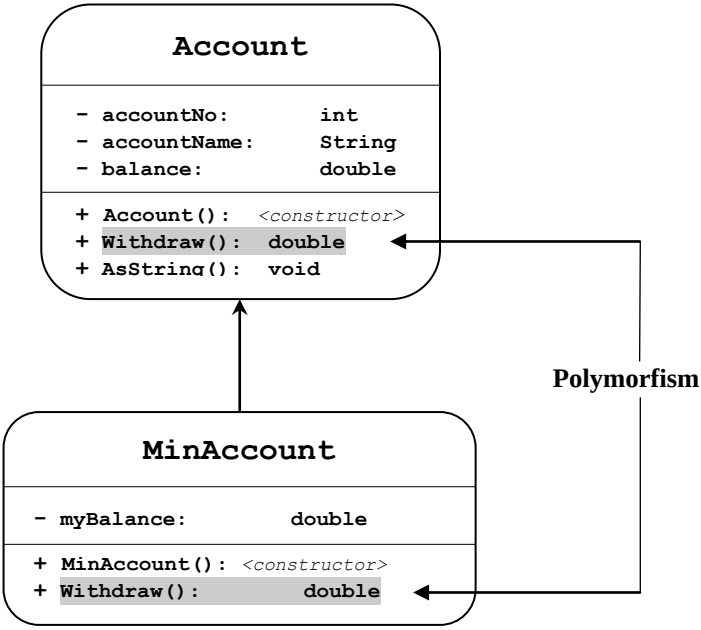
Att kompilatorn går ”upp” i klasshierarkin kan man ta som en analogi till att pilen i klassdiagrammet är riktad *uppåt* från subklassen till superklassen.

Programmet **Inheritance** producerar följande utskrift:

```
Anders Larsson är född 1978-6-16
och har jobbat sedan 2001-3-12
```

1.9 Polymorfism

I följande exempel ärver klassen **MinAccount** 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 **MinAccount**. Utan arvrelation skulle detta varit ett exempel på vanlig *överlagring av metoder* (sid 88). Men eftersom **MinAccount** ”är” ett speciellt **Account** kan **MinAccount** 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 tas upp senare (sid 88) innebär samma namn för olika metoder. 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 **MinAccount** som ett **Account** med en **Withdraw()**-metod som implementerar denna affärslogik.

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

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

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

   public String Withdraw(double amount) // Definierar om su-
   {                                     // perklassen Kontos
      if (balance - amount < myBalance)  // metod: Inte < 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";
      }
   }
}
```

MinAccount 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 **MinAccount** 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 kon-

cept 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 **`MinAccount`** 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 **`MinAccount`**, vilket underlättar programmeringen. Subklassen **`MinAccount`** 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 **MinAccount**-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 MinAccount-objekt (pelle)
using System;

class PolymorphTest
{
   static void Main()
   {
      Account kalle = new Account(12345, "Kalle", 200);
      MinAccount pelle =
                     new MinAccount(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
                                  // superklassens 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
                                    // subklassens 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 **MinAccount**. De initieras med var sin konstruktor. **kalle** får **200** kr insatt på sitt konto, **pelle 100**. Eftersom **pelle** har ett **MinAccount** 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 **MinAccount**. Så, **pelle** får maximalt ta ut 50 kr från sitt konto. Försöker han ta ut t.ex. **100** kr – vilket vi gör i körexemplet 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 **MinAccount** 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 1

1.1 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`**.

1.2 Modularisera programmet **`All_in_Main`** från övn 1.1 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`** f ör radien. Då ska arean bli π pga π **`r`**^{2} = π och omkretsen bli 2π pga 2π **`r`** = 2π.

1.3 Modularisera programmet **`All_in_Main`** från övn 1.1 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.

1.4 ***Triangeln (Inlämningsuppgift)***

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 re-

ferensen. 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?

1.5 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.

1.6 Modifiera klassen **`Rectangle`** från övn 1.5 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?

1.7 Kaffeautomaten (Projektuppgift)

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. **`Change_in_coins()`**[*] är en

[*] 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.

metod som ska 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:

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.

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 **`%`**. ”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.

1.8 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.

1.9 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 2

Objektorienterad design

Ämne	Sida	Program
2.1 Modellering med UML	64	
- Projekt Lönespecifikation	64	
- Kundens kravspecifikation	64	
- UML design och modellering i fyra steg	64	
2.2 Implementation av modellen	68	
- Klassen Time	68	**`Time`**
- Klassen Employee	69	**`Employee`**
- Array av referenser	70	**`EmploTest`**
- Skrivning till fil	73	**`WriteFile`**
Övningar till kapitel 2	74	

2.1 Modellering med UML

Nu ska vi använda det vi lärt oss om den objektorienterade programmeringens grundläggande principer på en verklig situation. Vi vill diskutera design- och modelleringsfrågor som, speciellt i större projekt, måste besvaras innan man börjar koda. Ska det slutliga programmet vara objektorienterat måste redan *modellen* vara det. Koden måste baseras på en objektorienterad modell som fundament. Det hela går ut på att bygga en *modell* av en verklig situation, i enlighet med den objektorienterade synen på *program* som nämnts tidigare:

Program = Modell av verkligheten

Vi vill realisera denna syn genom att genomföra följande praktiskt uppdrag som ställs till oss av en kund. Så här formulerar kunden sin kravspecifikation:

Projekt Lönespecifikation

”Vi vill datorisera lönespecifikationen till våra timanställda. Varje vecka får vi timrapporter över deras arbetstider i timmar och minuter. Ett program behövs som läser in en timanställds namn, timlön och arbetstider för varje veckodag. Sedan ska programmet summera arbetstiderna, beräkna veckolönen samt visa både veckans totala lön och arbetstid i timmar och minuter. En kontrollräkning ska bekräfta resultatet. Lönespecifikationen ska skrivas ut till en fil, så att den kan skickas till våra medarbetare.”

Kundens kravspecifikation

Vi döper uppdraget till *projekt Lönespecifikation* och bestämmer oss för att lösa problemet med ett objektorienterat program. Men hur ska vi lägga upp en objektorienterad lösning till detta projekt? Det finns inga fasta tillvägagångssätt som är allmängiltiga, inga generella recept som kan tillämpas i alla situationer. Ändå vill vi försöka att visa en steg för steg-algoritm som är någorlunda användbar i de flesta fallen. Vi ska först bygga en objektorienterad modell av projektet och sedan implementera modellen dvs förverkliga den i C# kod.

UML design och modellering i fyra steg

Steg 1: Förstå problemet

Läs kravspecifikationen (rutan ovan) *flera gånger* och försök att få en så exakt uppfattning som möjligt av kundens uppdrag. Det låter som en självklarhet. Men en korrekt uppfattning av problemställningen är faktiskt avgörande. Med andra ord för att lösa problemet måste vi *förstå* det. För att *förstå problemet*, måste vi vara förtrogna med den praktiska situationen och ha en någorlunda god insikt i problemets viktigaste aspekter utan att därför behöva vara expert i ämnet. Glöm för ett tag programmeringen och sätt dig mentalt in i en annan *roll* – i en ansvarige för ett

företags verksamhet som vill betala ut löner till sina timanställda. OBS! Det här är en attitydfråga – svårare än programmeringen.

Steg 2: Identifiera problemets nyckelbegrepp

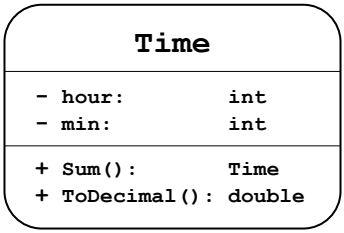
De kommer vid implementeringen att bli programmets ***klasser***. I databasmodellering används begreppet ***Entitet***. *Det är något viktigt för verksamheten – reellt eller virtuellt – som man kommer att behöva lagra information om.* Närmare bestämt handlar det om en *kategori av saker och ting som är relevanta för verksamheten*. Det är inte alltid enkelt att avgöra vad som är relevant. Och därför är det möjligt att ställa upp olika modeller av en och samma situation. Vilka *nyckelbegrepp* finns det i projektet Lönespecifikation? Vi bestämmer oss redan för en viss modell när vi t.ex. konstaterar att i händelsernas centrum står en *timanställd* som vi kommer att behöva lagra information om. Därför väljer vi nyckelbegreppen *anställd* och *tid*.

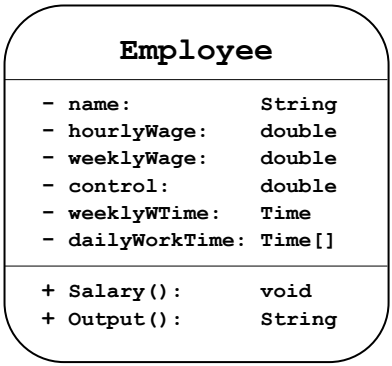
Steg 3: Identifiera datamedlemmar till varje klass

Har man hittat ett nyckelbegrepp så är nästa fråga: Vad har detta nyckelbegrepp för *egenskaper* eller *attribut*. Ett begrepp kan ofta definieras som mängden av alla sina egenskaper. Detta kommer att avgöra vilka *datamedlemmar* vi kommer att ha i den klass som definierar begreppet. Vad är det som utgör en anställd? Läser man projektets beskrivning noga hittar man en anställds *namn*, *timlön* och *arbetstid*. Ett annat sätt att hitta attribut till ett nyckelbegrepp är den s.k. *har*-relationen.

”Har”-relationen

För att konstatera om *namn*, *timlön* och *arbetstid* är en anställds attribut, är det ofta nyttigt att testa den s.k. *har*-relationen: *Har* en anställd ett namn, en timlön och en arbetstid? Svaret är ja. Dessutom måste vi ha en anställds arbetstider som *dagarbetstider*. Så, vi kan redan definiera *namn*, *timlön (hourlyWage)* och *dagarbetstider (dailyWorkTime)* som datamedlemmar till klassen anställd. Projektets krav på att beräkna ”veckans totala lön och arbetstid i timmar och minuter” samt kontrollräkningen leder till att även inkludera *veckolön (weeklyWage)*, *kontroll* och *veckoarbetstid (weeklyWTime)*. Vi ställer upp följande **klassdiagram:**





Enligt standarden UML *(Unified Modeling Language)* sätts i klassdiagrammen ovan symbolen **+** framför metoderna medan symbolen **–** skrivs framför datamedlemmarna. Dessutom är det standard i klassdiagrammen att ange datamedlemmarnas datatyper samt metodernas returtyper inkl. **`void`** för metod utan returvärde. Anmärkningsvärt i diagrammen är att klassen **`Time`** förekommer som returtyp till metoden **`Sum()`** och även i arrayform **`Time[]`** som datatyp till datamedlemmen **`dailyWorkTime`**. **`Time[]`** är en array av referenser till **`Time`**-objekt.

Steg 4: Att identifiera metoder till varje klass

Nästa fråga vi ställer till nyckelbegreppet anställd är: Vilka *operationer* är relevanta för en anställd? Svaret på denna fråga kommer att avgöra vilka *metoder* vi kommer att definiera i denna klass. En blick på uppgiftens beskrivning visar att det är *löneberäkningen* som är intressant för en anställd. Så, vi kommer att ta upp en metod i modellen, säg *Salary()*, som beräknar en anställds veckolön. Kravet på utskrift av veckolön och veckoarbetstid leder till ytterligare en metod som vi t.ex. kan beteckna med *Output()*. Därmed är behandlingen av nyckelbegreppet *anställd* avslutad.

För att ta reda på om det finns fler nyckelbegrepp i projektet Lönespecifikation, återvänder vi till beskrivningen (sid 64). Där handlar det mycket om att ”addera arbetstiderna, beräkna veckolönen …” och utföra någon form av kontrollräkning. Den här delen av projektet har att göra med tider och kräver att vi har rutiner som kan hantera tider. Vi skulle kunna införa nyckelbegreppet *arbetstid*. Men arbetstid är en underkategori av begreppet *tid*. Så för att vara mer generell och kunna använda koden även i andra program där tid är av intresse, betecknar vi nyckelbegreppet som *tid*. Vi kan sedan välja **`hour`** och **`min`** som datamedlemmar samt **`Sum()`** som metod i klassen *tid*.

En annan omständighet som motiverar införandet av nyckelbegreppet *tid*, är datamedlemmen *dagarbetstider* i klassen **`Employee`**. Varje datamedlem måste ju få en datatyp när vi är klara med modelleringen och vill implementera modellen. Datamedlemmen *name* kan få datatypen **`String`**. Datamedlemmen *hourlyWage* (*timlön*) kan bli en **`double`**. Men vilken datatyp ska datamedlemmen *dagarbetstider* ha? Det finns ingen fördefinierad datatyp för den. Så, vi måste själva definiera en ny datatyp genom att skapa nyckelbegreppet *tid* och därmed klassen **`Time`** med datamedlemmarna **`hour`**, **`min`** och metoden **`Sum()`**.

Återstår problemet med *kontrollräkningen* som projektet kräver. En meningsfull kontroll måste använda sig av ett *annat* beräkningsförfarande än det ”vanliga” för att kunna verkligen kontrollera beräkningens resultat. Man kan t.ex. addera tider genom att räkna i ”timme-minut-systemet” dvs addera timmarna och minuterna för sig och få resultatet i timmar och minuter. Detta blir vårt ”vanliga” beräkningsförfarande. Men man kan addera tider även genom att omvandla tiderna till decimaltal först – t.ex. 2 timmar och 45 minuter till 2,75 – och addera dem sedan som vanliga decimaltal, dvs räkna i det decimala talsystemet. Detta alternativa sätt att addera

tider kan vi använda för kontroll. Därför måste vi komplettera klassen **`Time`** med ytterligare en metod som utför omvandlingen av tider till decimaltal. Låt oss kalla den för **`ToDecimal()`**.

Implementeringen av modellen ovan som vore nästa steg behandlas nedan. Men i fall att man redan gjort en sådan implementering skulle ett körresultat se ut så här:

```
Mata in anställdens för- & efternamn:   Kalle Karlsson

Mata in timlön: 92,50

Arbetstid dag 1 i veckan (tim mellanslag min):  5 30
Arbetstid dag 2 i veckan (tim mellanslag min):  4 45
Arbetstid dag 3 i veckan (tim mellanslag min):  6 15
Arbetstid dag 4 i veckan (tim mellanslag min):  7 10
Arbetstid dag 5 i veckan (tim mellanslag min):  3 50

Den anställda Kalle Karlsson
har arbetat denna vecka:        27 timmar och 30 minuter

Veckolönen är                   2.543,75 kr

Kontrollräkning:                2.543,75 kr
```

De sista fyra raderna av utskriften ovan visar själva lönespecifikationen. Raderna innan levererar materialet till den (indata).

2.2 Implementation av modellen

En modells *implementation* är modellens förverkligande i kod. Den ska vara skild från själva modelleringen, därför att modellen ska vara oberoende av programvaran. Olika språk och olika versioner av programvaran ska kunna användas i olika datormiljöer vid olika tider. Det var också anledningen varför vi inte gick in på hur koden såg ut så länge vi modellerade. Nu ska vi gå vidare och utveckla C#-implementationen enligt den modell vi ställde upp för projektet Lönespecifikation. Vi börjar med implementationen av klassen **`Time`** enligt klassdiagrammet på sid 65:

```
// Time.cs
// Deklarerar klassen Time med 2 datamedlemmar och 2 metoder
// Metoden Sum() adderar två tider i timmar och minuter.
// Metoden ToDecimal() omvandlar tiden till decimaltal.
class Time
{
   public int hour, min;          // Datamedlemmar

   public Time Sum(Time t)        // Metod som tar in en Time,
   {                              // adderar den med klassens
      Time temp = new Time();     // datamedlemmar och retur-
      temp.min = (min + t.min) % 60; // nerar summan som Time
      temp.hour = (hour + t.hour) + (min + t.min)/60;
      return temp;
   }      // Om t1 är ett Time-objekt blir t1.Sum(t2) = t1 + t2

   public double ToDecimal()          // Metod som omvandlar
   {                                  // tiden till decimaltal
      return (60*hour + min)/60.0;    // 60.0 påtvingar double
   }
}
```

Metoden **`Sum()`** adderar sin egen parameter, tiden **`t`**, med klassens datamedlemmar och returnerar summan som ett **`Time`**-objekt. Först adderas minuterna modulo **`60`**. Dvs multipla av **`60`** tas bort för att tillfoga dem till timmarna i nästa sats. *Modulo-operatorn* **`%`** ger *resten* vid heltalsdivision. I nästa sats adderas timmarna. Båda returneras via ett lokalt **`Time`**-objekt **`temp`** till **`Sum`**. Om t_1 är ett **`Time`**-objekt blir **`t1.Sum(t2)`** = $t_1 + t_2$. Metoden **`Sum()`** fungerar som (simulerar) additionsoperatorn + mellan t_1 och t_2 .

Metoden **`ToDecimal()`** omvandlar tiden till decimaltal. Uttrycket **`(60*hour + min)/ 60.0`** utför omvandlingen från tidens *två* heltalsvärden **`hour`** och **`min`** till *ett* decimaltal. Parentesen omvandlar allt till minuter, sedan divideras minuterna med **`60`**. För att förhindra heltalsdivision divideras med **`60.0`** för att påtvinga vanlig decimal division.

Man kan ju undra varför vi överhuvudtaget behöver metoden **`Sum()`** när vi med metoden **`ToDecimal()`** kan omvandla alla tider till decimaltal och addera dem sedan som vanligt. Anledningen till det är projektets krav på att ”skriva ut veckans arbetstid i timmar och minuter” (sid 64). Detta kan endast åstadkommas med metoden **`Sum()`** som ger summan i *timmar och minuter*. **`ToDecimal()`** ska bara *kontrollera* resultatet med en annan, oberoende metod.

Följande klass **`Employee`** implementerar modellens klassdiagram på sid 65:

```
// Employee.cs
// Deklarerar klassen Employee. Datamedlemmen dailyWorkTime
// deklareras som en array av referenser till Time-objekt.
// Objekten skapas i klassen EmploTest, metoden Main().
// Metoden Salary() beräknar veckolönen och kontrollen.
// Metoden Output() presenterar klassens data som sträng.
using System;
class Employee
{
  public String name;
  public double hourlyWage, weeklyWage, control;
  public Time weeklyWTime;                    // Veckoarbetstid
  public Time[] dailyWorkTime = new Time[5];// Dagarbetstider
                                          // Array av referenser
  public void Salary()
  {
    weeklyWTime = dailyWorkTime[0]; // Dagarbetstider adderas
    for (int i=1; i<5; i++)          // med metoden Sum()
        weeklyWTime = weeklyWTime.Sum(dailyWorkTime[i]);
    weeklyWage = weeklyWTime.ToDecimal() * hourlyWage;

    double decimalSum = 0;
    for (int j=0; j<5; j++)        // Decimal kontrollräkning
        decimalSum += dailyWorkTime[j].ToDecimal();
    control = decimalSum * hourlyWage;
  }

  public String Output()
  {
   return "\n\tDen anställde " + name + "\n"                +
   "\thar arbetat denna vecka:\t"   + weeklyWTime.hour      +
   " timmar och " + weeklyWTime.min + " minuter\n\n"        +
   "\tVeckolönen är\t\t\t" + weeklyWage.ToString("c") + "\n"+
   "\n\tKontrollräkning:\t\t" + control.ToString("c") + "\n";
  }
}
```

I projektet Lönespecifikation (sid 64) handlar det om timlöner under en tidsperiod av en vecka och därmed – om man förutsätter vanlig arbetstid – om *fem dagarbetstider*, en för varje arbetsdag i veckan. Därför måste klassen **`Employee`** egentligen

ha fem olika datamedlemmar av typ **Time**. Men frågan är, kan man inte förenkla det? Vad gör man om man har ännu större antal data, t.ex. arbetstiderna under en hel månad? Vi har lärt oss hur man i ett C#-program hanterar större datamängder: Det är den sammansatta datatypen *array* som grupperar flera variabler av samma typ under ett namn. Det kan vi även göra här för att gruppera de fem enskilda dagarbetstiderna till en array **dailyWorkTime**. Även hur man definierar en array av *referenser* lärde vi oss i i förra avsnitt. Möjligheten att kunna bilda en array av klassen **Time** utnyttjas i klassen **Employee** för att deklarera datamedlemmen **dailyWorkTime** som en array av fem referenser till objekt av typ **Time**. Array använder vi för att inte behöva skapa fem enskilda **Time**-objekt i fem olika satser och för att kunna bearbeta dem senare i en **for**-loop.

Array av referenser

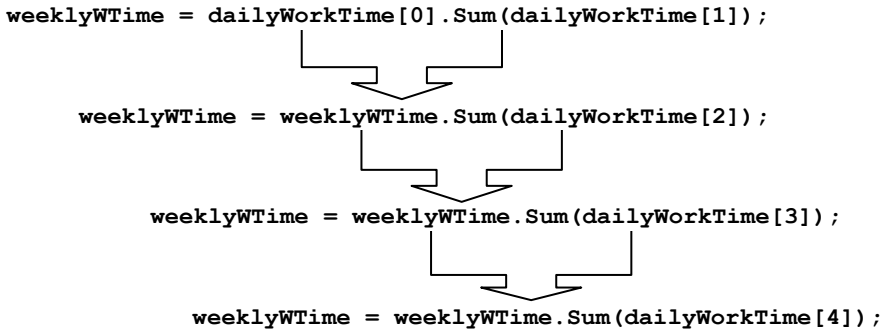
I klassen **Employee** på förra sidan skapas en array av referenser av typ **Time**:

```
Time[] dailyWorkTime = new Time[5];
```

Storleken är 5 dvs **dailyWorkTime** är en referens till en array av 5 referenser som kan lagra adresser till **Time**-objekt. Vi måste *addera* veckodagarnas arbetstider. I klassen **Employee** finns det två **for**-satser i metoden **Salary()**. Båda genomför addition av dagarbetstider. Den första gör det i timmar och minuter med metoden **Sum()** som är definierad i klassen **Time**, den andra med vanlig addition av decimaltal m – för kontrollens skull. Här förklaras hur metoden **Sum()** adderar de fem veckodagarnas arbetstider som en array av referenser:

```
weeklyWTime = dailyWorkTime[0];
for (int i=1; i<5; i++)
weeklyWTime = weeklyWTime.Sum(dailyWorkTime[i]);
```

Först initieras veckans arbetstid **weeklyWTime** till den *första* veckodagens arbetstid **dailyWorkTime[0]**. Sedan lägger **for**-satsen till nästa veckodagens arbetstid genom att anropa metoden **Sum()** med denna arbetstid som parameter och tilldela resultatet till veckans arbetstid **weeklyWTime**. Detta gör **for**-satsen i fyra varv:



dailyWorkTime[0] är måndagens, **dailyWorkTime[1]** tisdagens arbetstid osv. I **for**-satsens första varv läggs måndagens och tisdagens summa i **weeklyWTime**. I den andra läggs till denna summa onsdagens arbetstid osv. I de följande varven adderas veckans resterande arbetstider genom att ackumulera arbetstiderna i **weeklyWTime** med anrop av metoden **Sum()**. På så sätt kommer veckans samtliga dagarbetstider vara ackumulerade i **weeklyWTime**.

Kontrollen

Nu förklaras hur den andra **for**-satsen i metoden **Salary()** fungerar: Alla tider omvandlas först till decimaltal och adderars sedan på vanligt sätt:

```
double decimalSum = 0;
for (int j=0; j<5; j++)
    decimalSum += dailyWorkTime[j].ToDecimal();
```

for-satsen är en kortform för följande sats som gör exakt samma sak:

```
decimalSum = dailyWorkTime[0].ToDecimal() +
             dailyWorkTime[1].ToDecimal() +
             dailyWorkTime[2].ToDecimal() +
             dailyWorkTime[3].ToDecimal() +
             dailyWorkTime[4].ToDecimal() ;
```

Veckans fem dagarbetstider omvandlas med **ToDecimal()** till decimaltal, adderas och lagras i variabeln **decimalSum**. I **for**-satsen används **+=** som först adderar operatorns båda sidor och lägger summan sedan i **decimalSum**. Därför måste variabeln **decimalSum** vara tilldelad ett värde redan i **for**-satsens första varv, annars blir det kompileringsfel. Denna tilldelning görs före **for**-satsen till **0** för att inte förfalska summan.

Test av klasserna **Employee** *och* **Time**

I programmet **EmploTest** sakapas i **Main()**:s första sats ett objekt **emp** av typ **Employee**. I och med detta skapas bl.a. en array av 5 referenser till **Time**-objekt därför att datamedlemmen **dailyWorkTime** i klassen **Employee** är en sådan array, nämligen de 5 dagarbetstiderna som sedan läses in. Anropen av metoderna **Salary()** och **Output()** följer.

```
// EmploTest.cs
// Skapar ett objekt av klassen Employee, läser in namn, tim-
// lön samt en veckas dagarbetstider och anropar metoden Sa-
// lary() som adderar arbetstiderna och beräknar lönen.
// Anropet av metoden Output() ger resultaten som skrivs ut.
// Sedan skickas utskrift till filen SalarySpecification.txt.
using System;
class EmploTest
{
  static void Main()
```

```
  {
    Employee emp = new Employee();           // Objekt skapas

    Console.Write("\n\tMata in anställdens namn:\t");
    emp.name = Console.ReadLine();

    Console.Write("\n\tMata in timlön:\t");
    emp.hourlyWage = Double.Parse(Console.ReadLine());
    Console.WriteLine();

    String input;
    for (int i=0; i<5; i++)
    {
      emp.dailyWorkTime[i] = new Time();     // Initiering av
             // elementen i referensarrayen till Time-objekt
      Console.Write("\tArbetstid " + "dag " + (i + 1)   +
                    " i veckan (tim mellanslag min):\t");
      input = Console.ReadLine();            // Inläsning av
                                             // dagarbetstider
      emp.dailyWorkTime[i].hour =
                     Int16.Parse(Convert.ToString(input[0]));
      emp.dailyWorkTime[i].min  =
                10 * Int16.Parse(Convert.ToString(input[2]))
                   + Int16.Parse(Convert.ToString(input[3]));
    }

    emp.Salary();
    Console.WriteLine(emp.Output());   // Skrivs till skärmen
    WriteFile.Write(emp.Output(), "SalarySpecification.txt");
  }                                    // Skrivs till filen
}
```

I **Main()** blir alla datamedlemmar till objektet **emp** – inklusive arrayen **dailyWorkTime** – initierade innan **Salary()** anropas: Vi matar in namn och timlön till den anställda **emp** samt hans/hennes arbetstider för varje dag under en vecka. Inläsningen av dagarbetstiderna i timmar och minuter sker i form av en sammanhängande sträng. Sedan tas inmatningens olika delar – timmarna och minuterna – ut med **input[0]**, **input[2]** och **input[3]**. **input[1]** är mellanslaget. Resten av koden i detta sammanhang är omvandlingar av den inlästa datatypen **String** till **int**. Initieringen av objektets *alla* datamedlemmar före anropet av metoden **Salary()** behövs för att kunna skicka de relevanta data till löneberäkningen.

Arrayen **dailyWorkTime** av *referenser* av typ **Time** som tidigare skapats som en datamedlem i klassen **Employee**, initieras nu i en **for**-sats till *objekt* av typ **Time**:

```
for (int i=0; i<5; i++)
{
    emp.dailyWorkTime[i] = new Time();
    . . .
```

I varje varv skapas ett **Time**-objekt som tilldelas ett element i referensarrayen **dailyWorkTime** som är en datamedlem i **Employee**-objektet **emp**.

Skrivning till fil

I sista satsen av programmet **EmploTest** anropas metoden **Write()** som är definierad i följande klass **WriteFile**:

```
// WriteFile.cs
// Metod som skriver texten t till filen fileName
using System.IO;
class WriteFile
{
   public static void Write(string t, string fileName)
   {
      StreamWriter fileForWrite = new StreamWriter(fileName);
      fileForWrite.WriteLine(t);
      fileForWrite.Close();
   }
}
```

Det är samma klass som vi skrev när vi krypterade filer. Så kan vi dra fördel av modularisering.

Här följer ett körresultat:

```
   Mata in anställdens för- & efternamn:   Kalle Karlsson

   Mata in timlön: 92,50

   Arbetstid dag 1 i veckan (tim mellanslag min):  5 30
   Arbetstid dag 2 i veckan (tim mellanslag min):  4 45
   Arbetstid dag 3 i veckan (tim mellanslag min):  6 15
   Arbetstid dag 4 i veckan (tim mellanslag min):  7 10
   Arbetstid dag 5 i veckan (tim mellanslag min):  3 50

   Den anställda Kalle Karlsson
   har arbetat denna vecka:        27 timmar och 30 minuter

   Veckolönen är                   2.543,75 kr

   Kontrollräkning:                2.543,75 kr
```

De sista fyra raderna av utskriften ovan – själva lönespecifikationen – skrivs samtidigt till filen **SalarySpecification.txt** som kan hittas i projektmappen, undermappen \bin\Debug. För att se den högerklicka i Visual Studios Solution Explorer på projektnamnet och välj Open Folder in File Explorer. Gå sedan till \bin\Debug.

Övningar till kapitel 2

2.1 *Lönespecifikation med veckonummer och datum*

Vidareutveckla projektet Lönespecifikation som implementerades i detta kapitel genom att infoga aktuellt veckonummer och datum i lönespecifikationen, både på skärmen och till filen **`SalarySpecification.txt`**.

Läs om klassen **`DateTime`** nedan. Använd den för att hämta datorns tid till ditt program. Årets veckonr finns dock inte bland klassens datamedlemmar. Försök att beräkna veckonumret med hjälp av de fördefinierade datamedlemmarna i **`DateTime`**. Om du inte lyckas kan du använda följande klass:

```
// WeekOfYear.cs
// Metoden Week() beräknar årets veckonummer när årets
// dag och veckans dag är givna.
using System;

class WeekOfYear
{
   static int Week()
   {
      int dayY = DateTime.Now.DayOfYear;
      int dayW = (int) DateTime.Now.DayOfWeek;
      return (dayY - dayW + 7) / 7;
   }
}
```

Fundera över hur man får formeln i **`return`**-satsen. Integrera klassen **`WeekOfYear`** i projektet Lönespecifikation för att infoga aktuellt veckonummer och datum i lönespecifikationen.

Klassen `DateTime`

I C# finns den fördefinierade klassen **`DateTime`** som har datamedlemmar som lagrar datorns tid. Vi behöver inte skriva något nytt **`using`**-direktiv för att få tag i denna klass eftersom den finns med i biblioteket **`System`**. Klassen **`DateTime`** har bl.a. ett objekt som heter **`Now`** som är av typ **`DateTime`** och initieras till datorns aktuella datum och tid. Detta **`DateTime`**-objekt har t.ex. datamedlemmarna **`Hour`** och **`Minut`**. De tar ut timmen och minuten ur den aktuella tiden som ett heltalsvärde. Därför kan vi få tag i dem genom att i vårt program skriva:

```
int hour = DateTime.Now.Hour;    // Tar ut datortidens
int min  = DateTime.Now.Minute;  // timme resp. minut
```

Samma sak kan man göra med den aktuella datortidens andra delar.

Kapitel 3

Metoder i OOP

Ämne	Sida	Program
3.1 Accessmetoder	76	**`Empl/GetSet`**
3.2 Property i C#	79	**`Property`**
3.3 Statiska datamedlemmar och metoder	82	**`StatDemo`**
- Klass- och instansvariabler	82	**`StatDemoTest`**
- Allokeringsmodifieraren **`static`**	84	**`RandTest`**
3.4 Överlagring av metoder	88	**`Overload`**
3.5 Variablers livslängd	92	**`Block`**
3.6 Överskuggning av variabler	96	**`OverrideVar`**
- Referensen **`this`**	97	
3.7 Lambdauttryck	102	**`Lambda`**
3.8 Delegater	105	**`Delegate`**
- Delegat som parameter i metoder	106	**`DelegParam`**
- Varianter av **`Console.WriteLine()`**	108	**`WriteLOverl`**
- Lösningen med LINQ	109	**`CountLINQ`**
- Metodgrupper	110	**`MethodGroup`**
Övningar till kapitel 3	112	

3.1 Accessmetoder

Här återanknyter vi till vår diskussion kring inkapsling och klassens konstruktor (sid 28). 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 metoden
// 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 lönen

      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
// datamedlemmen 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 **MyFirst** introducerades sades så här om metoden **Main()**:

> ”Det är Virtual Machine (VM) som exekverar vårt program **MyFirst** genom att anropa metoden **Main()**. För att detta anrop ska kunna utföras behövs den s.k. *modifieraren* **static** i metodens huvud.”

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 så här:

static	att den kan anropas utan att skapa objekt av klassen **MyFirst**,
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** tillhör kategorin *returtyper*. Medan **public** och **private** är s.k. *åtkomstmodifierare*, är **static** en s.k. *allokeringsmodifierare* som vi kommer att behöva när vi skriver egna metoder som ska anropas från **Main()** utan att skapa ett objekt. Men **static** kan inte bara skrivas framför metoder utan även framför klasser och datamedlemmar. Vi tar upp först datamedlemmar som man kan dela upp i två kategorier:

Klass- och instansvariabler

En icke-statisk datamedlem kallas för *instansvariabel*. *Instans* är synonym till objekt. Följande klass visar ett exempel på båda dessa kategorier:

```
// 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;         // Instansvariabel kommer att
                               // 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 initi" +
         "eras 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 för icke-statiska datamedlemmar är *instansvariabler* (instans = objekt).

Statiska datamedlemmar kallas för klassvariabler, därför att de tillhör klassen och inte något specifikt objekt och instansvariabler används för icke-statiska datamed-

lemmar, 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.

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;
class 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 på något sätt 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. Detta behov kommer att återigen visas när vi senare kommer att ta upp s.k. *anonyma funktioner* som även kallas *Lambdauttryck*, se sid 102.

3.4 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 parameter-
//    listor: 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 exponent 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:   abcdefghijklmnopqrstuvwxyzåää
Substring(10)   = delsträng från index 10 till strängens slut
                = klmnopqrstuvwxyzåää
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 **`klmnopqrstuvwxyzåäö`** 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`**.

3.5 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

```
Int16.Parse(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.

Blockstruktur

> 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 stor är 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 block-struktur. Nästa avsnitt tar upp ett sådant fall.

3.6 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. 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 97). **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 data-

medlemmar 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 97).

3.7 Lambdauttryck

Lambdauttryck (eng. *lambda expressions*) är korta funktioner utan namn. De anropas i samma kod som de definieras. Ex.: `(a, b) => a+b`
Lambdaoperatorn `=>` 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 i C#:

```
// Lambda.cs
// Lambdauttryck skrivs med Lambdaoperatorn => som separerar
// funktionens parametrar (vänster) från dess kropp (höger)
// => betyder "ska skickas till" (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**. 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
```

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.

Innan vi går vidare följer en parentes om LINQ som vi använde i programmet **Lambda**.

Vad är LINQ ?

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 an-

vänds 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`**.

3.8 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 deklarerar 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 program-exempel 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 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 dele-
                                   // gattyp 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 dele-
                                      // gat som parameter
  {                                   // Räknar antal element
                                      // i v som uppfyller det
    int counter = 0;                  // villkor som skickas
    foreach (int element in v)    // med delegat-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

Läs om LINQ på sid 103.

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 107) 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 102) 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 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 108): 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 över-
// lagrade 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 **WriteLine-Overl**:

```
         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 3

3.1 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()**.

3.2 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_3_1
{
   static void Main()
   {
      {
         int t = 30;
      }
      Console.WriteLine("t = " + t);
   }
}
```

3.3 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.

3.4 Följande program är givet:

```
// 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
```

```
using System;

class Non_modularized_1
{
  static void Main()
  {
    Console.Write("\n\tMata in ett heltal:\t\t");
    int no1 = Int16.Parse(Console.ReadLine());
    Console.Write("\n\tMata in ett heltal till:\t");
    int no2 = Int16.Parse(Console.ReadLine());

    Console.WriteLine("\n\n\t"                                    +
      no1 + " plus "   + no2 + " är " + (no1 + no2) + "\n\t" +
      no1 + " minus " + no2 + " är "  + (no1 - no2) + "\n\t" +
      no1 + " gånger " + no2 + " är " + (no1 * no2) + "\n\t" +
      no1 + " heltalsdividerad med "  +
                        no2 + " är " + (no1 / no2) + "\n\t" +
      no1 + " modulo " + no2 + " är " + (no1 % no2) + "\n\t");
   }
}
```

Modularisera programmet **`Non_modularized_1`** för att vidareutveckla det till en liten kalkylator: 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()`**.

3.5 Modularisera programmet **`Non_modularized_2`** nedan 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");
      years = Int16.Parse(Console.ReadLine());

      Console.Write("\n\tAnge antal månader:\t");
      months = Int16.Parse(Console.ReadLine());

      Console.Write("\n\tAnge antal veckor:\t");
      weeks = Int16.Parse(Console.ReadLine());

      Console.Write("\n\tAnge antal dagar:\t");
      days = Int16.Parse(Console.ReadLine());

      /* B e a r b e t n i n g */
      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");
   }
}
```

3.6 a) Vänd om problemet från övn 3.5. 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?

3.7 Modularisera programmet **`Non_modularized_3`** efter eget godtycke.

Kapitel 4

Tillämpningar

Ämne	Sida	Program
4.1 Tvådimensionella arrays	116	**DoubleArray**
4.2 Tabellhantering i filer	121	
- 2D array som parameter i metoder	122	**SetTable**
- Att skriva tabeller till filer	125	**WriteTable**
- Att uppdatera tabeller i filer	127	**UpdateTable**
4.3 Primtalsfaktorisering	129	**Prime**
- Rekursiv algoritm för primtalsfaktorisering	131	**PrimeFactors**
4.4 Rekursion	135	**Fibonacci**
- Collatz algoritmen, rekursiv	138	**Collatz_reku**
Övningar till kapitel 4	141	

4.1 Tvådimensionella arrays

Med array bearbetar man med fördel större mängder av data. Men ibland är inte kvantiteten avgörande utan datas *struktur*. Följande problem illustrerar detta:

> 6 elever i en klass har skrivit 4 olika prov och fått poäng i dem. Elevernas poäng ska lagras och skrivas ut. Man ska ha möjligheten att ändra ett provresultat samt lagra och skriva ut det uppdaterade resultatet.

6 elevers poäng i 4 prov kan lagras i en *tabell* med 6 rader och 4 kolumner.

Den naturliga datastrukturen för att lagra tabeller är *tvådimensionell (2D) array*.

Följande program löser problemet ovan med hjälp av en 2D array:

```
// DoubleArray.cs
// 6 elevers poäng i 4 prov lagras i en 2D-array (table)
// En elevs poäng i ett prov uppdateras och visas. Både den
// ursprungliga och uppdaterade poängtabellen skrivs ut.
// 2D array modellerar tabellen och kommer åt arrayelementen.

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();
   }
}
```

Tabeller har 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. 2D array är i C# och andra programspråk den datastruktur som kan även lagra tabeller så att man kan bearbeta dem. I programmet **`DoubleArray`** definieras och initieras 2D arrayen **`table`** med koden:

`int[ , ] table = { {...} ... {...} }`

Detta är en array vars element i sin tur är arrays, dvs en dubbel eller nästlad array. Den är av typ **`int`**. Storleken är 6 x 4, dvs en stor array bestående av 6 små arrays som var och en har 4 **`int`**-element. Därför är **`table`** en (6 x 4)-array. Strukturen kan jämföras med en tabell av 6 rader och 4 kolumner. Storleken får inte anges explicit i hakparentesen om man fortsätter initiera arrayen med initieringslistan. I detta fall avläses storleken automatiskt från initieringslistan. I själva verket är det inget annat än en array av 6 arrays av 4 **`int`** arrays, 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 komman inpm hakparentesen. Vi nöjer oss dock med två dimensioner där vi har den enkla tabellanalogin.

Deklarationen av 2D arrayen **`table`** använder sig av initieringslistan som introducerades för endimensionella arrays. 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 2D array. Ett annat kännetecken är kommat i den 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 2D array

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 2D arrays. Hakparenteserna i koden **table[3, 1]** i programmet **DoubleArray** innehåller *indexen* till ett element i arrayen **table**. Det är 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. Det gäller: elementets position = index + 1. 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 116). 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 nivån. 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.

Metoden **PrintTable()**

Den ovan besrivna indexeringsregeln för 2D arrays används även i metoden **PrintTable()** där den nästlade **for**-satsen skriver ut hela poängtabellen:

```
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();
        }
```

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**. 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++)      // Skriver ut den 4:e upp-
        Console.Write(table[3, k] + "   ");  // daterade raden
```

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++)
                Console.Write(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. 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
```

4.2 Tabellhantering i filer

Betygsregistrering med 2D array har vi tidigare lärt känna i programmet **DoubleArray** (sid 116). Men en skarp applikation för betygsregistrering kan inte nöja sig med en utskrift på skärmen. Man måste kunna spara och lagra betygen någonstans, kunna visa upp och även uppdatera dem vid behov. Även de uppdaterade värdena borde kunna sparas och lagras. Dessutom är det i praktiken meningslöst att hårdkoda betygen i själva programkoden vilket vi gjort i **DoubleArray**. Helst ska man ha tillgång till och kunna läsa betygen efter att administrativ personal lagt in dem. Sedan vill man kunna ändra uppgifterna vid behov.

Idealt vore det förstås att registrera betygen i en databas som vi kommer att ta upp senare. Men inte långt därifrån ligger filhantering som vi lärt oss. Vi ska nu kombinera våra kunskaper i filhantering för att programmera betygsregistrering i filer. För att ge programmet en bra struktur och göra det lätt uppdaterbart, när det ska vidareutvecklas och växa, ska koden vara modulariserad. Uppdelningen i moduler har konsekvensen att vi kodar 2D arrays som parametrar i metoder – vilket också kommer att introduceras här. Låt oss titta på följande problemställning.

Problemställningen:

> Skriv en klass **SetTable** med en metod för betygshantering som genererar rådata genom att slumpa fram elevpoäng i ett antal prov i en 2D array. En metod i en annan klass **WriteTable** ska skriva dessa poäng till en fil. En tredje klass **ReadShowTable** ska läsa poängen därifrån och visa dem på skärmen. Slutligen ska en klass **UpdateTable** ha möjligheten att ändra elevers poäng (t.ex. efter omtenta), spara ändringarna i samma fil genom att anropa metoden i **WriteTable** och visa den uppdaterade tabellen. Programmet **TableFile** ska anropa dessa metoder i **Main()**.

Ett projekt med huvudprogrammet **TableFile** (sid 124) följer denna problemställning och består av följande fem moduler lagrade i resp. filer (lägg till ***.cs**):

TableFile innehåller:	**Main()** som definierar en 2D array och anropar alla andra metoder.
SetTable	metoden **Set()** som initierar en 2D array genom att anropa metoden **Next()**.
WriteTable	metoden **Write()** som skriver tabellen till en fil.
ReadShowTable	metoden **ReadShow()** som läser tabellen från filen och visar den på skärmen.
UpdateTable	metoden **Update()** som ändrar tabellen i filen genom att anropa **ReadShow()** och **Write()**.

I problemställningen ovan har man redan tänkt igenom strukturen hos uppgiften och formulerat en modell så att endast implementeringen i C# kvarstår. Vi börjar att implementera modellen med klassen **SetTable**:

```
// SetTable.cs
// Initierar 2D arrayen c med slumpvärden genom att anropa
// biblioteksmetoden Next() från klassen Random
// 2D array som parameter i en metod
using System;
class SetTable
{
   public static void Set(Random s, int[,] c,
                          int noOfStudents, int noOfProofs)
   {
      for (int r=0; r < noOfStudents; r++)
          for (int k=0; k < noOfProofs; k++)
              c[r,k] = s.Next(10, 101);
   }
}
```

2D array som parameter i metoder

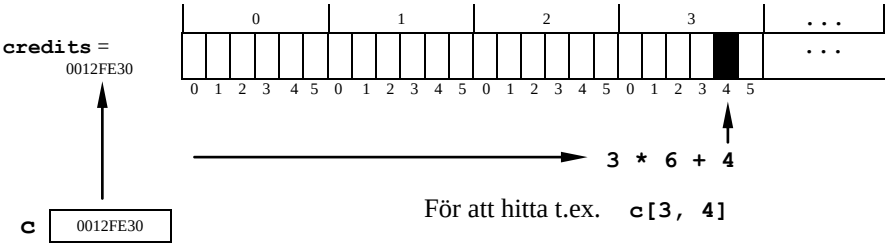
Det nya i klassen ovan är syntaxen i huvudet till metoden **Set()**, andra parametern **c** (framhävd med vit bakgrund). Där deklareras **c** som en 2D array. En enkel array som parameter hade vi redan behandlat tidigare. Det finns både likheter och skillnader i behandlingen av 1D och 2D arrays när man skickar dem som parametrar till metoder. I metoden **Set()** är den formella parametern **c** en 2D array som i parameterlistan deklareras så här:

```
int[,] c
```

Likheten till enkel array **int[] c** är hakparentesen. Skillnaden är till synes att hakparentesen i 2D arrayen *inte* är tom utan innehåller ett komma – en avgörande skillnad som gör arrayen till 2D. Här är **c** en referens till en 2D array som tar emot adressen till den 2D array vars referens **credits** som aktuell parameter skickas till metoden **Set()** när denna anropas. Arrayens storlek skrivs inte i den formella parameterns deklaration utan tas över från den aktuella parametern, där den anropande metoden har allokerat minne för 2D arrayen.

2D arrays allokeras så att de två indexnivåerna måste ändå linearisera vid allokering i ett sammanhängande minnesområde på ett sätt som följande bild försöker att illustrera:

Om varje elev gör **6** prov kan vi föreställa oss minesbilden ungefär så här:



credits är namnet på den array som vid anrop av metoden **Set()** kommer att överföras till parametern **c**. Från enkel array vet vi att **c** pga referensanrop endast lagrar adressen till arrayen **credits**. För att sedan i metoden kunna komma åt t.ex. värdet **c[3, 4]** går kompilatorn från denna adress till arrayen **credits**' index **3** på första nivå (elev 4), närmare bestämt till minnescell nummer **3 * 6 = 18** och därifrån till index **4** på andra nivå (prov 5). Men för denna beräkning behöver kompilatorn utöver indexen **3** och **4** även informationen om arrayens storlek på andra nivå dvs **6** som är antalet prov. Denna information är inte med i metoden där arrayparametern **c** endast lagrar arrayens adress, men inte dess storlek. Denna information nämligen **numbOfProofs** hämtas från den aktuella parametern **credits** i den anropande metoden **Main()**.

Resten av metoden **Set()** består av kända koncept som nästlad **for**-sats och anropet av metoden **Next()** som tidigare använts i andra program. Här anropas den för att fylla arrayen **credits** – via referensen **c** som pekar på **credits** – med rådata i form av slumptal mellan **10** och **100**. Dessa kommer att användas som material för betygshantering i programmets andra metoder. Den 3:e parametern **noOfStudents** tar emot elevernas antal och behövs för att avsluta den yttre **for**-slingan. Samma sak gör den 4:e parametern **noOfProofs**. Den nästlade **for**-satsen initierar 2D arrayen **credits** med hjälp av arrayparametern **c** så att 2D arrayen **poäng** efter anropet av metoden **Set()** i **Main()** är fylld med slumptal mellan **10** och **100** vilka simulerar elevernas poäng.

För att visa **Set()**-anropet i **Main()** och vad som i övrigt händer i denna applikation ska vi nu avvika från den ordning metoderna nämndes i uppgiften och presentera klassen **TableFile** med **Main()** innan vi går vidare.

I klassen **TableFile** på nästa sida definieras den tvådimensionella arrayen **credits** i **Main()** till storleken 14 x 6 (1:a framhävda raden med vit bakgrund), dvs 14 elever och 6 prov. Den är tom än så länge, men vi har allokerat minne för den med en **new**-sats för att sedan kunna fylla den med slumptal. Detta görs med anropet av metoden **Set()** enligt beskrivningen ovan. Efter anropet av **Set()** innehåller **credits** 14 x 6 slumptal (2:a framhävda raden med vit bakgrund). Att detta fungerar beror på att både **credits** och **c** refererar till samma 2D array dvs pekar på eller är adresser till samma minnesutrymme. Sedan skrivs arrayen **credits** till filen

Table.txt med anropet av **Write()**, läses från samma fil och visas på skärmen med **ReadShow()** för att slutligen ändras, om så önskas, med anropet av metoden **Update()** i slutet av **Main()**. De namngivna konstanterna **numbOfStudents** och **numbOfProofs** är arrayen **credits**:s kolumn- och radantal som skickas som aktuella parametrar till alla metoder där de överförs till de formella parametrarna **noOfStudents** och **noOfProofs**.

```
// TableFile.cs
// Anropar metoder som genererar elevernas poäng i olika prov
// i en tabell (2D array), skriver tabellen till en fil, lä-
// ser från filen och visar tabellen på skärmen
// Tabellen kan ändras och uppdateras i filen
using System;
class TableFile
{
  static void Main()
  {
    Random s = new Random();
    Console.Write("\n\tAnge antalet elever:\t"); // Radantal
    int numbOfStudents = Int16.Parse(Console.ReadLine());

    Console.Write("\n\tAnge antalet prov:\t"); // Kolumnantal
    int numbOfProofs = Int16.Parse(Console.ReadLine());
                                    // 2D arrayen deklareras:
    int[,] credits = new int[numbOfStudents, numbOfProofs];
                                    // 2D arrayen initieras:
    SetTable.Set(s, credits, numbOfStudents, numbOfProofs);
                                   // Arrayen skrivs till fil:
    WriteTable.Write(credits, numbOfStudents, numbOfProofs);

    Console.Write("\n\t" + numbOfStudents + " elever har " +
    "gjort" + numbOfProofs + " prov" +" och fått följande" +
                                             " poäng:\n\n");
    ReadShowTable.ReadShow();      // Arrayen läses från samma
                                   // fil och skrivs ut
    Console.Write("\tVill du göra ändringar i tabellen? " +
                                               "(J/N): ");
    char startChange = Convert.ToChar(Console.ReadLine());
                                 // Arrayen uppdateras:
    if (startChange == 'j' || startChange == 'J')
       UpdateTable.Update(credits,
                          numbOfStudents, numbOfProofs);
    Console.WriteLine();
  }
}
```

Nu går vi igenom de andra metoder som anropas i **Main()** efter **Set()**.

Att skriva tabellen till filen

```
// WriteTable.cs
// Skapar filen Table.txt för skrivning
// Skriver elevernas poäng som en tabell (2D array) till fil
using System.IO;

class WriteTable
{
   public static void Write(int[,] c, int noOfStudents,
                                          int noOfProofs)
   {
      StreamWriter fileForWrite = new StreamWriter
              ("Table.txt");   // Filen skapas för skrivning

      for (int r=0; r < noOfStudents; r++)
      {
          for (int k=0; k < noOfProofs; k++)
              fileForWrite.Write(c[r,k] + "\t");
          fileForWrite.WriteLine();
      }
      fileForWrite.Close();
   }
}
```

Metoden **`Write()`** skapar filen **`Table.txt`** på hårddisken i projektmappens undermapp **`C:\C#\MyProject\bin\Debug`**, öppnar den för att skriva i den och skickar till den med en nästlad **`for`**-sats hela arrayen som ligger lagrad vid adressen **`c`**. Eftersom **`Write()`** anropas i **`Main()`** med arrayen **`credits`** som aktuell parameter, är det denna arrays värden som skrivs till filen. I **`Main()`** anropas metoden **`Write()`** så här:

```
WriteTable.Write(credits, numbOfStudents, numbOfProofs);
```

Precis som i alla nästlade **`for`**-satser används parametern **`noOfStudents`** för att avsluta den yttre **`for`**-slingan, medan variabeln **`noOfProofs`** används för att avsluta den inre **`for`**-slingan vilket även gäller för nästa funktion **`ReadShow()`** som läser från samma fil som **`Write()`** skriver till.

Att läsa tabeller från filer

I klassen **`ReadShowTable()`** på nästa sida förekommer 2D arrayen inte alls i koden. Det är inte heller nödvändigt då filen **`Table.txt`** redan innehåller en tabellstruktur som endast behöver läsas och visas på skärmen. Vi behöver här inte komma åt enskilda arrayvärden. För att endast läsa och visa innehållet i filen kan vi behandla den som en ren textfil. Filen öppnas för läsning med filvariabeln **`fileForRead`** av typ **`ifstream`** så här:

```
StreamReader fileForRead = new StreamReader("Table.txt");
```

På sid 124 anropas metoden **ReadShow()** i **Main()** så här:

```
ReadShowTable.ReadShow();
```

Inga parametrar behöver skickas i anropet eftersom läsandet och visandet av filinnehållet är helt oberoende av vad som händer i **Main()**.

```
//ReadShowTable.cs
// Läser från filen Table.txt och visar innehållet på skärmen
using System.IO;
class ReadShowTable
{
   public static void ReadShow()
   {
      StreamReader fileForRead = new StreamReader
                                          ("Table.txt");
      Console.WriteLine(fileForRead.ReadToEnd());
      fileForRead.Close();
   }
}
```

Att uppdatera tabeller i filer

Slutligen ska vi även titta på den sista av programmets moduler, nämligen klassen **UpdateTable** med metoden **Update()** som ger användaren möjligheten att göra ändringar i elevernas poängtabell.

På sid 124 anropas metoden **Update()** i **Main()** så här:

```
if (startChange == 'j' || startChange == 'J')
   UpdateTable.Update(credits, numbOfStudents, numbOfProofs);
```

Anropet är inbakat i en **if**-sats för att ge användaren alternativet mellan att uppdatera tabellen eller inte göra det. Denna valmöjlighet kombineras med en **do**-loop i metoden **Update()** vars kod visas på nästa sida. Genom **do**-loopen har användaren möjligheten att göra *flera* ändringar i elevernas poängtabell. Varje ändring skrivs direkt till filen genom att lägga anropet av **Write()** i **do**-loopen (framhävd med vit bakgrund). Ändringen av ett arrayelement görs i varje **do**-varv genom inläsning från tangentbordet och skickas vidare till filen via den formella parametern **c** som finns i anropet av **Write()**. Själva ändringen görs med satsen:

```
c[rad-1, kolumn-1] = Int16.Parse(Console.ReadLine());
```

Den samlade uppdateringen av poängtabellen läses sedan från filen och visas på skärmen efter **do**-loopen med anropet av metoden **ReadShow()**.

```
// UpdateTable.cs
// Ändrar värden i elevernas poängtabell, skriver den uppda-
// terade tabellen tillbaka till filen Table.txt, läser däri-
// från och visar den uppdaterade tabellen på skärmen.
using System;
class UpdateTable
{
  public static void Update(int[,] c,
                            int noOfStudents, int noOfProofs)
  {
    int row, column, old;
    char goOnChange;
    do
    {
      Console.Write("\n\tVilken elevs poäng vill du "    +
            "ändra på? Elev nr (1-" + noOfStudents + "): ");
      row = Int16.Parse(Console.ReadLine());
      Console.Write("\tVilket provs poäng vill du "      +
            "ändra på? Prov nr (1- " + noOfProofs + "): ") ;
      column = Int16.Parse(Console.ReadLine());
      old = c[row-1, column-1];       // Gammalt värde sparas
                                      // Nytt värde läses in
                                      // och ändras i arrayen:
      Console.Write("\n\tÄndra poängen till: ");
      c[row-1, column-1] = Int16.Parse
                                   (Console.ReadLine());
      Console.Write("\n\tElev " + row + ":s gamla poäng: " +
               old + ", nya poäng: " + c[row-1, column-1]  +
                              " i prov " + column + "\n\n");
                        // Uppdaterad tabell skrivs till fil:
      WriteTable.Write(c, noOfStudents, noOfProofs);

      Console.Write("\tVill du fortsätta ändra? (J/N): ");
      goOnChange = Convert.ToChar(Console.ReadLine());
    } while (goOnChange  == 'j' || goOnChange  == 'J');

    Console.WriteLine("\n\tUppdaterad tabell:\n");
    ReadShowTable.ReadShow();     // Uppdaterad tabell läses
   }                              // från fil och skrivs ut
}
```

Programmet **`TableFile`** liknar mycket hanteringen av en databas som administrerar betyg. Filen **`Table.txt`** som lagrar en tabell skulle kunna importeras till en databas. Programmet kan läsa tabellen, skriva i den, ändra den och spara alla uppdateringar i filen. När programmet **`TableFile`** körs i sin helhet kan olika dialoger föras. Efter körningen står den uppdaterade slutliga poängtabellen i filen **`Table.txt`**. Dessutom kan t.ex. följande dialog med två uppdateringar ses på skärmen:

```
     14 elever har gjort 6 prov och fått följande poäng:

96      29      99      40      45      47
27      13      59      82      89      89
23      47      22      22      54      31
63      94      68      17      14      23
14      25      29      89      24      85
65      19      55      11      47      78
68      23      84      95      82      68
74      16      14      85      62      65
13      28      32      38      21      52
10      89      61      86      36      30
28      75      90      75      63      80
44      58      10      45      91      41
48      69      58      10      80      44
55      66      100     65      86      97

     Vill du göra ändringar i tabellen? (J/N): j

     Vilken elevs poäng vill du ändra på? Elev nr (1-14): 7
     Vilket provs poäng vill du ändra på? Prov nr (1- 6): 5

     Ändra poängen till: 55

     Elev 7:s gamla poäng: 82, nya poäng: 55 i prov 5

     Vill du fortsätta ändra? (J/N): j

     Vilken elevs poäng vill du ändra på? Elev nr (1-14): 14
     Vilket provs poäng vill du ändra på? Prov nr (1- 6): 6

     Ändra poängen till: 99

     Elev 14:s gamla poäng: 97, nya poäng: 99 i prov 6

     Vill du fortsätta ändra? (J/N): n

     Uppdaterad tabell:

96      29      99      40      45      47
27      13      59      82      89      89
23      47      22      22      54      31
63      94      68      17      14      23
14      25      29      89      24      85
65      19      55      11      47      78
68      23      84      95      55      68
74      16      14      85      62      65
13      28      32      38      21      52
10      89      61      86      36      30
28      75      90      75      63      80
44      58      10      45      91      41
48      69      58      10      80      44
55      66      100     65      86      99
```

4.3 Primtalsfaktorisering

Primtal är positiva heltal > 1 som är jämnt delbara *endast* med sig själv och med 1. Alla heltal är jämnt delbara med sig själv och med 1. De flesta av dem är dessutom jämnt delbara med andra tal. Men primtal kan inte delas jämnt med *något* annat tal än med sig själv och med 1. I den meningen är primtal odelbara – talsystemets *atomer*. Exempel:

Primtal: 2, 3, 5, 7, 11, 13, 17, 19, 23, …

Inga primtal: 4, 6, 8, 9, 10, 12, 14, … *Sammansatta* tal – sammansatta av primtal.

Aritmetikens fundamentalsats

Varje positivt heltal kan på ett entydigt sätt delas upp i en produkt av primtal, bortsett från faktorernas inbördes ordning.

Primfaktorernas inbördes ordning är utan betydelse eftersom multiplikation är kommutativ, dvs $a \cdot b = b \cdot a$. Satsen ovan samt exemplen nedan påminner om att primtalen är talsystemets byggstenar, jämförbart med materiens odelbara atomer:

$$12 = 2 \cdot 2 \cdot 3$$

$$98 = 2 \cdot 7 \cdot 7$$

$$11111111 = 11 \cdot 73 \cdot 101 \cdot 137$$

Medan de första två exemplen är någorlunda begripliga är det svårt att kontrollera det tredje exemplet utan miniräknare. Hur man utför själva faktoriseringen är, beroende på talets storlek, en jobbig procedur, även om man kan det. Det vore kul att ha ett program som gör det åt oss för vilket tal som helst, där vi matar in ett tal och får dess uppdelning i primfaktorer – en slags digital realisering av aritmetikens fundamentalsats.

I detta avsnitt ska vi ägna oss åt att hitta alla primfaktorer av ett givet heltal. Den teoretiska garantin för att en sådan faktorisering alltid *existerar* är den ovannämnda fundamentalsatsen om unik uppdelning av positiva heltal i primfaktorer.

Primtalsfaktorisering

Med division menas i hela detta avsnitt *heltals*division. Ett sätt att hitta ett tals primfaktorer är att dividera det med 2, 3, 4, … tills resten blir 0. Det första tal man delar med som ger resten 0 är en primfaktor. Om man sedan delar talet med denna primfaktor blir kvar antingen ett primtal eller produkten av de resterande primfaktorerna.

Här följer två enkla exempel:

```
>>> En slags avancerad kalkylator    Uppgift: Faktorisera 65

>>> 65 % 2                          65 delad med 2 ger resten 1
1                                   2 delar inte 65
>>> 65 % 3
2                                   3 delar inte 65
>>> 65 % 4
1                                   4 delar inte 65
>>> 65 % 5
0                                   5 delar 65, dvs:
>>> faktorlista = [5]               5 primfaktor -> Lägg i listan
>>> 65 / 5                          Tar bort faktorn 5 från 65
13                                  13 är primtal -> Lägg i listan
>>> faktorlista = faktorlista + [13]
>>> faktorlista                     65:s primtalsfaktorisering:
 [5, 13]
>>> 5 * 13                          Kontroll
65
>>>
```

Vi dividerar 65 med 2, 3, 4, … tills resten blir 0. Det första tal som ger resten 0 är 5, dvs 5 är 65:s första primfaktor. Vi definierar en lista som vi kallar för **`faktorlista`** och lägger 5 i den: **`faktorlista = [5]`**. Sedan ”förkortar” vi 65 genom att dela den med 5. Egentligen borde vi *upprepa förfarandet* med resultatet av 65/5. Men vi konstaterar att resultatet 13 är ett primtal. Alltså lägger vi 13 i listan: **`faktorlista = faktorlista + [13]`**. Dvs vi konkatenerar den gamla listan med elementet **`13`**. Sist skriver vi ut listan **`[5, 13]`** och gör kontroll. Talet 65:s primtalsfaktorisering är $65 = 5 \cdot 13$. Syntaxen är Pythons.

Vad vi i exemplet ovan kallade för *förfarandets upprepning* kunde inte genomföras eftersom resultatet blev primtalet 13. Om det hade varit sammansatt hade vi kunnat utnyttja det för att förkorta och förenkla algoritmen.

Följande exempel med ett större tal skulle kunna ge oss en bättre förståelse för upprepningsmomentet och bidra till algoritmens allmänna formulering:

```
>>> En slags avancerad kalkylator    Uppgift: Faktorisera 132

>>> 132 % 2
0                                 2 delar 132
>>> faktorlista = [2]             2 primfaktor -> Lägg i listan
>>> 132 / 2                       Tar bort faktorn 2 från 132
66
>>> 66 % 2                        Upprepa faktoriseríngen med 66
0
>>> faktorlista = faktorlista + [2]  Lägg en till 2:a i listan
```

```
>>> 66 // 2                    Tar bort faktorn 2 från 66
33
>>> 33 % 2                     Upprepa faktoriseríngen med 33
1
>>> 33 % 3
0                              3 är primfaktor -> Lägg i listan
>>> faktorlista = faktorlista + [3]
>>> 33 / 3                     Tar bort faktorn 3 från 33
11                             11 är primtal -> Lägg i listan
>>> faktorlista = faktorlista + [11]
>>> faktorlista
[2, 2, 3, 11]                  132:s primtalsfaktorisering
>>> 2 * 2 * 3 * 11             Kontroll
132
>>>
```

Skillnaden med det första exemplet är att vi nu efter den första divisionen av 132 med 2 som ger resten 0, *inte* får ett primtal utan 66. Vi tar in 2 i listan över primfaktorer och upprepar förfarandet med 66: Division av 66 med 2 med resten 0 ger igen ett sammansatt tal: 33. Vi tar in denna andra 2:a i primfaktorlistan och upprepar förfarandet med 33: Efter två försök, division med 2 och 3, ger det sista resten 0. Därför tas 3 in i listan. Resultatet 11 är ett primtal och hamnar i listan: **`[2, 2, 3, 11]`**. Så blir talet 132:s primtalsfaktorisering: $132 = 2 \cdot 2 \cdot 3 \cdot 11$.

Om vi hade en algoritm som dividerade ett tal med 2, 3, 4, … tills resten blir 0, hade vi kunnat skicka det sammansatta talet som blir resultatet av den sista divisionen (med rest 0) ”tillbaka” till denna algoritm för att *upprepa förfarandet*. Detta tillbakaskickande för att upprepa samma sak ska vi nu utnyttja i algoritmens allmänna formulering.

När vi sammanfattar våra lärdomar från de två experimenten ovan kan vi formulera följande allmän algoritm för uppdelning av positiva heltal i primfaktorer. En gång till: med division menas hela tiden *heltals*division.

Rekursiv algoritm för primtalsfaktorisering

Låt **n** vara det tal som ska delas upp i primfaktorer. Skapa en tom lista.

1. Dividera talet **n** med 2, 3, 4, … tills resten blir 0.
2. Låt **k** vara det första tal du dividerat med som gav resten 0. **k** är primfaktor.
3. OM **n** är primtal lägg **n** i listan ovan och avsluta algoritmen,
   ANNARS lägg primfaktorn **k** i listan ovan,
   gå tillbaka till punkt **1** med: **n = n / k** ***(Rekursion!)***

Algoritmen är rekursiv pga *tillbakagången* i sista raden. Nedan följer en implementation av denna algoritm i C# som således blir en *rekursiv metod*. Vi kallar den för **`Factorize()`** och definierar den i klassen **`Prime`** på nästa sida.

I metoden **`Factorize()`** dividerar en **`while`**-loop talet **`n`** med 2, 3, 4, … tills resten blir 0. Då har den hittat den faktor **`k`** som delar **`n`** och därför är en primfaktor. I **`else`**-delen av **`if`**-satsen läggs denna faktor i listan **`t`**. Sedan bryts **`k`** loss från **`n`** genom **`n/k`**. Sedan startas hela algoritmen om, nu med **`n/k`**: **`Factorize()`** anropar sig själv, därför *rekursivt*.

Implementation av algoritmen för primtalsfaktorisering

```
// Prime.cs
// Klass som definierar den rekursiva metoden Factorize()
// Faktoriserar ett positiv heltal i sina primfaktorer
using System.Collections.Generic;  // Krävs för klassen List

class Prime
{
  public static void Factorize(int n, List<int> t)
  {
    int k = 1, rest = 1;    // n är talet som ska faktoriseras
                            // t är primfaktorlistan
    while (k <= n-2 && rest != 0) // Letar efter primfaktor k
    {
       k++;
       rest = n % k;          // Delar n med k och tar resten
    }

    if (rest != 0) t.Add(n); // k delar inte n => n primtal
                             // Lägg n i listan och avsluta
    else
    {                        // k delar n:
       t.Add(k);             // Lägg primfaktorn k i listan
       Factorize(n/k, t);    // Ta bort faktorn k från n:
    }                        // Faktorisera resten (Rekursion)
  }
}
```

Vi testar klassen **`Prime`** i följande program:

C# Program för primtalsfaktorisering

```
// PrimeFactors.cs
// Skapar en lista och anropar metoden Faktorize() ovan
using System;
using System.Collections.Generic;  // Krävs för klassen List
class PrimeFactorizing
{
```

```
  static void Main()
  {
    List<int> factorList = new List<int>();  // List-objekt
    Console.Write("\n\t Mata in ett positivt heltal:\t");
    int tal = Int32.Parse(Console.ReadLine());
    Prime.Factorize(tal, factorList);     // Metodens anrop
    Console.Write("\n\t Talets primfaktorer:\t\t");
    foreach (int element in factorList)   // Listans utskrift
       Console.Write(element + " ");
    Console.WriteLine("\n");
  }
}
```

Nedan följer några körresultat:

```
     Mata in ett positivt heltal:   12
     Talets primfaktorer:           2, 2, 3
```

```
     Mata in ett positivt heltal:   98
     Talets primfaktorer:           2, 7, 7
```

```
     Mata in ett positivt heltal:   997
     Talets primfaktorer:           997
```

```
     Mata in ett positivt heltal:   11111111
     Talets primfaktorer:           11, 73, 101, 137
```

```
     Mata in ett positivt heltal:   4294967297
     Talets primfaktorer:           641 6700417
```

Matematikern Fermat trodde på 1600-talet att $2^{32} + 1 = 4\ 294\ 967\ 297$ var ett primtal. På 1700-talet fann Euler att det var sammansatt. Körexemplet ovan bekräftar Eulers resultat och visar även faktoriseringen.

Just det sista körexemplet går inte att exekvera med den angivna versionen av programmet **`PrimeFactors`**. Anledningen är att vi i den aktuella versionen har dekla-

rerat variablerna till datatypen **`int`**. Talet `4 294 967 297` överskrider max-gränsen för **`int`** som är **`int.MaxValue`** `= 2 147 483 647`. Försök själv att byta ut mot **`double`** för att klara av den sista körningen.

Omskrivning till iterativ lösning

Avslutande fråga: Kan man skriva den rekursiva metoden **`Factorize()`** även som en *iterativ* metod, dvs ersätta rekursionen med en vanlig upprepningsslinga (loop), t.ex. **`while`**? Svaret är ja. Frågan återfinns som övning 4.3 (sid 141) och lösningsförslaget på sid 185.

Generellt gäller att alla rekursioner kan skrivas om till iterationer (loopar). Det omvända är inte alltid sant.

4.4 Rekursion

Rekursion är ett koncept för problemlösning som använder sig av *successiv upprepning*. Ordet *rekursiv* kommer från latinska *recurrere*, på engelska: *to run back, to run again* dvs att gå tillbaka och köra igen. I modellering är rekursiva algoritmer sådana som går tillbaka och upprepar tidigare steg. I programmering implementeras rekursion i metoder som *anropar sig själva*.

Upprepade beräkningar är datorn bra på, vilket vi hittils löst med kontrollstrukturen repetition dvs med loopar. Sådana lösningar kallas för *iterativa*. Rekursion är alltså ett slags alternativ till loopar: Alla rekursiva lösningar kan kodas även med loopar. Men rekursiva algoritmer är inom programmering av intresse i sig – av olika skäl: De bygger på instruktioner som redan är formulerade och inte behöver nämnas om. Ett exempel var algoritmen för primtalsfaktorisering i förra avsnitt (sid 131). Ett annat skäl är att de genererar kort och elegant kod som baseras på s.k. *rekursionsformler* – ett matematiskt koncept som ofta är enkelt att programmera. Ett exempel på problem som kan lösas med sådana formler är *Fibonaccis problem*:

År 1202 formulerade den italienske matematikern *Leonardo Pisano Fibonacci* i sin bok *Liber abaci* (Boken om räknekonsten) följande uppgift som lustigt nog handlar om kaniners fortplantning. Han observerade följande:

> 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:

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 konstrueras de enligt regeln ovan:

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. …

Det blir 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 smartare lösning vore att upptäcka ett mönster, en struktur, t.ex. ett samband mellan antal månader och kaninpar, en slags laglighet i bildandet av fibonaccitalen. Ett sådant samband skulle kunna beskrivas i form av en algoritm för att sedan kunna implementeras i kod.

Undersöker man tabellen ovan noga träder fram följande enkelt mönster: Summan av två på varandra följande fibonaccital ger nästa fibonaccital. För att beskriva detta mönster inför vi beteckningarna:

n = Antalet månader
F_n = Antalet kaninpar i månaden n

Mönstret vi upptäckte ovan kan beskrivas med följande matematisk formel.

Fibonaccis rekursionsformel:

$$F_1 = 1,$$
$$F_2 = 1$$
$$F_n = F_{n-1} + F_{n-2} \quad \text{för } n = 3, 4, 5, \ldots$$

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. Men vad är det rekursiva i formeln ovan? I en 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 fibonacci-
// talet. Rekursiv därför att metoden anropar sig själv

class Fibonacci
{
   public static int 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()`**. 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\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\t");    // Radbyte
    }
    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, speciellt när de baseras på rekursionsformler. T.ex. pratar man i Fibonacci rekursionen om exponentiellt växande tidskomplexitet av typ 2^n för att beräkna **`Fib(n)`**. Dvs tidsåtgången växer med en faktor 2^n, dvs exponentiellt, där n är antalet element i talföljden. 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 iterativa algoritmer, t.ex. sortering. Man borde alltså avväga från fall till fall om rekursiv eller iterativ metod ska användas.

Collatz algoritmen, rekursiv

Vi vill inte avsluta behandlingen av rekursion utan ett exempel som *inte* baseras på en rekursionsformel. Därför ska vi ta upp en algoritm som vi redan behandlat iterativt tidigare, för att beskriva den rekursivt. Collatz algoritmen lämpar sig för att visa att den ovannämnda tidskomplexiteten kan undvikas, när för det första beräkningarna inte bygger på en rekursionsformel och för det andra algoritmen genererar en avtagande, inte en stigande talföljd.

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

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

Det visar sig att talföljderna i denna algoritm alltid slutar med 1 oavsett startvärde. Dock är detta påstående matematiskt hittills obevisat [*].

Tidigare har Collatz algoritmen implementerats *iterativt* dvs med en loop, både i modulär och icke-modulär version. Den modulariserade varianten har dessutom be-

[*] Man kan testa Collatz algoritmen i appen *Mattekollen* där den är kodad i Python. Ladda ned appen eller kör den som Webbapp: **app.mattekollen.se → En mobil pythonmiljö**. Eller kör den direkt som webbapp: beta.mattekollen.se/#/app/coding. Kör koden med olika startvärden för att kolla om algoritmens talföljder alltid slutar med 1. I denna variant är Collatz algoritmen förstås inte rekursiv utan iterativ, närmare bestämt kodad med en **`while`**-loop. Du kan nu gärna försöka att koda en rekursiv version i Python i denna miljö.

handlats både som **void**-metod och formulerats som metod med returvärde i en övningsuppgift. Följande klass definierar nu algoritmen som en *rekursiv* metod:

```
// Collatz_reku.cs
// Definierar metoden Collatz() rekursivt.
// Får ett startvärde som parameter, tar det gånger 3 + 1,
// om det är udda, annars delar det med 2.
// Avslutar när talföljden har nått 1.
using System;
class Collatz_reku
{
   public static int Collatz(int n)      // Rekursiv metod
   {
      Console.Write(n + "\t");
      if (n == 1)
         return n;
      else
      {
         if (n % 2 == 1)
            return Collatz(3 * n + 1);  // Rekursivt anrop
         else
            return Collatz(n / 2);      // Rekursivt anrop
      }
   }
}
```

Metoden **Collatz()** som är definierad i klassen **Collatz_reku** ovan, är rekursiv därför att den anropar sig själv i sin egen definition. De rekursiva anropen står i den nästlade **if-else**-satsens **return**-satser. I själva verket skickas i dessa anrop de nya elementen i algoritmens talsekvens till metoden igen, efter att de bestämts enligt algoritmens regler. På så sätt simulerar de rekursiva anropen loopen som fanns i de tidigare iterativa implementationerna.

Vi testar den rekursiva metoden **Collatz()** ovan i följande klass:

```
// Collatz_reku_Test.cs
// Läser in ett startvärde och skickar det till den rekursiva
// metoden Collatz() definierad i klassen Collatz_reku.
using System;

class Collatz_reku_Test
{
   static void Main()
   {
      Console.Write("\n\tMata in ett positivt heltal:\t");
      int number = int.Parse(Console.ReadLine());  // Start-
      Console.Write("\n\t");                       // värde

      Collatz_reku.Collatz(number);                // Anrop
```

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

Körexemplet nedan skiljer sig förstås inte från de tidigare testerna med de iterativa lösningarna:

```
        Mata in ett positivt heltal:    135

135     406     203     610     305     916     458     229
688     344     172     86      43      130     65      196
98      49      148     74      37      112     56      28
14      7       22      11      34      17      52      26
13      40      20      10      5       16      8       4
2       1
```

Övningar till kapitel 4

4.1 Följande algoritm i pseudokod beräknar summan av de första *n* positiva heltalen:

```
Start Sum(n)
    OM n = 1
        returnera 1
    ANNARS
        returnera n + Sum(n-1)
Slut Sum(n)
```

Vi vill låta datorn beräkna: Sum(10) = 1 + 2 + 3 + . . . + 9 + 10
Sum(100) = 1 + 2 + 3 + . . . + 99 + 100 osv.

a) Implementera algoritmen som en metod i C#. Bygg metoden in i ett program och anropa den för n = 10, 100, 1000 och 10 000. Skriv ut resultaten på ett användarvänligt sätt.

b) Är algoritmen (metoden) rekursiv? Om ja, var finns rekursiviteten? Förklara!

c) I vilken ordning adderar algoritmen de pos. heltalen, fram- eller baklänges? Förklara varför den gör så.

4.2 Skriv en rekursiv metod **`Faculty()`** som implementerar den matematiska funktionen *n Fakultet*: $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$

Testa metoden i en klass genom att anropa den för n = 1, 2, 3, … , 12.
Skriv ut resultaten på ett användarvänligt sätt.
Kontrollera dina resultat med en miniräknare.

4.3 Skriv om den rekursiva metoden **`Factorize()`** (sid 132) som implementerar faktoriseringen av primtal, till en *iterativ* metod. Dvs ersätt rekursionen med en vanlig loop, t.ex. **`while`**, och testa den nya metoden genom att anropa den i programmet **`PrimeFactors`** (sid 132).

Kapitel 5

Windowsprogrammering

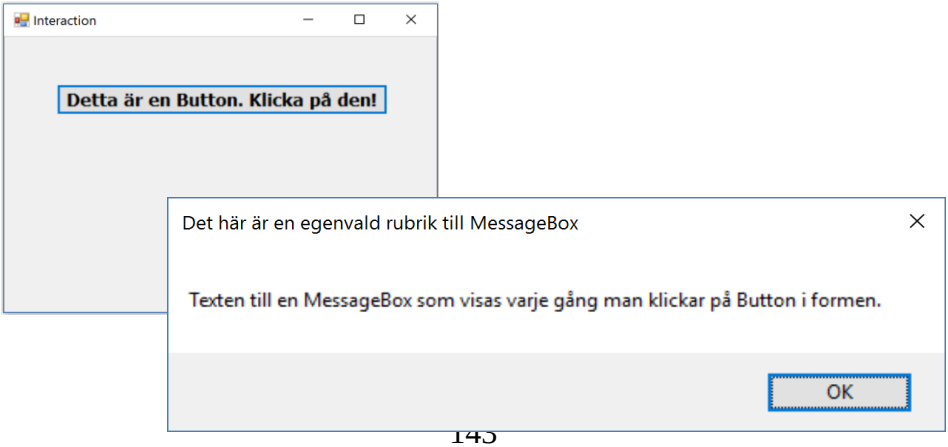
Ämne	Sida	Program
5.1 Interaktiva grafiska gränssnitt	143	`Interaction`
- Controls	144	
- Windows Forms Applications	144	
- Händelsemetoder	148	
5.2 TextBoxar, Buttons & Labels	150	`PassWTextBox`
5.3 Checkboxar och radioknappar	152	`Bartender`
5.4 En egen webbläsare	156	
- En första webbläsare	158	`FirstBrowser`
5.5 En mer utvecklad webbläsare	160	`DevBrowser`
- Dialogrutan Navigate	161	
Övningar till kapitel 5	167	

5.1 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*.

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 innehåller en knapp (Button). Först när man klickar på den (händelse) får man en meddeladeruta (MessageBox) till höger:



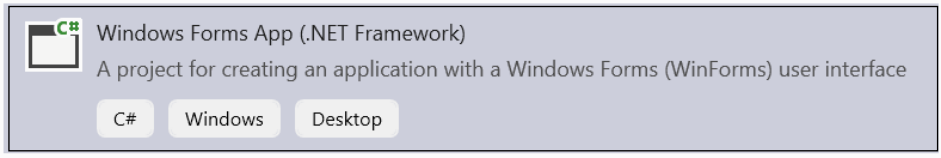
Controls

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* i Visual Studio.

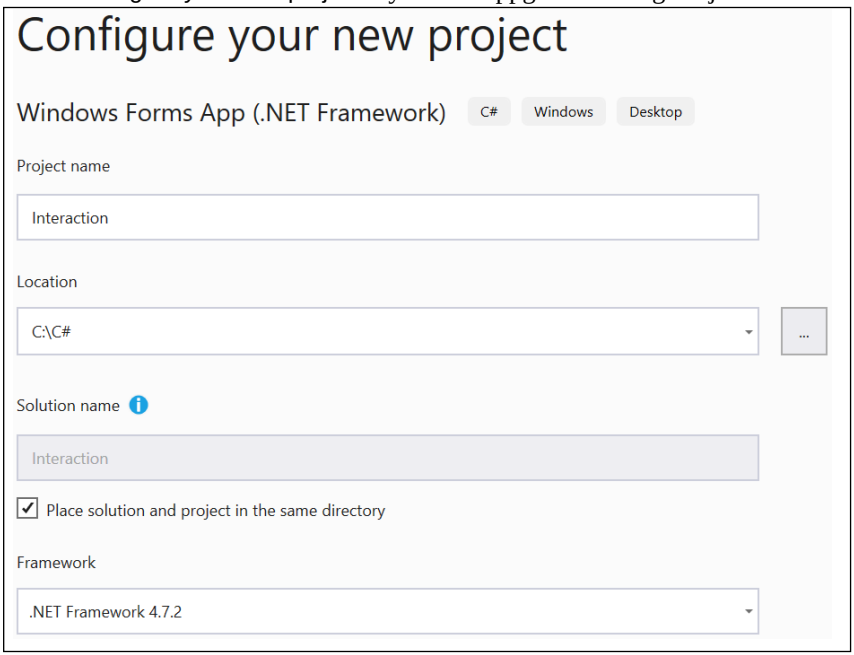
Windows Forms Application

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:

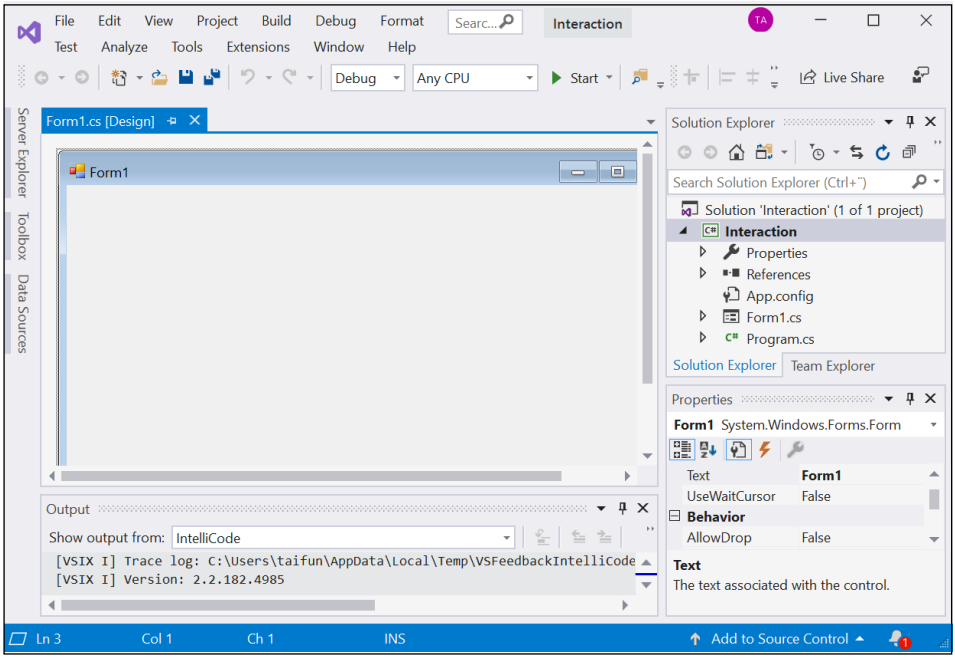


Markera rutan ovan. Klicka sedan i dialogrutan Create a new project som omfattar denna ruta, på knappen Next längst ned till höger. En ny dialogruta dyker upp med rubriken Configure your new project. Fyll i den uppgifterna enligt följande:



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.

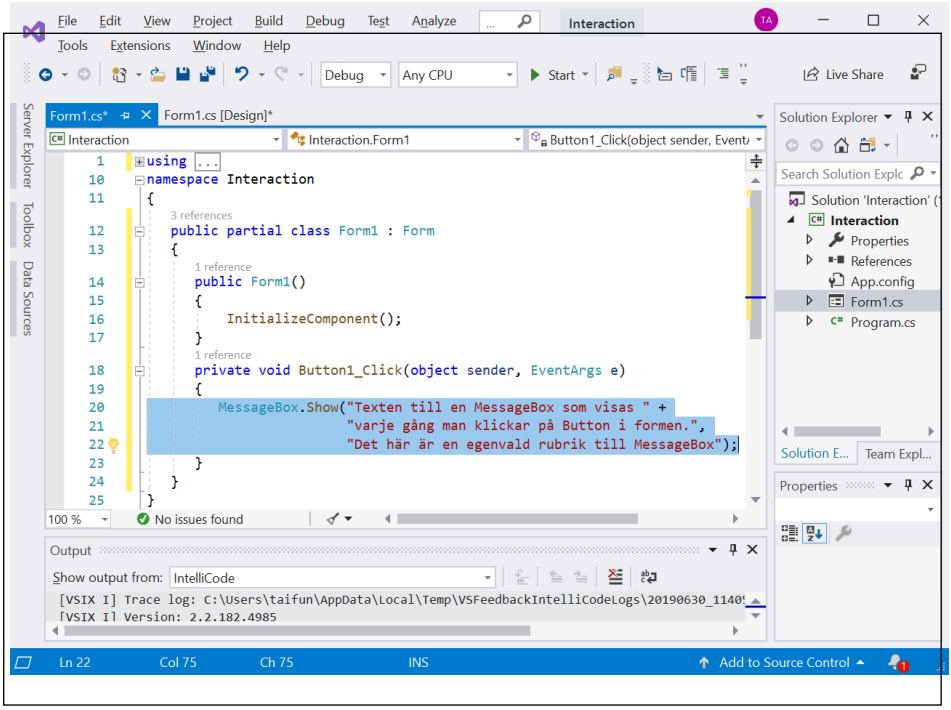
Gå till menyraden längst upp och välj menyn: View → Toolbox
Expandera Common Controls och dubbelklicka på kontrollen Button, så att den hamnar i formfönstret. När du flyttar markören till formen stängs Toolbox-fönstret. Markera den nya kontrollen button1 på din form för att få fram dess egenskaper i 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:

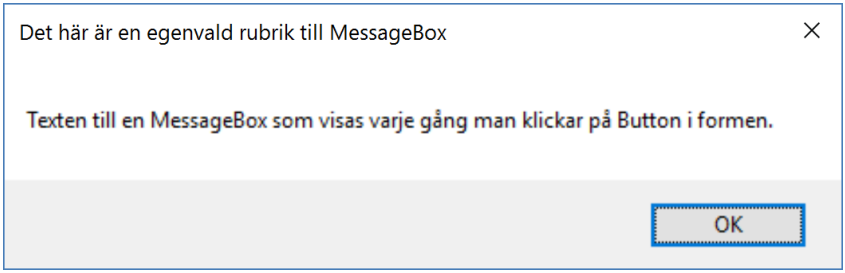
Egenskap	Värde
AutoSize	True
Font	Tahoma; 12pt; style=Bold
Location	110; 100
Text	Detta är en Button. Klicka på den!

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. Ursprung-

ligen 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 146), 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

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

(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.

5.2 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 143). 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 PassWTextBox.

Gör så här för att skapa applikationen:

- Skapa en *Windows Forms Application* och döp den till PassWTextBox. Hur man gör har vi lärt oss i projektet Interaction (sid 143).

  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	PassWTextBox
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;
```

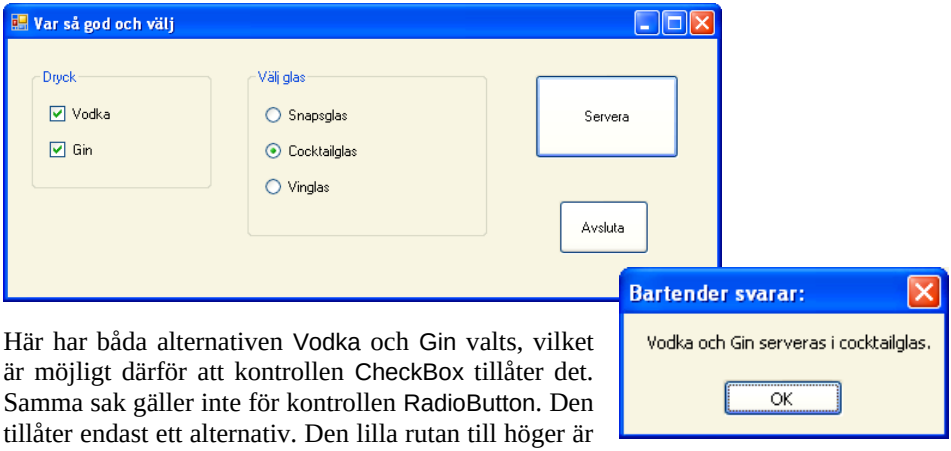
- Kompilera och kör. Skriv något i textboxen. Det visas bara stjärnor. Klickar du på Show Me-knappen visas texten i labeln.

5.3 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-grupp-boxen 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.

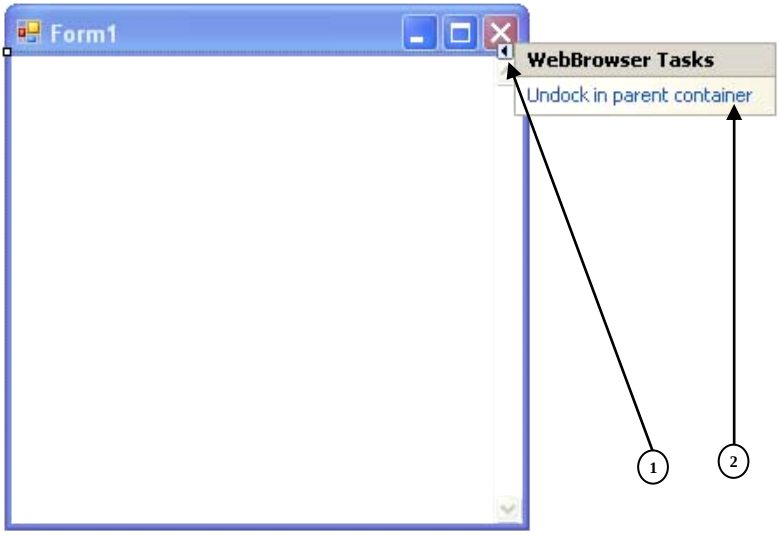
5.4 En egen webbläsare

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 FirstBrowser. 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:

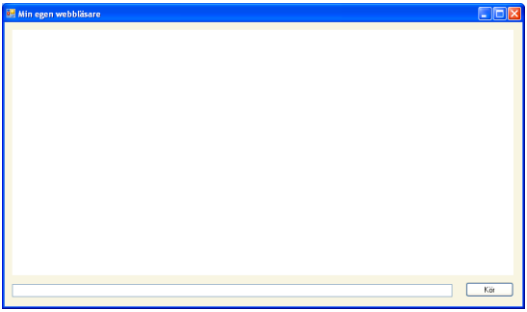
Egenskap	Värde
(Name)	tbURL
Location.X	12
Location.Y	650
Size.Width	1020
Size.Height	26

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 88, polymorfism, sid 50). 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

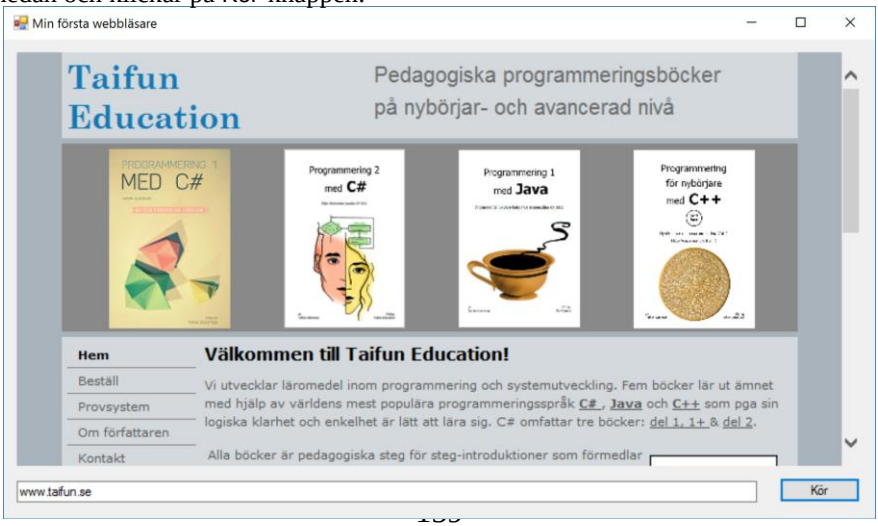
```
private void btnGo_Click(object sender, EventArgs e)
```

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 FirstBrowser, skriver en webbadress i textfältet nedan och klickar på Kör-knappen:



5.5 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.

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. Markera formen. 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 FirstBrowser, 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 **`NavigateToolStripMenu-Item_Click()`**.

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

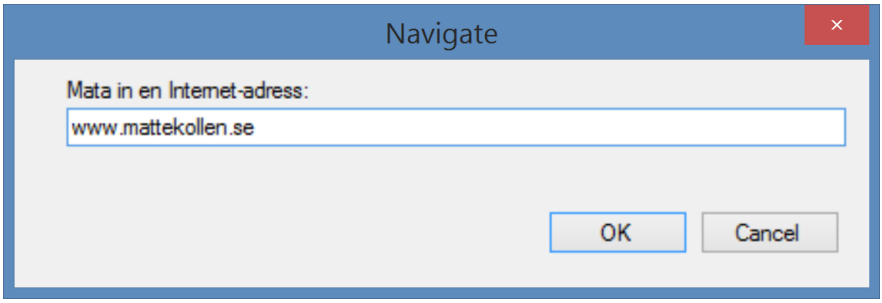
```
Navigate myNavigateBox = new Navigate();
```

    Därmed skapar du ett objekt av typ **`Navigate`** och döper det till **`myNavi-gateBox`**.

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);
         }
      }
   }
}
```

Övningar till kapitel 5

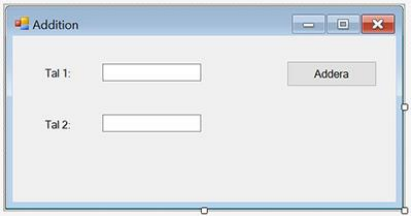
5.1 Tänk dig (eller skriv) en Console Application, kallad AdditionC, som med hjälp av heltalsvariabler producerar följande utskrift till konsolen:

```
Summan av 9 och 2 är 11
```

9 och **2** ska vara de värden som två av de heltalsvariablerna inirierats till.

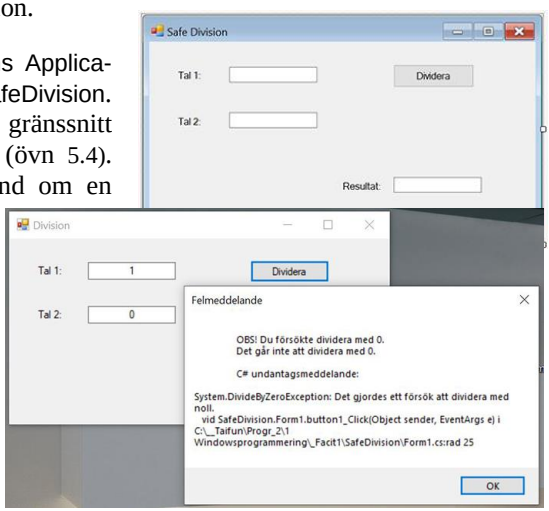
5.2 Skapa en Windows Forms Application och kalla den för AdditionW. Den ska göra samma sak som lösningen i övning 5.1 (AdditionC), bara att utskriften inte hamnar i konsolen utan i en MessageBox. Den ska visas när man klickar på en knapp med texten *Visa MessageBox* i formfönstret. Förse MessageBoxen med rubriken *Windows Addition*.

5.3 I både övn 5.1 och 5.2 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.

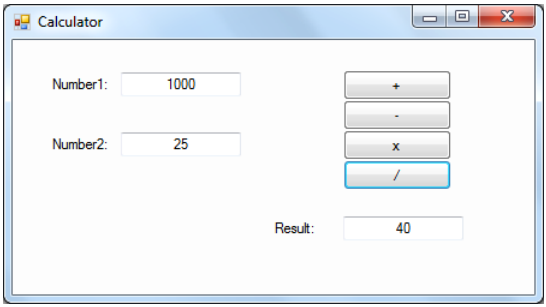


5.4 Skapa en Windows Forms Application och kalla den Division. Modifiera lösningen i övn 5.3 så att beräkningens resultat inte skrivs ut till en MessageBox utan placeras i ett textfält som läggs till i formen. Välj den här gången division som räkneoperation.

5.5 Skapa en Windows Forms Application och kalla den för SafeDivision. Skapa samma grafiska gränssnitt som i projektet Division (övn 5.4). Applikationen ska ta hand om en eventuell division med 0. Modifiera koden i Form1.cs genom att hantera fallet att användaren vid division matar in 0 i det andra textfältet. Styr meddelandena från felhanteringen till en MessageBox.



5.6 Vidareutveckla övningsserien 5.1-5.5 (ovan) till en komplett kalkylator som inkluderar de fyra räknesätten. Det grafiska gränssnittet kan se ut som bilden till höger. Förse divisionen med en egen felhantering som förhindrar division med 0 och skriver ut ett felmeddelande till en MessageBox.



5.7 ***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.

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 Objektorienterad programmering (OOP), sid 57:

Ovn_1_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_1_2

Modularisera programmet All_in_Main från övn 1.1 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_1_3_Class

Modularisera programmet All_in_Main från övn 1.1 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_1_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_1_5_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_1_5_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
                                              // Rektangel skapas
                                              // och klassens kon-
                                              // struktor anropas
      Console.WriteLine("\n\tArea    = " + rekt.Area()    + '\n' +
                  "\n\tOmkrets = " + rekt.Circumference()  + '\n');
   }
}
```

Ovn_1_6_Class
Modifiera klassen Rectangle från övn 1.5 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_1_6_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 = Int16.Parse(Console.ReadLine());

      Console.Write("\n\tMata in bredd:\t");
      int no2 = Int16.Parse(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_3_1_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_3_1_SuperClass

Förse superklassen Circle med en privat datamedlem radius, en konstruktor, en getmetod & med metoderna 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_3_1_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 = Int16.Parse(Console.ReadLine());

     Console.Write("\n\tMata in höjd:\t");
     int no2 = Int16.Parse(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");
   }
}
```

// Ovn_3_2.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_3_2
//           {
//              static void Main()
//              {
//                 {
//                    int t = 30;
//                 }
//                 Console.WriteLine("t = " + t);
//              }
```

```
//            }

using System;
class Ovn_3_2
{
   static void Main()
   {
      int t;
      {
         t = 30;
      }
      Console.WriteLine("\n\tt = " + t + '\n');
   }
}
/* Kompileringsfelet i programmets första variant berodde på att va-
   riabeln t var definierad i ett inre block och att programmet re
   fererade till den utanför det inre blocket där t inte längre var
   giltig.
*/
**********************************************************************
```

Ovn_3_3_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 objektorien-terad. OBS! Följande lösningsförslag visar endast den optimala va-rianten.

```
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_3_Class
Separat fil i samma projekt som filen Ovn_3_3_Test

```
class Cube
{
   public double side;

   public double Volume()
   {
      return side * side * side;
   }

   public double Surface()
   {
      return 6 * side * side;
   }
}
```

Ovn_3_4a
Modularisera programmet Non_modularized_1 (sid 112) 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_4a
{
   static void Main()
   {
      Console.Write("\n\tMata in ett heltal:\t\t");       // Ledtext
      int no1 = Int16.Parse(Console.ReadLine());      // Input
      Console.Write("\n\tMata in ett heltal till:\t");
      int no2 = Int16.Parse(Console.ReadLine());
      Console.WriteLine("\n\n\t"                                    +
         no1 + " plus "  + no2 + " är "  + (no1 + no2) + "\n\t"    +
         no1 + " minus " + no2 + " är "  + (no1 - no2) + "\n\t"    +
                                    // Anropet:
         no1 + " gånger " + no2 + " är " + Mult(no1, no2) + "\n\t" +
         no1 + " heltalsdividerad med "   +
                        no2 + " är " + (no1 / no2) + "\n\t"   +
         no1 + " modulo " + no2 + " är " + (no1 % no2) + "\n\t");
   }
// ------------------------------------------------------------------
// Metoden Mult() som tar in två heltal via sina parametrar a och b
// och returnerar ett heltal som är a * b
   static int Mult(int a, int b)                  // Metoden Mult()
   {
      return a * b;
   }
// ------------------------------------------------------------------
}
```

Ovn_3_4b

Modularisera programmet Non_modularized_1 (sid 112) 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_4b
{
   static void Main()
   {
      Console.Write("\n\tMata in ett heltal:\t\t");     // Ledtext
      int no1 = Int16.Parse(Console.ReadLine());    // Inläsning
      Console.Write("\n\tMata in ett heltal till:\t");
      int no2 = Int16.Parse(Console.ReadLine());
      Console.WriteLine("\n\n\t"                                          +
        no1 + " plus "   + no2 + " är " + (no1 + no2) + "\n\t"     +
        no1 + " minus " + no2 + " är "  + (no1 - no2) + "\n\t"     +
        no1 + " gånger " + no2 + " är " +      // Anropet:
                                  Multip.Mult(no1, no2) + "\n\t" +
        no1 + " heltalsdividerad med "  +
                           no2 + " är " + (no1 / no2) + "\n\t"     +
        no1 + " modulo " + no2 + " är " + (no1 % no2) + "\n\t");
   }
}

// Ny klass Multip i samma fil som Ovn 3 3b
class Multip                                          // Klassen Multip()
{
   public static int Mult(int a, int b)              // Metoden Mult()
   {
      return a * b;
   }
}
```

Ovn_3_4c

Modularisera programmet Non_modularized_1 (sid 112) 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_3c
{
   static void Main()
   {
      Console.Write("\n\tMata in ett heltal:\t\t");
      int no1 = Int16.Parse(Console.ReadLine());
      Console.Write("\n\tMata in ett heltal till:\t");
```

```
      int no2 = Int16.Parse(Console.ReadLine());
      Console.WriteLine("\n\n\t"                                        +
         no1 + " plus "   + no2 + " är " + (no1 + no2) + "\n\t"    +
         no1 + " minus " + no2 + " är "  + (no1 - no2) + "\n\t"    +
         no1 + " gånger " + no2 + " är " +      // Anropet:
                                  Multip.Mult(no1, no2) + "\n\t" +
         no1 + " heltalsdividerad med "   +
                        no2 + " är " + (no1 / no2) + "\n\t"     +
         no1 + " modulo " + no2 + " är " + (no1 % no2) + "\n\t");
   }
}
```

Ovn_3_4c

Separat fil som borde ligga i samma projekt som filen Ovn_3_3c och när programmet Ovn_3_2d körs, i samma projekt som filen Ovn_3_3d

```
class Multip                                          // Klassen Multip
{
   public static int Mult(int a, int b)              // Metoden Mult()
   {
      return a * b;
   }
}
```

Ovn_3_4d

Modularisera programmet Non_modularized_1 (sid 112) 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_4d
{
   static void Main()
   {
      Console.Write("\n\tMata in ett heltal:\t\t");
      int no1 = Int16.Parse(Console.ReadLine());
      Console.Write("\n\tMata in ett heltal till:\t");
      int no2 = Int16.Parse(Console.ReadLine());
      Console.WriteLine("\n\n\t" +                    // Anropen
      no1 + " plus "  + no2 + " är " + Addit.Add(no1, no2) + "\n\t" +
      no1 + " minus " + no2 + " är " + Subtr.Sub(no1, no2) + "\n\t" +
      no1 + " gånger "+ no2 + " är " +      // Anropet:
                                  Multip.Mult(no1, no2) + "\n\t" +
      no1 + " heltalsdividerad med "                                 +
                  no2 + " är " + Div.IntDiv(no1, no2) + "\n\t"    +
      no1 + " modulo " + no2 + " är " + Modu.Mod(no1, no2) +"\n\t") ;
   }
}
```

Ovn_3_4dA
Separat fil i samma projekt som filen Ovn_3_3d

```
class Addit                                          // Klassen Addit
{
   public static int Add(int a, int b)              // Metoden Add()
   {
      return a + b;
   }
}
```

Ovn_3_4dS
Separat fil i samma projekt som filen Ovn_3_3d

```
class Subtr                                          // Klassen Subtr
{
   public static int Sub(int a, int b)              // Metoden Sub()
   {
      return a - b;
   }
}
```

Ovn_3_4dD
Separat fil i samma projekt som filen Ovn_3_3d

```
class Div                                           // Klassen Div
{
   public static int IntDiv(int a, int b)          // Metoden IntDiv()
   {
      return a / b;
   }
}
```

Ovn_3_4dM
Separat fil i samma projekt som filen Ovn_3_3d

```
class Modu                                           // Klassen Modu
{
   public static int Mod(int a, int b)               // Metoden Mod()
   {
      return a % b;
   }
}
```

Ovn_3_5
Modularisera programmet Non_modularized_2 (sid 113) 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_5
{
   static void Main()
   {
      /* I n m a t n i n g */
      Console.Write("\n\tAnge antal år:\t\t");
      int years = Int16.Parse(Console.ReadLine());

      Console.Write("\n\tAnge antal månader:\t");
      int months = Int16.Parse(Console.ReadLine());

      Console.Write("\n\tAnge antal veckor:\t");
      int weeks = Int16.Parse(Console.ReadLine());

      Console.Write("\n\tAnge antal dagar:\t");
      int dag = Int16.Parse(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_6a

a) Vänd om problemet från övn 3.5: 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 %

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 = Int16.Parse(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_6b

b) Modularisera programmet Non_modularized_3 (Ovn_3_5a) 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_6b
{
   static void Main()
   {
      /* I n m a t n i n g */
      Console.Write("\n\tAnge antal dagar:\t");
      int totalDays = Int16.Parse(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_7_Test
Modularisera programmet Non_modularized_3 (sid 181) 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 = Int16.Parse(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_7_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 Tillämpningar, sid 141:

Ovn_4_1
Följande pseudokod beräknar summan av de första n positiva heltalen:

```
             Start Sum(n)
                 OM n = 1
                    returnera 1
                 ANNARS
                    returnera n + Sum(n-1)
             Slut Sum(n)
Vi vill Beräkna: Sum(10)  = 1 + 2 + . . . + 9 + 10
                 Sum(100) = 1 + 2 + 3 + . . . + 99 + 100  osv.
```

a) Implementera algoritmen som en metod i C#. Bygg metoden in i ett program och anropa den för n = 10, 100, 1000 och 10 000. Skriv ut resultaten på ett användarvänligt sätt.
b) Är metoden rekursiv? Om ja, var finns rekursiviteten? Förklara!
c) I vilken ordning adderar algoritmen de positiva heltalen, fram- eller baklänges? Förklara varför den gör så.

a)

```
using System;
class Ovn_1_9
{
   static int Sum(int n)
   {
      if (n == 1)
         return 1;
      else
         return n + Sum(n - 1);
   }

   static void Main()
   {
      for (int i = 1; i < 5; i++)
          Console.WriteLine("\n\tSumman 1 + 2 + ... + " +
          Math.Pow(10, i) + " = " + Sum((int) Math.Pow(10, i)));
      Console.WriteLine();
   }
}
```

b)
Metoden **Sum(n)** är rekursiv därför att den anropar sig själv med **Sum(n - 1)** i sista raden.

c)
Algoritmen summerar de positiva heltalen baklänges, t.ex. om n = 10 :

10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1

Det framgår av uttrycket **n + Sum(n - 1)** som returneras i sista raden.
Algoritmen gör så för att få stopp på rekursionen när **n** har nått **1**.

****

Ovn_4_2
Skriv en rekursiv metod **Faculty()** som implementerar den matematiska funktionen n Fakultet n! = 1· 2 · 3· . . . · n
Testa metoden i en klass genom att anropa den för n = 1, 2, 3,…, 12. Skriv ut resultaten på ett an¬vändarvänligt sätt. Kontrollera dina resultat med en miniräknare.

```
using System;
class Ovn_3_10
{
   static int Faculty(int n)
   {
      if (n == 1)
         return 1;
      else
```

```
        return n * Faculty(n - 1);
    }

    static void Main()
    {
        for (int i = 1; i < 13; i++)
            Console.WriteLine("\n\t" + i + "! = " + Faculty(i));
            Console.WriteLine();
    }
}
```

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

Ovn_4_3

Skriv om den rekursiva metoden **Factorize()** (sid 132) som implementerar faktoriseringen av primtal, till en iterativ metod. Dvs ersätt rekursionen med en vanlig loop, t.ex. **while**, och testa den nya metoden genom att anropa den i programmet **PrimeFactors** (sid 132).

```
using System.Collections.Generic;
class Ovn_4_3
{
  public static void FactorIt(int n, List<int> t)  // Iterativ metod
  {
    int rest = 0;
    int k;
    while (rest == 0)                    // Loop som erätter rekursion
    {
       k = 1; rest = 1;
       while (k <= n - 2 && rest != 0) // Leta efter primfaktor k
       {
          k++;
          rest = n % k;                 // Dela n med k och ta resten
       }
       if (rest != 0)                   // k delar inte n => n primtal:
          t.Add(n);                     // Lägg n i listan och avsluta
       else
       {                                // k delar n:
          t.Add(k);                     // Lägg primfaktorn k i listan
          n = n / k;                    // Ta bort faktorn k från n:
       }                                // Börja om loopen (Iteration)
    }
  }
}
```

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

Kapitel 5 Windowsprogrammering, sid 167:

Övning 5.1

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 5.2
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 5.3
I både övn 5.1 och 5.2 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 5.4
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 & 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 5.5
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 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 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 & 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 5.6
Vidareutveckla övningsserien 5.1-5.5 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
      }
   }
}
```