Algoritmer, data-strukturer & design patterns

Med C#, relationsdatabaser och SQL

Med övningar,
fullständiga lösningar
&
projektuppgifter

TechPages Förlag AB

Innehåll

Ämne	Sida	Program/Algoritm
Kapitel 1 Algoritmer och programmering	**5**	
1.1 Programmeringens historia	6	
- Från maskinkod till Assembler	6	
1.2 Olika paradigm inom programmering	11	
- Paradigmskifte	14	
1.3 Algoritmer och deras beskrivning	15	
- Historiens första algoritm	15	
- Definition och exempel på algoritmer	16	
- Olika sätt att beskriva algoritmer	18	
1.4 Traditionell design pattern med flödesschema	20	
- Pseudokod till algoritmen Morgonsyssla	20	`Morgonsyssla`
- Kontrollstrukturer i algoritmer	22	
- Flödesschema till algoritmen Morgonsyssla	23	
1.5 Tillägg av C# i Visual Studio	25	
1.6 C# Console Applications	26	
1.7 De enkla datatyperna i C#	31	`PrimitivesCs`
1.8 Inläsning av data	34	`InputCs`
- Metoden ReadLine()	35	
- Villkorlig initiering	36	`(Un)CondInit`
1.9 Collatz algoritmen	39	`Collatz`
- Metoder och program i C#	41	`Collatz_mod`
- Modularisering av `Collatz`	42	`Collatz_Test`
1.10 Algoritm för platsbyte	44	`MiniSort`
- Försök att modularisera `MiniSort`	45	`NoSort`
1.11 Parameteröverföring i metoder	48	
- Värdeanrop (Call by value)	48	`CallByVal`
- Referensanrop (Call by reference)	50	`CallByRef`
- Modularisering av `MiniSort`	42	`Swapping`
1.12 In- och utparametrar	53	`OutParam`
Övningar till kapitel 1	56	
Kapitel 2 Logik för blivande programmerare	**61**	
2.1 Logiska operatorer	62	`AND_OR`
- Sanningstabeller	64	
2.2 Datatypen `bool`	67	`TruthTab`
2.3 NEGATION som logisk operator	69	`GuessNEG`
- Logiska uttryck	71	
2.4 Programserien *Testa lösenord*	73	`Passwd`
- Kombination av NEGATION, OCH, ELLER	75	`PasswdCaps`
- De Morgans lagar	77	

Ämne	Sida	Program/Länk
2.5 Mängdlära och logik	79	*Mängder*
- Mängdoperationer och deras logik	79	
- Cartesisk produkt	83	
Övningar till kapitel 2	84	
Kapitel 3 Datastrukturer och abstrakta datatyper	**87**	
3.1 Vad är objektorienterad programmering?	88	
3.2 Objektorienterad design med UML	94	
- Projekt Lönespecifikation	94	
- Kundens kravspecifikation	94	
- UML design och modellering i fyra steg	94	
3.3 Array som objekt	98	`ArrayObj`
- **`foreach`**-satsen	102	
3.4 Hantering av array med referens	105	`ArrayRef`
3.5 Array av referenser	107	`ArrayOfRef`
3.6 Array som parameter i metoder	111	`ArrayParam`
3.7 Hantering av slumptal i C#	115	`DoRand`
- Array av slumptal	116	`RandArray`
3.8 Sökning och sortering	118	`Search`
- Bubbelsortering	121	`Bubble`
3.9 Generiska metoder	124	`G_Bubble`
3.10 Listor	129	`Lista`
- Klassen **`RandList`**	130	`RandList`
- **`foreach`** i listor	131	`Print`
Övningar till kapitel 3	133	
Kapitel 4 Tillämpningar	**135**	
4.1 Kryptering av strängar	136	`EncryptStr`
4.2 Kryptering av text, teckenvis	139	`EncryptChar`
4.3 Filhantering	142	`WriteReadFile`
- Append	145	`AppendFile`
4.4 Slumplösenord	147	`RandPasswdTest`
4.5 Kryptering av filer	151	`EncryptFile`
Övningar till kapitel 4	156	
Kapitel 5 Datastrukturer i relationsdatabaser	**158**	*Databaser*
5.1 Introduktion till databaser	159	
5.2 Relationsdatabaser	161	
- Modularisering	161	
- Liknelse med klass och objekt	163	
- Vad är en relation i databaser?	164	
- Primär- och främmande nycklar	168	

Ämne	Sida	Program/Länk
5.3 Introduktion till SQL	169	
- Databashanterare	169	
- Klient – Server-modellen	170	
- SQL – databasers språk	172	
- SELECT-satsen	173	
- CREATE TABLE-satsen	178	
5.4 Vår första SQL Server databas	180	**FirstDatabase**
- Att koppla upp sig till SQL Servern	181	
- Att visa databasens innehåll	184	
5.5 En SQL klient i C#	186	**SQLclient**
- Att skriva och exekvera egna SQL satser	188	
- Grafiskt gränssnitt till SQL klienten	193	
5.6 Att skapa och designa en databas i C#	198	**Kursverksamhet**
- Databasmodellering	199	
- Att skapa databasen Kursverksamhet	199	
- Att skapa tabeller i databasen	200	
- Att koppla projektets Dataset till databasen	203	
- Att skapa relationer mellan tabeller	206	
- Att lägga in data i tabellerna	208	
5.7 Att förse databasen med funktionaliteter	211	**AddressBook**
Övningar till kapitel 5	217	
Fullständiga lösningar till alla övningar (Facit)	**223**	
Projektuppgifter		
• Kalkylatorn	59	
• Kryptering av databas	157	
• Human resources	219	
• Kaffeautomaten	220	
Programförteckning	**246**	
Register	248	

Kapitel 1

Algoritmer och programmering

	Ämne	Sida	Program/Algoritm
1.1	Programmeringens historia	6	
	- Från maskinkod till Assembler	6	
1.2	Olika paradigm inom programmering	11	
	- Paradigmskifte	14	
1.3	Algoritmer och deras beskrivning	15	
	- Historiens första algoritm	15	
	- Exempel på algoritmer	16	
	- Definition av algoritm	17	
	- Olika sätt att beskriva en algoritm	18	
1.4	Design pattern med flödesschema	20	
	- Pseudokod till algoritmen Morgonsyssla	20	**Morgonsyssla**
	- Kontrollstrukturer i algoritmer	22	
	- Flödesschema till algoritmen Morgonsyssla	23	
1.5	Tillägg av C# i Visual Studio	25	
1.6	De enkla datatyperna i C#	31	**PrimitivesCs**
1.7	Inläsning av data	34	**InputCs**
	- Metoden ReadLine()	35	
	- Villkorlig initiering	36	**(Un)CondInit**
1.8	Collatz algoritmen	39	**Collatz**
	- Metoder och program i C#	41	**Collatz_mod**
	- Modularisering av **Collatz**	42	**Collatz_Test**
1.9	Algoritm för platsbyte	44	**MiniSort**
	- Försök att modularisera **MiniSort**	45	**NoSort**
1.10	Parameteröverföring i metoder	48	
	- Värdeanrop (Call by value)	48	**CallByVal**
	- Referensanrop (Call by reference)	50	**CallByRef**
	- Modularisering av **MiniSort**	42	**Swapping**
1.11	In- och utparametrar	53	**OutParam**
Övningar till kapitel 1		56	

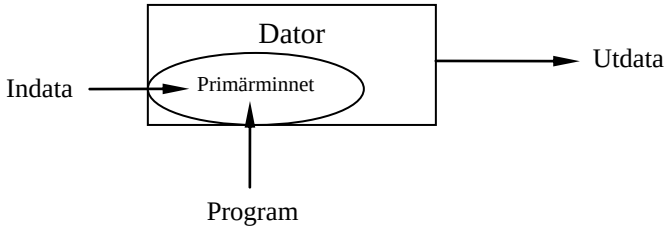
1.1 Programmeringens historia

Programmeringens historia skulle kunna fylla en hel bok. Vi får nöja oss med ett urval, de mest kända programspråken. Denna framställning gör alltså inte något anspråk på fullständighet. Men samtidigt ska den förklara varför det finns flera hundra olika programspråk. Det är nämligen *funktionaliteten* som är avgörande.

Från vävstolarna till John von Neumann

Redan på 1800-talet programmerade man vävstolarna med jättelika slags trähålkort – en form av manuell programmering. Speldosor av olika slag vars melodier är förprogrammerade och stansade i cylinderformiga metalltrummor som rullar över en spik (1800-talets iPhones!), är ett annat exempel på manuell programmering. Även när de första datorerna konstruerades på 1930/40-talet, skedde all programmering manuellt. Man matade de stora maskinerna med både information (data) och instruktion (program) för att åstadkomma en liten beräkning. Dessa jätteapparater med en bråkdel av datorkraften hos en modern PC – en av dem: 35 ton och 16 meter lång – kunde lagra endast *data*. Men att även kunna lagra *instruktioner*, var inte löst än.

Den tekniska innovation som inledde programmeringens historia i modern bemärkelse var *John von Neumanns* datormodell: **1944** lyckades han konstruera en dator som kunde lagra både *data* och *instruktioner*. De matades in via hålkort och kunde sedan bearbetas och t.o.m. ändras i datorn. John von Neumann-modellen såg ut så här:



John von Neumanns modell var ett genombrott i programmeringens historia. Än idag fungerar i princip exekvering av kod i datorns processor enligt denna modell: Kör man ett program laddas koden från hårddisken, där det lagrats i en fil, till datorns primärminne. Indata matas in från tangentbordet eller hämtas från en annan fil. I datorns processor bearbetas indata enligt programmets instruktioner, utdata produceras och matas ut om det önskas. Enda skillnaden från idag: då bestod instruktionerna av långa talkedjor som omvandlades till ettor och nollor, dvs man programmerade i *maskinkod*, ett språk som datorns processor förstod. Idag använder vi *källkod* i något programmeringsspråk.

Från maskinkod till Assembler

Så småningom kom man på idén att använda sig av kortkommandon på engelska som motsvarade instruktionerna i talform. Ett program tolkade sedan kommandona

till maskinkod. Programmet kallades *assembler* eller *assemblator*. Kortkommandona var de första nyckelorden av programmeringsspråket *Assembler*.

50-talet **Assembler** betecknas som *lågnivåspråk* eftersom det är nära *datorns* språk utan att vara maskinkod. Fördelen med Assembler är att det är snabbt. Än idag finns det ingen kod skriven av människan som kan köras på datorn snabbare. Nackdelen med Assembler är att det inte finns *ett* språk som heter så, utan varje processor har sitt *eget* assemblerspråk. Dvs program skrivet för en datortyp kan inte köras på en annan. På 40-talet var datorerna tekniska underverk, byggda för hand. Varje dator hade sin egen programmerare, oftast tillverkaren själv som var specialiserad på just sin maskins assemblerspråk. I längden var detta ohållbart. Lösningen var att komma bort från maskinberoende språk.

De första högnivåspråken

1957 **FORTRAN** = **FOR**mula **TRAN**slator är historiens första *högnivåspråk* i den bemärkelsen att det ligger nära *människans* språk. Avståndet till maskinkod är större än hos Assembler. Därför måste en källkod i Fortran först översättas till maskinkod. Denna översättning kallas *kompilering* och är mer invecklad än assemblering. Den nya maskinkod som direkt kan köras, är mycket större än källkoden och lagras separat på hårddisken. Fortran är till skillnad från Assembler ett kompilerande språk. Dessutom är det som namnet antyder, i första hand inriktat på beräkning av matematiska formler. Än idag används fortranprogram av ingenjörer och vetenskapsmän som behöver snabba beräkningar. Men det finns även administrativa tillämpningar av Fortran. Språket har utvecklats och marknadsförts av företaget IBM.

1959 **COBOL** = **CO**mmon **B**usiness **O**riented **L**anguage är, som namnet säger, specialiserat på administrativa och ekonomiska tillämpningar. Det kräver hantering av stora datamängder vilket Cobol är bra på. Många stora banker och försäkringsbolag har kvar sina program som en gång var skrivna i Cobol. Även om det numera finns modernare språk, håller man ofta fast vid det gamla pga de stora kostnader som ett byte skulle innebära. Även Cobol är ett högnivåspråk och därmed kompilerande. Cobol är utvecklat av USA:s försvarsdepartement i samarbete med den amerikanska datorindustrin.

1960 **ALGOL** = **ALGO**rithmic **L**anguage är det första språk som utvecklades i Europa. Det hade akademisk bakgrund: Initiativet låg hos det tyska *Gesellschaft für Angewandte Mathematik und Mechanik (GAMM)*. Man var ute efter ett verktyg för att utnyttja datorkraften för teknisk-vetenskapliga beräkningar på ett mer strukturerat sätt än Fortran. Beräkningarna skulle baseras på numeriska algoritmer snarare än matematiska formler. Algol som var ett kompilerande högnivåspråk, berikade programmeringen med många nya idéer och introducerade bl.a. *kontrollstrukturer* som används i

algoritmer. Dessa har tagits över och vidareutvecklats i de moderna programspråken. Algol själv används inte så mycket idag, inte minst pga brist på marknadsföring.

1963 **BASIC** = **B**eginners **A**ll-purpose **S**ymbolic **I**nstruction **C**ode är ett av de få högnivåspråk som inte är kompilerande utan *interpreterande*. Dvs källkoden tolkas rad för rad av datorns processor, utförs direkt och glöms bort sedan. Det uppstår ingen ny kod som lagras på hårddisken. Interpretering av källkod är alltid långsammare än exekveringen av redan kompilerad maskinkod. Däremot är interpretering snabbare än kompilering av källkod. I Basic finns inget kompileringssteg. Basic är, som namnet berättar, inriktat på att lära ut programmering för nybörjare. Därför har man hållit språket så enkelt som möjligt, så enkelt att man struntat i kontrollstrukturer som redan fanns i Algol och därmed lagt grunden för hoppsatser. Basic utvecklades ursprungligen av Dartmouth College i USA, men har sedan tagits över av Microsoft och integrerats som *QuickBasic* i DOS och Windows. På 90-talet har Microsoft lanserat vidareutvecklingen *Visual Basic* som blivit ett modernt och populärt utvecklingsverktyg. Den nyaste versionen heter *Visual Basic.NET* och är objektorienterad. I Visual Basic kan man även generera en exekverbar kod i efterhand genom att kompilera källkoden.

1971 **Pascal** är ingen förkortning för något utan har uppkallats efter *Blaise Pascal* som konstruerade räknemaskinen 1652. Pascal utvecklades av Niklaus Wirth på ETH (Eidgenössische Technische Hochschule) i Zürich. Tanken var att skapa ett kompilerande språk för att lära ut programmering för nybörjare genom att kombinera Basics enkelhet med Algols logiska strukturer och dess algoritmiska upplägg. På 80-talet utvecklade mjukvaruföretaget Borland *Turbo-Pascal* som blev en stor succé pga kompilatorns snabbhet och den *integrerade programutvecklingsmiljön (IDE)* som möjliggjorde kompilering, felsökning, editering och online hjälp i en och samma miljö. Idag marknadsför Borland Pascals objektorienterade vidareutveckling *Delphi*.

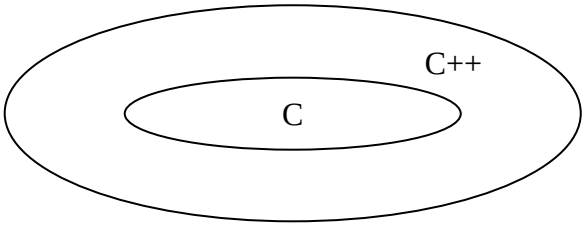
Från procedural (C) till objektorienterad programmering (C++)

70-80-talet **C++** är en direkt utvidgning och vidareutveckling av programmeringsspråket **C** som **1972** utvecklades av Dennis Ritchie på Bell Laboratories med syftet att skapa ett språk för programmering av operativsystemet *Unix*. I den bemärkelsen är C en biprodukt av Unix. Därför finns många logiska paralleller mellan C/C++ och Unix. Idag är inte bara Unix utan även andra operativsystem inkl. Windows skrivna i C/C++. Styrkan i C består av en kombination mellan enkelhet, strukturering och möjligheten att lätt kunna kommunicera med datorns hårdvara. C har bland de moderna språken den bästa förmågan att hantera och kontrollera hårdvaran, vilket favoriserar C som programspråk t.ex. för operativsystem. Den stora frihet som C erbjuder för hantering av bl.a. datorns primärminne med hjälp av *pekare*, kod

som ger åtkomst till den fysiska adressen till data och på gott och ont tillåter manipulationer av minnesadresser genom *pekararitmetik.*

Det var dansken Bjarne Stroustrup som la grunden till vidareutvecklingen av C. Under 70-talet hade man nämligen konstaterat att *procedural programmering* (Algol, Pascal, C, …) inte längre tillgodosåg alla krav som stora komplexa program ställde med avseende på underhåll, förnyelse och ändringsbarhet. Ingen kunde sätta sig in i, ändra och vidareutveckla ett stort program om programmeraren hade lämnat företaget. Det innebar ett enormt slöseri med resurser. Dessutom utvecklades hårdvaruteknologin så snabbt att program som kunde köras på de allt mer avancerade datorerna blev allt större och mer komplexa, speciellt när det gällde grafiska tillämpningar. Mjukvaruteknologin utvecklades inte alls i samma takt. För att lösa alla dessa problem uppkom den nya programmeringsfilosofin *objektorienterad programmering* (OOP) som en vidareutveckling av den traditionella *procedurala programmeringen.*

1983 presenterade Bjarne Stroustrup programmeringsspråket C++. Han bibehöll hela C och la till de nya objektorienterade elementen, bl.a. klassbegreppet, som hade redan funnits t.ex. i *Simula*, ett norskt programmeringsspråk från 1967 som i sin tur var en direkt utbyggnad av *Algol (Algorithmic language).* Simulas klasser hade ”glömts bort”. Den ovan beskrivna problematiken på 70-talet gjorde att man kom ihåg dem. Förhållandet mellan C och C++ illustrerar bäst den ”nya” filosofin tilläggskaraktär:



C är nämligen en delmängd av C++. Därför gäller all C-kod även i C++, men inte tvärtom. Med andra ord, en C++ kompilator kan kompilera all kod skriven i C, men inte tvärtom. Så den som lär sig C++ lär sig automatiskt C. Delmängdrelationen mellan C och C++ är unik bland högnivåspråken.

C++ är ett kraftfullt och populärt programmeringsspråk, vars styrka ligger på textbaserade konsolapplikationer. Inte att C++ vore olämpligt för grafiska tillämpningar, bara att det är lite jobbigt att skriva C++ kod för att åstadkomma grafik. En anledning är att, när C++ skapades, hade grafiska tillämpningar bara en begränsad spridning. Med utvecklingen av webben och dess grafiska miljö, med spridningen av Windows och grafiska användargränssnitt blev grafiken dominant. Idag har C++ fått en renässans med uppkomsten av IoT pga sin snabbhet och maskinnära egenskap.

90-talet **Java**s uppkomst motiverades av en annan utveckling inom IT som man skulle kunna kalla den grafiska eller Webbrevolutionen. Urspungligen har Java utvecklats av *Sun Microsystems* som ett projekt för att skapa ett språk för programmering av hushållsmaskiner. Men detta projekt visade sig vara en bubbla som sprack som mycket annat inom IT. Webben, som revolutionerade IT, blev räddaren i nöden för Java. Men Java är inte bara grafik och webb. Sun satsade på att utveckla Java till ett universellt objektorienterat språk som var plattformsoberoende. Idag används Java bl.a. för webbapplikationer, t.ex. *Java Server Pages (JSP).*

Sedan *Sun Microsystems* köpts upp av *Oracle*, är Java en Oracle-produkt. Oracle är en av världens ledande utvecklare av databashanterare. Java står inte i fokus av deras affärsverksamhet. Senaste tiden har Java tappat på popularitet inte minst pga sin lite krångliga kod jämfört med nyare utvecklingar som Python och C#.

90-talet **Python** skapades år 1989 av Guido van Rossum, en forskare på *National Research Institute for Mathematics and Computer Science* i Amsterdam och är en av de ovannämnda nyare utvecklingarna. Språket är *interpreterande* – liknande goda gamla BASIC – dessutom universellt. Python kan enkelt och gratis installeras på alla plattformar utan att man behöver bry sig om licenser. Koden är nästan självbeskrivande, ligger nära pseudokod och återspeglar algoritmen. I vissa avseenden är Python t.o.m. revolutionerande. Med små tekniska detaljer har man underlättat kodningen avsevärt. T.ex. har man avskaffat de obligatoriska symbolerna **{ }** för ett block. Det är inte längre nödvändigt att avsluta en sats med semikolon. De logiska indragningar som gör koden läsligare, har man lyft till obligatorisk syntax. Man är tvungen att följa god programmeringsstil. Variabler behöver inte explicit deklareras. Löpande kod och funktioner behöver inte nödvändigtvis skrivas i klasser. Språkets interpreterande karaktär gör det möjligt att på ett lekfullt sätt experimentera med kod. Pga dessa fördelar och sin enkla, smdidiga och kloka kodningsteknik har Python mer eller mindre konkurrerat bort Java och kan idag anses som världens mest populära programmeringsspråk åtminstone inom utbildning.

2000 **C#** har sina rötter i programspråken C, C++ och Java och är därmed byggt på det gamla, beprövade och välkända. Den allra första versionen av C# släpptes år 2000 av *Microsoft*. Man tog över allt som var bra och skrotade allt som var lite krångligt hos de andra språken. Men den viktigaste förnyelsen var att det nya språket integrerades i Microsofts .NET-miljö för att göra det utbytbart mot de andra språken inom .NET. En stor del av världens mjukvara utvecklas idag i C#.

1.2 Olika paradigm inom programmering

Vad är ett paradigm? T.ex. är fri marknadsekonomi ett paradigm inom ekonomi, statligt styrd ekonomi ett annat. I programmering är t.ex. att koda på ett sätt som kan återanvändas i andra program ett paradigm, att inte göra det, ett annat. Enkelt uttryckt är paradigm inom programmering en programmeringsfiolsofi. Generellt:

> *Ett paradigm* är en samling av regler, rekommendationer, normer, konventioner, mönster, standards, metoder och teorier inom ett ämne, som delas och följs av de flesta inom ämnet under en viss tidsperiod.

Ett paradigm ger en orientering som styr handlingen och föreligger därför *före* erfarenheten (a priori), likt en fördom. *Efter* erfarenheten jämförs och bedöms handlingen med paradigmet (a posteriori), likt en lärdom. Överensstämmer resultatet efteråt inte med paradigmet, kan paradigmet (åtminstone delvis) ifrågasättas.

Ämnets naturliga progression leder efter längre tidsperioder ofta till byten av paradigm, s.k. *paradigmskiften*, förutsatt att ett nytt paradigm har ställts upp som bättre uppfyller de önskade kraven. I programmeringens historia är vi vittnen för många sådana paradigmskiften, se *Paradigmskifte* (sid 14). Här följer en kort beskrivning av de viktigaste paradigm som har dykt upp under programmeringshistorien:

Maskinorienterad programmering

Även kallad maskin*nära* programmering, vilket innebär att man skriver instruktioner som enkelt och snabbt, ja nästan direkt kan utföras av datorns processor (CPU). Maskinorienterade programmeringsspråk ligger allra närmast hårdvaran. Ursprungligen kan sådana maskinorienterade instruktioner endast utföras på en konkret maskin, eftersom de är definierade just för den aktuella hårdvaran. Ett typiskt exempel för ett sådant språk är *Assembler* som fortfarande är läsbar källkod som omvandlas till maskinkodens ettor och nollor av ett speciellt program som heter *assemblator*. Själva översättningsprocessen kallas för *assemblering*. Fördelen med maskinnära språk är den enkla och därmed snabba åtkomsten till hårdvaran, vilket kan vara avgörande i vissa sammanhang, t.ex. för spelkonsoler. Nackdelen är den svårt läsbara och icke-portabla koden.

Deklarativ programmering

Innebär att man anger *vad* som ska göras, inte *hur* det ska gå till. Man nöjer sig med att säga vad man vill ha. Tillvägagångssättet tas hand om av programmeringsspråket. Ett typiskt exempel för ett sådant språk är *SQL* som står för *Structured Query Language* och är standardspråket för kommunikation med databaser. Med en SQL-sats ställer man en fråga till en databas. Man får som svar den datamängd som är efterfrågad i SQL-satsen. Hur SQL letar efter och hittar denna datamängd i den väldigt komplexa databasen, behöver programmeraren inte bry sig om. Man

deklarerar endast sitt önskemål, precis som man beställer en maträtt på en restaurang. Deklarativ programmering har många underkategorier.

Funktionell programmering

En typ av deklarativ programmering är *funktionell programmering*. I detta paradigm består ett program av en samling matematiska funktioner som definieras och exekveras direkt med minsta möjliga tidsåtgång (runtime). Man undviker kod som anses vara onödig overhead och fokuserar på effektivitet och funktionalitet hos de mest små moduler utan att behöva ange i vilken ordning de ska exekveras. Ett typiskt funktionellt språk – dessutom det äldsta – är *Lisp*. I Visual Studio finns även ett funktionellt språk som heter *F#*. Historiskt har funktionell programmering sitt ursprung i ett matematiskt forskningsprojekt på 30-talet som resulterade i den s.k. *Lambdakalkylen*. I C# har man integrerat dessa tankar i språket *(Lambdauttryck)*.

Logikprogrammering

En annan typ av deklarativ programmering är *logikprogrammering* som baseras på matematisk logik. Ett logikprogram består i första hand av ett antal *axiomer* som kan anses vara en bas av definitioner och regler som alla följande instruktioner måste följa. All kod som skrivs kommer att exekveras endast enligt dessa axiomer. Man ställer en fråga och får svaret som en logisk slutsats ur axiomsystemet. Logikprogrammering har sitt ursprung i 70-talets forskningsaktiviteter kring *artificiell intelligens*. Det mest kända logikprogrammet är *Prolog*.

Händelsestyrd programmering

Detta paradigm är typiskt för grafiska applikationer *(GUI)*. Programkörningen är inte längre till 100% 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 rent textbaserade program. Exekveringen startar ofta 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. *Händelsestyrd programmering* används bl.a. i Windowsprogrammering med s.k. *Controls*.

Spaghettiprogrammering

Självklart finns det inte ett uttalat paradigm som heter så. Det är snarare en ironisk beteckning, ett smeknamn som man ur ett kritiskt perspektiv gett denna typ av programmeringsvana som man i brist på bättre lösningar använt i de äldre språken.

Så länge det inte fanns kontrollstrukturer använde man sig av s.k. *hoppsatser* för att åstadkomma loopar. Det reserverade ordet **`goto`** skickar programflödet till ett annat ställe i koden vilket man markerar med en *Label*, t.ex. med **`L`**. En label är ingen variabel utan en symbol som endast markerar ett ställe i koden. Den används i

goto-satsen för att skicka programflödet till det markerade stället. Ironiskt nog finns det reserverade ordet **goto** fortfarande i C#. T.e.x. kan det se ut så här:

```
L: Console.Write("\n\tGissa ett tal mellan 1 och 20:\t");
   guessedNo = int.Parse(Console.ReadLine());
                                  ...
   if (guessedNo != 17) goto L;
```

Om det gissade talet *inte är 17* dvs om användaren *gissat fel*, ska programmet hoppa till **L** där användaren ges åter möjligheten att göra en ny gissning som sedan prövas, osv. Om däremot det gissade talet *är 17*, dvs om användaren *gissat rätt*, äger hoppet inte rum. Man har med en **if**-sats, som är en enkel *selektion*, i kombination med **goto** lyckats konstruera en loop.

Varför kallar vi detta för spaghettiprogrammering när koden ovan fungerar? Anledningen är att hoppsatser leder i större program till förvirring. Föreställ dig att man har ett stort program, använder väldigt många **goto**-satser och utnyttjar fullt ut friheten att placera labels var som helst. Resultatet blir en kod som är svårt att kontrollera, uppdatera och underhålla. Programflödet liknar till sist en spaghettirätt. Sådana program uppfyller inte längre kraven om läslighet, förståelighet och ändringsbarhet. Det märks speciellt när en programmerare byter jobb och en efterträdare ska vidareutveckla programvaran. Ofta blir det helt omöjligt för efterträdaren att sätta sig in i koden. Redan på 60-talet ledde detta till en programvarukris och initierade utvecklingen av procedurala programmeringsspråk som Algol, Simula, Pascal, C, … där **goto**-satser kan och bör undvikas. Procedural programmering bannlyser användningen av **goto**-satser då en okontrollerad användning av hoppsatser i större program leder till spaghettiprogram som inte längre är läsliga, förståeliga och ändringsbara. Procedural programmering ersätter alla hoppsatser med kontrollstrukturer där det inte längre finns några labels då dessa är hårdkodade och placerade på fasta platser. Man borde ersätta **goto**-satsen med en kontrollstruktur av typ repetition, t.ex. en **while**-sats.

Procedural programmering

Motsatsen till deklarativ programmering är *imperativ programmering*. Procedural programmering är den äldsta typen av imperativ programmering. Här anger man inte bara *vad* som ska göras, utan även – och framför allt – *hur* det ska gå till. Tillvägagångssättet är en väsentlig del av imperativa språk. Ett tillvägagångssätt som exakt och entydigt *beskriver* hur man löser ett problem, kallas för *algoritm*. Man kan *beskriva* en algoritm på många olika sätt, t.ex. på vanligt språk, med hjälp av grafik, med pseudokod, i form av ett flödesschema osv. Väljer man programkod för att beskriva algoritmen, har man ett datorprogram. Ofta måste även viss *data* (t.ex. indata) läggas till för att lösa problemet. Därför ställde upp Niklaus Wirth, skaparen av programspråket *Pascal*, på 60-talet följande definition:

Program = algoritm + data

Data är information i organiserad, strukturerad form. Men vad exakt är en algoritm, och framför allt hur kan algoritmer *beskrivas*? Dessa frågor kommer vi att ägna oss åt i resten av det här kapitlet. Wirths definition återspeglar en algoritmorienterad syn på programmering som även kallas *procedural programmering*. Procedur är ett annat ord för algoritm. Modern till alla procedurala språk är *Algol*.

Objektorienterad programmering (OOP)

Om man i Wirths definition *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*. Den nya definition som kom upp på 80-talet och återspeglar den objektorienterade synen på programmering är:

Program = Modell av verkligheten

OOP 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 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. Beskrivningen kodas i *klasser*.

Ett objekt kan i regel utföra vissa aktioner eller operationer. I den objektorienterade programmeringens terminologi kallas de för *metoder* – samma som i den procedurala programmeringen heter funktioner. En metod är en funktion som definieras i en klass. Namnbytet beror på att man i OOP måste definiera sina funktioner i klasser, därför att metoderna i regel ska vara bundna till objekt. Förenklat kan man säga: när ett objektorienterat program körs anropar metoder varandra och skickar därvid objekt till varandra. På så sätt simuleras verkligheten.

Paradigmskifte

Det som i programmeringshistorien gjorde att man behövde objektorienterad programmering var den växande komplexiteten hos program under 70-talet. Programmens storlek var avgörande för den växande komplexiteten. Man insåg att det inte längre räckte till att skriva och testa program som fungerade just då. Det var nödvändigt att med rimliga kostnader kunna även *underhålla* stora program, *förnya* och *vidareutveckla* dem så att de fungerade även i flera år och att de framför allt kunde anpassas till nyuppkomna situationer utan oöverkomliga svårigheter. Det i sin tur krävde att man redan i designstadiet behövde ett annorlunda upplägg. Fokuset förskjöts från problemlösning till modellering av verkligheten. Objektorienterad design kom in i bilden. Allt detta var endast med procedural programmering inte längre möjligt. Ett s.k. *paradigmskifte* hade blivit nödvändigt, dvs en ändring av helhetssynen på programmering.

1.3 Algoritmer och deras beskrivning

Många tror att algoritmer endast har med matematik att göra. Även om algoritmer historiskt har introducerats av matematiker kan de användas på all problemlösning. Man kan t.o.m. tillämpa algoritmer på vardagliga problem. Samtidigt ligger de till grund för all programmering. Ett datorprogram är ingenting annat än en algoritm beskriven i datorns språk. Men även följande vägbeskrivning är ett fullgott exempel på en algoritm:

> *” ... gå ut från ditt hus till vänster, fortsätt rakt fram, sväng till höger vid trafikljuset, fortsätt sedan andra korsningen till vänster, där finns ett gult hus, på 2:a våningen bor jag ...”*

En *algoritm* är alltså ett tillvägagångssätt att lösa ett problem – vilket som helst. Och det behöver inte heller vara datorn som löser det. Vi kommer att precisera denna definition lite senare (sid 17). Problemet som ska lösas kan sakna lösning – då kan det inte heller finnas någon algoritm. Om däremot problemet är lösbart, kan det ha ingen, en eller flera algoritmer. Vi sysslar här endast med sådana problem som har minst en algoritm.

Historiens första algoritm

Det är alltid lärorikt att blicka tillbaka till historien. Själva ordet *algoritm* härstammar från ett namn på en person: namnet på den framstående persiska matematikern *Al-Kharazmi*[*]. Namnet har sedan latiniserats och blivit *algoritm.* Han levde på 800-talet. I sin berömda bok om *Algebra* ställde han upp historiens första algoritm som beskriver addition och multiplikation av heltal. Den används även idag. Men kunde man inte addera eller multiplicera heltal på 800-talet? Jo, redan långt tidigare kunde man räkna med tal i Egypten, Indien, Persien och Grekland. Vad var i så fall Al-Kharazmis historiska prestation? Ja, det var inte att komma på hur man *adderar* eller *multiplicerar* heltal – för det var ju redan känt, utan hur man i allmänna ordalag *beskriver* tillvägagångssättet, dvs formulerar en algoritm för dessa operationer.

1000 år mellan praktisk lösning och formell beskrivning

Det är anmärkningsvärt att *beskrivningen* av hur man räknar med heltal kom till mer än 1000 år efter den praktiska lösningen. Orsaken är att den korrekta, allmänna beskrivningen som ska hålla i *alla* tänkbara situationer, är mycket svårare att åstadkomma än den faktiska lösningen av ett eller en klass av problem. Att själv gå en väg som man känner till är enklare än att formulera en korrekt vägbeskrivning som alla förstår och kan följa. Anledningen är att algoritmer är *generella* till sin natur, och just det är tjusningen: Att försöka beskriva dem så att de håller i *alla* situationer, det är konsten. Detta gäller även idag: Program – det moderna sättet att beskri-

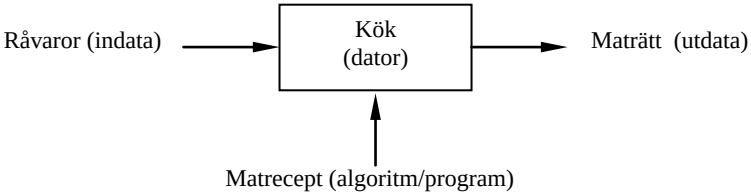
[*] Så uttalas hans namn på persiska idag (utan prefixet *Al-* som är arabiska). Han är född i *Kharazm*, en antik region som fanns i nuvarande nordöstra delen av Iran (Khorasan) mot Turkmenistan och Uzbekistan. På den tiden var Iran ockuperat av araberna.

va algoritmer – måste fungera under alla omständigheter och ska helst aldrig krascha. Dessvärre vet vi ju att så inte är fallet. En av utmaningarna inom programmering ligger just i att skriva program som fungerar i *alla* situationer. Det vi kan lära oss av det 1000-åriga glappet mellan praktisk lösning och formell beskrivning är: Satsa tid och energi på att först *analysera* det problem du vill lösa med ett program. Fokusera på att *beskriva* lösningen av problemet så generellt som möjligt.

Exempel på algoritmer

I vardagen använder vi algoritmer hela tiden, om än omedvetet. Här några exempel:

- **Matrecept** vars användning kan jämföras med programkörning på datorn:



  Matrecept skrivs fortfarande med vanligt språk men man kan konstatera att det finns en viss stil som är typisk för alla matrecept.

- **IKEA:s monteringsanvisningar** för att sätta ihop delarna till en möbel. Här används en kombination av text och grafik som är mycket effektiv. Grafiken förenklar algoritmen avsevärt. ”En bild säger mer än tusen ord.” På köpet får man en slags internationalisering, ett oberoende av det lokala språket, vilket gör att algoritmen förstås över hela världen.

- **Bruksanvisningar** av alla slag är exempel på algoritmer, även om många av dem i praktiken är värdelösa. Men det finns dåliga algoritmer på andra områden också.

- **Manualer** för datorprogram som visar hur ett program ska användas.

- **Konstruktionsritningar** som ingenjörer gör för att en viss produkt ska kunna tillverkas i fabrik. En arkitektritning av ett hus är ett specialfall av det. Här har grafiken tagit över helt och hållet.

- **Partiturer:** Noter i musik som används för att spela ett musikstycke och som omfattar noggranna anvisningar om hur en hel orkester ska spela. Ett speciellt ”språk” används som varken består av text eller grafik, utan snarare av symboler längs en tidslinje.

- **Spelregler** är snarare ett negativt exempel: De talar mest om vad man *inte* får göra och lämnar ett stort utrymme för hur man får spela inom reglernas ram. Därför finns två skilda problemställningar. Den ena är: ”Hur *får* jag spela?”

Spelregler ger delvis (negativa) svar på det. En helt annan problemställning är: ”Hur *vinner* jag spelet?” *Spelteori* som involverar sannolikhetslära behandlar denna fråga. I spelteori brukar man tala om *strategier* snarare än algoritmer. Här befinner vi oss i ett gränsområde där problem inte alltid har en entydig lösning eller saknar algoritm. I fortsättningen kommer vi att undvika sådana frågeställningar. Vi betraktar endast problem som är lösbara och har minst en algoritm. Exemplet belyser dock en viktig aspekt: Inte bara vägen till lösning måste beskrivas. Först måste *problemställningen* vara klart och exakt formulerad så att man kan avgöra om det finns en entydig lösning och minst en algoritm.

Definition av algoritm

Låt oss titta på vad som är *gemensamt* för exemplen ovan (utom spelreglerna), för att kunna fornulera en generell definition. Vilka typiska faktorer förekommer i alla exempel?

För det första består de alla av en rad anvisningar om vad som ska göras för att lösa det givna problemet. Frågan är: Ska man tillåta alla slags anvisningar? Om de leder till problemets lösning, varför inte? Men leder alla slags anvisningar till lösningen? T.ex. anvisningen ”Bygg ett hus!” är helt värdelös. Ingen av oss kan bygga ett hus med bara denna anvisning. Problemet är ju just *hur* man bygger huset. Anvisningarna måste vara mycket enklare och mer detaljerade. Vem som helst ska kunna utföra dem. Sådana anvisningar kallas *elementära instruktioner*. Bara sådana kan tillåtas i en algoritm om de ska leda till problemets lösning.

För det andra. Undersöker man de ovannämnda exemplens innehåll kan man konstatera att anvisningarna måste utföras i en viss *ordning.* Det går inte att kasta om ordningen. Man inser redan vid receptexemplet att man *först* måste knåda degen och *sedan* ställa in den i ugnen, inte vice versa. Vid partiturexemplet är ju ordningen helt avgörande. Och så är det i alla algoritmer. Ordningsföljden för de elementära instruktionerna måste finnas med i algoritmen. Självklart måste en algoritm också ange när instruktionerna ska upphöra. Om vi sammanfattar kan vi formulera följande definition:

> En *algoritm* är en följd av precisa anvisningar, s.k. *elementära instruktioner*, som löser ett givet problem, inklusive anvisningar om i vilken *ordning* instruktionerna ska utföras och när de ska avslutas. Dvs en algoritm måste ha ett exakt *avslutningskriterium*.

Av stor betydelse, speciellt för datoriseringen, är att algoritmen måste vara tolkningsbar *på ett enda sätt*. Det får inte finnas tvetydigheter i formuleringen. Datorn kan ju bara tolka våra anvisningar på ett enda sätt. Svårigheten ligger alltså i algoritmens *beskrivning*, vilket är en god illustration till det 1000-åriga glappet mellan praktisk lösning och formell beskrivning som nämndes på sid 15. Det är i regel svårare att *beskriva* en algoritm än att lösa ett specifikt problem i en specifik situation.

Anledningen är att algoritmer måste vara *generella* till sin natur: De måste hålla i *alla* situationer. Följande dilemma uppstår:

Hur beskriver man en algoritm bäst, så att den kan tolkas *endast på ett sätt*, men samtidigt behålla sin *generella* karaktär? Vi ska nu diskutera några hjälpmedel som kan användas för att formulera sådana algoritmer:

Olika sätt att beskriva algoritmer

- **Vanligt språk** är ett sätt att beskriva algoritmer, t.ex. vägbeskrivningen till en kompis. Största fördelen med det är att alla som kan språket direkt förstår algoritmen utan att behöva lära sig något nytt. Nackdelen är att det ofta kan tolkas på olika sätt. Och tur är det! Annars skulle man ju t.ex. inte kunna skriva en dikt eller njuta av den. Men just i samband med algoritmer då man eftersträvar entydighet, är möjligheten till olika tolkningar en nackdel.

- **Pseudokod** är en hybrid (blandning) mellan vanligt språk och formaliserad kod, ett försök att minska det vanliga språkets tvetydighet genom att införa vissa strukturer och t.o.m. grafiska stilmedel i layouten. Allt som på ett entydigt sätt beskriver en algoritm, även en matematisk formel, kan användas som pseudokod. I nästa avsnitt tar vi upp ett exempel på pseudokod med vanligt språk kombinerad med generella *kontrollstrukturer* (sid 22) som förekommer i alla algoritmer. På så sätt uppnår det vanliga språket en högre grad av entydighet, noggrannhet och struktur.

- **Flödesschema** eller flödesschema är en variant av IKEA:s monteringsanvisningar som kombinerar text och grafik med en klar dominans mot det senare. Man använder sig av geometriska figurer som symboliserar algoritmens byggstenar och av pilar som visar flödet i algoritmen och definierar instruktionernas ordning. Med dessa få stilmedel uppnår man en hög noggrannhet i beskrivningen, eliminerar tvetydigheter och åskådliggör algoritmens logiska struktur. Det tänkta händelseförloppet syns tydligt. I det avseendet är flödesschema överlägset både vanligt språk och pseudokod. Flödesschemassymbolik är ett utmärkt medel som lämpar sig inte bara för beskrivning av fullständiga algoritmer, utan också för att åskådliggöra logiken hos mindre, men kritiska delar av ett program. Vi kommer att använda oss av detta medel i hela boken.

- **Programkod** är den variant av algoritmbeskrivning som används för att låta en dator utföra algoritmen. Därför måste den kunna tolkas av datorn. Programkoden översätts till ett språk, kallat *maskinkod* som datorns processor förstår. Programkoden däremot – även kallad *källkod* – är skriven i något programmeringsspråk som man måste lära sig. Medan källkod förstås av människan, men inte av datorn, förstås maskinkod av datorn, men inte av människan.

- **Andra sätt** att beskriva algoritmer finns också. Inget av dem har lyckats etablera sig som standard. Anledningen är att det är oförutsägbart vilka metoder som i allmänhet kan lösa problem. Många av de traditionella sätten kan betecknas med det samlande namnet *pattern designs*. Andra använder begrepp

som *strukturdiagram*, *Mind Maps* eller *beslutstabeller*. Mest känt är dock *UML = Unified Modeling Language* som är ett språk för objektorienterad design och modellering. Man använder UML för att att planera, utveckla och visa strukturen hos avancerade objektorienterade system. UML används för att lägga upp och modellera stora programmeringsprojekt, vilket förutsätter bekantskap med den objektorienterade programmeringens terminologi. Vi kommer att ta up UML senare i avsnitt 4.9 (sid 94). I nästa avsnitt ska vi börja utveckla de traditionella struktureringsverktygen *pseudokod* och *flödesschema*.

1.4 Traditionell design pattern med flödesschema

Låt oss som exempel ta följande beskrivning på ren svenska av en vardaglig syssla:

> ”K. går upp kl. 6 och duschar tills kroppen känns fräsch. Sedan torkar K. sig, tar på sig kläderna och äter frukost. Vid frukosten lyssnar K. på radions trafikinformation. Om det är mycket biltrafik, går K. ut, väntar tills ingen bil kommer, går över gatan och tar bussen till jobbet. Annars tar K. bilen till jobbet.”

Det är en beskrivning av en algoritm, låt oss kalla den för *Morgonsyssla*, som använder sig av det vanliga språket. Egentligen kan den knappast misstolkas när den används med lite sunt förnuft. Ändå vill vi skriva om den, först som *pseudokod* och sedan som *flödesschema* för att lära känna de nya begreppen. Som vi ska se kommer detta att leda till en precisering av algoritmen.

Pseudokod till algoritmen Morgonsyssla

```
Gå upp kl. 6
Duscha TILLS kroppen känns fräsch
Torka och ta på dig kläderna
Ät frukost och lyssna på radio
OM det är mycket biltrafik
    gå ut
    vänta TILLS ingen bil kommer
    gå över gatan och ta bussen till jobbet
ANNARS
    ta bilen till jobbet
```

Låt oss analysera denna pseudokod lite närmare. Vad skiljer den från vanligt språk? Vi har gett texten en ny *form* utan att ändra *innehållet*. Nya ”regler” för formen har införts: För det första finns det varken punkter eller kommatecken mellan satserna. För att skilja dem åt, börjar istället varje sats på en ny rad. För det andra innehåller varje sats endast *en* elementär instruktion. För det tredje är vissa rader indragna vilket visar att instruktionerna på dessa rader, är underordnade andra instruktioner dvs är delar av dem. Så kan vi skilja mellan huvud- och underinstruktioner. Algoritmen har 5 huvudinstruktioner:

```
 I.  Gå upp kl. 6
 II. Duscha TILLS kroppen känns fräsch
 III. Torka och ta på kläderna
 IV. Ät frukost och lyssna på radio
 V.  OM ...
     ANNARS ...
```

Att vi räknar **OM-ANNARS**-satsen som *en* instruktion, beror på att de hör ihop och bildar ett par: **ANNARS** skulle förlora sin mening om det skiljdes från **OM**. Sedan har algoritmen 4 underinstruktioner, 3 under **OM** och 1 under **ANNARS**. De är alla indragna. Underinstruktionen ”gå ut” skulle kunna betecknas med V.a eftersom den tillhör huvudinstruktion V. Undersinstruktionen ”vänta **TILLS** *ingen bil kommer*” skulle i så fall få beteckningen V.b. Undersinstruktionen ”gå över gatan och ta bussen till jobbet” blir V.c och ”ta bilen till jobbet” V.d. Hela algoritmen består av 5 huvud- och 4 underinstruktioner.

Villkor

Låt oss nu fördjupa analysen av pseudokoden och ta itu med de lite mer invecklade instruktionerna, t.ex. med II:an:

Duscha **TILLS** *kroppen känns fräsch*

Hur länge står K. under duschen? Innebörden av **TILLS** säger att detta avgörs av hur länge *kroppen känns ofräsch.* Dvs K. frågar sig ständigt, självfallet omedvetet: *känns kroppen fräsch, ja eller nej?* Om nej, fortsätt duscha! Om ja, sluta! Detta händer kontinuerligt under duschandet. Hur många gånger, är inte bestämt, utan avgörs av K.:s subjektiva svar på frågan. Menar K. att kroppen förblir ofräsch trots duschandet, då ska K. enligt algoritmen fortsätta att duscha i all evighet – rent hypotetiskt! I pseudokoden formuleras *känns kroppen fräsch* däremot inte som fråga, utan som ett *villkor* som ingår i **TILLS**-satsen, ett villkor för att fortsätta eller avsluta duschandet. Villkoret testas gång på gång: är det sant, ska K. avsluta duschen. Är villkoret falskt, ska K. duscha vidare. Valet avgörs av villkorets s.k. *sanningsvärde*, dvs om det är sant eller falskt. Ett villkor kan antingen vara sant eller falskt. På så sätt skiljer sig ett villkor från en instruktion. En instruktion utförs, medan ett villkor *testas*. Testet avgörs av villkorets sanningsvärde. Därmed avgörs även om den instruktion som knyts till villkoret, ska utföras eller ej.

Det finns flera villkor i pseudokoden, utmärkta i kursiv stil. Nästa villkor förekommer i huvudinstruktion V:

OM *det är mycket biltrafik*
. . .
ANNARS ta bilen till jobbet

Den kursiva texten är ett villkor som avgör om K. ska gå över gatan och ta bussen eller ta bilen till jobbet. Är villkoret sant (mycket trafik), då ska K. gå över gatan och ta bussen. Är villkoret falskt (inte mycket trafik), ska K. ta bilen till jobbet. Men till skillnad från **TILLS**-satsen testas villkoret här endast en gång, beroende på den annorlunda logiska innebörden av **OM**.

Ett tredje villkor finns i underinstruktionen V.b:

vänta **TILLS** *ingen bil kommer*

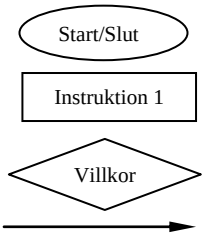
Logiken avgörs igen av **TILLS** dvs K. ska vänta så länge det kommer någon bil. När det inte längre kommer någon bil, ska K. sluta vänta. K. ställer sig gång på gång frågan: *kommer någon bil, ja eller nej?* Om ja, fortsätt vänta! Om nej, sluta vänta! Kommer det bilar hela tiden, då ska K. enligt algoritmen vänta i all evighet!

Kontrollstrukturer i algoritmer

Har vi därmed kartlagt pseudokoden till algoritmen Morgonsyssla? Nästan! Vi har identifierat *instruktioner* (normal stil) och *villkor* (kursiv stil). Vi nämnde även orden **TILLS** och **OM-ANNARS** (fet, versal stil), men vi har ännu inte identifierat dessa ord. De är ju varken instruktioner eller villkor, så vad är de? Låt oss för ett ögonblick glömma algoritmen Morgonsyssla och tänka oss en helt annan algoritm som ska lösa ett helt annat problem. Vilka ord skulle även förekomma i den nya algoritmen? Säkert ingen K.*, inget jobb, ingen dusch, ingen bil, ingen ... . Men just det! Orden **TILLS** och **OM-ANNARS** kan finnas i den nya algoritmen också. Och de kan förekomma inte bara i denna algoritm utan i alla algoritmer. De är nyckelord och fungerar som algoritmens byggstenar. I programmering kallas de för *kontrollstrukturer* eftersom de är generella strukturer som styr och kontrollerar hela algoritmen. Ja, alla algoritmer är uppbyggda av dessa kontrollstrukturer. Behärskar man dem, har man tagit ett stort steg mot förståelse av algoritmer och därmed förståelse för programmering. Det finns tre grundläggande kontrollstrukturer i alla procedurala programmeringsspråk:

- **Sekvens (följd)**
- **Selektion (val)**
- **Repetition (upprepning, loop)**

För att rita flödesschema används följande symboler:



Algoritmens start och slut ritas med en oval.

En **instruktion** ritas som rektangel. Ett **villkor** ritas som romb.

Villkoret skrivs in i romben och kan även formuleras som fråga.

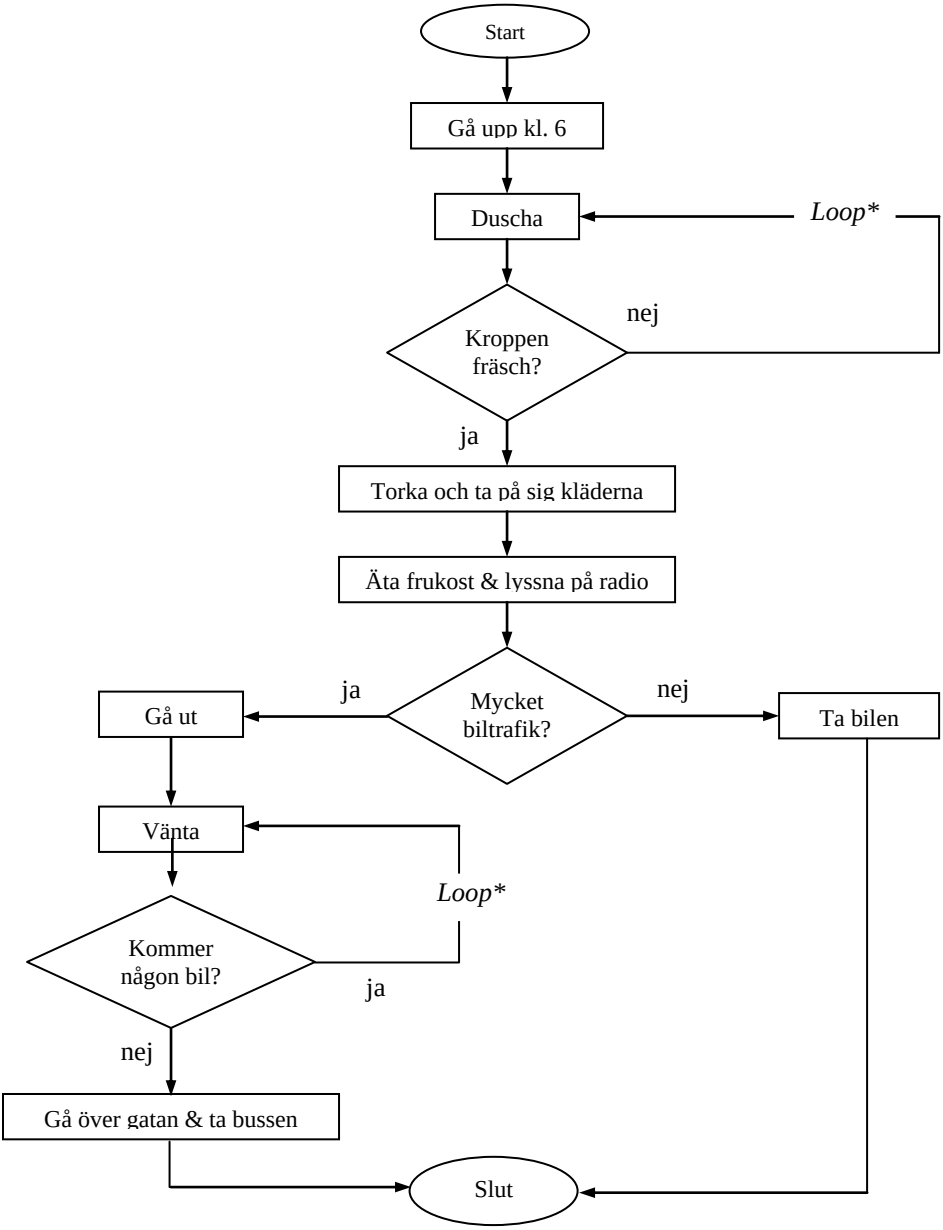
Ordningen i algoritmen (flödet) visas med pilar.

Det finns fler symboler än de som använts i flödesschemat till algoritmen Morgonsyssla som ska vara en exakt översättning av den algoritm som vi ursprungligen formulerade först på vanligt språk och sedan som pseudokod. Precis som vi gav texten i vanligt språk en ny *form* utan att ändra *innehållet* när vi skrev om den till pseudokod, ska även vid översättning till flödesschema ytterligare en ny form ges till algoritmen utan att ändra innehållet, framför allt inte den logiska innebörden. Flödesschemats fördel kan beskrivas med ordspråket *En bild säger mer än tusen ord*. Nu ska vi rita algoritmen Morgonsysslas flödesschema.

---

* Precis som i litterära verk protagonisten kan vara vem som helst (t.ex. Kafkas romanfigur ”Herr K.”) kan även algoritmens K. stå för *vem som helst*. I pseudokoden och även i flödesschemat på nästa sida förekommer inte ens K., vilket visar att det inte handlar om personen utan om *problemet ”Att ta sig till jobbet”*. Vi har att göra med problemlösning (procedural), inte med modellering av verkligheten (objektorientering).

Flödesschema till algoritmen Morgonsyssla



* *Loop* = upprepningsslinga med inbyggt villkor som testas gång på gång.

När vi säger att Morgonsyssla-algoritmens flödesschema ska bli en exakt översättning av den algoritm som vi ursprungligen formulerade på sid 20 menade vi förstås den *logiska* likheten, inte den språkliga. T.ex. står i pseudokoden ”vänta **TILLS** *ingen bil kommer*” medan i flödesschemat står ”Kommer någon bil?” och flödesschemat svarar på denna fråga: ”om ja, vänta” vilket innebär ”vänta **SÅ LÄNGE** *det kommer någon bil*”. Formuleringen är logiskt likvärdig med ”vänta **TILLS** *ingen bil kommer*”. Hade vi formulerat frågan negativt ”Kommer *ingen* bil?” hade det lett till dubbel negation vid svaret nej, vilket försvårar förståelsen. För att förenkla har frågan i flödesschemat formulerats positivt. Undersök själv om det finns flera exempel på språklig olikhet men logisk likhet mellan den ursprungliga texten och flödesschemat. Det är en utmärkt övning att kontrollera om vi på vägen från vanligt språk till flödesschema verkligen inte ändrat algoritmens innehåll.

Om man jämför pseudokoden med flödesschemat till Morgonsyssla kan man konstatera att det är avsevärt enklare att få en snabb överblick över algoritmen när man tittar på flödesschemat. Frågan uppstår varför man i så fall överhuvudtaget ska syssla med pseudokod. Svaret är att det är programkod som vi slutligen ska skriva, och programkod liknar pseudokod mer än flödesscheman. Vi kan inte mata datorn med grafik som är huvudingrediensen i flödesscheman. Pseudokodens värde ligger i närheten till programkod. Dessutom är den oberoende av programmeringsspråk. Flödesschema däremot är ett utmärkt hjälpmedel som kan användas *innan* man skriver programkod för att strukturera sina tankar om ett problems lösning som ska tas fram med ett datorprogram. Även detta verktyg är helt oberoende av programmeringsspråk. Är problemet enkelt eller om en klar struktur för lösningen redan finns, behövs ingen flödesplan. Växer problemets komplexitet rekommenderas en flödesschema kombinerad med pseudokod.

1.5 Tillägg av C# i Visual Studio

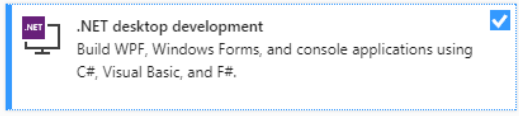
Här förutsätts att du redan installerat Visual Studio på din dator och använt denna IDE för att köra andra programmeringsspråk. Nedan beskrivs hur du kan lägga till språket C# till din befintliga miljö, utan att behöva av- eller ominstallera den tunga programvaran i sin helhet.

1) Gå till dators Start-knapp och starta Visual Studio Installer.

2) Klicka på knappen Modify som befinner sig till höger om ikonen och texten Visual Studio Community 2022.

3) Visual Studio Installer öppnar ett stort vitt fönster med den lilla rubriken Modifying – Visual Studio Community 2022 … och den blå understrukna fliken **Workloads**. I den finns ett antal rutor. Leta efter följande ruta (3:e till höger):

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



5) Klicka sedan på **Install** i det nedre högra hörnet av det stora vita fönstret Installing – Visual Studio Community 2022 … . Du kan själv välja bland alternativen Install while downloading eller Download all, then install. Detta kan ta ett tag, ev. ganska länge – beroende på din Internet-uppkoppling och din dators prestation.

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

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

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

7) Efter lyckad tilläggsinstallation av C# följ strikt de anvisningar du hittar i nästa avsnitt ***1.6 C# Console Applications***. Små, men avgörande detaljer skiljer sig från de anvisningar du lärt dig tidigare. Det gäller speciellt *typen av projekt* direkt i början, punkt **1. a)** och *filtypen* lite senare, punkt **2. a)**. Skynda inte på utan var noga i att strikt följa instruktionerna, även om mycket känns redan bekant. Bokens alla program kommer att vara av typ *C# Console Applications*. Du kan även här använda samma projekt för alla konsolapplikationer.

1.6 C# Console Applications

Starta Visual Studio från Start-knappen: Start → Visual Studio 2022

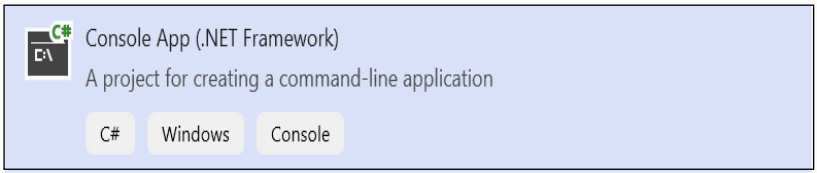
Ett vitt fönster öppnas med rubriken Visual Studio 2022. I kolumnen till höger under rubriken Get started finns ett antal rutor.

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

   a) Om vi vill skapa ett nytt projekt – och det vill vi nu – klickar vi i det vita Visual Studio 2022-fönstret på rutan

   Create a new project

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

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

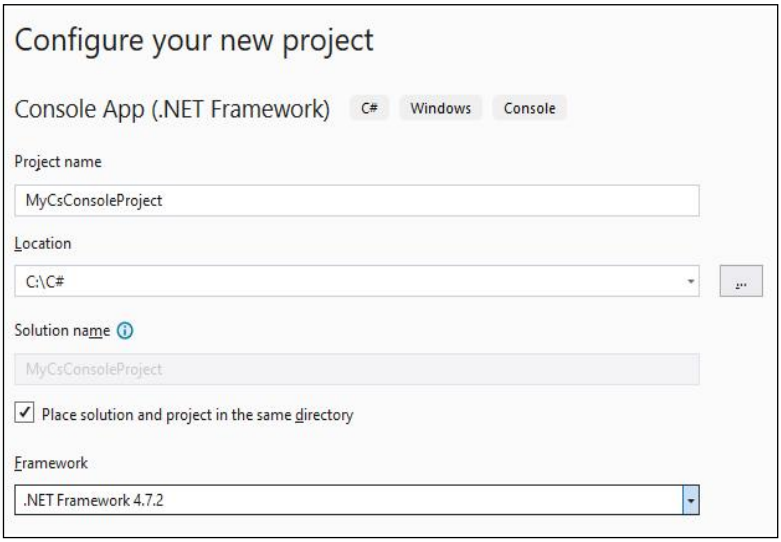
   Console App (.NET Framework)

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

   Markera rutan ovan. Klicka sedan på knappen Next.

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

   Fyll i den uppgifterna enligt följande:



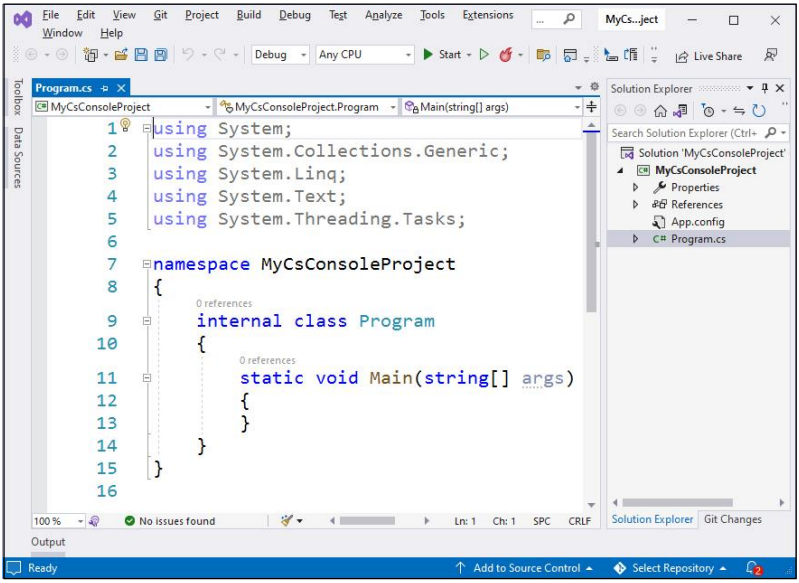
Dvs i den övre delen av dialogrutan döper vi vårt projekt till MyCsConsoleProject. I textrutan Location anger vi den fullständiga sökvägen till den mapp vi vill placera vårt projekt i. Låt oss säga vi vill samla våra C# program i en mapp som vi kallar C# och placerar i enheten C:\ på vår dator. I så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

b) Om vi vill öppna ett redan befintligt projekt – det gör vi kanske senare – klickar vi i det vita Visual Studio 2022-fönstret på rutan

Open a project or solution

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

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



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

Exclude From Project

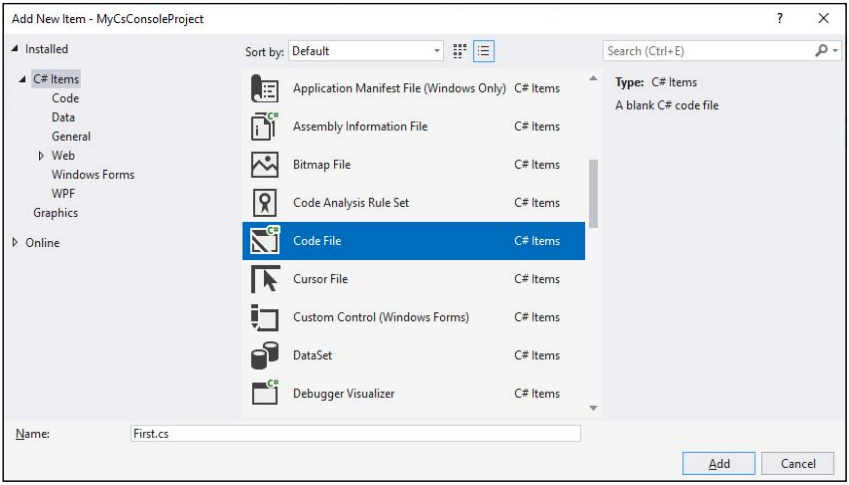
Därmed har vi avlägsnat denna fil från projektet för att kunna infoga vårt eget C# program i projektet. Det finns två alternativ att göra det: Antingen vill vi skapa ett helt nytt program, skriva in koden, spara den i en fil och infoga den i projektet eller vi vill lägga till en redan befintlig fil som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

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

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

Add → New Item…

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



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

```
using System;

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

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

b) Att lägga till en befintlig fil till projektet:

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

Add → Existing Item…

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

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

   View → Output

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

   Build → Build Solution

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

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

   Meddelandet ovan, speciellt `0 failed`, säger att koden inte innehåller några kompileringsfel. Har du syntaxfel i koden kommer du att få felmeddelanden i Output-fönstret. Åtgärda alltid endast det allra första kompileringsfelet och kompilera om. Ett möjligt kompileringsfel kan vara att du glömt att exkludera filen Program.cs från projektet, se sid 28.

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

   Debug → Start Without Debugging

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



1.7 De enkla datatyperna i C#



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

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

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

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

```
    "           double       " + sizeof(double)  + '\n' +
    "           decimal    "  + sizeof(decimal) + " bytes\n");
  }
}
```

De enkla datatypernas gränser

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

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

decimal              -79228162514264337593543950335 och
                      79228162514264337593543950335

bool tar endast värdena    True    och    False
```

Till skillnad från de andra datatyper som kan anta både positiva och negativa värden, kan de *teckenlösa* datatyperna (**u** = **unsigned** dvs utan tecken **+** eller **–**) endast anta positiva värden: De heter så därför att deras värden varken behöver ha plus- eller minustecknet framför talet. Dessa enkla datatyper har precis lika mycket minnesutrymme till förfogande som sina motsvarande vanliga datatyper med tecken. Detta innebär att nödvändigheten att lagra tecknet faller bort hos **unsigned**-typerna. Om vi resonerar vidare i exemplet med **short** skulle datatypen **ushort** ha alla 16 bitar till förfogande för själva positiva heltalet. Det största binära heltal som kan lagras i 16 bitar består av 16 ettor dvs 1111 1111 1111 1111. I det decimala talsystemet blir detta 65 535. Därför är gränsen för datatypen **ushort** dubbelt så

stort (fast +1 pga nollan) som för **short**. Och så är det med alla **unsigned**-typer: deras gränser är dubbelt så stora fast de har lika stort minnesutrymme till förfogande, därför att de inte behöver lagra tecknet och därmed har 1 bit mer för att lagra själva positiva heltalet. Av samma anledning har **byte** en dubbelt så stor övre gräns som **sbyte** fast båda tar endast 1 byte minne. Decimaltalstyperna **float** och **double**:s gränser visas i utskriften ovan i s.k. **E**xponentiellt format, även kallat *grundpotensform* (eng.: *Scientific notation*) vilket innebär att t.ex. **float**:s positiva gräns **3.4028235E38** är lika med 3,4028235 gånger 10 upphöjt till 38 dvs $3{,}4028235 \cdot 10^{38}$.

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

class Limits
{
  static void Main()
  {
    Console.WriteLine("Enkla datatypernas gränser:\n"       +
                      "--------------------------\n"       +
            "sbyte  finns mellan  " + sbyte.MinValue        +
            "               och    " +   sbyte.MaxValue     +
            "\nbyte                  " +    byte.MinValue    +
            "                        " +    byte.MaxValue   +
            "\nchar                  " + (int) char.MinValue +
            "                        " + (int) har.MaxValue +
            "\nshort                 "  + short.MinValue     +
            "                    "      + short.MaxValue     +
            "\nushort                " + ushort.MinValue     +
            "                        " +  ushort.MaxValue   +
            "\nint                   "  +  int.MinValue      +
            "               "           +  int.MaxValue      +
            "\nuint                  "  +  uint.MinValue     +
            "                        " +  uint.MaxValue     +
            "\nlong                  "  +  long.MinValue     +
            "     "                     +  long.MaxValue     +
            "\nulong                 "  + ulong.MinValue     +
            "                        " + ulong.MaxValue     +
            "\nfloat                 "  + float.MinValue     +
            "            "              + float.MaxValue     +
            "\ndouble                "  + double.MinValue    +
            "   "                       + double.MaxValue    +
            "\n\ndecimal\t              " + decimal.MinValue +
            " och \n\t\t      "           + decimal.MaxValue +
            "\n\nbool tar endast värdena    " + true        +
            "    och    "  + false + '\n');
  }
}
```

1.8 Inläsning av data

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

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

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

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

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

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

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

```
        Vad heter du?           Peter

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

        Välkommen till C#-kursen!
```

Data som matas in från tangentbordet eller läses in från filer, är indata. Till skillnad från utdata som inte behöver mellanlagras, måste indata lagras i minnet. Både indata och programkod måste lagras i RAM-minnet. Programkoden laddas från hårddisken till RAM-minnet när maskinkoden i den exekverbara filen körs. Indata däremot måste matas in under programmkörning och mellanlagras i en minnescell i RAM-

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

Metoden `ReadLine()`

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

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

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



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

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

Metoden `int.Parse()`

Hade **`string`**-variabeln **`name`** i programmet **`InputCs`** varit t.ex. **`number`** och dessutom av datatypen **`int`** istället, hade vi behövt att läsa in den så här:

```
int number;

Console.Write("\n\tMata in ett heltal:\t");  // Ledtext
number = int.Parse(Console.ReadLine());      // Inläsning
                                    // och omvandling till int
```

Vi vet ju att returtypen av metoden **`ReadLine()`** är **`string`**. För att kunna läsa in även heltal måste vi omvandla returtypen till **`int`**. Just detta gör den fördefinierade metoden **`int.Parse()`** åt oss. Den tar emot i sin parentes en parameter som är av typ **`string`**, omvandlar den till heltal och returnerar den som en **`int`**.

Satsen **`number = int.Parse(...);`**

utför denna omvandling och lagrar resultatet i variabeln **`number`** som är deklarerad som **`int`**. Även här är anropet av metoden **`int.Parse()`** inbakat i en tilldelningssats för att ta hand om metodens returvärde. De tre punkterna **`...`** är i sin tur returvärdet till metoden **`ReadLine()`** som är **`string`**. I själva verket står till höger om tilldelningstecknet i satsen ovan ett s.k. *nästlat anrop* av de två metoderna **`ReadLine()`** och **`int.Parse()`**. Observera att nästlade anrop av två (flera) metoder sker alltid *inifrån*.

Villkorlig initiering

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

```
// CondInit.cs                  // Kan ej kompileras
// Ger kompileringsfel pga villkorlig initiering av variabeln
// tecken i if-satsen

using System;

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

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

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

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

```
// UncondInit.cs                // Kan kompileras

using System;

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

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

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

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

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

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

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

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

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

Förbudet mot villkorlig initiering är inte begränsad till **`if`**-satser utan gäller även i andra kontrollstrukturer där villkor är inblandade.

Villkorlig **`return`**-*sats*

Även i metoder med returvärde får metodens **`return`**-sats inte stå villkorligt, vare sig villkoret tillhör en **`if`**-sats eller en loop. I sådana fall ger C# kompilatorn följande felmeddelande:

"... not all code paths return a value."

Dvs: Inte alla teoretiskt möjliga alternativ i koden returnerar ett värde. Och det går inte eftersom din metods huvud är definierat med en returtyp **`int`**, **`char`**, **`float`**, **`double`**, **`string`**, … istället för **`void`**. Du måste infoga **`return`**-satser i kodens *alla* teoretiskt möjliga alternativ, även om det sker rent formellt. Observera att tomma **`return`**-satser av typ **`return;`** inte är tillåtna i C# – till skillnad från C++.

Förbuden mot villkorliga **`if`**- och **`return`**-satser har införts i C# för att göra programmen mer stabila och tillförlitliga, så att de med en så liten arbetsinsats som möjligt kan vidareutvecklas till senare versioner.

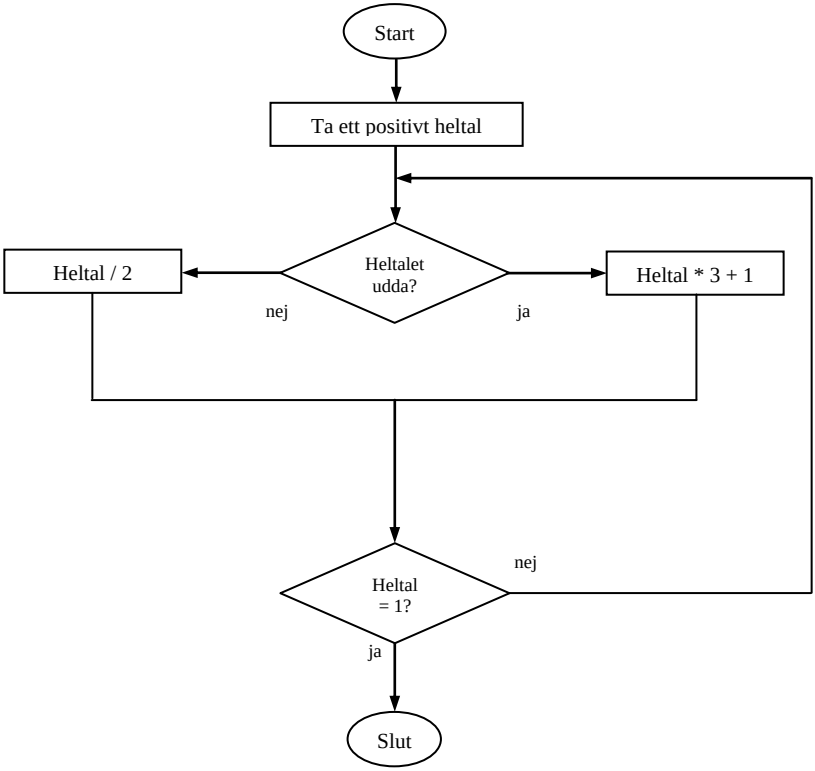
1.9 Collatz algoritmen

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

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

Det visar sig att talföljderna i denna algoritm, även känd som *Collatz-förmodan* eller *(3n+1)-problemet*, alltid slutar med 1 oavsett startvärde. Dock är detta påstående matematiskt hittills obevisat [*]. Så här ser flödesschemat ut för denna algoritm:

Flödesschemat till Collatz algoritmen



[*] Man kan testa Collatz algoritmen i appen *Mattekollen* där den är kodad i Python. Ladda ned appen eller kör den som Webbapp: **app.mattekollen.se → En mobil pythonmiljö**. Eller kör den direkt som webbapp: beta.mattekollen.se/#/app/coding. Prova koden med olika startvärden för att kolla om algoritmens talföljder alltid slutar med 1.

Flödesschemat visualiserar algoritmens logisla struktur som är grundläggande. Men för att koda kan det vara fördelaktigt att formulera algoritmen även som pseudokod som ligger närmare programkoden än flödesschemat.

Pseudokoden till Collatz algoritmen

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

Som man ser har vi redan anpassat texten i pseudokoden till programmering, t.ex. med formuleringar som `Läs in ...` och `Skriv ut ...` . I följande program implementerar vi Collatz algoritmen:

```
// Collatz.cs
// Läser in ett positivt heltal och tar det gånger 3 + 1 om
// det är udda, annars delar det med 2, tills det blir 1
using System;
class Collatz
{
  static void Main()
  {
    Console.Write("\n\tMata in ett positivt heltal:\t";
    int number = int.Parse(Console.ReadLine());

    Console.Write("\n" + number);  // Startvärde
    while (number != 1)            // Sålänge talet inte är 1
    {
      if (number % 2 == 1)         // Om talet är udda, gångra
         number = 3 * number + 1; // det med 3 och addera 1
      else                         // Om talet är jämnt,
         number = number / 2;      // dela med 2
      Console.Write("\t" + number);
    }
    Console.WriteLine("\n");
  }
}
```

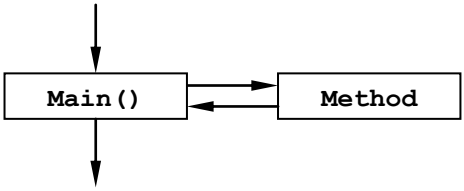
I följande körexempel matas in ett tresiffrigt startvärde. Du kan försöka med andra.

```
        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
```

Metoder och program i C#



De flesta känner till begreppet *funktion* från matematiken. Där kan en funktion t.ex. beskrivas med en formel y = f(x) som beräknar ett värde y utgående från ett annat värde x. Även i programmering finns den matematiska synen på funktion som underliggande koncept och historisk utgångspunkt. Men under tiden har den fått en bredare tolkning då den tillämpats på all datoriserad problemlösning.

En *metod* är en funktion som definieras i en *klass*. I objektorienterade programmeringsspråk är metoder inkapslade i klasser. I C# är detta obligatoriskt. Därför finns det i C# till skillnad från C++ inga fristående funktioner. Bortser man från denna överordnade struktur och ser på det ”inifrån”, är funktioner och metoder identiska.

> En metod i C# är en namngiven kodmodul (ett antal satser) i en klass som utförs när metoden anropas. Vid anropet kan den ta emot indata, s.k. parametrar, bearbeta dem och returnera utdata, s.k. returvärde.

Som ”ett antal satser” är en metod en del av en klass som isoleras och skrivs separat som en anropbar modul för att kunna användas även i andra klasser. Man kan jämföra en metod med en ”svart låda” i vilken man stoppar in indata och får ut utdata: Indata kallas även *parametrar* och utdata *returvärde*:



En metod kan ha 0, 1 eller flera parametrar. Den kan ha 0 eller 1 returvärde. En metod kan alltså inte ha flera returvärden. Både parametrarna och returvärdet kan vara tal, tecken, strängar, sanningsvärden eller referenser till objekt. Metoden bearbetar de ev. inkommande parametrarna på ett visst sätt och returnerar ev. ett värde.

Det finns metoder *med* och sådana *utan* returvärde. De senaste kallas för **void**-metoder.

Vi har hittills använt några C#-biblioteksmetoder, t.ex. **Console.Write()**, **Console.WriteLine()**, **Console.Read()**, **Console.ReadLine()**, **int.Parse()**, … utan att behöva veta hur de var kodade, därför: ”svarta lådor”. De var förprogrammerade åt oss och vi använde dem bara för att åstadkomma vissa funktionaliteter. I detta avsnitt ska vi nu lära oss att själva skriva metoder. Men en metod som vi redan har skrivit själva – och det har vi gjort i alla våra programexempel – är metoden **Main()**, för den är obligatorisk. Så här definieras *C# program*:

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

Modularisering av Collatz

Här vill vi modularisera programmet **Collatz** på sid 40. Dvs vi vill separera en del av koden som känns meningsfullt att isolera och skriva den i en metod. Vilken del det ska vara är inte självklart. Nedan ser du ett förslag för ett sådant beslut:

```
// Collatz_mod.cs
// Deklarerar klassen Collatz_mod och definierar i
// den metoden Collatz som utför Collatz algoritmen
using System;

class Collatz_mod
{
   public static void Collatz(int n)  // n formell parameter
   {
      while (n != 1)                  // Så länge talet inte är 1
      {
         if (n % 2 == 1)              // Om talet är udda gångra
            n = 3 * n + 1;            // det med 3 och addera 1
         else                         // Om talet är jämnt,
            n = n / 2;                // dela med 2
         Console.Write("\t" + n);
      }
   }
}
```

Som man ser har vi isolerat algoritmens kärna som utgör själva logiken i det hela, dvs det som Collatz algoritmen väsentligen innehåller och låtit alla andra tekniska detaljerna vara utanför, t.ex. inläsningen av startvärdet, utformningen av utskriften osv. Alla dessa delar stannar kvar i metoden **Main()** som anropar metoden **Collatz()** exakt på samma ställe som koden för Collatz algoritmen stod. Nedan ser vi dessa delar:

```
// Collatz_Test.cs
// Läser in ett positivt heltal number och anropar
// metoden Collatz i klassen Collatz_mod som tar in
// number som parameter och utför Collatz algoritmen
using System;

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

      Console.Write("\n" + number);     // Startvärde
      Collatz_mod.Collatz(number);      // Anrop av metoden
                                        // Collatz med aktuell
                                        // parameter number
      Console.WriteLine("\n");
   }
}
```

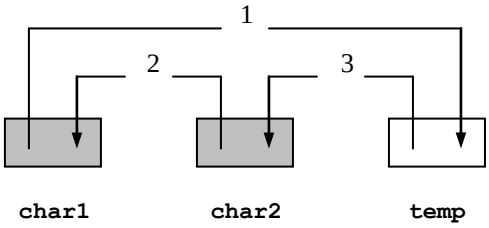
Det modulatiserade programmet ovan producerar samma utskrift som det ursprungliga programmet **Collatz** på sid 41. Bara att ”det modulatiserade programmet” till skillnad från tidigare nu består av *två* klasser, lagrade i *två* filer: **Collatz_mod.cs** och **Collatz_Test.cs**. Därför måste båda filerna laddas i samma projekt i Visual Studio när man kör programmet. Annars kan den avgörande satsen

```
Collatz_mod.Collatz(number);
```

dvs anropet av metoden **Collatz()** inte hitta klassen **Collatz_mod** som innehåller metodens definition. Vi har ju separerat den från **Main()**. Den står i en annan klass som i sin tur finns i en annan fil. Båda filer utgör *ett* program och därmed också *ett* projekt i Visual Studio. Det är ju just meningen med modularisering. Nu kan man anropa metoden **Collatz()** även från alla andra program som man ev. skriver, även från sådana som andra skulle skriva. Metoden **Collatz()** har blivit en generell modul som alla utvecklare kan använda sig av.

1.10 Algoritm för platsbyte

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



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

```
// MiniSort.cs
// Läser in 2 tecken och sorterar dem i teckentabellens ord-
// ning med hjälp av en algoritm för platsbyte av två objekt
using System;
class MiniSort
{
   static void Main()
   {
      char char1, char2, temp;
      Console.Write("\n\tTvå osorterade tecken:\n\n\t" +
              "Mata in två tecken skilda med mellanslag:\t");
      string text = Console.ReadLine();

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

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

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

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

```
Två osorterade tecken:

Mata in två tecken skilda med mellanslag:       Z A

De två tecknen sorterade:                       A Z
```

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

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

Försök att modularisera MiniSort

I programmet **MiniSort** (sid 44) lyckades vi att implementera algoritmen som kan användas för att sortera även större datamängder, eftersom en sådan algoritm bygger på sortering av två objekt. Men för att kunna göra det måste vi separera den från det aktuella program som vi testade algoritmen i, dvs vi måste modularisera den och skriva den som en separat metod. Detta ska vi försöka göra nu. Så här skulle en sådan metod se ut, när vi separerar koden som utgör algoritmen från **MiniSort**. På engelska kallas denna algoritm för *Swap* eller *Swapping*.

```
// NoSort.cs
// Klass med metoden TrySwap() som tar in 2 tecken t1 och t2
// och byter plats på dem enligt algoritmen MiniSort (sid 44)

class NoSort
{
   public static void TrySwap(char t1, char t2)
   {
      char temp;
      if (t1 > t2)
      {
         temp = t1;                // Algoritm för platsbyte
         t1   = t2;                // av de två tecknen
         t2   = temp;              // t1 och t2
      }
   }
}
```

Algoritmdelen av **MiniSort** (sid 44) har flyttats till en metod där **t1** och **t2** är *formella parametrar*[*]. Så kallas parametrar som skrivs i en metods *definition* till skillnad från de *aktuella parametrar* som skrivs i metodens *anrop*. Metoden **TrySwap** anropas i följande program med de aktuella parametrarna **char1** och **char2**:

```
// NoSortTest.cs
// Läser in 2 tecken char1 och char2, skickar dem till meto-
// den TrySwap() i klassen NoSort som ska sortera dem
using System;
class NoSortTest
{
   static void Main()
   {
      char char1, char2;
      Console.Write("\n\tTvå osorterade tecken:\n\n\t" +
         "Mata in två tecken skilda med mellanslag:\t");
      string text = Console.ReadLine();

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

      NoSort.TrySwap(char1, char2);  // Metodanrop

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

Att vi kallar klassen som definierar metoden **TrySwap** för **NoSort** förstår man när man testkör programmet **NoSortTest**. Koden kan både kompileras och exekveras. Det finns inget syntax- eller annat fel i programmet. Det är bara att ingen sortering sker. Tecknen förblir osorterade. Matar man in dem i fel ordning skrivs de ut även i fel ordning – till skillnad från programmet **MiniSort**.

Följande körexempel visar att programmet inte gör som vi vill:

```
        Två osorterade tecken:

        Mata in två tecken skilda med mellanslag:       Z A

        De två tecknen sorterade:                       Z A
```

Testa gärna själv. Och om du tror att det beror på att de formella parametrarna **t1** och **t2** i metoden **TrySwap** har andra namn än de aktuella **char1** och **char2** i programmet **NoSortTest** prova gärna att välja samma namn i båda. Det är inte fel ur

[*] Andra beteckningar som förekommer i litteraturen är *anropsparametrar* eller *argument*. Speciellt *argument* används ofta som är en inkörd matematisk term: T.ex. är **√3** ett anrop av funktionen f(**x**) = **√x** där **x** är – i matematiska termer – *variabeln* och **3** *argumentet*. I programmeringen brukar vi kalla **x** för den *formella* och **3** för den *aktuella* parametern.

varken kompilerings- eller exekveringssynpunkt. Bara att det inte hjälper att sortera tecknen.

Felet är ett tanke- resp. ett kunskapsfel, om man nu kan beteckna det så. Vi har nämligen inte tillräckliga kunskaper för att förstå vad som händer när man modulariserar, dvs separerar kod och lägger den i två olika moduler. Närmare bestämt vet vi inte exakt *hur* parametrarna överförs från den ena till den andra modulen. Därför behandlar vi i nästa avsnitt denna fråga. Det finns nämligen inte bara i C# utan i alla programmeringsspråk *olika* mekanismer för överföring av parametrar mellan en metods definition och dess anrop. Avgörande för valet mellan dessa mekanismer är parametrarnas datatyper. Vi kommer att precisera detta i nästa avsnitt.

1.11 Parameteröverföring i metoder

I det här avsnittet ska vi lära oss *på vilket sätt* parametrar överförs mellan metoder. Det finns som sagt olika typer för parameteröverföring. En av dem är *värdeanrop* (*Call by Value*) som demonstreras i följande program. En annan heter *referensanrop* (*Call by reference*) och tas upp efteråt.

Värdeanrop (Call by value)

```
// CallByVal.cs
// Demonstrerar Värdeanrop: Vid metodanrop överförs VÄRDENA
// De formella parametrarna (kopior) ändras i metoden
// Ändringen påverkar inte aktuella parametrarna (originalen)
using System;

class CallByVal
{
  static void Main()
  {
    int hour = 5, min = 35, sec = 49;

    Console.WriteLine("\nI Main() FÖRE anrop av metod:\ttim="
                   + hour + ", min=" + min + ", sec=" + sec);

    int total = totalsek(hour, min, sec); // Anrop av metod:
                                          // De aktuella pa-
                                          // raparametrarnas
                                          // VÄRDEN skickas
    Console.WriteLine("\nI Main() EFTER anrop av metod:" +
       "\ttim=" + hour + ", min=" + min + ", sec=" + sec +
       "\n\t\t\t\tger " + total + " sekunder totalt." +
       "\nVÄRDEANROP:\n\nÄndringen av" + "  de formella " +
       "parametrarna (kopior)\npåverkar inte de " +
       "aktuella parametrarna (originalen).\n") ;
  }
/*********************************************************/
  static int totalsek(int t, int m, int s)
  {
    Console.WriteLine("\n\tI metoden FÖRE ändringen:\n\tt=" +
                                t + ", m=" + m + ", s=" + s);
    int resultat = 3600 * t + 60 * m + s;
    t = m = s = 0;                        // Ändring av formella
                                          // parametrar
    Console.WriteLine("\n\tI metoden EFTER ändringen:\n\tt="
                              + t + ", m=" + m + ", s=" + s);
    return resultat;
  }
/*********************************************************/
}
```

Varför har vi valt andra namn för de aktuella **hour, min, sec** än för de formella parametrarna **t, m, s** fast de lagrar samma värden? Båda representerar timmar, minuter och sekunder. Frågan är: Lagras dessa värden i 3 eller 6 minnesceller? Om det är 3 vore valet av samma namn motiverat, därför att de lagrar samma värden. Men om det är 6 vore det bättre att återspegla verkligheten även i koden genom att välja olika namn för de aktuella än för de formella parametrarna.

I exemplet ovan läses in de i **Main()**. De formella parametrarna – i vårt exempel **t, m, s** – måste alltid vara variabler som definieras i metoden **totalsek()**:s parameterlista när denna skapas. Sina värden får de *första gången* inte tilldelade i metodens kropp utan från de aktuella parametrarna vid metodens anrop. Sedan ändras deras värden i metoden: De sätts allihop till 0 för att testa vilken påverkan denna ändring har på de formella parametrarna. Men för att ändå kunna få resultatet med de ursprungliga värdena beräknas antalet totalsekunder och sparas undan i variabeln **resultat** som slutligen returneras från metoden. Innan dess skrivs ut värden som ändrats till 0.

I **Main()** skriver vi ut de aktuella parametrarnas värden före och efter anropet av metoden för att se om de formella parametrarnas ändring i metoden påverkar de aktuella parametrarna. Följande körexempel visar att detta inte är fallet:

```
I Main() FÖRE anrop av metod:   hour=5, min=35, sec=49

        I metoden FÖRE ändringen:
        t=5, m=35, s=49

        I metoden EFTER ändringen:
        t=0, m=0, s=0

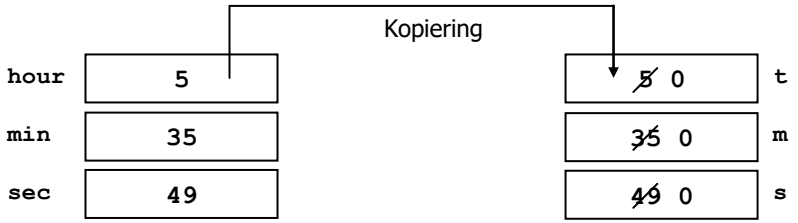
I Main() EFTER anrop av metod:  hour=5, min=35, sec=49
                                ger 20149 sekunder totalt.
VÄRDEANROP:

Ändringen av  de formella parametrarna (kopior)
påverkar inte de aktuella parametrarna (originalen).
```

Körexemplet visar att de formella och aktuella parametrarna har var sitt eget liv. Det enda som relaterar dem till varandra är att de tar över värdena från varandra. Ändringen av de formella parametrarna påverkar inte alls de aktuella parametrarna. Av detta kan man dra slutsatsen att **hour, min, sec** och **t, m, s** är två olika uppsättnigar variabler. De lagras i 6 olika minnesceller. Även om vi skulle välja samma namn för dem – vilket vore tillåtet då de ligger i två olika metoder och därmed i två olika block – kommer namnen fortfarande beteckna 6 olika minnesceller. Även om beteckning är av sekundär betydelse vill vi i fortsättningen välja andra namn för de aktuella än för de formella parametrarna för att återspegla denna verklighet. Kodens läsare ska inte luras som om de vore samma variabler pga namnvalet.

En annan slutsats av körningen ovan är: Parameteröverföringen mellan metoderna **`totalsek()`** och **`Main()`** realiseras genom kopiering av värdena från de aktuella till de formella parametrarna. Denna parameteröverföringsmetod kallas *värdeanrop* därför att det är själva *värden* som kopieras över när metoden aropas. Minnesbilden av värdeanrop ser ut så här:

Värdeanropets minnesbild:



Ändring av kopiorna, de formella parametrarna `t, m, s` påverkar inte originalen, de aktuella parametrarna `hour, min, sec`.

Vid denna parameteröverföringsmetod skapas alltid en *dubbel* uppsättning av minnesceller: 6 om vi har 3 parametrar. Därför leder värdeanrop oundvikligen till fördubblad minnesåtgång. Datatypen till respektive parameter är avgörande för den automatiska tillämpningen av värdeanrop. Det gäller följande regel:

> I C# väljs automatiskt värdeanrop (Call by Value) för parameteröverföring vid metodanrop, om parametern är av enkel datatyp.

Fördubblingen av minnesåtgången anses inte som ett stort problem eftersom enkla datatyper i alla fall tar upp relativt litet minnesutrymme. För datatyper som kräver större minnesutrymme används en annan teknik som undviker denna fördubbling och som heter *referensanrop*.

Ur minnessynpunkt är förstås fördubblingen av minnesåtgången en nackdel. Men värdeanrop har även fördelen att just pga minnesbilden ovan de formella och de aktuella parametrarna har var sitt liv och inte påverkar varandra. I vissa sammanhang är detta önskvärt, i andra inte. Så, beroende på applikationen kan man välja bland de två parameteröverföringsmetoderna värde- och referensanrop genom att välja rätt datatyp till sina parametrar. Enkel datatyp leder automatiskt till värdeanrop. Vilken datatyp som automatiskt leder till referensanrop ska vi ta upp på de följande sidorna.

Referensanrop (Call by reference)

Värdeanrop använder sig av kopiering av parametervärdena till nya minnesceller och tillämpas när parametrarna är enkla datatyper. Nackdelen med värdeanrop är

att den medför fördubbling av minnesåtgången. Alternativet till det är *referensanrop* som överför minnesadressen istället för värdet och där man slipper denna nackdel. Referensanrop är relaterad till datatypen referens som behandlades tidigare varifrån också namnet härstammar. Anledningen är att parametrarnas datatyp automatiskt styr valet av överföringsmetoden. Det gäller nämligen:

> I C# väljs automatiskt referensanrop (Call by reference) för parameteröverföring vid metodanrop, om parametern är av datatypen referens.

Samtidigt kommer vi att se att det för vissa problem t.o.m. är nödvändigt att använda referensanrop då det inte går att modularisersa dem med värdeanrop. Man vill t.ex. skicka vissa parametrar till en metod där de ändras och man vill få tillbaka ändringen till huvudprogrammet. Som exempel tar vi:

Modularisering av MiniSort

På sid 46 lyckades vi inte att modularisera **`MiniSort`** (sid 44). Det berodde på att metoden vi skrev tillämpade värdeanrop pga att dess parametrar var deklarerade till den enkla datatypen **`char`**. Ändringen av de formella parametrarna **`t1`** och **`t2`** i metoden **`TrySwap`** (kopior) påverkade inte de aktuella parametrarna **`char1`** och **`char2`** (originalen). De förblev oförändrade, se värdeanropets minnesbild (sid 50). Det var ju de som vi skrev ut i **`Main()`** där vi anropade metoden. Vi skrev alltså ut **`char1`** och **`char2`** som *inte* var ändrade, medan vi aldrig skrev ut **`t1`** och **`t2`** som *var* ändrade. Vill vi ha ändringen kvar i **`Main()`** måste vi använda referensanrop genom att deklarera våra parametrar till datatypen **`ref char`**, se referensanropets regel ovan. Det gör vi nu:

```
// Swapping.cs
// Klass med metoden Swap() som tar in 2 tecken och byter
// plats på dem om de är i fel ordning enligt teckentabellen
// De ombytta parametrarna i Swap() blir även ombytta i den
// anropande metoden pga parametrarna är deklarerade som
// referenser med det reserverade ordet ref: Referensanrop

class Swapping
{
   public static void Swap(ref char t1, ref char t2)
   {
      char temp;
      if (t1 > t2)
      {
         temp = t1;                // Algoritm för platsbyte
         t1   = t2;                // av de två teckenvärdena
         t2   = temp;              // t1 och t2
      }
   }
}
```

Bearbetningsdelen av **MiniSort** (sid 44) har flyttats till en **void**-metod. Parametrarna **t1** och **t2** är definierade som referenser. De tar inte emot några teckenvärden från **char1** och **char2** (se nedan) utan endast deras adresser. **t1** och **ref char1** är två olika referenser till samma värde **char1**. Samma sak är det med **t2** och **ref char2**. När värdena ändras i metoden genom referenserna **t1** och **t2** kan ändringen ses i **Main()** med **char1** och **char2**:

```
// CallByRef.cs
// Läser in 2 tecken, skickar dem till metoden Swap() i klas-
// sen Swapping som sorterar dem i teckentabellens ordning
// Ändringen är synlig i Main() pga referensanrop som påtvin-
// gas med ref: adresserna överförs vid anrop, inte värdena
using System;

class CallByRef
{
   static void Main()
   {
      char char1, char2;
      Console.Write("\n\tTvå osorterade tecken:\n\n\t"      +
              "Mata in två tecken skilda med mellanslag:\t");
      string str = Console.ReadLine();

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

      Swapping.Swap(ref char1, ref char2);      // Metodanrop

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

Metoden **Swap()** ställer i rätt ordning tecken som är inmatade i fel ordning vilket en körning av ovanstående program visar:

```
        Två osorterade tecken:

        Mata in två tecken skilda med mellanslag:       Z A

        De två tecknen sorterade:                       A Z
```

Gör gärna följande test: Ta bort **ref** från definitionen av båda parametrarna i parameterlistan av metoden **Swap()**, så att **t1** och **t2** blir vanliga **char**-variabler. Ta även bort **ref** från de aktuella parametrarna i anropet av metoden **Swap()** i **Main()** så att värdena skickas och inte adresserna. Du kommer inte få tecknen sorterade i rätt ordning om du matar in dem i fel ordning. Anledningen är att genom borttagningen av **ref** blir **t1** och **t2** variabler av *enkel* datatyp så att värdeanrop tillämpas automatiskt. Ändringen av **t1** och **t2** i metoden kommer inte att påverka **char1** och **char2** i **Main()**.

1.12 In- och utparametrar

Nu har vi lärt oss en hel del om metoder, med och utan returvärde, med en, flera eller inga parametrar, värde- och referensanrop osv. Ändå kan vi inte returnera *flera* värden från en metod. Det beror på att alla metoder i C# returnerar endast ett eller inget värde. Men för att vara mer noggrant, borde vi lägga till *med* **return**-*satsen*. Begreppet *returvärde* används i programmeringsterminologin endast för värden som skickas med **return**-satsen via metodnamnet. I denna bemärkelse finns det inga metoder med *flera* returvärden. Men metodens gränssnitt mot omgivningen dvs mot andra metoder är inte begränsad till metodnamnet. Även parameterlistan tillhör gränssnittet och kan användas för kommunikation med andra metoder. Hittills har denna kommunikation varit *enkelriktad*: Våra parametrar importerade data bara in i metoden. Frågan är: Kan man inte använda dem även för export av data ut ur metoden? I så fall skulle vi kunna få tillbaka även *flera* värden från en metod genom att använda *flera* parametrar. Detta är möjligt fast man kallar sådana data inte längre för returvärden då de inte skickas med **return**-satsen via metodnamnet, utan via parametrarna. De kallas för *utparametrar*. Hittills har vi använt bara *in*parametrar. I detta avsnitt ska vi lära känna *ut*parametrar. Det enda som behövs för att kännetecknа en parameter som *ut*parameter är nämligen att definiera den i parameterlistan som *referens* vilket kan göras med **ref** eller **out**.

I följande metod finns det en inparameter som tillför metoden ett värde och fem utparametrar vars värden exporteras ur metoden. De kommer in i metoden oinitierade, initieras där och används sedan i **Main()** som anropar metoden. I själva verket är utparametrarna endast referenser till de aktuella parametrarna i **Main()**. Där är de endast definierade. I metoden sker initieringen med referenserna.

```
// OutParam.cs
// Tar in växelbeloppet a och delar upp det i antalet t 10-
// kronor, f 5-kronor, o 1-kronor, h 50-öringar och resten r
// i ören. Endast b är en inparameter pga enkel datatyp t, f,
// o, h och r är utparametrar pga referensdatatypen out int
class OutParam
{
  public static void Change(double a, out int t, out int f,
                                      out int o, out int h,
                                                 out int r)
  {
    int total = (int) (a * 100);                // Växeln
    t =    total / 1000;                        // 10-kr
    f =   (total % 1000) / 500;                 // 5-kr
    o =  ((total % 1000) % 500) / 100;          // 1-kr
    h = (((total % 1000) % 500) % 100) / 50;    // 50-öringar
    r = (((total % 1000) % 500) % 100) % 50;    // rest i ören
   }
}
```

Den reala bakgrunden till metoden är följande problem: I en automat erbjuds vissa varor. Man väljer en vara och stoppar in en viss summa pengar, i regel mer än varan kostar. Sedan ska automaten ge tillbaka växelpengar vilket endast är möjligt med ett antal myntslag som är föreskrivna i automaten. Låt oss säga det är 10-, 5-, 1-kr och 50-öringar. I så fall måste växelbeloppet omvandlas till detta mynt"system". Just denna beräkning utförs av **`void`**-metoden **`Change()`** ovan. Men hur genomförs omvandlingen med de uttryck för **`t, f, o, h`** och **`r`** som står i metoden? Följande algoritm löser problemet:

Algoritm för omvandling av ett belopp till olika myntslag

Eftersom denna algoritm endast fungerar för heltal måste växelbeloppet **`b`** som är en **`double`** först konverteras till **`int`**, vilket görs i metoden **`Change()`**:s första sats explicit eftersom automatisk typkonvertering inte kan omvandla nedåt i datatypshierarkin. Växelbeloppet i kronor och ören konverteras till ett rent örebelopp som lagras i **`int`**-variabeln **`total`**. I fortsättningen står alltså det givna växelbeloppet i variabeln **`total`**.

1. För att få antalet 10-kronor divideras **`total`** med **`1000`** då 10-kr är **`1000`** ören:

```
t = total / 1000;
```

Hur många gånger ryms **`1000`** – eller 10-kronor – i **`total`**? Det antalet tilldelas till **`t`**. Eller med andra ord: **`1000`** dras av från **`total`** så många gånger tills resten blivit mindre än **`total`**. Det antalet som tilldelas till **`t`** blir antalet 10-kronor. Divisionen ovan är inte vanlig division utan heltalsdivision då både **`total`** och **`1000`** är heltal. Dvs **`total`** divideras med **`1000`**, resultatet tas, resten ignoreras, t.ex. **`6975/1000`** ger **`6`**. Se körexemplet på nästa sida. Resten **`975`** ignoreras här, men används i fortsättningen.

2. För att få antalet 5-kronor divideras resten som blev kvar från punkt **1** med **`500`** då 5-kronor är **`500`** ören: `f = (total % 1000) / 500;`

"Resten som blev kvar från punkt **1**" är just **`(total % 1000)`**. Här används en annan operator som är besläktad med heltalsdivision, nämligen modulooperatorn **`%`** som inte har att göra med procenträkning utan ger *resten* vid heltalsdivision. T.ex. **`6975 % 1000`** ger **`975`**. Efter att ha dragit av alla 10-kronor från **`total`** divideras resten med **`500`** för att få reda på hur många 5-kronor som finns i **`total`**. T.ex. **`975/500`** ger **`1`**. Resultatet av denna division ges till **`f`**, resten ignoreras och används i fortsättningen.

I ytterligare tre steg skulle man kunna förklara de övriga formlerna för beräkning av **`e`**, **`h`** och **`r`**. Men nu har mönstret i algoritmen kommit fram: Man tar förra stegets formel, ersätter **`/`** med **`%`** och lägger till en heltalsdivision med den nya enhetens örebelopp. I det allra sista steget däremot, där man är ute efter allra sista resten i öre, måste **`%`** användas hela vägen. Självklart är restörebeloppet inte av praktiskt intresse när automaten inte kan spotta ut det.

För att testa algoritmen ovan anropas metoden **`Change()`** av följande program:

```
// OutparamTest.cs
// Efter inköp av en vara i en automat ska växeln ges till-
// baka i form av ett antal föreskrivna myntslag:
// 10-kr, 5-kr, 1-kr, 50-öringar (och en rest i öre)
// Main() läser in ett växelbelopp, skickar det till metoden
// Change() i klassen OutParam som omvandlar växeln till mynt
using System;

class OutparamTest
{
  static void Main()
  {
    double amount;
    int ten, five, one, half, rest; // Iinitiering behövs ej
    Console.Write("\nAnge ett växelbelopp i kr & ören:  ");
    amount = Convert.ToDouble(Console.ReadLine());

    OutParam.Change(amount, out ten, out five,  // Endast ut-
                            out one, out half,  // paramet-
                                     out rest); // rarnas ad-
                                                // resser skickas
    Console.WriteLine("\n" + amount + " kr =\t"            +
                      ten  + " tio-kronor\n\t\t"           +
                      five + " fem-krona\n\t\t"            +
                      one  + " en-kronor \n\t\t"           +
                      half + " femtio-öring\n\nDet blir\t" +
                      rest + " ören kvar\n");
   }
}
```

Växelbeloppet läses in. Metoden **`Change()`** anropas varvid förutom **`belopp`** de aktuella parametrarna **`ten`**, **`five`**, **`one`**, **`half`** och **`rest`**:s *adresser* skickas. Dessa tas emot i **`Change()`** av **`t`**, **`f`**, **`o`**, **`h`** och **`r`**, referenserna till **`ten`**, **`five`**, **`one`**, **`half`** och **`rest`**. När beräkningen görs där med hjälp av referenserna kan man komma åt resultaten i **`Main()`** därför att **`t`** är en referens till **`ten`**. Samma sak är det med de andra parametrarna.

Ett körexempel visar att vi får tillbaka de värden som beräknas i metoden pga referensanrop som automatiskt tillämpas vid utprametrar av referenstyp.

```
Ange ett växelbelopp i kronor, ören:  69,75

69,75 kr =      6 tio-kronor
                1 fem-krona
                4 en-kronor
                1 femtio-öring

Det blir        25 ören kvar
```

Övningar till kap 1

Läs kap 1 Algoritmer och programmering, 1.1 – 1.3, sid 6-14

Besvara följande frågor:

1.1 Med vilket namn betecknas de språk som de första datorerna programmerades med? Vilka egenskaper hade de? Vad är deras största skillnad till dagens programmeringsspråk?

1.2 Vad bestod den tekniska innovationen av som *John von Neumann* utvecklade 1944?

1.3 Vad karaktäriserar de programmeringsspråk som kallas för *lågnivåspråk?* Varför *”låg”?*

1.4 Vilket var det första *högnivåspråket?* Varför *”hög”?*

1.5 Redogör för skillnaderna mellan begreppen *assemblering, kompilering och interpretering.*

1.6 Nämn ett exempel på programmeringsspråk som använde en av metoderna i fråga 5.

1.7 Vad var det första användningsområdet för programmering?

1.8 Finns det fortfarande kod som används som är skriven i något av de första programmeringsspråken? Nämn några sådana samt deras användningsområde.

1.9 Vilket var det första programmeringsspråk som introducerade *kontrollstrukturer* i programmeringen?

1.10 Vad menas med *deklarativ programmering?* Är C# ett deklarativt språk?

1.11 Nämn några underkategorier till deklarativ programmering.

1.12 Vilken programmeringsfilosofi ligger till grund för den algoritmorienterade synen?

1.13 Beskriv med egna ord *händelsestyrd programmering.* Nämn exempel.

1.14 Vad karaktäriserar det som kallas för *spaghettiprogrammering?* Vad är huvudkritiken mot den?

1.15 Vilket programmeringstekniskt koncept kan ersätta spaghettiprogrammering?

1.16 Vad är den traditionella, procedurala synen på programmering som rådde på 60- och 70-talet?

1.17 Vad är den objektorienterade synen på programmering som kom upp på 80-talet?

1.18 Mellan vilka två programmeringsspråk går historiskt skiljelinjen mellan procedural och objektorienterad programmering? När ungefär inträffade övergången?

1.19 Vad var anledningen till paradigmskiftet inom programutveckling?

1.20 Vilka för- och nackdelar har enligt din åsikt den procedurala synen på programmering? Besvara samma fråga angående den objektorienterade synen

1.21 Följande pseudokod beskriver en algoritm för hårtvätt:

```
Start hårtvätt
Blöt håret
SÅ LÄNGE håret känns smutsigt
    massera in shampo
    skölj
OM solen skiner
    låt håret självtorka
ANNARS
    använd hårtorken
Slut hårtvätt
```

a) Vilka delar av pseudokoden är *instruktioner*, vilka är *villkor* och vilka är *kontrollstrukturer*? Förklara ditt svar.
b) Dela in instruktionerna i huvud- och underinstruktioner.
c) Rita ett flödesschema till pseudokoden ovan.

1.22 Följande algoritm – *Kalle-algoritmen* – är formulerad på vanligt språk:

På vardagar går Kalle upp. Han tvättar sig, om mamman tittar på. På söndagar sover Kalle vidare tills mamman ropar honom till frukost, i så fall gör han som på vardagar.

a) Rita ett flödesschema till Kalle-algoritmen. Anta att lördag är en vardag.

b) Översätt Kalle-flödesschemat till pseudokod.

c) Finns det i Kalle-algoritmen möjligheten till en evighetsloop? När skulle den kunna inträffa? Hur kan den förhindras?

1.23 Är följande pseudokod logiskt identisk med *Kalle-algoritmen* från övn 1.22?

```
Start Kanske_Kalle?
OM  det är söndag
        sover Kalle vidare
    TILLS  mamma ropar till frukost
ANNARS
    går han upp
OM  mamma tittar på
    tvättar han sig
Slut Kanske_Kalle?
```

1.24 Rita flödesschemat till följande pseudokod:

```
Sätt på radion
Välj en kanal och  lyssna
SÅ LÄNGE  du inte har hittat ett bra program
      byt kanal
      lyssna
Fortsätt att lyssna på det valda programmet
Stäng av radion
```

1.25 Skriv ett C# program som läser in två heltal, multiplicerar dem med varandra och skriver ut resultatet blandat med förklarande text. Om du t.ex. matar in `3` till det första och `4` till det andra heltalet, ska programmet skriva ut: **`3 gånger 4 är 12.`** Utveckla programmet vidare med ytterligare räkneoperationer, kanske så småningom till en liten kalkylator, se ***1.29 Kalkylatorn (Projektuppgift 1)***.

1.26 Rita ett flödesschema till följande pseudokod:

```
Start Vinterklädsel_1
Läs av temperaturen
OM  temperatur <  0
    ta sjal, mössa och handskar
ANNARS OM  temperatur <  5
    ta sjal och mössa
ANNARS OM  temperatur < 10
    ta sjal
ANNARS
    slipper du vinterklädsel
Slut Vinterklädsel_1
```

Använd dina programmeringskunskaper för att koda pseudokoden ovan och flödesschemat du ritat, till ett C# program. Läs in ett värde för temperatur och låt programmet avgöra val av klädsel genom att skriva ut "Ta sjal, mössa, handskar... " eller liknande. För kontrollstrukturen flervägsval kan du använda **`if-else`**-stegen som kodas i C# på samma sätt som i C++.

1.27 Algoritmen i övn 1.26 ovan kan formuleras med följande pseudokod:

Start *Vinterklädsel_2*
Läs av temperaturen
VÄLJ *fall* ur
    temperatur < *0*: ta sjal, mössa och handskar
    temperatur < *5*: ta sjal och mössa
    temperatur < *10*: ta sjal
    Annars: slipper du vinterklädsel
Slut *Vinterklädsel_2*

Rita flödesschemat till pseudokoden ovan och undersök den logiska likheten mellan flödesscheman i övn 1.26 och övn 1.27.

1.28 Collatz algoritmen har modulariserats med **`void`**-metoden **`Collatz()`** som är definierad i klassen **`Collatz_mod`**, se sid 42. Modularisera Collatz algoritmen med en metod med returvärde istället. Dvs definiera en metod **`public static int Collatz()`** som endast returnerar ETT tal i Collatz-sekvensen. Anropa metoden från en annan klass' **`Main()`**.

Tips: Placera loopen samt utskriftssatsen i huvudprogrammet som anropar metoden. För att dataflödet mellan loopen och metoden ska fungera tillämpa referensanrop.

1.29 ***Kalkylatorn (Projektuppgift 1)*** I denna uppgift ska skapa en klass **`Calculator`** skapas som stödjer följande funktionaliteter: addition, subtraktion, multiplikation, division och potentiering samt att kunna ange det största och minsta av två inmatade tal.

Dessutom ska din kalkylator vara igång kontinuerligt tills användaren väljer att stänga av den, vilket innebär att du måste lägga in en loop. De olika räkneoperationerna ska definieras i separata metoder och anropas i **`Main()`**.

Följande metoder ska definieras i klassen **`Calculator`**:

```
public double Add(double operand1, double operand2)
{
      // Additon av operand1 och operand2
}

public double Sub(double operand1, double operand2)
{
      // operand1 - operand2
      // Även subtraktion av negativa tal ska vara möjligt
}

public double Mult(double operand1, double operand2)
{
      // Multiplikation av parametrarna
}
```

```
public double Div(double operand1, double operand2)
{
      // operand1 / operand2
      // Division med 0 får ej förekomma (operand2 != 0)
}

public double Potens(double operand1, double operand2)
{
      // Beräkning av potens: operand1 upphöjt till operand2

}

public double max(double operand1, double operand2)
{
      // Returnera det större värdet av operand1 och operand2
      // Här kan du använda dig av den födefinierade metoden
      // Math.Max(double a, double b) för att snabbt
      // avgöra vilken av operanderna som är större
}

public double Min(double operand1, double operand2)
{
      // Returnera det mindre värdet av operand1 och operand2
      // Math.Min(double a, double b) kan användas
}
```

Programmet skall exekvera kontinuerligt tills användaren väljer att avsluta körningen. För att åstadkomma detta kan du exempelvis använda dig av en **do**-sats. Kalkylatorn kan avslutas genom att användaren matar in t.ex. tecknet ’q’ (Quit) istället för en operator.

Du får själv bestämma om du vill placera all kod i en fil eller om du hellre skapar en separat fil för klassen **Calculator** med alla ovannämnda metoder och en klass med **Main()** i en annan fil som testar klassen **Calculator**. Det senare är att föredra.

Det är upp till dig om du lägger in kod för att kunna hantera fel inmatning av operator eller andra felaktiga inmatningar.

Kapitel 2

Logik för blivande programmerare

Ämne	Sida	Program
2.1 Logiska operatorer	62	**`AND_OR`**
- Sanningstabeller	64	
2.2 Datatypen **`bool`**	67	**`TruthTab`**
2.3 NEGATION som logisk operator	69	
- Gissa tal med NEGATION	69	**`GuessNEG`**
- Logiska uttryck	71	
2.4 Programserien *Testa lösenord*	73	**`Passwd`**
- Metoden **`Equals()`**	74	
- Kombination av NEGATION, OCH, ELLER	75	**`PasswdCaps`**
- De Morgans lagar	77	
Övningar till kapitel 2	84	

2.1 Logiska operatorer

Att syssla med logik inom programmering är inte så konstigt. Vi har redan gjort det redan i förra kapitlet när vi använde avslutningsvillkor för våra kontrollstrukturer. Logiskt korrekt formulerade avslutningsvillkor är avgörande för hantering av kontrollstrukturer, t.ex. för att undvika evighetsloopar. Begreppet *villkor* har följt oss redan från bokens allra första kapitel: Då diskuterades skillnaden mellan *instruktion* och *villkor* i algoritmen Morgonsyssla (sid 21). Medan en instruktion (sats) är ett kommando, ett befäl som måste *utföras* kan ett villkor endast *testas* för att fatta ett beslut, träffa ett val mellan olika alternativ, t.ex. för att avgöra om en loop ska fortsätta eller upphöra. Alla villkor vi använt hittills i våra program med kontrollstrukturerna **`if`**, **`if-else`**, **`do`**, **`while`** och **`for`** har varit s.k. *enkla villkor*. Ett villkor heter *enkelt* om dess sanningsvärde – sant eller falskt – kan bestämmas *direkt*, utan att blanda in andra villkor eller använda s.k. *logiska operatorer* som vi ska lära känna i detta avsnitt. Exempel på enkla villkor är **`tal == 0`**, **`i < 5`** eller **`a <= 9`**. Enkla villkor kan bildas med jämförelseoperatorer. Nu ska vi gå ett steg vidare:

När man sätter ihop enkla villkor och kombinerar dem med varandra uppstår *sammansatta villkor*. Men hur ska man sätta ihop två enkla villkor? Det kan endast göras om det finns något som binder samman dem. Detta ”något” kallas för en *logisk operator*. Exempel på *logiska operatorer* är det logiska OCH som i C# kodas med **`&&`** och det logiska ELLER som symboliseras av dubbeltecknet **`||`**. De opererar på *två* enkla villkor och returnerar ett sanningsvärde. Man kallar dem för *operatorer*, jämförbara med aritmetiska operatorer därför att även de ”räknar” på ett visst sätt, bara att deras operander inte är tal utan villkor och deras returvärde inte heller är tal utan ett sanningsvärde. Man sätter dem mellan två enkla villkor och får på detta sätt ett sammansatt villkor. Här är några enkla exempel på sammansatta villkor bildade med de logiska operatorerna OCH (**`&&`**) och ELLER (**`||`**):

```
(number == 0)     || (number > 0)
(temp <= 10)      && (temp >= 25)
(guessedNo < 17) || (guessedNo > 17)
```

Vi kan se att sammansatta villkor är kombinationer av enkla villkor, logiska operatorer och parenteser. Att de returnerar ett sanningsvärde beror på att de bildar *ett* sammansatt villkor av två enkla. Man kan jämföra det med att bilda ett tal av två genom att sätta **+** eller **–** mellan dem. Sammansatta villkor skiljer sig från enkla genom inblandningen av logiska operatorer. Deras sanningsvärde kan inte längre bestämmas direkt utan är beroende av de logiska operatorernas logiska innebörd.

När behöver man sammansatta villkor? Programmet **`AND_OR`** på nästa sida visar att det är ganska enkla, vardagliga situationer där sammansatta villkor förekommer som kräver användningen av logiska operatorer. Programmet använder sammansatta villkor för att lösa ett problem som liknar *Gissa tal*: Ett val mellan tre alternativ. Trevägsvalet ska nu lösas utan **`switch`**-satsen med en kombination av nästlad

if-else och sammansatta villkor med logiska operatorer – ytterligare en generell metod att programmera flervägsval som vi tidigare hade nämnt (sid 186).

```
// AND_OR.cs
// Hämtar datorns tid och avgör om det är dags för dagens
// lunch: Trevägsval med sammansatta villkor och de logiska
// operatorerna OCH (&&) och ELLER (||). Klassen DateTime:s
// egenskap (datamedlem) Now ger ett objekt av typ DateTime
// som sätts till datorns aktuella datum och tid.
using System;

class AND_OR
{
  static void Main()
  {
    int hour = DateTime.Now.Hour;     // Tar ut datortidens
    int min = DateTime.Now.Minute;    // timme resp. minut
    Console.WriteLine("\n\tKlockan är " + hour + '.' + min);

    if ((hour >= 11) && (hour < 14))
       Console.WriteLine("\n\tDagens lunch kan serveras:\t");

    if ((hour < 11) || (hour >= 14))
    {
       Console.Write("\n\tDagens lunch kan ej serveras ");
       if (hour < 11)
          Console.WriteLine("eftersom det är för tidigt:");
       else
          Console.WriteLine("eftersom det är för sent:");
    }

    Console.WriteLine("\n\tDagens lunch serveras mellan" +
                                          " kl 11 och 14\n");
  }
}
```

Körs programmet ovan vid olika tidpunkter som motsvarar de tre alternativen *före kl 11*, *mellan 11-14* och *efter kl 14* får man de tre olika utskrifterna nedan:

```
    Klockan är 10.45

    Dagens lunch kan ej serveras eftersom det är för tidigt:

    Dagens lunch serveras mellan kl 11 och 14
```

```
     Klockan är 12.11

     Dagens lunch kan serveras:

     Dagens lunch serveras mellan kl 11 och 14
```

```
Klockan är 14.21
Dagens lunch kan ej serveras eftersom det är för sent:
Dagens lunch serveras mellan kl 11 och 14
```

Det första sammansatta villkoret i programmet ovan är:

```
(hour >= 11) && (hour < 14)
```

Den logiska operatorn **&&** kombinerar de två enkla delvillkoren **hour >= 11** och **hour < 14** till ett sammansatt villkor vars sanningsvärde beror på de båda enkla delvillkorens sanningsvärden samt den logiska innebörden av operatorn **&&**. Parenteserna kring de två enkla delvillkoren kan utelämnas därför de i alla fall evalueras först. Vi har skrivit dem bara för att vara på den säkra sidan vad gäller prioriteten mellan operatorerna. Den intuitiva innebörden av det logiska OCH i vanligt språk är: Om **hour**:s värde är större än eller lika med **11** *och samtidigt* mindre än **14**, så är det sammansatta villkoret sant. Dvs om **hour**:s värde ligger mellan **11** och **14**, är villkoret sant. Det sammansatta villkoret beskriver alltså i det här fallet ett intervall. För att testa om ett värde ligger i ett intervall är ett villkor av sammansatt typ med operatorn **&&** en lämplig konstruktion. I programmet **AND_OR** ska dagens lunch serveras mellan klockan 11 och 14. Före kl 11 eller efter kl 14 ska ingen dagens lunch serveras. Dvs om bara *ett* enkelt delvillkor **hour >= 11** eller **hour < 14** är falskt blir också hela det sammansatta villkoret falskt. För att det sammansatta villkoret ska bli sant måste *båda* delvillkoren vara sanna. Dvs klockan måste vara över (eller prick) 11 och samtidigt före 14.

Den logiska operatorn OCH

Logiken hos operatorn **&&** kunde i exemplet ovan härledas från det vanliga språkets betydelse för ordet OCH. Men hur avgör datorn som inte förstår vanligt språk, sanningsvärdet hos ett villkor av sammansatt typ med den logiska operatorn **&&** ? Hur är denna operator definierad? En sådan allmän definition av operatorn **&&** är lagrad i datorn för att kunna bestämma sanningsvärdet till alla villkor som involverar **&&** vilket förstås gäller för alla logiska operatorer. Precis som det finns definitioner för de aritmetiska operatorerna **+**, **–** , ***** och **/** , som datorn använder för att beräkna aritmetiska uttryck, finns även definitioner för de logiska operatorerna, som datorn använder för att evaluera sammansatta villkor. Att *evaluera* ett villkor betyder att bestämma dess sanningsvärde.

Sanningstabeller

Varje logisk operator definieras med en s.k. *sanningstabell* som definierar de sanningsvärden som gäller för just denna operator – jämförbart med de vanliga räkneoperatorerna, t.ex. multiplikationen som definieras med multiplikationstabellen. Den logiska operatorn OCH:s sanningstabell t.ex. ser ut så här:

OCH:s sanningstabell

```
  p       q              p && q
-------------------------------
true    true             true
true    false            false
false   true             false
false   false            false
```

I sanningstabellen ovan symboliserar **p** *ett enkelt* delvillkor, t.ex. **hour >= 11** och **q** det *andra enkla* delvillkoret, t.ex. **hour < 14**. Då blir **p && q** det sammansatta villkoret, sammansatt av de två enkla delvillkoren med hjälp av operatorn **&&**. Tabellen ska läsas radvis. Första raden (under strecket) säger: Om båda de enkla delvillkoren **p** och **q** har sanningsvärdet **true**, får det sammansatta villkoret **p && q** sanningsvärdet **true**. Den andra raden säger: Om delvillkor **p** har sanningsvärdet **true** och delvillkor **q** sanningsvärdet **false**, får det sammansatta villkoret **p && q** sanningsvärdet **false** osv. Sanningstabellen behandlar alla möjliga kombinationer av värdena **true** och **false** för de enkla delvillkoren **p** och **q**. Det finns sammanlagt fyra sådana kombinationer som är uppställda i tabellens två första kolumner. Resultaten – sanningsvärdena för **p && q** – står i den tredje kolumnen. I och med att tabellen innehåller *alla möjliga* kombinationer, definieras den logiska operatorn **&&** generellt och återspeglar också den intuitiva innebörden av det logiska OCH i vanligt språkbruk, nämligen: Om - och endast om - de *båda* enkla delvillkoren **p** och **q** är sanna, är det sammansatta villkoret **p && q** sant, annars är det sammansatta villkoret falskt.

Liknande gäller för sanningstabellen till den andra logiska operatorn som förekommer i programmet **AND_OR**.

Den logiska operatorn ELLER

Det andra sammansatta villkoret i programmet **OCH_ELLER** är:

```
(hour < 11) || (hour >= 14)
```

Villkoret är sammansatt av de två enkla delvillkoren **hour < 11** och **hour >= 14** med hjälp av den logiska operatorn **||**. Den intuitiva innebörden av det logiska ELLER i vanligt språk är: Om **hour**:s värde är mindre än **11** *eller* större än eller lika med **14**, är det sammansatta villkoret sant, vilket i **AND_OR** innebär att ”Dagens lunch” inte ska serveras före 11 eller efter (eller prick) 14. Det räcker att endast *ett* av delvillkoren antingen **hour < 11** eller **hour >= 14** är sant för att det sammansatta villkoret ska bli sant. Endast om båda är falska, blir resultatet falskt. Man inser att klockan inte samtidigt kan vara före 11 och efter (eller prick) 14, därför används här ELLER och inte OCH. Även här kan logiken hos operatorn **||** härledas från det vanliga språkets betydelse för ordet ELLER, närmare bestämt för

ANTINGEN ELLER. Men den exakta logiska innebörden definieras som vanligt av sanningstabellen:

ELLER:s sanningstabell

```
  p       q             p || q
------------------------------
true    true            true
true    false           true
false   true            true
false   false           false
```

I sanningstabellen ovan står **p** för *ett enkelt* delvillkor, t.ex. **hour < 11** och **q** för det *andra enkla* delvillkoret, t.ex. **hour >= 14**. Operatorn **||** binder samman dessa två enkla delvillkor och bildar det sammansatta villkoret **(hour < 11) || (hour >= 14)**.

Förutom OCH och ELLER är NEGATION en viktig logisk operator. Den negerar sin operand dvs vänder alla dess sanningsvärden till motsatsen. Vi kommer att behandla den logiska operatorn NEGATION senare.

Programmet **AND_OR** hämtar den aktuella tiden från datorn för att avgöra om det är dags för dagens lunch. Fö att göra det kan bibliotekssklassen **DateTime** användas.

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 i sin tur 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.

2.2 Datatypen `bool`

I C# finns möjligheten att definiera logiska variabler med datatypen **`bool`** som är en enkel datatyp och representerar sanningsvärdena sant och falskt. **`bool`** är namngiven efter den engelske matematikern *George Boole* som verkade på 1800-talet och formulerade logikens lagar genom att använda matematisk notation. Variabler av typen **`bool`** kan endast anta sanningsvärdet **`true`** eller **`false`**.

Observera att **`true`** och **`false`** inte är vanliga strängar utan reserverade ord i C# som representerar sanningsvärdena sant och falskt. De är alltså *logiska konstanter* som kan ges till *logiska variabler* dvs variabler av typ **`bool`**. Datatypen **`bool`** tillåter lagringen av sådana variabler. Detta utvidgar programmerarens möjligheter avsevärt. T.ex. kan de sanningstabeller vi ställde upp i förra avsnitt för de logiska operatorerna **`&&`** och **`||`**, även genereras av följande program som använder logiska variabler dvs sådana av den nya datatypen **`bool`**:

```
// TruthTab.cs
// Lagrar sanningsvärden i logiska variabler som deklareras
// med den enkla datatypen bool
// Skriver ut sanningstabellerna till de logiska operatorerna
// && (OCH) och || (ELLER)
using System;

class TruthTab
{
  static void Main()
  {
    bool p, q;          // Deklaration av logiska variabler

    Console.Write("\n  p \t  q\t\tp && q \t\tp || q\n" +
        "-------------------------------------------\n");
    p = q = true;       // Initiering av logiska variabler
    Console.Write(p + "\t" + q + "\t\t " + (p && q) +
                             "  \t\t " + (p || q) + '\n');
    p = true; q = false;
    Console.Write(p + "\t" + q + "\t\t " + (p && q) +
                               "  \t " + (p || q) + '\n');
    p = false; q = true;
    Console.Write(p + "\t" + q + "\t\t " + (p && q) +
                               "  \t " + (p || q) + '\n');
    p = q = false;
    Console.Write(p + "\t" + q + "\t\t " + (p && q) +
                               "  \t " + (p || q) + "\n\n");
  }
}
```

Följande sanningstabeller till operatorerna OCH och ELLER skrivs ut när programmet ovan körs:

```
  p       q             p && q          p || q
-----------------------------------------------
True    True              True            True
True    False             False           True
False   True              False           True
False   False             False           False
```

I programmet **TruthTab** är variablerna **p** och **q** definierade som **bool** och kan därför tilldelas värdena **true** eller **false**. De kallas även för *booleska* variabler vilket är synonym med logiska variabler. Först initieras **p** och **q** båda till **true** (framhävd med vit bakgrund) och skrivs ut i de första två kolumnerna i sanningstabellens första rad (efter rubriken). Sedan kombineras de med varandra i **p && q** samt **p || q** vars sanningsvärden skrivs ut i tabellens tredje och fjärde kolumn. I sanningstabellens andra rad (efter rubriken) upprepas utskriften, men den här gången med en ny tilldelning av **true** till **p** och **false** till **q**. I den tredje och fjärde raden gås igenom de andra kombinationerna **false** till **p** och **true** till **q** samt **false** till båda. Jämför man resultaten med de enskilda sanningstabellerna vi hade ställt upp för de logiska operatorerna OCH och ELLER på sid 65/66 konstaterar man överensstämmelse. Skillnaden är bara att vi då hade endast *påstått* sanningsvärdena och motiverat dem med vår intuitiva uppfattning av logiken hos OCH och ELLER, medan här låter vi programmet *generera* sanningsvärdena till de sammansatta villkoren **p && q** och **p || q** via koden. Man kan också säga, vi låter programmet applicera de fördefinierade operatorerna **&&** och **||** på de fyra möjliga kombinationerna av **true** och **false** och skriva ut deras resultat.

Man kan använda programmet **TruthTab** även för andra sammansatta villkor och få fram deras sanningstabeller genom att skriva in dem i utskriftssatsen istället för **p && q** resp. **p || q**.

2.3 NEGATION som logisk operator

I de föregående avsnitten lärde vi känna de logiska operatorerna OCH och ELLER. Nu ska vi komplettera vår lilla samling av logiska operatorer med NEGATIONen vars symbol är **!** . Men förväxla den inte med utropstecknet som förekommer i *jämförelse*operatorn **!=** som står mellan två *aritmetiska* uttryck. Som exempel tar vi ett Gissa tal-spel som använder slumptal som hemligt tal i dialog med en **do**-loop. Vi ska utveckla spelets logik, speciellt loopens avslutningsvillkor med den *logiska operatorn* NEGATIONen **!** som kan skrivas framför ett logiskt uttryck för att negera det.

```
// GuessNEG.cs
// Gissa tal-spelet med NEGATION
using System;

class GuessNEG
{
  static void Main()
  {
    Random r = new Random();
    int guessedNo, secretNo = r.Next(1, 21);
    bool wrongGuess;      // Deklaration av logisk variabel

    do                    // do-loopen (avslutas med while)
    {
      Console.Write("\n\tGissa ett tal mellan 1 och 20 "  +
                                        (Avsluta med 0):\t");
      guessedNo = int.Parse(Console.ReadLine());
      Console.Write("\n\t");
      wrongGuess = !(guessedNo == secretNo);// Initiering av
      if (guessedNo == 0)                   // logisk variabel
      {
         Console.WriteLine("Avbrott: Programmets hemliga " +
                              "tal var " + secretNo + '\n');
         break;              // Bryter do-loopen
      }
      if (guessedNo < secretNo)
         Console.Write("För LITET, försök igen!\n");
      if (guessedNo > secretNo)
         Console.Write("För STORT, försök igen!\n");
    } while (wrongGuess);     // Fortsätter så länge fel gissat
                              // Stoppar när gissningen är rätt
    if (!wrongGuess)
       Console.Write("\aGrattis, du har gissat rätt!\n\n");
  }
}
```

En körning av programmet ovan kan se ut så här:

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

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

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

Till skillnad från OCH/ELLER som alltid har två operander, har NEGATIONEN endast *en* operand, t.ex. **p**. Negationen sätts *framför* den: **!p**. Sanningsvärdet vänds om: sant blir falskt och falskt blir sant. Därför har **!** följande enkla sanningstabell:

```
  p        ! p
----------------
true       false
false      true
```

I programmet **GuessNEG** är **p** i följande situation villkoret **guessedNo** == **secretNo** som först negeras och sedan tilldelas den logiska variabeln **wrongGuess**:

```
bool wrongGuess;
...
do
{
   ...
   wrongGuess = !(guessedNo == secretNo);
                ←———————   true eller false
   ...
} while (wrongGuess);
```

Variabeln **wrongGuess** deklareras till datatypen **bool**. I **do**-satsen tilldelas den det *logiska uttrycket* **!(guessedNo == secretNo)**, närmare bestämt uttryckets sanningsvärde. **wrongGuess** är sant när **guessedNo** *inte* är lika med **secretNo** dvs när man gissat fel och då fortsätter **do**-loopen. **wrongGuess** är falskt när **guessedNo** är lika med **secretNo** dvs när man gissat rätt och då stoppas **do**-loopen.

Logiska uttryck

Ett *logiskt uttryck* är en kombination av enkla villkor, logiska variabler, de logiska konstanterna **true** och **false**, logiska operatorer och vanliga parenteser som till slut, när det hela evalueras, returnerar ett sanningsvärde. Exempel på *logiska uttryck* är sammansatta villkor. Även **!(guessedNo == secretNo)** är ett logiskt uttryck vars värde är sant om **guessedNo** inte är lika med **secretNo**, annars falskt. I satsen på bilden ovan får den logiska variabeln **wrongGuess** detta värde. I denna sats är = tilldelningsoperatorn som tilldelar värdet **true** eller **false** till variabeln **wrongGuess**, medan == är en jämförelseoperator som returnerar ett sanningsvärde, dvs bestämmer om värdet inom parentesen blir **true** eller **false**.

Observera även skillnaden mellan utropstecknet som förekommer i *jämförelse*-operatorn **!=** och utropstecknet **!** som *logisk* operator. Jämförelseoperatorn **!=** står som ett dubbeltecken (utan mellanslag) *mellan* två *aritmetiska* uttryck, jämför uttryckens talvärden och returnerar ett sanningsvärde. Den logiska operatorn **!** skrivs *framför* ett *logiskt* uttryck och returnerar uttryckets omvända sanningsvärde.

Dubbel negation

I villkoret till **if**-satsen som följer **do**-satsen skrivs negationsoperatorn **!** *framför* den logiska variabeln **wrongGuess** för att negera den:

```
if (!wrongGuess)
{   ...
```

Nu sätter vi in det logiska uttryck som via satsen **wrongGuess = !(guessedNo == secretNo);** hade tilldelats **wrongGuess**, i **if**-satsens villkor:

```
if (!(!(guessedNo == secretNo)))
{   ...
```

Därmed träffar nu två negationer på varandra som enligt negationens sanningstabell tar ut dvs neutraliserar varandra. *Dubbel negation* av ett sanningsvärde reproducerar sanningsvärdet vare sig det är **true** eller **false**, i symbolisk form **!(!p) = p**, vilket är en allmän logisk lag som gäller för alla utsagor **p**. Man kan också säga: NEGATIONen är som operator sin egen *invers* dvs sin egen *motsatt* operator. Löser vi upp den dubbla negationen ovan enligt denna lag så avslöjas **if**-villkorets logiska innebörd:

```
if (guessedNo == secretNo)
{   ...
```

Dvs om spelets användare gissar rätt kommer **`if`**-satsens kropp att utföras vilket innebär att ”Grattis”-meddelandet skrivs ut. Detta åstadkommer man i programmet **`GuessNEG`** genom att negera den logiska variabeln **`GissaFel`** – samma variabel som i **`do`**-loopens villkor används i positiv (icke-negerad) form för att avsluta den. Men, kan man undra, kommer ”Grattis”-meddelandet inte i alla fall att skrivas ut även utan något **`if`**-huvud? Detta speciellt med tanke på att **`do`**-loopen endast avslutas när man gissat rätt och turen automatiskt kommer till ”Grattis”-meddelandet om man skriver det efter **`do`** utan **`if`**. Resonemanget vore korrekt om det i loopen inte fanns möjligheten till att avsluta med inmatning av **`0`** och därmed bryta loopen. I ett sådant fall ska nämligen **`if`**-villkoret förhindra att ”Grattis”-meddelandet skrivs ut efter att man avbrutit spelet och redan fått ”Avbrott”-meddelandet samtidigt som programmets hemliga tal avslöjats.

Är användningen av NEGATION överhuvud taget inte onödigt? Svaret hänger ihop med hela strukturen i programmet **`GuessNEG`**: En enda logisk variabel som initieras till ett logiskt uttryck ska styra både **`do`**-loopen som tillåter flera spelomgånger och **`if`**-satsen som skriver ut ”Grattis”-meddelandet. Men eftersom **`do`** ska fortsätta när man gissat *fel*, medan **`if`** ska utföras när man gissat *rätt*, alltså tvärt om, kan man åstadkomma en klar logisk struktur om man använder en och samma logisk variabel i båda och negerar den antingen i **`do`**- eller i **`if`**-villkoret – en teknik som kan användas i andra problem som har samma logiska struktur. Vi kommer att göra det i en programserie i slutet av detta kapitel där NEGATIONen i kombination med de andra logiska operatorerna OCH och ELLER tillämpas på verifiering av lösenord.

2.4 Programserien Testa lösenord

Här inleds programserien *Testa lösenord* med ett exempel som tillämpar våra kunskaper om loopar och logik på verifiering av lösenord. Vi börjar först med ett enkelt test av endast ett lösenord (programmet **Passwd**) och kommer sedan att utvecklas till att mata in lösenord även om **Caps Lock**-tangenten är påslagen (programmet **PasswdCaps**). När man tillåter påslagen **Caps Lock**-tangent gäller det att verifiera två lösenord. Men först det enkla testet:

```
// Passwd.cs
// Enkelt test av endast ett lösenord
using System;

class Passwd
{
   static void Main()
   {
      String input;
      bool wrongPasswd;

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

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

En körning av programmet **Passwd** ger följande dialog om man vid andra försöket matar in korrekt lösenord och beaktar att man inte har **Caps Lock** på, annars kan det bli ännu fler inloggningsförsök.

```
        Skriv ditt lösenord:    HEMLIGT

        Fel lösenord. Försök igen!

        Skriv ditt lösenord:    hemligt

        OK, nu är du inloggad!
```

Jämförelsen mellan strängar är nämligen alltid case sensitive eftersom den görs tecken för tecken varvid tecknens ASCII-koder jämförs med varandra. Och versaler

har ju andra ASCII-koder än gemener. Därför kommer inte heller inmatningen av **Hemligt** leda till lyckad inloggning.

Logiken i **Passwd** består av den logiska variabeln **wrongPasswd** som initieras till det logiska uttrycket **!input.Equals("hemligt")** och styr både **do**-loopen och **if**-satsen som ingår i den. **do**-loopen ser till att dialogen mellan program och användare fortsätter så länge **wrongPasswd** är **true** dvs så länge man matar in felaktigt lösenord, någon sträng som är skild från **hemligt**. Strängen läses in och lagras i **String**-variabeln **input**. Jämförelsen mellan den inlästa strängen och lösenordet **hemligt** görs endast en gång i det logiska uttryck vars sanningsvärde tilldelas **wrongPasswd** som används både i **do**-loopens och **if**-satsens villkor. **do**-loopen avslutas om **wrongPasswd** blir **false** dvs om strängen **hemligt** matas in. Då skriver **if**-satsen inte ut något pga **wrongPasswd**:s **false**-värde, utan användaren får ok-meddelandet som står efter **do**-loopen innan programmet avslutas. Man ser fördelen med att använda en och samma logiska variabel med en och samma initiering både i **do**- och **if**-satsen – en teknik som vi redan använt i programmet **GuessNEG** (sid 71). Den logiska strukturen i båda programmen är den samma: En dialog förs vars avslutning beror på en viss (korrekt) inmatning. Ett meddelande om fortsatt dialog skrivs ut i fall av felaktig inmatning. Detta meddelande måste placeras *inuti* loopen. I fall av korrekt inmatning skrivs ut ett annat meddelande som måste placeras *efter* loopen.

Metoden `Equals()`

I programmet **Passwd** anropas metoden **Equals()** så här:

```
input.Equals("hemligt")
```

för att testa om den inmatade strängen **input** är identisk med strängen **"hemligt"**. Redan *hur* **Equals()** anropas visar att metoden är definierad i klassen **String** därför att före punkten står variabeln **input** som är av typ **String**. Datatypen **string** – med lilla **s** – som vi använt hittills för strängvariabler, är endast ett alias för klassen **String** – med stora **S**. Metoden **Equals()** testar två strängar på likhet och returnerar **true** om de är lika, annars **false**. Anropet ovan returnerar **true** om variabeln **input** refererar till en sträng som är identisk med strängkonstanten **"hemligt"**, annars **false**. Därför kan anropets returvärde först negeras med **!** och sedan tilldelas **bool**-variabeln **wrongPasswd**.

Man kan ju undra varför strängarna inte jämförs med den vanliga likhetsoperatorn ==, så här: **input == "hemligt"** vilket är enklare än att anropa metoden **Equals()**. I C# går det bra att även skriva så. I själva verket har jämförelseoperatorn == i C# när den tillämpas på datatypen **String**, betydelsen ”anrop av metoden **Equals()**” som jämför strängarnas *innehåll* och inte deras referenser. Med andra ord är operatorn == ett alias för metoden **Equals()**. Både variabeln **input** och konstanten **"hemligt"** är strängt taget referenser dvs adresser till objekt av klassen **String**. Men både == och **Equals()** jämför *objekten*. Därför spelar det i C# ingen roll – till skillnad från Java – om vi jämför strängar med ==

eller **`Equals()`**. Vi kommer dock i fortsättningen att föredra metodnotationen **`Equals()`**.

Kombination av NEGATION, OCH, ELLER

När man loggar in på sitt konto på datorn, måste man se upp att **`Caps Lock`** inte är aktiverad, annars blir lösenordet felaktigt och man kan inte komma in. Det beror på att i de flesta operativsystem lösenord (till skillnad från användarnamn) är case sensitive. Vill man från systemsidan slippa **`Caps Lock`**-problematiken och underlätta inloggningen genom att tillåta även lösenord i versaler, måste ett program ingå i operativsystemet som testar lösenordet både med små och stora bokstäver. Ur säkerhetssynpunkt behöver detta inte vara något problem. När en användare känner till sitt lösenord spelar det väl ingen roll om han/hon matar in det med gemener eller versaler. En liknande frågeställning kan förekomma i andra tillämpningar, där både *ja* och *Ja* eller *nej* och *Nej* skall tillåtas som svar på en fråga om fortsatt dialog med programmet. Följande program löser **`Caps Lock`**-problematiken på två olika, men logiskt likvärdiga sätt:

```
// PasswdCaps.cs
// Användaren skall kunna mata in lösenord i versal eller
// gemener. Lösning med negerade delvillkor kombinerade med
// OCH. Alternativt: Negation på det hela sammansatta ELLER-
// villkoret
using System;

class PasswdCaps
{
   static void Main()
   {
      String input;                        // Lokala variabler
      bool wrongPasswd;                    // i Main()

      do
      {
         Console.Write("\n\tSkriv ditt lösenord:\t");
         input = Console.ReadLine();
         wrongPasswd = !input.Equals("hemligt") &&
                       !input.Equals("HEMLIGT");
//       wrongPasswd = !(input.Equals("hemligt") || // Alter-
//                       input.Equals("HEMLIGT"));  // nativt
         if (wrongPasswd)
            Console.WriteLine("\n\tFel lösenord. " +
                              "Försök igen!");
      } while (wrongPasswd);

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

En körning med inmatningen **`HEMLIGT`** i versaler ger lyckad inloggning:

```
Skriv ditt lösenord:    HEMLIGT

OK, nu är du inloggad!
```

Samma resultat skulle förstås ge en körning med inmatningen **hemligt** i gemener. Alla andra inmatningar kommer att misslyckas. Detta beror på **do**-loopens logik, närmare bestämt på dess avslutningsvillkor **wrongPasswd**: Så länge det är sant ska loopen fortsätta. Den logiska variabeln **wrongPasswd** i sin tur har värdet **true** om det logiska uttryck som den är tilldelad till, nämligen:

```
!input.Equals("hemligt") && !input.Equals("HEMLIGT")
```

har värdet **true**. Detta sammansatta uttryck är i sin tur sant endast om *båda* deluttrycken är sanna dvs om **input** är varken lika med **hemligt** eller **HEMLIGT**.

Är däremot den inmatade strängen **input** lika med **hemligt** eller **HEMLIGT**, ska loopen stoppas. Då kommer efter **do**-satsen ok-meddelandet att skrivas ut och programmet avslutas. Viktigt för att få det hela att fungera korrekt är också att **if**-satsen i loopen som skriver ut meddelandet om misslyckat inloggningsförsök har samma logiska variabeln **wrongPasswd** med samma värde som villkor.

I uttrycket ovan har vi: **!p && !q** där **p = input.Equals("hemligt")** och **q = input.Equals("HEMLIGT")**

Men det finns i logiken en lag som säger att uttrycket ovan dvs det sammansatta OCH-uttrycket bildat av de negerade delutsagorna, är ekvivalent (logiskt likvärdigt) med det negerade sammansatta ELLER-uttrycket:

```
!p && !q = !(p || q)
```

Ett vardagligt exempel på denna lag är: *”Jag dricker kaffe utan socker OCH utan mjölk.”* betyder samma sak som *”Jag dricker kaffe varken med socker ELLER med mjölk.”*. Lagen kallas efter den brittiske matematikern *De Morgan*. Formuleringen ovan är *De Morgans första lag*. Enligt denna lag kan vi alternativt till uttrycket i programmet **PasswdCaps** även tilldela följande uttryck till den logiska variabeln **wrongPasswd**:

```
!(input.Equals("hemligt") || input.Equals("HEMLIGT"))
```

Detta är i den aktuella versionen av programmet **PasswdCaps** borkommenterat. Alternativet innebär: Om det *inte* är sant att den inmatade strängen **input** är lika med **hemligt** eller **HEMLIGT**, ska **do**-loopen fortsätta dvs inloggningsförsöket är misslyckat och användaren måste göra om försöket. Är däremot **input** lika med **hemligt** eller **HEMLIGT**, ska dialogen stoppas. Då kommer ok-meddelandet att skrivas ut och programmet avslutas. Observera att negationsoperatorn i detta alternativ måste hållas *utanför* det sammansatta ELLER-villkoret. Vart negationen ska sättas, är intuitivt inte självklart, utan framgår av *De Morgans lag.*

De Morgans lagar

De Morgans lagar skrivs i mängdnotation så här:

$$\mathbf{C}\ (A \cap B) = (\mathbf{C}\ A) \cup (\mathbf{C}\ B)$$

$$\mathbf{C}\ (A \cup B) = (\mathbf{C}\ A) \cap (\mathbf{C}\ B)$$

Där A och B är mängder, **C** är symbolen för komplementmängden, ∩ för snittet och ∪ för unionen av två mängder. Man kan *illustrera* dem med Venndiagram. När vi däremot vill *bevisa* dem, formulerar vi dem i logisk notation, vilket kan ske på olika sätt beroende på vilka symboler man använder. Två sätt presenteras här.

1. En formellt logisk formulering av dessa lagar ser ut så här:

$$\neg\,(p \text{ OCH } q) \quad \leftrightarrow \quad \neg\, p \text{ ELLER } \neg\, q$$

$$\neg\,(p \text{ ELLER } q) \quad \leftrightarrow \quad \neg\, p \text{ OCH } \neg\, q$$

där p och q är utsagor, ¬ är symbolen för negation och ↔ för logisk ekvivalens.

2. Inför man programmeringskoderna **&&** för OCH, **||** för ELLER och **!** för negation (*logiska operatorer*) och **=** för ekvivalens, kan vi skriva De Morgans lagar så här:

```
!p && !q = !(p || q)
!p || !q = !(p && q)
```

Beviset

Formellt är två logiska uttryck ”lika” med varandra om deras *sanningstabeller* är identiska. Man säger då att de är *ekvivalenta*, dvs logiskt likvärdiga. I själva verket handlar det om de logiska sanningsvärdenas likhet. I praktiken betyder det att uttrycken i både VL och HL har samma sanningstabell. Vad är en *sanningstabell* ?

Operatorerna OCH och ELLER har följande sanningstabeller:

```
  p       q             p && q          p || q
-----------------------------------------------
True    True              True            True
True    False             False           True
False   True              False           True
False   False             False           False
```

Den logiska negationen **!** har följande sanningstabell:

```
  p        ! p
----------------
true       false
false      true
```

Den första av De Morgans lagar kan man bevisa genom att gå igenom de enskilda operatorerna **&&**, **||** och **!**:s respektive sanningstabeller, som visas ovan, och sätta ihop sedan sanningsvärdena:

```
  p       q           !p && !q       !(p || q)
-----------------------------------------------
true    true            false           false
true    false           false           false
false   true            false           false
false   false           true            true
```

Man ser att båda uttryckens sanningstabeller är identiska.

Mönstret som blir tydligt är följande: Appliceras negationen på det sammansatta uttrycket och sätts framför parentesen istället för att appliceras på varje enskild operand, måste **&&** bytas ut mot **||**.

Analogt till samma mönster gäller den andra av De Morgans lagar:

```
!p || !q = !(p && q)
```

Även denna logiska ekvivalens (**=**) kan visas på samma sätt som den första, nämligen genom att applicera de enskilda operatorerna **&&**, **||** och **!**:s respektive sanningstabeller och sätta ihop sedan sanningsvärdena:

```
  p       q           !p || !q       !(p && q)
-----------------------------------------------
true    true            false           false
true    false           true            true
false   true            true            true
false   false           true            true
```

Man ser att uttrycken i både VL och HL har samma sanningstabell.

Därmed har vi bevisat De Morgans lagar.

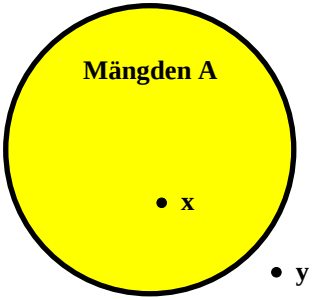
2.5 Mängdlära och logik

Vad har *mängder* med *logik* att göra? Och varför blandar vi in ett nytt begrepp i diskussionen om logik? Det enkla svaret är just nu: allt vi t.ex. sagt i förra avsnitt om De Morgans lagar kan man lika bra – kanske t.o.m. bättre – formulera, förklara och förstå med mängder. Så kan man göra även med andra logiska lagar. Logiken är abstrakt, men mängder kan man *föreställa sig* därför att de består av konkreta saker och ting. Vi kan med hjälp av mängder *visualisera* logiken, vilket inte bara ökar förståelsen utan också skapar – vi kommer att se det – en vacker analogi som har ett värde i sig. Men även rent praktiskt kommer vi att ha nytta av mängdlärans begrepp senare när vi behandlar *databaser* (sid 158). Att vi tar upp temat just nu beror på kopplingen med logiken som behandlats i detta avsnitt.

Mängdoperationer och deras logik

En väldefinierad samling av saker och ting (föremål, objekt) kallas för *mängd*. En mängd kallas *väldefinierad*, om man alltid kan avgöra om något element tillhör mängden eller ej. Vi betraktar endast väldefinierade mängder och utesluter icke-väldefinierade mängder. För, om vi inte gör det hamnar vi förr eller senare i svårigheter av den typ som man gjorde i början av 1900-talet. Förenklat kan man illustrera dessa svårigheter med *Russells paradox*[*] (antinomi, motsägelse). Så i fortsättningen förutsätter vi att alla mängder vi pratar om, är väldefinierade, vilket betyder att man för *alla* element i en mängd kan avgöra om elementet tillhör mängden eller ej. Vi inför ett antal symboler för mängder och mängdoperationer:

Element av en mängd



Låt mängden A bestå av ett antal element.

Att elementet x tillhör mängden A

uttrycks med: $x \in A$

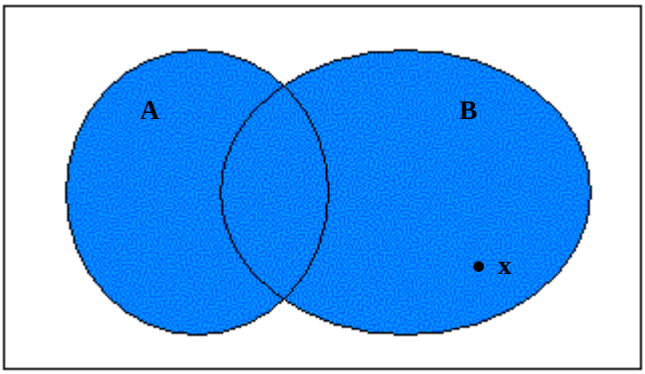
y tillhör inte A: $y \notin A$

Det lilla tecknet ε kallas *epsilon* i det grekiska alfabetet och står för element.

[*] *Russells paradox:* Att det även finns icke-väldefinierade mängder har Bertrand Russell visat med sin berömda antinomi om barberaren i en by (1903): En liten by har endast en barberare. Byborna delas i två mängder: 1. Alla som inte rakar sig själva och därför rakas av barberaren. 2. Alla som rakar sig själva och inte rakas av barberaren. Frågan är: *Vem rakar barberaren?* Denna fråga leder till en oupplösbar motsägelse: Om han rakar sig själv, tillhör han mängden 2, men då får han inte raka sig själv. Om han inte rakar sig själv, tillhör han mängden 1, men då måste han raka sig själv. Det kan inte avgöras vilken mängd han tillhör. Därför skapar indelningen av byborna i två mängder enligt ovan inga väldefinierade mängder. Den logiska motsägelsen löstes senare av *Russell*, *Wittgenstein* och andra filosofer.

Unionen av två mängder

Man slår ihop (sammanfogar, förenar) två mängder:

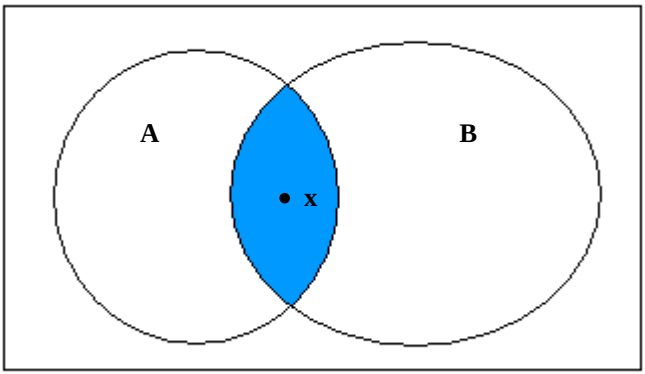


Resultatet är **unionen** av mängderna A och B och betecknas med: **A U B**

x ε A U B om x ε A ELLER x ε B

Unionen motsvarar den logiska operatorn ELLER.

Snittet av två mängder



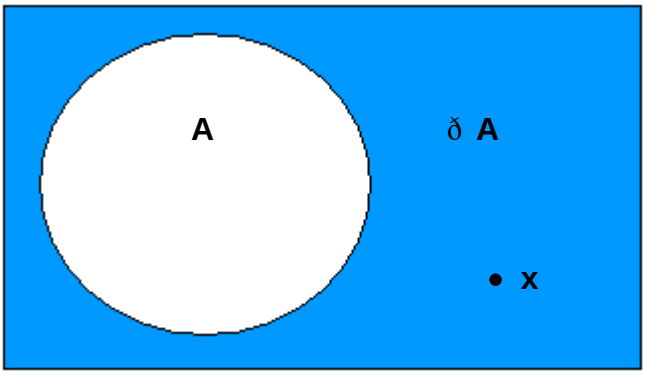
Resultatet är **snittet** (skärningsmängden, det gemensamma) av mängderna A och B och betecknas med:

A ∩ B

x ε A ∩ B om x ε A OCH x ε B

Snittet motsvarar den logiska operatorn OCH.

Komplementet av en mängd

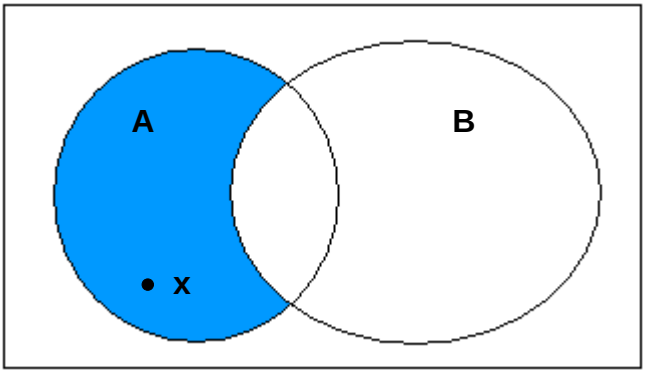


Resultatet är **komplementmängden** av mängden A och betecknas med: **ð A**

$$x \in \eth A \quad \text{om} \quad x \notin A$$

Komplementet motsvarar den logiska operatorn NEGATION.

Differensen av två mängder



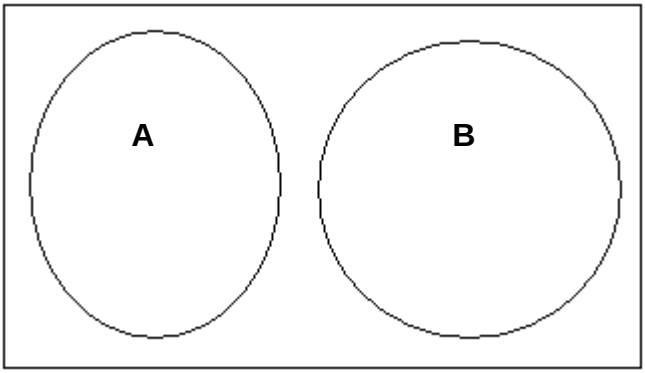
Resultatet är **mängddifferensen** av mängderna A och B och betecknas med:

$$A \setminus B$$

$$x \in A \setminus B \quad \text{om} \quad x \in A \text{ OCH } x \notin B$$

Den tomma mängden

En mängd som inte har något element betecknas med den **tomma mängden** och har symbolen **Ø** . Ex.:



Två mängder utan något gemensamt element kallas **disjunkta**.

Snittet av disjunkta mängder är den tomma mängden: **A ∩ B = Ø**

Cartesisk produkt

Den cartesiska produkten av två mängder A och B:

A X B (A "kryss" B)

är mängden av samtliga ordnade par (x, y) där x tillhör A och y tillhör B.

Exempel: *Person* = { Ola, Eva, Jimmy, Alexander, Helen, David, Diana }

Lägenhet = { 1, 2, 3 }

Den cartesiska produkten av dessa två mängder består av mängden:

Person X *Lägenhet* = {

(Ola, 1),	(Ola, 2),	(Ola, 3),
(Eva, 1),	(Eva, 2),	(Eva, 3),
(Jimmy, 1),	(Jimmy, 2),	(Jimmy, 3),
(Alexander, 1),	(Alexander, 2),	(Alexander, 3),
(Helen, 1),	(Helen, 2),	(Helen, 3),
(David, 1),	(David, 2),	(David, 3),
(Diana, 1),	(Diana, 2),	(Diana, 3) }

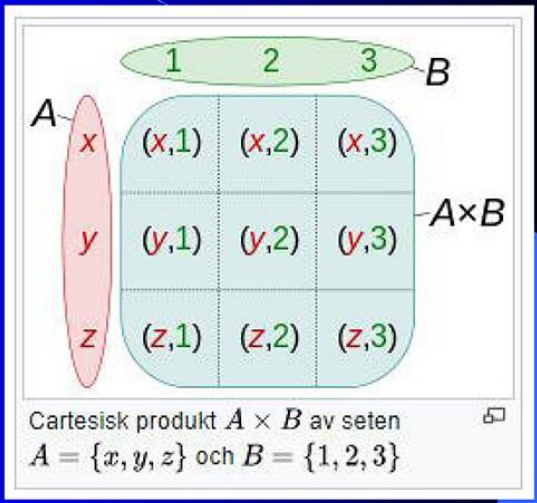
Cartesisk produkt = Kombination av alla med alla = Alla möjliga par.

Varför "Cartesisk" produkt?



René Descartes

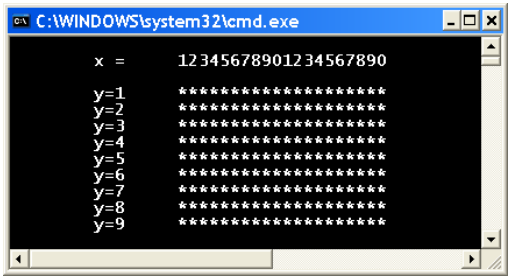
2D Cartesiskt koordinatsystem med B som x- och A som y-axeln.



Cartesisk produkt $A \times B$ av seten $A = \{x, y, z\}$ och $B = \{1, 2, 3\}$

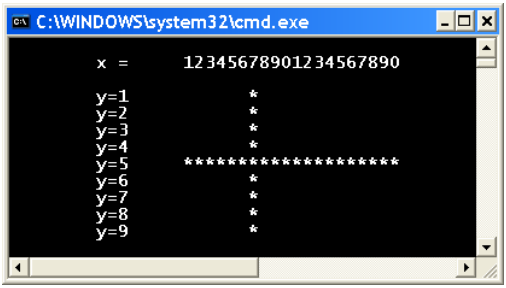
Övningar till kap 2

2.1 Skriv ett program som med hjälp av en nästlad **`for`**-sats skriver ut en rektangel fylld med stjärnor (**`*`**) till konsolen, bestående av 9 rader och 20 kolumner.

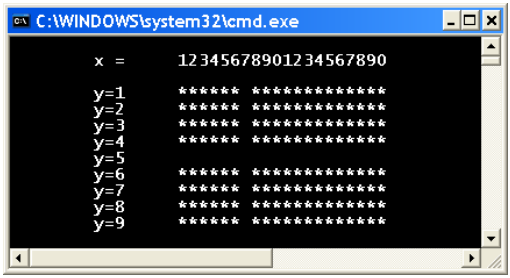


Numrera raderna och kolumnerna utan att förstöra helhetsbilden. Denna uppgift är knappast någon övning i logik, utan snarare i nästlade loopar. Men den förbereder de följande två övningar i sammansatta villkor och logiska operatorer.

2.2 Selektera (skriv ut) från den stjärnfyllda rektangeln från övn 2.1 endast den 5:e raden och den 7:e kolumnen så att det visas ett kors. Lägg in i den inre **`for`**-slingan som skriver ut en rad, en **`if-else`**-sats som i varje varv skriver ut en stjärna om ett sammansatt villkor med ELLER är uppfyllt, annars ett mellanslag. Hur blir det om du byter ut ELLER mot OCH?



2.3 Omvandla korset från övn 2.2 till dess *negativ*, dvs skriv ut alla stjärnor från övn 2.1 *utom* den 5:e raden och den 7:e kolumnen. Använd den logiska operatorn NEGATION. Negera en gång hela det sammansatta ELLER-villkoret från övn 2.2 och en gång det sammansatta villkorets delvillkor. I båda fall borde du få samma resultat.



2.4 Skriv ett program som läser in tre tal, hittar och skriver ut det största av dem. Lös problemet genom att använda tre *enkla* **`if`**-satser med sammansatta villkor och den logiska operatorn **`&&`**. På så sätt kan du i varje **`if`**-sats jämföra ett tal med de två andra. Varför måste variabeln som lagrar det största talet, initieras vid deklarationen?

2.5 Skriv ett program som skriver ut sanningsvärdet till det enkla villkoret **`a < 10`** där **`a`** är en heltalsvariabel vars värde läses in. Testa ditt program genom att mata in t.ex. 9, 10 resp. 11.

2.6 Bestäm sanningsvärden hos de följande logiska uttrycken, först med papper och penna, sedan i ett C#-program:

a) **`(8 < 7) && (true || false)`**
b) **`!(3 < 3.01) || (!(0==0) && true)`**
c) **`(true || !false) && !(!(4*5==1) && false)`**

2.7 Följande enkel version av Gissa tal-spelet tillåter endast *en* spelomgång (utan loop). För att koda ett trevägsval nästlar programmet en **`if-else`** -sats i en annan **`if-else`**-sats:

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

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

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

Modifiera programmet ovan genom att använda logiska operatorer och sammansatta villkor i syftet att förenkla nästlingen. Det nya programmet ska göra samma sak som **`GuessIfElse`**. Bedöm i slutet själv om det har blivit mer förståelig kod.

2.8 Modifiera programmet **`PasswdCaps`** (sid 75) genom att lägga in kod som begränsar antalet inloggningsförsök till t.ex. 3. Överskrider man denna gräns ska programmet avslutas efter att ha skrivit ut ett meddelande av typ *Du har försökt 3 gånger. Nu avslutas programmet!*

Tips: Använd en **`if`**-sats som avslutar programmet genom att bryta loopen med **`break`**.

2.9 Operationer med mängder kan illustreras grafiskt. Hur man gör det kan du läsa i avsnitt *2.5 Mängdlära och logik* på sid 79. Diagrammen du ser där kallas för *Venndiagram* efter den brittiske logikern John Venn (1834-1923).

Med Venndiagram kan man illustrera även logiska lagar när de är skrivna i mängdnotation, där en *mängd* motsvarar en *utsaga*.

De Morgans lagar som togs upp i kap 2 (sid 77) kan då formuleras så här:

$$\neg (p \text{ OCH } q) \quad \leftrightarrow \quad \neg p \text{ ELLER } \neg q$$

$$\neg (p \text{ ELLER } q) \quad \leftrightarrow \quad \neg p \text{ OCH } \neg q$$

där p och q är utsagor, ¬ är symbolen för logisk negation och ↔ symbolen för logisk ekvivalens. Så här kan man skriva om dem till samband mellan mängder:

Anta att A och B är mängder och **C** är symbolen för komplementmängden, ∩ för snittet och ∪ för unionen av två mängder, se definitionerna i avsnitt *2.5 Mängdlära och logik* på sid 79. Då kan De Morgans lagar skrivas i mängdnotation så här:

$$\mathbf{C}\ (A \cap B) = (\mathbf{C}\ A) \cup (\mathbf{C}\ B)$$

$$\mathbf{C}\ (A \cup B) = (\mathbf{C}\ A) \cap (\mathbf{C}\ B)$$

Illustrera De Morgans lagar i mängdnotation med Venndiagram.

Kapitel 3

Datastrukturer

och abstrakta datatyper

	Ämne	Sida	Program
3.1	Vad är objektorienterad programmering?	88	
3.2	Objektorienterad design med UML	94	
	- Projekt Lönespecifikation	94	
	- Kundens kravspecifikation	94	
	- UML design och modellering i fyra steg	94	
3.3	Array som objekt	98	**ArrayObj**
	- **foreach**-satsen	102	
3.4	Hantering av array med referens	105	**ArrayRef**
3.5	Array av referenser	107	**ArrayOfRef**
3.6	Array som parameter i metoder	111	**ArrayParam**
3.7	Hantering av slumptal i C#	115	**DoRand**
	- Array av slumptal	116	**RandArray**
3.8	Sökning och sortering	118	**Search**
	- Bubbelsortering	121	**Bubble**
3.9	Generiska metoder	124	**G_Bubble**
3.10	Listor	129	**Lista**
	- Klassen **RandList**	130	**RandList**
	- **foreach** i listor	131	**Print**
Övningar till kapitel 3		133	

3.1 Vad är objektorienterad programmering?

En given definition på programmering är problemlösning med hjälp av datorn. Om man då beskriver problemets lösning i form av en *algoritm* kan man säga *Program = algoritm + data*. Denna definition ställdes upp av Niklaus Wirth på 60-talet och återspeglar den procedurala synen på programmering. Fokuset ligger på *algoritmen* dvs att inte bara hitta utan även *beskriva* tillvägagångssättet (proceduren) för att lösa ett problem. Sedan återstår bara att koda denna beskrivning. En annan definition som kom upp på 80-talet och återspeglar den objektorienterade synen på programmering är:

Program = Modell av verkligheten

Om man i formeln *Program = algoritm + data* lägger betoningen på data istället för på algoritmen och inte längre betraktar data som ett slags bihang till algoritmen utan som *objekt* kommer man till *objektorienterad programmering*. Denna nya programmeringsfilosofi genomsyr alla våra program, eftersom C# med alla sina fördefinierade biblioteksprogram är i högsta grad objektorienterade.

Paradigmskifte

Det som i programmeringshistorien gjorde att man behövde objektorienterad programmering var den växande komplexiteten hos program under 70-talet. Programmens storlek var avgörande för den växande komplexiteten. Man insåg att det inte längre räckte till att skriva och testa program som fungerade just då. Det var nödvändigt att med rimliga kostnader kunna även *underhålla* stora program, *förnya* och *vidareutveckla* dem så att de fungerade även i flera år och att de framför allt kunde anpassas till nyuppkomna situationer utan oöverkomliga svårigheter. Det i sin tur krävde att man redan i designstadiet behövde ett annorlunda upplägg. Fokuset förskjöts från problemlösning till modellering av verkligheten. Objektorienterad design kom in i bilden. Allt detta var endast med procedural programmering inte längre möjligt. Ett s.k. *paradigmskifte* hade blivit nödvändigt, dvs en ändring av helhetssynen på programmering.

Objektorienterad programmering syftar åt att efterlikna verkligheten. Man vill avbilda den reala världen – åtminstone den del som tillåter datorisering – och konstruera en modell av den i sina datorprogram för att kunna simulera verkligheten genom att testa modellen. För att undvika filosofiska diskussioner kan vi anta att den reala världen består kort sagt av *objekt*. Världen kring oss är full med sådana objekt: Människor, byggnader, bilar, tåg, flygplan, träd, möbler, böcker, butiker, skolor, bibliotek, kontor, anställda, kunder, varor, fakturor, order, bokningar, kurser osv. Objekten kan vara verkliga eller virtuella. Ett datorprogram försöker att beskriva dessa objekt. Låt oss precisera detta:

Objekt, klass och metod

Ett *objekt* har vissa egenskaper. Generellt kan man säga att ett objekt är summan av alla sina egenskaper. Ett annat ord för egenskap är *attribut*. Ett objekt består av alla sina attribut. Attributen tillhör objektet. T.ex. har objektet bil som attribut fabrikat, modell, färg, årsmodell, antal körda mil, antal hästkrafter, maximala hastigheten, antal och storlek på cylindrar i motorn osv. Alla dessa data ger svar på frågan ”Vad är det för bil?”. Men bilden vore ofullständig om vi nöjde oss med dessa intressanta, men statiska data. Vi vill också veta vad man kan *göra* med bilen. Ett objekt kan i regel även utföra vissa aktioner eller operationer. I den objektorienterade programmeringens terminologi kallas de för *metoder*. Typiska metoder för en bil är t.ex. att köra fram, att backa, att accelerera, att bromsa, att parkera, att byta olja osv. Den fullständiga definitionen på en bil som objekt vore alltså att ange *både* dess attribut *och* metoder. Bilfabrikanten måste förse bilen med alla dessa färdigheter för att kunna sälja den. Därför går man i bilfabriken efter en plan när man tillverkar bilen. I den objektorienterade programmeringens terminologi kallas denna plan för bilens *klass*. När vi skriver ett program måste vi först formulera *klassen Bil* för att sedan kunna skapa objekt av den. Klassen skrivs bara en gång, medan objekt kan skapas enligt klassens beskrivning i obegränsat antal. I klassen måste vi ta upp alla attribut och metoder som är relevanta eller av någon anledning önskvärda för en bil. Den praktiska användningen avgör från fall till fall vad som är relevant eller önskvärt.

Vad är skillnaden mellan *objekt* och *klass*? Om vi byter ut bilar mot pepparkakor kan man säga att pepparkaks*formen* är klassen och själva pepparkakorna är objekten. Klassen är alltså en slags mall, en förskrift för produktion av objekt: En enda pepparkaksform kan producera tusentals pepparkaksgubbar. Gubbarna kan skiljas från varandra i vissa detaljer, t.ex. materialet, smaken osv. Man kan t.o.m. måla dem i olika färger eller modifiera på annat sätt efteråt. De förblir pepparkaksgubbar av den ursprungliga formen. I formen ingår det som är gemensamt hos alla pepparkaksgubbar. Man har, när man byggde formen, bortsett från oväsentliga skillnader och tagit hänsyn endast till det väsentliga, det gemensamma hos alla pepparkakor.

Att *bortse* från skillnader och att bibehålla det gemensamma hos olika verkliga objekt, är en *abstraktion* (*abstrahera* betyder på latin: att ta bort, att dra av). Man tar bort allt som skiljer saker och ting av samma kategori eller typ och kommer på det viset till själva kategorin. Abstraktion leder till *begrepp*sbildning, till *klass*ificering eller *kategori*sering av den reala världen. Ett växande barn går igenom samma abstraktionsprocess, ser först sina föräldrar (objekt), abstraherar sedan via erfarenhet så småningom till begreppet *människa* (klassen) och inser att sina föräldrar är två konkreta exemplar av den abstrakta klassen människa. Så gör barnet med alla saker och ting omkring sig och lär sig vuxenvärldens begreppsapparat. Det abstrakta begreppet *penna* (klassen) t.ex. bildas efter att man sett hundratals verkliga pennor (objekt). Objektorienterad programmering återspeglar denna naturliga tankeprocess från det konkreta till det abstrakta, från objekt till klass.

Metoder

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

```
minVolvo.Kör();
```

Observera att *före* punkten står ett objekt, inte klassen. Det är ju den specifika bil som jag använder just nu som ska köras. Först *efter* punkten står själva anropet av metoden **`kör()`**. Det här sättet att skriva kallas *punktnotation*. Metoder måste alltid anropas med punktnotation, vilket har sin grund i att de endast är deklarerade i klasser, så att de endast existerar i objekt av en klass. Till skillnad från fristående *funktioner* kan metoder varken definieras utanför klasser eller anropas utanför objekt. I C# finns endast metoder, inga funktioner. Om vi bortser från bilexemplet kan det i andra sammanhang även förekomma en klass (istället för objekt) före punkten i anropet av en metod. I så fall är metoden definierad i klassen på ett speciellt sätt nämligen som en *statisk* metod, vilket tas upp senare när vi behandlar metoder i detalj.

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

```
minVolvo.Kör(40);
```

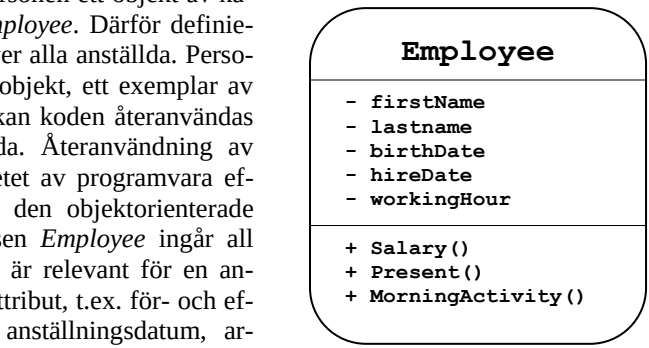
Det kan t.ex. betyda: Kör bilen med hastigheten **`40`** km/h. Värdet **`40`** kallas då en *parameter* som skickas till metoden när den anropas. I så fall måste även metoden **`Kör()`** vara definierad så att den har beredskapen att ta emot denna parameter. Så det kan inte vara samma metod som anropades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

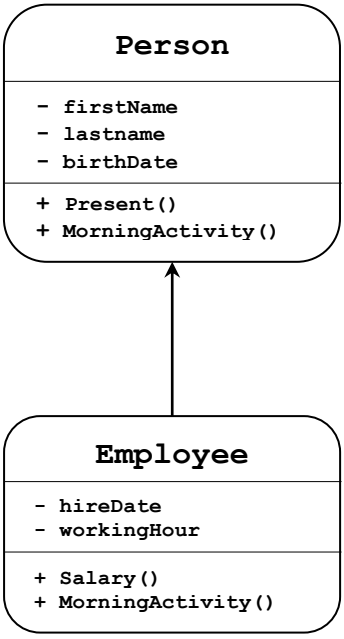
Klassdiagram

Låt oss ta som exempel en algoritm som beskriver hur man går upp, duschar, tar på sig kläderna och åker till jobbet (algoritmen *Morgonsyssla* i *Progr1+, 1.4*). Detta är ett typiskt fall av problemlösning: Det löser problemet *hur* man tar sig till jobbet. Tillvägagångssättet och framför allt hur vi beskriver det, är föremål för algoritmer. Men

vem eller *vilka* gör det, dvs vilka *objekt* som är involverade i algoritmen och hur man beskriver dessa objekt, är en annan aspekt på saken. Objektorienterad programmering prioriterar objektaspekten framför algoritmaspekten. Den primära frågan är inte längre vad man *gör* utan vem man *är* dvs *hur kan personen beskrivas?* Hur man gör för att ta sig till jobbet kommer att ingå som en del i denna beskrivning. Algoritmen Morgonsyssla blir en metod i *objektet* Person. Det är objektet som utför metodens instruktioner för att ta sig till jobbet.

Personen kan t.ex. vara en anställd vilket förresten skulle förklara varför han tar sig till jobbet. I så fall är personen ett objekt av kategorin eller klassen *Employee*. Därför definieras en klass som beskriver alla anställda. Personen i fråga görs till ett objekt, ett exemplar av denna klass. På så sätt kan koden återanvändas även för andra anställda. Återanvändning av kod gör utvecklingsarbetet av programvara effektivare och är en av den objektorienterade synens fördelar. I klassen *Employee* ingår all typ av information som är relevant för en anställd, det vi kallar för attribut, t.ex. för- och efternamn, födelse- och anställningsdatum, arbetstid osv. Dessutom tar vi upp allt som en anställd kan göra, det vi kallar för metoder, t.ex. att få lön, att presentera sig eller också att ta sig till jobbet. På så sätt blir algoritmen Morgonsyssla i den objektorienterade programmeringens terminologi en metod i klassen *Employee*. Ett verktyg speciellt för objektorienterade modelleringar är UML *(Unified Modeling Language)*. Enligt det här modelleringsspråket skulle klassen *Employee* beskrivas med diagrammet till höger som kallas för *klassdiagram*. Där står tecknet **–** för attribut och **+** för metoder. Andra beteckningar för attribut är *datamedlem* eller *egenskap*. Dessa termer är synonymer. En klass består av datamedlemmar och metoder. Klassen **Employee** t.ex. har fem datamedlemmar och tre metoder.





Klassens konstruktor

Eftersom klassens datamedlemmar i regel är inkapslade (privata) och inte åtkomliga utifrån klassen – detta gör man bl.a. ur datasäkerhetssynpunkt – måste programmeraren använda sig av ett verktyg för att på ett kodat sätt ändå kunna komma åt dem, läsa och ändra dem osv. Detta verktyg kallas klassens *konstruktor* och är en speciell metod vars namn är identiskt med klassens namn. Den initierar automatiskt klassens

privata datamedlemmar när ett objekt skapas. För enkelhetens skull har vi inte tagit upp den i klassdiagrammet ovan bland klassens metoder. Konstruktorn har ju endast programmeringsteknisk karaktär och behandlas i detalj 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.

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

Objektorienterad programmering bygger på tre hörnstenar:

- Inkapsling
- Arv
- Polymorfism

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

3.2 Objektorienterad design 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

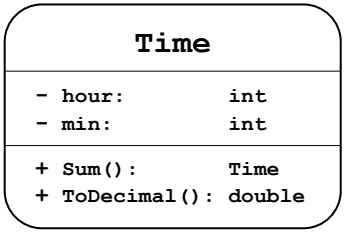
De kommer vid implementeringen att bli programmets ***klasser***. I databasmodellering används begreppet ***Entitet***. *Det är något viktigt för verksamheten – reellt eller virtuellt – som man kommer att behöva lagra information om.* Närmare bestämt handlar det om en *kategori av saker och ting som 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*.

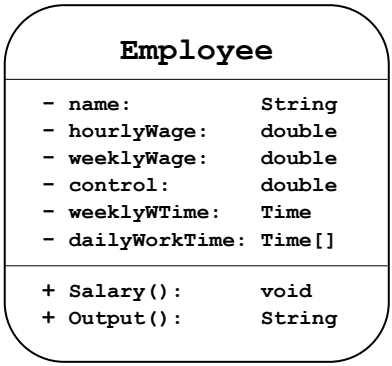
Steg 3: Identifiera datamedlemmar till varje klass

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 94). 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 inte här. Men i fall att man gjorde 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).

3.3 Array som objekt

Ordet *array* betyder i engelskan *ordnad skara* eller *ordnad uppställning* (*battle array* = stridsordning). Som datalogisk term hittar man i litteraturen begreppen *fält*, *vektor*, *matris*, *lista*, … . Ibland används även *härledd datatyp* som syftar åt att den är baserad på en *annan* datatyp. Vi kommer att använda den enkla termen *array*.

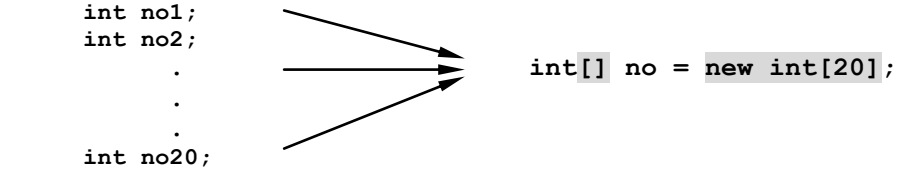
En **array** är en datastruktur, en ordnad mängd av variabler av *samma* datatyp grupperade under *samma* namn.

En array består av ett antal **element** vars *position* kallas för **index**.

Index är synonym till nummer och specificerar varje elements position i arrayen för att ”adressera” elementet. Elementen kan i sin tur vara av enkel, sammansatt eller av referenstyp. Så man kan även – med hjälp av referenser – gruppera objekt till en array. En array är den enklast tänkbara sammansatta datatypen. Som exempel tar vi en array som är sammansatt av den enkla datatypen **`int`**. Varje element i en sådan array kan betraktas som en indexerad dvs numrerad variabel av typ **`int`**.

Hittills behövde vi skriva 20 satser för att definiera 20 heltalsvariabler. Men nu ger array oss möjligheten att göra samma sak med endast *en* sats:

Hittills: enkel datatyp **`int`**: **Nu:** sammansatt datatyp ”*array av* **`int`**”:



Vi definierar *en* variabel **`no`** av datatypen **`int[]`**, använder **`new`** och lägger till informationen om antalet element inom hakparentes: **`[20]`**. Men vad är **`int[]`** för datatyp? Det reserverade ordet **`new`** avslöjar att det är ett *objekt*. **`new`** allokerar minnesutrymme för ett objekt bestående av 20 **`int`**-värden och returnerar den sammanhängande ”minneskedjans” adress – närmare bestämt adressen till dess första cell – till referensvariabeln **`no`**. Därmed har vi att göra med en *referenstyp*: Datatypen **`int[]`** är en *referens till en* **`int`**-*array* som i själva verket är ett *objekt*. För att göra det ännu tydligare kan man skriva den nya koden även i två separata satser:

```
int[] no;
no = new int[20];
```

Det är inte den första utan den andra satsen, närmare bestämt koden **`new int[20]`** som *skapar* själva arrayen. Därför står också storleken **`20`** där det behövs, nämligen i satsen där **`new`** allokerar minne. Typiskt för array är *hakparenteserna* **`[ ]`**, på engelska *brackets*. I satserna ovan har **`[ ]`** två olika betydelser: I den första satsen specificerar **`int[]`** variabeln **`no`**:s *datatyp* som en referens till en **`int`**-array, i den

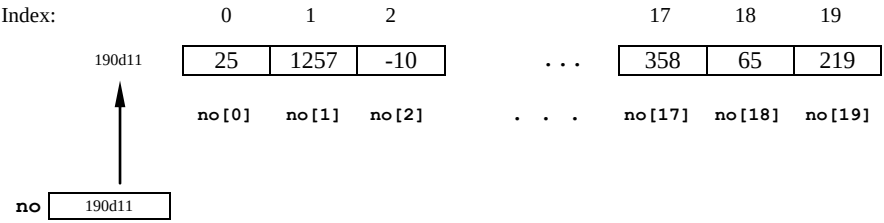
andra satsen innehåller **[20]** arrayens *storlek*. Referensvariabeln **no** ersätter de 20 vanliga **int**-variablerna **no1**, **no2**, …, **no20**, vilket medför en stor effektivitet i koden. Tänk dig att det är inte **20** utan fler data vi vill jobba med. **no** pekar fysiskt på det *första* elementet av arrayen som allokeras i ett sammanhängande minnesutrymme. Därför kan man komma åt de *andra* elementen via *indexering* som är bara ett annat namn för numrering.

Indexering i en array

Låt oss anknyta till exemplet ovan där arrayen och dess referens **no** definieras:

```
int[] no = new int[20];
```

Låt oss ytterligare anta att vissa *värden* – de som visas i bilden nedan – har tilldelats arrayens element efter satsen ovan. Eftersom elementen lagras i ett sammanhängande minnesområde uppstår följande minnesbild av arrayen i datorns RAM:



Medan själva arrayens allokering (den övre delen) görs av **new int[20]**, allokeras minnescellen **no** (den undre delen) av **int[] no**. Kopplingen mellan dem görs av tilldelningsoperatorn, vilket gör att arrayens adress (t.ex. 190d11 – ett hexadecimalt tal) som **new** har genererat, hamnar i minnescellen **no**. Den så uppkomna situationen innebär att **no** *pekar på* eller *refererar till* arrayen. Under arrayens minnesceller har vi skrivit C#-kod som kommer åt varje elements värde: **no[0]** ger den första minnescellens värde 25 som har index **0**, **no[1]** ger den andra minnescellens värde 1257 som har index **1** osv. **no[0]** lagras vid adressen till arrayens första minnescell. **no[1]** lagras vid adressen till den andra minnescellen som ligger **1** x 4 bytes – storleken för en **int** – längre bort från **no**. **no[2]** lagras vid adressen som ligger **2** x 4 bytes längre bort från **no** osv. Adressering i RAM sker nämligen bytevis, så att bytes som är grannar till varandra, har adresser som skiljer sig på *en* enhet. Avgörande för denna indexeringsteknik är att en array alltid allokeras i ett *sammanhängande* minnesområde. Ser man på det hela ur hårdvarans synpunkt kan man förstå varför indexnumreringen börjar med **0** och inte med **1**: **no[0]** kan tolkas som den *adress* som ligger **0** x 4 bytes längre bort från **no**, dvs **no[0]**:s adress är identisk med adressen **no**. Därför gäller:

> Indexregeln: I en array börjar numreringen av index alltid med 0.
> Därför gäller: elementets position = index + 1

Med *position* menas numret som *människan* använder för att numrera elementen. Människor är vana vid att påbörja numreringen av saker och ting med 1. Med *index*

menas numret som *datorn* använder för samma sak. C# och de flesta andra programmeringsspråken börjar numreringen av index i en array med **0**. Tillämpad på exemplet: Det 1:a elementet i den array som **no** refererar till har *värdet* 25 och *index* 0: Positionen är 1 medan indexet är **0**. Det 2:a elementet (värdet 1257) har index **1** och koden **no[1]**, det 3:e elementet (värdet –10) har index **2** och koden **no[2]** osv. Det n:e elementet har alltid index n-1. Därför har också det 20:e elementet (värdet 219) index **19**.

Det är avgörande när man arbetar med array och är samtidigt felkälla nr 1 – om man glömmer det – att hålla isär det mänskliga sättet att numrera som börjar med 1 från C#-kodens sätt som börjar med 0. I exemplet ovan har vi definierat en array av 20 heltalselement med referenserna **no[0], ..., no[19]**. Antalet element är 20. Indexen däremot går från **0** till **19**. Felkälla nr 2 är att förväxla en arrayelements *index* med dess *värde*: Det sista elementet i exemplet ovan har index **19**, men värdet 219. Man har alltid med *två* tal att göra, index (position) och värde (innehåll). Det gäller att hålla isär positionen från innehållet.

Tre egenskaper skiljer objekt från array:

- Indexering
- Allokering i ett sammanhängande minnesområde
- Alla arrayelement har samma datatyp.

Annars behandlas array i C# som objekt: Båda måste skapas med **new** och man kan komma åt båda endast med referensvariabler. Båda initieras till defaultvärden även om de kan förekomma som lokala variabler i metoder.

Definition och initiering av en array

Här testas allt vi sagt hittills om array speciellt indexregeln. Utöver det visas ytterligare en egenskap hos array som relaterar den till objekt, nämligen en datamedlem **Length** som lagrar arrayens storlek när den skapas. Programmet demonstrerar vad som händer om man överskrider arrayens maximala index: Man kan kompilera, men exekveringen stoppas vid överskridningen av indexgränsen, ett tecken på att arrayens minnesallokering sker vid *run time*, dvs programmet körs.

```
// ArrayObj.cs
// Definierar en array som objekt, visar default-initierings-
// värdena 0, tilldelar och skriver ut de nya värden
// Skriver ut arrayens storlek med datamedlemmmen Length
// Överskridning av arrayens index leder till exekveringsfel
using System;
class ArrayObj
{
   static void Main()
   {
     int[] no;                  // Deklarerar en referens no
                                // till en array av int
                                // typ array vars adress
```

```
     no = new int[4];          // new skapar ett objekt av
                               // tilldelas referensen no
//   int[] no = new int[4];    // Alternativt i EN sats

     Console.WriteLine("\n\tArray-storleken:\t\t"+no.Length);
     Console.Write("\n\tArrayens default-initiering:\t");
     foreach (int element in no)
        Console.Write(element + "\t");
     no[0] = 64;               // Tilldelar 1:a elementet
     no[1] = 86;               // värdet 64 osv. Överskriver
     no[2] = 34;               // default-initieringen
     no[3] = -6;
     Console.Write("\n\n\tArrayen efter tilldelning:\t");
     foreach (int element in no)
        Console.Write(element + "\t");
     Console.WriteLine(
       "\n\n\tÖverskridning av arrayens index leder till " +
       "programavbrott:\n\n\t\tno[4] inte definierad\n\t"  +
       "\tIndex 4 överskrider gränsen: Exekveringsfel!")   ;
     no[4] = 1;                // no[4] kan kompileras, men
  }                            // leder till exekveringsfel
}
```

Inte alla satser i programmet **ArrayObj** exekveras. Det blir avbrott när den kompilerade koden **no[4]** i allra sista satsen ska exekveras där index **4** överstiger arrayens tillåtna maximala indexgräns som är **3** därför att **new** i början av programmet allokerar endast **4** minnesceller åt arrayen, nämligen de med index **0**, **1**, **2** och **3**. Någon minnescell med index **4** är inte allokerad. Därför kan vi inte heller referera till den. Men eftersom arrayens allokering sker med **new** och därmed under exekveringstid leder detta till exekveringsfel, medan kompilatorn godtar den syntaxmässigt korrekta koden **no[4]**. Programmet **ArrayObj** ger följande utskrift när den körs:

```
   Arrayens storlek:             4

   Arrayens default-initiering:  0       0       0       0

   Arrayen efter tilldelning:    64      86      34      -6

   Överskridning av arrayens index leder till programavbrott:

            no[4] inte definierad
            Index 4 överskrider gränsen: Exekveringsfel!

Unhandled Exception: System.IndexOutOfRangeException: Index
was outside the bounds of the array.
...
```

Default-initiering av array

Det är anmärkningsvärt att det som gäller för referensen **no** – att den är oinitierad när den skapas – inte gäller för själva arrayen. Referensen **no** är oinitierad och måste initieras explicit eftersom den är en lokal variabel i **Main()**. Men trots att även arrayen är lokal i **Main()** initieras dess element till **0** som är defaultvärdet till datamedlemmar av datatypen **int**. Detta visar att arrayen behandlas som ett objekt. Programmet **ArrayObj** skriver ut arrayelementens värden en gång *innan* och en andra gång *efter* att de har fått värdena **64**, **86**, **34** och **-6**. Generellt gäller:

I C# måste alla lokala variabler i en metod initieras innan de används.
Datamedlemmar i ett objekt initieras automatiskt till default-värden.
Att arrayelementen initieras till 0 (default) visar att arrayen är ett objekt.

En annan slutsats från utskriften av programmet **ArrayObj** är:

Att referera till icke-definierade element i en array leder till exekveringsfel.

C#-kompilatorn kontrollerar inte en arrays indexgränser: **ArrayObj** leder inte till kompileringsfel. Däremot kontrollerar C#-interpretatorn *(C# Virtual Machine)* indexgränserna och tillåter inte åtkomsten till icke-allokerade minnesplatser, dvs stoppar skräpvärden. Detta är ur datasäkerhetssynpunkt är en fördel. Programmen blir stabilare. C++ har i detta avseende en mer liberal attityd. Där ligger ansvaret för kontroll av indexgränserna helt och hållet hos programmeraren.

Att **no[4]** inte är definierat, fast talet **4** ”förekommer” i definitionssatsen **new int[4]**, beror på att **4** i hakparentesen av **no[4]** betyder *index*, medan **4** i **new int[4]** betyder *storlek*. Den korrekta tolkningen av **[]** beror på sammanhanget. **[]** är symbolen för tre olika operatorer som överlagrar varandra dvs betyder olika i olika sammanhang, se sid 103.

foreach-satsen

Denna sats som används i programmet **ArrayObj** (sid 100) är en ny kontrollstruktur som inte kunde tas upp i kapitlet om kontrollstrukturer *(Progr1)* därför att den förutsätter array-begreppet eller liknande sammansatta datatyper, som vi inte hade hunnit gå igenom då.

foreach-satsen är idealisk för att skriva ut sammansatta datatypers värden. Den gör samma sak som **for**-satsen, men har en lite annorlunda – ja t.o.m. lite enklare syntax, om man är förtrogen med arrays. I programmet **ArrayObj** (sid 100) ser satsen ut så här:

```
foreach (int element in no)
    Console.Write(element + "\t");
```

Översatt till svenska:

För varje **element** av arrayen **no**
Skriv ut elementet följt av en tabulator.

element – ett namn som är valt av oss – kallas för **foreach**-satsens *iterationsvariabel*. Den definieras till **int** och motsvarar **for**-satsens räknare. **element** pekar på värdet (innehållet) som står i arrayen. Iteration betyder upprepning och innebär här att satsens kropp upprepas: Programflödet fortskrider från element till element tills alla element är genomgångna. Det reserverade ordet **in** betyder *av* eller *element av*. **no** pekar på arrayen som ska loopas igenom. Därför: ” För varje **element** *av* arrayen **no**”.

foreach-satsens enkelhet består i att den till skillnad från **for**-satsen varken behöver ett start-, steg- eller slutvärde resp. avslutningsvillkor. Den går helt enkelt igenom arrayens *alla* element, från det första till det sista. Det är själva arrayen som bestämmer start-, steg- och slutvärdena. Variabeln **element** pekar i varje varv av loopen på resp. arrayelementets värde och kan sedan användas i loopens kropp för att göra det man önskar. I vårt exempel för att skriva ut arrayens element följt av en tabulator.
foreach-satsens iterationsvariabel måste ha samma datatyp som arrayelementen eller en sådan datatyp som arrayelementens datatyp automatiskt kan konverteras till. I vårt exempel har vi **int**. Det är t.o.m. möjligt att ha egendefinierade datatyper dvs klasser. Ett exempel på det är programmet **ArrayOfRef** (sid 108). Där deklareras iterationsvariabeln i en **foreach**-sats till den egendefinierade klassen **Fish** (sid 107), för att skriva ut ett **Fish**-objekts sort, vikt, längd, pris och frakt.

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

Hakparentesernas tre olika betydelser

1. **[] som storleksoperator** omsluter i definitioner med **new** *antalet* element i arrayen specificerar därmed arrayens *storlek*. T.ex. innebär koden

```
new int[4]
```

i programmet **ArrayObj** att **new** skapar en array av **int** med **4** element dvs att **4** minnesceller reserveras för lagring av **int**-värden. Det gemensamma för alla dessa element är att de lagras en efter den andra vid adressen eller referensen **no**:



Här är frågan om ”Hur många element?”. I matematiken kallas det *kardinaltal*.

2. **`[]` som indexeringsoperator** omslutar *indexet* till varje element av en array. Här handlar det om ett elements *position* i arrayen. Man anger index inom hakparenteser för att referera till elementet när man vill hämta eller tilldela det ett värde. Indexregeln (sid 99) tillämpas enligt vilken indexeringen börjar med `0`. Därför är `no[4]` i arrayen ovan inte definierat:

`no`	`no[0]`	`no[1]`	`no[2]`	`no[3]`

   Här är frågan om ”Vilket element?”. I matematiken kallas detta *ordinaltal*.

3. **`[]` som en del av datatypen** ”referens till array” omsluter ingenting utan är tom och skrivs direkt efter en datatyp för att definiera en ny referenstyp. T.ex. innebär satsen

```
int[] no;
```

   i programmet `ArrayObj` att en minnescell allokeras (en referensvariabel med namnet `no` definieras) för lagring av en adress till en `int`-array. Vi kan i fortsättningen använda namnet `no` för att komma åt arrayen vid denna adress. I satsen ovan har referensen `no` inte initierats. Det sker inte heller automatiskt, för `no` är en lokal variabel i `Main()`. Det sker först med tilldelningen `no = new int[4];` som initierar referensen explicit.

3.4 Hantering av array med referens

Man kan effektivisera hanteringen av arrays inte bara med **foreach**-satser utan även genom att använda sig av en s.k. *initieringslista* som slår ihop definitionen med initieringen – en kortform som ersätter koden **new**, men bibehåller dess egenskaper:

```
// ArrayRef.cs
// Initieringslista: Kortform för definition och initiering
// av en array i en och samma sats, inkluderar new implicit
// Utskrift av arrayens element med foreach-satsen
using System;

class ArrayRef
{
   static void Main()
   {
      int[] no = { 64, 86, 34, -6 };     // Initieringslista:
                                         // Definition OCH ini-
                                         // tiering av en array
//    int[] no = new int[4] { 64, 86, 34, -6 }; Gör samma sak

      Console.Write("\n\tVärdena från arrayen skrivs ut " +
                    " med referensen:\n\n\t");
      foreach (int element in no)
         Console.Write(element + "\t");
      int[] copy = no;                  // Ny referens copy tillde-
                                        // las referensen no
      Console.Write("\n\n\tArrayens värden skrivs ut" +
                    " med den nya referensen copy:\n\n\t");
      foreach (int element in copy)
         Console.Write(element + "\t");
      Console.WriteLine("\n\n\tEndast referensen kopieras," +
                                         " inte arrayen.\n");
   }
}
```

En körning visar att värdena i initieringslistan som först tillelas arrayen **no** verkligen kopierats över till arrayen **copy**, för det är de som skrivs ut:

```
      Arrayens värden skrivs ut med referensen no:

      64      86      34      -6

      Arrayens värden skrivs ut med den nya referensen copy:

      64      86      34      -6

      Endast referensen kopieras, inte arrayen.
```

Både definitionssatsen och initieringssatserna i programet **ArrayObj** (sid 100) – det är de 5 första satserna i **Main()** – kan slås ihop till en enda sats:

```
int[] no = { 64, 86, 34, -6 };
```

Satsen ovan är bara en förkortning på:

```
int[] no = new int[4] { 64, 86, 34, -6 };
```

Dvs initieringslistan kan skrivas efter **new int[4]** som egentligen skapar eller definierar arrayen. Men **new int[4]** får utelämnas. Detta visar att den förkortade versionen gör två saker: Först, fram till tilldelningstecknet definieras referensen **no** (*utan* någon uppgift om arrayens storlek). Sedan, från och med tilldelningstecknet tilldelas arrayen **no**:s element fyra värden som står i en kommaseparerad lista grupperad inom klamrarna **{ }** som kallas arrayens *initieringslista*. Kortformen gör precis samma sak som satsen med **new**. Kompilatorn får informationen om arrayens storlek genom att i initieringslistan räkna antalet element inom klamrarna **{ }**. Det är inte ens tillåtet att explicit ange det korrekta antalet element inom hakparenteserna **[ ]**. Det blir kompileringsfel om man gör det, därför att **no** endast är en referens till en array, inte arrayen själv. Observera även att man inte får använda initieringslistan separat utan endast i samma sats som definitionen.

Valet av variabelnamnet **copy** kan vara missledande i följande sats av programmet **ArrayRef** om man inte beaktar skillnaden mellan referens och array:

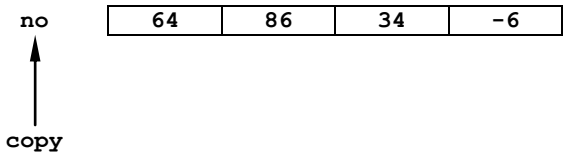
```
int[] copy = no;
```

copy blir nämligen en kopia av referensen **no** i satsen ovan, inte av arrayen – en ny referens som kommer att peka på samma array som den gamla referensen **no** pekar på. Det skapas ingen ny array eftersom det varken finns någon **new** eller någon initieringslista som skulle ersätta **new**. Anledningen till detta är – som vi konstaterat tidigare – följande viktigt faktum:

> En array i C# är alltid ett objekt som behöver en referens.

För att skapa ett objekt måste en **new**-sats skrivas. En referens definieras utan **new**.

Minnesmässigt lagras arrayen på *en och samma* adress som från programmet kan nås med referenserna **no** eller **copy**:



3.5 Array av referenser

Hittills har vi bildat arrays endast av den fördefinierade datatypen **int**. På samma sätt kan man också definiera arrays av alla andra enkla datatyper. Men kan man bilda även arrays av klasser dvs egendefinierade datatyper? Frågan måste preciseras: Menar man arrays av *referenser*, är svaret ja, därför att klasser – referensernas datatyper – har exakt samma ”rättigheter” som vilka andra datatyper som helst och kan därför skrivas överallt i koden där en fördefinierad datatyp kan stå. Precis som referensvariabler kan skrivas överallt, där även en variabel av enkel typ kan stå. Menar man arrays av *objekt*, är svaret nej, vilket vi kommer att förklara i detta avsnitt. Vi kommer att inse att en array av objekt inte är nödvändig, när man har en array av referenser vars element pekar på ett objekt. Array av referenser gör oss samma tjänst som array av objekt.

Vi börjar med att deklarera en klass so§m vi sedan i programmet **ArrayOfRef** (nästa sida) kommer att använda för att konstruera en array av referenser som i sin tur ska användas för att peka på objekt av denna klass:

```
// Fish.cs
// Deklarerar klassen Fish med 3 datamedlemmar och 2 metoder
using System;

class Fish
{
   public string sort;
   public float weight, size;

   public int Price()
   {
      return (int) Math.Round(weight * 7.25 / 100);
   }

   public int Shipping()
   {
      return (int) Math.Round(weight * 0.02 + size * 0.1);
   }
}
```

Klassen **Fish** modellerar en fisk med datamedlemmarna **sort**, **weight** och **size**. En laxforell t.ex. med en viss vikt i gram och en viss längd i cm kan vara ett objekt av denna klass, där laxforell är fiskens sort. Metoden **Price()** beräknar priset på fisken oberoende av sort, med 7,25 kr per hekto. Metoden **Shipping()** beräknar transportkostnaden utifrån fiskens vikt och längd genom att t.ex. multiplicera kostnadsfaktorn 0,02 med vikten och 0,1 med längden och addera dem. Båda Metoder returnerar priset och frakten i hela kronor utan ören. Biblioteksmetoden **Math.Round()** avrundar till närmaste heltal. Självklart kan man anmärka att den här modelleringen har vissa brister ur praktisk synpunkt: För det första är fiskpriser i praktiken inte oberoende av sorten. För det andra är både pris

och frakt i regel belopp i kronor och ören dvs decimaltal och inte heltal. Men vi gör medvetet båda förenklingar i modellen för att förenkla implementeringen och koncentrera oss på det programmeringstekniska konceptet av *array av referenser*. Vi vill nämligen använda detta koncept, för att på ett effektivt sätt skapa och hantera många objekt av klassen **Fish**. För det här ändamålet är de nämnda bristerna i modelleringen irrelevanta. Följande program skapar en array av referenser till **Fish**-objekt och anropar metoderna **Price()** och **Shipping()** för att sedan registrera (skriva ut) alla uppgifter till varje objekt:

```
// ArrayOfRef.cs
// Skapar först en array av 5 referenser till Fish-objekt, skapar
// sedan 5 Fish-objekt och tilldelar dem till referenserna.
using System;

class ArrayOfRef
{
  static void Main()
  {
    Fish[] f = new Fish[5];                    // Array av referenser
                                               // OBS! Inga objekt
    for (int i = 0; i < f.Length; i++)
    {
       f[i] = new Fish();                      // Skapar objekt och
                                               // tilldelar adressen
                                               // till en referens
       Console.Write("\n\tMata in sorten till fisk" + (i+1) + ":\t");
       f[i].sort = Console.ReadLine();         // InputCs
       if (f[i].sort.Length <= 7) f[i].sort += '\t';

       Console.Write("\tMata in vikten till fisk" + (i+1) + ":\t");
       f[i].weight  = (float) Convert.ToDecimal(Console.ReadLine());

       Console.Write("\tMata in längden till fisk" + (i+1) + ":\t");
       f[i].size = (float) Convert.ToDecimal(Console.ReadLine());
    }

  Console.Write("\nFisksort\tVikt i g\tLängd i cm\tPris\tFrakt\n"   +
  "-------------------------------------------------------------\n");
    foreach (Fish element in f)
    {
       Console.WriteLine(element.sort + "\t "  +
       element.weight  + "\t\t   " + element.size       + "\t\t " +
       element.Price() + "\t "     + element.Shipping() + "\n")   ;
    }
  }
}
```

I programmet **ArrayOfRef** skapas en array av 5 *referenser* till **Fish**-objekt med satsen:

```
Fish[] f = new Fish[5];
```

Observera att denna sats inte skapar något objekt alls, för då skulle det behövas koden **new Fish()** – OBS! parentesen – som inte finns med i satsen ovan. Förväntar man sig att en ”array av 5 **Fish**-objekt” skulle skapas med **new Fish()[5]** så är det fel, för den här koden kan inte kompileras – ett tecken på att begreppet ”array

av objekt" måste förkastas. Istället måste man gå *två* steg: Först måste en array av rena *referenser* definieras som i satsen ovan. Initieringsproblematiken löses automatiskt pga att en array alltid initieras till sin datatyps defaultvärden och att datatypen *referens* default-initieras till **`null`**. Då spelar det ingen roll om det handlar om referenser till objekt av klassen **`Fish`** eller av någon annan klass. Sedan kan man fundera hur man explicit initierar referenserna så att de pekar på verkliga objekt av typ **`Fish`**. Detta görs i programmet **`ArrayOfRef`** med:

```
f[i] = new Fish();
```

som står i **`for`**-satsen. Först efter den här satsen har vi allokerat minnesutrymme för *ETT objekt* av typ **`Fish`**, *inte* för en *array* av objekt, för i koden ovan finns inget spår av en sådan array. Detta objekts minnesadress tilldelas referensarrayelementet **`f[i]`** där **`i`** tack vare **`for`**-loopen går från **`0`** till **`4`**. Vi har endast att göra med en *array av referenser* till **`Fish`**-objekt, för hakparentesen – arrayens symbol – står efter referensvariabeln **`f`** som pekar på denna referensarray. Varje element i denna referensarray pekar i sin tur på ett separat **`Fish`**-objekt. De två stegen som tas är: Först från **`f`** till referensarrayen och sedan från den till objekten. Det första steget står utanför och det andra steget i **`for`**-loopen. Efter objektens definition initieras varje objekts datamedlemmar **`sort`**, **`weight`** och **`size`** i **`for`**-loopen till värden som läses in från konsolen. Sedan skrivs de fullständiga uppgifterna till varje objekt, dvs även priset samt fraktkostnaden, ut. Anropet av metoderna **`Price()`** och **`Shipping()`** är inbakade i utskriftssatsen. En körning av programmet **`ArrayOfRef`** kan ge följande slutlig dialog:

```
Mata in sorten till fisk1:      Laxforell
Mata in vikten till fisk1:      719
Mata in längden till fisk1:     38,5

Mata in sorten till fisk2:      Torsk
Mata in vikten till fisk2:      423
Mata in längden till fisk2:     28,7

Mata in sorten till fisk3:      Aborre
Mata in vikten till fisk3:      550
Mata in längden till fisk3:     25,5

Mata in sorten till fisk4:      Gädda
Mata in vikten till fisk4:      985
Mata in längden till fisk4:     58

Mata in sorten till fisk5:      Gös
Mata in vikten till fisk5:      395
Mata in längden till fisk5:     14
```

```
Fisksort        Vikt i g        Längd i cm      Pris    Frakt
-------------------------------------------------------------
Laxforell         719               38,5          52      18

Torsk             423               28,7          31      11

Aborre            550               25,5          40      14

Gädda             985               58            71      26

Gös               395               14            29      9
```

”Array av objekt” ?

För att kunna datorisera en verksamhet med fiskar behöver vi objekt av typ **`Fish`**. Självklart skulle man kunna skapa sådana objekt t.ex. med **`Fish f1 = new Fish();`** osv. Men vad gör man om man vill modellera en handel med stora fiskmängder under en längre period? Array skulle då vara den givna lösningen för att effektivisera kodningen. Men funderar man närmare på begreppet ”array av objekt” av typ **`Fish`** dyker upp följande fråga: Vilket defaultvärde ska t.ex. en array av **`Fish`**-objekt få vid initieringen? Till de enkla datatyperna i C# kommer de fördefinierade defaultvärdena **`0`**, tom sträng, **`null`**, nolltecknet och **`false`**. Men **`Fish`** är ju ingen fördefinierad datatyp. Det finns ingen begränsning på egendefinierade datatyper (klasser) och det går inte att förutsäga vilka man kan skapa i C#. Och därför går det inte heller att fastslå vilken default-initiering en sådan array skulle få. Vi ser att begreppet ”array av objekt” leder till en återvändsgränd. Lösningen är *array av referenser* – referenser till objekt dvs en tvåstegslösning som användes i programmet **`ArrayOfRef`** (sid 108).

3.6 Array som parameter i metoder

Array som bearbetar större datamängder ger upphov till mer komplexa och sofistikerade program. Exempel på det är applikationer som söker, sorterar eller krypterar data. Vi kommer i fortsättningen att behandla enkla varianter av sådana program. Modularisering är metoden för att bryta ned stora komplexa program i mindre och enklare moduler. Helst vill man ha program som består av ett antal enkla, överskådliga metoder där varje metod löser ett specifikt problem. Sedan vill man sätta ihop dem dvs anropa dem med ett antal parametrar från **`Main()`** och kontrollera hela händelseförloppet från denna metod som helst ska ha så lite kod som möjligt. Ju mer avancerade datatyper man använder i sitt program desto större blir behovet av modularisering. Självklart vill man även modularisera program som använder array. I C# är det möjligt att skicka en array som parameter till en metod dvs att definiera en array i parameterlistan. I nästa program definieras en **`void`**-metod **`Method()`** med en array av **`int`** som parameter:

```
// ArrayParam.cs
// Skickar en stor array till en metod, men:
// Array som parameter i en metod behandlas som en referens
// Parameteröverföring sker med referensen: adressen skickas
using System;

class ArrayParam
{
  static void Method(int[] b)       // Array som parameter
  {
    Console.WriteLine("\n\tI metoden\n\tär arrayens sista " +
                      "element före ändringen " + b[999]);
    b[999] = 1;                     // Ändringen
    Console.WriteLine("\n\t\t\t     och efter ändringen "   +
                      b[999] + '\n');
   }

/*********************************************************/

  static void Main()
  {
    int[] a = new int[1000];     // Array med 1000 nollor
    Console.WriteLine("\n\tI Main()\n\tär arrayens sista " +
                      "element FÖRE anropet   " + a[999]);

    Method(a);                   // Referensanrop: arrayens
                                 // adress skickas till metod
    Console.WriteLine("\tI Main()\n\tär arrayens sista " +
               "element EFTER anropet  " + a[999] + '\n');
   }
}
```

Låt oss börja titta på **`Main()`** innan vi går in på hur arrayen **`b`** i metoden **`Method()`** behandlas. I **`Main()`** har vi en **`int`**-array **`a`** med 1000 element, alla initierade till

default-värdet **0**. En körning av **ArrayParam** avslöjar även en del intressanta nyheter för oss. Den viktigaste är att en ändring som görs i en annan metod återspeglas i **Main()**:

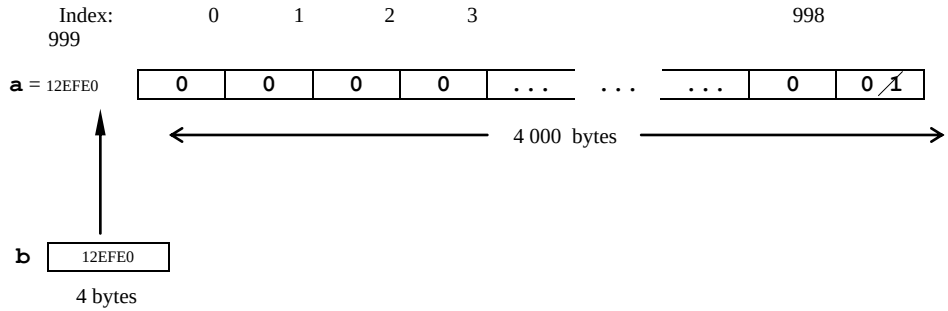
```
I Main()
är arrayens sista element FÖRE anropet   0

I metoden
är arrayens sista element före ändringen 0

                        och efter ändringen 1

I Main()
är arrayens sista element EFTER anropet  1
```

Som man ser har arrayen **a**:s sista element **a[999]** – kom ihåg att indexeringen hos arrays börjar med **0** – som hade initialvärdet **0**, EFTER anropet av metoden fått värdet **1**, fast denna ändring inte gjorts i **Main()** utan i metoden **Method()**, dessutom med arrayen **b** och inte med **a**. Detta verkar bryta mot de regler vi lärt oss om lokala variablers livslängd, därför att **a** trots allt är en lokal variabel i **Main()** och därmed inte giltig i **Method()**. Samma sak gäller för **b** som är lokal variabel i **Method()** och därmed inte giltig i **Main()**. Gåtans lösning är att det handlar endast om *en och samma* array till vilken **a** och **b** är bara två olika *referenser*. Därför pratar vi i utskriften ovan inte om arrayen **a** och inte om arrayen **b** utan om *arrayen*, för det finns bara en. För att första detta bättre låt oss titta på följande minnesbild som ska förtydliga vad som händer i programmet **ArrayParam**:



Vi vet att varje **int** tar 4 bytes i minnesutrymme. Därmed tar hela arrayen **a** med 1 000 **int**-element 4 000 bytes. Detta ”stora” minnesutrymme allokeras av satsen:

```
int[] a = new int[1000];
```

a är en referensvariabel som lagrar ett hexadecimalt tal, säg 12EFE0 (decimalt: 1241056) som är arrayens adress. Adresser visas i datavärlden – det är en de factostandard – som tal i hexadecimalt format. Med *adress* menas alltid en plats i datorns *RAM*-minne (*Random Access Memory*). När en array definieras lagras den vid en adress och arraynamnet blir en länk mellan programmet och denna fysiska ad-

ress. När arrayen **a** sedan i metodanropet **Method(a);** skickas som en aktuell parameter, då överförs inte arrayens värden utan arrayens *adress* till metoden **Method()**. Denna adress tas emot av den formella parametern **b** som är definierad i metodens parameterlista som en array av **int**. På så sätt hamnar **a**:s adress, det hexadecimala talet 12EFE0 i minnescellen **b**. Dvs **b** lagrar **a**:s adress som tar 4 bytes. Därmed pekar både **a** och **b** på en och samma array. Någon kopiering av arrayinnehållet på 4 000 bytes till en ny plats förekommer inte. Endast adressen på 4 bytes kopieras till **b** vid metodanropet. I **Main()** kommer man åt arrayen med **a** och i **Method()** gör man det med **b**. När vi sedan i **Method()** ändrar värdet i arrayens sista element med **b** från **0** till **1**, kan ändringen ses i **Main()** med **a**.

Den ovan beskrivna metoden för överföring av parametrar kallas *referensanrop*. Dvs inte parametrarnas värden utan deras adresser överförs vid metodanropet. När parametrarnas *adresser* överförs och inte deras värden, förekommer ingen fördubbling av minnesåtgång. Alla eventuella ändringar i metoden återspeglas i **Main()**. Valet av parameteröverföringsmetod styrs av datatypen:

I C# väljs automatiskt referensanrop (Call by reference) för parameteröverföring vid metodanrop, om parametern är av datatypen array.

Låt oss nu även gå in på med vilken syntax programmet **ArrayParam** använder en array som en parameter i en metod.

1. Att definiera en metod med array som parameter
har gjorts i metoden **Method()** genom att definiera den formella parametern som en array av **int** dvs samma datatyp som den aktuella parametern har i anropet:

```
int[] b
```

Antalet element inom hakparentesen får inte anges. Att antalet element inte behövs här beror på att en formell parameter får sitt initialvärde från den anropande metoden. Även arraystorleken följer med vid anropet. Detta har i sin tur att göra med att hela definitionen av en metod endast är en mall, en föreskrift om vad som ska hända om metoden anropas, en potentiell kod som blir aktuell först när vi anropar metoden. I metoden **Method()** står definitionen av parametern **b** till datatypen array av **int** som vanligt i parameterlistan och därmed i metodhuvudet:

```
static void Method(int[] b)
```

2. Att anropa en metod med array som parameter
sker genom att skriva den aktuella parametern som array utan hakparenteser i anropet:

```
Method(a);
```

Anmärkningsvärt är att det för första gången dyker upp en array utan hakparenteser. Så, tittar man inte på definitionssatsen några rader ovan kan man inte känna igen **a** som array. Anledningen till att hakparentesen inte får stå efter arrayen **a** i anropssatsen är just det vi sade ovan om referensanrop: Anropet skickar inte hela

arrayen med dess vär-den till **`Method()`** utan endast referensen **`a`**. En hakparentesens skulle tolkas som kod som anger index som specificerar ett visst element i arrayen. En anropssats av typen **`Method(a[999]);`** skulle skicka endast *ett element* av arrayen nämligen det med index **`999`**. Det blir i så fall ett *tal* av typ **`int`** som skickas till metoden. Man kommer att få kompileringsfel i alla fall eftersom metodens formella parameter **`b`** är definierad som en *array* av **`int`** och inte som en vanlig **`int`**. Den enkla datatypen **`int`** kan inte konverteras till den sammansatta datatypen array av **`int`**. De automatiska typkonverteringsreglerna gäller endast för enkla datatyper. Det tänkbara alternativet **`Method(a[]);`** fungerar inte heller av samma anledning: Det handlar om en icke-definitionssats där hakparentesens innehåll tolkas som index. Men index får aldrig utelämnas (se punkt **1**). För att skicka en array som parameter till en metod måste alltså arrayen i metodanropet skrivas endast med arraynamnet *utan hakparentes*. Självklart måste arrayen innan anropet vara definierad i **`Main()`** som vanligt med hakparentes och en uppgift om storleken. Arraynamnet används vid anropet som adressen till arrayen.

3.7 Hantering av slumptal i C#

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

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

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

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

```
// DoRand.cs
// Skriver ut 5 slumptal mellan 1 och int.MaxValue samt
//            20         mellan 1 och 20
// Anropar två varianter av Random-metoden Next() en gång
// med ingen parameter, en gång med två paramtrar
using System;
class DoRand
{
  static void Main()
  {
    int i = 1, j = 1;
    Random r = new Random(); // Objekt av klassen Random
    Console.WriteLine("Slumptal mellan 1 & int.MaxValue:\n");
    do                       // do-loop
       Console.WriteLine("\t" + r.Next());
    while (i++ < 5);         // i testas först, ökar sedan

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

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

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

```
Slumptal mellan 1 & int.MaxValue:

        1460841191
        225482400
        1438321568
        1700127070
        1513406452

Slumptal mellan 1 och 20:

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

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

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

Array av slumptal

Eftersom vi i fortsättningen kommer att jobba med flera program som använder slumptal lagrade i en array vill vi skriva en metod som kan användas av alla dessa program. Vi har valt formen av en **`void`**-metod för att generera ett antal slumpvärden och tilldela dem till elementen i en array:

```
// RandArray.cs
// Definierar en metod Rand() som lagrar slumptal
// i arrayen no och skriver ut dem
// Anropar biblioteksmetoden Next(a, b) för att få ETT
// slumptal mellan a och b i varje varv av for-loopen

using System;
```

```
class RandArray
{
   public static void Rand(Random r, int[] no, int a, int b)
   {
      Console.Write("\n\t" + no.Length + " heltal mellan " +
                  a + " och " + b + " slumpas fram:\n\n\t");
      for (int i=0; i < no.Length; i++)
      {
          no[i] = r.Next(a, b);
          Console.Write(no[i] + " ");
          if ((i % 16 == 0) && (i != 0))
             Console.Write("\n\t");
      }
      Console.WriteLine("\n\n");
   }
}
```

För förståelse av biblioteksmetoden **`Next()`** hänvisas till hantering av slumptal. Det nya i koden ovan är att slumptalen lagras i en array som kommer att användas av fler program vilket demonstrerar inte bara modularisering utan även återanvändning av kod. Filen ovan innehåller inte ett fullständigt program utan endast en klass med **`void`**-metoden **`Rand()`** som har fyra parametrar varav den ena är en array av **`int`**, kallad **`no`** som lagrar slumptalen. Arrayen deklareras i parameterlistan och tilldelas i kroppen mellan **`a`** och **`b`** via satsen:

```
no[i] = r.Next(a, b);
```

som i en **`for`**-sats anropar den biblioteksmetoden **`Next()`** som i sin tur i varje varv av loopen slumpar fram *ett* slumptal mellan **`a`** och **`b`**. Vi har använt denna metod tidigare i andra program. **`for`**-satsen som anropar metoden skriver ut slumptalen.

3.8 Sökning och sortering

Ett viktigt – numera självklart – användningsområde för datorer är sökning i och sortering av stora datamängder. Programmeringstekniskt sett kan sådana applikationer inte skrivas utan array (eller högre datastrukturer). Därför är sökning och sortering klassiska tillämpningar för sammansatta datatyper. Samtidigt ökar behovet av modularisering ju mer avancerade datatyper man använder i sitt program. Nu när vi lärt oss att skicka arrays som parametrar till metoder, kan vi modularisera program som arbetar med arrays. Detta är nödvändigt för att koncentrera sig på den egentliga uppgiften nämligen sökning, sortering eller andra applikationer som t.ex. kryptering som kommer att tas upp i nästa avsnitt. När man söker eller sorterar data finns redan ett material i form av databaser, tabeller eller listor osv. som man använder.

För att skaffa underlag för våra testprogram har vi valt att producera slumptal och lagra dem i en array.

Följande program skapar med hjälp av metoden **`Rand()`** som är definierad i programmet **`RandArray`** (sid 116) en array av 200 slumptal mellan 1 och 1000 och skriver ut dem. Sedan läser det in ett tal som ska hittas gemom att anropa metoden **`MySearch()`** som är definierad i programmet **`Search`** (sid 118):

```
// SearchTest.cs
// Skapar en array och skickar den till metoden Rand() där
// den tilldelas slumptal. Ändringen fås tillbaka pga refe
// rensanrop. Den tilldelade arrayen skickas vidare till
// metoden MySearch() som söker efter ett inläst tal bland
// slumptalen.
using System;

class SearchTest
{
  static void Main()
  {
    Random r = new Random();
    int a = 1, b = 1000, searchedNo;
    int[] intArray = new int[200];      // Default-initiering
    RandArray.Rand(r, intArray, a, b);  // Slump-tilldelning

    Console.Write("\tAnge tal som programmet ska söka " +
                                          + "efter:\t");
    searchedNo = int.Parse(Console.ReadLine()); // Sökt tal

    Search.MySearch(intArray, searchedNo);      // Anrop av
  }                                              // sökmetoden
}
```

Även om vi inte gått igenom programmets alla delar – klassen **Search** med metoden **MySearch()** – ska vi titta på en körning för att bättre förstå vad som händer:

```
200 heltal mellan 1 och 1000 slumpas fram:

237 255 104 898 422 575 712 34 775 299 192 530 442 17 656 344 276
18 929 282 720 967 336 17 934 378 427 667 600 787 581 838 346
525 224 576 710 484 865 211 360 686 858 798 455 501 142 521 138
405 101 747 951 13 889 271 567 88 612 45 796 46 82 989 366
355 832 918 441 728 635 440 801 719 570 35 757 539 563 434 237
907 177 843 334 835 535 981 637 954 657 623 520 468 63 315 252
870 80 101 317 872 728 58 771 662 594 880 444 502 162 676 173
179 809 890 517 887 303 532 468 852 282 488 719 660 568 981 657
256 784 888 460 463 118 13 180 120 73 673 242 303 538 783 793
982 98 342 660 174 446 13 215 549 281 113 591 241 987 759 95
261 224 836 719 922 217 711 709 444 358 398 815 631 938 166 962
147 696 738 563 874 322 484 811 419 674 912 830 653 423 587 781
962 226 982 80 703 712 519

Ange tal som programmet ska söka efter: 519
Det sökta talet 519 är det 200:e elementet bland talen ovan.
```

Anrop av RandArray.Rand()

Anrop av Search.MySearch()

I programmet **SearchTest**:s **Main()**-metod finns bara anrop av två metoder samt definition av deras aktuella parametrar och inläsning av det sökta talet. En array av **int** har definierats med **200** element och tilldelats referensen **intArray**. I anropssatsen **RandArray.Rand(r, intArray, a, b);** skickas arrayen till metoden. Det anmärkningsvärda är följande: När arrayen **intArray** som aktuell parameter i anropet överförs till den formella parametern **no** i metoden **RandArray.Rand()**, är den definierad och default-initierad till **0**-värden. Faktum är att, när parametern är en array, så används en metod för parameteröverföring där den aktuella parametern **intArray**, och den formella parametern **no**, behandlas som endast två olika *referenser* till ett och samma minnesområde, dvs till en och samma fysisk array. Metoden kallas för *referensanrop*. Med **intArray** definierar vi arrayen i **Main()** och anropar **RandArray.Rand()**. Med **no** tilldelar vi samma array i metoden **RandArray.Rand()** slumpvärden som överskriver arrayens default-värden. En sådan ”arbetsdelning” mellan olika metoder kan endast göras med referensanrop.

Efter anropet av slumpmetoden läses in ett värde till variabeln **searchedNo** som tillsammans med arrayen **intArray** skickas till metoden **Search.MySearch()**. När **MySearch()** anropas är arrayen **intArray** både definierad och tilldelad slumpvärden. Sökmetoden får alltså slumptalsvärden som överförs till den formella parametern **t**. Vid sidan om **no** är **t** nu en till minnescell som lagrar arrayen **intArray**:s adress i detta program. Även den här parameteröverföringen sker med referensanrop. Vid anropet skickas inte värdena i arrayelementen till metoden utan endast adressen som lagras i **intArray**. I själva verket är det arrayens adress som överförs till **MySearch()**, tas emot av **t** och används sedan i sökmetoden för att hitta det sökta talet i arrayen:

```
// Search.cs
// Metoden MySearch() tar emot två parametrar: arrayen t och
// heltalet s, det sökta elementet. Söker efter den första
// förekomsten av s bland arrayelementen.
using System;

class Search
{
  public static void MySearch(int[] t, int s)
  {
    int i;
    for (i = 0; i < t.Length; i++)  // Söker igenom array t
        if (t[i] == s)              // Sökkriteriet
        {
           Console.WriteLine("\n\tDet sökta talet " + t[i] +
                       " är det " + (i+1) + ":e elementet" +
                       " bland talen ovan.\n\n");
           break;                   // Bryter for-satsen
        }                           // när det sökta hittats
    if (i == t.Length)
       Console.WriteLine("\n\tDet sökta talet finns ej " +
                                  "bland talen ovan.\n\n");
  }
}
```

Det sökta talet skickas med den aktuella parametern **searchedNo** och tas emot av den formella parametern **s**. Nu ska vi titta på vad **void**-metoden **MySearch()** egentligen gör och hur den hittar eller inte hittar det sökta talet. Arrayen och det sökta talet är givna. Frågan är: finns det sökta talet i arrayen? Om ja, på vilken position? Algoritmen är väldigt rak och enkel och kallas för *linjär sökalgoritm*:

1. Gå igenom alla element i arrayen dvs sök igenom arrayen **t** från början till slutet (linjär sökning).
2. Jämför varje element med det sökta talet. Finns likhet med något element, skriv ut ett hittat-meddelande samt elementets position som är lika med index + 1. Har du hittat en likhet avbryt sökningen.
3. Har du gått igenom alla arrayelement utan att hitta någon likhet skriv ut ett ej-hittat-meddelande.

Denna algoritm hittar endast den första förekomsten av det sökta talet i arrayen och tar inte hänsyn till att det ev. kan finnas flera exemplar av det sökta talet i arrayen. Progammeringstekniskt har vi översatt algoritmens punkt **1** till C#-kod genom att i metoden **MySearch()** skriva en **for**-sats som söker igenom arrayen **t** från index **0** till **t.Length-1**. I denna **for**-sats finns en **if**-sats som implementerar algoritmens punkt **2** och i sin tur innehåller två satser: Hittat-meddelandet och **break**-satsen. En **break**-sats avbryter alltid den loop eller den **switch**-sats i vilken den står, här alltså **for**-satsen. Det är den som enligt anvisningen i punkt **2** gör att programmet endast hittar den första förekomsten av det sökta talet i arrayen. I punkt

3:s implementering – den sista **if**-satsen i **MySearch()** – utnyttjar vi att **for**-satsens räknare **i** är väl definierad även *efter* **for**-satsen och att den har kvar det värde den fick där. Om sökningen gått igenom alla arrayelement utan att hitta något element som är lika med det sökta talet, har **for**-satsens räknare **i** nått värdet **t.Length** eftersom detta är första värdet som inte uppfyller **for**-satsens villkor **i < t.Length**. I detta fall avslutas **for**-satsen utan **break** så att villkoret till den efterföljande **if**-satsen blir uppfyllt och skriver ut ett Ej-hittat-meddelande.

Bubbelsortering

Sökning i och sortering av stora datamängder är klassiska tillämpningar för sammansatta datatyper, speciellt för arrays. Medan sökning i förra exemplet baserades på en linjär algoritm, bygger sortering på en ny algoritm, även om den har vissa likheter med sökning. Vi ska fortsätta kapitlet om arrays med en sorteringsalgoritm som är en vidareutveckling av algoritmen för platsbyte av *två* värden. Vi har i programmet **MiniSort** (sid 44) använt denna algoritm på två tecken:

```
if (char1 > char2)
{
   temp = char1;
   char1 = char2;
   char2 = temp;
}
```

Om tecknen står i fel ordning ska de byta plats. För att göra det läggs **char1**:s värde undan i en tredje, temporär variabel **temp**. Sedan tar vi **char2**:s värde och lägger det i **char1**. Till sist läggs värdet i **temp** (som ju har mellanlagrat **char1**:s värde) in i **char2**. Illustrationen på sid 44 bör underlätta förståelsen av denna process. I själva verket beskriver den en algoritm för sortering av *två* värden. För att utvidga algoritmen till *flera* värden kopplar vi den till den linjära sökalgoritmen som vi använde för sökning. Principen där var en **if**-sats inbakad i en **for**-sats. **for**-satsen söker igenom värdena i en array och **if**-satsen innehåller sökkriteriet. När det gäller sortering måste **if**-satsen istället byta plats på två värden om de står i fel ordning. Denna **if**-sats har vi ju redan skrivit för två tecken (se ovan). Det gäller bara att formulera den för två arrayelement och stoppa in den i en **for**-sats:

```
for (i = 0;  i < n-1;  i++)
     if (t[i] > t[i+1])
     {
        temp = t[i];
        t[i] = t[i+1];
        t[i+1] = temp;
     }
```

där **t** är en array som innehåller värdena som ska sorteras och **n** antalet element i arrayen. När två på varandra följande arrayelement **t[i]** och **t[i+1]** står i oönskad ordning ska de byta plats där **i** genomlöper alla index. Man skulle kunna tro att problemet vore löst med detta. Men eftersom **if**-satsen endast testar om *två grannvärden* står i fel ordning och byter sedan plats på *dem*, räcker koden ovan in-

te till att sortera arrayen fullständigt, även om **for**-satsen söker igenom hela arrayen. Jämförelsen mellan två grannvärden tar inte hänsyn till värden som står längre bort. Om man tillämpar koden ovan på en array av 20 heltal som med metoden **RandArray.Rand()** är utvalda ur intervallet [1, 100] får man följande resultat:

```
20 heltal mellan 1 och 100 slumpas fram:

75 2 24 94 30 88 10 42 50 54 27 47 45 83 34 86 67 66 14 14

De 20 slumptalen efter koden ovan:

2 24 75 30 88 10 42 50 54 27 47 45 83 34 86 67 66 14 14 94
```

Resultatet visar att sorteringen inte är klar, men att vi är på rätt väg. Arrayen är *delvis* sorterad. Bara om två grannvärden stod i fel ordning har de bytt plats och detta har gjorts löpande genom hela arrayen. Denna delsortering kallas för ett *pass* i en sorteringsalgoritm som är känd under beteckningen *bubbelsortering*. För att uppnå en fullständig sortering måste detta pass upprepas flera gånger vilket innebär att lägga in ovanstående **for**-sats i en ny **for**-sats som går igenom flera pass. I varje pass kommer en del värden att placera sig i rätt ordning. Metoden kan jämföras med luftbubblor i vattnet som så småningom stiger upp till vattenytan. Därav namnet *bubbel*sortering vars algoritm är implementerad i följande **void**-metod:

```
// Bubble.cs
// Sorterar heltal lagrade i arrayen t med en algoritm
// (bubbelsortering) som baseras på algoritmen för plats-
// byte av två objekt i programmet MiniSort (sid 44)
using System;
class Bubble
{
  public static void sort(int[] t)
  {
    int temp;
    for (int pass=0; pass<t.Length-1; pass++)
        for (int i=0; i<t.Length-1; i++)
            if (t[i] > t[i+1])      // Sortering i stigande
            {                       // ordning
               temp = t[i];         // Algoritm för platsbyte
               t[i] = t[i+1];       // av två grannelement:
               t[i+1] = temp;       // t[i] och t[i+1]
            }
    Console.WriteLine("\tDe " + t.Length +
                      " slumptalen efter sortering:");
    Console.Write("\n\t");
    for (int i=0; i < t.Length; i++)  // Sorterad utskrift
        Console.Write(t[i] + " ");
    Console.WriteLine("\n\n");
  }
}
```

Bubbelsorteringsalgoritmen består alltså av en **if**-sats inbakad i en nästlad **for**-sats där **if**-satsen implementerar algoritmen för platsbyte av *två* värden. Den inre **for**-satsen söker igenom arrayelementen, utför *ett* sorteringspass och den yttre **for**-satsen upprepar sorteringspassen. Metoden **sort()** har arrayen **t** som ska sorteras som parameter och används i den inre **for**-satsen. Den anropas från **Main()** i följande program efter definitionen av arrayen **intArray** och dess tilldelning i metoden **RandArray.Rand()**:

```
// BubbleTest.cs
using System;

class BubbleTest
{
   static void Main()
   {
      Random r = new Random();
      int a = 1, b = 100;
      int[] intArray = new int[17];
      RandArray.Rand(r, intArray, a, b);
      Bubble.sort(intArray);
   }
}
```

En körning av programmet **BubbleTest** visar att sorteringen nu genomförts fullständigt:

```
        17 heltal mellan 1 och 100 slumpas fram:

        23 76 23 31 67 94 79 38 46 10 85 100 87 61 17 71 14


        De 17 slumptalen efter sortering:

        10 14 17 23 23 31 38 46 61 67 71 76 79 85 87 94 100
```

Andra algoritmer

Som en sista anmärkning till kapitlet sökning och sortering bör påpekas att de algoritmer som avhandlats här, är enkla och elementära. De är däremot inte de mest effektiva när det gäller att minimera antalet operationer och maximera snabbheten. Det finns effektivare (och mer komplicerade) algoritmer både när det gäller sökning och sortering som vi inte tar upp här. Vi nämner bara en algoritm som kallas *binärsökning* som heter så för att den i varje steg halverar arrayen man ska söka i. Den behöver ett mindre antal operationer och är därmed snabbare. När det gäller sortering finns den effektiva algoritmen *Quicksort* som bygger på *rekursion*. Rekursiva metoder är metoder som anropar sig själva – ett alternativ till repetition (loopar).

3.9 Generiska metoder

I programmering är variabler → platshållare för värden.
I *generiska metoder* kan variabler även användas som platshållare för ***datatyper*.**

Generiska metoder är metoder vars parametrar har variabla datatyper. Ex.: I metoden `public static void G_out <T>(T[] t)` är parametern `t` är en array av typ `T` där `T` är en platshållare för datatyper. Den variabla datatypen `T` (`T`ype) definieras med `<T>` och kan användas istället för vilken datatyp som helst: `int`, `double`, `char`, `string`, … .

I generiska metoder är de formella parametrarnas datatyper inte specifierade. De bestäms först när metoderna anropas, närmare bestämt av de aktuella parameternas datatyper. Detta innebär en generalisering som kallas för *Generics* som även kan tillämpas på klasser. Man kan skriva ETT program för många tillämpningar.

Generics

I de flesta programmeringsspråken har man infört *Generics* som ett tillägg till standarden först i de nyare versioner av språket. T.ex. i C++ kom motsvarigheten till generics först på 90-talet och kallades för *Templates*. I Java introducerades generics 2004. I C# har det funnits stöd för *Generics* sedan 2005.

Genom att använda *Generics* behöver man inte längre skriva olika varianter av ett program som i praktiken löser (nästan) samma problem. Dessa skiljer sig programmeringstekniskt endast i datatypen till de involverade parametrarna. Alla dessa varianter kan förenas i ett och samma – numera *generiskt* – program i vilka datatyperna är variabler. Låt oss säga, vi vill skriva ett program för sortering av olika slags objekt. Det kan handla om sortering av heltal, decimaltal, bokstäver, strängar, eller … . Sorteringsalgoritmen till alla dessa program är den samma oavsett man sorterar heltal, decimaltal, bokstäver eller strängar. Metoden som implementerar algoritmen skrivs då generiskt, dvs med variabla datatyper, så att den kan användas för att sortera olika typer av objekt beroende på i vilket syfte den anropas. Låt oss titta på följande exempel:

```
// G_Output.cs
// Generisk metod G_out <>() skriver ut en array av variabel
// datatyp T som kan vara int, double, char eller string.
// foreach loopar igenom och skriver ut listans alla element
using System;
using System.Collections.Generic;
class G_Output
{
   public static void G_out <T>(T[] t)
   {
```

```
      Console.Write("\t");
      foreach (T element in t)
         Console.Write(element + "  ");
      Console.WriteLine("\n");
   }
}
```

Det som gör att metoden **G_out <>()** ovan som är definierad i klassen **G_Output** är generisk är den annorlunda syntaxen i metodhuvudet:

```
public static void G_out <T>(T[] t)
```

Till skillnad från vanliga metoder har denna metod *två* parameterlistor. Den ena är den vanliga med runda parenteser **(T[] t)** som innehåller parametern **t**, bara att dess datatyp är en array av **T**. Den andra är den ”generiska parameterlistan” **<T>** där **T** definieras som en formell parameter för en datatyp som bestäms när metoden anropas, t.ex. så här: **G_Output.G_out(hel);** **T** får den datatyp som i det anropande programmet har tilldelats variabeln **hel**. Har vi t.ex. definierat **hel** som en **int**, så antar den formella parametern **T** den aktuella parametern **int**. I generiska metoder finns det alltid en sådan *typ-parameter*. I det program där vi testar generiska metoder, anropas **G_out <>()** fyra gånger, varje gång med en annan datatyp, närmare bestämt med **int**, **double**, **char** och **string**. Med hjälp av dessa bildas sedan med koden **T[]** arrays av **int**, **double**, **char** och **string**. Den vanliga parametern **t** definieras då med koden **T[] t** till sådana arrays. Här följer nu det program som testar och anropar två generiska metoder:

```
// GenericTest.cs
// Testar de generiska metoderna G_out <>() och G_sort <>()
// Skapar 4 arrays av olika typer: int, double, char, string
// och skickar dem till G_out <>() för utskrift och till
//                      G_sort <>() för sortering
// Generiska metoderna anropas som vanliga metoder
// Utskrift sker före och efter sortering
using System;
class GenericTest
{
   static void  Main()
   {
     int[] hel = { 9,  7,  2,  1,  8,  5,  4,  3,  6 };
     double[] deci =
            { 9.9, 7.7, 2.2, 1.1, 8.8, 5.5, 4.4, 3.3, 6.6 };
     char[] boks = {'h','c','f','a','e', 'i', 'b', 'd', 'g'};
     string[] text = {"zeta","beta","gamma", "psi", "alpha"};
     Console.WriteLine(
      "\n\tOlika datatyper skrivs ut med samma generiska" +
      " metod \n\tFÖRE SORTERING:\n");  // Osorterad utskrift
     G_Output.G_out(hel);               // Anrop av generisk
     G_Output.G_out(deci);              // metod G_out <>()
     G_Output.G_out(boks);
```

```
      G_Output.G_out(text);
      Console.WriteLine(
      "\tDe olika typerna sorteras med samma generisk metod");
      G_Bubble.G_sort(hel);               // Sortering: Anrop
      G_Bubble.G_sort(deci);              // av generisk metod
      G_Bubble.G_sort(boks);              // G_sort <>()
      G_Bubble.G_sort(text);
      Console.WriteLine("\toch skrivs ut EFTER SORTERING:\n");
      G_Output.G_out(hel);                // Sorterad utskrift
      G_Output.G_out(deci);
      G_Output.G_out(boks);
      G_Output.G_out(text);
   }
}
```

Den vitmarkerade koden visar fyra anrop av den generiska metoden **`G_out <>()`**. Det anmärkningsvärda är att dessa anrop inte skiljer sig alls från anrop av vanliga metoder. De aktuella parametrarna **`hel`**, **`deci`**, **`boks`** och **`text`** är definierade som arrays av **`int`**, **`double`**, **`char`** resp. **`string`** och skickar, när de anropas, inte bara sina vanliga värden – heltalen, decimaltalen, bokstäverna och strängarna – till de anropade metoderna, utan även sina datatyper. Medan de vanliga värdena i resp. array går till den formella parametern **`t`** i resp. metods runda parameterlista, går datatyperna arrays av **`int`**, **`double`**, **`char`** och **`string`** till parametern **`T`** i resp. metods ”generiska” parameterlista **`<T>`**. Därmed blir varje datatyp specificerad och insatt på alla ställen där **`T`** står i den generiska metoden, vare sig i huvudet eller i kroppen. Så här blir resultatet av en körning av programmet **`GenericTest`**:

```
Olika datatyper skrivs ut med samma generiska metod
FÖRE SORTERING:

9  7  2  1  8  5  4  3  6

9,9  7,7  2,2  1,1  8,8  5,5  4,4  3,3  6,6

h  c  f  a  e  i  b  d  g

zeta  beta  gamma  psi  alpha

De olika typerna sorteras med samma generiska metod
och skrivs ut EFTER SORTERING:

1  2  3  4  5  6  7  8  9

1,1  2,2  3,3  4,4  5,5  6,6  7,7  8,8  9,9

a  b  c  d  e  f  g  h  i

alpha  beta  gamma  psi  zeta
```

Som man ser har heltalen, decimaltalen, bokstäverna och strängarna dvs värdena i de fyra olika arrays skrivits ut som ett resultat av de vitmarkerade anropen i programmet **GenericTest** på förra sidan. Alla fyra anrop har gått till en och samma generisk metod **G_out <>()** (sid 124) som skriver ut dem. Visserligen behöver man skriva fyra olika anrop i programmet **GenericTest**. Men man behöver definiera och koda själva metoden bara en gång, vilket innebär en stor effektivitet i utvecklingsarbetet.

Generisk bubbelsortering

Men körresultatet ovan har också andra delar, precis som själva programmet **GenericTest**. Efter att värdena skrivits ut skickas de till en annan generisk metod som sorterar dem. Detta görs i **GenericTest** med anropen:

```
G_Bubble.G_sort(hel);
G_Bubble.G_sort(deci);
G_Bubble.G_sort(boks);
G_Bubble.G_sort(text);
```

Även dessa anrop kan man inte skilja från anrop till vanliga metoder, fast metoden **G_sort <>()** är generisk. Efter sorteringen skickas arrayvärdena igen till utskrift, så att vi ser dem sorterade i utskriften ovan – och detta sker inte bara för hel- och decimaltalen samt bokstäverna utan även för strängarna. Även här använder vi oss av en enda generisk metod som vi nu ska titta närmare på:

```
// G_Bubble.cs
// Genersik metod G_sort <>() sorterar en array av variabel
// datatyp T som kan vara int, double, char eller string
using System;
using System.Collections.Generic;

class G_Bubble
{
  public static void G_sort <T>(T[] t) where T :
                                                IComparable<T>
  {                                     // Krävs för CompareTo()
    T temp;
    for (int pass=0; pass<t.Length-1; pass++)
        for (int i=0; i<t.Length-1; i++)
            if (t[i].CompareTo(t[i + 1]) > 0) // Om t[i] >
                                                 t[i+1]
            {                                 // Sortering i sti-
               temp   = t[i];                 // gande ordning
               t[i]   = t[i + 1];             // Algoritm för
               t[i+1] = temp;                 // platsbyte
            }
  }
}
```

Metoden **G_sort <>()** i klassen **G_Bubble** är en generisk variant av den vanliga metoden **sort()** i klassen **Bubble** som presenterades när vi behandlade sökning och sortering. Här gäller samma som vi sa om metoden **G_out <>()**: Den generiska formella parametern **T** står för datatyper som är kopplade till den aktuella anropsparametern som skickas till den vanliga formella parametern **t**, dvs för datatyperna till de objekt som ska sorteras.

Constraints

Till skillnad från **G_out <>()** har vi i den generiska metoden **G_sort <>()** ett tillägg i metodhuvudet:

```
public static void G_sort <T>(T[] t) where T : IComparable<T>
```

Tillägget **where T : IComparable<T>** är en s.k. *constraint*, dvs en *restriktion* som läggs på **T**. Den är nödvändig eftersom vi i metodens kropp använder oss av ett villkor i **if**-satsens huvud som ska jämföra två på varandra följande element i arrayen:

```
if (t[i].CompareTo(t[i + 1]) > 0)
```

Motsvarigheten till detta i den vanliga icke-generiska metoden **sort()** är:

```
if (t[i] > t[i + 1])
```

Anledningen till att denna kod inte fungerar i den generiska metoden är att vi inte längre har att göra med en array av **int** vars element ska jämföras med varandra, utan med en generaliserad datatyp **T** som kan vara vilken datatyp som helst. Hur ska koden avgöra sanningsvärdet till ett sådant villkor om **T** är t.ex. en sträng? Självfallet måste den ta strängarnas begynnelsebokstäver och jämföra deras ASCII-koder med varandra för att avgöra vilken som är större. Men en sådan ”intelligens” finns inte automatikst inlagd i den generaliserade datatypen **T**, utan den är förprogrammerad i metoden **CompareTo()**. För att kunna åt denna kod måste **T** ärva denna metod som i sin tur finns i Interfacet **IComparable<>**. Därför måste följande tillägg skrivas i huvudet till metoden **G_sort <>()**:

```
where T : IComparable<T>
```

Annars kan vi inte kompilera **if**-villkoret **(t[i].CompareTo(t[i + 1]) > 0)** .

Det enklare alternativet **t[i] > t[i + 1]** som betyder samma sak, fungerar inte heller när vi arbetar med den generaliserade datatypen **T** istället för med **int** eller en annan specifik datatyp.

I generisk programmering kallas konstruktionen **where T : IComparable<T>** en *constraint* dvs en *restriktion* som man lägger på **T**. Just denna constraint innebär att data av typ **T** ska vara jämförbara. Man ska kunna använda jämförelseoperatorerna **>**, **<**, **==** osv. på dem. Interfacet **IComparable<>** innehåller ett antal fördefinierade metoder som implementerar denna möjlighet.

3.10 Listor

Listor är *dynamiska arrays*. Datastrukturen *array* har många fördelar när det gäller hantering av stora datamängder, men också en stor nackdel, nämligen att man i förväg måste ange storleken på arrayen utan att ha möjligheten att ändra den vid behov under programmets gång, s.k. *statisk minnesallokering*, dvs minnesutrymmets storlek bestäms när man definierar arrayen. När koden kompileras reserveras minne av den angivna storleken som inte längre kan ändras under exekveringen. Anta att vi vill ha ett program som läser data, t.ex. laddar ned text, bild eller ljud – från någon källa, säg en fil, och vi vet inte hur mycket data filen innehåller, när vi skriver kod. Därför kan en array inte klara av den här uppgiften. När man läser data från en fil ska minnesallokeringen helst göras samtidigt som filen läses under programmets körning. Man vill helst läsa in data till ett C#-program utan att på förhand behöva ange dess storlek. Lösningen vore *dynamisk minnesallokering*, dvs minnesutrymmet kan utökas efter behov under programmets exekvering. En slags dynamisk array behövs. Och just en sådan dynamisk array är den nya datastrukturen **`List`** som vi ska stifta bekantskap med i detta avsnitt. **`List`** är inte bara dynamisk utan har även en mängd fördefinierade kraftfulla metoder som sorterar, söker i eller på annat sätt manipulerar listor, så att man själv inte behöver koda så mycket. I denna bemärkelse är listor bättre arrays.

Följande program visar ett exempel på denna nya datastruktur:

```
// Lista.cs
// Skapar en lista och skickar den till metoden RandL() där
// den fylls med slumptal. Listan skickas vidare till List-
// metoden Sort() där den sorteras. Utskrift sker före +
// efter sortering.
using System;
using System.Collections.Generic;       // Krävs för List

class Lista                             // OBS! INTE List
{
   static void Main()
   {
      List<int> intList = new List<int>(); // List-objekt
      Random r = new Random();             // av int
      int a = 1, b = 1000;
      Console.WriteLine("\n\t100 heltal mellan " + a +
           " och " + b + " slumpas till ett List-objekt:\n");
      RandList.RandL(r, intList, a, b); // Slumptilldelning
      Print.Out(intList);               // Osorterad utskrift
      intList.Sort();                   // List-sortering
      Console.WriteLine(
           "\tHeltalen sorteras med List-metoden Sort():\n");
      Print.Out(intList);               // Sorterad utskrift
   }
}
```

Klassen List

Klassen **List** är fördefinierad i C#-biblioteket **System.Collectins.Genetric**. För att använda listor måste vi skapa ett objekt av denna klass. Det gör man med satsen:

```
List<int> intList = new List<int>();
```

Variabeln som refererar till det nya objektet kallar vi **intList**. Det speciella med klassen **List** är att den måste kopplas till en datatyp. Här är den kopplad till **int**, dvs klassen heter egentligen **List<int>**. Vi har skapat en lista av **int**, ganska liknande en array av **int**, bara att vi nu inte behöver ange antal element. Det är just det dynamiska i listor till skillnad från arrays. Som en konsekvens får vi tilldela till en lista av **int** också bara heltal av typ **int**. Varje försök att tilldela till den andra än **int**-värden kommer att leda till kompileringsfel. Man kan förstås skapa även objekt av listor av alla andra datatyper inkl. andra klasser. Har man t.ex. definierat en klass **Person** kan man med **List<Person> p = new List<Person>();** skapa en lista över personer. **p** refererar då till ett objekt av typ **List<Person>**. Varje element i denna lista är i sin tur ett objekt av typ **Person**.

Listan **intList** vi skapat ovan är just nu tom. Den blir inte heller tilldelad i koden på förra sidan. För att fylla den med värden skickar vi den som parameter till metoden **RandL()** som vi definierar i klassen **RandList**:

```
// RandList.cs
// Metod RandL() slumpar fram heltal mellan a och b och
// lagrar dem i ett List-objekt med List-metoden Add()
using System;
using System.Collections.Generic;

class RandList
{
   public static void RandL(Random r, List<int> no, int a,
                                                    int b)
   {
      for (int i=0; i < 100; i++)          // Här fylls listan
          no.Add(r.Next(a, b));            // med slumptal
   }
}
```

Deklarationen av parametern i metoden **RandL()**:s parameterlista sker med koden **List<int> no**. Namnet **no** på den formella parametern är oväsentligt. Eftersom referensanrop tillämpas, pekar **no** i alla fall på samma objekt som **intList** dvs den lista som skapades i **Main()**. Så fyller vi den i **for**-satsen med 100 slumptal genererade av den gamla **Rand()**-metod som vi använt tidigare och som i varje varv skapar *ett* slumptal mellan a och b (**1** och **1000**). För att placera dem i listan använder vi oss av metoden **Add()** som är definierad i klassen **List**, därför anropet **no.Add()**. Varje anrop infogar *ett* slumptal i listan. Vi behöver inte ange i förväg hur lång listan ska vara. Den är öppen och växer vid behov. Det är fördelen

med dynamiska arrays som tillhandahålls i klassen **List**. Slumptalsgenereringsmetoden **Next()** anropas i **Add()**-metodens parameterlista med **r.Next(a, b)** som är definierad i biblioteksklassen **Random**.

Vi har även modulariserat utskriftsproceduren med all layout som tillhör den, i metoden **Out()** i den externa klassen **Print** som ser ut så här:

```
// Print.cs
// Metoden Out() skriver ut en lista med en foreach-sats som
// loopar igenom listans ALLA element
using System;
using System.Collections.Generic;

class Print
{
   public static void Out(List<int> t)
   {
      Console.Write("\t");
      int i = 1;
      foreach (int element in t)
      {
         Console.Write(element + " ");
         if (i % 14 == 0)                    // Radbyte var
            Console.Write("\n\t");           // 14:e utskrift
         i++;
      }
      Console.WriteLine("\n");
   }
}
```

I metodens huvud väljs namnet **t** för den formella parametern. Eftersom metodens anrop i **Main()** sker med den aktuella parametern **intList**, pekar **t** på samma lista som **intList**. Därför skrivs ut listans innehåll – de 100 slumptalen – när **Out()** anropas första gången direkt efter att listan blivit tilldelad i **Rand()**-metoden. Andra gången sker anropet efter sorteringen. All utskrift i **Out()** sker med hjälp av en kontrollstruktur som är typisk för listor och arrays och som inleds med det reserverade ordet **foreach**.

foreach-satsen i listor

Det är en kontrollstruktur som behandlades tidigare, fast då var det i samband med array. Nu används **foreach** med listor. Skillnaden är dock obetydlig. I klassen **Print** (ovan) ser huvudet till **foreach**-satsen ut så här:

```
foreach (int element in t)
```

Översatt till svenska:

För varje **element** av listan **t** gör:

Iterationsvariabeln **element** definieras till **int**. Men till skillnad från **for**-satsens räknare är **element** inget index (nr) i listan utan en variabel som pekar på själva värdet (innehållet) som står i listan. **t** är en referens till listan som ska loopas igenom. **foreach**-satsen går igenom listans *alla* element, från det första till det sista. Variabeln **element** som i varje varv pekar på resp. listelementets värde, används sedan i loopens kropp för att göra det man önskar. I vårt exempel sätts den i följande anrop för att skriva ut listans element följt av ett mellanslag:

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

Mellanslaget samt resten av koden i metoden **Out()** är till för att få en snygg layout i utskriften. Räknaren **i** som vi själva definierar, håller reda på loopens varv och ger oss möjligheten att i följande **if**-sats infoga ett radbyte samt tabulator var 14:e utskrift utom i den allra första:

```
if (i % 14 == 0)
   Console.Write("\n\t");
```

Äntligen kan vi testa programmet **Lista** som *kan* resultera i följande utskrift:

```
100 heltal mellan 1 och 1000 slumpas till ett List-objekt:

378 297 220 134 803 115 218 227 346 300 508 559 845 872 417
829 559 105 477 869 602 493 117 713 541 92 572 988 796
982 184 431 259 39 566 724 465 722 14 817 235 751 446
256 650 231 413 914 907 297 464 943 557 957 999 533 181
155 594 359 191 231 79 365 764 725 948 454 307 341 12
485 739 661 635 852 695 862 711 958 680 659 729 147 166
242 522 303 688 681 544 958 129 656 274 652 320 82 493
573

Heltalen sorteras med List-metoden Sort():

12 14 39 79 82 92 105 115 117 129 134 147 155 166 181
184 191 218 220 227 231 231 235 242 256 259 274 297 297
300 303 307 320 341 346 359 365 378 413 417 431 446 454
464 465 477 485 493 493 508 522 533 541 544 557 559 559
566 572 573 594 602 635 650 652 656 659 661 680 681 688
695 711 713 722 724 725 729 739 751 764 796 803 817 829
845 852 862 869 872 907 914 943 948 957 958 958 982 988
999
```

”*Kan* resultera”, därför att det blir andra siffror i varje körning pga at det är slumptal som genereras och som är olika varje gång man kör programmet. Sorteringen görs i programmet **Lista**:s anrop (sid 129) av metoden **Sort()** som är fördefinierad i klassen **List**.

Övningar till kap 3

3.1 Modifiera klassen **Fish** (sid 107) så här: Deklarera datamedlemmarna som **private** och metoderna som **public**. Förse klassen med ytterligare två publika metoder, så att den nya klassen **Fish_priv** har följande utseende:

```
using System;
class Fish_priv
{
   private string sort;
   private float weight, size;

   public Fish_priv(string S, float w, float s)
   {
      sort   = S;
      weight = w;
      size   = s;
   }

   public int Price()
   {
      return (int) Math.Round(weight * 7.25f / 100);
   }

   public int Shipping()
   {
      return (int) Math.Round(weight * 0.02f + size * 0.1f);
   }

   public string AsString()
   {
      return sort    + "\t "     +
             weight  + "\t\t   " + size       + "\t\t " +
             Price() + "\t "     + Shipping() + "\n"    ;
   }
}
```

Modifiera programmet **ArrayOfRef** (sid 108) så att det modifierade programmet gör samma sak som det ursprungliga.

3.2 Skriv ett program som läser in 10 heltal från konsolen, lagrar dem i en array och skriver ut dem i omvänd ordning.

3.3 Skriv ett program som läser in text i gemener, lagrar den i en array av **char** och skriver ut den framhävd i versaler och med mellanslag mellan varje tecken.

3.4 Skriv ett program som frågar efter användarens för- *och* efternamn, hälsar sedan användaren i en utskrift med fullständiga namnet, förnamnets längd samt efternamnets första och sista bokstav. Lös uppgiften generellt utan att använda information om något speciellt för- och efternamn.

3.5 Skriv ett program där **`Main()`** läser in en persons fullständiga namn och hälsar tillbaka med namnets initialer. Dessa ska bestämmas och skrivas ut i en annan metod – med huvudet **`static void Initials(char[] name)`** – som anropas i **`Main()`**.

3.6 Skriv ett program som slumpar fram 1000 heltal mellan 60 och 140 (tänkbara hastigheter på en motorväg), lagrar dem i en array kallad **`hastighet`**, beräknar och skriver ut deras medelvärde med förklarande text. Använd klassen **`RandArray`** (sid 116) som extern modul.

3.7 Modifiera programmet **`Lista`** (sid 129) så att sorteringen av slumptalen görs med vår egen bubbelsorteringsmetod **`sort()`** (sid 122) istället för med den fördefinierade **`List`**-metoden **`Sort()`**. Testa först med array-notationen som **`sort()`** är skriven i. Försök sedan att skriva om **`sort()`** till en **`List`**-version.

Kapitel 4

Tillämpningar

Ämne	Sida	Program
4.1 Kryptering av strängar	136	**EncryptStr**
4.2 Kryptering av text, teckenvis	139	**EncryptChar**
4.3 Filhantering	142	**WriteReadFile**
- Append	145	**AppendFile**
4.4 Slumplösenord	147	**RandPasswdTest**
4.5 Kryptering av filer	151	**EncryptFile**
Övningar till kapitel 4	156	

4.1 Kryptering av strängar

I C# är det inte själva objekten som skickas och fås tillbaka utan snarare deras referenser, när man har dem i metoder. Det vore slöseri med datorns resurser (minnesutrymme) om man kommunicerade tunga objekt istället för lätthanterade referenser till objekt. Så, det är inget nytt utan snarare det normala att använda referenser som företrädare för objekt. I metoden **`Rand(Random s, int a, int b)`** har vi redan använt objektreferenser som parametrar, där **`s`** är en referens till ett objekt av klassen **`Random`**. Samma sak kan man göra med returvärden.

Referens som parameter och returvärde

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

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

class EncryptStr
{
   public static String Encrypt(String t, int n)
   {
      char ch;
      String temp = "";            // Tom sträng

      for (int i=0; i <= t.Length - 1; i++)
      {
          ch = t[i];               // Tar tecknen från t
          ch = (char) (ch + n);    // Ändrar tecknen
          temp += ch;              // Lägger tecknen i temp
      }

      return temp;                 // Skriver till Encrypt
   }
}
```

Med den första parametern **`t`** får metoden **`Encrypt()`** tillgång till det **`String`**-objekt som skapas i den anropande metoden **`Main()`**. Adressen till detta objekt kopieras över till referensvariabeln **`t`** när **`Encrypt()`** anropas. Samma sak sker med krypteringsnyckeln vars värde kopieras till den andra parametern **`n`**. Sedan har vi i kroppen av metoden två lokala variabler **`ch`** och **`temp`**. Den första som är av typ

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

```
// EncryptStrTest.cs
// Skickar i ett första anrop strängen text samt en slumpad
// krypteringsnyckel till metoden Encrypt() och anropar den
// en andra gång med den krypterade texten och inverterad
// (negativ) krypteringsnyckel för att återställa strängen
using System;
class EncryptStrTest
{
   static void Main()
   {
      String text = "abcdefghijklmnopqrstuvwxyz";
      Random r = new Random();
      int key = r.Next(50, 200);              // Krypterings-
                                              // nyckeln
      Console.WriteLine("\n\tKryptering av text:   ");
      Console.Write("\n\tOkrypterad text:    " + text);

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

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

Ett körresultat visar följande utskrift:

```
Kryptering av text:

Okrypterad text:    abcdefghijklmnopqrstuvwxyz

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

Krypteringsnyckeln: 68

Återställd text:    abcdefghijklmnopqrstuvwxyz
```

Det engelska alfabet som använts som teststräng har krypterats med slumpnyckeln **68** och återställts med **-68**. Båda operationer utförs i programmet ovan med anrop av metoden **Encrypt()**, definierad i klassen **EncryptStr** (sid 136). Det första anropet sker med den **key** som anropet **r.Next(50, 200)** genererar, dvs ett heltalsslumpvärde mellan **50** och **200**.

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

4.2 Kryptering av text, teckenvis

Vi ska nu dra lite praktisk nytta av våra samlade kunskaper om bl.a. slumptal, ASCII-koder, array, stränghantering, metoder och referensanrop, för att med ganska enkla medel skriva en liten applikation om kryptering av text. Egentligen har vi redan skrivit en sådan, nämligen klassen **`EncryptStr`** med **`return`**-metoden **`Encrypt()`**. Men då löstes problemet med bibliotekslkassen **`String`**. Nu ska vi göra det med en egen array av **`char`** och en **`void`**-metod istället. Följande program läser in text som en **`char`**-array, skickar den till **`void`**-metoden **`Encrypt()`** där den krypteras resp. återställs teckenvis med ett slumptal som krypteringsnyckel. Tekniken som används för kryptering är samma som i **`EncryptStr`**-metoden, fast ännu enklare i och med man arbetar på **`char`**-nivå. Ett **`String`**-objekt kan inte manipuleras på **`char`**-nivå. Nu behöver strängen själv inte kopieras till en annan plats utan kan pga referensanrop krypteras på samma ställe, varför **`char`**-programmet behöver hälften av det minnesutrymme som det gamla **`String`**-programmet behövde.

```
// EncryptCharTest.cs
// Läser in text som en char-array och skickar den med en
// krypteringsnyckel till metoden Encrypt() där den krypteras
// Referensanrop gör den krypterade texten tillgänglig i
// Main(). Encrypt() anropas en andra gång med den krypterade
// texten och en inverterad (negativ) krypteringsnyckel för
// att återställa den.
using System;
class EncryptCharTest
{
   static void Main()
   {
      Random r = new Random();
      int key = r.Next(50, 200);            // Slump-krypte-
                                            // ringsnyckeln
      Console.Write("\nSkriv text som ska krypteras:\t");
      char[] text = Console.ReadLine().ToCharArray();
      Console.Write("\n\tOkrypterad text:\t");
      Output(text);

      EncryptChar.Encrypt(text, key);       // 1:a anropet
                                            // krypterar
      Console.Write("\n\n\tKrypterad text:\t\t");
      Output(text);                         // text är ändrad

      EncryptChar.Encrypt(text, -key);      // 2:a anropet
                                            // återställer
      Console.Write("\n\n\tÅterställd text:\t");
      Output(text);                         // text är ändrad
      Console.WriteLine("\n\nKrypteringsnyckeln:\t\t" +
                                               key + '\n');
   }
```

```
   static void Output(char[] a)                    // Metod som
   {                                               // skriver ut
      foreach (char element in a)                  // en array
         Console.Write(element);
   }
}
```

Med en array av **char** allokeras minne för texten med en maximal längd som är föreskriven av metoden **Console.ReadLine()**, något antal tecken som ryms på en rad, kanske **80** eller lite fler. Sedan överförs parametern **text** med ett första anrop av metoden **Encrypt()**:

```
EncryptChar.Encrypt(text, key);
```

som är definierad i klassen **EncryptChar** (se nedan), till metoden **Encrypt()**. I detta anrop används automatiskt referensanrop eftersom **text** är definierad som array. Därför är ändringarna som görs med **text** i metoden **Encrypt()**, tillgängliga efter anropet. Texten är okrypterad före och krypterad efter anropet både i **Encrypt()** och i **Main()**. Den andra parametern **key** däremot överförs med vanligt värdeanrop – dvs med kopiering av värdena – eftersom denna parameter är definierad till den enkla datatypen **int**. Efter **Encrypt()**:s första anrop skrivs den krypterade texten ut. Sedan anropas **Encrypt()** andra gången med **-key**, det negativa värdet av **key**, för att återställa texten som sedan skrivs ut för kontroll. Hur krypteringsmetoden fungerar, förstår man bäst om man samtidigt tittar på metoden **Encrypt()**:

```
// EncryptChar.cs
// Tar emot en text via arrayen t och krypterar den genom att
// förskjuta alla tecken med n steg i teckentabellen
// Kontrollerar textens slut med arrayegenskapen Length

class EncryptChar
{
   public static void Encrypt(char[] t, int n)
   {
      for (int i = 0; i < t.Length; i++)
          t[i] = (char) (t[i] + n);
   }
}
```

Krypteringsmetoden är väldigt enkel: tecknens ASCII-värden ökas med **n** i satsen **t[i] = (char) (t[i] + n);** genom vanlig addition. Att det verkligen adderas **n** till *ASCII-koden* till **t[i]** beror på att **t[i]** är av typ **char** och att en teckenvariabel i aritmetiska uttryck tolkas som sin ASCII-kod – ett tal man kan räkna med. **for**-satsen som går igenom hela strängen genom att koppla loopens räknare till arrayens index, gör att hela texten förskjuts med **n** steg i ASCII-tabellen. **n** får sitt värde genom kopiering (värdeanrop) från **key** vid första och från **-key** vid

andra anropet. **key**:s värde i sin tur slumpas fram i **Main()** med hjälp av **Random**-metoden **Next()**. Dess anrop med parametrarna **1** och **501** gör att vi får ett slumpvärde som är ett heltal mellan **1** och **500** som sedan skickas som krypteringsnyckel till **Encrypt()** via dess andra parameter. Vid andra anropet av **Encrypt()** skickas **-key** för att återställa texten. Genom att ersätta **t[i] + n** med mer sofistikerade formler kan man utveckla mer avancerade krypteringsalgoritmer.

Programmet **EncryptCharTest** kan köras på olika sätt. Varje körning ger en annan slumpmässig krypteringsnyckel. Här ett exempel på en körning:

```
Skriv text som ska krypteras:   abcdef

        Okrypterad text:        abcdef

        Krypterad text:         åæçèéê

        Återställd text:        abcdef

Krypteringsnyckeln:             132
```

Man kan kontrollera krypteringen för hand: Man ser att bokstaven **a** förskjutits till **å**. Krypteringsnyckeln har vid denna körning varit **132**. ASCII-koden till **a** som är 97, har förskjutits **132** steg vidare till 97 + 132 = 229 som är koden till tecknet **å**. Därför har **a** förskjutits till **å** med krypteringsnyckeln **132**. På samma sätt görs det med de andra tecknen i texten **abcdef**.

Självklart borde i en skarp applikation krypteringsnyckeln inte skrivas ut utan endast sparas i variabeln **key** för att använda den vid återställningen. Vi gör det här endast för experimentens skul.

Lägger man till filhantering i programmet **EncryptCharTest** kan samma metod **Encrypt()** användas för kryptering av filer.

4.3 Filhantering

Alla våra program hittills har haft en sak gemensam: Så snart vi avlutat programkörningen har all data försvunnit från datorn utom programmets källkod som vi sparat på hårddisken. Vi har efter exekveringen inte kunnat komma åt varken programmets in- eller output. Anledningen är att, när vi startar körningen, laddas både källkoden och programmets variabler samt in- och utdata till datorns primärminne RAM. När körningen är avslutad ”dör” all data i RAM. Ska utdata användas efteråt måste den under körningen skickas till och sparas i filer. Samma sak gäller för indata: När dess mängd är så stor att den inte kan matas in från tangentbordet, måste den läsas in från filer. På så sätt kan filhantering bli en nödvändighet.

```
// WriteReadFile.cs
// Skapar filen  WriteRead.txt eller öppnar den om den finns.
// Raderar gammalt innehåll om filen redan finns.
// Skriver en text från programmet till filen, läser den
// sedan från samma fil och skriver ut den på skärmen.
using System;
using System.IO;                    // Krävs för StreamWriter
                                    //           StreamReader
class WriteReadFile
{
  static void Main()
  {
   string word;
   StreamWriter fileForWrite = new StreamWriter
        ("WriteRead.txt");  // Objekt av klassen StreamWriter

   fileForWrite.WriteLine("\n\t\tDenna text är innehållet" +
     " i filen WriteRead.txt."); // Skriver texten till filen
   fileForWrite.Close();     // Skriver över gammalt innehåll

   StreamReader fileForRead = new StreamReader
        ("WriteRead.txt");  // Objekt av klassen StreamReader

   Console.WriteLine("\n\tFöljande text har skrivits "  +
                  " från programmet till filen.\n\n\t"  +
                           "Nu läses den från filen:\n");

   while (!fileForRead.EndOfStream)   // Så länge filsluts-
   {                                  // tecknet inte är nått
      word = fileForRead.ReadLine();  // ska en sträng läsas
                                      // från fileForRead och
      Console.WriteLine(word);        // tilldelas word
   }
   fileForRead.Close();
   Console.WriteLine('\n');
   }
}
```

Programmet **WriteReadFile** ovan skapar en fil eller öppnar den, om den redan finns i projektmappen (t.ex. *C:\C#\MyCsConsoleProj\bin\Debug*), skriver en text i den och läser sedan texten från filen samt visar innehållet. Programmet inleds bl.a. med följande **using**-direktiv som behövs för att kunna använda filhanteringsklasser:

```
using System.IO;
```

IO står för **I**nput/**O**utput. De filhanteringsklasser från C# biblioteket som används i programmet är **StreamWriter** för skrivning till filer och **StreamReader** för läsning från filer. De innehåller metoder för skrivning till filer från ett C# program (output) och inläsning från filer till ett C# program (input). Att det första heter out- och det andra input beror på att man ser på det från C# programmets synvinkel. Då innebär skrivning till fil *output* därför att data går från programmet till fil, medan inläsning från fil innebär *input* därför att data går från fil till programmet. Utgångspunkten är alltid C# programmet, inte filen.

Att skriva till en fil

I programmet **WriteReadFile** definieras referensen **fileForWrite** i satsen:

```
StreamWriter fileForWrite = new StreamWriter("WriteRead.txt");
```

Det är en kraftfull sats. Vi ska gå igenom vad den gör: Variabeln **fileForWrite** definieras som en referens till det nya objektet av klassen **StreamWriter**. Man kan endast skriva ut data från programmet till en sådan fil, inte omvänt. Utgångspunkten för att bestämma ”riktningen” av *out*- och *in*put är som sagt alltid C# programmet: *out*put innebär *ut*data *från* programmet till en fil, medan *in*put innebär *in*data från en fil *till* programmet.

Men vad gör parentesen **("WriteRead.txt")**? Den anropar konstruktorn till klassen **StreamWriter**. Observera att konstruktorns parameter tar emot en sträng varför filnamnet måste skrivas inom citationstecken. Samtidigt som referensvariabeln **fileForWrite** definieras och tilldelas det nya **StreamWriter**-objektet, initierar konstruktorn objektet till **"WriteRead.txt"**: Ett *logiskt*, dvs programmeringstekniskt filnamn **fileForWrite** skapas och kopplas till det *fysiska* filnamnet **WriteRead.txt**, en fil som antingen redan finns eller skapas på hårddisken. Det som kompilatorn gör är att söka i projektmappens undermapp **C:\C#\MyCsConsoleProj\bin\Debug** efter en fil med detta namn. Om den finns där kommer satsen ovan att radera filens innehåll utan förvarning när programmet **WriteReadFile** exekveras. Samtidigt sätts filens markör i början av den tomma filen, redo för att skriva i den. Om filen inte finns (i den nämnda mappen) kommer satsen att skapa en fil med namnet **WriteRead.txt**, sätta markören i början av filen, redo för att skriva i den.

Referensvariabeln **fileForWrite** används sedan för att skriva till filen med:

```
fileForWrite.WriteLine(
     "\n\t\tDenna text är innehållet i filen WriteRead.txt.");
```

För första gången används här metoden **`WriteLine()`** inte efter **`Console.`** dvs inte för att skriva ut till konsolen, utan efter filvariabeln **`fileForWrite`**. Istället för att skriva ut till konsolen skriver den ut till den fil som **`fileForWrite`** pekar på, dvs till den fysiska filen **`WriteRead.txt`**.

Slutligen stängs filen med **`fileForWrite.Close();`** Metoden **`close()`** är definierad i den klass som **`fileForWrite`** refererar till dvs i klassen **`StreamWriter`**. Den explicita stängningen av filen är av betydelse då den sätter *filslutstecknet* som är avgörande för filens korrekta återanvändning. När man t.ex. senare vill läsa från filen används ofta en loop vars avslutningskriterium är just detta filslutstecken som representeras på olika sätt i olika operativsystem, t.ex. *ctrl-z* i Windows och *ctrl-d* i Unix. I C# tar **`EndOfStream`** reda på om filslutstecknet är nått eller ej. Vi kommer att använda oss av **`EndOfStream`** när vi läser från filen.

Att läsa från en fil

I programmet **`WriteReadFile`** definieras referensen **`fileForRead`** i satsen:

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

Ett nytt objekt skapas av klassen **`StreamReader`**, för input. Variabeln **`fileForRead`** definieras som en referens till det nya objektet. Input innebär att man med det här objektet endast kan läsa data från filen till programmet, inte omvänt. Samtidigt initieras objektet till filen **`"WriteRead.txt"`** – samma fil som vi skrev till i programmets första del. Markören sätts i början av filen, redo för att läsa från den. Operationerna för inläsning är definierade i klassen **`StreamReader`** medan de för skrivning finns i **`StreamWriter`**.

Sedan används **`while`** för att läsa från filen **`WriteRead.txt`** och skriva det lästa till skärmen:

```
while (!fileForRead.EndOfStream)
{
   word = fileForRead.ReadLine();
   Console.WriteLine(word);
}
```

`EndOfStream` är en datamedlem av typ **`bool`** i klassen **`StreamReader`** och får sanningsvärdet **`true`** när filslutstecknet påträffas, annars **`false`**. Så länge filslutstecknet *inte* är nått, ska **`while`**-loopen fortsätta. När det är nått ska den avslutas. Den logiska operatorn NEGATION **`!`** kan sättas framför **`EndOfStream`** eftersom det är av typ **`bool`**. Så länge **`EndOfStream`** är **`false`** ska **`while`**-loopen leda dataströmmen från filen **`fileForRead`** till strängvariabeln **`word`**. Detta är innebörden i satsen **`word = fileForRead.ReadLine();`** Programmet läser data från filen sträng för sträng, där mellanslag mellan strängarna i filen tolkas som avskiljare. Innan **`while`**-loopens nästa varv läser nästa sträng från filen och skriver över variabeln **`word`**:s värde skickas den aktuella strängen till skärmen.

Efter läsning ska filen stängas på korrekt sätt med satsen **`fileForRead.close();`** även om den inte återanvänds i detta program. En körning av **`WriteReadFile`** ger följande utskrift:

```
Följande text har skrivits från programmet till filen.

Nu läses den från filen:

     Denna text är innehållet i filen WriteRead.txt.
```

Sedan kan man kolla att utskriftens tredje rad även finns i filen **`WriteRead.txt`**. Det gör man genom att gå till mappen **`C:\C#\MyConsoleProj\bin\Debug`** och **`WriteRead.txt`** öppna filen som finns där.

Append

Programmet **`WriteReadFile`** innehåller i den del som skriver till filen, följande sats:

```
StreamWriter fileForWrite = new StreamWriter("WriteRead.txt");
```

Om filen **`WriteRead.txt`** redan finns i mappen **`C:\C#\MyConsoleProj\bin\Debug`** raderar satsen ovan filens innehåll utan förvarning varje gång programmet exekveras. Vill man inte ha det så, utan önskar att filens gamla innehåll bibehålls och det nya kommer till som ett tillägg, kan man med följande ändring åstadkomma detta:

```
StreamWriter fileForWrite = new StreamWriter("WriteRead.txt",
                                              append:true);
```

Ändringen, dvs tillägget av 2:a parametern **`append:true`** i konstruktorns parameterlista gör att filen **`WriteRead.txt`** öppnas i s.k. *append mode* vilket innebär att man kan lägga till data i filen utan att radera befintlig data.

Den syntax som används för konstruktorns 2:a parameter är ny för oss:

```
append:true
```

Parametern **`append`** är av typ **`bool`**. Dess värde i anropet ovan sätts till **`true`**. Detta ändrar helt och hållet filskrivningens beteende: Markören sätts inte i början utan i slutet av filen. Filens gamla innehåll överskrivs inte utan sparas. Märkören lägger till ny text till den gamla. Filen växer. Detta beteende kan man testa i följande program:

```
// AppendFile.cs
// Öppnar filen WriteRead.txt som skapades i programmet
// WriteReadFile utan att radera filens gamla innehåll.
// Lägger till text från programmet till filen, läser sedan
// hela innehållet från samma fil och skriver ut det.
using System;
using System.IO;              // Krävs för StreamWriter och
class AppendFile              //          StreamReader
{
  static void Main()
  {
    String word;
    StreamWriter fileForWrite = new StreamWriter
                          ("WriteRead.txt", append:true);
                          // Objekt av klassen StreamWriter:
                          // Bibehåller filens gamla innehåll
    fileForWrite.WriteLine("\n\t\tDenna text har lagts till"
                                + " filen WriteRead.txt.");
                          // Lägger till ny text till filen
    fileForWrite.Close();

    StreamReader fileForRead = new StreamReader
                                          ("WriteRead.txt");
    Console.WriteLine("\n\tFöljande text har skrivits " +
                  " från programmet till filen.\n\n\t" +
                           "Nu läses den från filen:\n");
    while (!fileForRead.EndOfStream)
    {
       word = fileForRead.ReadLine();
       Console.WriteLine(word);
    }
    fileForRead.Close();
    Console.WriteLine('\n');
  }
}
```

Resultatet är följande:

```
    Följande text har skrivits från programmet till filen.

    Nu läses den från filen:

         Denna text är innehållet i filen WriteRead.txt.

         Denna text har lagts till filen WriteRead.txt.
```

Den sista raden har kommit till i och med exekveringen av programmet **AppendFile** medan de tre första raderna härstammer från programmet **WriteReadFile**.

4.4 Slumplösenord

Det är var och en systemadministratörs önskemål att snabbt kunna få en lista över ett antal användarnamn samt slumpvis genererade lösenord som följer en viss policy, för att dela ut dem till sina användare. När lösenorden är slumpade är det praktiskt taget omöjligt att hantera dem utan att spara dem i en fil. Detta för att effektivt kunna administrera och dela ut konton med användarnamn och lösenord till alla användare. För att kunna göra det behövs en lösenordspolicy som ingår t.ex. i följande problemställning:

Problemet:

> ”Skriv ett program som skriver till en fil med två kolumner. I den första ska stå några användarnamn av typ **user1**, **user2**, … . I den andra ska till varje användare stå ett slumpvis genererat lösenord med 8 tecken, nämligen 3 små bokstäver, 2 siffror och 3 stora bokstäver. Programmet ska sedan visa filens innehåll.”

Lösningen:

```
// RandPasswdTest.cs
// Skapar en fil, skriver i den ett antal användarnamn och
// slumpvis genererade lösenord med metoden RandPasswd()
// Läser sedan från samma fil och skriver ut innehållet
using System;
using System.IO;

class RandPasswdTest
{
  static void Main()
  {
   char[] password = new char[8];
   Random r = new Random();
   string word;
   Console.Write("\n\tHur många användarnamn med lösenord "
                                        + "vill du ha?    ");
   int number = Convert.ToInt32(Console.ReadLine());
   StreamWriter fileForWrite = new StreamWriter
                                          ("userPasswd.txt");
   for (int i=1; i <= number; i++)
   {
     RandPasswd.OnePassword(r, password);  // Slumplösenord
     fileForWrite.WriteLine("\tuser" + i + // Skrivs till fil
                "\t\t" + new String(password));
   }
   fileForWrite.Close();
```

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

   Console.WriteLine("\n\tVarsågod, detta står nu" +
                     " i filen userPasswd.txt:\n");
   while (!fileForRead.EndOfStream)
   {
      word = fileForRead.ReadLine();      // Läses från fil
      Console.WriteLine(word);            // Skrivs till skärm
   }
   fileForRead.Close();
   Console.WriteLine();
  }
}
```

Lösningen består av programmet ovan samt klassen **RandPasswd** på nästa sidan. I den första med vit bakgrund framhävda raden kopplar programmet **RandPasswdTest** filvariabeln **fileForWrite** till den fysiska filen **"userPasswd.txt"**. Så alla inloggningsuppgifter kommer att hamna i denna fil. I den andra med vit bakgrund framhävda raden anropas metoden **EttLösenord()** från klassen **RandPasswd**, vilket genererar ett slumplösenord. Sedan skrivs till filen med följande sats:

```
fileForWrite.WriteLine("\tuser" + i       +
                       "\t\t" + new String(password));
```

Både anropet och denna sats är inbyggda i en **for**-loop där räknaren **i** går från **1** till **number** användare man matar in vid körning. Intressant ur en programmeringsteknisk synpunkt är nu den i satsen ovan med grå bakgrund framhävda koden. Frågan är: Varför kan man inte bara enkelt skriva **password** i koden för att få ut strängen som representerar slumplösenordet som genererats av metoden **OnePassword()**? Anledningen är att **password** endast är en referens till en **char**-array och inte en sträng, ja inte ens själva arrayen. Detta kan man se när man tittar på den sats som i början av programmet definierar **password**:

```
char[] password = new char[8];
```

För att få ut den **char**-array som **password** refererar till, som en sträng, måste vi skapa ett strängobjekt med samma referens som pekar på **char**-arrayen. Annars – om vi bara skriver **password** – får vi endast ut referensen. Därför: **new String(password)**. Testa gärna för att se vad du får i utskriften.

I den tredje med vit bakgrund framhävda raden kopplar programmet **RandPasswdTest** filvariabeln **fileForRead** till en filtyp för input och initieras till samma fil som vi skrev till. För läsning från filen till skärmen används samma **while**-loop som i programmet **WriteReadFile** (sid 142).

Det ursprungliga målet var ju att skriva en lista över användarnamn och lösenord till filen **userPasswd.txt** för att dela ut konton. Lösenorden kan initialt vara vad

som helst, bara de följer en policy med vissa säkerhetskrav. Sedan kan användarna efter den första inloggningen själva bestämma sina individuella lösenord. Följande metod som i programmet **RandPasswdTest** anropas i samma **for**-sats som skriver till filen, löser problemet. Direkt efter anropet sparas användarnamn och lösenord i filen.

```
// RandPasswd.cs
// Genererar ETT slumplösenord med policyn:
// 8 tecken = 3 små bokstäver: ASCII-intervall (97, 122) +
//            2 siffror                        (48,  57) +
//            3 stora bokstäver                (65,  90)
using System;

class RandPasswd
{
   public static void OnePassword(Random r, char[] p)
   {
      for (int i=0; i < 3; i++)
          p[i] = (char) r.Next(97, (122 + 1));   // 3 små
                                                 // bokstäver
      for (int i=3; i < 5; i++)
          p[i] = (char) r.Next(48, (57 + 1));    // 2 siffror

      for (int i=5; i < 8; i++)
          p[i] = (char) r.Next(65, (90 + 1));    // 3 stora
   }                                             // bokstäver
}
```

Metoden tar emot en array av **char** och tilldelar dess 3 första element – vars index är **0**, **1**, och **2** – tecken som slumpvis tas ur ASCII-intervallet **(97, 122)**. En blick i ASCII-tabellen *(Progr1, 3.3)* visar att det är tecknen **a, b, c, ..., z** dvs det engelska alfabetet i gemener. Den här tilldelningen kan göras med en **for**-sats eftersom det engelska alfabetet finns sammanhängande i ASCII-tabellen.

Det 4:e och 5:e elementet – med index **3** och **4** – tilldelas slumpvis ett tecken ur ASCII-intervallet **(48, 57)**. Enligt ASCII-tabellen är det siffrorna **0-9**.

De 3 sista elementen, dvs element nr 6, 7 och 8 – med index **5**, **6** och **7** – tilldelas något av tecknen i ASCII-intervallet **(65, 90)**. Det är tecknen **A, B, C, ..., Z** dvs det engelska alfabetet i versaler.

I alla intervall ingår även gränserna därför att metoden **Next()** som anropas här flera gånger, även inkluderar intervallgränserna vid slumptalsgenereringen.

Metoden **OnePassword()** anropas i programmet **RandPasswdTest** i den **for**-sats som skriver till filen. Därvid skickas två parametrar. Den första är referensen **r** till **Random**-objektet som skapas i början av programmet. Det är nödvändigt för att metoden **OnePassword()** ska kunna anropa **Random**-metoden **Next()** som skapar slumptal. Den andra parametern är **password** som pekar på en **char**-array av

längden 8. I metoden tas den emot av referensen **p** av samma typ och initieras där. Efter anropet är arrayen även initierad i **Main()** pga referensanrop. Så hamnar innehållet – ett slumplösenord av 3 gemener, 2 siffror och 3 vesaler – i filen.
Ett körresultat av programmet **RandPasswdTest** kan se ut så här:

```
Hur många användarnamn med lösenord vill du ha?    20

Varsågod, detta står nu i filen userPasswd.txt:

user1           oya00GDB
user2           vjb54XVL
user3           zae83HHS
user4           jdl84YLE
user5           tja91QGS
user6           noe52ZGC
user7           jqs54CDG
user8           qhs88HQX
user9           ywt18WIJ
user10          uli71UMJ
user11          wim72WSR
user12          guj89KXG
user13          ygp32DFN
user14          hjv07KCV
user15          viz47VSC
user16          ecx04MLK
user17          nbv82CET
user18          czn80QXV
user19          rna53KMC
user20          onf12DAU
```

Samtidigt skapas filen **userPasswd.txt** på hårddisken i projektmappens undermapp **C:\C#\MyProject\bin\Debug** med ovanstående listan över 20 användarnamn och lösenord som innehåll. Vill man placera filen på en annan plats på hårddisken, måste i den sats som skapar filen, sökvägen till denna plats anges:

```
StreamWriter fileForWrite =
                new StreamWriter("C:\\ ... \\userPasswd.txt");
```

Sökvägen **...** måste börja med diskens enhetsbokstav om man väljer absoluta sökvägar. Men även relativa sökvägar av typ **..\\userPasswd.txt** är möjliga som placerar filen t.ex. i mappen strax ovanför den aktuella mappen. Självklart borde samma sökväg anges senare i programmet i den sats som läser filen. Anledningen till användningen av **** i sökvägen är att **** är reserverad för escapesekvensernas inledningssymbol. För själva tecknet **** inom en sträng måste escapesekvensen **** användas *(Progr1, 3.4)*.

4.5 Kryptering av filer

Tidigare behandlades kryptering av text (sid 136 och 139). De verktyg som utvecklades där kan med fördel användas för att kryptera även filer, nu när vi lärt oss att hantera filer. För avväxlingens skull presenterar vi först körresultatet av ett filkrypteringsprogram som vi sedan tar upp och och går igenom koden:

```
        Okrypterad fil:

This text is coming from a file called OriginalText.txt.
The C# program EncryptFile reads it from the hard disk, encrypts
the content and writes the encrypted text to the file Encrypted.txt.
In order to test the encryption, the program decrypts the text
and writes the recovered text to the file Recovered.txt.
At the same time the content of the files are displayed.

        Krypterad fil:

¢¶·ÁnÂ³ÆÂn·Án±½»·¼µn´À½»n¯n´·°³n±¯°°³²n?À·µ·¼¯°¢³ÆÂ|ÂÆÂ|n[X¢¶³n?qn¾À½
µÀ¯»n?¼±ÀÇ¾Â?·°³nÀ³¯²Án·Ân´À½»nÂ¶³n¶¯À²n²·Á¹zn³¼±ÀÇ¾ÂÁn[XÂ¶³n±½¼Â³¼Ân
¯¼²nÅÀ·Â³ÁnÂ¶³n³¼±ÀÇ¾Â³²nÂ³ÆÂnÂ½nÂ¶³n´·°³n?¼±ÀÇ¾Â³²|ÂÆÂ|n[X?¼n½À²³ÀnÂ
½nÂ³ÁÂnÂ¶³n³¼±ÀÇ¾Â·½¼znÂ¶³n¾À½µÀ¯»n²³±ÀÇ¾ÂÁnÂ¶³nÂ³ÆÂn[X¯¼²nÅÀ·Â³ÁnÂ¶³
nÀ³±½Ä³À³²nÂ³ÆÂnÂ½nÂ¶³n´·°³n ³±½Â³À³²|ÂÆÂ|n[X?ÂnÂ¶³nÁ¯»³nÂ·»³nÂ¶³n±½¼
Â³¼Ân½´nÂ¶³n´·°³Án¯À³n²·Á¾°¯Ç³²|[X

        Återställd fil:

This text is coming from a file called OriginalText.txt.
The C# program EncryptFile reads it from the hard disk, encrypts
the content and writes the encrypted text to the file Encrypted.txt.
In order to test the encryption, the program decrypts the text
and writes the recovered text to the file Recovered.txt.
At the same time the content of the files are displayed.

        Krypteringsnyckeln:     78
```

Det här är bara ett av flera möjliga körresultat man kan få när man kör programmet **EncryptFile** (sid 152), därför att krypteringsnyckeln slumpas fram och kan därför vara olika vid varje körning. Just här vid den aktuella körningen är den **78**. Samma teknik användes när vi krypterade text. Skillnaden är att vi nu använder *filer* som källa och mål för texten som ska krypteras. Programmet **EncryptFile** som genererar utskriften ovan och visas på nästa sida, slumpar först fram ett heltal som används som krypteringsnyckel – vi kallar det i fortsättningen kort slumpnyckel.

Programmet **EncryptFile** som visas nedan, är i högsta grad modulariserat och består av följande klasser:

EncryptFile	innehåller	**Main()** som anropar alla andra metoder
EncryptText		metoden **Encrypt()** som krypterar text
WriteFile		metoden **Write()** som skriver text till en fil
ReadShowFile		metoden **ReadShow()** som läser en fils innehåll och visar det på skärmen

```
// EncryptFile.cs
// Läser text från en fil, krypterar den med en slumpnyckel,
// skriver den krypterade texten till en annan fil och visar
// den. Dekrypterar sedan texten och skriver den till en 3:e
// fil samt visar både den återställda filen och slumpnyckeln
// Slumpnyckeln ger vid varje körning en annan kryptering
using System;
using System.IO;

class EncryptFile
{
  static void Main()
  {
    Console.WriteLine("\n\tOkrypterad fil:\n");
    string fileText = ReadShowFile.ReadShow
                                    ("OriginalText.txt");
    Random r = new Random();
    int key = r.Next(50, 251);               // Slumpnyckeln

    fileText = EncryptText.Encrypt(fileText, key);// Krypte-
                                          // rar med slumpnyckel
    WriteFile.Write(fileText, "Encrypted.txt");  // Skriver
                                             // till filen Enc…
    Console.WriteLine("\tKrypterad fil:\n");
    fileText = ReadShowFile.ReadShow("Encrypted.txt");

    fileText = EncryptText.Encrypt(fileText, -key);
                      // Dekrypterar med negativ slumpnyckel
    WriteFile.Write(fileText, "Recovered.txt");  // Skriver
                                             // till filen Rec…
    Console.WriteLine("\tÅterställd fil:\n");
    fileText = ReadShowFile.ReadShow("Recovered.txt");

    Console.WriteLine("\tKrypteringsnyckeln:\t" + key +
                                                    "\n");
   }
}
```

I **`Main()`** finns endast variabeldefinitioner och anrop av de externlagrade metoder vilka gör det egentliga jobbet, nämligen slumptalsgenereringen, filläsningen, filvisningen, krypteringen och filskrivningen. I början av **`Main()`** skapas **`string`**-objektet **`fileText`** för att lagra filens innehåll. Det kan inte förutsägas hur stor filen är som ska krypteras, men det spelar ingen roll. Vi har förbrett en liten textfil, döpt den till **`OriginalText.txt`** och lagt den i projektmappens undermapp **`C:\C#\-MyProject\bin\Debug`**. I följande sats anropas metoden **`ReadShow()`** som är definierad i den separata klassen **`ReadShowFile`**:

```
string fileText = ReadShowFile.ReadShow("OriginalText.txt");
```

Anropet läser filen **OriginalText.txt** och returnerar innehållet till **string**-objektet **fileText**. Sedan låter vi metoden **Next()** – från bibliotheksklassen **Random** – generera ett slumptal mellan **50** och **250** som tilldelas variabeln **key**, slumpnyckeln som används vid kryptering. Därför skickas den tillsammans med **fileText** till **Encrypt()** med anropet som ingår i följande sats:

```
fileText = EncryptText.Encrypt(fileText, key);
```

En blick på metoden **Encrypt()** som är externlagrad i klassen **EncryptText** förklarar saken:

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

class EncryptText
{
   public static string Encrypt(string t, int n)
   {                                    // t = filinnehåll
      char ch;
      string temp = "";                 // Initierar temp

      for (int i=0; i <= t.Length-1; i++)
      {
          ch = (char)(t[i] + n);        // Ändrar tecknen från t
          temp += ch;                   // Lägger tecknen i temp
      }

      return temp;                      // Reurnerar krypterat
   }                                    // filinnehåll
}
```

Med den första parametern **t** får metoden **Encrypt()** tillgång till det **string**-objekt som skapas i den anropande metoden **Main()**. Adressen till detta objekt kopieras över till referensvariabeln **t** när **Encrypt()** anropas. Samma sak sker med krypteringsnyckeln vars värde kopieras till den andra parametern **n**. Sedan har vi i kroppen av metoden två lokala variabler **ch** och **temp**. Den första som är av typ **char** initieras i **for**-loopen och lagrar det krypterade tecknet för att slutligen överföra det via konkatenering till strängen **temp**. **for**-satsen går igenom alla tecken i **t** genom att initiera sin räknare **i** till **0** och avsluta loopen när räknaren har nått strängens sista tecken. Att man börjar med **0** beror på att C# räknar strängens första tecken med index **0**, det andra med index **1** osv. så att det sista tecknet får t.ex. index **25** om strängen innehåller **26** tecken. **Length** är en **string**-egenskap som ger antalet tecken i strängen, här **t**. Därför har vi i **for**-loopen avslutningsvillkoret **i <= t.Length - 1**. I varje varv av den läggs det uttagna tecknet från **t** i den lokala **char**-variabeln **ch** och görs om till ett nytt tecken med satsen **ch = (char)**

(t[i] + n); där tecknet **ch**:s Unicode adderas med heltalet **n**. Resultatet omvandlas med explicit typkonvertering till **char** för att sedan tilldelas **ch**. Utan explicit typkonvertering skulle vi få kompileringsfel pga C#:s vägran att automatiskt typomvandla nedåt från **int** till **char**. **for**-loopens sista sats bygger den krypterade strängen **temp** som efter **for** returneras när **Encrypt()** anropas två gånger i **Main()** – en gång för kryptering, en andra gång för dekryptering (framhävda med vit bakgrund i koden på sid 152).

Den andra gången anropas krypteringsmetoden i följande sats:

```
fileText = EncryptText.Encrypt(fileText, -key);
```

där tecknet **–** framför **key** inte ska tolkas som bindestreck utan som det matematiska *förtecknet minus* till variabeln **key**:s talvärde, där **key** är deklarerad som heltal av typ **int**. Vi skickar alltså **key**:s *negativa* värde till samma krypteringsmetod **Encrypt()** för att sätta tillbaka alla tecken på sina ursprungliga platser i ASCII-tabellen. Den aktuella parametern **key** överförs vid anrop till den formella parametern **n**. När **n** får ett positivt **key**-värde, ökas tecknens ASCII-kod med **n**. Ett negativt **key**-värde minskar ASCII-koderna med samma belopp. Därför kan vi använda samma metod även för dekryptering.

Men mellan de två anropen av **Encrypt()** – en gång för kryptering, en gång för dekryptering – har vi två andra anrop, först:

```
WriteFile.Write(fileText, "Encrypted.txt");
```

som skriver den krypterade texten **fileText** till filen **Encrypted.txt**. Den anropade metoden **Write()** är definierad i den externlagrade klassen **WriteFile**:

```
// WriteFile.cs
// Metod som skriver texten t till filen filnamn
using System.IO;

class  WriteFile
{
   public static void Write(string t, string filnamn)
   {
      StreamWriter fileForWrite = new StreamWriter(filnamn);
      fileForWrite.WriteLine(t);
      fileForWrite.Close();
   }
}
```

Det andra anropet mellan de två anropen av **Encrypt()** sker i satsen:

```
fileText = ReadShowFile.ReadShow("Encrypted.txt");
```

Anropet läser den krypterade texten från filen och visar den på skärmen. Resultatet kan beskådas på sid 151 och visar att filen verkligen är krypterad. Den anropade Metoden **ReadShow()** är definierad i den externlagrade klassen **ReadShowFile**:

```
// ReadShowFile.cs
// Metod som läser innehållet i filen filnamn, visar det på
// skärmen och returnerar filinnehållet som en sträng
using System;
using System.IO;

class ReadShowFile
{
   public static string ReadShow(string filnamn)
   {
      string word, temp = "";
      StreamReader fileForRead = new StreamReader(filnamn);
      while (!fileForRead.EndOfStream)
      {
         word = fileForRead.ReadToEnd();
         Console.WriteLine(word);
         temp += word;
      }
      fileForRead.Close();
      return temp;
   }
}
```

Metoden **ReadToEnd()** i **while**-satsen som är fördefinierad i klassen **StreamReader** läser filen **filnamn** ord för ord, lagrar innehållet i **string**-variabeln **word**. **EndOfStream** i **while**-satsens villkor flyttar markören till nästa tecken i filen och returnerar **true** om det finns ord kvar och **false** om det stöter på filslutstecknet. Så läses filen till slutet, lagras i **word** samt samlas i **temp**.

Nu återstår beviset på att krypteringen gjorts på ett sätt att vi alltid har möjligheten att återställa filen och att vi verkligen får filens ursprungliga skick. Efter det andra anropet av krypteringsmetoden med negativ slumpnyckel skrivs den återställda texten till filen **Recovered.txt** med följande anrop:

```
WriteFile.Write(fileText, "Recovered.txt");
```

Och med följande anrop läses den återställda texten från samma fil och skrivs ut på skärmen:

```
fileText = ReadShowFile.ReadShow("Recovered.txt");
```

Resultatet kan beskådas i den sista delen av utskriften på sid 151. Man ser att den ursprungliga texten från filen **OriginalText.txt** är helt återställd. T.o.m. radbrytningarna är på plats i den återställda versionen, däremot inte synliga i krypteringen.

Övningar till kap 4

4.1 Skriv ett program som läser in en sträng, lagrar den i en array av **`char`** och skriver ut den baklänges. Använd tekniken i programmet **`EncryptChar-Test`** (sid 139) för att omvandla den inlästa strängen i en array av **`char`**.

4.2 Skriv ett program som skapar en tom fil, skriver i den texten ”Den här texten kommer från mitt första C# filhanteringsprogram” och sedan läser från den samt skriver ut innehållet på skärmen. Som mall kan du ta programmet **`-WriteReadFile`** (sid 142) och modifiera den.

4.3 Modifiera programmet från övn 4.2 ovan: Istället för att hårdkoda texten i programmet, läs in den så att programmet skriver vilken inläst text som helst till filen och läser den sedan därifrån.

4.4 Varje gång man kör programmen från övn 4.2 eller 4.3 efter första gången, rensas och återställs filen och endast den senaste texten hamnar i den. Skriv ett program som gör samma sak som övn 4.2 men bibehåller filens gamla innehåll och lägger till den nyinlästa texten utan att radera gammal data. Du kan åstadkomma det genom att öppna filen i *append mode*.

4.5 Modifiera klassen **`RandPasswd`** (sid 149) som genererar ett slumplösenord, genom att använda en annan, ny lösenordpolicy: 3 gemener, 2 versaler (samt ? och @) och 2 specialtecken. Testa den nya policyn i programmet **`Rand-PasswdTest`** (sid 147) för att skriva ut de nya slumplösenorden samt tillhörande användarnamn till en fil.

4.6 ***Kryptering av fil (Projekt)*** Modifiera klassen **`EncryptText`** (sid 153) genom att implementera följande ny krypteringsmetod. Gör så här:

- Döp om klassen **`EncryptText`** till en ny klass **`EncryptText_New`**.
- Döp om krypteringsmetoden **`Encrypt(string t, int n)`** till **`Encrypt_New (string t, int k, int m)`**.
- Definiera krypteringen i den nya metoden med funktionen $y = k\,x + m$, dvs ersätt satsen
  ```
  t[i] = (char) (t[i] + n);
  ```
  med
  ```
  t[i] = (char) (k*t[i] + m);
  ```
- Lägg till en ny metod **`Decrypt(string t, int k, int m)`** som ska dekryptera tecknen med den *inversa* funktionen $y = (x - m) / k$, dvs:
  ```
  t[i] = (char) ((t[i] - m)/k);
  ```
- Anropa båda metoderna från **`Main()`** genom att skicka värdena **`3`** till **`k`** och **`-40`** till **`m`**. Krypteringsfunktionen blir då $y = 3\,x - 40$ och dekrypteringsfunktionen $y = (x + 40) / 3$.

- I övrigt ska all skrivning till och läsning från fil kodas precis som i det ursprungliga programmet **`EncryptFile`** (sid 152).

4.7 ***Kryptering av databas (Projekt)*** Skriv ett program som krypterar en redan existerande databastabell i *Access* som ingår i Microsofts Office-paket vilket förutsätter att du har tillgång till programvaran. I så fall skapa i Access en liten tabell, t.ex. ett adressregister över dina kompisar och exportera tabellen till en textfil av typ **`*.txt`** (Arkiv → Exportera → Filformat *.txt, …). Välj vid exporten semikolonet som avskiljare mellan tabellens kolumner. Kryptera textfilen med något av programmen i detta kapitel. Men lägg till kod som gör att semikolonet inte krypteras. Skriv det krypterade innehållet till en annan textfil. Importera textfilen till en ny tabell i Access. Spara krypteringsnyckeln och använd den för att återställa den krypterade tabellen och verifiera resultatet.

Kapitel 5

Datastrukturer i relationsdatabaser

Ämne	Sida	Program/Länk
5.1 Introduktion till databaser	159	
5.2 Relationsdatabaser	161	
- Modularisering	161	
- Liknelse med klass och objekt	163	
- Vad är en relation i databaser?	164	
- Primär- och främmande nycklar	168	
5.3 Introduktion till SQL	169	
- Databashanterare	169	
- Klient – Server-modellen	170	
- SQL – databasers språk	172	
- SELECT-satsen	173	
- CREATE TABLE-satsen	178	
5.4 Vår första SQL Server databas	180	**FirstDatabase**
- Att koppla upp sig till SQL Servern	181	
- Att visa databasens innehåll	184	
5.5 En SQL klient i C#	186	**SQLclient**
- Att skriva och exekvera egna SQL satser	188	
- Grafiskt gränssnitt till SQL klienten	193	
5.6 Att skapa och designa en databas i C#	198	**Kursverksamhet**
- Modelldatabasen Kursverksamhet	199	
- Att skapa tabeller i databasen	200	
- Att koppla projektets Dataset till databasen	203	
- Att skapa relationer mellan tabeller	206	
- Att lägga in data i tabellerna	208	
5.7 Att förse databasen med funktionaliteter	211	**AddressBook**
Övningar till kapitel 5	217	

5.1 Introduktion till databaser



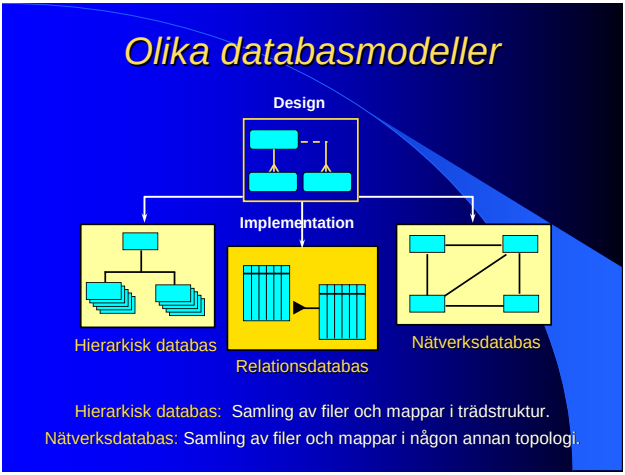
Information som samlas och lagras på ett ställe för att kunna användas senare, genererar en databas. Stället där informationen samlas behöver inte vara en dator. En samling papper eller kort med viktig information som man förser med namn eller nummer så att de kan sorteras, kan göra samma tjänst. Förr i tiden förvarades sådan information i tunga, låsbara *arkivskåp* av massiv järn, för att säkra informationen mot olovlig användning, inbrott, eld osv. (se bilden ovan). Arkivskåpet kunde innehålla information om t.ex. delbetalande kunders skulder i ett varuhus, sjukvårdsjournal i ett sjukhus, låneböcker i ett bibliotek, register över elevärenden i en skola, medlemsregister i en förening eller kund- resp. varuregister i ett företag. Idag är databaser bakom webbsidor av stort intresse, där uppdaterbar information lagras.

Datoriseringen har gjort att arkivskåpet gått till historien för gott. Idag finns det inget effektivare medium för lagring av information än datorn. Men varken mediet eller den lagrade informationen i sig har någon egentlig betydelse, när det gäller effektivitet. Det enda som räknas är informationens *struktur*, dvs *hur* information lagras. Strukturen avgör hur man *hittar* information man är ute efter. Strukturen avgör hur lagringen av information är *organiserad* från början. Man pratar om databasens *modell*. Avgörande är den modell man tillämpat när man skapade databasen. Det finns olika *databasmodeller*, se nästa sida. Bland dem har *relationsdatabasmodellen* visat sig vara både mest effektiv och enklast att underhålla.

Själva begreppet *databas* används i många olika sammanhang. I minst två av dem kan man precisera betydelsen av databas så här:

1. *en samling av information* i form av tabeller, relationer, nycklar och andra databasobjekt (vyer, sekvenser, index, ...), t.ex. HR-databasen (sid 219).

2. *en programvara* som hanterar databaser, en s.k. *databashanterare* (sid 169), t.ex.: SQL Server, Access, MySQL, Oracle, DB2, ...



Dessa tre databasmodeller har bl.a. använts sedan datoriseringen av databaser. Det är inte ens idag ovanligt att folk samlar information i filer och lagrar filerna i mappar. Så länge mängden av data håller sig inom en viss gräns är det inte heller något fel med det, så länge man hittar den information man sedan letar efter.

Under åren har de *hierarkiska databaserna* växt fram, helt oplanerat och spontant. Hierarkiska heter de eftersom filerna läggs i mappar organiserade i en trädhierarki liknande mappsystemet i de flesta operativsystemen (Windows, Linux, Unix, …).

Nätverksdatabasmodellen liknar de olika topologier som förr i tiden fanns i de datornätverk som byggdes med kablar *(Buss, ring, stjärna, …)*. Både den hierarkiska och nätverksdatabasmodellen används inte längre för lagring av stora datamängder. Anledningen är att *relationsdatabasmodellen* i praktiken har visat sin överlägsenhet. Vi kommer snart att inse detta. I fortsättningen kommer vi att endast ha att göra med denna databasmodell vars principer vi börjar att lära oss i detta kapitel.

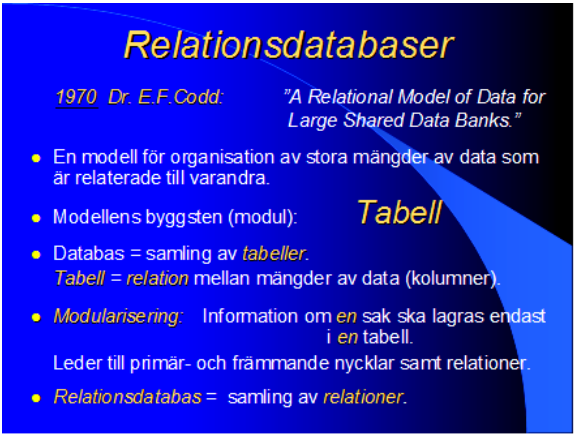
Relationsdatabasmodellen

1970 introducerade *Codd*, forskare inom datavetenskap på IBM, denna modell i sin doktorsavhandling *”A Relational Model of Data for Large Shared Data Banks”*. Han kallade sin modell för en *relationsmodell*, för den bygger på begreppet *relation* som vi kommer att behandla på sid 164. Relationsdatabasmodellens fördelar är så stora att de flesta databaser i världen idag är relationsdatabaser. De överträffar alla andra modeller med avseende på:

- Effektivitet
- Tillförlitlighet
- Stabilitet

Det är anmärkningsvärt att databasspråket SQL (sid 172) utvecklades samtidigt av en annan forskargrupp på samma företag IBM och är logiskt uppbyggt på samma principer som relationsdatabaser och fungerar bäst med dem.

5.2 Relationsdatabaser

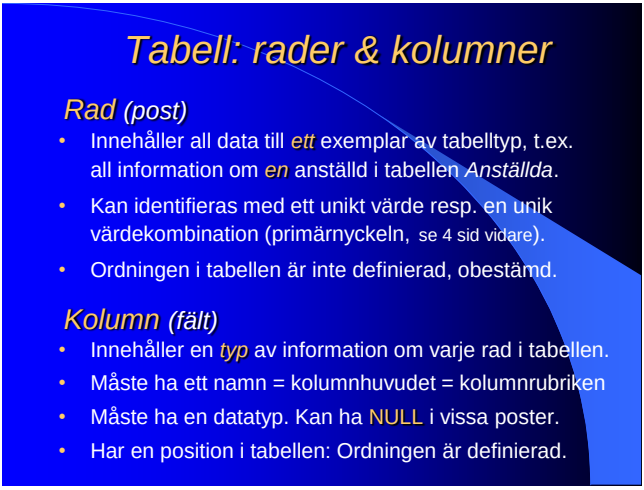


Relationsdatabasmodellens minsta byggsten (modul) är *tabellen*. Intuitivt har man en någorlunda klar uppfattning om en tabell som en samling av rader och kolumner. Lite svårare är det att inse att en tabell kan definieras som en relation mellan mängder, där *relation* själv också är en mängd (sid 164). Kopplingen mellan *tabell* och *relation* är inte intuitiv. För att förstå den måste vi reda ut andra begrepp.

Modularisering

Ett av dessa begrepp är *modularisering* – ett koncept som används i all problemlösning, bl.a. i programmering. Modularisering används för att bryta ned stora program i mindre och enklare hanterbara moduler för att åstadkomma bättre strukturering samt effektivitet, t.ex. genom återanvändning av kod. I databassammanhang innebär modularisering att man samlar information om *ett* nyckelbegrepp (*en* kategori av saker och ting, en s.k. *entitet*) endast i *en* tabell och inte blandar data av olika typer i en och samma tabell. Har man t.ex. i ett företag data om anställda och avdelningar ska man inte samla dem allihopa i *en* tabell, utan skapa en tabell för anställda och en annan för företagets avdelningar. Vilka fördelar denna princip av relationsdatabasmodellen har kommer att visas längre fram (sid 166, 167).

Modularisering – att separera databasens tabeller i fristående moduler – har vissa konsekvenser. En av dem är att en viss information i, säg tabell A, inte längre är direkt tillgängling i samma tabell utan har lagrats i en annan tabell B. T.ex. finns i tabellen över anställda (A) inte namnet på den avdelning de jobbar, för tabellen över avdelningar (B) har separerats från A. För att ändå kunna komma åt anställdas avdelningar måste en relation etableras mellan A och B. Detta leder till att man måste införa *nycklar*, närmare bestämt *Primär-* och *främmande nycklar* (sid 168). Nycklarna beskriver relationen mellan tabellerna. På så sätt blir databasen en samling av relationer och därmed en *relationsdatabas*.



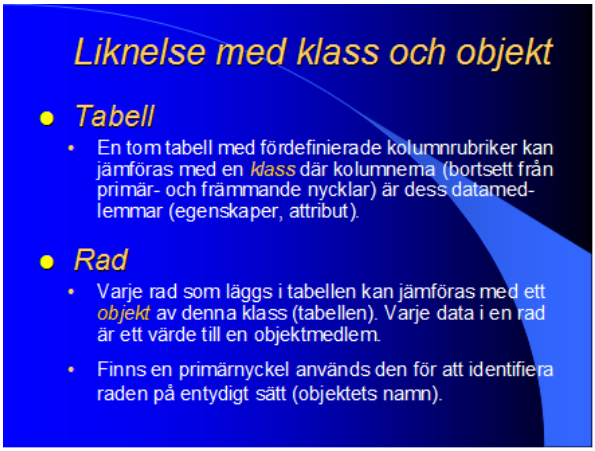
Att relationsdatabasmodellens minsta modul är *tabell* föranleder oss att närmare titta på dess beståndsdelar: rader och kolumner. Ibland kallar man raderna även för *poster*, och kolumnerna för *fält*. Vi kommer dock att hålla oss till *rader* och *kolumner*. De har fått vissa roller som återspeglar deras funktionaliteter. Vi ska precisera:

En *rad* i en tabell t.ex. om ett företags avdelningar (tabell*typen*) får endast innehålla information om en speciell avdelning: Avdelningens namn, ort, postnr, gatuadress, etableringsdatum osv. Raden utgör ett exemplar (objekt) av typen (kategorin, klassen) *Avdelningar*.

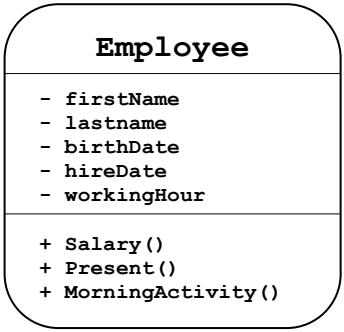
En *kolumn* däremot får endast innehålla information till en kolumnrubrik. T.ex. rubriken *postnr* kan ha talet 18047 som värde, rubriken *avdelningsnamn* kan ha texten IT eller *etableringsdatum* datumet 02-JAN-08. Alla värden i en kolumn måste vara samma typ av data, antingen tal, text, datum eller någon annan typ av data. Därför måste en kolumn ha en *datatyp*. På vissa rader kan information saknas. Då blir det en tom cell i kolumnen, och man säger att denna cell innehåller NULL – ett ofta förekommande nyckelord i databassammanhang som betyder *ingen information* och som inte borde förväxlas med *talet 0* som *är* information.

Medan en kolumn alltid måste ha ett namn (rubriken) som den entydigt kan identifieras med, behöver en rad inte a priori ha ett sådant. Man kan dock ge den en identifieringsnyckel i form av ett nummer i en extra kolumn eller en nummerkombination i flera kolumner för att kunna hitta den i tabellen, vilket är rekommenderat att göra. Denna nyckel kallas tabellens *primärnyckel* och får inte innehålla varken dubbletter eller NULL (sid 168).

Ytterligare en skillnad mellan rader och kolumner är deras *ordning*. Medan kolumnerna har en fast ordning i tabellen, är radernas ordning odefinierad. I vissa sammanhang kan man t.o.m. referera till en kolumn genom att använda dess plats i tabellen, t.ex. kolumn nr 1 eller 2 osv. istället för att ange kolumnrubriken. Raderna däremot är oordnade.

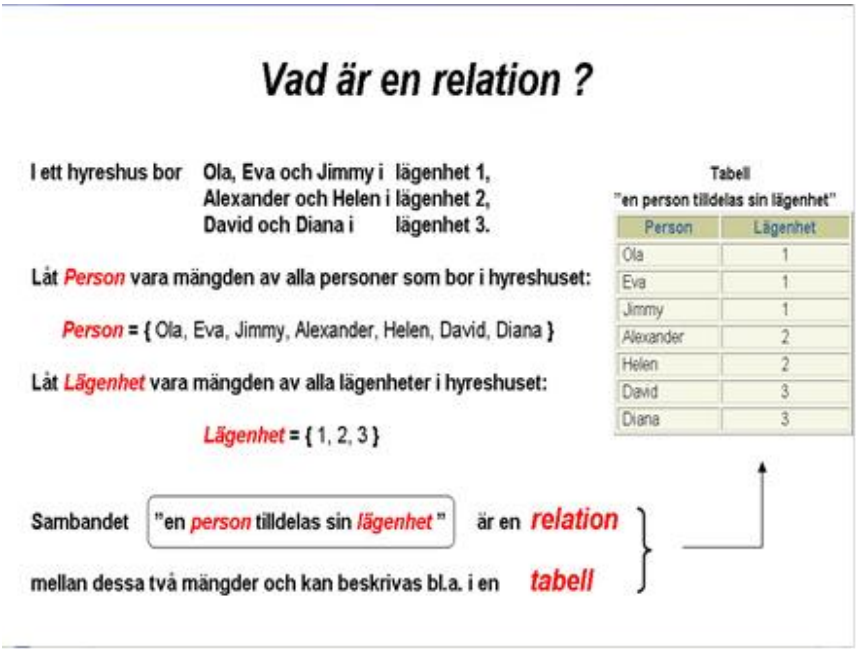


Låt oss titta på klassdiagrammet till höger. Om vi bortser från metoderna (markerade med **+**) och koncentrerar oss på datamedlemmarna (markerade med **–**) kan vi jämföra klassen **`Employee`** med en tom tabell vars kolumner är klassens datamedlemmar, se nedan. Tabellens struktur är identisk med klassens struktur när det gäller datamedlemmarna, vilket ger oss en ledtråd om hur vi ska bygga våra tabeller. Klassens metoder kommer att bli funktionaliteter som sedan måste läggas till med kod.



Förnamn	Efternamn	Födelsedatum	Anställn.datum	Arbetstid

Tabellen borde döpas till **`Employees`**. Just nu är den tom. Men när vi lägger in några anställda i den motsvarar detta att skapa objekt av klassen **`Employee`**. Varje cell i en sådan rad innehåller information om just denna anställd, vilket kan jämföras med de värden som man skickar med konstruktorn när man med **`new`** skapar ett objekt av klassen **`Employee`**. Objektet initieras med dessa värden. När tabellen sedan växer innebär det att man skapar ytterligare objekt av samma klass, dvs lägger in flera anställda i tabellen. I en relationsdatabas borde man lägga till tabellen ovan en kolumn bestående av t.ex. ett löpande nummer (utan dubbletter) som ska sedan fungera som tabellen **`Employees'`** primärnyckel (sid 168). Primärnyckeln kan jämföras med objketets (radens) namn och är till för att på ett entydigt sätt kunna identifiera raden och kunna relatera tabellen till databasens andra tabeller. Sådana *relationer* behandlas på de följande 4 sidorna.



Begreppet relation har sitt ursprung i *mängdläran* där man betraktar mängder av saker och ting (föremål, objekt) – reella eller virtella – och definierar operationer mellan dem (union, snitt, …). Varje operation genererar en ny mängd. Läs mer om operationer mellan mängder i avsnittet ***2.5 Mängdlära och logik*** på sid 79. I databassammanhang är mängdbegreppet av intresse därför att vi har att göra med *mängder av data* och med relationer mellan dem. De saker och ting som ingår i en mängd kallas *element*. En kolumn i en tabell kan anses som en mängd av sina celler. En tabell kan betraktas som en mängd av sina kolumner. Den *tomma mängden* är den som inte har något element alls. En mängd kallas *väldefinierad*, om man alltid kan avgöra om något element tillhör mängden eller ej. Vi utesluter icke-väldefinierade mängder (Fotnot sid 79). I exemplet ovan har vi två väldefinierade mängder: *Person* och *Lägenhet*. Läs mer om ***mängder***.

En *relation* är ett samband som tilldelar ett element ur en mängd ett element ur en annan mängd. Relationen mellan *Person* och *Lägenhet* definieras av den inledande informationen om vem som bor i vilken lägenhet. Det finns olika sätt att beskriva en relation. Tabellformen ovan är ett sätt att göra det. Praktiskt relevant blir relationsbegreppet först när man ställer upp relationer mellan *tabeller*. Därav har relationsdatabasmodellen fått sitt namn. Men en relation mellan *tabeller* bygger i sin tur på relation mellan *kolumner*, i exemplet ovan mellan kolumnen *Person* och kolumnen *Lägenhet*. I en relationsdatabas blir detta en relation mellan kolumnerna som bildar tabellens primär- och främmande nycklar (sid 168).

Relation mer exakt

Ex.: Relationen ”en person tilldelas sin lägenhet” – låt oss kalla den R – är definierad så här:

” I ett hyreshus bor Ola, Eva och Jimmy i lägenhet 1, Alexander och Helen i lägenhet 2, David och Diana i lägenhet 3 ”

Denna relation kan beskrivas i en tabell →

Tabell R:
”en person tilldelas sin lägenhet”

Person	Lägenhet
Ola	1
Eva	1
Jimmy	1
Alexander	2
Helen	2
David	3
Diana	3

Relationen R är en delmängd av den cartesiska produkten ***Person x Lägenhet*** :

R = { (Ola, 1), (Eva, 1), (Jimmy, 1), (Alexander, 2), (Helen, 2), (David, 3), (Diana, 3) }

Relation = delmängd av den cartesiska produkten.
Tabell = relation mellan tabellens kolumner. Raderna *beskriver* relationen.

Läs om cartesiska produkten i avsnittet ***2.5 Mängdlära och logik*** på sid 79.

Varför är 2 tabeller bättre än 1?

Exempel:

Information om personers jobb och avdelningars plats där de jobbar lagras i 1 tabell:

ID	Namn	Jobb	Avdelning	Plats
1	Ola	IT-proffs	IT	Kista
2	Eva	Programmerare	IT	Kista
3	Jimmy	Revisor	Ekonomi	Stockholm
4	Alexander	Säljare	Marknad	Göteborg
5	Helen	VD	Marknad	Göteborg
6	David	Chaufför	Logistik	Älvsjö
7	Diana	Grafiker	PR	Liljeholmen

Uppdatering: **IT-avdelningen flyttas till Stockholm.**

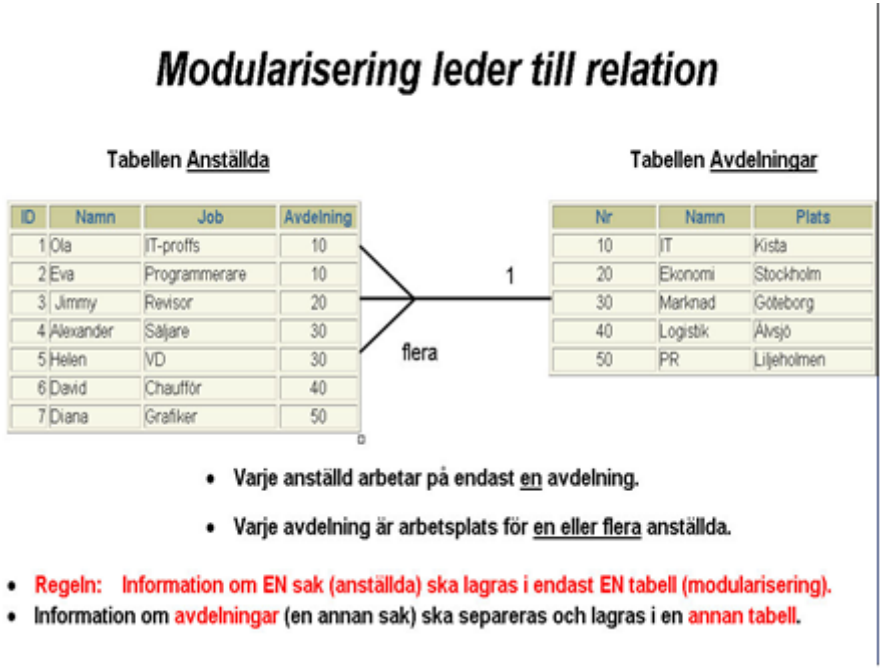
1 tabell: **Ändringen måste göras på flera ställen.** **Ingen Modularisering!**

2 tabeller: **Ändringen måste göras på ett ställe.** **Modularisering!**

Personerna i mängden *Person* (på förra sidan) bor inte bara i lägenheter utan har även vissa jobb. Låt oss anta att de arbetar på ett företag som har ett antal avdelningar som är belägna på olika platser. Tabellen ovan lagrar denna information. Inget konstigt med den tabellen, skulle man kunna tycka. Men den går emot relationsdatabasmodellens princip om modularisering (sid 161), enligt vilken information om ett företags anställda ska lagras i *en* och information om företagets avdelningar i en *annan* tabell. Här är båda samlade i en tabell. Varför är det inte bra ur effektnivitetssynpunkt?

Information i tabellerna ska ju inte bara lagras utan även *underhållas* dvs uppdateras så att den alltid återger den aktuella situationen korrekt. Låt oss anta att det sker en ändring i företaget och avdelningen IT flyttas från Kista till Stockholm. I tabellen ovan måste denna ändring registreras på två olika rader i tabellen, därför att det finns två personer, Ola och Eva, som jobbar på IT-avdelningen. Men det här är ju bara ett exempel. Om det finns hundratals anställda på den avdelning som ska flyttas blir det en massa jobb som bara kostar en massa onödiga pengar. Onödiga, därför att man hade kunnat reducera jobbet till *en* ändring på *en enda* rad i en tabell om informationen om avdelningar från början hade funnits i en separat tabell. Dvs om man hade valt en annan modellering av databasen och modulariserat upplägget av tabellerna.

Frågan som uppstår i den modulariserade modellen är nu: Hur får vi fram svaret på frågan ”Var jobbar en anställd”? när informationen inte längre finns i en tabell utan i två tabeller? Nästa sida ger svar.



Här har tabellupplägget modulariserats och vi har två tabeller. Men tittar man noga och jämför antalet kolumner i den gamla (förra sidan) och den nya modellen (denna sida), kan man konstatera att det finns 5 kolumner i 1-tabellmodellen, medan 7 kolumner sammanlagt i 2-tabellmodellen, tabellerna Anställda och Avdelningar. Dvs det har kommit till två nya kolumner. Modellen har fått en mer komplex struktur. På ytan ser det ut som om vi hade krånglat till det hela. Men i själva verket är det tvärtom! Vi har effektiviserat och förenklat tabellernas underhåll. Om vi tar upp exemplet från förra sidan, då IT-avdelningen skulle flyttas från Kista till Stockholm, behöver vi nu uppdatera endast *ett* värde på *en enda* rad i tabellen Avdelningar, nämligen texten Kista på första raden i kolumnen Plats och inget mer. I 1-tabellmodellen var vi tvungna att uppdatera två värden på två rader i tabellen.

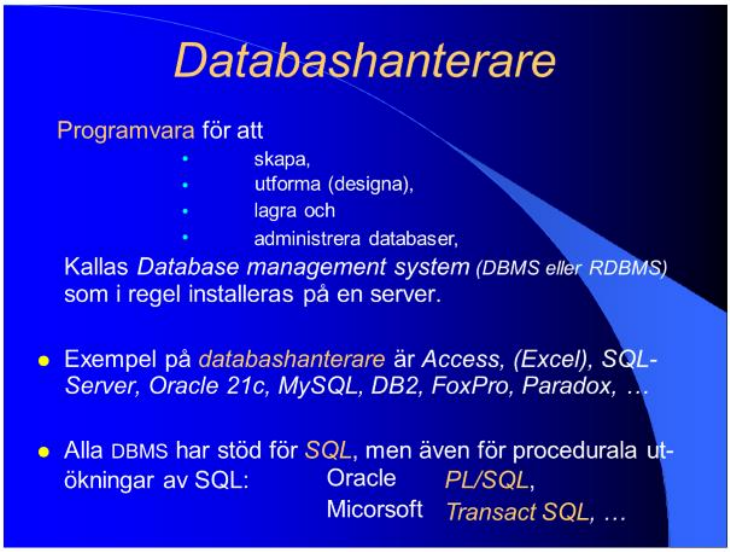
Frågan ”Var jobbar en anställd”? besvaras nu i den modulariserade 2-tabellmodellen på följande sätt: Anställden Alexander t.ex. som är säljare, jobbar enligt tabellen Anställda på avdelning nr 30. Med denna information går vi till tabellen Avdelningar, söker där i kolumnen Nr efter värdet 30 och hittar informationen att 30 är numret till avdelningen Marknad som ligger i Göteborg. Alltså jobbar Alexander i Göteborg. Vi kunde besvara frågan tack vare de kolumner som vi laggt till i den nya modellen: Kolumnen Avdelning i tabellen Anställda och kolumnen Nr i tabellen Avdelningar. Man kallar den första kolumnen för *främmande* och den andra för *primärnyckeln*. De definierar en relation mellan de två tabellerna.



Relationsdatabasmodellens viktigaste praktiska konsekvens är införandet av nycklar i databasen. Det finns två typer nycklar, *primary* och *foreign keys*. Primärnyckeln behövs för att på ett entydigt sätt kunna identifiera en rad och ange ett exakt sökvillkor som hittar just denna rad bland tusen- eller kanske miljontals rader i en tabell. Dessutom behövs primärnyckeln för att kunna definiera främmande nycklar, varför den heter *primär*. Främmande nycklar behövs för att relatera tabeller till varandra och kunna hitta information som enligt modulariseringsprincipen finns i olika tabeller, t.ex. ”Vilka anställda jobbar på vilka (namngivna) avdelningar?”.

I praktiken består en primärnyckel av en (eller flera) kolumner som inte innehåller någon genuin information om själva tabelltypen, utan snarare administrativ data för effektiv hantering av tabellen. T.ex. är 174 ett nummer som anställden Ellen Abel fått i tabellen EMPLOYEES ovan. Så har kolumnen EMPLOYEE_ID blivit tabellens primärnyckel. En tabell får endast ha *en* primärnyckel, men den kan bestå av flera kolumner, i vilket fall man pratar om en *sammansatt* primärnyckel. Kolumnen DEPARTMENT_ID däremot är en främmande nyckel i tabellen EMPLOYEES, därför att den innehåller endast data från en annan tabells – nämligen DEPARTMENTS-tabellens – primärnyckel. Värdena i den talar om – via numret – på vilken avdelning en anställd arbetar. Dessa nummer är primärnyckelvärden i tabellen DEPARTMENTS. Där är de unika. Men som främmande nycklar i EMPLOYEES förekommer de flera gånger, eftersom flera anställda kan jobba på samma avdelning (sid 167). En främmande nyckel är en relations konkreta realisering.

5.3 Introduktion till SQL



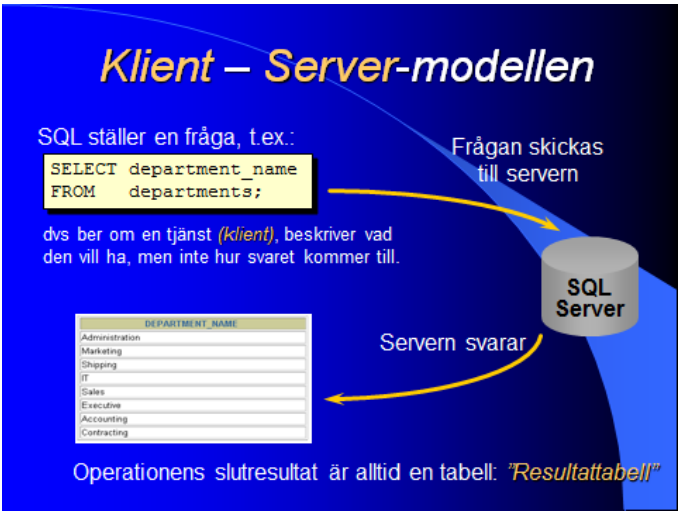
Begreppet *databashanterare* betecknar en programvara som hanterar databaser som i sin tur består av tabeller, nycklar osv. *SQL Server*, *Access*, *MySQL*, *Oracle*, *DB2*, … är exempel på databashanterare. Facktermen är *Database Management System (DBMS)*, ibland med tillägget *R* som syftar åt *Relational DBMS*. Vi använder, för att köra bokens kodexempel och övningar, databashanteraren Microsoft SQL Server. Även om programvaran *i regel* – dvs när den används i skarp produktionsmiljö – installeras på en server (dator med serverversionen av ett operativsystem, t.ex. Windows Server), är det fullt möjligt att installera den även på en vanlig klientdator (t.ex. Windows 10) för test- och utbildningsändamål – vilket vi gjort för att testa våra koder. Då finns både *databas*servern (DBMS) och klienten på en och samma dator. Det spelar ingen roll när det gäller att lära sig användningen av databashanteraren.

Microsoft SQL Server innehåller bl.a. en SQL-interpretator (tolk), en *Transact SQL*-kompilator (översättare) – båda förenade i en s.k. *parser* – ett integrerat verktyg för generering av maskinkod. Andra tillverkare har motsvarande verktyg i sina databashanterare, som alla stöder SQL.

Transact SQL, även kallad *T-SQL*, är Microsofts utökning av SQL, ett programmeringsspråk för databashanteraren Microsoft SQL-Server. För att övervinna SQL-språkets begränsningar, har man integrerat SQL i T-SQL, där man kan utnyttja programmeringens alla konster. Alla databastillverkare har utvecklat sina egna procedurala utökningar av den allmänna SQL-standarden. Microsofts utökningsprodukt är *T- SQL*, Oracle:s motsvarighet heter *PL/SQL* som står för *Procedural Language extensions to SQL*.

Klient – Server-modellen

För att förstå vad som egentligen pågår när man från C# ansluter sig till en databas och vilka program som är ansvariga för vilken del av denna kommunikation, speciellt samspelet mellan C# och SQL, ska vi i detta avsnitt titta på en modell som är typisk för arbetet med en databas i en skarp miljö som bäst kan beskrivas med den s.k. klient-server-modellen. Det är inte bara C# och SQL som är involverade i denna process utan också en annan, helt ny programvara vom vi inte använt hittills, nämligen SQL Server. På sid 169 hade vi nämnt den som ett exempel på en *databashanterare*, i princip jämförbar med Access, MySQL, Oracle, DB2 osv. En sådan programvara måste vara installerad på en serverdator i en skarp miljö för att vi som klienter ska kunna kommunicera med en databas. Så här t.ex. kan den se ut:



`SELECT`-satsen skrivs på en (klient)dator och ska ta fram kolumnen **`department_name`** från tabellen **`departments`**. Men databasen som administrerar denna tabell finns i regel inte på samma dator utan på en annan (server)dator som databashanteraren *SQL Server* är installerad på. Observera att du inte förväxlar *SQL Server* som är ett program dvs mjukvara med servern som är en dator dvs hårdvara.

Vi som sitter vid klientdatorn skickar SQL-frågan till servern som sedan svarar med en resultattabell, i det här fallet kolumnen **`department_name`**:s innehåll. Serverns svar är alltid i tabellform, vare sig den har en, två eller flera kolumner. Hur servern utför själva sökoperationen och hur den hittar tabellen **`departments`** i databasen samt kolumnen **`department_name`** i den, behöver vi inte bry oss om. Det enda vi behöver göra är att exakt beskriva vad operationens slutresultat ska bli. Och det gör vi i SQL-frågan. Det är därför SQL också kallas ett *deklarativt* språk: Man beskriver bara *vad* man vill ha, inte *hur* det ska göras – till skillnad från procedurala språk där man kodar algoritmer dvs i allra högsta grad beskriver *hur* ett problem ska lösas. Språk som beskriver *hur* ett problem ska lösas kallas *procedurala*. C, C++,

Java, C#, PL/SQL, Transact SQL är exempel på procedurala språk, även om några av dem dessutom är objektorienterade. SQL beskriver bara vad problemets – dvs sökoperationens – slutresultat ska bli.

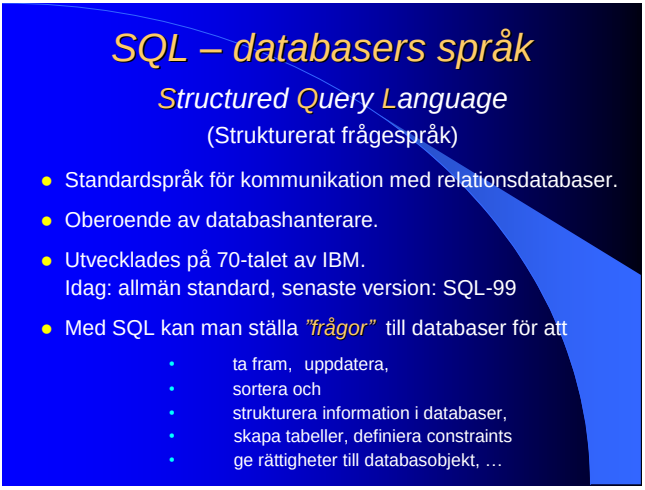
Det viktigaste i klient-server-modellen är kanske förståelsen för att det endast är via databashanteraren – i vårt fall *SQL Servern* – vi kan komma åt databasen och dess tabeller, öppna dem och med hjälp av SQL titta på deras innehåll. Inte den fysiska distinktionen mellan klient- och serverdatorn är avgörande – den kan i vissa fall t.o.m. slopas – utan den logiska skillnaden mellan programmen på klient- och serversidan. På serversidan måste en databashanterare vara installerad. Den administrerar inte bara databasen och underhåller dess tabeller. Den har även ett verktyg som kan exekvera SQL-kod dvs kan tolka SQL-språket till maskinkod – en s.k. SQL-interpretator (tolk).

Lyckligtvis är databashanteraren *SQL Server* en integrerad del av Visual Studio som (förhoppningsvis) installerades med när vi började programmera med C# *(Progr1, Appendix B)*. Så den borde vara installerad på våra datorer, bara att vi inte har använt den hittills. Nu ska vi börja göra det. Till denna databashanterare kommer vi att skicka våra SQL-satser via C#. Dvs C# kommer att vara den klientmiljö hos oss som kommunicerar med *SQL Servern*. I och med detta kännetecknas vår miljö av en annan omständighet som skiljer sig från den skarpa miljön som beskrivs på förra sidans bild: Klient och server finns i en och samma dator. Rent tekniskt sett är det möjligt. Så länge det handlar om en test- och utbildningsmiljö är det t.o.m. ganska bekvämt att inte behöva administrera två olika datorer samt deras uppkoppling till varandra. Men då blir det ännu viktigare att vi i denna miljö som förenar klient och server i en och samma maskin, skiljer klient- och serverfunktionerna, C# och *SQL Server*, från varandra och relaterar dem till rätt program, även om båda är integrerade i Visual Studio. Vi kommer nu att använda denna miljö i hela databaskapitlet.

ADO.NET-objektmodellen

ADO.NET står för *ActiveX Data Objects for .NET* och är ett bibliotek av fördefinierade C#-klasser som ingår i *.NET*-plattformen och kan användas för att komma åt databastjänster på *SQL Server* och andra databashanterare. ADO.NET är en ny produkt som ersätter Microsofts gamla *ActiveX Data Objects*. Även om vi kanske inte direkt kommer att ha att göra med ADO.NET-objekt i våra enkla databasapplikationer, är det värt att känna till den bakomliggande teknologin. De viktigaste namnutrymmen i ADO.NET-objektmodellen med sina tillhörande klasser är:

- **System.Data**
  - **DataSet**
  - **DataTable**
- **System.Data.OleDb**
- **System.Data.SqlClient**
  - **SqlConnection**
  - **SqlCommand**
  - **SqlDataAdapter**

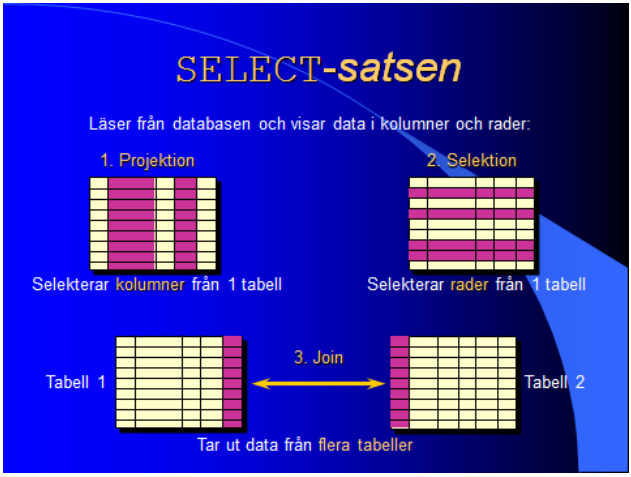


SQL är världens mest använda språk för kommunikation med databaser – den allmänna standarden i hela världen som gäller i alla databashanterare. Även om SQL själv kallar sig för ett strukturerat ”fråge”språk, är dess användningsområdet långt ifrån begränsad till att ”fråga” för att få fram en viss information. Med SQL kan man även ändra innehållet i tabeller, skapa tabeller och andra databasobjekt, definiera primär- och främmande nycklar (därmed relationer) samt andra s.k. *constraints*, skapa användarkonton, tilldela dem rättigheter och mycket mer. *Constraints* (restriktioner) är regler som ställs upp för att upprätthålla och bevaka databasens konsistens och integritet (helhet), vilket bl.a. innebär att det aldrig får finnas någon motsägelsefull information i databasen.

Pga SQL:s stora användningsområde skulle man kunna överge den historiska beteckningen *fråge*språk och prata om SQL som ett *kommunikationsspråk* istället där ”kommunikation” även omfattar uppdatering samt underhåll och administration.

SQL utvecklades i början av 70-talet efter Dr. E.F.Codd:s banbrytande arbete om relationsdatabaser (sid 160) av ett forskarteam på IBM, kommersialiserades 1979 av *Relational Software* (föregångaren till Oracle) och standardiserades 1986 av ANSI, det amerikanska och 1987 av ISO, det internationella organet för standardisering. Sedan dess har ISO ökat SQL:s funktionaliteter. Den senaste standarden är SQL-99 som kompletterades 2003/2006 med bl.a. stöd för XML *(eXtensible Markup Language)*, ett språk med syftet att kunna utbyta data mellan olika informationssystem.

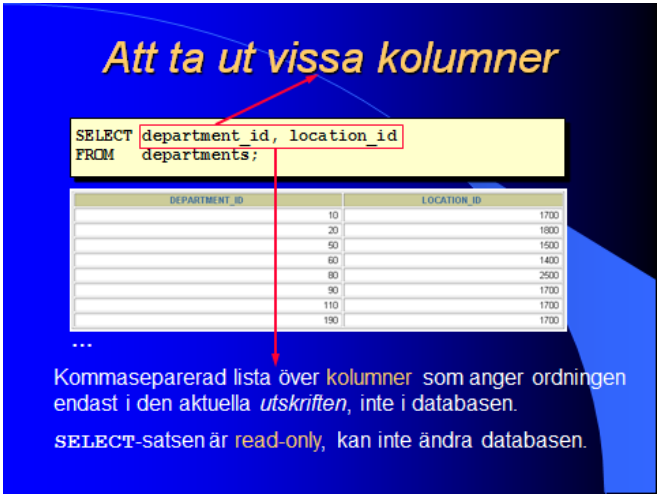
Har du installerat Visual Studio på din dator så finns det även SQL med installerad på din dator, närmare bestämt databashanteraren Microsoft SQL Server. Du kan etablera kontakt med den när du öppnar Visual Studio, vilket vi kommer att lära oss i *5.4 Vår första SQL Server databas med C#* (sid 180).



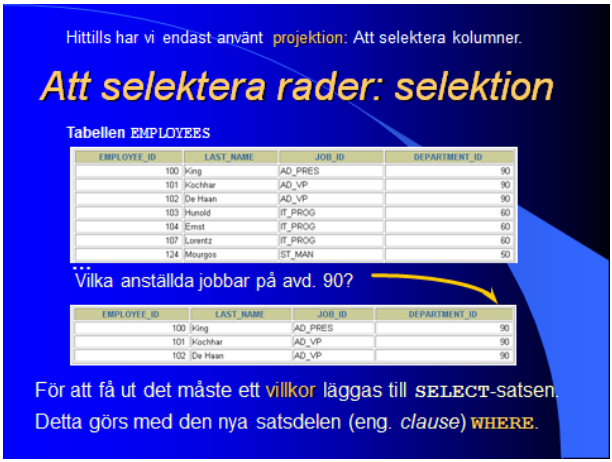
SELECT-satsen är SQL-språkets mest använda sats, har många varianter och kan kombineras med de flesta andra satser i SQL. Vad den kan göra visas kortfattat på bilden ovan: Att selektera (välja ut) data från databasens tabeller, antingen kolumner (projektion) eller rader (selektion). Detta kan göras från en, två eller flera tabeller. I praktiken innehåller ju en skarp databas stora mängder av information. Men i det dagliga arbetet behöver man ofta bara en liten bråkdel av denna väldiga information. **SELECT**-satsen ger oss möjligheten att selekter och få ut exakt den information som vi önskar just då. I så fall måste vi definiera var denna information är lagrad i databasen, dvs i vilken tabell, i vilken kolumn och på vilken rad av denna tabell osv. Relationsdatabasens struktur gör det möjligt att hitta den sökta informationen med en enkel och logisk syntax i **SELECT**-satsen som visas på bilden nedan. Denna sats tar fram alla kolumner – med symbolen ***** – från tabellen **departments**. När denna sats skickas till databasen svarar servern med att visa tabellen **departments'** alla kolumner dvs hela tabellinnehållet. Början av denna utskrift är avbildad under **SELECT**-satsen.

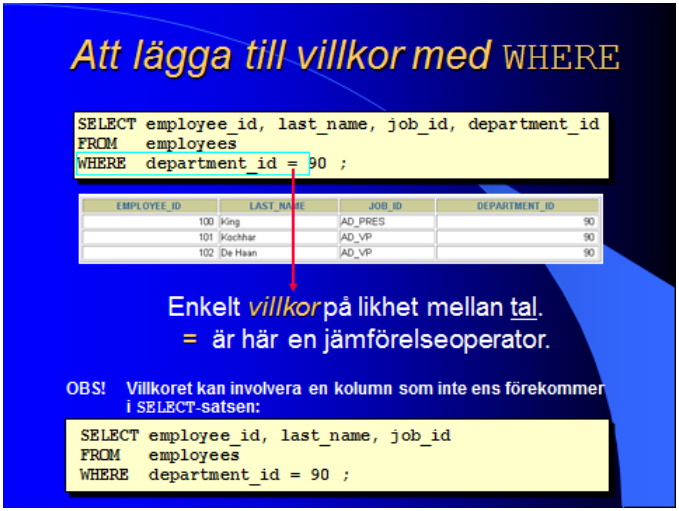


Tabellen har fyra kolmner vars rubriker syns i första raden. Listar man upp dessa efter **SELECT** får man en alternativ syntax för **SELECT**-satsen som ger samma resultat.



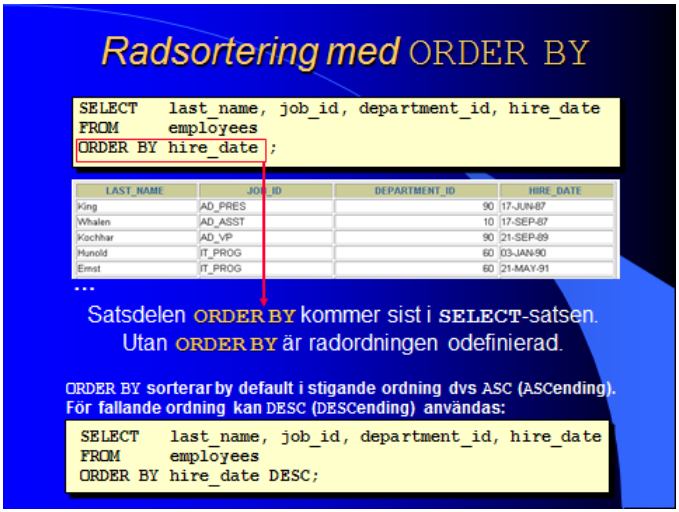
SELECT och **FROM** är reserverade ord i SQL. Efter **SELECT** står kolumnernas och efter **FROM** tabellens namn. Här selekterar **SELECT**-satsen endast kolumnerna **department_id** och **location_id** från tabellen **departments**. Svaret från servern är kolumnerna med rubriker och innehåll som visas under **SELECT**-satsen i den ordning vi angett dem i satsen, vilket inte har att göra med i vilken ordning de är lagrade i tabellen. Att innehållet i dessa kolumner är tal beror på att de är nycklar i tabellen: **department_id** är primärnyckeln och **location_id** är en främmande nyckel i tabellen **departments**. Främmande nyckeln hänvisar till en annan tabell, närmare bestämt till tabellen **locations**. Där hittar man dessa nummer som är är tilldelade vissa orter som är säten till resp. avdelning vars nummer i sin tur står i kolumnen **department_id**. På så sätt kan man få reda på en eller flera avdelningars säten. Men detta kräver bl.a. att man selekterar inte bara kolumner (projektion) utan även rader (selektion: Tabellen EMPLOYEES lagrar information om företagets anställda. Ett utdrag ur denna tabell på bilden till höger visar att tre anställda jobbar på avdelningen 90. Hur kan vi selektera från denna tabell de rader som i kolumnen **department_id** har värdet 90? Vi får lägga till **SELECT**-satsen en ny satsdel som inleds med det SQL-reserverade ordet **WHERE**.





Här utvidgas **SELECT**-satsen med **WHERE**: Medan efter **SELECT** står kolumner och efter **FROM** tabellen, skrivs efter **WHERE** ett villkor som jämför värdena i kolumnen **department_id** med talet 90. Servern svarar med endast de rader för vilka detta villkor visar sig vara sant. Dessa visas under **SELECT**-satsen. Istället för detta enkla villkor kan efter **WHERE** även stå ett sammansatt villkor som man kan formulera med logiska operatorer. Istället för en jämförelse mellan kolumnvärden och tal kan även jämförelser göras mellan kolumnvärden och tecken, strängar eller delar av strängar. Ja t.o.m. mönstermatchning mot delsträngar är möjlig. Med det reserverade ordet **LIKE** som man skriver istället för likhetstecknet i villkoret efter **WHERE** kan ganska avancerade sökningar göras i tabellen för att selektera just de rader man behöver. Den enda begränsning man har är att de jämförda objektens datatyper måste överensstämma. Ett tal kan inte jämföras med en sträng. I exemplet ovan måste värdena i kolumnen **department_id** vara av samma typ som talet 90. I en relationsdatabas har varje kolumn i en tabell en datatyp. Värden av en annan typ kan inte lagras i kolumnen. All data i en kolumn är av samma datatyp som vi måste känna till när vi använder kolumnen i ett villkor i satsdelen **WHERE**. Det är villkorets sanningsvärde som avgör vilka rader som skrivs ut.

Den första **SELECT**-satsen i bilden ovan skriver ut de anställda som jobbar på avdelning 90 tillsammans med sina andra uppgifter i kolumnerna **employee_id**, **last_name**, **job_id** och **department_id**. Den sista kolumnen skrivs ut endast i syftet att kontrollera att de utskrivna anställda verkligen jobbar på avdelning 90. I en verklig situation har man inte behov av denna information. Man har ju själv angett den i sökvillkoret. Då skulle man skriva **SELECT**-satsen utan kolumnen **department_id** vilket visas i den undre delen av bilden ovan. Även den kommer att fungera och skriva ut samma information utan avdelningsnumren. Detta visar att **WHERE**-villkoret kan involvera kolumner som inte förekommer i **SELECT**-satsen. Det räcker att de finns i tabellen.



När man med **SELECT**-satsen tar ut ett antal kolumner från en tabell presenteras kolumnerna i den ordning man angett dem i **SELECT**-satsen. Men i vilken ordning visas raderna? Det är obestämt och kan ej förutsägas. Servern skriver ut raderna mer eller mindre slumpmässigt, även om man kan förmoda att den tar dem i den ordning de står i databasens tabell. Men även där finns ingen tillgänglig information om radernas ordning. I princip är radernas ordning odefinierad.

Men vill man att raderna ska visas i en viss ordning, finns det möjligheten att lägga till **SELECT**-satsen en ny satsdel som inleds med den reserverade ordkombinationen **ORDER BY**, följd av ett (eller flera) kolumnnamn. I exemplet ovan står kolumnen **hire_date** efter **ORDER BY**. Då kommer raderna i utskriften att sorteras efter de anställningsdatum som står i kolumnen **hire_date**, närmare bestämt i stigande ordning. Dvs först kommer den anställd som blivit anställd tidigast av alla. Sedan följer anställda sorterade efter sina anställningsdatum. Självklart kan man ange en annan kolumn efter **ORDER BY**, t.ex. **lastname**, så att sorteringen görs efter efternamnen. Skriver man inte något explicit (by default) görs sorteringen i stigande ordning. Vill man ha sorteringen i fallande ordning kan man lägga till det reserverade ordet **DESC** (som står för **DESC**ending) efter kolumnnamnet i satsdelen **ORDER BY**. Har man flera satsdelar i **SELECT**-satsen, måste **ORDER BY** placeras sist i **SELECT**-satsen, t.ex.:

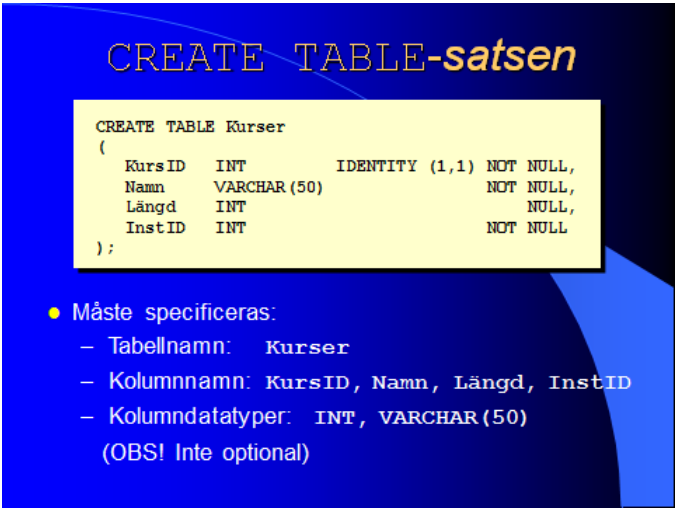
```
SELECT   last_name, salary
FROM     employees
WHERE    salary > 12000
ORDER BY salary DESC;
```

Denna **SELECT**-sats visar efternamn och lön till de anställda vars lön är över 12 000, sorterade efter lönerna i fallande ordning. Dvs vi kommer att se anställden med maximal lön först, följd av alla andra vars löner successivt faller, men ligger över 12 000.



Sökning i en tabell är en av de mest förekommande användningarna för databaser. Därför finns det i SQL en uppsjö av möjligheter att jämföra data med varandra. Beroende på vilken typ av data vi har att göra med – tal, tecken, text, datum osv. – har vi s.k. jämförelseoperatorer av olika slag. Det vanliga likhetstecknet = är en av dem. Men ofta har man inte möjligheten att testa på *exakt* likhet. Man kanske inte kommer ihåg det exakta namnet på en person man söker. Av en anställd i företaget kommer vi bara ihåg att hans eller hennes förnamn börjar på S och slutar på n. Då kan vi skicka SELECT-satsen på bilden ovan till databasen genom att i **`WHERE`**-villkoret skriva **`LIKE 'S%n'`**. Tecknet **`%`** är i SQL ett mönstermatchningstecken som står för vilket och hur många tecken som helst. Ett annat mönstermatchningstecken är _ och står också för vilket tecken som helst, men endats *ett*. En kombination av båda ger väldigt effektiva sökningar, se följande exempel:





Alla våra satser i SQL var hittills SELECT-satser. Det gemensamma hos dem är att de är *read-only* dvs de kan inte ändra databasen. Alla SELECT-satser, oavsett i vilken variant de förekommer, gör utdrag ur databasen och visar oss delar av den i form av en utskrift. Efter dessa utdrag är databasen i sitt gamla skick. När man däremot vill skapa tabeller är detta en ingrep i databasen som gör ändringar. Därför har man i SQL en helt annan grupp av satser med befogenheten att inte bara kunna läsa från (read-only) utan även kunna *skriva* i databasen. En av dem är **`CREATE TABLE`**-satsen.

`CREATE TABLE`-satsen tillhör gruppen *Data Definition Language (DDL)* i SQL. Ett exempel på hur man skriver den ser man på bilden ovan. Denna sats skapar en tabell som heter **`Kurser`** med kolumnerna **`KursID`**, **`Namn`**, **`Längd`** och **`InstID`**. Varje kolumn måste få en datatyp tilldelad när man definierar tabellen. I exemplet ovan har kolumnerna **`KursID`**, **`Längd`** och **`InstID`** datatypen **`INT`** och kolumnen **`Namn`** datatypen **`VARCHAR(50)`** vilket betyder text av längden max **`50`** tecken. **`NULL`** betyder ingen information dvs tom cell i tabellen. Vissa kolumner tillåts att ha tomma celler, andra ine (**`NOT NULL`**).

Identity

Dessutom ska kolumnen **`KursID`** vara **`Identity`**. I *Microsoft SQL Server* kallas den kolumn som ska automatiskt få en sekvens av löpande nummer som värden för **`Identity`**. Det är inte samma sak som primärnyckel, utan endast en automatisk numrering av raderna med ett startvärde *(Identity Seed)* och ett steg *(Identity Increment)*. **`Identity(1,1)`** betyder att startvärdet och steget ska ha värdena 1. En konsekvens av detta är att du numera inte får ge denna kolumn några värden själv, när du lägger in rader i tabellen, eftersom den får sina värden automatiskt pga **`Identity`**-egenskapen. I andra databassystem, t.ex. i *Oracle*, heter denna egenskap *sekvens* (Sequence) och är ett eget databasobjekt.



När vi skrev våra exempel på SQL-satser förklarade vi inte varför vi skrev dem just i den form vi gjorde. Här kommer några regler och konventioner om SQL-satsernas form (layout). Observera skillnaden mellan *regler* och *konventioner*. Regler måste följas, annars kan man inte exekvera koden. Konventioner är rekommendationer som är till för att strukturera koden på bäst möjliga sätt, så att den blir optimal ur läsbarhets- och förståelighetssynpunkt. Konventioner tillhör god programmeringsstil. Koden kan exekveras även utan att man följer dem.

Till skillnad från de flesta programmeringsspråken är SQL inte case sensitive, dvs det spelar ingen roll om man skriver de reserverade orden med stora eller små bokstäver: **`select`** fungerar lika bra som **`SELECT`**. Ändå ges rekommendationen att skriva **`SELECT`** av den enkla anledningen att bättre kunna skilja mellan SQL-reserverade ord å ena och databasens objekt som tabell-, kolumn- och andra namn å andra sidan.

Till skillnad från de flesta programmeringsspråken behöver man inte avsluta en SQL-sats med semikolon. Ändå ges rekommendationen att göra det. Den här rekommendationen har kanske inte lika stark skäl som förra. Ett skäl kan vara att skilja mellan sats*delar* och satser. Ett annat skäl är att skilja mellan olika satser, vilket inte förekommer bland våra exempel, men blir påtagligt när man skriver ett s.k. *script* bestående av flera SQL-satser. I enlighet med andra programmeringsspråk finns det ingen regel för radbrytning varken mitt i en sats eller mellan olika satser. Koden i alla våra exempel skulle kunna exekveras om vi skrev den på en enda rad. Men för att strukturera koden och optimera läsligheten samt förståeligheten rekommenderas att påbörja en ny satsdel med en ny rad.

Det finns mycket mer att säga om SQL i allmänhet och om **`SELECT`**-satsen i synnerhet, men vi sätter punkt här för att återvända till C# och börja använda SQL med C#.

5.4 Vår första SQL Server databas

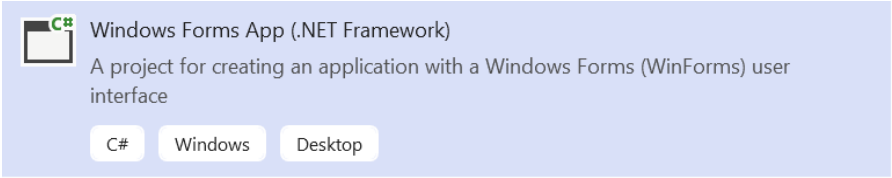
Efter de inledande avsnitten om databaser och SQL är det dags att bygga vårt första C#-projekt som ansluter sig till en databas och utför enkla operationer: att ansluta sig till SQL Server som följde med vid installationen av Visual Studio, att öppna databasen och att visa tabellerna osv. Allt detta ska dessutom göras via ett grafiskt gränssnitt i Visual Studio. Dvs vårt första databasprojekt blir en Windows Forms Application. I det här avsnittet kommer vi att lära oss att:

- ladda en databas till en C# Windows Forms Application, etablera kontakt med den och använda den som en datakälla,
- visa databasens tabeller i en DataGridView-kontroll,
- kunna med SQL lägga till och ta bort rader från tabellerna samt spara tabellernas nya tillstånd via det grafiska gränssnittet.

Som datakälla kommer vi att använda oss av en exempeldatabas som är lagrad i filen **Books.mdf**. Filändelsen **mdf** står för Microsoft SQL Server database file, ett filformat som Microsoft använder för fysiska databasfiler. Ändelsen visar att filen är genererad i SQL Server och kan därför endast öppnas och läsas i SQL Server. Vi kommer att använda den från C# genom att ansluta oss till SQL Server. Du kan ladda ner filen från webbsidan www.taifun.se. Klicka där på bokomslaget *Programmering 2 med C#*, skrolla ned och leta efter länken **Books.mdf**, klicka på den för att ladda ned den. Extrahera sedan zip-filen. Gör så här:

Steg 1: Att skapa ett projekt av typ Windows Forms Application

Öppna Visual Studio och välj Create a new project. Bland de många typer av projekt (templates) som visas, välj följande variant av en Windows Forms Application:



Markera denna ruta och klicka på Next-knappen. Dialogrutan Configure your new project dyker upp. Döp projektet till FirstDatabase. Ange Location, bocka för rutan Place solution … och klicka på Create-knappen. Ett grafiskt gränssnitt dyker upp med ett s.k. *formfönster* i mitten som har rubriken Form1. Formfönstret har ett antal egenskaper som är samlade i fönstret Properties i det nedre högra hörnet. Om du inte ser Properties-fönstret kan du få fram det genom att från menyraden välja View → Properties Window. För att enklare hitta egenskaperna, ordna dem i alfabetisk ordning med ikonen Alphabetical till höger:



Markera formfönstret och leta i Properties-fönstrets vänstra kolumn efter egenskapen Text. Dess defaultvärde kan avläsas i den högra kolumnen, just nu: Form1. Markera det och ändra det till FirstDatabase. Så snart du gjort detta syns den nya

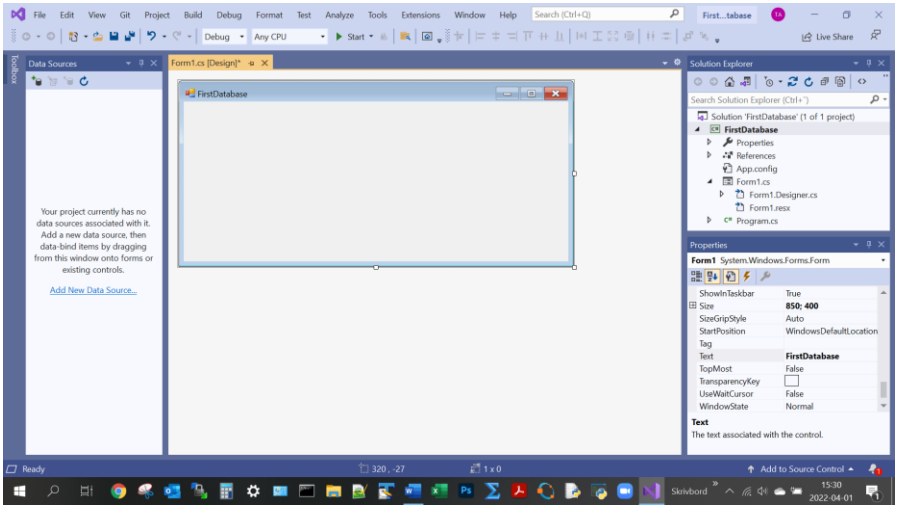
texten på formfönstrets rubrik. Gå vidare till egenskapen Size i Properties-fönstret och sätt storleken till 850; 400, så här:

Form1:

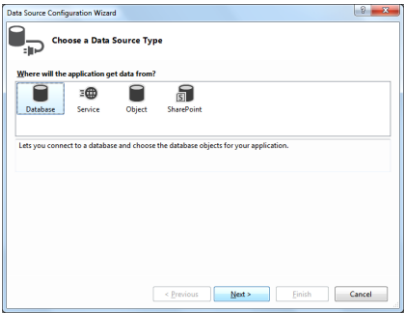
Egenskap	Värde
Text	FirstDatabase
Size	850; 400

Steg 2: Att koppla upp sig till SQL Servern

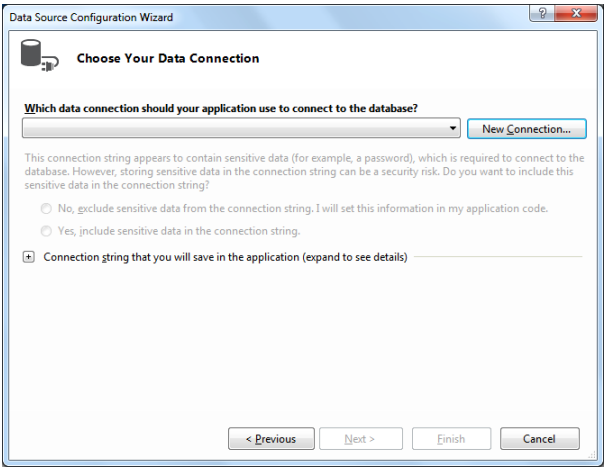
Här vill vi infoga exempeldatabasen **Books.mdf** i vårt projekt FirstDatabase. För att göra det behöver vi ett nytt fönster som heter Data Sources. Gå i menyraden till ett textfält längst till höger som det står Search i och skriv Data Sources i det. Klicka på den lilla triangeln ovan på rubrikraden och välj Dock för att fästa det i Visual Studio. Vid det här läget borde ditt fönster i Visual Studio se ut så här:



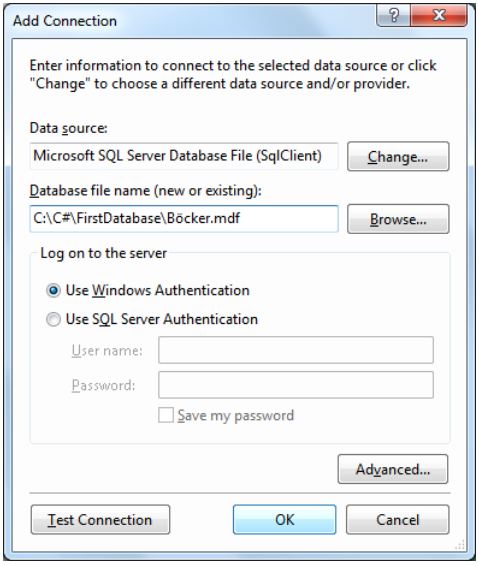
Gå till det nya fönstret Data Sources och klicka på länken Add New Data Source.... Du får följande dialogruta som frågar efter typen av datakälla som du vill använda i projektet:



- Markera Database under frågan Where will the application get data from? Klicka på Next.
- Nästa dialogruta frågar efter typen av databasmodell (visas inte här). Markera Dataset och klicka på Next.
- Nästa dialogruta frågar efter databasen som ditt projekt ska kopplas till. Klicka på knappen New Connection… :



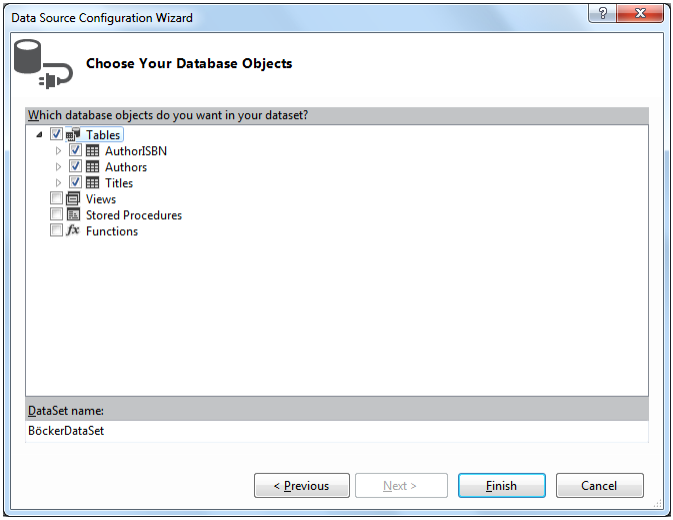
- Du får ytterligare en dialogruta som heter Choose Data Source (visas inte här). Välj som Data source: Microsoft SQL Server Database File. Klicka på knappen Continue. Här kan det vara att du måste installera packages för SQL Server Support. Om det är så, gör det. Dialogrutan Add Connection dyker upp. Klicka i den på Browse-knappen och navigera genom filsystemet på din dator för att ladda databasfilen **Books.mdf** till projektet. Klicka på OK. Det kan vara att Visual Studio vill uppgradera databasfilen så att den blir kompatibel med din aktuelle version av Visual Studio. I så fall svara bara ja.



- Du återvänder till dialogrutan Choose Your Data Connection, bara att det nu har tillfogats namnet på databasfilen. Klicka på Next. Svara Ja på frågan om du vill kopiera filen till projektet.

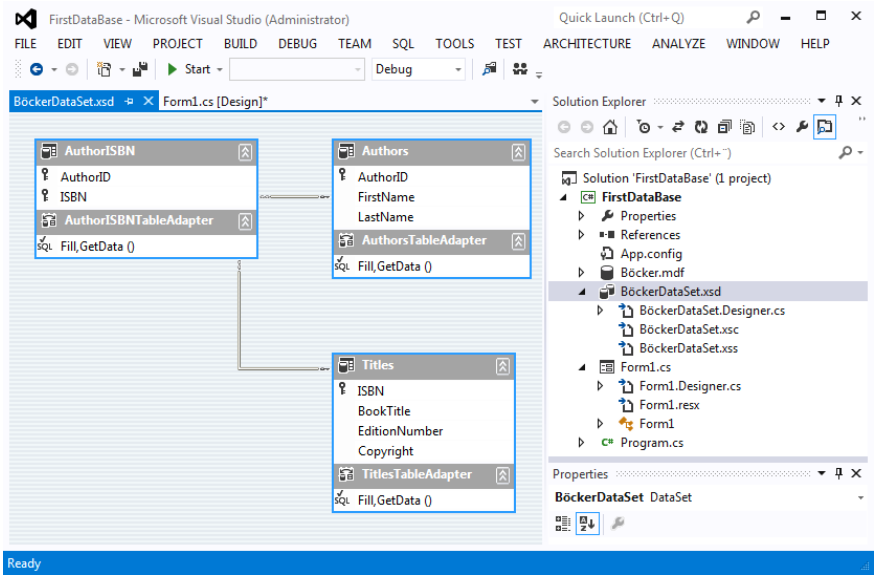
- I nästa dialogruta som heter Save the Connection String to the Application Configuration File är namnet BooksConnectionString redan förvalt för den förbindelse du skapade ovan. Bocka för lilla rutan Yes, save the connection as: (om den inte redan är förbockad) och klicka på Next.

- I nästa och sista dialogruta som du ser på nästa sida ska du välja de delar av databasen, s.k. *databasobjekt* som du vill använda i ditt projekt. Vår databas i filen **Books.mdf** har bara tabeller. Så bocka den lilla rutan vänster om Tables och expandera Tables med den lilla pilen till vänster Du får en första inblick i databasens innehåll.



Som man ser har databasen tre tabeller: AuthorISBN, Authors och Tiltles. Klicka på Finish.

Du återvänder till projektets ursprungliga miljö. Men nu ser man databasens struktur i fönstret Data Sources till vänster och det har kommit till i Solution Explorer databasfilen Books.mdf som en del av projektet. Dessutom har Visual Studio skapat bl.a. ett s.k. *XML Schema document* med namnet BooksDataSet.xsd. Markera det, högerklicka och välj View Designer för att se databasens struktur i ett diagram som kallas för **DataSet Designer:**



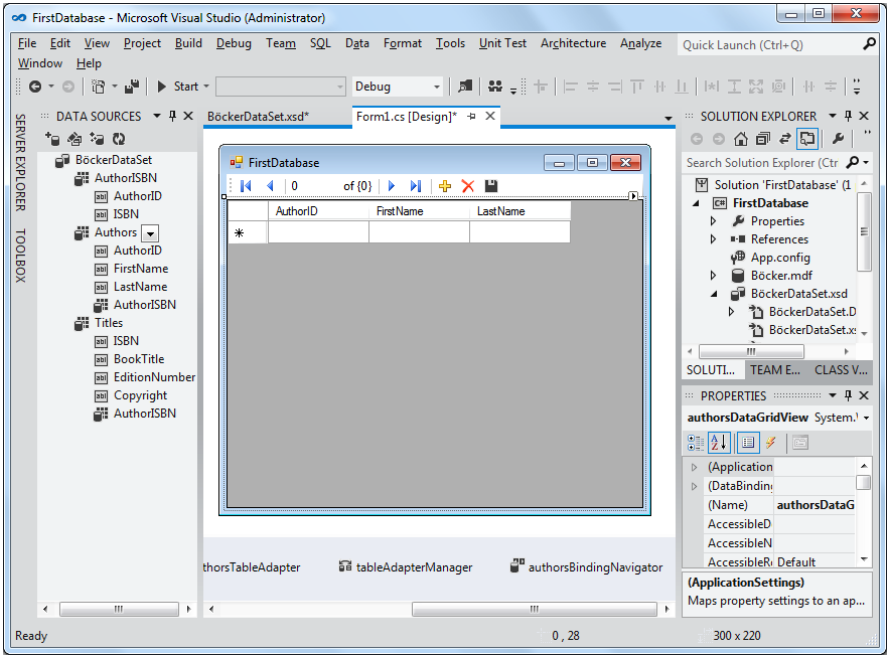
Diagrammet **DataSet Designer** (på förra sidan) visar 3 tabeller: Varje ruta representerar en tabell. Överst står tabellernas namn, under dem kolumnerna. De kolumner som är markerade med en nyckel är tabellens nycklar. I tabellen Authors är kolumnen AuthorID primärnyckeln. I tabellen Titles är kolumnen ISBN primärnyckeln. Tabellen AuthorISBN:s båda kolumner är däremot främmande nycklar. Dessutom visar diagrammet även relationerna mellan databasens tabeller. De är ritade med linjer försedda med pilar. Läs om relationer och primär- och främmande nycklar på sid 164-167 / 168.

Steg 3: Att visa databasens innehåll

Diagrammet ovan och fönstret Data Sources visar databasens innehåll: Vi ser t.ex. att det finns en tabell som heter Authors. Vi vill titta i den. För att öppna tabellen Authors och visa dess rader och kolumner gör så här:

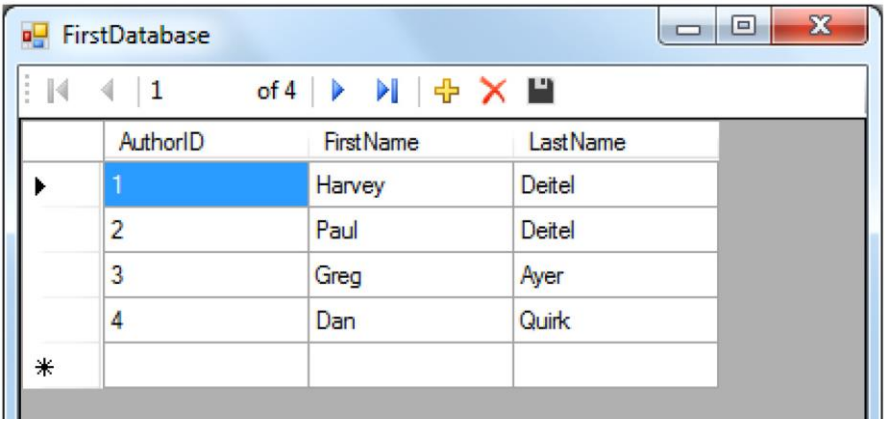
- Återvänd till din form Form1 via fliken eller genom att i Solution Explorer markera Form1.cs, högerklicka på den och välja View Designer.

- Gå till det nya fönstret Data Sources på vänstersidan, markera tabellen Authors och dra den med musen (genom att hålla ned den vänstra musknappen) till formen. Därmed har du skapat två nya s.k. *kontroller* i formen: Den ena heter authorsBindingNavigator och är en rad med ett antal navigeringsmenyer som lägger sig direkt under formens rubrik. Den andra heter authorsDataGridView och är en plats där tabellen Authors' innehåll kommer att visas. Klicka på authorsDataGridView:s *Smart Tag* (lilla pilen till höger ovan) och klicka på Dock in parent container, så att authorsDataGridView fyller hela formen. Så här borde resultatet bli:



Som man ser har den nya kontrollen authorsDataGridView samma kolumner som tabellen Authors, nämligen AuthorID, FirstName och LastName. Observera också att authorsDataGridView inte än visar tabellens innehåll utan förbereder denna visning genom att skapa en plats som är anpassad till tabellens struktur.

Allt vi gjort hittills skedde i designläge. Vi har inte använt en enda rad kod. Ändå är programmet nu redo att visa tabellens innehåll när vi nu går över från design- till körläge. Kompilera från menyraden med → Build → Build Solution och exekvera med → Debug → Start Without Debugging. Tabellen Authors har fyra rader och tre kolumner och ser ut så här:



Nu kan vi i körläge använda menyerna under formens rubrik för att hantera tabellen. I själva verket är det kontrollen authorsBindingNavigator:s menyer vi använder. Med hjälp av dessa självinstruerande knappar kan vi:

- navigera genom tabellens rader
- ändra radernas innehåll
- lägga till nya rader (med **+**)
- ta bort rader (med **x**)
- spara dina ändrigar

I början sa vi att detta projekt genomförs med Visual Studios visuella verktyg utan att vi själva behöver skriva någon kod. Så har vi också gjort. Men det betyder inte att programmet fungerar helt utan kod. Det har skapats automatiskt genererad kod som ligger i filen Form1.cs: Två händelsemetoder har lagts till klassen **`Form1`**. Den ena heter **`Form1_Load()`** och ser till att, när formen laddas, dvs när programmet exekveras, data kopieras från databasfilen till projektets DataSet, så att vi ser tabellens innehåll i DataGridView-kontrollen. Den andra händelsemetoden heter authorsBindingNavigatorSaveItem_Click() och ser till att data sparas i DataSet när man klickar på Save-menyn i nya kontrollen authorsBindingNavigator. Allt detta hände när vi skapade de två nya kontrollerna authorsDataGridView och authorsBindingNavigator i vårt projekt genom att med musen dra tabellen Authors från fönstret Data Sources till formen.

5.5 En SQL klient i C#

I det här avsnittet kommer vi att lära oss att:

- skicka SQL-frågor från en ComboBox-kontroll till databasen Books,
- visa frågornas resultattabell i en DataGridView-kontroll.

Steg 1: Att skapa projektet och förse det med databasen Books

- Detta steg liknar ***Steg 2*** i projektet ***FirstDatabase***. Öppna Visual Studio, skapa en Windows Forms Application och döp projektet till SQLclient. Ändra formfönstrets rubrik och storlek enligt följande:

  Form1:

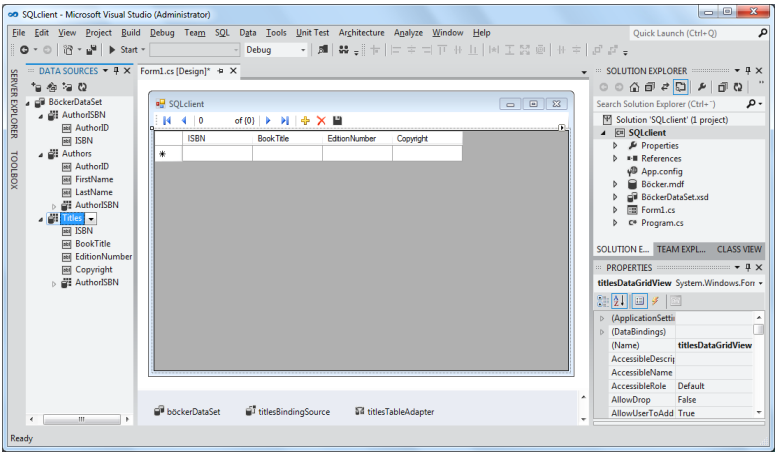
Egenskap	Värde
Text	En SQL klient
Size	1200; 600

- Genomför de steg från projektet FirstDatabase som var nödvändiga för att ladda databasen Books.mdf till projektet (sid 181-184), dvs förkortat:

- I huvudmenyraden: Search → Data Sources → Dock.
- Klicka i det nya fönstret Data Sources på ikonen Add New Data Source.
- Choose a Data Source Tye: Database → Next.
- Choose a Database Model: Dataset → Next.
- Choose Your Data Connection: New Connection…
- Add Connection: Browse → Books.mdf →… OK → Next.
- Choose Your Data Connection → Next → Ja.
- Save the Connection String to the Application Configuration File → Yes, save the connection as: → Next.
- Choose Your Database Objects → Expand Tables → Finish.

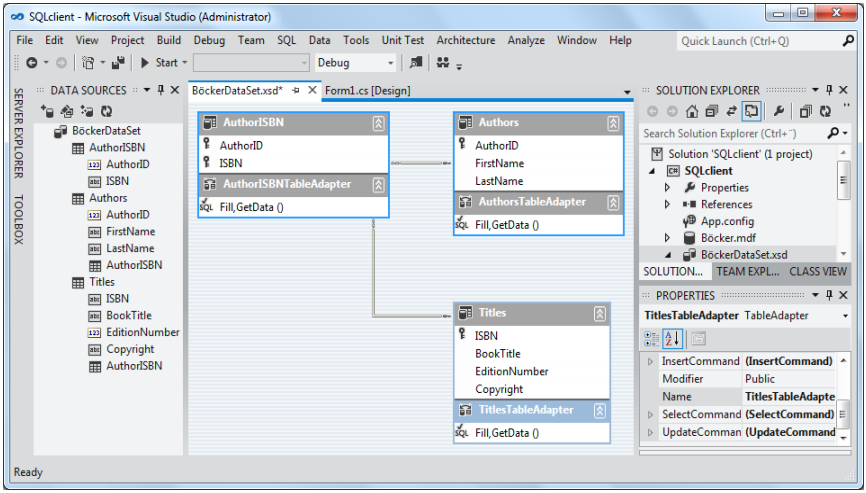
Du återvänder till projektets ursprungliga miljö med formfönstret osv. Databasen Books har infogats i projektet. Gå till Data Sources och expandera BooksDataSet för att se databasens innehåll: tre tabeller med sina resp. kolumner.

- Markera tabellen Titles i fönstret Data Sources och dra den med musen till formen. Så skapar du kontrollerna titlesBindingNavigator och titlesDataGridView som placeras i formen. Klicka på titlesDataGridView:s *Smart Tag* och klicka på Dock in parent container. Så här borde din miljö nu se ut.



- Markera i Solution Explorer BooksDataSet.xsd, högerklicka på det och välj View Designer. Du ser databasen Books:s struktur i ett diagram med alla tabeller, relationer, primär- och främmande nycklar osv. Den här visuella representationen av databasen, DataSet Designer, känner vi till från tidigare projekt.

DataSet Designer:



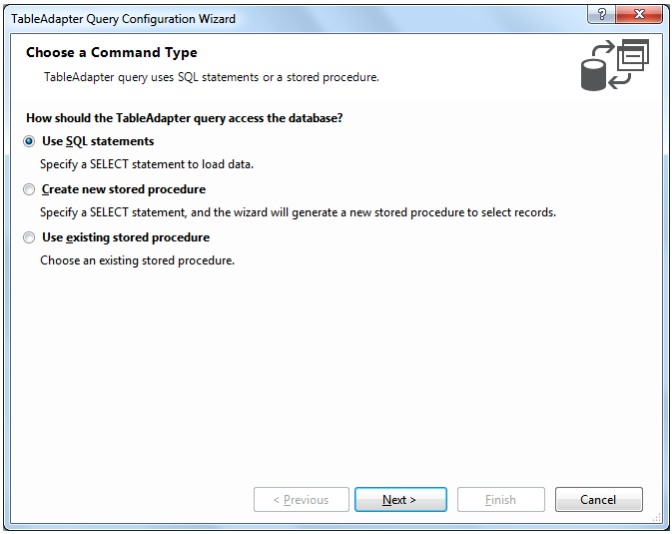
Diagrammet visar i slutet av varje tabellruta en s.k. TableAdapter. Det är en *klass* som automatiskt genereras av Visual Studio till varje tabell. Klassen TableAdapter har bl.a. en *metod* Fill() som anropas i händelsemetoden **`Form1_Load()`** som i sin tur anropas när formen laddas. Och detta sker när vi exekverar programmet. Koden finns i filen Form1.cs.

Kompilera och kör projektet nu för att se *hela* tabellen Titles' innehåll.

Steg 2: Att skriva och exekvera egna SQL satser

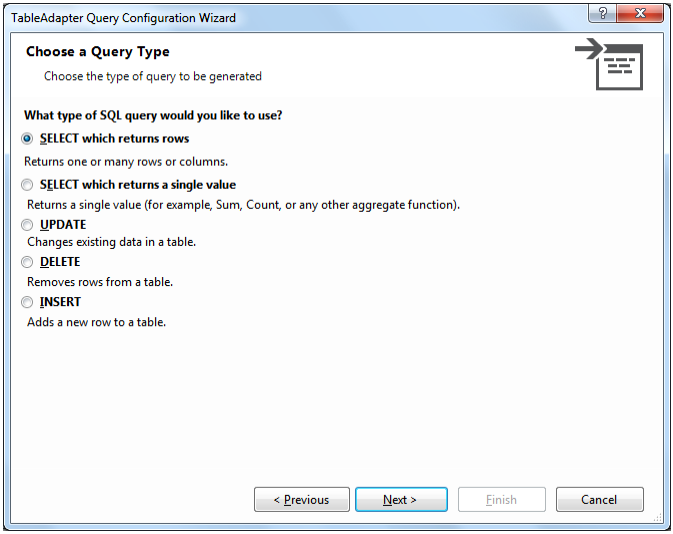
Om vi nu vill med SQL-frågor selektera och visa endast vissa delar av tabellen Titles måste vi lägga till egna metoder till klassen TableAdapter och formulera våra SQL-frågor i dem. Det är tekniken att skicka SQL satser till servern. Gör så här:

- Markera i diagrammet DataSet Designer på förra sidan, i rutan som representerar tabellen Titles, klassenTitles' TableAdapter, högerklicka och välj Add Query… . Dialogrutan Choose a Command Type öppnas:



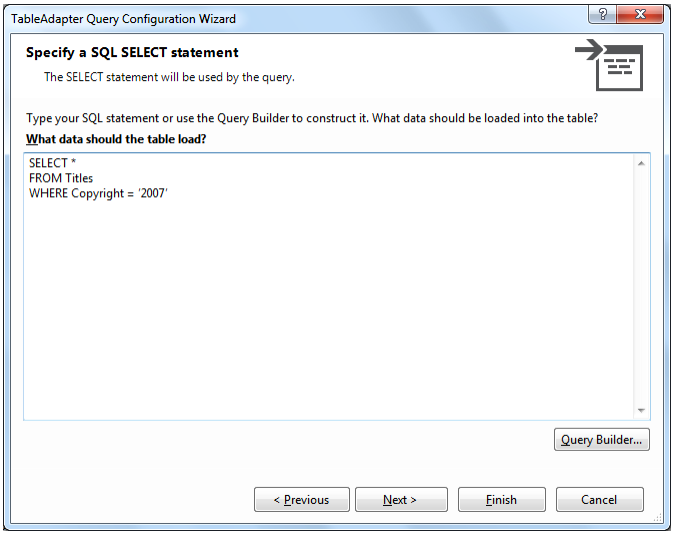
Välj alternativet Use SQL Statement och klicka på Next.

- Nästa dialogruta: Choose a Query Type.
  Välj SELECT which returns rows och klicka på Next.

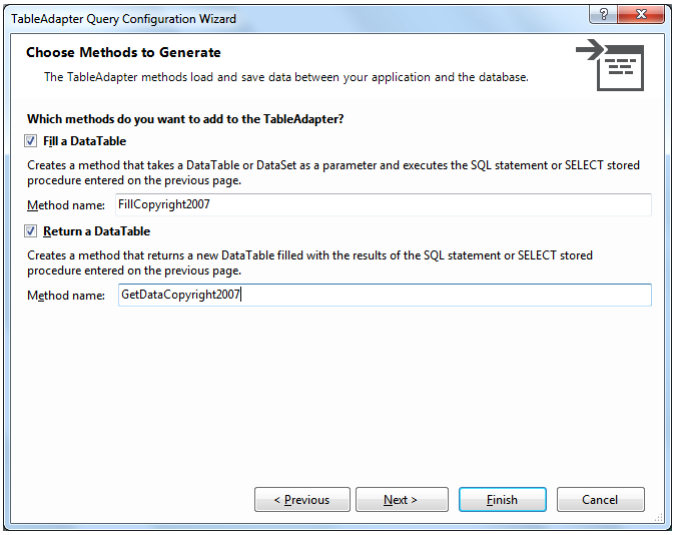


Nästa dialogruta heter Specify a SQL SELECT statement. Skriv in följande SQL-fråga i textfältet med rubriken What data should the table load? och klicka på Next (OBS! inte på finish!):

```
SELECT *
FROM Titles
WHERE Copyright = '2007';
```



- Dialogrutan Choose Methods to Generate dyker upp:



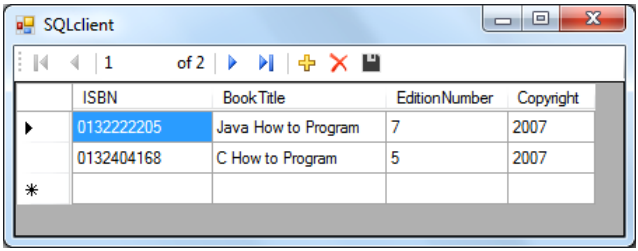
Här måste vi specificera de metoder som ska exekvera vår SQL-fråga. Vi vill använda våra egna metoder. Därför ändrar vi de förvalda namnen FillBy till FillCopyright2007 och och GetDataBy till GeDataCopyright2007.

Klicka nu på finish för att återvända till diagrammet DataSet Designer. De två nydefinierade metoderna har nu kommit till rutan som visar tabellen Titles, längst ned under TitlesTableAdapter. Kompilera och kör: Fortfarande ser man *hela* tabellen Titles.

- Nu ska vi *exekvera* den nya SQL satsen. Markera Form1.cs i Solution Explorer, högerklicka och välj View Code för att se formens kod. Sist bland klassens **`Form1`**:s metoder finns händelsemetoden **`Form1_Load()`**. Ersätt anropet av metoden Fill() i den med anropet av den nya metoden FillCopyright2007(). Så här blir då **`Form1_Load()`**:s fullständiga kod:

```
private void Form1_Load(object sender, EventArgs e)
{
    this.titlesTableAdapter.FillCopyright2007
                            (this.booksDataSet.Titles);
}
```

- Kompilera och kör. Här är resultatet av den nya SQL satsen:



Som man ser visas endast två böcker med värdet 2007 i Copyright-kolumnen pga att vi i formens kod hade ersatt metoden **`Fill()`** med metoden **`FillCopyright2007()`**.

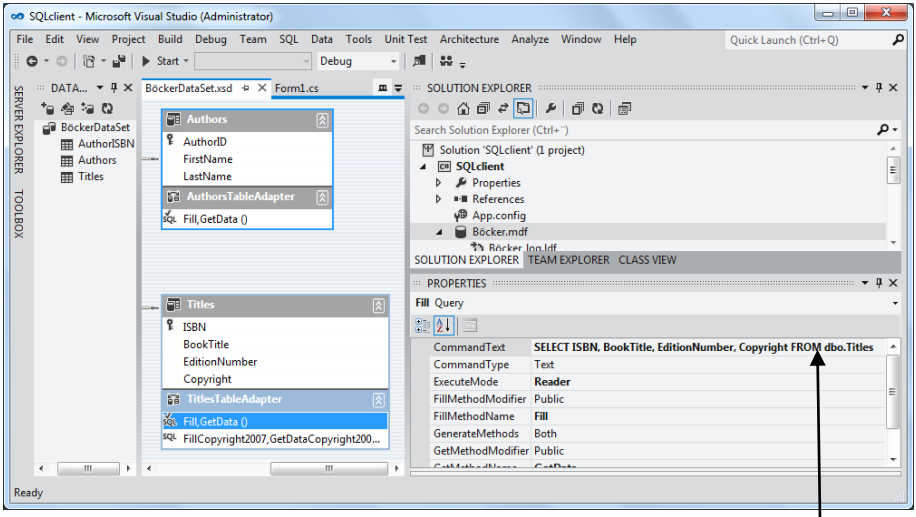
- Testa även anropet av metoden **`Fill()`** istället för metoden **`FillCopyright2007()`**:

```
private void Form1_Load(object sender, EventArgs e)
{
this.titlesTableAdapter.Fill(this.booksDataSet.Titles);
}
```

Resultatet blir att man får tabellen Titles' fulla innehåll dvs alla rader.

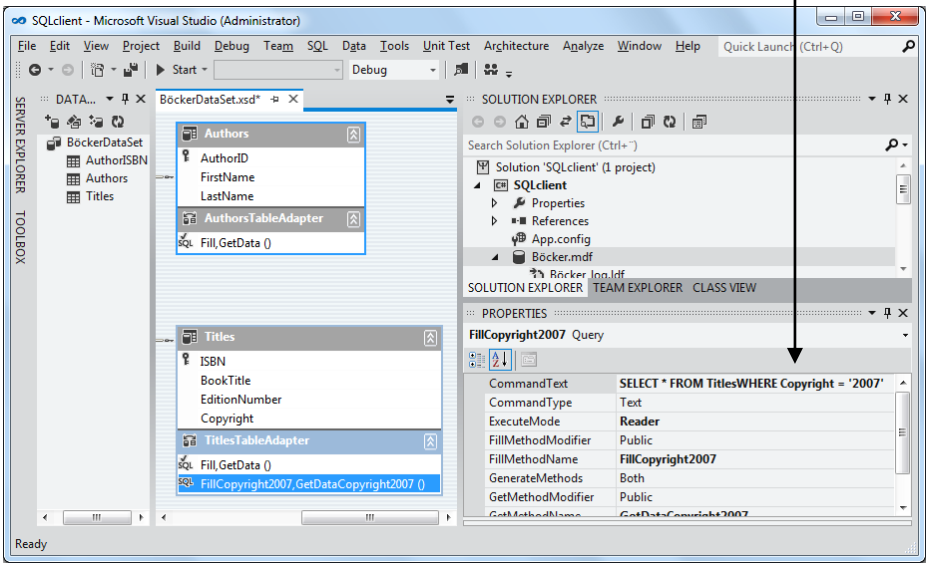
Hur vet man att metoden **`Fill()`** exekverar den SELECT-sats som visar alla rader, och metoden **`FillCopyright2007()`** exekverar den SELECT-sats som endast visar de rader med värdet 2007 i Copyright-kolumnen? Vi kan få reda på det om vi gör så här:

Markera i Solution Explorer BooksDataSet.xsd, högerklicka, välj View Designer:



Markera i rutan som visar tabellen Titles, längst ned under TitlesTableAdapter, raden Fill,GetData() så får du i Properties-fönstret metoden Fill():s egenskaper. I egenskapen CommandText kan du läsa den SELECT-sats som är kopplad till Fill().

Om du i samma ruta markerar raden FillCopyright2007,GetDataCopyright2007() kan du i Properties-fönstret läsa SELECT-satsen i metoden FillCopyright2007():



Eftersom vi i det här databasprojektet för första gången har tillfogat lite kod till de visuella verktyg som byggde projektet vill vi här sammanfattningsvis visa den viktigaste delen av kod som vid sidan av den stora mängden automatiskt genererad

kod, styr exekveringen av projektet och som vi har modifierat lite grann. Denna kod finns i filen Form1.cs. Du får fram den genom att markera Form1.cs i Solution Explorer, högerklicka och välja View Code. För enkelhetens skull har vi tagit bort all onödig automatiskt genererad kod och behållit det som behövs för detta projekt:

```
// Form1.cs i projektet SQLclient, ver 1. Visar data från
// en databastabell i en DataGridView-kontroll.
// Klassen Form1 ärver klassen Form från System.Windows.Forms
// Deklarerar tre metoder: en konstruktor & 2 händelsemetoder
using System;
using System.Windows.Forms;

namespace SQLclient
{
   public partial class Form1 : Form  // Form1 ärver Form
   {
      public Form1()                  // Klassens konstruktor
      {
         InitializeComponent();
      }

      private void titlesBindingNavigatorSaveItem_Click(
                                 object sender, EventArgs e)
      {
         this.Validate();
         this.titlesBindingSource.EndEdit();
      this.tableAdapterManager.UpdateAll(this.booksDataSet);
      }

      private void Form1_Load(object sender, EventArgs e)
      {
         this.titlesTableAdapter.FillCopyright2007
                                 (this.booksDataSet.Titles);
      }
   }
}
```

Klassen Form1 definierar tre metoder: Den första metoden **`Form1()`** är klassens konstruktor som initierar formens grafik. Den andra metoden **`titlesBindingNavigatorSaveItem_Click()`** är en händelsemetod som definieras här och anropas när Save-knappen i kontrollen titlesBindingNavigator klickas. Då sparas alla gjorda ändringar i projkektets DataSet. Den tredje metoden **`Form1_Load()`** är också en händelsemetod som definieras här, men anropas när formen laddas. Och formen laddas när vi exekverar projektet. I metoden **`Form1_Load()`** anropas i sin tur metoden **`FillCopyright2007()`** som vi lagt in där. Den exekverar SELECT-satsen som vi skrev in i dialogrutan Specify a SQL SELECT statement, nämligen:

```
SELECT *
FROM Titles
WHERE Copyright = '2007';
```

Steg 3: Att lägga till ett grafiskt gränssnitt till SQL klienten

Hittills är detta projekt inte särskilt intressant ur praktisk synpunkt. Det var mer lämpat för att lära känna de mest grundläggande rutinerna i hanteringen av en databas. Man förväntar sig lite mer av en ”SQL klient”, framför allt en smidigare kommunikation mellan klienten C# och SQL Servern. För att åstadkomma detta ska vi nu vidareutveckla projektet och förse det med två nya grafiska komponenter. En av dem är en kontroll som heter ComboBox som kommer att tjäna som en plats där vi från en dropplista kan så att säga online välja våra SQL-frågor och skicka dem till SQL Servern. Den andra är en kontroll som heter Label som instruerar användaren. Svaret från servern ska precis som hittills visas i den DataGridView-kontroll som vi skapat och redan använt i den första delen av projektet.

Den nya kontrollen ComboBox är en dropplista där användaren kan välja mellan olika alternativ. Den kräver lite mer kod som ska avgöra vilket alternativ användaren valt just vid den aktuella körningen för att kunna exekvera rätt SQL-sats. Vi kommer att realisera detta genom att skriva en **`switch`**-sats ”bakom” den grafiska komponenten ComboBox.

Gör så här:

- Återvänd till formfönstret Form1 genom att i Solution Explorer högerklicka på Form1.cs och välja View Designer.
- Gå till huvudmenyraden, klicka på View och välj Toolbox. Fönstret Data Sources till vänster ersätts med Toolbox-fönstret. Expandera Toolboxens Common Controls. Markera kontrollen ComboBox, dra den med musen (genom att hålla ned den vänstra musknappen) till formen och lägg den längst ned i formen. Genomför i Properties-fönstret följande ändringar i den nya ComboBox-kontrollens egenskaper:

  comboBox1:

Egenskap	Värde
Location	0; 515
Size	1180; 28

- Markera formen, hämta från Toolbox en Label-kontroll till formen och gör i Properties-fönstret följande ändringar i den Labelns egenskaper:

  label1:

Egenskap	Värde
Location	400; 485
Text	Välj en SQL-fråga från dropplistan:
Font	Arial; 12pt; style=Bold

- Markera kontrollen comboBox1 och klicka på dess *Smart Tag* (den lilla pilen till höger). Välj Edit Items, skriv in följande texter i dialogrutan String Collection Editor och klicka på OK:

```
SELECT * FROM Titles;
SELECT * FROM Titles WHERE Copyright = '2007';
SELECT * FROM Titles WHERE Copyright = '2009';
SELECT * FROM Titles WHERE EditionNumber > 4;
SELECT * FROM Titles ORDER BY BookTitle;
```

- Dubbelklicka på ComboBox-kontrollen när den är markerad i formen. Filen Form1.cs visas där huvudet till en händelsemetod automatiskt skapats som heter **`comboBox1_SelectedIndexChanged()`**. Den kommer att anropas så snart man väljer resp. byter till ett alternativ i ComboBoxens dropplista. Lägg in följande **`switch`**-sats i metoden **`comboBox1_SelectedIndexChanged()`** vars kod är markerad med vit bakgrund:

```
// Form1.cs i projektet SQLclient
// Skickar SQL-frågor till en databas från en ComboBox
// Visar serverns svar i en DataGridView-kontroll
using System;
using System.Windows.Forms;
namespace SQLclient
{
  public partial class Form1 : Form
  {
    public Form1()
    {
      InitializeComponent();
    }

    private void titlesBindingNavigatorSaveItem_Click(
                                  object sender, EventArgs e)
    {
      this.Validate();
      this.titlesBindingSource.EndEdit();
      this.tableAdapterManager.UpdateAll(this.booksDataSet);
    }

    private void Form1_Load(object sender, EventArgs e)
    {
    }

    private void comboBox1_SelectedIndexChanged
                                 (object sender, EventArgs e)
    {
      switch (comboBox1.SelectedIndex)
      {
        case 0:
          titlesTableAdapter.Fill(this.booksDataSet.Titles);
          break;
```

```
            case 1: titlesTableAdapter.FillCopyright2007
                                        (this.booksDataSet.Titles);
                break;
            case 2: titlesTableAdapter.FillCopy2009
                                        (this.booksDataSet.Titles);
                break;
            case 3: titlesTableAdapter.FillEdNo4
                                        (this.booksDataSet.Titles);
                break;
            case 4: titlesTableAdapter.FillOrderBy
                                        (this.booksDataSet.Titles);
                break;
        }
    }
  }
}
```

Samtidigt ta bort *kroppen* till formens händelsemetod **`Form1_Load()`**, inte hela metoden. Detta därför att anropen av metoderna **`Fill()`** och **`FillCopyright-2007()`** är flyttade till ComboBoxens händelsemetod **`comboBox1_SelectedIndexChanged()`**, närmare bestämt till **`case 0`** och **`1`** av **`switch`**-satsen. Ingen SQL-sats ska exekveras när formen laddas, utan först när man väljer ett alternativ i ComboBoxens dropplista. I och med detta val tilldelas ComboBoxens variabel **`comboBox1.SelectedIndex`** ett av värdena **`0-4`**. Då kommer den metod att anropas i **`switch`**-satsen som svarar mot detta värde.

För att koden ovan ska fungera måste vi komplettera TitlesTableAdapter-klassens metoder med de metoder vi anropar i **`switch`**-satsen ovan. Därför gör så här:

- Återvänd till DataSet Designer. Markera i rutan som visar tabellen Titles, klassen TitlesTableAdapter, högerklicka och välj Add → Query… .
- Gå igenom de dialogrutor från projektets första del, förkortat:
- Choose a Command Type → Use SQL statements → Next.
- Choose a Query Type → SELECT which returns rows → Next.
- Skriv i dialogrutan Specify a SELECTstatement följande SQL-fråga i textfältet:

  ```
  SELECT *
  FROM Titles
  WHERE Copyright = 2009;
  ```

  Klicka på Next (OBS! inte på finish!). Dialogrutan Choose Methods to Generate dyker upp. Ändra de förvalda namnen FillBy till FillCopy2009 och GetDataBy till GetDataCopy2009. Klicka nu på finish för att återvända till DataSet Designer.

- Upprepa förfarandet: Markera TitlesTableAdapter i rutan som visar tabellen Titles, högerklicka och välj Add Query… . Gå vidare i de två följande dialogrutorna genom att klicka på Next.
- Skriv i dialogrutan Specify a SELECTstatement följande SQL-fråga i textfältet:

```
SELECT *
FROM Titles
WHERE EditionNumber > 4;
```

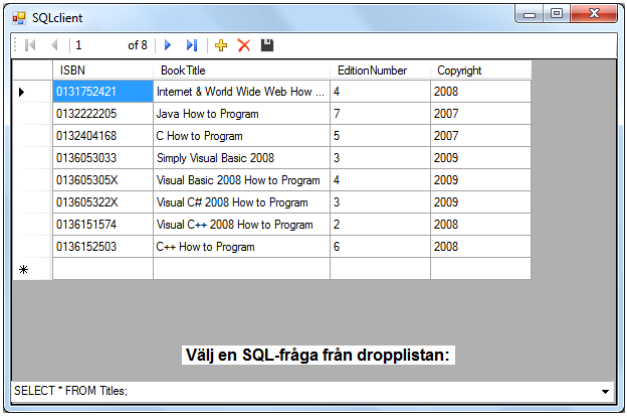
  Klicka på Next (OBS! inte på finish!). Dialogrutan Choose Methods to Generate dyker upp. Ändra de förvalda namnen FillBy till FillEdNo4 och GetDataBy till GetDataEdNo4. Klicka nu på finish för att återvända till DataSet Designer.

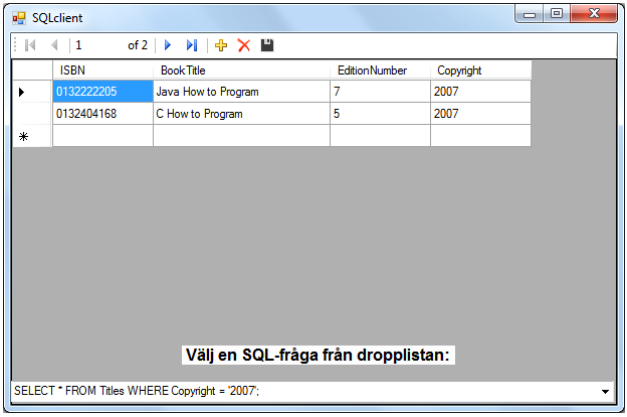
- Upprepa förfarandet: Markera TitlesTableAdapter i rutan som visar tabellen Titles, högerklicka och välj Add Query… . Gå vidare i de två följande dialogrutorna genom att klicka på Next.
- Skriv i dialogrutan Specify a SELECTstatement följande SQL-fråga i textfältet:

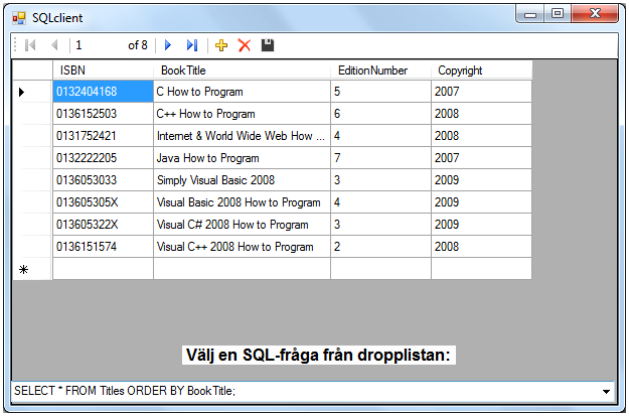
```
SELECT *
FROM Titles
ORDER BY BookTitle;
```

  Klicka på Next (OBS! inte på finish!). Dialogrutan Choose Methods to Generate dyker upp. Ändra de förvalda namnen FillBy till FillOrderBy och GetDataBy till GetDataOrderBy. Klicka nu på finish för att återvända till DataSet Designer.

- Kompilera och kör. Testa dina SQL-frågor från ComboBoxen. Så här kommer körresultaten med den 1:a, 2:a och 5:e SQL-satsen i i ComboBoxens dropplista att se ut:







Ytterligare två körresultat som inte visas här, kan fås med den 3:e och 4:e SQL-satsen i ComboBoxens dropplista.

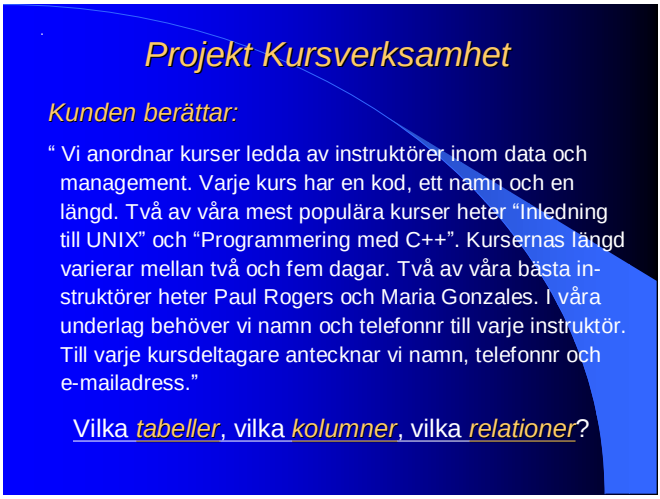
5.6 Att skapa och designa en databas i C#

Hittills har vi arbetat med den redan befintliga databasen Books.mdf. Men hur kommer en sådan databas till? En sak är ju att öppna och visa innehållet av en befintlig databas och skicka några SQL-satser till den och få svar. En annan sak är det att *skapa* en ny databas, att kanske t.o.m. *designa* den dvs ge den en struktur genom att ställa upp tabeller, bestämma tabellernas relationer, ange primär- och främmande nycklar bland tabellernas kolumner osv. Detta kräver kunskap om design och modellering av databaser. *Databasmodellering* och *-design* är ett ämne som har beröringspunkter med programmering – ganska liknande problemlösning med algoritmer, deras beskrivning med flödesschema och UML modellering. Modelleringsproblematiken finns alltid med och måste lösas *innan* vi fysiskt skapar databasen. Modellen av en databas måste finnas innan vi *implementerar* den genom att skapa tabeller, relationer, nycklar och andra databasobjekt.

I detta avsnitt ska vi designa en databas *och* implementera modellen genom att:

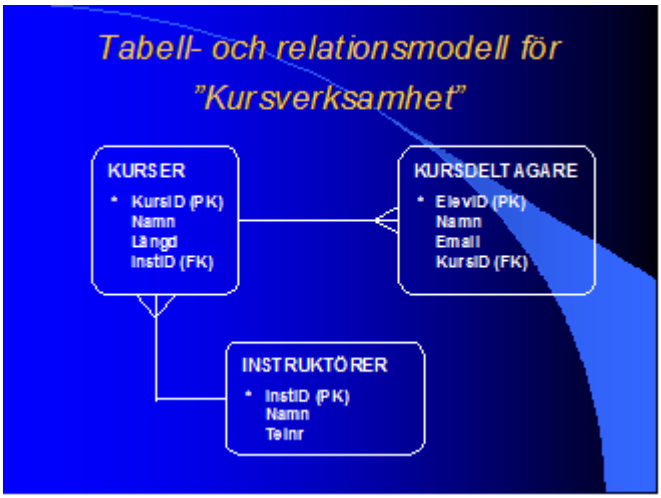
- skapa en tom databas i en C# Windows Forms Application, etablera kontakt med den och fylla den med tabeller,
- specificera tabellernas kolumner samt deras datatyper,
- definiera tabellernas primär- och främmande nycklar,
- bestämma relationer mellan databasens tabeller,
- fylla tabellerna med data.

För att uppnå dessa mål behöver vi en konkret fallstudie: Låt oss anta att vi har en kund som bedriver en kursverksamhet och vill datorisera sin verksamhet i form av en effektiv och stabil databas med vissa funktionaliteter. Vi går till ett första samtal och lyssnar på kundens behov. Så här lyder kundens kravspecifikation:



Databasmodellering

Tillbaka från kundsamtalet är vi helt ställda mot väggen: Hur ska vi skapa en databas som svarar mot kundens kravspecifikation? Men som tur är kommer vi ihåg vår kompis Kalle som har läst en kurs i *databasmodellering*. Vi mailar Kalle kundens beskrivning och får tillbaka följande diagram *(Kalles modell)*:

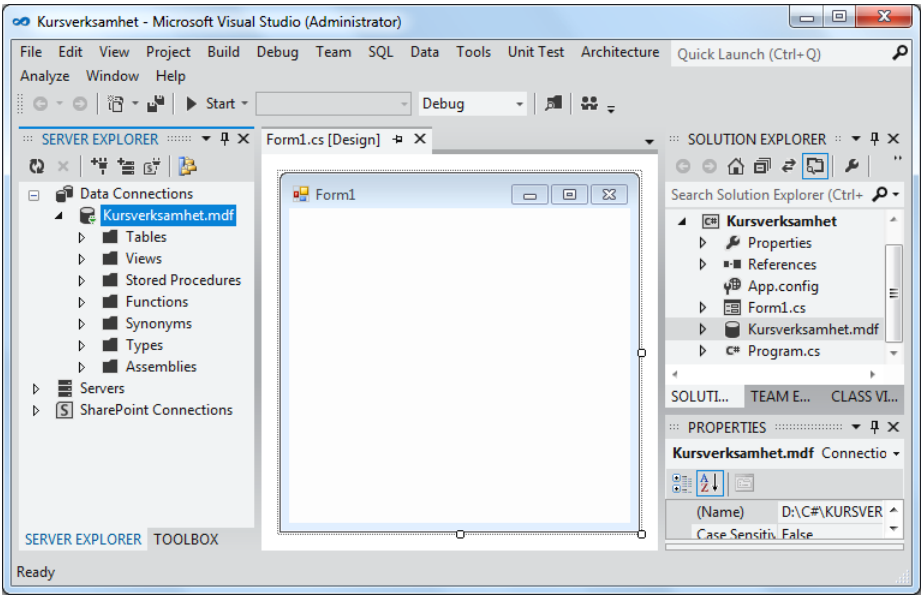


Kalle skriver att detta är ett s.k. ***ER****-diagram* där ***ER*** står för ***E****ntity-****R****elationship*. *ER-modellering* är en standard inom databasmodellering som lagrar all information i s.k. *entiteter*. En *entitet* är ett nyckelbegrepp, något viktigt för verksamheten – reellt eller virtuellt – som man behöver lagra information om – jämförbart med *klasser* i objektorienterad programmering. Kalle har utifrån kundens berättelse kommit fram till att entiteterna i detta projekt är KURSER, KURSDELTAGARE och INSTRUKTÖRER. Det är de som vi måste lagra information om. För varje entitet har Kalle ritat en ruta i diagrammet ovan. Han lägger till att vid implementeringen av modellen alla entiteter i modellen borde göras till *tabeller*. I varje entitets ruta står ett antal *attribut* dvs egenskaper som vid implementeringen ska bli *kolumner*. Kalle har även avgjort vilka kolumner som ska bli nycklar: PK (Primary Key) står för primärnyckel och FK (Foreign Key) för främmande nyckel. Enligt Kalles modell ska varje tabell ha en primärnyckel. Relationerna är ritade *mellan* tabellerna och de främmande nycklarna. Vi tar Kalles modell som en plan för att bygga en databas i C# för projektet Kursverksamhet.

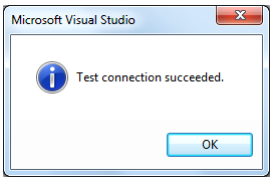
Steg 1: Att skapa databasen Kursverksamhet

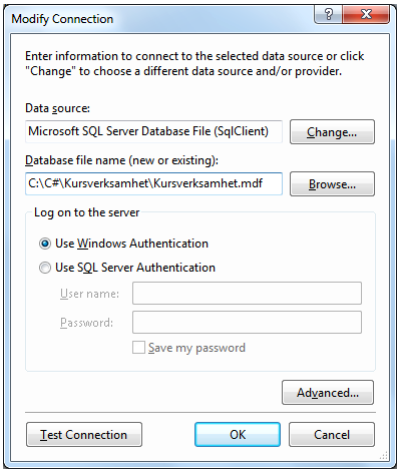
- Skapa en Windows Forms Application av typ C# Windows Forms App (.NET Framework) och döp den till Kursverksamhet.
- Gå till Solution Explorer, markera projektnamnet Kursverksamhet och högerklicka på det. Välj Add → New Item.... Dialogrutan Add New Item dyker upp. Scrolla ner den mellersta kolumnen och välj Service-based Data-

base. Skriv i textfältet Name: Kursverksamhet.mdf. Klicka på Add. Du har skapat en ny, **tom databas:** I Solution Explorer har kommit till lagringsfilen Kursverksamhet.mdf för den nya databasen. Markera den, högerklicka och välj Open.



- Ett nytt fönster öppnas i Visual Studio: Server Explorer. Dock fönstret. Högerklicka på Kursverksamhet.mdf i det nya fönstret och välj Modify Connection… . En ny ruta öppnas (till höger): Klicka på Test Connection för att kolla om du är ansluten till databasen. Om ja, meddelas: Test connection succeded. Klicka på OK i båda rutor.





Steg 2: Att skapa tabeller i databasen

- Markera Tables i Server Explorer och högerklicka på det. Välj Add New Table. En ny flik öppnas och fyller hela stora fönstret i mitten. Den heter dbo.Table[Design] och består av tre delfönster: I det undre delfönstret (fliken T-SQL), står SQL-kod som skapar en tabell. Den inleds på rad 1 med:

CREATE TABLE [dbo].[Table]

dbo står för database owner och sätts automatiskt framför tabellnamnet för att skilja mellan olika användares tabeller med ev. samma namn. Gå dit med musen och ersätt tabellnamnet med Kurser:

CREATE TABLE Kurser

Därmed har vi enligt Kalles modell (sid 199) döpt tabellen till Kurser.

- Fliken dbo.Table[Design] (vänster ovan) har nu döpts om till dbo.Kurser-[Design]. I den övre delen av den finns det en rubrikrad med Name, Data Type, Allow Nulls, Default och under den textfältet där vi kan mata in våra kolumners uppgifter.

- Enligt Kalles modell ska första kolumnen vara KursID: Ändra i textfältet under rubriken Name den redan befintliga texten Id till KursID. Gå vidare till textfältet under rubriken Data Type och välj int som datatyp om det inte redan står där. Tillåt i denna kolumn inga Null-värden, dvs inga tomma celler. Bocka därför inte Allow Nulls.

- Kolumnen KursID ska bli primärnyckel i tabellen Kurser. Nyckelsymbolen står redan till vänster om KursID.

- Dessutom ska kolumnen KursID vara Identity. I *Microsoft SQL Server* kallas den kolumn som ska automatiskt få en sekvens av löpande nummer för Identity. Det är inte samma sak som primärnyckel, utan en automatisk numrering av raderna med ett *startvärde* (Identity Seed) och ett *steg* (Identity Increment). Högerklicka i tabellens rubrikrad, t.ex. höger om rubriken Name, avbocka Default och bocka för Identity, Identity Seed och Identity Increment. Default tas bort och dessa tre tillfogas rubrikraden. Bocka för rutan under Identity. Låt både Identity Seed och Identity Increment ha värdet 1. Du får numera inte ge denna kolumn några värden själv, när du lägger in data i tabellen, eftersom den får sina värden automatiskt.

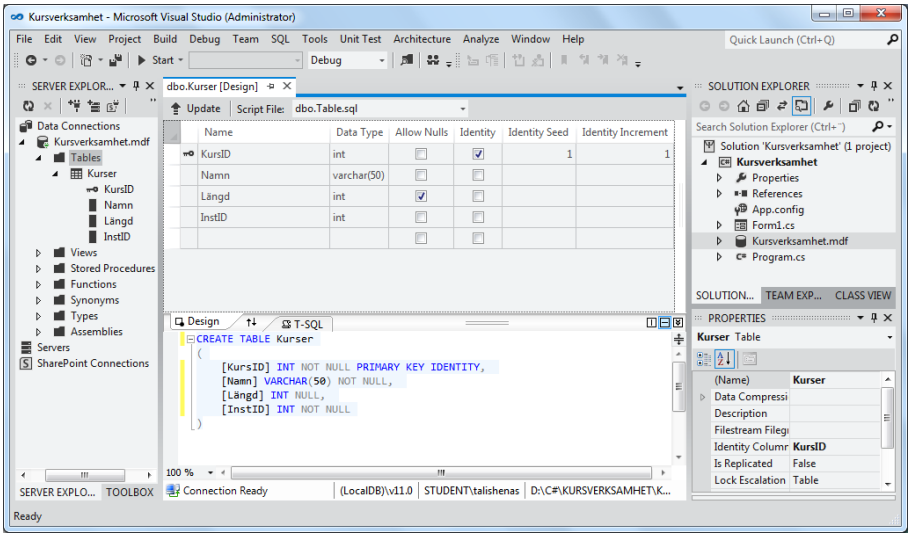
- Skapa ytterligare tre kolumner i tabellen Kurser: Namn, Längd och InstID, enligt Kalles databasmodell. Ge till Namn datatypen nvarchar(50), bocka annars för ingenting. Ge till Längd datatypen int, bocka för Allow Nulls. Ge till InstID datatypen int, bocka annars för ingenting.

Update:

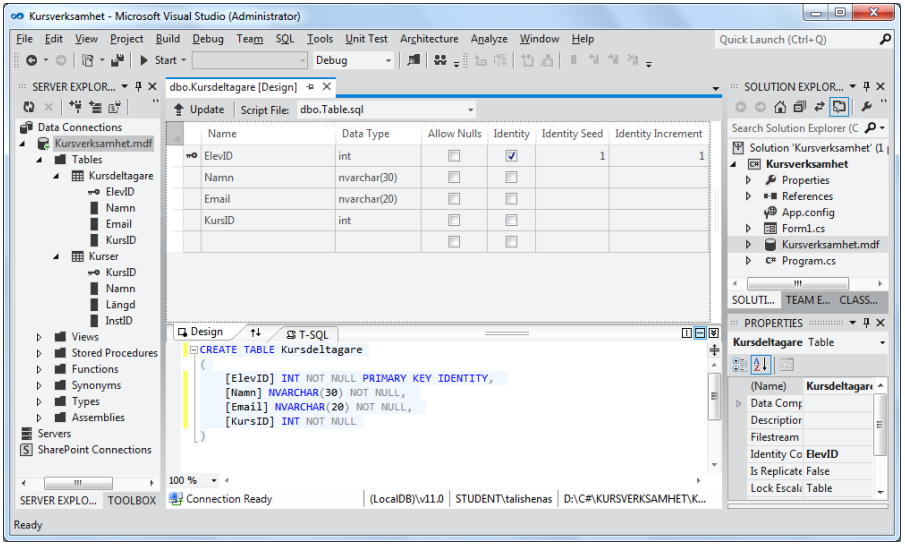
- Klicka på knappen Update (ovanför rubriken Name) för att spara allt i databasen. Bekräfta genom att klicka på Update Database. Om detta mot förmodan inte skulle fungera, spara allt med File → Save All, stäng Visual Studio, öppna det igen och öppna även igen projektet Kursverksamhet. En sådan ”omstart” hjälper ibland.

- Högerklicka på Kursverksamhet.mdf i Server Explorer-fönstret och välj Refresh. Den nya tabellen Kurser dyker upp under Tables. Expandera den

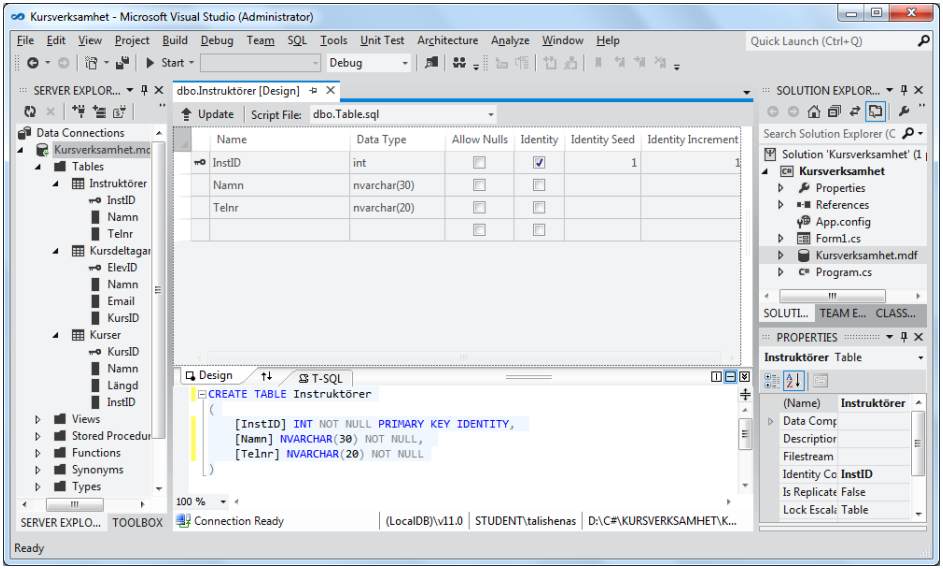
för att se kolumnerna vi just skapade. Till slut borde du ha följande bild som design för tabellen Kurser:



- Börja om att skapa ytterligare två tabeller Kursdeltagare och Instruktörer enligt ***Steg 2*** (sid 200): Server Explorer → Tables → Add New Table. Ändra koden till de nya tabellnamnen. Skapa i varje tabell kolumner enligt vår modell genom att följa instruktioner för tabellen Kurser. Definiera primärnyckeln till varje tabell. Tilldela Identity-egenskapen till alla tabellers primärnycklar. Se upp för steget **Update** på förra sidan. Efter att ha skapat t.ex. tabellen Kursdeltagare ser det ut så här:



- Gör samma sak för tabellen Instruktörer med sina resp. kolumner enligt Kalles databasmodell. Slutligen borde tabellen Instruktörer:s definition se ut så här:



Samtidigt är detta tabellen Instruktörer:s tabelldefinition som man även får i efterhand genom att i Server Explorer högerklicka på tabellen Instruktörer och välja Open Table Definition. Samma sak kan man göra med de andra tabellerna.

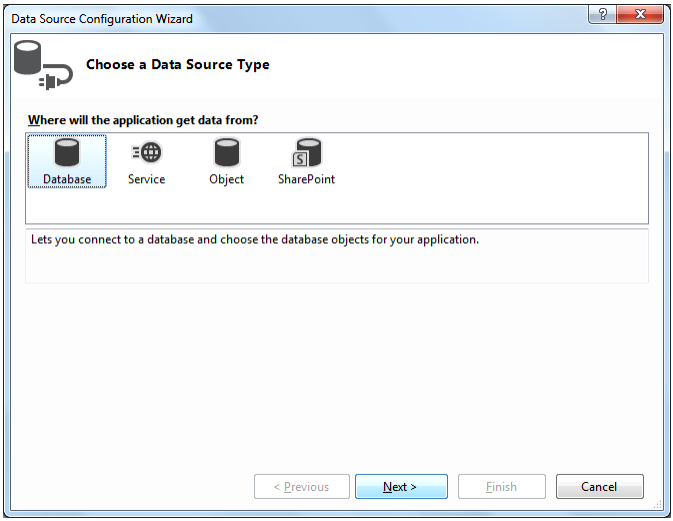
Nu har vi skapat alla tabeller vi behöver i projektet Kursverksamhet och även definierat tabellernas primärnycklar enligt projektets databasmodell. Det som kvarstår är att definiera främmande nycklar och att skapa relationer enligt modellen.

Steg 3: Att koppla projektets Dataset till databasen

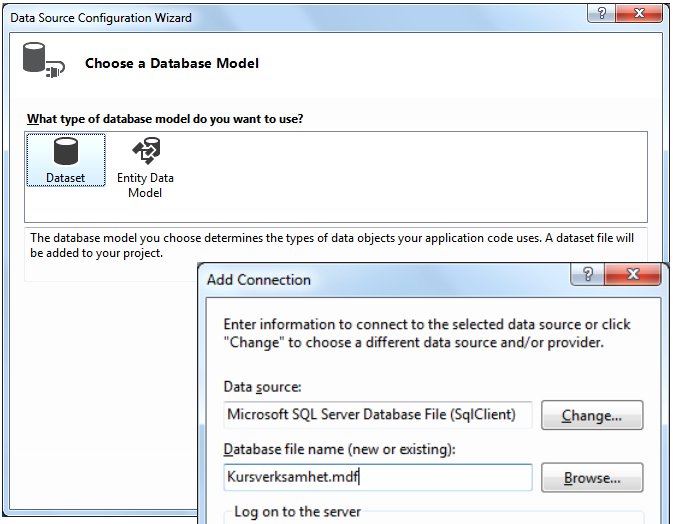
För att skapa relationer och definiera främmande nycklar – vilket är samma sak – kommer vi att använda oss av ett grafiskt verktyg i Visual Studio som heter *XML Schema*, ett diagram som visar strukturen till en databas – den digitala varianten till Kursverksamhetens databasmodell som Kalle ritade åt oss i början av detta avsnitt. Det kallades för ER-diagram (sid 199). *XML* diagrammet lagras i Visual Studio i en fil av typen *XML Schema document* som får ändelsen xsd. I våra tidigare projekt har vi redan visat diagrammet, se **DataSet Designer** (sid 183). Att det inte dykt upp i detta projekt beror på att xsd-filen är relaterad till ett s.k. **Dataset**. Och ett sådant har vi inte definierat än i projektet. Det ska vi göra nu och – när vi gjort det – koppla det till projektets databasfil Kursverksamhet.mdf. Gör så här:

- Gå i huvudmenyraden till menyn PROJECT och välj PROJECT → Add New Data Source… . Gå till Solution Explorer och klicka på projektnamnet Kursverksamhet. Gå till fönstret Data Sources och klicka på länken Add New Data Source... . Du får följande dialogruta som frågar efter typen av datakälla som vi har i projektet:

- Markera Databasen under frågan Where will the application get data from? Klicka på Next.

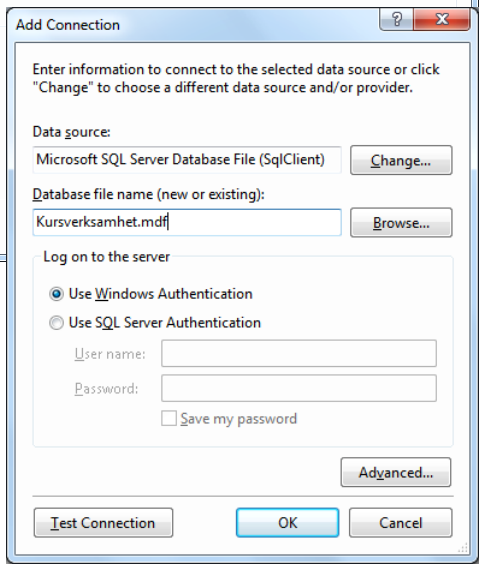


- Nästa dialogruta frågar efter typen av databasmodell. Här definieras det Dataset som vi nämnde ovan. Det kommer att tillfogas till projektet i form av en fil. Markera Dataset och klicka på Next.



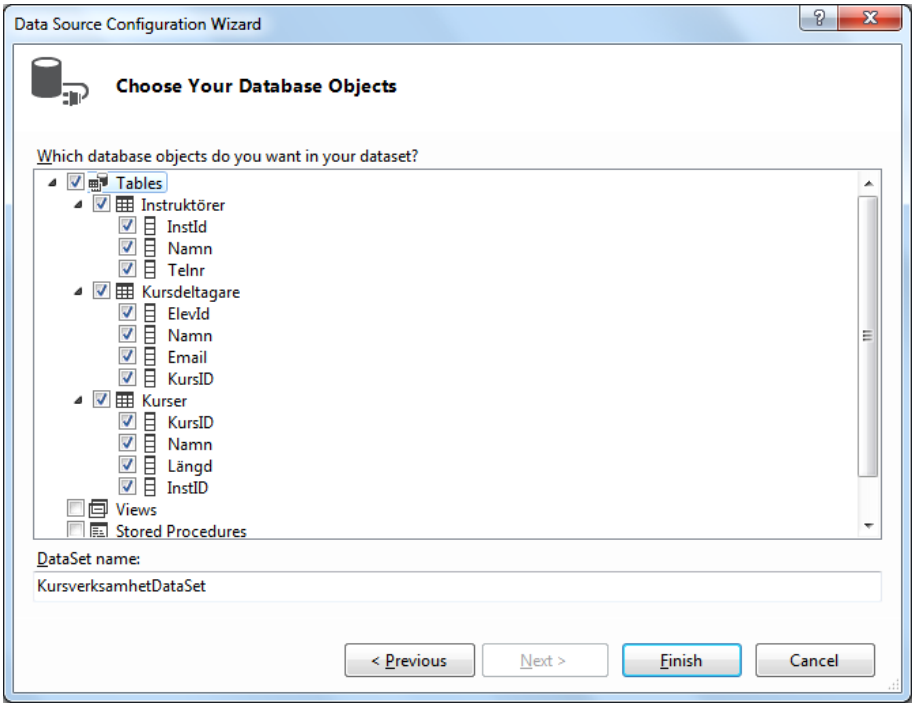
- Nästa dialogruta som inte visas här heter Choose Your Data Connection. Klicka på knappen New Connection… .

- Du får ytterligare en dialogruta som heter Add Connection (bilden till höger). Skriv Kursverksamhet.mdf i textfältet Database file name (new or existing). Kilcka på OK.

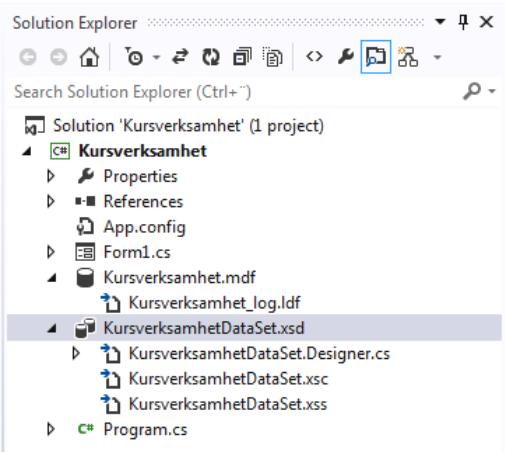


- Du återvänder till dialogrutan Choose Your Data Connection,. Klicka på Next.

- I nästa och sista dialogruta som visas på följande bild ska du välja de delar av databasen, s.k. *databasobjekt* (tabeller, vyer, lagrade procedurer,

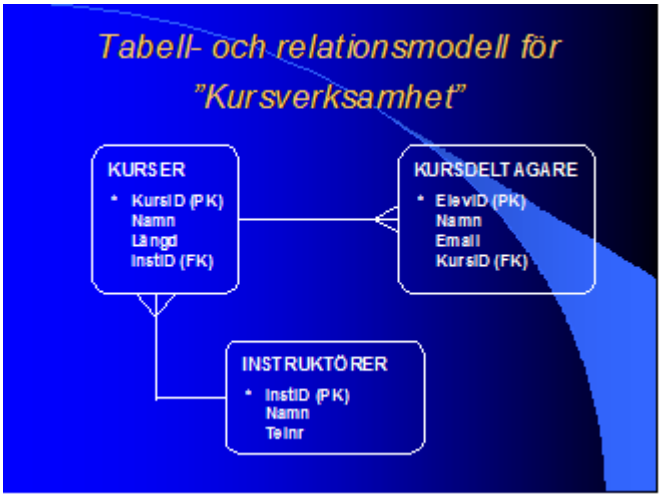


funktioner osv.) som du vill använda i detta projekt. Vår databas har bara tabeller. Så bocka den lilla rutan vänster om Tables. Samtidigt kan du, om du expanderar Tables med den lilla pilen till vänster och gör samma sak med alla tabeller, se hela databasen Kursverksamhet:s struktur. Du får en inblick i databasens innehåll. Det finns även möjligheten att ge hela Dataset ett nytt namn i textfältet DataSet name. Vi har ingen anledning att ändra det förvalda namnet KursverksamhetDataSet. Så klicka på Finish.



Som en för oss viktig konsekvens av proceduren ovan har det nu i Solution Explorer skapats filen KursverksamhetDataSet.xsd, vilket gör att vi kan ta fram det diagram som behövs för att på ett enkelt sätt skapa *relationer* mellan våra tabeller och definiera *främmande nycklar*. Innan vi gör det följer lite förklaring av dessa begrepp.

Steg 4: Att skapa relationer mellan tabeller



Vi avbildar än en gång Kalles modell till projektet *Kursverksamhet* för att vi behöver att hänvisa till den hela tiden. Att skapa relationer mellan tabeller och att definiera främmande nycklar, s.k. *Foreign Keys (FK)* är två olika uttryckssätt för en och samma sak. Vilka de främmande nycklarna ska vara, framgår av databasmodellen till höger. En främmande nyckel (FK) i en tabell, t.ex. KursID i tabellen Kursdeltagare, är en primärnyckel (PK) i en annan tabell, nämligen i tabellen Kurser. FKn KursID i Kursdeltagare lagrar informationen om i vilken kurs en elev deltar. Linjen i diagrammet mellan tabellerna Kurser och Kursdeltagare symboliserar denna relation. Gaffelsymbolen intill tabellen Kursdeltagare talar om att det i denna tabell finns en FK som refererar till tabellen Kurser:s PK, inte tvärtom. Dvs en kurs kan ha många elever, medan en elev deltar endast i en kurs. Det kan vara annorlunda i vissa skolor, men just i vår modell är det så, åtminstone enligt den föregivna modellen på förra sidan. Vår kunds berättelse (sid 198) motsäger inte detta. FK-kolumnen KursID i Kursdeltagare kommer att ha samma värden som PK-kolumnen KursID i Kurser. Efter att vi definierat relationen (med tillhörande FK) i databasen kommer ingen användare av databasen att kunna lägga in värden i FK-kolumnen KursID i Kursdeltagare som inte finns i PK-kolumnen KursID i Kurser. I praktiken innebär detta att en elev inte kan gå på en kurs som inte finns i tabellen Kurser.

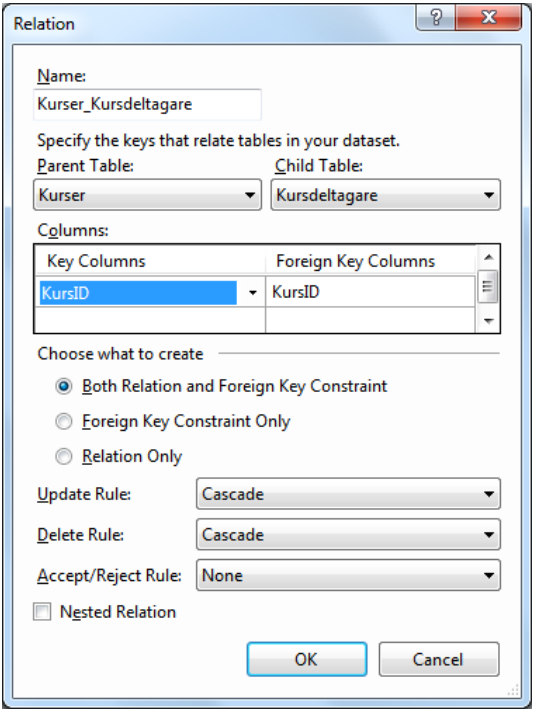
Samma sak är det med den andra relationen mellan tabellerna Instruktörer och Kurser: FKn InstID i tabellen Kurser lagrar informationen om i vilken kurs en instruktör undervisar. Linjen mellan tabellerna Instruktörer och Kurser med gaffelsymbolen intill Kurser talar om att det i tabellen Kurser finns en FK, nämligen InstID, som refererar till tabellen Instruktörer:s PK, inte tvärtom. Dvs en instruktör kan undervisa i många kurser, medan en kurs har endast en instruktör. FK-kolumnen InstID i Kurser kommer att ha samma värden som PK-kolumnen InstID i Instruktörer. Efter att vi definierat relationen (med tillhörande FK) i databasen kommer ingen användare av databasen att kunna lägga in värden i FK-kolumnen InstID i Kurser som inte finns i PK-kolumnen InstID i Instruktörer. I praktiken innebär detta att en kurs inte kan ha en instruktör som inte finns i tabellen Instruktörer.

Det vi ska göra nu är att implementera denna modells relationer i Visual Studio, närmare bestämt i *SQL Server* som används här mer som en databashanterare, vilket är möjligt pga integrationen av *Microsoft SQL Server* i Visual Studio.

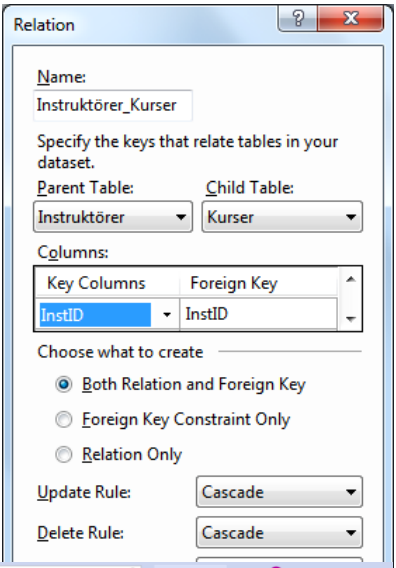
Vi använder oss av de grafiska verktyg i Visual Studio för att rita relationerna mellan databasens tabeller.

Steg 5: Att ta fram databasdiagrammet DataSet Designer

- Markera i Solution Explorer KursverksamhetDataSet.xsd, högerklicka och välj View Designer för att se databasen Kursverksamhet:s struktur i ett diagram med alla tabeller och kolumner som vi skapat i detta projekt.
- Ställ om med musen tabellerna i diagrammet så att de står relativ till varandra ungefär så som de är ritade i vår modell på förra sidan.
- Markera exakt den lilla nyckeln i tabellen Kurser:s kolumn KursID. Högerklicka och välj Add → Relation… . Dialogrutan Relation kommer upp. Skriv i textfältet Name: Kurser_-Kursdeltagare. Välj som Parent Table: Kurser och som Child Table: Kursdeltagare. Välj under Columns: som Key Columns KursID och som Foreign Key Columns också KursID. Välj under Choose what to create radioknappen Both Relation and Foreign Key Constraint. Avsluta med OK. Se bilden till höger.

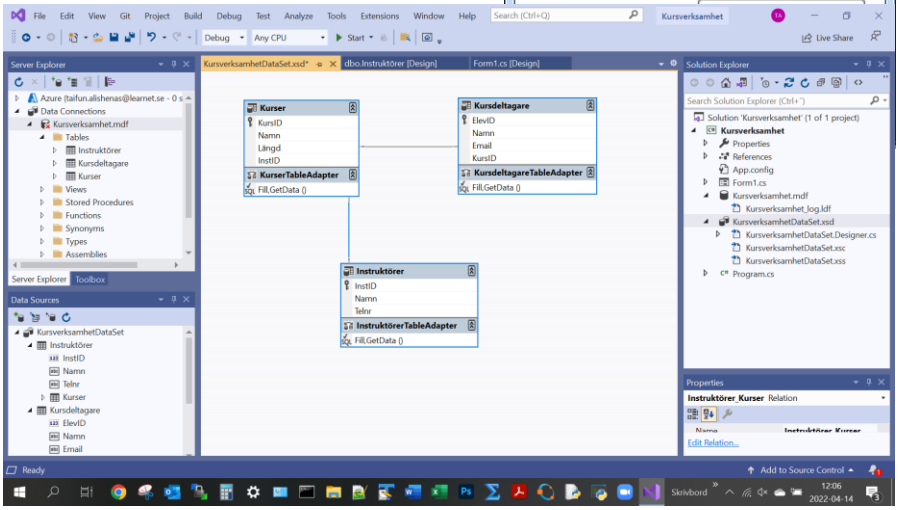


- Gör liknande med den andra relationen mellan tabellerna Instruktörer och Kurser: Få upp dialogrutan Relation med högerklick på den lilla nyckeln i tabellen Instruktörers kolumn InstID. Sedan: Add → Relation… . Skriv i Name: Instruktörer_Kurser. Välj som Parent Table: Instruktörer och som Child Table: Kurser, som Key Columns InstID och som Foreign Key Columns också InstID. Välj Both Relation and Foreign Key Constraint. Avsluta med OK.



Så här borde nu databasdiagrammet i Visual Studio se ut.

DataSet Designer:

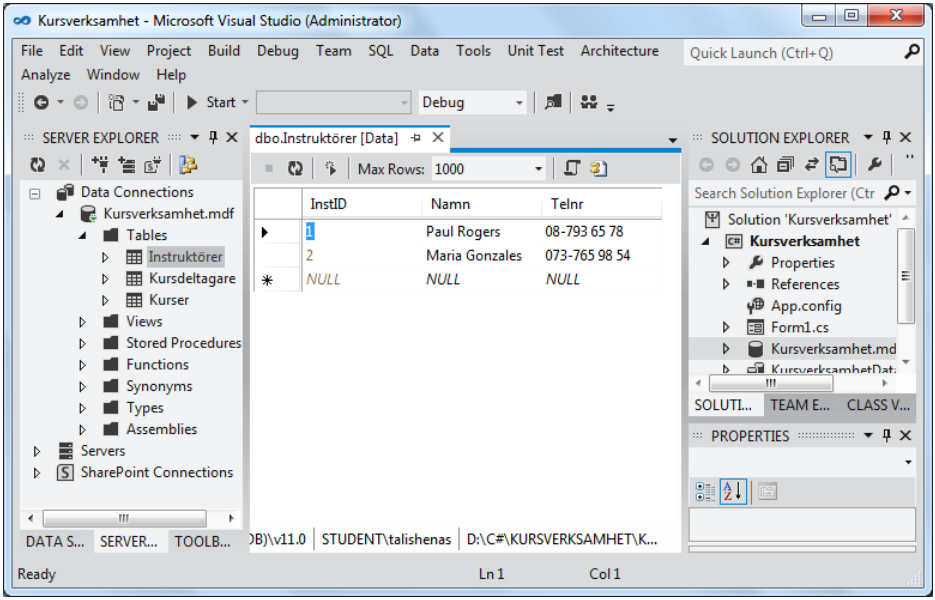


Steg 6: Att lägga in data i tabellerna

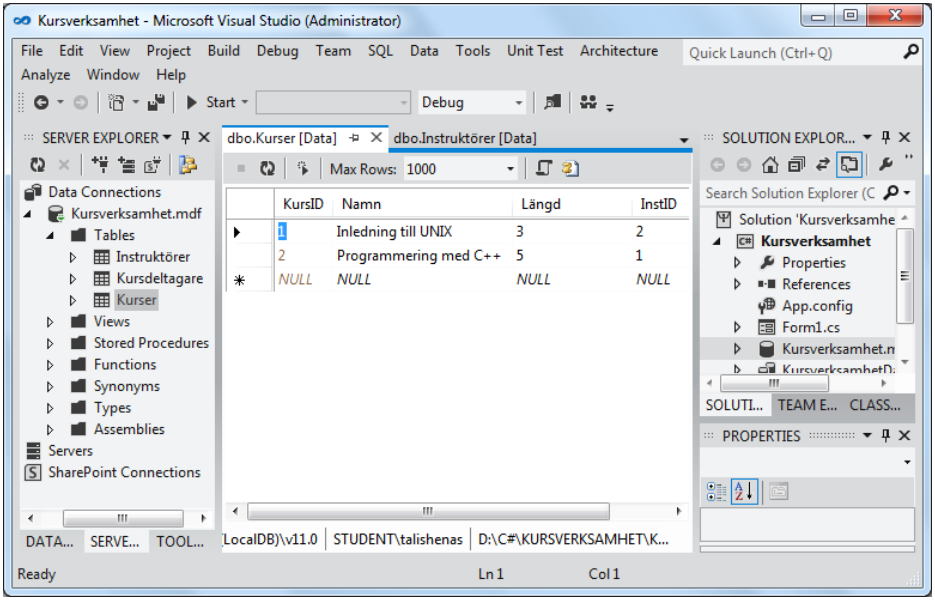
- Expandera i Server Explorer Tables och högerklicka på tabellen Instruktörer. Välj Show Table Data. Mata in de två instruktörer som nämns i kundens berättelse på sid 198 osv. Resultatet visas på nästa sida.

  Observera att man inte kan mata in några värden för kolumnen InsID, därför att den är definierad som Identity. Vi har ju själva, när vi designade tabellen Instruktörer, bestämt att InstID ska vara Identity. När vi designade tabellen Kurser anmärkte vi att det inte går att själv sätta värden på kolumner som har Identity-egenskapen (sid 201). Deras värden bestäms

automatiskt. Meningen med att lägga in data i tabellerna är alltså – just i det här fallet – att lägga in data i kolumnerna Namn och Telnr.



- Expandera i Server Explorer Tables och högerklicka på tabellen Kurser. Välj Show Table Data. Mata in de två kurser som nämns i kundens berättelse på sid 198. Så här blir blir det:



I denna tabell har vi bestämt att kursen Inledning till Unix undervisas av instruktör med InstID 2. Och när vi tittar i tabellen Instruktörer kan vi konstatera att det är Maria Gonzales som har InstID 2. Dvs Maria Gonzales undervisar kursen Inledning till Unix. På exakt samma sätt hittar SQL denna information, om vi t.ex. skickar följande SELECT-sats till databasen:

```
SELECT Instruktörer.Namn, Kurser.Namn
FROM Instruktörer, Kurser
WHERE Instruktörer.InstID = Kurser.InstID;
```

Den här varianten av SELECT-satsen är lite mer avancerad så att vi inte tagit upp den i bokens introduktion till SQL (sid 169). Konstruktionen kallas JOIN och ger ett smakprov på vad SQL kan åstadkomma. Villkoret i **WHERE**-satsdelen kallas JOIN-villkoret. I **FROM**-satsdelen kopplas ihop två tabeller – därför JOIN. Ur mängden av kombinationer av alla rader från tabellen **Instruktörer** med alla rader från tabellen **Kurser** selekterar JOIN-villkoret bara de rader där de båda instruktörsnumren (InstID) överensstämmer. Namnen på instrktörer och deras resp. kurser skrivs ut. Vi kommer att få informationen att Maria Gonzales undervisar kursen Introduktion till Unix och Paul Rogers kursen Programmering med C++.

- Mata in data efter eget godtycke till tabellen Kursdeltagare och även fler data till både tabellen Kurser och Instruktörer.
- Spara hela projektet med → File → Save All.

5.7 Att förse databasen med funktionaliteter

Databasen vi skapade i förra avsnitt var väldigt enkel. Den hade visserligen tabeller, nycklar och relationer. Men den saknade helt och hållet funktionaliteter, t.ex. en sökfunktion som hjälper oss att hitta information i databasen. Sådana funktionaliteter ska vi bygga in i en ny exempeldatabas som vi kommer att använda i detta projekt. Den är lagrad i filen **AddressBook.mdf** som du kan ladda ner filen från webbsidan www.taifun.se: Klicka där på boken *Programmering 2 med C#*:s omslagsbild, sedan på länken AddressBook.mdf. En zip-fil laddas ned: extrahera den.

I det här avsnittet kommer vi att utveckla projektet AddressBook och lära oss att:

- inkludera exempeldatabasen AddressBook.mdf i ett projekt av typen C# Windows Forms Application och använda den som lagringsplats för våra kompisars adressuppgifter.
- låta databasen själv skapa sina Labels och Textboxar.
- tillfoga funktionaliteter till databasen.

Gör så här:

- Skapa en Windows Forms Application och döp den till AddressBook. Ändra formfönstrets rubrik och storlek enligt följande:

  Form1:

Egenskap	Värde
Text	AddressBook
Size	520; 500

- Stäng fönstret Server Explorer på vänstersidan, om det fortfarande är öppet. Öppna istället fönstret Data Sources, så här:

- Skriv i textfältet Search i menyraden längst till höger Data Sources. Klicka på den lilla triangeln ovan på rubrikraden och välj Dock. Klicka i Data Sources-fönstrets menyrad på ikonen Add New Data Source. Välj i dialogrutan Choose a Data Source Type, Database och klicka på Next. Välj i nästa dialogruta Choose a Database Model, Dataset och klicka på Next.

- I Choose Your Data Connection, klicka på knappen New Connection… för att öppna dialogrutan Add Connection. Låt i textfältet Data source stå Microsoft SQL Server Database File (SqlClient).

- Klicka på Browse-knappen och navigera genom filsystemet på din dator för att ladda filen AddressBook.mdf till projektet. Klicka på OK. Visual Studio vill vill uppgradera databasfilen så att den blir kompatibel med din nyaste version av Visual Studio. I så fall svara bara ja.

- Du återvänder till dialogrutan Choose Your Data Connection, bara att det nu har tillfogats namnet på databasfilen du valt i förra steg, nämligen AddressBook.mdf. Klicka på Next. Svara Ja på frågan om du vill kopiera filen till ditt projekt.

- I nästa dialogruta som heter Save the Connection String to the Application Configuration File är namnet AddressBookConnectionString redan förvalt för den förbindelse du skapade ovan. Bocka för lilla rutan Yes, save the connection as: (om den inte redan är förbockad) och klicka på Next.

- I Choose Your Database Objects välj Tables. Expandera Tables samt tabellen Addresses. Behåll det förvalda namnet AddressBookDataSet som DataSet name. Avsluta med Finish.

- Du återvänder till din ursprungliga miljö. I Solution Explorer har kommit till: AddressBook.mdf. Markera den, högerklicka och välj Open. Server Explorer-fönstret öppnas till vänster med den nya databasens innehåll.

- Även Data Sources visar den nya databasens innehåll under DataSet-namnet AddressBookDataSet: Den har endast en tabell som heter Addresses och har fem kolumner. Expandera tabellen Addresses.

Automatiska Labels och Textboxar

Här vill vi låta databasen själv skapa Labels och Textboxar.

- Markera tabellen Addresses i fönstret Data Sources. Observera att det till höger om namnet Addresses finns en dropplista. Klicka på dropplistans lilla pil för att se alternativen. Klicka på Details. På ytan hander ingenting. Men i själva verket har du valt att ha ett *detaljerat* grafiskt gränssnitt på din form, när du med musen drar tabellen Addresses från Data Sources till formen. Istället för att få en DataGridView samt en BindingNavigator, vilket är default-alternativet som valdes i vårt första databasprojekt FirstDatabase, får du nu en helt annorlunda bild. Gör nu följande för att se:

- Markera tabellen Addresses i Data Sources och dra den med musen (genom att hålla ned den vänstra musknappen) till formen. Det skapas fem par Label- och TextBox-kontroller som motsvarar tabellen Addresses' fem kolumner. Ja, t.o.m. kolumnrubrikerna hamnar som text från databasen på Label-kontrollerna. Även en BindingNavigator följer med som lägger sig under formrubriken. Placera med musen gruppen med fem Labels och fem Textboxar i den övre delen av formfönstret, en bit under BindingNavigator, centrerat horisontellt.

- Klicka på formens lediga plats. Markera TextBoxen som står höger om Labeln Address ID. Gå till Properties-fönstret och ändra denna TextBox' värde för ReadOnly-egenskapen till True, eftersom databaskolumnen Add-

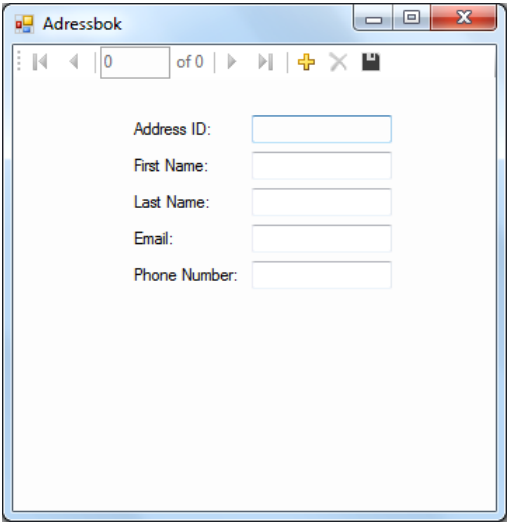
ress ID som motsvarar denna TextBox borde vara en Identity vars värden genereras automatiskt och inte får överskrivas av databasens användare:

addressIDTextBox:

Egenskap	Värde
ReadOnly	True

- Kompilera och kör. Observera att tabellen Addresses är tom. Bilden nedan visar hur resultatet av en körning borde bli. Behåll körläget.

- Testa projektet. Inled inmatningen av en post alltid med BindingNavigatorns **+** knapp (Add New). Mata in t.ex. för- och efternamn, emailadress och telefonnr till dina kompisar. Skriv in data i textfälten och avsluta posten med Save Data-knappen. Efter att ha matat in några poster kan du testa hur navigeringsknapparna och Delete-knappen fungerar.



Att lägga till egna funktionaliteter

- För att kunna söka efter en viss post i tabellen, genom att t.ex. ange efternamnet, måste vi lägga till en SQL-fråga till tabellens TableAdapter-klass. Gå till fönstret Data Sources, högerklicka på tabellen Addresses i och välj Edit Data Set with Designer. Databasens diagram dyker upp som består av en anda ruta som representerar tabellen Addresses. Markera den, högerklicka på AddressesTableAdapter längst ned och välj Add Quer. TableAdapter Query Configuration Wizard öppnas. Klicka dig fram med Next, utan att ändra något, till dialogrutan Specify a SQL SELECT statement. Skriv in SQL-satsen:

```
SELECT *
FROM Addresses
WHERE LastName = @lastname;
```

`@` framför **`lastname`** gör att **`@lastname`** blir en variabel *parameter* som kommer att ersättas av ett värde när SQL-frågan exekveras. Klicka på Next (OBS! inte på finish!).

- Ändra i Wizardens nästa dialogruta Choose Methods to Generate de förvalda namnen FillBy och GetDataBy till FillByLastName och GetDataByLastName. Klicka på finish. Observera att de två nya metoder som innehåller SQL-satsen ovan (inkl. parametern **`@lastname`**), har kommit till under AddressesTableAdapter.

- Återvänd till Form1:s design. Stäng fönstret Data Sources till vänster. Öppna istället Toolbox från huvudmenyraden: View → Toolbox. Expandera All Windows Forms. Hämta en GroupBox-kontroll till formen. Gör följande ändringar i GroupBox-kontrollens egensakper:

  groupBox1:

Egenskap	Värde
Location	20; 260
Size	450; 70
Text	Hitta en post via efternamnet:

- Impandera (krymp) All Windows Forms och expandera Common Controls. Markera GroupBox-kontrollen i formen. Dubbelklicka i Toolbox på kontrollen Label så att den hamnar i GroupBoxen. Gör följande ändringar i Label-kontrollens egensakper:

  label1:

Egenskap	Värde
Location	6; 38
Text	Last Name

- Markera GroupBox-kontrollen i formen. Dubbelklicka i Toolbox på kontrollen TextBox så att den hamnar i GroupBoxen. Gör följande ändringar i TextBox-kontrollens egensakper:

  textBox1:

Egenskap	Värde
Location	100; 35
Size	200; 30

- Markera GroupBox-kontrollen i formen. Dubbelklicka i Toolbox på kontrollen Button så att den hamnar i GroupBoxen. Gör följande ändringar i Button-kontrollens egensakper:

  button1:

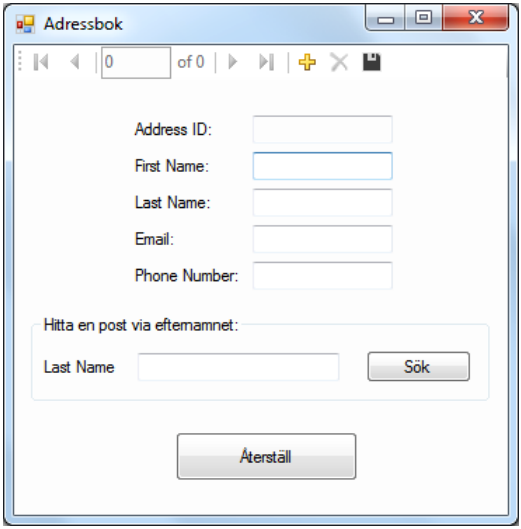
Egenskap	Värde
Location	360; 30
Size	75; 35
Text	Sök

- Lägg dessutom en Återställ-knapp under GroupBoxen längst ned i formen. Dvs markera formen. Dubbelklicka i Toolbox på kontrollen Button. Gör följande ändringar i Button-kontrollens egensakper:

  Button2:

Egenskap	Värde
Location	190; 375
Size	130; 35
Text	Återställ

Kompilera och kör. Så här borde resultatet av en körning bli:



Just nu kan man bara lägga in poster (rader). Sök- och Återställ-knapparna ger inget resultat eftersom det inte finns någon kod bakom dem. Stäng körningen och återvänd till designläget. För att ge liv åt Sök- och Återställ-knapparna gör så här:

- Dubbelklicka på Sök-knappen och lägg in kod i kroppen till händelsemetoden **`button1_Click()`** i klassen Form1 enligt nedan.
- För att kunna fortsätta med att navigera genom tabellens alla rader, efter att man sökt en speciell post via efternamnet, dubbelklicka på Återställ-knappen och lägg in kod i kroppen till händelsemetoden **`button2_Click()`** i klassen Form1 enligt nedan.

```
// Form1.cs i projektet AddressBook
// Data från en databas kan visas, läggas till eller tas bort
// Funktionalitet: Skickar en SQL-fråga från en Button
// Söker via efternamn och visar den sökta radens innehåll
using System;
using System.Windows.Forms;

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

      private void addressesBindingNavigatorSaveItem_Click(
                                    object sender, EventArgs e)
      {
         this.Validate();
         this.addressesBindingSource.EndEdit();
         this.tableAdapterManager.UpdateAll
                                    (this.addressBookDataSet);
      }

      private void Form1_Load(object sender, EventArgs e)
      {
         this.addressesTableAdapter.Fill
                        (this.addressBookDataSet.Addresses);
      }

      private void button1_Click(object sender, EventArgs e)
      {
         addressesTableAdapter.FillByLastName(
               addressBookDataSet.Addresses, textBox1.Text);
      }

      private void button2_Click(object sender, EventArgs e)
      {
         addressesTableAdapter.Fill
                           (addressBookDataSet.Addresses);
         textBox1.Text = "";
      }
   }
}
```

- Kompilera och kör. Mata in ett antal poster i databasens tabell Addresses. Testa applikationens alla möjligheter.

Övningar till kap 5

Bakom länken ***Databaser*** hittar du ***kap 5***:s PowerPoint-bilder.
I extra materialet ***Mängder*** kan du läsa om mängder och mängdoperationer.

5.1 En fabrik tillverkar tuschpennor i tre olika storlekar: *liten*, *mellan* och *stor* och i fyra olika färger: *blå*, *svart*, *röd* och *grön*.

Låt A vara mängden av alla storlekar och B mängden av alla färger av de tuschpennor som fabriken tillverkar.

a) Läs om *cartesiska produkten* på sid 83. Bilda den cartesiska produkten A x B.
b) Hur många olika typer av tuschpennor tillverkar fabriken?
c) Beskriv fabrikens sortiment i en tabell.

En butik som köper av denna fabrik, lagerför endast mellanstorleken i alla färger och storleken *liten* i *blå*. Låt R beteckna mängden av de ordnade par som butiken lagerför.

a) Bilda mängden R.
b) Hur många olika typer av tuschpennor lagerför butiken?
c) Ställ upp relationen R (butikens sortiment) i tabellform.

5.2 En möbeltillverkare producerar fem möbeltyper: *skåp, bord, säng, stol, soffa* i tre olika träslag: *björk, ek, bok.* Möblerna tillverkas *oljade, målade* eller *obehandlade*.

En av tillverkarens kunder, en möbelaffär lagerför *bord* och *stolar* av *björk* eller *ek* som är *oljade* eller *målade*.

a) Hur många modeller lagerför möbelaffären?
b) Ställ upp en tabell över de möbelmodeller (typ, träslag, behandling) möbelaffären lagerför.

5.3 En bostadsförening lagrar data om sina medlemmar i en tabell, kallad Members. Tabellens första 6 rader ser ut så här:

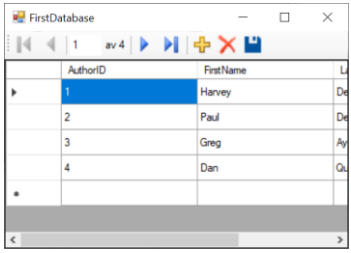
No	First name	Last name	Birth date
1	Peter	Larsson	1971
2	Emma	Carlsson	1949
3	Ingrid	Lundquist	1998
4	Hans	Lundquist	2000
5	Emma	Pettersson	1976
6	Germund	Dahlquist	1980

Skriv en SQL-sats som ger en lista över de medlemmar som:

a) heter Emma i förnamn. Listan ska innehålla all tillgänglig information om medlemmarna. Visa även svaret på din SQL-fråga.

b) heter Lundquist i efternamn. Listan ska innehålla endast medlemmarnas för- och efternamn. Visa även svaret på din SQL-fråga.
c) är födda senare än 1975. Listan ska innehålla endast medlemmarnas medlemsnr. och födelseår. Visa även svaret på din SQL-fråga.

5.4 När man exekverar projektet FirstDatabase (sid 180) med de uppgifter som anges i projektets beskrivning får man fönstret som är avbildad till höger.



Gör följande ändringar i projektet för att modifiera och vidareutveckla det:

a) Ändra formfönstrets storlek för att vid exekvering se *hela* innehållet i tabellen Authors utan att behöva justera utskriftsfönstret efteråt.
b) Modifiera projektet så att det vid exekvering visar innehållet i tabellen Titles istället för tabllen Authors. Se till att navigeringsmenyn (längst upp) är kvar.
c) Gör samma sak som i b) med tabellen AuthorISBN.

5.5 Ladda ned databasfilen AddressBook.mdf på samma sätt som du gjorde med Books.mdf. Skapa ett nytt projekt Ovn_5_5 i Visual Studio och genomför alla steg som i projektet FirstDatabase (kursboken, sid 179-185). Vilket innehåll finns i databasfilen AddressBook.mdf?

5.6 I projektet SQLclient (sid 186) hat vi skrivit några SQL-satser och skickat dem till servern via en en ComboBox' dropplista. Skriv ytterligare SQL-satser och exekvera dem i projektet SQLclient. De ska visa följande delar av databasen Books.mdf:

a) Visa alla boktitlar ordnade efter EditionNumber.
b) Visa alla böcker med Copyright 2008 ordnade efter boktitlar.
c) Visa hela innehållet i tabellen Authors.
d) Visa autorerna ordnade efter deras efternamn.

5.7 Vidareutveckla projektet Kursverksamhet (sid 199-210) genom att lägga till ytterligare data till de befintliga tabellerna Kurser, Kursdeltagare och Instruktörer. Visa tabellernas innehåll i en DataGridView-kontroll när man kör projektet, på samma sätt som i projektet SQLclient (sid 186).

a) Lägg till ytterligare tre valfria kurser till tabellen Kurser.
b) Lägg tio elever i tabellen Kursdeltagare. Tilldela varje elev till endast en kurs.
c) Lägg till ytterligare två instruktörer till tabellen Instruktörer. Avgör själv vilka instruktörer ska undervisa i vilka kurser.
d) Visa eleverna ordnade efter deras förnamn.

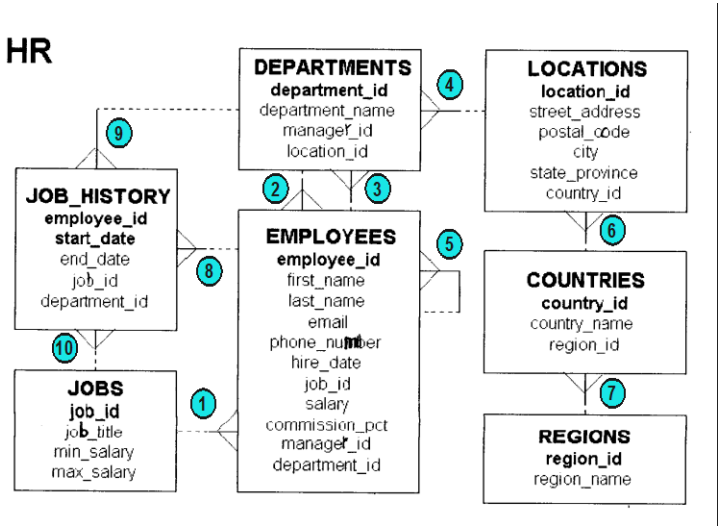
5.8 Exekvera projektet AddressBook (sid 211) och mata in via det grafiska gränssnittet följande data till tabellen Addresses:

First name	Last name	Email	Phone Number
Hans	Riesel	hriesel@kth.se	073 765 28 32
Emma	Carlet	ecarlet@lbs.se	070 329 56 79
Ingrid	Mellinder	imellind@ih.se	08 792 37 54
Ian	Cohen	icohen@kth.se	073 562 29 02
Erik	Pettersson	epetter@lbs.se	070 562 30 69
Germund	Dahlquist	gdahlq@kth.se	070 863 92 12

Se till att du inleder inmatningen av varje post med BindingNavigatorns **+** knapp (Add new) samt avslutar med Save Data-knappen.

a) Använd sedan Sök-knappen för att ta reda på Germunds emailadress.
b) Gör samma sak med Eriks telefonnummer.
c) Ta bort Ingrids post från tabellen och lägg istället till en valfri post.
d) Lägg till ytterligare en SQL-fråga till tabellen Addresses som letar efter en post via förnamnet.
e) Ta reda på med Sök-knappen Ians efternamn.

5.9 ***Human Resources (Projekt)*** Studera databasen HR (Human Resources) vars diagram visas på nästa sida. Diagrammet visar sju tabeller: Varje ruta representerar en tabell med resp. kolumner. De kolumn(er) som bildar tabellens primärnyckel står i fet stil. Identifiera varje tabells primärnyckel, alla främmande nycklar. Varje främmande nyckel ger upphov till en relation mellan databasens tabeller. Försök att läsa och beskriva relationerna 1-10 enligt relationsdatabasmodellen (sid 160).



5.10 ***Kaffeautomat (Projekt)*** Du får i uppdrag att programmera en kaffeautomat som ska användas i en cafeteria. 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 som även kan databaser. Skriv koden så generellt som möjligt så att programmet lätt kan modifieras för vilken varuautomat som helst, dessutom lätt kan översättas till vilket programmeringsspråk som helst.



Projektet går ut på att simulera en kaffeautomat med grafiskt gränssnitt och en databas som lagrar drycksortimentet samt priserna. Man ska kunna variera sortimentet dvs lägga till eller ta bort dryck med tillhörande pris – en post – från sortimentet, genom att ändra databasen utan att behöva ändra programmet.

Börja utan databas

Använd en array av kontroller för dryckernas namn och en annan array för dryckernas pris. När programmet fungerar och du har lärt dig hanteringen av databaser kan du koppla kaffeautomaten till en databas. Programmet ska innehålla en betalningsdel med möjlighet att kunna betala med fyra olika myntslag: 10 kr, 5 kr, 1 kr, 50 öre. Det grafiska gränssnittet kan t.ex. se ut som på bilden till vänster.



Programmet ska ha möjligheten att kunna välja dryck ur ett sortiment med, säg, fem olika drycker samt deras priser.

En växel- och serveringsdel ska in-

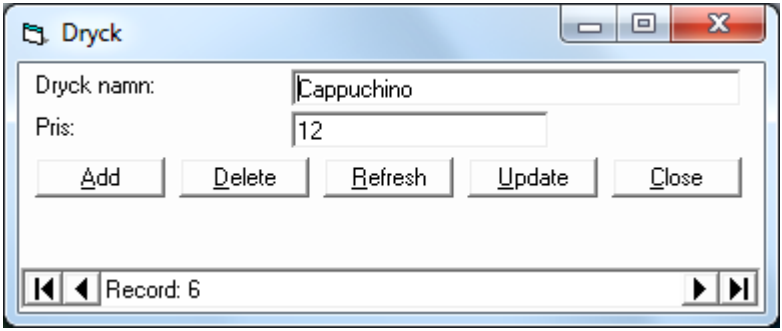
gå. Efter val av dryck samt betalning ska rätt växel lämnas tillbaka. En liten bild som föreställer en kopp ska visas upp. I exemplet på bilden har Cappuchino valts som dryck och ett 10 kr- samt två 1 kr-mynt har betalats.

Gränssnittet ska ha en menyrad med en Exit-funktion för att avsluta och en Reset-funktion för att nollställa kaffeautomaten.

Komplettera programmet med att ta hand om en eventuellt felaktig eller otillräcklig betalning från användarens sida.

Växelbeloppet är ett decimaltal i programmet. Men automaten behöver ”veta” hur många av varje myntslag som är tillåtet i automaten – endast 10 kr, 5 kr, 1 kr och 50-öringar [*] – den ska ge tillbaka. Ett växelbelopp av t.ex. 12,50 måste omvandlas i ett 10 kr- (eller två 5 kr-), två 1 kr-mynt och en 50-öring. Dessa antal är heltal. Det decimala växelbeloppet måste delas upp i automatens tillåtna mynt”system”. För att åstadkomma denna omvandling, kan du använda dig av den algoritm som beskrivits tidigare. Den skiljer sig endast i siffror från den algoritm som används för att omvandla ett antal dagar till antal år, månader, veckor och restdagar. Nyckeloperationen för alla sådana omvandlingar är modulooperatorn **%**.

Lägg till databaskoppling



För att underhålla kaffeautomaten över längre tidsperioder, t.ex. för att kunna ändra sortiment och/eller priser, utan att behöva skriva och kompilera om C#- koden, är det lämpligt att lagra sortiment- och prisinformationen i en databas och låta C#-programmet hämta aktuell, alltid uppdaterad information från databasen.

[*] Inkluderingen av 50-öringen i myntbetalningen beror inte på nostalgi utan snarare 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.

När allting fungerar felfritt, kan du ersätta arraysna för namn och pris med tabeller i en databas. Databaskopplingen ska finnas i en separat form där det ska finnas möjligheten att radera, lägga till och editera posterna i databasen.

Lägg till i menyraden ett menyval för att ladda databasformen.

Utskriften av menyn samt priserna som visades i början inte behöver hårdkodas i C# utan blir resultat av en hämtning (**`SELECT`**-sats) från databasen. På så sätt kan man alltid aktualisera menyn genom att uppdatera databastabellen.

Fortsätt med att registrera även varje transaktion i automaten dvs lägga in den med en **`INSERT`**-sats i en annan tabell som sedan kan användas både för kontroll av automaten och som underlag för ekonomisk redovisning. Avgör själv vilka uppgifter som är lämpliga att registreras. Skriv dina SQL-satserna så att de kan inbäddas i C#-kod.

Kaffeautomatkonceptet kan generaliseras inte bara till andra automater utan även till små och stora butiker eller varuhus.

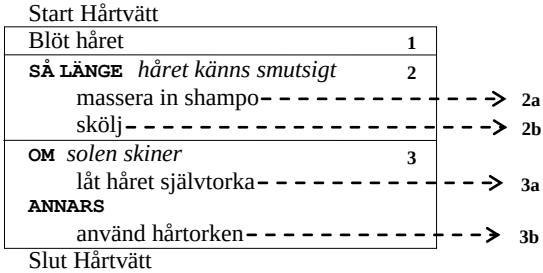
Fullständiga lösningar till övningar (Facit)

I programmering finns alltid flera möjliga lösningar till en uppgift. Därför är det, som slarvigt kallas för lösningar, i själva verket endast *lösningsförslag*. Till *projektuppgifter* eller uppgifter relaterade till ett projekt ges inga lösningsförslag. Istället finns det i projektens lydelse ofta en utförlig ledning, ibland en algoritm till lösningen.

Kapitel 1 Algoritmer och programmering, sid 56

Övn 1.1 - 1.20 i kap 1 består av frågor vars svar kan hittas i boken på sidorna 6-14.

Övn 1.21 Följande pseudokod beskriver algoritmen Hårtvätt:



a) Vilka delar av pseudokoden är *instruktioner*, vilka är *villkor* och vilka är *kontrollstrukturer*? Förklara ditt svar.

Svar:
Allt som är tryckt i normal stil i texten ovan, är *instruktioner*, allt som står i kursiv stil, är *villkor* och allt som är skrivet i fet, versal stil (annorlunda typsnitt) är *kontrollstrukturer*. *Start* och *slut* har en särställning, de är varken det ena eller det andra, utan markerar algoritmens början och slut.

Instruktioner är de delar av texten som ska *utföras*. Man skulle kunna kalla dem även kommandon. Villkor kan inte utföras, utan endast *testas* vars resultat endast kan vara sant eller falskt. De kan likställas med frågor vars svar endast kan vara ja eller nej. Svaren avgör vad som ska göras, dvs vilka (under)instruktioner ska utföras. I hårtvättalgoritmen finns endast två villkor: *håret känns smutsigt* och *solen skiner*. De är kopplade till kontrollstrukturerna **SÅ LÄNGE** och **OM-ANNARS**. Tillsammans styr de algoritmens förlopp.

Kontrollstrukturer är nyckelord vars logiska innebörd är avgörande för förloppet. **SÅ LÄNGE**:s logiska betydelse skiljer sig från **OM-ANNARS**: Det första inleder en repetition, medan det andra formulerar ett val mellan två alternativ: **SÅ LÄNGE** *håret känns smutsigt* innebär att man måste massera in shampo och skölja ev. flera gånger, medan **OM** *solen skiner* betyder att man antingen låter håret självtorka eller använder hårtorken, beroende på om solen skiner, men endast en gång.

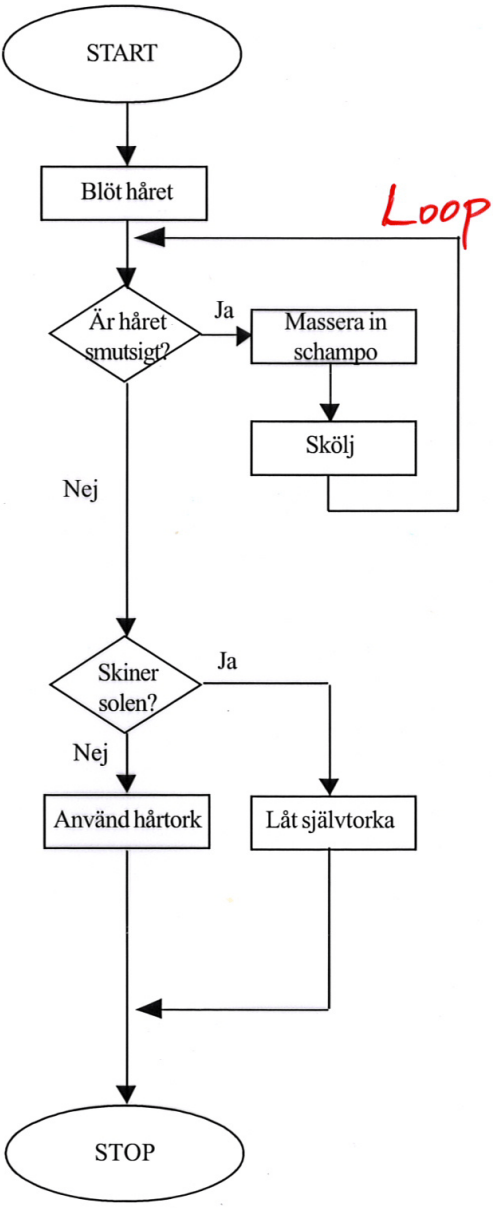
b) Dela in instruktionerna i *huvud-* och *underinstruktioner*.

Hela algoritmen kan delas in i tre huvud- och fyra underinstruktioner: I pseudokodens text ovan är de tre huvudinstruktionerna markerade med **1**, **2** och **3**. De fyra underinstruktionerna

2a, **2b**, **3a** och **3b** är indragna för att visa att **2a**, **2b** tillhör huvudinstruktion **2** och att **3a**, **3b** är delar av huvudinstruktion **3**.

c) Rita ett flödesschema till pseudokoden ovan.

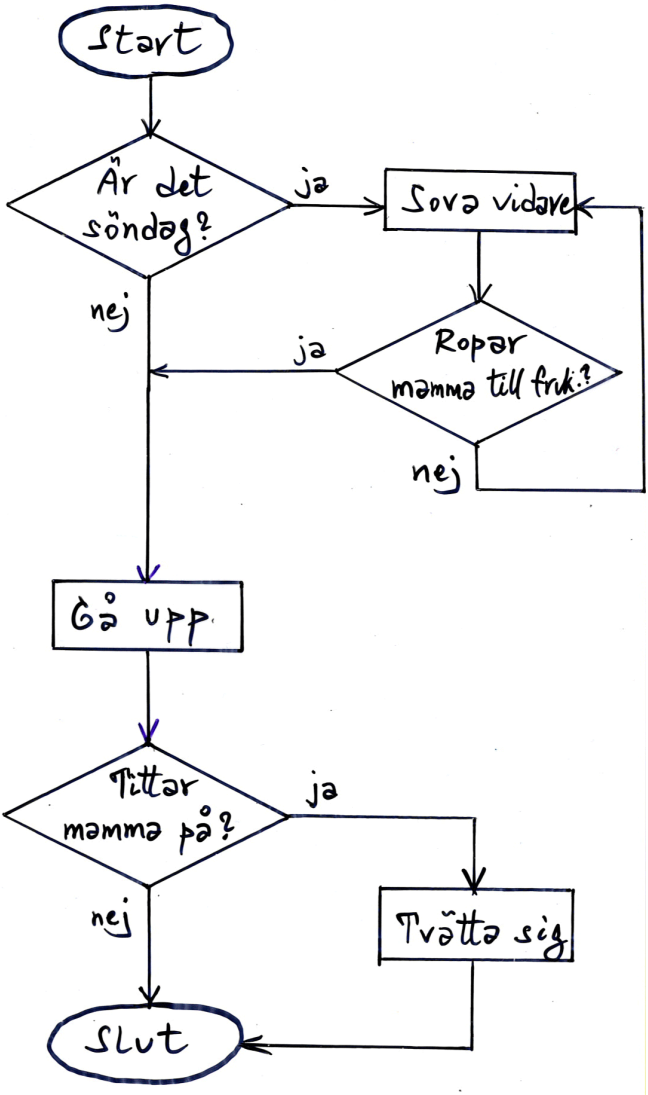
Utgående från analysen av pseudokoden Hårtvätt i **a)** och **b)** ges följande förslag till flödesschema som är en ren översättning av pseudokoden – en annan form av samma algoritm :



Övn 1.22 Följande algoritm – låt oss kalla den *Kalle-algoritmen* – är formulerad på vanligt språk:

> *På vardagar går Kalle upp. Han tvättar sig, om mamman tittar på. På söndagar sover Kalle vidare tills mamman ropar ho-nom till frukost, i så fall gör han som på vardagar.*

a) Rita flödesschemat till Kalle-algoritmen. Anta att lördag är en vardag.



b) Översätt flödesschemat till pseudokod.

```
Start Kalle
OM det är söndag
        sover Kalle vidare
    TILLS mamma ropar till frukost
Kalle går upp
OM mamma tittar på
    tvättar han sig
Slut Kalle
```

c) Finns det i Kalle-algoritmen möjligheten till en evighetsloop? När skulle den rent teoretiskt kunna inträffa? Hur kan den förhindras?

Kallealgoritmen innehåller möjligheten till en evighetsloop som kan inträffa om mamma aldrig ropar till frukost. *Möjligheten* till en evighetsloop finns i alla loopar.

Om den verkligen inträffar eller ej, beror på hur loopens avslutningsvillkor är formulerat och hur villkoret realiseras. För att undvika evighetsloop måste villkorets sanningsvärde ändras under algoritmens realisering. Dvs mamma måste få chansen att ropa till frukost.

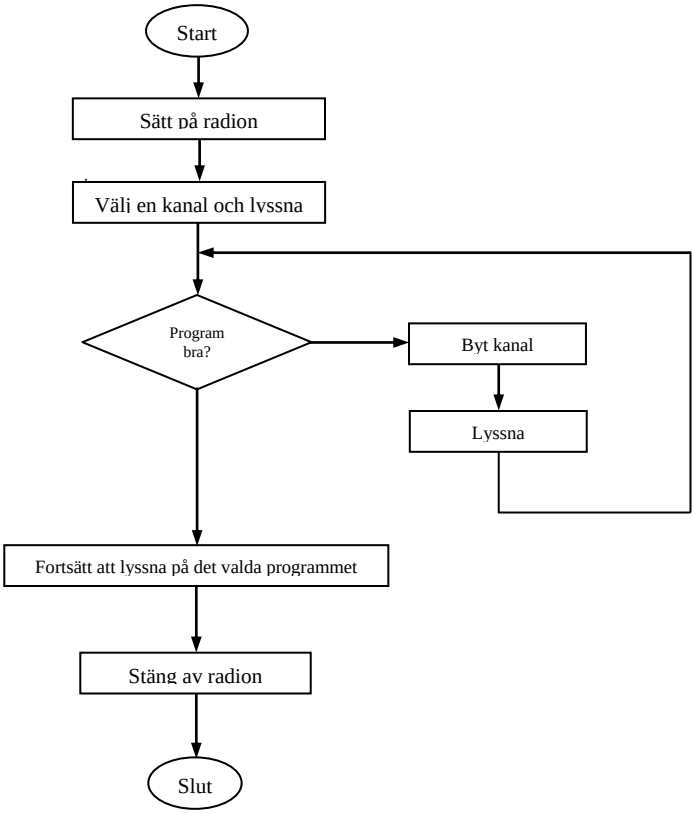
Övn 1.23 Är följande pseudokod logiskt identisk med Kalle-algoritmen från övn 1.22?

```
Start Kanske_Kalle?
OM det är söndag
        sover Kalle vidare
    TILLS mamma ropar till frukost
ANNARS
    går han upp
OM mamma tittar på
    tvättar han sig
Slut Kanske_Kalle?
```

Nej, denna pseudokod är logiskt inte identisk med Kalle-algoritmen från övn 1.22. Skillnaden är att Kalle enligt denna pseudokod aldrig går upp på söndagar, därför att den logiska innebörden av tvåvägsvalet **OM-ANNARS** skiljer sig från den enkla **OM**-satsen (utan **ANNARS**). **OM** och **ANNARS** utesluter varandra, dvs när det verkligen är söndag, utelsluts det som står under **ANNARS**. Först när man stryker **ANNARS** från denna pseudokod blir den logiskt identisk med Kalle-algoritmen.

Övn 1.24 Rita flödesschemat till följande pseudokod:

```
Sätt på radion
Välj en kanal och  lyssna
SÅ LÄNGE du inte har hittat ett bra program
byt kanal
lyssna
Fortsätt att lyssna på det valda programmet
Stäng av radion
```



Övn 1.25 Skriv ett C# program som läser in två heltal, multiplicerar dem med varandra och skriver ut resultatet blandat med förklarande text. Om du t.ex. matar in **3** till det första och **4** till det andra heltalet, ska programmet skriva ut: **3 gånger 4 är 12.** Utveckla programmet vidare med ytterligare räkneoperationer, kanske så småningom till en liten kalkylator. tioner, kanske så småningom till en liten kalkylator, se 1.29 Kalkylatorn.

```
using System;
class Ovn_1_25
{
   static void Main()
   {
      Console.Write("\n\tMata in ett heltal:\t\t");       // Ledtext
      int no1 = int.Parse(Console.ReadLine());            // Inläsning
      Console.Write("\n\tMata in ett heltal till:\t");
      int no2 = int.Parse(Console.ReadLine());

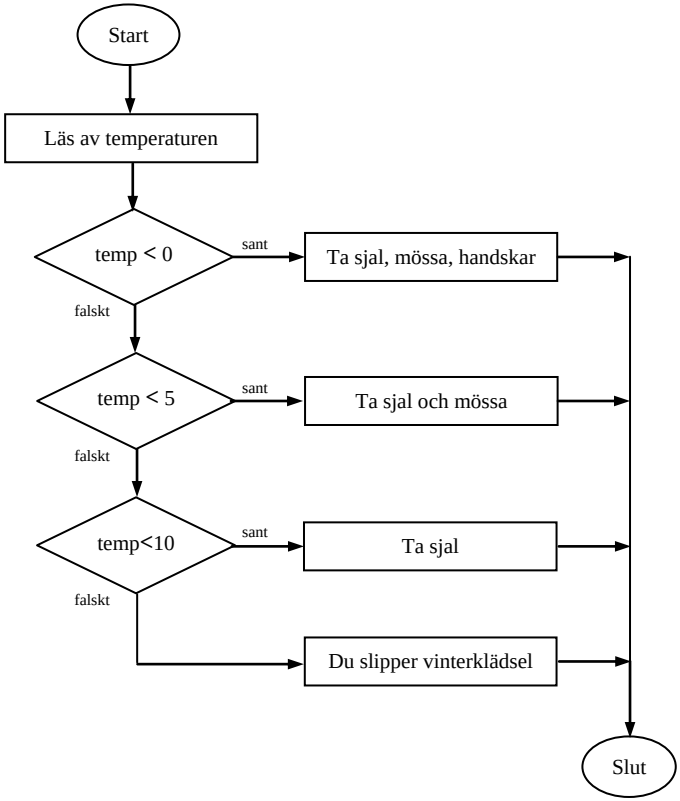
      Console.WriteLine("\n\n\t"                                    +
         no1 + " plus "   + no2 + " är " + (no1 + no2) + "\n\t" +
         no1 + " minus " + no2 + " är "  + (no1 - no2) + "\n\t" +
         no1 + " gånger " + no2 + " är " + (no1 * no2) + "\n\t" +
         no1 + " heltalsdividerad med "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

Övn 1.26 Rita ett flödesschema till följande pseudokod:

```
Start Vinterklädsel_1
Läs av temperaturen
OM temperatur < 0
    ta sjal, mössa och handskar
ANNARS OM temperatur < 5
    ta sjal och mössa
ANNARS OM temperatur < 10
    ta sjal
ANNARS
    slipper du vinterklädsel
Slut Vinterklädsel_1
```

Använd dina programmeringskunskaper för att koda pseudokoden ovan och flödesschemat du ritat, till ett C# program. Läs in ett värde för temperatur och låt programmet avgöra val av klädsel genom att skriva ut "Ta sjal, mössa, handskar... " eller liknande. För kontrollstrukturen flervägsval kan du använda **`if-else`**-stegen som kodas i C# på samma sätt som i C++.

Ett flödesschema till pseudokoden *Vinterklädsel_1* kan se ut så här:



```
using System;
class Ovn_1_26
{
   static void Main()
   {
     Console.Write("\n\tMata in temperatur:\t");
     int temperatur = int.Parse(Console.ReadLine());

     if (temperatur < 0)
        Console.WriteLine("\n\tTa sjal, mössa och handskar!\n");
     else if (temperatur < 5)
        Console.WriteLine("\n\tTa sjal och mössa!\n");
     else if (temperatur < 10)
        Console.WriteLine("\n\tTa sjal!\n");
     else
        Console.WriteLine("\n\tDu slipper vinterklädsel.\n");
   }
}
```

****

Övn 1.27 Algoritmen i övn 1.26 ovan kan formuleras med följande pseudokod:

Start *Vinterklädsel_2*
Läs av temperaturen
VÄLJ *fall* ur
    temperatur < *0*: ta sjal, mössa och handskar
    temperatur < *5*: ta sjal och mössa
    temperatur < *10*: ta sjal
    Annars: slipper du vinterklädsel
Slut *Vinterklädsel_2*

Rita flödesschemat till pseudokoden ovan och undersök den logiska likheten mellan flödesscheman i övn 1.26 och övn 1.27.

Flödesschemat till pseudokoden *Vinterklädsel_2* är identisk med flödesschemat på förra sidan. Dvs *Vinterklädsel_1* och *Vinterklädsel_2* har samma flödesschema, eftersom båda är logiskt identiska och beskriver samma algoritm. Endast pseudokodens formulering är annorlunda.

****

Övn 1.28 Collatz algoritmen har modulariserats med **void**-metoden **Collatz()** som är definierad i klassen **Collatz_mod**, se sid 42. Modularisera Collatz algoritmen med en metod med returvärde istället. Dvs definiera en metod **public static int Collatz()** som endast returnerar ETT tal i Collatz-sekvensen. Anropa metoden från en annan klass' **Main()**.

Tips: Placera loopen samt utskriftssatsen i huvudprogrammet som anropar metoden. För att dataflödet mellan loopen och metoden ska fungera tillämpa referensanrop.

```
// Collatz_return.cs
// Definierar metoden Collatz() med returvärde
// Metoden beräknar endast ETT tal i sekvensen
// Parametern n är av typ referens till int
using System;
```

```
class Collatz_return
{
   public static int Collatz(ref int n)
   {
      if (n % 2 == 1)
         n = (3 * n + 1);
      else
         n = (n / 2);
      return n;
   }
}
```

```
// Collatz_return_Test.cs
// Läser in startvärdet till Collatz algoritmen
// Anropar metoden Collatz() definierad i klassen Collatz_return,
// i en loop. Referensanrop tillämpas på metoden
using System;
class Collatz_Test_return
{
   static void Main()
   {
      Console.Write("\n\tMata in ett positivt heltal:\t");
      int number = int.Parse(Console.ReadLine());
      Console.Write("\n\t" + number + "\t");     // Startvärdet

      while (number != 1)                        // Anropet i en loop
            Console.Write(Collatz_return.Collatz(ref number) + "\t");

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

Kapitel 2 Logik för blivande programmerare, sid 84

Ovn_2_1

Skriv ett program som med hjälp av en nästlad for-sats skriver ut en rektangel fylld med stjärnor (*) till konsolen, bestående av 9 rader och 20 kolumner. Försök att numrera raderna och kolumnerna utan att förstöra helhetsbilden.

```
using System;
class Ovn_2_1
{
   static void Main()
   {
      Console.WriteLine("\n\tx = \t12345678901234567890\n");

      for (int y=1; y<=9; y++)                // Yttre slinga ordnar
      {                                       // 9 rader med radbyte.
         Console.Write("\ty=" + y + '\t');
         for (int x=1; x<=20; x++)            // Inre slinga ritar en
             Console.Write('*');              // rad av 20 stjärnor.
         Console.WriteLine();                 // Radbyte i rektangeln
      }
      Console.WriteLine();                    // Radbyte utanför
   }
}
```

Ovn_2_2.cs

Selektera (skriv ut) från den stjärnfyllda rektangeln från övn 2.1 endast den 5:e raden och den 7:e kolumnen så att det visas ett kors. Lägg in i den inre for-slingan som skriver ut en rad, en if-else-sats som i varje varv skriver ut en stjärna om ett sammansatt villkor med ELLER är uppfyllt, annars ett mellanslag. Hur blir det om du byter ut ELLER mot OCH?

```
using System;
class Ovn_2_2
{
   static void Main()
   {
      Console.WriteLine("\n\tx = \t12345678901234567890\n");

      for (int y=1; y<=9; y++)            // Yttre slinga ordnar
      {                                   // 9 rader med radbyte.
         Console.Write("\ty=" + y + '\t');
         for (int x=1; x<=20; x++)        // Inre slinga ritar en rad
             if (y==5 || x==7)            // Sammansatt villkor:ELLER
//           if (y==5 && x==7)            // OCH ger skärningspunkten
                Console.Write('*');
             else
                Console.Write(' ');

         Console.WriteLine();             // Radbyte i rektangeln
      }

      Console.WriteLine();                // Radbyte utanför
   }
}
```

****

Ovn_2_3

Omvandla korset från övn 2.2 till dess negativ, dvs skriv ut alla stjärnor från övn 2.1 utom den 5:e raden och den 7:e kolumnen. Använd den logiska operatorn NEGATION. Negera en gång hela det sammansatta ELLER-villkoret från övn 2.2 och en gång det sammansatta villkorets delvillkor. I båda fall borde du få samma resultat.

```
using System;
class Ovn_2_3
{
   static void Main()
   {
      Console.WriteLine("\n\tx = \t12345678901234567890\n");

      for (int y=1; y<=9; y++)
      {
         Console.Write("\ty=" + y + '\t');

         for (int x=1; x<=20; x++)
             if ( !(y==5 || x==7) )       // NEG.av ammansatt villkor
//           if (!(y==5) && !(x==7) )     // NEGATION av delvillkoren
                Console.Write('*');
             else
                Console.Write(' ');
```

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

****

Ovn_2_4

Skriv ett program som läser in tre tal, hittar och skriver ut det största av dem. Lös problemet genom att använda tre enkla if-satser med sammansatta villkor och den logiska operatorn &&. På så sätt kan du i varje if-sats jämföra ett tal med de två andra. Varför måste variabeln som lagrar det största talet, initieras vid deklarationen?

```
using System;
class Ovn_2_4
{
   static void Main()
   {
      int max = 0;                    // Initiering vid deklarationen

      Console.Write("\n\tMata in no1:\t");
      int no1 = int.Parse(Console.ReadLine());
      Console.Write("\n\tMata in no2:\t");
      int no2 = int.Parse(Console.ReadLine());
      Console.Write("\n\tMata in tal3:\t");
      int tal3 = int.Parse(Console.ReadLine());
      if ((no1 > no2) && (no1 > tal3))
         max = no1;
      if ((no2 > no1) && (no2 > tal3))
         max = no2;
      if ((tal3 > no1) && (tal3 > no2))
         max = tal3;

      Console.WriteLine("\n\tDet största talet är " +  max + '\n');
   }
}
```

Variabeln max måste initieras vid deklarationen, för annars kan koden inte kompileras pga villkorlig initiering av max i if-satserna.

****

Ovn_2_5

Skriv ett program som skriver ut sanningsvärdet till det enkla villkoret a < 10 där a är en heltalsvariabel vars värde läses in. Testa ditt program genom att mata in t.ex. 9, 10 resp. 11.

```
using System;
class Ovn_2_5
{
   static void Main()
   {
      Console.Write("\n\tAnge ett heltal:\t");
      int a = int.Parse(Console.ReadLine());
      Console.WriteLine("\n\t" + a + " < 10 är " + (a < 10) + '\n');
   }
}
```

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

Ovn_2_6a

Bestäm sanningsvärden hos de följande logiska uttrycken,
först med papper och penna, sedan i ett C#-program:
a) (8 < 7) && (true || false)

```
using System;
class Ovn_2_6a
{
   static void Main()
   {
      Console.WriteLine(
              "\n\tUttrycket (8 < 7) && (true || false) är " +
                            ((8 < 7) && (true || false)) + '\n');
   }
}
```

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

Ovn_2_6b

Bestäm sanningsvärden hos de följande logiska uttrycken,
först med papper och penna, sedan i ett C#-program:
b) !(3 < 3.01) || (!(0==0) && true)

```
using System;
class Ovn_2_6b
{
   static void Main()
   {
      Console.WriteLine(
         "\n\tUttrycket !(3 < 3.01) || (!(0==0) && true) är " +
                        (!(3 < 3.01) || (!(0==0) && true)) + '\n');
   }
}
```

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

Ovn_2_6c.cs

Bestäm sanningsvärden hos de följande logiska uttrycken,
först med papper och penna, sedan i ett C#-program:
*c) (true || !false) && !(!(4*5==1) && false)*

```
using System;
class Ovn_2_6c
{
   static void Main()
   {
      Console.WriteLine("\n\tUttrycket" +
            " (true || !false) && !(!(4*5==1) && false) är " +
             ((true || !false) && !(!(4*5==1) && false)) + '\n');
   }
}
```

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

Ovn_2_7

Följande enkel version av Gissa tal-spelet tillåter endast en spel-
omgång (utan loop). För att koda ett trevägsval nästlar programmet
*en **if-else**-sats i en annan **if-else**-sats:*

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

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

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

Modifiera programmet ovan genom att använda logiska operatorer och sammansatta villkor i syftet att förenkla nästlingen. Det nya programmet ska göra samma sak som GuessIfElse. Bedöm i slutet själv om det har blivit mer förståelig kod.

```
using System;
class Ovn_2_7
{
   static void Main()
   {
      Console.Write("\n\tGissa ett tal mellan 1 och 20:\t");
      int guessedNo = int.Parse(Console.ReadLine());

      if ((guessedNo < 17) || (guessedNo > 17))
         if (guessedNo < 17)
            Console.WriteLine("\n\tFör litet!\n");
         else
            Console.WriteLine("\n\tFör stort!\n");
      else
      {
         Console.Write('\u0007');
         Console.WriteLine("\n\tGrattis, du har gissat rätt!\n");
      }
   }
}
```

****

Ovn_2_8

Modifiera programmet PasswdCaps (sid 75) genom att lägga in kod som begränsar antalet inloggningsförsök till t.ex. 3. Överskrider man denna gräns ska programmet avslutas efter att ha skrivit ut ett meddelande av typ "Du har försökt 3 gånger. Nu avslutas programmet!" Tips: Använd en if-sats som avslutar programmet genom att bryta loopen med break.

```
using System;
class Ovn_2_8
{
```

```
   static void Main()
   {
      String input;
      bool wrongPasswd;
      int antalFörsök = 0;

      do
      {
         antalFörsök++;
         Console.Write("\n\tSkriv ditt lösenord:\t");
         input = Console.ReadLine();
         wrongPasswd = !(input == "hemligt") &&
                       !(input == "HEMLIGT");
//       wrongPasswd = !(input == "hemligt" ||        // Alternativt
//                       input == "HEMLIGT");
         if (wrongPasswd && (antalFörsök > 2))
         {
            Console.WriteLine("\n\tDu har försökt 3 gånger. " +
                                    "Nu avslutas programmet!\n");
            break;
         }
         if (wrongPasswd)
            Console.WriteLine("\n\tFel lösenord. Försök igen!\n");
      } while (wrongPasswd);

      if (!wrongPasswd)
         Console.WriteLine("\n\tDet är OK. Nu är du inloggad!\n");
   }
}
```

****

Ovn_2_9

Operationer med mängder kan illustreras grafiskt. Hur man gör det kan du läsa i avsnitt 2.5 Mängdlära och logik på sid 79. Diagrammen du ser där kallas för Venndiagram efter den brittiske logikern John Venn. Med Venndiagram kan man illustrera även logiska lagar när de är skrivna i mängdnotation, där en mängd motsvarar en utsaga.

De Morgans lagar togs upp i kap2 (sid 77) & kan då formuleras så här:

$$\neg\ (p\ \ OCH\ \ q) \quad \leftrightarrow \quad \neg\ p\ ELLER\ \neg\ q$$

$$\neg\ (p\ ELLER\ q) \quad \leftrightarrow \quad \neg\ p\ \ OCH\ \ \neg\ q$$

där p och q är utsagor, ¬ är symbolen för logisk negation och ↔ symbolen för logisk ekvivalens. Så här kan man skriva om dem till samband mellan mängder:

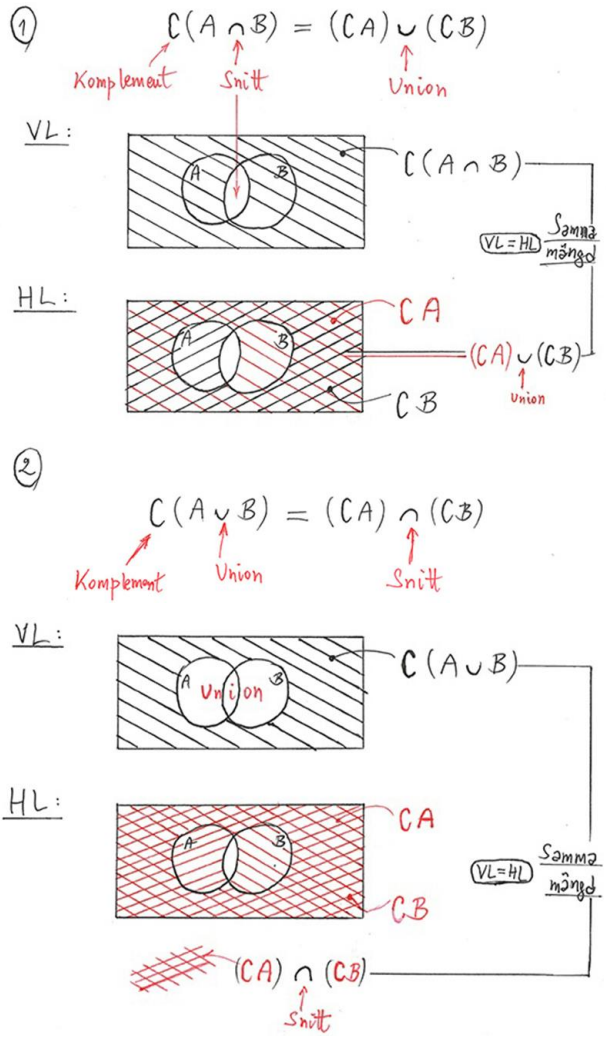
Anta att A och B är mängder och ð är symbolen för komplementmängden, ∩ för snittet och U för unionen av två mängder (se definitionerna i avsnitt 2.5 Mängdlära och logik på sid 79. Då kan De Morgans lagar skrivas i mängdnotation så här:

$$ð\ (A\ \cap\ B) \ = \ (ð\ A)\ \ U\ \ (ð\ B)$$

$$ð\ (A\ \ U\ \ B) \ = \ (ð\ A)\ \cap\ (ð\ B)$$

Illustrera De Morgans lagar i mängdnotation med Venndiagram.

Lösningen:



****

Kapitel 3 Datastrukturer och abstrakta datatyper, sid 133

Ovn_3_1_Class

Modifiera klassen Fish (sid 107) så här: Deklarera datamedlemmarna som private och metoderna som public. Förse klassen med ytterligare två publika metoder, så att den nya klassen **Fish_priv** *har följande utseende. Modifiera programmet ArrayOfRef (sid 108) så att det modifierade programmet gör samma sak som det ursprungliga.*

```
using System;
class Fish_priv
```

```
{
   private string sort;
   private float weight, size;

   public Fish_priv(string S, float w, float s)
   {
      sort   = S;
      weight = w;
      size   = s;
   }

   public int Price()
   {
      return (int) Math.Round(weight * 7.25f / 100);
   }

   public int Shipping()
   {
      return (int) Math.Round(weight * 0.02f + size * 0.1f);
   }

   public string AsString()
   {
      return sort    + "\t "     +
             weight  + "\t\t   " + size       + "\t\t " +
             Price() + "\t "     + Shipping() + "\n"    ;
   }
}
```

Ovn_3_1_Test

Modifiera programmet ArrayOfRef (sid 108) så att det modifierade programmet gör samma sak som det ursprungliga.

```
using System;
class ArrayOfRef_ny
{
  static void Main()
  {
    string fiskSort;
    float  fiskVikt, fiskLängd;
    Fish_priv[] f = new Fish_priv[5];       // Array av referenser

    for (int i = 0; i < f.Length; i++)
    {
        Console.Write("\n\tMata in sorten till fisk" + (i+1) +
                                                        ":\t");
        fiskSort = Console.ReadLine();      // Input
        if (fiskSort.Length <= 7) fiskSort += '\t';
        Console.Write("\tMata in vikten till fisk" + (i+1) +
                                                      ":\t");
        fiskVikt = (float) Convert.ToDecimal(Console.ReadLine());
        Console.Write("\tMata in längden till fisk" + (i+1) + ":\t");
        fiskLängd = (float) Convert.ToDecimal(Console.ReadLine());

        f[i] = new Fish_priv(fiskSort, fiskVikt, fiskLängd);
    }
    Console.Write("\nFisksort\tVikt i g\tLängd i cm\tPris\tFrakt\n" +
  "-------------------------------------------------------------\n");
    for (int i = 0; i < f.Length; i++)
```

```
        Console.WriteLine(f[i].AsString());
    }
}
```

****

Ovn_3_2
Skriv ett program som läser in 10 heltal från konsolen, lagrar dem i en array och skriver ut dem i omvänd ordning.

```
using System;
class Ovn_2_2
{
    static void Main()
    {
        int[] no = new int[10];

        Console.WriteLine("\n\tSkriv in 10 heltal:\n");
        for (int i = 0; i <= 9; i++)
        {
            Console.Write("\tTal nr " + (i+1) + ":\t");
            no[i] = int.Parse(Console.ReadLine());
        }

        Console.WriteLine("\nDina tal i omvänd ordning:\n");
        for (int i = 9; i >= 0; i--)
            Console.Write(no[i] + "\t");

        Console.WriteLine();
    }
}
```

****

Ovn_3_3.cs
Skriv ett program som läser in text i gemener, lagrar den i en array av char och skriver ut den framhävd i versaler och med mellanslag mellan varje tecken.

```
using System;
class Ovn_3_3
{
    static void Main()
    {
        Console.Write("\n\tSkriv in text:\t\t");
        char[] text = Console.ReadLine().ToCharArray();

        Console.Write("\n\tTexten framhävd:\t");
        for (int i = 0; i < text.Length; i++)
            Console.Write("" + (char) (text[i] - 32) + ' ');
        Console.WriteLine('\n');
    }
}
```

****

Ovn_3_4
Skriv ett program som frågar efter användarens för- & efternamn, hälsar

sedan användaren i en utskrift med fullständiga namnet, förnamnets längd samt efternamnets första & sista bokstav. Lös uppgiften generellt utan att använda information om något speciellt för- och efternamn.

```
using System;
class Ovn_3_4
{
  static void Main()
  {
    char surname0 = '0';          // Undviker villkorlig initiering
    Console.Write("\n\tSkriv in ditt för- och efternamn:\t");
    string input = Console.ReadLine();
    char[] name = input.ToCharArray();

    int i = 0;
    while (name[i] != ' ')        // Går igenom endast förnamnet
    {
       i++;
       if (name[i] == ' ') // Hittar för- och efternamnets avskiljare
          surname0 = name[i+1]; // Hittar efternamnets första bokstav
    }
    Console.WriteLine("\n\tHej, "   + input     +
      "\n\tDitt förnamns längd är " + i         +
      "\n\tDitt efternamns första bokstav är "  + surname0 +
      "\n\tDitt efternamns sista  bokstav är "  +
                                      name[name.Length-1] + '\n');
  }
}
```

****

Ovn_3_5

*Skriv ett program där **Main()** läser in en persons fullständiga namn och hälsar tillbaka med namnets initialer. Dessa ska bestämmas och skrivas ut i en annan metod - med huvudet: static void Initials-(char[] name) - som anropas i **Main()**.*

```
using System;
class Ovn_3_5
{
   static void Main()
   {
      Console.Write("\n\tSkriv in ditt för- och efternamn:\t");
      string input = Console.ReadLine();
      char[] dittNamn = input.ToCharArray();

      Console.Write("\n\tHej, " + input +
                    "\n\n\tDina initialer är\t\t\t" );
      Initials(dittNamn);              // Anropet
      Console.WriteLine('\n');
   }

   static void Initials(char[] name)   // Metoden
   {
      int i = 0;
      Console.Write(name[i]);          // Första initialen
      while (name[i] != ' ')           // Går igenom endast förnamnet
      {
         i++;
         if (name[i] == ' ')          // Hittar för- och efternamnets
                                      //                  avskiljare
```

```
            Console.Write(name[i+1]); // Andra initialen
        }
    }
}
```

****

Ovn_3_6

Skriv ett program som slumpar fram 1000 heltal mellan 60 och 140 (tänkbara hastigheter på en motorväg), lagrar dem i en array kallad hastighet, beräknar och skriver ut deras medelvärde med förklarande text. Använd klassen RandArray (sid 116) som extern modul.
För att detta program ska fungera måste klassen RandArray på sid 116 inkluderas i samma Visual Studio projekt.

```
using System;
class Ovn_3_6
{
  static void Main()
  {
    Random r = new Random();
    int[] hastighet = new int[1000];
    RandArray.Rand(r, hastighet, 60, 140);
    int sum = 0;
    for (int i = 0; i <= 999; i++)
        sum += hastighet[i];
    Console.WriteLine("\tMedelvärdet av 1000 möjliga hastigheter " +
                      "mellan 60 och 140 är:    "  +  sum/1000 + '\n');
  }
}
```

****

Ovn_3_7

*Modifiera programmet **Lista** (sid 129) så att sorteringen av slumptalen görs med vår egen bubbelsorteringsmetod sort() (sid 122) istället för med den fördefinierade **List**-metoden **Sort()**. Testa först med array-notationen som **sort()** är skriven i. Försök sedan att skriva om **sort()** till en **List**-version.*

```
using System;
using System.Collections.Generic;              // Krävs för List
class Lista
{
   static void Main()
   {
      List<int> intList = new List<int>();     // List-objekt av int
      Random r = new Random();
      int a = 1, b = 1000;
      Console.WriteLine(
              "\n\t100 heltal mellan " + a + " och " + b +
              " slumpas till ett List-objekt:\n");
      RandList.Rand(r, intList, a, b);         // Slump-tilldelning
      Print.Out(intList);                      // Osorterad utskrift
      Bubble.sort(intList);                    // List-sortering
      Console.WriteLine(
              "\tHeltalen sorteras med List-metoden Sort():\n");
      Print.Out(intList);                      // Sorterad utskrift
   }
}
```

BubbleList.cs (List-versionen av Bubble.cs sid 122)

```
Separat fil i samma projekt som filen Ovn_5_7.cs
Sorterar heltal lagrade i arrayen t med en bubbelsorteringsalgoritm
using System;
using System.Collections.Generic;
class Bubble
{
   public static void sort(List<int> t)
   {
      int temp;
      for (int pass=0; pass<t.Count-1; pass++)
          for (int i=0; i<t.Count-1; i++)
              if (t[i] > t[i+1])            // Sortering i stigande
              {                             // ordning
                 temp = t[i];               // Algoritm för platsbyte
                 t[i] = t[i+1];             // av de två elementen
                 t[i+1] = temp;             // t[i] och t[i+1]
              }
   }
}
```

Print.cs (sid 131)

Separat fil i samma projekt som filen Ovn_3_8.cs
*Metoden **Out()** skriver ut en lista med en **foreach**-sats som*
loopar igenom listans ALLA element

```
using System;
using System.Collections.Generic;
class Print
{
   public static void Out(List<int> t)
   {
      Console.Write("\t");
      int i = 0;
      foreach (int element in t)                 // Loop
      {
         Console.Write(element + " ");
         if ((i % 14 == 0) && (i != 0))          // Radbyte var
            Console.Write("\n\t");               // 14:e utskrift
         i++;
      }
      Console.WriteLine("\n");
   }
}
```

RandList.cs (sid 130)

Separat fil i samma projekt som filen Ovn_3_8.cs
Metod Next() slumpar fram heltal mellan a och b och
lagrar dem i ett List-objekt med List-metoden Add()

```
using System;
using System.Collections.Generic;
class RandList
{
```

```
   public static void Rand(Random r, List<int> no, int a, int b)
   {
      for (int i=0; i < 100; i++)              // Här fylls listan
          no.Add(r.Next(a, b));                // med slumptal
   }
}
```

****

Kapitel 4 Tillämpningar, sid 156:

Ovn_4_1

Skriv ett program som läser in en sträng, lagrar den i en array av **char** *och skriver ut den baklänges. Använd tekniken i programmet* **EncryptCharTest** *(sid 139) för att omvandla den inlästa strängen i en array av* **char**.

```
using System;
class Ovn_4_1
{
   static void Main()
   {
      Console.Write("\n\tSkriv in text:\t\t");
      char[] text = Console.ReadLine().ToCharArray();

      Console.Write("\n\tTexten baklänges:\t");
      for (int i = text.Length-1; i >= 0; i--)
          Console.Write(text[i]);
      Console.WriteLine('\n');
   }
}
```

****

Ovn_4_2

Skriv ett program som skapar en tom fil, skriver i den texten ”Den här texten kommer från mitt första C# filhanteringsprogram” och sedan läser från den samt skriver ut innehållet på skärmen. Som mall kan du ta programmet WriteReadFile (sid 142) och modifiera den.

```
using System;
using System.IO;
class Ovn_4_2
{
   static void Main()
   {
      string word;
      StreamWriter fileForWrite = new StreamWriter("Ovn_4_2.txt");
      fileForWrite.WriteLine(           // Skriver texten till filen
                             "\tDen här texten kommer från mitt " +
                             "första C# filhanteringsprogram.")   ;
      fileForWrite.Close();

      StreamReader fileForRead  =  new StreamReader("Ovn_4_2.txt");
      Console.WriteLine("\n\tFöljande text har skrivits från " +
                        "programmet till filen.\n\n\t"         +
                       "Nu läses den från filen:\n")          ;
      while (!fileForRead.EndOfStream)
```

```
      {
         word = fileForRead.ReadLine();   // Läser texten från filen
         Console.WriteLine(word);         // Visar texten på skärmen
      }
      fileForRead.Close();
      Console.WriteLine();
   }
}
```

Ovn_4_3

Modifiera programmet från övn 4.2 ovan: Istället för att hårdkoda texten i programmet, läs in den så att programmet skriver vilken inläst text som helst till filen och läser den sedan därifrån.

```
using System;
using System.IO;
class Ovn_4_3
{
   static void Main()
   {
      string word, text;
      Console.Write("\n\tSkriv en text som ska lagras i en fil:\t");
      text = Console.ReadLine();        // Läser texten från skärmen

      StreamWriter fileForWrite = new StreamWriter("Ovn_4_3.txt");
      fileForWrite.WriteLine(text);     // Skriver texten till filen
      fileForWrite.Close();

      StreamReader fileForRead  =  new StreamReader("Ovn_4_3.txt");
      Console.WriteLine("\n\tFöljande text har lästs från skärmen" +
                        " och skrivits till filen.\n\n\t"          +
                        "Nu läses den från filen och visas här:\n");
      while (!fileForRead.EndOfStream)
      {
         word = fileForRead.ReadLine();   // Läser texten från filen
         Console.WriteLine("\t\t" + word); // Visar texten på skärmen
      }
      fileForRead.Close();
      Console.WriteLine();
 }
}
```

Ovn_4_4

Varje gång man kör programmen Ovn_4_2 eller Ovn_4_3 efter första gången, rensas och återställs filen och endast den senaste texten hamnar i den. Skriv ett program som gör samma sak som Ovn_4_3 men bibehåller filens gamla innehåll och lägger till den nyinlästa texten utan att radera gammal data. Du kan åstadkomma det genom att öppna filen i append mode.

```
using System;
using System.IO;
class Ovn_4_4
{
   static void Main()
   {
```

```
      string word, text;
      Console.Write("\n\tSkriv en text som ska lagras i en fil:\t");
      text = Console.ReadLine();        // Läser texten från skärmen

      StreamWriter appendFil = new StreamWriter("Ovn_4_4.txt",
                                                append:true);
                                        // Filen öppnas för append
      appendFil.WriteLine(text);        // Inläst text läggs till
      appendFil.Close();                // filen

      StreamReader fileForRead  =  new StreamReader("Ovn_4_4.txt");
      Console.WriteLine("\n\tFöljande text har lästs från skärmen" +
                        " och lagts till filen.\n\n\t"          +
                        "Nu läses den från filen och visas här:\n");
      while (!fileForRead.EndOfStream)
      {
         word = fileForRead.ReadLine();    // Läser texten från filen
         Console.WriteLine("\t\t" + word); // Visar texten på skärmen
      }
      fileForRead.Close();
      Console.WriteLine();
   }
}
```

Ovn_4_5_Class

Modifiera klassen RandPasswd (sid 149) som genererar ett slumplösenord, genom att använda en annan, ny lösenordpolicy: 3 gemener, 2 versaler samt ? och @) och 2 specialtecken.

```
using System;
class RandPasswd_Ny
{
  public static void OnePassword(Random r, char[] p)
  {
    for (int i=0; i < 3; i++)
       p[i] = (char) r.Next(97, (122 + 1));// 3 små bokstäver
    for (int i=3; i < 5; i++)
       p[i] = (char) r.Next(63, (90 + 1)); // 2 versaler samt ? och @
    for (int i=5; i < 7; i++)
       p[i] = (char) r.Next(33, (47 + 1)); // 2 specialtecken
   }
}
```

Ovn_4_5_Test

Testa den nya policyn i programmet RandPasswdTest för att skriva ut de nya slumplösenorden samt tillhörande användarnamn till en fil

```
using System;
using System.IO;

class RandPasswdTest
{
   static void Main()
   {
      char[] password = new char[8];
      Random r = new Random();
```

```
      string word;
      Console.Write("\n\tHur många användarnamn med lösenord "  +
                                                  "vill du ha?    ");
      int antal = Convert.ToInt32(Console.ReadLine());
      StreamWriter fileForWrite = new StreamWriter("userPasswd.txt");

      for (int i=1; i<=antal; i++)
      {
          RandPasswd_Ny.OnePassword(r, password); // Slumplösenord
          fileForWrite.WriteLine("\tuser" + i +   // Skrivs till fil
                "\t\t" + new String(password));
      }
      fileForWrite.Close();

      StreamReader fileForRead  =  new
                                StreamReader("userPasswd.txt");
      Console.WriteLine("\n\tVarsågod, detta står nu" +
                        " i filen userPasswd.txt:\n");
      while (!fileForRead.EndOfStream)
      {
          word = fileForRead.ReadLine();        // Läses från fil
          Console.WriteLine(word);              // Skrivs till skärm
      }
      fileForRead.Close();
      Console.WriteLine();
   }
}
```

****

Programförteckning

Program	Ämne	Sida
Kapitel 1 Algoritmer och programmering		
Morgonsyssla	Algoritm: Ex. på pseudokod / flödesschema	20/23
PrimitivesCs	Enkla datatyper i C#	31
InputCs	Inläsning av data	34
(Un)CondInit	Villkorlig initiering	37
Collatz	Algoritm & program med selektion och repetition (loop)	39
Collatz_mod	Metoder och program i C#	42
Collatz_Test	Modularisering av programmet **Collatz**	42
MiniSort	Algoritm för platsbyte av två objekt	44
NoSort	Misslyckad modularisering av programmet **MiniSort**	45
CallByVal	Värdeanrop	48
CallByRef	Referensanrop	52
Swapping	Modularisering av programmet **MiniSort**	51
OutParam	In- och utparametrar	53
Kapitel 2 Logik för blivande programmerare		
AND_OR	De logiska operatorerna OCH och ELLER	63
TruthTab	Logiska variabler med datatypen **bool**, sanningstabeller	67
GuessNEG	*Gissa tal* med NEGATION som logisk operator Logiska uttryck, dubbel negation	69
Passwd	Programserien *Testa lösenord* med NEGATION **String**-metoden **equals()**	73
PasswdCaps	Test av två lösenord med De Morgans lag	75
Kapitel 3 Datastrukturer och abstrakta datatyper		
ArrayObj	Ny datastruktur av sammansatt typ	100
ArrayRef		105
Fish	Deklarerar klassen **Fish**	107
ArrayOfRef	Array av referenser till **Fish**-objekt	108
ArrayParam	Array som parameter i en metod	111
DoRand	Hantering av slumptal i C#	115
RandArray	Metod som slumpar fram en array av heltal	118
Search	Metod som söker efter ett element i en array	120
Bubble	Läser en tabell från en fil och visar innehållet	122
G_Bubble	Generiska metoder	122

Lista	Demonstrerar dynamiska arrays: Listor	129
RandList	Klassen **RandList**	130
Print	**foreach** i listor	131

Kapitel 4 Tillämpningar

EncryptStr	Kryptering av strängar	140
EncryptChar	Kryptering av text, tekenvis	140
RandPasswTest	Skriver till en fil ett antal användarnamn samt slumpvis genererade lösenord, läser från den och visar innehållet	147
RandPasswd	Metod som skapar slumpvis genererade lösenord	149
EncryptFile	Kryptering av filer med en slumpkrypteringsnyckel	152
EncryptText	Krypterar text (annan variant)	153
ReadShowFile	Läser en fils innehåll och visar det på skärmen	155
WriteFile	Skriver text till en fil	154

Kapitel 5 Datastrukturer i relationsdatabaser

FirstDatabase	Laddar en databas till C# och etablerar kontakt med den Visar databasens tabeller i en grafisk miljö	180
SQLclient	Skickar SQL-frågor från C# till databasen Visar frågornas resultat i en grafisk miljö	186
Kursverksamhet	Skapar en tom databas i C#, etablerar kontakt med den och fyller den med tabeller Specificerar tabellernas kolumner samt deras datatyper Definierar tabellernas primär- och främmande nycklar Bestämmer relationer mellan databasens tabeller Fyller tabellerna med data.	198

Register

A

Abstraktion 89
ADO.NET-objektmodellen 171
Algol 7
Algoritm 15
  Exempel 16
Argument 46
Array 98
  Default-initiering 102
  Definition 100
  Hakparenteser 102
  Indexering 99
  Indexregeln 99
  Initiering 100
  Parameter i metoder 111
  Referensanrop 111
Array av referenser 107
Arv 92
Assembler 7
Attribut 89, 199

B

Basic 8
`bool` 67
Bubbelsortering 121

C

Cobol 7
ComboBox 193
`Convert.ToInt32()` 36
CREATE TABLE-satsen 178

D

Data Definition Language 178
Data Sources 181
Databas 159
  Modularisering 161
Databasmodellering 198
Databasobjekt 205
DataGridView-kontroll 180
Datamedlem 91
`DateTime`-klassen 66
De Morgans lagar 76
Deklarativt språk 170

E

Element 98
Entitet 199
Entity-Relationship Modeling 199
`Equals()` 74

F

Filhantering 142
  Append 145
  Append mode 145
Flödesplan 22
  Exempel 23
Fortran 7
FORTRAN 7
Främmande nyckel 168
Fält 162

H

Händelsestyrd programmering 12
Högnivåspråk 7

I

Identity 178, 201
Implementation 94
Indata 34
Index 98
Indexregeln 99
Inmatning 34
Instruktion
  Huvudinstruktion 20
  Underinstruktion 20

J

Java 10
Join 173

K

Klient-Server-modellen 170
Kontrollstruktur 22
Kryptering 139
  Fil 151
  Filer 151
  Text 139
Kryptering av filer 151
Kursverksamhet 198

L

Label 12
LIKE 177
Lista 129
Logisk operator 62
  ELLER 65
  NEGATION 69
  OCH 64
  Programexempel 64
Logiska lagar 64
Lågnivåspråk 7

M

Maskinkod 18
Metod 41, 91
  Begreppet 41
Modularisering 93
Mönstermatchning 177

N

NEGATION 69
  Dubbel 71
NULL i SQL 162

O

Objekt 89
Objektorienterad design 14, 88
Objektorienterad programmering 14, 88
Operator
  Logisk 69
ORDER BY 176

P

Paradigmskifte 14, 88
Parameter 41
Pascal 8
Polymorfism 92
Post 162
Primärnyckel 168
Programmering 13
  Historik 6
Projektion 173
Pseudokod 20
Punktnotation 90

R

Radsortering 176
Referensanrop 48
Relation 164
Relationsdatabasmodellen 160
Returvärde 41

S

Sanningstabell 64
Script 179
SELECT-satsen 173
Selektion 173, 174
`slumpArray`-klassen 116
Slumplösenord 147
Slumptal 115
  Array 116
Sortering 121
  Platsbyte 39, 44
SQL 172
  Regler och konventioner 179
SQL i C# 186, 211
SQL-klient 186
Structured Query Language 172
Strukturering av kod 93
Sökning 119

T

Tabell 161
  Liknelse med klass 163

U

UML 14, 19, 88, 91
Uttryck
  Logiskt 71

V,W

View Designer 183
Villkor 21
Villkorlig initiering 36
`WHERE`-satsdelen i SQL 175

Å

Återanvändning av kod 93