Operativsystem:

Alle de følgende eksempler er testet på Windows **98** og Internet Explorer **5.5.** Dette er i .NET en **minimums** installation. Der er nogle hjørner af .NET som ikke er tilgængelige med denne installation. Det fulde .NET (SDK .NET) kræver installation af Windows **XP**.

Men der er tale om **små** hjørner – og programmeringssproget C# fungerer fint sammen med Windows 98!

Vi vil altså indtil videre gå ud fra, at du har installeret denne minimumsinstallation!

Installer Microsoft .NET Runtime:

For at komme i gang er det nødvendigt at installere Microsofts .Net Runtime på din maskine.

Filen kan downloades fra http://msdn.microsoft.com.

Den downloadede fil – du skal søge efter den udgave af .NET som er 'redistributable' - hedder dotnetfx.exe eller **dotnetredist.exe** (ca 20 Mega Byte), en Win32 Cabinet Self Extractor. Denne fil er en minimums udgave af Dotnet.

Den store udgave – SDK – fylder ca 135 MegaByte og forudsætter at Windows XP er installeret.

SDK indeholder især megen dokumentation og nogle flere basisklasser.

På dette kursus vil vi have den lille udgave – dotnet redistributable – som **grundlag**.

Gør følgende:

1. Opret en ny mappe på skrivebordet 'dotnet'
2. Dobbeltklik filen
3. Vælg at ekstrahere filerne til den nye mappe 'dotnet'
4. Åbn mappen dotnet og dobbeltklik filen dotnetfx.exe.
5. Du guides nu igennem installationen
6. Programmet installerer først og fremmest en række EXE og DLL filer i en ny mappe: C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705.
7. De vigtigste er: **mscorlib.dll** (som indeholder den cenrale kode til .NET Runtime) og **csc.exe** (som er en compiler til C#). Der installeres også andre compilere fx **vbc.exe** til Visual Basic (herom senere).
8. NB 'path' eller stien til de nye EXE filer er altså: C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705.
9. Åbn filen C:\autoexec.bat og tilføj den nye sti sådan:
10. set path=C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705;%PATH%
11. Ellers kan dit operativ system ikke finde filerne fx compileren til C#!!

12. Du kan også skrive en bat fil ved navn **sharppath**.**bat** med indholdet:
    "path=C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705".
13. Hvis du dobbeltklikker bat filen sættes path korrekt.

Mit første program: Virker .NET Runtime?

Formålet med det følgende er især at teste om det installerede .NET Runtime virker.
Åbn Notestblok (eller en anden editor som kan gemme tekst **uformatteret**!) og skriv følgende C# kode som et testprogram (koden vil blive gennemgået lidt senere):

```
public class Test{
  public static void Main(string[] args){
                System.Console.WriteLine("Hello World!");
  }
}
```

Opret en ny mappe til dine C# filer fx: C:\csharp\test og gem koden som Test.cs. ALLE C# programmer (tekstfiler) skal have filtypen .cs. Ellers kan de ikke kompileres. Navnet på CS filen er i øvrigt ligegyldigt. **Test**.cs, **test**.cs eller **mitførsteprogram**.cs er alle OK (modsat reglerne i Java).

Åbn et DOS kommando prompt vindue og styr til mappen C:\csharp\test.

Tast: **csc Test.cs**.

Hvis alt går godt får du følgende output:

```
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.
```

Denne besked betyder følgende:

1. tekstfilen Test.cs er nu kompileret til en EXE fil ved navn Test.exe
2. kompileringen har ikke givet anledning til nogen fejl (error) eller advarsel (warning).

Hvis IKKE alt går som det burde, vil du få fejl meddelelser som fx:

```
error CS2001: Source file 'a.cs' could not be found
fatal error CS2008: No inputs specified
```

I dette tilfælde er indtastet csc a.cs – en fil som ikke findes i mappen!

Kompileren csc giver altså fejl/advarsler som kan bruges til at rette programmet til.
Prøv fx at slette semikolon efter sætningen: System.Console osv. Slet også Test.**exe** i mappen.
Gem filen Test.cs og prøv at kompilere den med: csc Test.cs.

Der fås nu en fejl meddelelse og der oprettes IKKE nogen EXE fil (Kompileren kan **kun** oprette en EXE fil hvis C# tekst filen er – syntaktisk - uden fejl – **men** en EXE oprettes selv om der er mange warnings!):

```
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

Test.cs(3,42): error CS1002: ; expected
```

Kompileren **gætter** på at der mangler et semikolon efter sætningen. System… Det manglende semikolon er en '**syntaktisk'** fejl. Kompileren kan opdage **'syntaktiske'** og **'semantiske'** fejl men ikke 'logiske' fejl. En cs fil kan altså godt indeholder mange logiske fejl og alligevel kompileres uden problemer! Mange programmer også professionelle har faktisk logiske fejl! (Mere herom senere).

Csc kompilerer tekstfilen til **'binær** kode' eller en 'binær' (dvs til en EXE, DLL eller NETMODULE fil). I dette eksempel fylder tekstfilen Test.cs 117 bytes, men den kompilerede binære kode (Test.exe) fylder 3.000 bytes – der er altså tale om to vidt forskellige objekter!!

Binær kode er kode som maskinens CPU (**processor**) kan 'forstå' og direkte anvende til at køre et program. Binær kode består af 0 og 1 taller – populært sagt.

Kompileren kan ikke skrive til en EXE fil hvis programmet allerede kører!! Husk derfor at lukke et program inden du kompilerer det igen! Dette glemmes ofte med Windows programmer, hvor kompileringen så giver en fejl.

Kør programmet:

Programmet køres ved at taste programmets navn i kommando prompten: **Test** (eller: **Test**.**exe**). Output bliver en linje skrevet til skærmen: "Hello World!" og et linjeskift til ny linje. Vi har altså produceret i virkeligt **program** eller **'applikation'**.

Du kan også dobbeltklikke ikonet Test.exe i Windows Stifinder – men DOS vinduet forsvinder med det samme. Løsningen herpå er simpelt hen at tilføje en ny linje nedenunder System.Console…:

```
…
System.Console.Read();
```

Programmet vil du stoppe op og vente på **input** fra brugeren. DOS vinduet vil nu først lukke når du har tastet ENTER. **Read**() betyder: Vent indtil brugeren har indtastet noget på linjen.

Kompiler med parametre eller flag:

C# kompileren kan startes med en lang række af forskellige **'flag'** eller parametre.

Her følger selve listen. En del af disse flag vil blive forklaret efterhånden:

```
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

                Visual C# .NET Compiler Options

                - OUTPUT FILES -
/out:<file>           Output file name (default: base name of file with main class or first file)
/target:exe           Build a console executable (default) (Short form: /t:exe)
/target:winexe        Build a Windows executable (Short form: /t:winexe)
/target:library       Build a library (Short form: /t:library)
/target:module        Build a module that can be added to another assembly (Short form: /t:module)
/define:<symbol list> Define conditional compilation symbol(s) (Short form: /d)
/doc:<file>           XML Documentation file to generate

                - INPUT FILES -
/recurse:<wildcard>   Include all files in the current directory and subdirectories according to the
wildcard specifications
/reference:<file list> Reference metadata from the specified assembly files (Short form: /r)
/addmodule:<file list> Link the specified modules into this assembly

                - RESOURCES -
/win32res:<file>      Specifies Win32 resource file (.res)
/win32icon:<file>     Use this icon for the output
/resource:<resinfo>   Embeds the specified resource (Short form: /res)
/linkresource:<resinfo> Links the specified resource to this assembly (Short form: /linkres)

                - CODE GENERATION -
/debug[+|-]           Emit debugging information
/debug:{full|pdbonly} Specify debugging type ('full' is default, and enables attaching a debugger to
a running program)
/optimize[+|-]        Enable optimizations (Short form: /o)
/incremental[+|-]     Enable incremental compilation (Short form: /incr)

                - ERRORS AND WARNINGS -
/warnaserror[+|-]     Treat warnings as errors
/warn:<n>             Set warning level (0-4) (Short form: /w)
/nowarn:<warning list> Disable specific warning messages

                - LANGUAGE -
/checked[+|-]         Generate overflow checks
/unsafe[+|-]          Allow 'unsafe' code

                - MISCELLANEOUS -
@<file>               Read response file for more options
/help                 Display this usage message (Short form: /?)
```

/nologo Suppress compiler copyright message
/noconfig Do not auto include CSC.RSP file

- ADVANCED -
/baseaddress:<address> Base address for the library to be built
/bugreport:<file> Create a 'Bug Report' file
/codepage:<n> Specifies the codepage to use when opening source files
/utf8output Output compiler messages in UTF-8 encoding
/main:<type> Specifies the type that contains the entry point (ignore all other possible entry points) (Short form: /m)
/fullpaths Compiler generates fully qualified paths
/filealign:<n> Specify the alignment used for output file sections
/nostdlib[+|-] Do not reference standard library (mscorlib.dll)
/lib:<file list> Specify additional directories to search in for references

---

Gennemgang af programmet Test:

Med en lille ændring ser vores Test.cs sådan ud:

```
//fil: test.cs
//postcondition: outputter en linje til skærmen

public class Test{
  public static void Main(string[] args){
                System.Console.WriteLine("Hello World!");
                System.Console.Read();
  }
}
```

Øverst er tilføjet 2 linjer som starter med //. Linjer som starter med // er **kommentarer** som skal gøre programmer mere forståelige. Kompileren springer altid disse linjer over – dvs her kan skrives hvad som helst **uden** krav til regler eller syntaks. Brug altid så mange kommentar linjer at programmet er læseligt og forståeligt.

Selve koden starter med en klasse erklæring som omfatter hele koden: **public class Test{}**.

public: betyder at klassen kan bruges 'udefra' af alle andre klasser/programer .(Mere herom når vi kommer til Objekt orienteret Programmering).

class: ALLE programmer i C# er en class – ligesom i Java (men modsat af C++).
ALLE programmer **skal** altså starte med **public class** (evt kun: **class**) XXX.

{} definerer en 'blok' dvs alt efter { og før } hører sammen i en blok – her definerer { og } klassens start og slutpunkt.

Inden i denne klasse findes kun en **metode**: **public static void Main()**:

Denne standard metode **skal** findes i alle C# **applikationer** dvs i alle C# programmer som skal kunne køres som et program (som vi gjorde ved at taste Test på DOS linjen). Metoden Main()(NB 'Main' med stort M modsat i C++) er programmets 'entry point'. Hvis Main() fjernes fra koden vil en kompilering give en fejl (Prøv det!):

error CS5001: Program 'Test.exe' does not have an entry point defined

En metode i C# har til formål at udføre en handling, at gøre noget.
Metoder er normalt **public** dvs de kan kaldes 'udefra'. **Static:** betyder at metoden kaldes på en klasse og ikke på et objekt (mere herom senere). **Void:** betyder at metoden Main() ikke returnerer noget som helst.

Main(string[] args) : 'args' er et array eller en liste over de evt. parametre som programmet kaldes med. Fx kunne Test kaldes sådan: Test 1 2 3. De 3 tal gemmes i listen 'args' og kunne bruges som input parametre til programmet. (Se eks senere). s**tring:** er en type/klasse. En string kan gemme en hvilken som helst værdi som fx '88' eller 'Johnny'.

NB: Der er i C# tradition for at skrive **string** og ikke **String** – men traditionen er vaklende. **String** er en klasse: System.String og en basal type. Så **både** string **og** String er OK!

Main() indledes lige som klassen med en { og slutter med en }. Main() er altså en blok.
Inden i blokken står to **sætninger** (en 'sætning' er en linje som afsluttes med ;).

```
System.Console.WriteLine("Hello World!");
System.Console.Read();
```

WriteLine(): er en metode som udskriver den streng/tekst som den har som **parameter –** dvs den tekst som står i citat mellem parenteserne. ALLE metoder kaldes med parenteser – de viser at der er tale om en metode. De parametre ('input') som metoden anvender står mellem parenteserne.
Read(): stopper programmet og venter på at brugeren indtaster eet tegn (fx 'a');.

Problemet lige nu – set fra kompilerens side – er: **hvilken** WriteLine metode tales der om og **hvor** findes den? Kompileren har en 'resolver' som prøver at finde ud af hvad jeg mener. (Problemet er det samme med Read()). Disse metoder er jo IKKE beskrevet i Test.cs!!

C# anvender her 'namespaces' således: WriteLine() er den metode som findes i namespace 'Console' som igen findes inden i namespace 'System'. De forskellige namespaces som findes i forskellige DLL filer kan ses i .NET mappen i Windows (se tidl).
Som det vil fremgå senere kunne jeg teoretisk have skrevet min egen metode 'WriteLine()' – derfor skal kompileren vide **hvilken** metode jeg taler om.
For at gøre det mere 'brugervenligt' kan man i C# indlede koden med en **using** sætning og derefter undgå at skrive den fulde 'sti' til metoderne. Koden kan altså skrives således:

```
//fil: test.cs
//postcondition: outputter  4 linjer til skærmen

using System;
```

```
public class Test{
  public static void Main(string[] args){
                Console.WriteLine("Dette er C Sharp.");
                Console.WriteLine("Programmet anvender en using System.");
                Console.WriteLine("using System er en slags shortcut!");
                Console.WriteLine("Hello World!");

                Console.Read();//teknisk af hensyn til Windows, stopper vinduet
  }

}
```

Som det kan ses er det væsentligt lettere at anvende en 'using' hvis der skal skrives/læses meget til/fra skærmen – derfor er output sætningerne her forøget og ændret!
Men det er meget vigtigt at forstå at 'using' kun er en 'tekst-hjælp' – 'using xxx' betyder **ikke** at kompileren inddrager/refererer de nødvendige DLL filer. Når Test.cs kan kompileres uden videre skyldes det at kompileren **csc automatisk** inddrager mscorlib.dll (som indeholder .NET og C# kerne klasser) (NB: 'using' fungerer altså **ikke** som import sætninger i Java eller som #include i C++!!).(Mere herom senere).
Som det ses kan en kommentar starte midt på en linje – resten af linjen er så ren kommentar og ignoreres af kompileren.

Signaturer for Main():

Main() metoden i C# kan skrives på 4 tilladte måder:

1. public static void Main(string[] args){}
2. public static int Main(string[] args){}
3. public static void Main(){}
4. public static int Main(){}

Hvis der skal anvendes input/parametre fra brugeren anvendes nr 1 eller 2.
Hvis man ønsker at programmet skal returnere en værdi til operativ systemet (fx return 0; eller return –1;) anvendes nr 2 eller nr 4, som også kan anvendes til at standse Main() et bestemt sted i programmet – så at sige 'i utide' (Senere et eksempel herpå).
Oftest anvendes nr1.
Det er vigtigt at huske at kommando linje parametre altid er **strings** (tekst) – hvis de skal anvendes som tal skal de **konverteres**!
En metodes **signatur** er metodens hoved uden selve koden i blokken. Signaturen definerer metodens **parametre** (fx string[]), **navn** (fx Main), **access** (fx public) og **retur** værdi (ex void eller int eller string…).

Generelt:

C# er 'case sensitive' dvs at et 'keyword' (reserveret ord) som fx 'while' IKKE må skrives 'While' eller Main() må IKKE skrives 'main()'.

Det er en meget god ide at genbruge kode som engang er skrevet til nye programmer. Dvs når du starter på en ny kode – start med den gamle cs-fil (som **skabelon**).
På den måde undgås det at skulle skrive en række gentagne formularer igen og igen – og fejl undgås!!

Der kan altid anvendes mellemrum eller linjeskift for evt at gøre koden lettere at læse. Følgende to eksempler opfattes **ens** af kompileren (dvs giver den samme binære kode/samme program):

```
if(x==22){y=9;u=7;i=44;}else{c='a';}
```

```
if(x==22){
   y=9;
   u=7;
   i=44;
}
else
{
   c='a';
}
```

Normalt vil den nederste boks være meget nemmere at forstå: hvis x er lig 22 så gør sådan og sådan ellers (hvis x ikke er lig 22) gør sådan og sådan.
Det er en virkeligt god ide at anvende **indrykninger** og **linje** skift på denne måde.

```
C:\csharp>test
Dette er C Sharp.
Programmet anvender en using System.
using System er en slags shortcut!
Hello World!
```

Note: Redirection eller omdirigering:

I Windows **DOS** kan man **redirecte** et programs udskrift. Hvis vi har et program **test** som udskriver en række data til skærmen med Console.WriteLine() kan vi i kommandoprompten skrive:

test > dokumentation.txt

Output bliver så redirectet til en fil dokumentation.txt og der skrives **intet** ud til særmen!
På samme måde kan udskriften omdirigeres til printeren med

test > prn

og udskriften bliver så udskrevet på **printeren**! (Output kan også redirectes til COM porte mv).

Dette er altså en nem måde at arbejde med filer på! Som vi siden skal se – er dette et eksempel på det som i C# kaldes **Streams**.

Omdirigering kan også gøres programmatisk i selve koden sådan:

```
using System;
using System.IO;

public class Omdiriger_konsol
{
  public static void Main()
  {
    // Console.WriteLine omdirigeres/re-directes til en fil:
    StreamWriter sw = new StreamWriter (
                new FileStream ("konsol.out", FileMode.Create));
    Console.SetOut(sw);

    Console.WriteLine("DETTE VIL BLIVE SKREVET TIL EN FIL...");
    sw.Close();
  }
}
```

Klassen Console har simpelthen en metode som hedder **SetOut**() som kan bruges til en omdirigering! Teksten ”DETTE VIL” osv bliver nu indskrevet i en fil ’konsol.out’.

Test i Windows udgaven:

Formålet med dette afsnit er at vise hvad der sker når C# skal ‘**referere’** eller bruge klasser, metoder, objekter som ikke automatisk inddrages af compileren – som ligger uden for ‘kernen’.

Start med at gemme Test.cs under et nyt navn som ’WinTest.cs’. Målet er nu at få det til at køre som Windows program.

Den nye WinTest.cs skal se sådan ud:

```
//fil: WinTest.cs
//postcondition: Viser en messagebox

using System.Windows.Forms;

public class Test{

  public static void Main(string[] args){
                string nr1="Dette er C Sharp.";
                string nr2="Dette er en System.Windows.Forms.MessageBox!\nHello World!";
                MessageBox.Show(nr2,nr1);
  }

}
```

Forklaring:

Hvis koden er blevet korrekt kompileret (csc WinTest.cs) viser WinTest.exe en messagebox med en titellinje og en tekst delt på 2 linjer. Tegnet '\n' angiver linje skift. (Escape tegn idet alle escape tegn indledes med \ og de er format koder som ikke udskrives).
Koden erklærer 2 string-objekter, som bruges til boksens titel og tekst.
Klassen MessageBox findes i namespace 'Forms' som findes i 'Windows' som findes i 'System'.
Ved at bruge 'using' undgås at skrive hele 'stien' hver gang. Klassen MessageBox har en metode 'Show()' som tager 2 parametre: en tekst og en titel.
Prøv at fjerne using-linjen og kompiler programmet. Kompileren meddeler nu en fejl:

```
WinTest.cs(11,2): error CS0246: The type or namespace name 'MessageBox' could not be found
(are you missing a using directive or an assembly reference?)
```

Kompileren forstår ikke henvisningen til 'MessageBox': hvilken/hvor?

Prøv derefter at tilføje en linje til Main() således:

```
public static void Main(string[] args){
                string nr1="Dette er C Sharp.";
                string nr2="Dette er en System.Windows.Forms.MessageBox!\nHello World!";
                Assembly.Load("System.Windows.Forms");
                MessageBox.Show(nr2,nr1);
  }
```

Den nye sætning loader/indlæser den DLL fil eller det namespace som vi ønsker at anvende. Sætningen er den mest sikre metode til at referere eller inddrage bestemte klasser som er 'eksterne' dvs ikke findes i denne cs-fil. Hvis du har problemer med at få dette Windows program (eller andre) til at køre skulle Assembly.Load() sætningen klare alle problemer. Senere i kurset vil vi skrive vore egne DLL filer og hvis der er problemer med at kunne 'referere' dem skulle Assembly.Load() også kunne klare disse problemer.

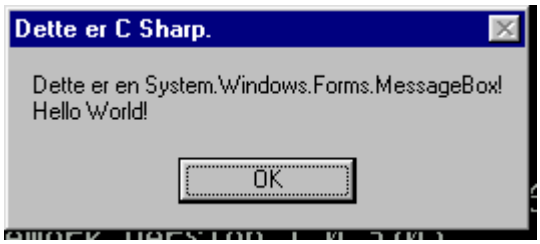
En alternativ metode er at kompilere filen WinTest.cs med en reference parameter. Skriv på kommandolinjen i prompten:

csc /reference:System.Windows.Forms.dll WinTest.cs
Eller**:**
csc /r:System.Windows.Forms.dll WinTest.cs

/reference: betyder at kompileren skal inddrage den pågældende dll fil.

Mange referencer kan stå efter hinanden med semikolon imellem fx således:

csc /r:System.Windows.Forms.dll;System.Drawing.dll;System.IO.dll WinTest.cs



Installer SharpDevelop:

Alle kode eksempler i dette kursus kan uden vanskelighed skrives i en primitiv editor som Notesblok i Windows. Men et program som SharpDevelop giver trods alt mange fordele og gør tingene lidt nemmere. Problemet er nogen gange at der produceres en del automatisk kode af programmet som programmøren måske ikke helt forstår!

Programmet SharpDevelop kan frit downloades (programmet er et GNU General Public License projekt) fra Internettet på adressen: http://icsharcode.net .

Filen hedder 092setup.exe på ca 3,2 Mbyte.

Når programmet er installeret kan du oprette forskellige typer af projekter (Combines) i C#, Visual Basic og Java.

Den kode som skrives farves efter syntaksen så at man i nogen grad kan se om ens kode er syntaktisk korrekt.

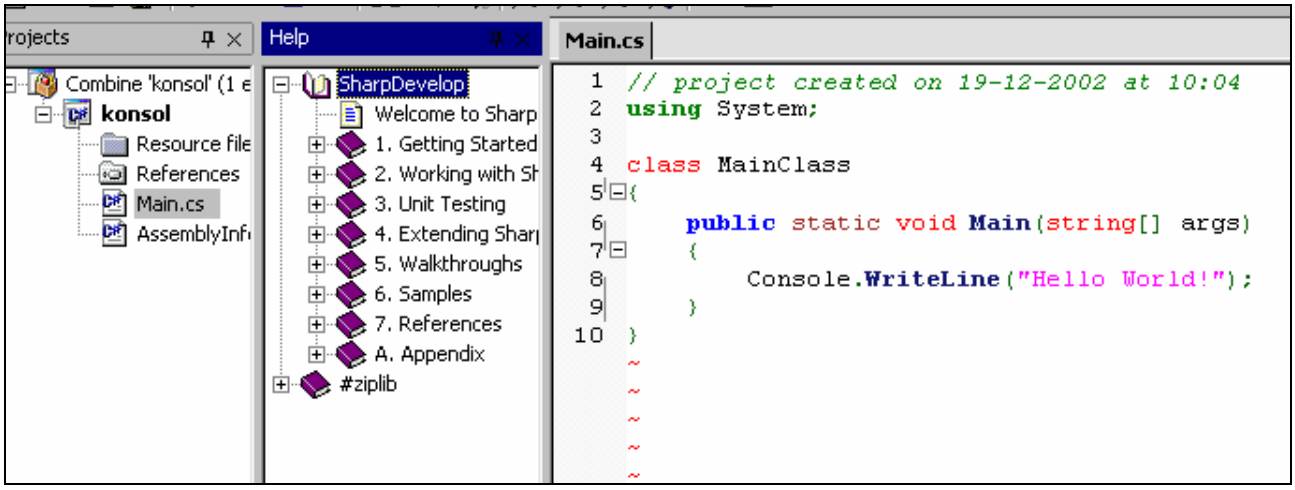
Som vi skal se senere er det også muligt at bruge SharpDevelop som en **'builder'** af grafiske windows programmer.

I SharpDevelop kan tilføjes en reference til projektet og gøres mange ting som vi her har gjort manuelt på kommando linjen.

Et projekt kan også kompileres og køres via SharpDevelop.

SharpDevelop har masser af Hjælp – brug den!

Et eksempel fra SharpDevelop:



Det bedste værktøj til C# og .NET er måske Microsofts professionelle Visual Studio.NET. Dette program er absolut ikke 'freeware', men der er visse muligheder for at bruge programmet on-line og

i nogle perioder kan man faktisk også downloade en evaluerings kopi af programmet. Oplysninger herom kan findes på Internettet.

Struktureret programmering:

At programmere på en fornuftig måde i C# indeholder to aspekter: **struktureret** og **objektorienteret** programmering.
De kommende mange afsnit handler om struktureret programmering.

Målet med **struktureret** programmering er at koden skal være modulopdelt og let forståelig. Et eksempel på meget ustruktureret kode ('spaghetti kode') ville være følgende:

```
//fil: spagetti.cs

//postcondition: Viser spaghetti kode eks - udskriver sætning:

using System;

public class Test{

public static int Main(string[] args){

                goto Z;
                X:
                Console.WriteLine("er"); goto V;
                Y:
                Console.WriteLine("C#"); goto EXIT;
                Z:
                Console.WriteLine("Dette"); goto X;
                V:
                Console.WriteLine("ogsaa"); goto Y;
                EXIT:
                return 0;

}

}
```

Hvis du indtaster programmet vil du opdage at det faktisk **virker** perfekt.

goto gør at programmet hopper rundt fra 'label' til label (en label er et tegn/tekst med et kolon efter som f.eks. **EXIT:**). På denne måde bliver koden meget vanskelig at læse og forstå. Meningen med struktureret programmering er at undgå denne slags 'spaghetti kode' (som var mere almindelig i 'gamle dage' i BASIC programmering). Eksempler på strukturerede sprog var i sin tid sprog som **Pascal** og **C** (fra omkring 1970).

Kommentarer:

Som nævnt er det vigtigt at forsyne koden med så mange **kommentarer** at andre (og en selv!) kan forstå koden.

Der findes i C# 3 slags kommentarer:

```
//  enkelt linje kommentar

/*
fler-
linje-
kommentar
*/

///  XML kommentar som kan producere et XML dokument (a la Javadoc, Herom senere).
```

Kommentarer **ignoreres** helt af kompileren. Brug kommentarer til at forklare programmets flow og gang.

Skriv **rigeligt** med kommentarer!

Keywords:

En række ord er reserverede eller keywords i C# - dvs de må **ikke** bruges som navne på variable. Eksempler på keywords er: class, if, string, public.

Variabelnavne/Identifiers:

Som kode eksempel se følgende

```
const int MAX=100;
string fornavn=”Erik”;
string efternavn;
efternavn=”Hansen”;
int side1=12;
int side2=33;
side2=123;
int areal=side1*side2;
```

C# kode består af sætninger som afsluttes med et semikolon. I eksemplet erklæres/oprettes 2 **variable** fornavn og efternavn som tildeles en **værdi**. En **literal** er en værdi som’44’ eller ’erik’. En literal ’er hvad den er’ – deraf betegnelsen **literal** (bogstavelig).

En **variabel** er et sted eller en adresse i RAM hukommelsen som tildeles et navn fx ’fornavn’.

Når en variabel erklæres tildeles den det antal bytes i RAM som er nødvendigt. Dvs. den **'allokeres'**. Fx bevirker sætningen 'int side1=12;' at der tildeles 4 bytes et bestemt sted (fx byte nummer 1000,1001,1002,1003 i RAM) og at tallet/værdien/literalen 12 gemmes på denne adresse.

De 4 bytes ser fx sådan ud:

byte 1003	byte 1002	byte 1001	byte 1000
0	0	0	12

Enhver identifier eller ethvert variabel-navn **skal** først erklæres med en **type**. C# skal vide hvilken type fornavn er: er det et tal, en tekst…? (Som det vil fremgå senere kan man oprette sine egne typer – det er selve ideen med OOP, objekt orienteret programmering. En type er bl.a. en klasse):

Variablen kan oprettes på en linje og tildeles en værdi på en anden linje og kan til hver en tid tildeles en ny værdi (se eksemplet med side2).

Sætningen **'string** efternavn;' er en **erklærings** sætning. Sætningen 'efternavn="Hansen;"' er en **tildeling** eller **assignment**.
Sidste sætning viser hvordan en ny variabel kan tildeles en værdi ved at gange 2 andre variable. (De almindelige 4 matematiske operatorer er: +, -, *, /).

Variabelnavne-**regler**:

1. et navn kan indeholde bogstaver, tal og understregninger
2. det **første** tegn i navnet **skal** være et bogstav
3. C# er case sensitive: **Fornavn** er ikke det samme som **fornavn**
4. **reserverede** ord må ikke bruges som navne

Konstanter skrives normalt med kun store bogstaver (versaler).

I eksemplet er MAX en konstant: erklæret med **const** int MAX. Dette betyder at MAX **ikke/**aldrig kan ændre værdi mens programmet afvikles. Programmet KAN ikke indeholde en linje som: **MAX=23;** når først konstanten er erklæret.

int er en type som betegner et helt tal uden decimaler - jvf senere.

I C# er det skik og brug (i nogen grad!) at anvende navne med små bogstaver og understregninger:

```
string kunde_navn;
string kunde_adresse;
```

Det er meget vigtigt at anvende navne som er **intuitive** og **beskrivende**. Hvis programmet skal gemme kundens navn i en variabel er der intet i vejen for at kalde variablen for: string xyz. MEN det vil blive ulige nemmere at forstå programmet og især undgå fejl hvis denne variabel kaldes: string kunde_navn!

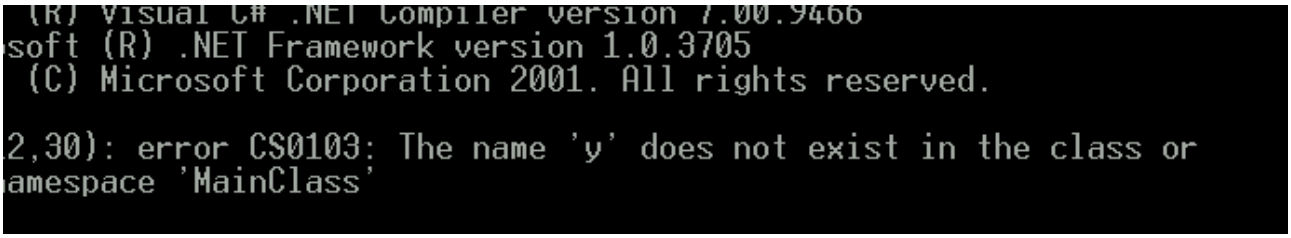
Variable og scope:

En variabel som int x har en **levetid** eller **scope** eller et område, hvor den er **'synlig'**. Helt grundlæggende opstår den i forbindelse med programmets start eller en metodes start og **dør** ved programmets eller metodens slutpunkt.

Se eks dette kode fragment:

```
public static void kvadrat(int x){x=x*x;int y=x;}

public static void Main(string[] args)
{
        int x=22;
        kvadrat(x);
        //NB dette giver kompiler fejl:
        //Console.WriteLine("y: {0}",y);

        //OBS x er stadig 22:
        Console.WriteLine("x: {0}",x);
        Console.Read();
}
```

Hvis man prøver at udskrive variablen y melder compileren fejl:



Variablen y er simpelt hen **ikke** synlig i Main() – den er **kun** synlig i den blok (fra { til }) som består af metoden kvadrat(). Y fødes og dør inden i metode blokken og er usynlig i Main().

Læg også mærke til at x ikke skifter værdi selv om metoden kvadrat sætter 'x' til x*x – men dette x inden i metoden er en lokal variabel som fødes og dør i metoden blokken. Når Main() bagefter udskriver x er x stadig det samme!

I C# kan man gennemtvinge at en variabel bruges som **reference** type med det reserverede ord **ref.** Ovenstående kunne omskrives således:

```
public static void kvadrat(ref int x){x=x*x;int y=x;}

public static void Main(string[] args)
{
        int x=22;
        kvadrat(ref x);
        //NB dette giver kompiler fejl:
        //Console.WriteLine("y: {0}",y);

        Console.WriteLine("x: {0}",x);
        Console.Read();
}
```

Når x udskrives er x nu 22*22, fordi metoden databehandler **selve** x (altså en **reference** til x) og **ikke** en kopi (**value** type) af x.

Static variable:

I C# kan erklæres **static** variable (modsat almindelige lokale variable) som er **klassevariable** som bevarer/husker deres værdi.

F.eks. kan static variable bruges som **tællere** – jvf dette lille kode eksempel:

```
static int antal=0;//tæller antal metode kald:

//to metoder som kaldes:
public static void engangtil(){antal++;}
public static void engangtiligen(){antal++;}

public static void Main(string[] args)
{
```

```
            for(int i=0;i<6;i++){
                engangtil();
                if(i%2==0)engangtiligen();//giver 9 metode kald i alt
            }
    }
```

Hvis antal udskrives giver den værdien 9 i dette tilfælde. Vi skal senere se på **static metoder** i klasser som er et udbredt fænomen i C#.

Fx er Console.Write() og Math.Sqrt() eksempler på **static** metoder som kaldes **direkte** på klassenavnet (Console og Math). En sætning som: int x = Math.Sqrt(16); -lægger 4 over i x.

C# indbyggede basale typer:

keyword	Antal **bytes**	**Eksempel/værdier**:
byte	1	Fra 0 til 255
sbyte	1	Med fortegn: -128 til +127
char	2	Unicode/også ikke eng. tegn
short	2	-32768 til +32767
ushort	2	Uden fortegn max 65535
int	4	2 i potens 31 – ca 2,1 milliard
uint	4	2 i potens 32 - ca 4,2 milliard
long	8	Max: 9223372036854775807!
ulong	8	Uden fortegn
float	4	Max: 3.4*10 i potens 38
double	8	Max: 1.7*10 i potens 308
decimal	16	Præcision: 28 pladser
bool	1	'true' eller 'false', IKKE 0 el. 1!

Et tegn gemmes som en **char,** der i C# er 2 bytes eller 16 bits. Alle tegn gemmes som en talværdi. Dvs. at der kan opereres med tegn fra nummer 0 til nummer 65.535!! På denne måde (Unicode systemet) kan langt flere sprog få deres tegn/bogstaver repræsenteret end tidligere med ASCII tabellen (hvor en char kun fyldte 1 byte). Eksempel: tegnet 'a' gemmes som værdien 97, 'b' som 98 osv.

Typen **decimal** er nødvendig ved matematik på decimaltal som ellers kan give fejl – jvf at decimal har en præcision på 28 pladser (det har double ikke). Eksempel: 10.0 – 9.9 kan give resultatet: 0.09999999 hvis de to tal er erklæret som double.

Ved mindre talstørrelser anvendes typisk ved hele tal **short** eller **int** (int x=1234567) og ved decimaltal **float** eller **double** (double x=1234.5678). Jo tungere en type der vælges jo mere fylder programmet i RAM og det kan nogle gange blive et problem.

Opgaver:

1. Skriv et program hvor du udskriver alle de forskellige numeriske typers **maximale** og **minimale** værdi. Dette kan ske ved formlen:

int.MaxValue;
int.MinValue;
Osv med de andre typer (fx double.MaxValue).

NB string er IKKE en 'basal type' men en **klasse** i C#: System.String. Men i virkeligheden er forskellen minimal fordi typer som int, byte eller char er et alias for klasserne Int32, Byte og Char, så reelt er alle typer i C# klasser - direkte eller indirekte.

I stedet for at skrive:

string s="Hej";
double tal=44.66;

kan altså **lige så godt** skrives:

System.String s="Hej";
System.Double tal=44.66;

Det første er nemmere og derfor mest brugt. NB klasser i C# staves **normalt** med stort begyndelses bogstav! (Og ofte med **'kamel** notation': op og ned – fx: **TcpListener** med stort T og L).

Value og reference:

De basale typer er alle **value**-typer mens string og alle andre klasser (objekter) i C# er **referencetyper**.

Vi vil senere komme ind på denne forskel i forbindelse med klasser, men kort sagt er forskellen denne: valuetyper som fx int x=77 gemmes lokalt i det RAM område som hedder programmets **'stack',** mens reference typer som arrays, strings eller objekter gemmes ikke lokalt i det RAM område som kaldes **'heap'**.

Hvis 2 valuetyper som fx to heltal (x=9 og y=8) sammenlignes for at afgøre om de er 'ens' sammenligner C# om de har samme værdi eller value. (Value baseret **semantik**).

Hvis to objekter sammenlignes – sammenlignes det om de to objekter findes på samme RAM adresse! (Reference baseret semantik).

Value typer slettes **automatisk** når programmet (eller metoden) slutter (de er kun gemt rent midlertidigt i metodens stack), mens reference typer skal slettes gennem en 'rense maskine' i C# kaldet systemets **'Garbage Collector'** (systemet kendes også i Java). Garbage Collector'en arbejder altså kun i heap'en!

Når en metode modtager en valuetype som fx et tal modtager metoden en **kopi** af tallet. Men hvis metoden modtager en reference type modtager den **selve** objektet – nemlig en 'reference' eller 'pointer' til objektet.

Strenge eller strings:

En C# **string** er en variabel som kan indeholde fra et enkelt ord til indholdet af en lang tekstfil. Typen string er et andet ord for klassen System.String som indeholder bl.a. disse medlemmer og metoder – hvis vi antager en erklæring som string s og string str:

Metode	**Betydning**
s.Length	Antal tegn i s
s=s.Concat(str)	To strenge lagt sammen
s.CompareTo(str)	S sammenlignet med str
s.Insert(3,str)	Indsætte anden streng i s
s.ToUpper() s.ToLower()	Til store/små bogstaver
s.IndexOf(str) (eller: s.LastIndexOf())	Hvis str findes i s returneres pladsen som indextal ellers returneres –1 (=str findes ikke I s)
if(s.StartsWith(”NET”))	hvis de 3 første tegn er ‘NET’
if(s.EndsWith(”exe”))	lignende
string sub=s.SubString(0,3);	sub er de 3 første tegn i s
s=”Dette er ikke meget bedre”; string[] tabel=s.Split(’ ’);	giver en tabel med 5 pladser/ord, dvs deler s i delstrenge ved mellemrum
s.Trim()	fjern indledende og afsluttende mellemrum fx ved Console.ReadLine() – ellers kan bruges: s.TrimEnd() og s.TrimStart()
string str=s.Replace(”Write”,”Read”);	str er lig med s men Write er erstattet med Read
str=s.PadLeft(30);	str højrejusteres i et felt på 30 tegn, foran str er mellemrum

Strenge i C# er nemme at arbejde med idet udtryk som disse er gyldige:

```
string s=”Hej med ”;
string str=”dig!”;
s+=str;
if(s!=str){…
if(s==str){…
```

Der kan anvendes **escape** tegn i strenge fx:

```
string s=”I dag er det mandag,\nog i morgen er det tirsdag!”;
```

\n giver en ny linje midt i strengen.

Nogle af de escape tegn som kan anvendes er:

Escape tegn	**Betydning**
\’ \”	Indsætter citat tegn inden i strengen!
\\	Indsætter en backslash inden i strengen fx S=”c:\\dokumenter\\papirer”;
\n	Ny linje linje skift
\t \v	Vandret og vertikal tabulator
\a	Beep eller alarm!

For at gøre det nemmere findes også I C# et streng tegn @ der bruges fx sådan:

Console.WriteLine(@"File: 'c:\dokumenter\mintekst.txt'");

På denne måde kan man undgå at skrive de mange \ som ellers var nødvendige (såkaldt 'quoted string literal notation').

En streng kan også refereres med en **indexer**[] således:

```
string s="Der var en gang";
Console.WriteLine(s[0]+" "+s[1]);
```

Dette udskriver 'D e'. 'D' har index 0 dvs står på den første plads i strengen. Denne indexer er readonly!!

Man kan splitte en streng med **mange** tegn samtidigt efter denne model:

```
char[] delimiter = {';',', ','.'};
string str = Console.ReadLine().Split(delimiter);
```

Opgaver:

- Skriv et programmer som indlæser en lang streng fra bruger (med ReadLine()) og derefter tæller hvor mange mellemrum (' ') , kommaer og punktummer strengen indeholder.
- Skriv et program som erklærer en lang streng med kode tekst (som fx: "class X{public X(){}public static void Main(){}…") og **kontrollerer** at der er lige mange højre og venstre parenteser/krøllede parenteser! (En såkaldt **'parser'** – det er det C# kompileren bl.a. gør!).

Arrays eller tabeller:

Et array er en liste eller tabel som indeholder en række objekter ('elementer') af **samme** type. (En ArrayList – som vi skal se senere – kan indeholde forskellige typer i samme liste).

Der kan altså oprettes et array af heltal, tegn (char), strenge eller person objekter (hvis en 'person klasse' ellers findes):

0	1	2	3	4
'a'	'b'	'c'	'd'	'e'

I dette eksempel er oprettet en tabel med 5 pladser. På hver plads står et tegn eller en **char**. Et array har **ALTID** en type. Dvs. **kun** char kan stå i dette array!!

Den **første** plads (første index) i en tabel er plads 0 eller **index** 0. Dette kan forklares således:

Et array er gemt på en bestemt **adresse** i RAM – fx på adressen 1000. Hvis vi har et array af char fylder hver char i C# to bytes eller 16 bits (fordi C# bruger Unicode).

Grunden til at den første plads i tabellen hedder 0 (og ikke 1!) er at 0 betyder at vi ikke skal gå frem i forholdet til arrayets adresse 1000. Plads 5 eller index 5 eller [5] betyder derfor:

Start med adressen 1000. Gå 5 pladser frem: Dvs gå 5*2 bytes frem. Index 5 ligger altså på adressen 1010!!

Et array gemmes som en **blok** i RAM, hvor alle elementerne ligger i **forlængelse** af hinanden! Dette giver en meget **effektiv** og **hurtig** tilgang til tabellen! Tabeller eller arrays bruges **overalt** i programmer i C#!! Men de har deres **fordele** og **ulemper** – som vi skal se!

Et array oprettes fx således:

int[] talliste=new int[100];

Denne sætning opretter et array med plads til max 100 heltal. Pladserne i tabellen kan refereres således:

```
Console.WriteLine(talliste[0]);
talliste[0]=44;
talliste[44]=33*15;
talliste[5]=talliste[8]*talliste[9];
```

string[] ordliste={”Dette”,”er”,”et”,”C#”,”program”};

Denne sætning opretter en tabel af typen string med 5 pladser -- som **initialiseres** med det samme!

Der kan oprettes to-dimensionelle, tre-dimensionelle osv arrays:

i**nt[,] talmatrix=new int[3,3];**

opretter en 2 D tabel (en ’matrix’ eller ’matrice’) med 3*3 elementer.

Dette forstås sådan at talmatrix[0] er 1 række i tabellen, talmatrix[1] er 2. række osv.

F.eks. kan skrives:

```
talmatrix[0]={1,2,3};
talmatrix[1]={11,22,33};
talmatrix[2]={111,222,333};
```

int x=talmatrix[1,2]; //x sættes lig med værdien i 2.række 3.element: her lig med: 33.

En 2 D tabel ligner, hvad vi plejer at se i en **database** tabel:

ID	**Fornavn**	**Efternavn**	**Email**
1	Jens	Jensen	jj@mail.dk
2	Lise	Johansen	lise@mail.dk

To dimensionelle arrays (2 D) kan altså fungere som primitive database-tabeller. Ud fra ovenstående kunne man erklære en tabel med:

string[,] personer=new string[100,4];

Tabellen personer har nu plads til 100 poster (personer) som hver kan gemmes med 4 egenskaber.

Tre dimensionelle tabeller (3 D) kan bruges til **tids** akser. Fx kan tages udgangspunkt i denne 2 D tabel over måneder og gennemsnitstemperatur:

Måned	Gennemsnitstemperatur
januar	0
februar	5
marts	10
april	

En 3 D tabel kunne så bruge denne tabel - plus en ekstra dimension 'År'. Fx

string[,,]temperaturer=new string[50,12,12];

I denne tabel er der plads til 50 år.

NB i andre programmeringssprog (fx C++) erklæres en 2D tabel som new int[8][9] – altså med **to** parenteser – men der er kun **een** parentes i C#!!

En tabels **størrelse** kan **ikke** (aldrig) ændres, når den først er oprettet. Dette kan gøre at arrays kan virke noget **ufleksible**. (Helt modsat har en **ArrayList** – som vi skal se - slet ingen oprindelig størrelse eller maximum).

Anvendelsen af arrays giver meget **hurtig** adgang til de data som er gemt i tabellen fordi disse data er lagret i forlængelse af hinanden i RAM. Operativsystemet ved fx med det samme hvor henne i RAM ordliste[2] ligger – der er derfor meget hurtig **'access'** til et arrays elementer.

Et **eksempel** på et array af char (tegn) er følgende kode:

```
//fil: arrays.cs

//postcondition: outputter en tabel til skærmen (Unicode tegn tabellen 0..255):
//kør programmet sådan: arrays > arrays.txt
//så bliver output gemt i fil hvor char koderne kan ses!
//Jvf at 'a' er 97, 'b' er 98 osv
```

```
// 171 og 174 er symboler for copyright og registreret varemærke!

using System;

public class app{
  public static void Main(string[] args){

                    //der oprettes en tabel af tegn/bogstaver med 256 pladser
                    char[] tegnliste=new char[256];
```

```
                    for(int i=0;i<256;i++){

                    //i castes gennem char og giver derved det tegn som tallet repræsenterer:
                    tegnliste[i]=(char)i;
                    }

                    for(int i=0;i<256;i++){
                    Console.Write("Index: {0}={1} ", i, tegnliste[i]);
                    if(i%6==0)Console.WriteLine();
                    }
                    Console.Read();//teknisk af hensyn til Windows
  }

}
```

Sortering af arrays:

Modsat andre programmeringssprog er et array i C# i virkeligheden (skjult) en **klasse** - nemlig System.Array (i den DLL fil som hedder mscorlib.dll)!

Alle arrays i C# har derfor indbyggede metoder f.eks. sorter (Array.Sort() er en **static** metode i klassen Array dvs den kaldes direkte på klassenavnet) – som det ses af følgende kode fragment:

```
int[] tal={22,5,66,55,1111,356,2,19,999,345,13};
Array.Sort(tal);
for(int i=0;i<tal.Length;i++){
Console.WriteLine("Index: {0}: {1}",i,tal[i]);
}
```

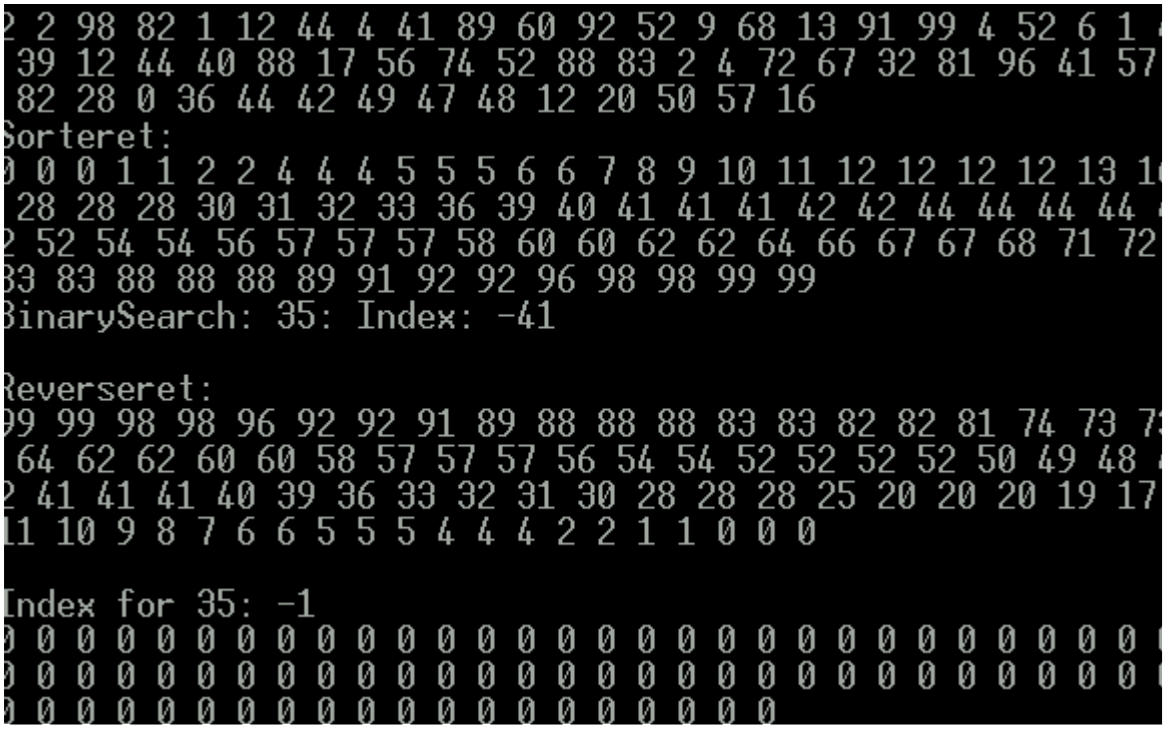
Som det ses med eksemplet med sortering af heltal - kommer man meget **nemt** om ved at sortere tabeller som indeholder grundlæggende **typer** som int, double, char eller streng! Sort() sorterer tabellen i **stigende** rækkefølge.

Array.Sort() kan **ikke** umiddelbart sortere objekter som huse eller personer – simpelt hen fordi det er 'uklart' hvem der 'kommer først'! Vi skal siden se hvordan man alligevel i C# kan få Array.Sort() til at sortere obejkter.

Alle arrays har en lignende metode der hedder Reverse(), som hvis den anvendes **efter** Sort() vil sortere tabellen i **faldende** rækkefølge.

Læg også mærke til at alle arrays kender deres egen størrelse: ex tabel.Length (i modsætning til fx C og C++ hvor dette giver utallige problemer).

System.Array har en lang række af andre metoder. På adressen http://csharpkursus.subnet.dk ligger et eksempel **arrays**.cs som viser nogle af metoderne. Programmet kan give følgende resultat:



Interessante metoder i System.Array er Array.**BinarySearch**(tabel,værdi) som søger efter om en værdi findes i tabellen. Binær søgning kræver at tabellen er **sorteret** men er langt den **hurtigste** søge procedure som findes. Dette er selvfølgelig mere tydeligt når programmet skal søge blandt 987.234 poster!

BinarySearch (se eksemplet) returnerer den **plads** tallet findes på eller fx –41 hvilket betyder at **hvis** tallet havde været med **skulle** det have stået før plads 41!

ArrayIndexOf(tabel,værdi) returnerer den plads hvor værdien står på eller –1 hvis værdien ikke bliver fundet.

Som det ses er 35 ikke med i de 100 tilfældige tal som programmet opretter.
Array.**Clear**(tabel,start,slut) nul stiller tabellen – se eksemplet til sidst!

Arrays **kan** også oprettes som egentlige objekter efter denne model:

```
Array array = Array.CreateInstance(typeof(string), 4);
array.SetValue("Jackson", 0);
array.SetValue("Lisa", 1);
array.SetValue("Zachary", 2);
```

Her oprettes en tabel med plads til 4 strenge med en static metode Array.CreateInstance().
Værdierne hentes med metoden **GetValue**(int).

Der er dog ikke normalt nogen grund til at bruge denne mere besværlige fremgangsmåde!

Opgaver:

1. Skriv et program der opretter et array af 10 bytes, giver dem en værdi og udskriver dem
2. Skriv et program med en 2 dimensionel tabel af strenge hvor hver række er et dansk ord og ordets oversættelse til engelsk – altså en ordbog.
3. Skriv et program hvor brugeren skal indtaste brugernavn og adgangskode hvorefter programmet kontroller om data er korrekte ved at gennemløbe en 2 D tabel af strenge!

Operatorer i C#:

En operator er et specialtegn som på en eller anden måde manipulerer med en eller flere værdier.

Tildelings operator:

Eksempel:
x=24;
betyder at værdien 24 lægges over i x. Retningen går altid **fra højre** mod venstre.
X=Y=1234;
Betyder at 1234 først lægges over i Y og at værdien af Y (1234) derefter lægges over i X.

Matematiske operatorer:

Eksempler:

```
X=12+23;//addition
X=X-3;//subtraktion
X=X*4;//multiplikation
X=X/2;//division
X=X%3;//modulus, dvs hvad er resten når X deles med 3
```

Eksempler på 'short cuts' dvs forkortede versioner:

```
X+=2;//kortere form der er lig: X=X+2;
X-=3;//X=X-3;
X*=5;//X=X*5;
X/=5;//X=X/5;
X%=2;//X=X%2;
X++;//X=X+1;
X--;//X=X-1;
++X;
--X;
```

Relationelle operatorer og if-sætninger:

De relationelle operatorer er:

Operator	Betydning	Eksempel
>	Større end	5>3
<	Mindre end	3<5
==	Lig med (NB to lighedstegn i C#)	3==3
!=	Ikke lig med	4!=3
>=	Er større end eller lig med	5>=2
<=	Mindre end eller lig med	1<=9

If sætninger bruges til at styre programmets **flow** – typisk til at vælge en sti til den ene eller anden side. If er en kontrol struktur af typen **selektion**. I struktureret kode bruges typisk selektioner og **iterationer** (loops, løkker, der kører et antal gange, jvf senere).
En if sætning består af en **betingelse** som ALTID er **enten** sand **eller** falsk.

Et eksempel:

```
//fil: relation.cs

//eksempel på if sætninger og relationelle operatorer
//postcondition: udskriver data om 3 tal:

using System;

public class Relation{
  public static void Main(string[] args){
                    int x=6,y=9,z=13;
                    if(x<=y){
                                        Console.WriteLine("{0} er mindre end eller lig med {1}.",x,y);

                     if(x<=z){
                                        Console.WriteLine("{0} er mindre end eller lig med {1} og {2}.",x,y,z);
                                        Console.WriteLine("{0} er det mindste tal.",x);

                     }
                     else{
                     Console.WriteLine("{0} er mindre end eller lig med {1}, men ikke mindre end eller lig med {2}",x,y,z);
                     }
                    }
                     else{
                     Console.WriteLine("{0} er ikke mindre end eller lig med {1}.",x,y);
                     }
                    Console.Read();//teknisk af hensyn til Windows
  }
```

```
}
```

Kommentar:
{0} bruges som format-tegn eller 'pladsholder'. Sætningen:

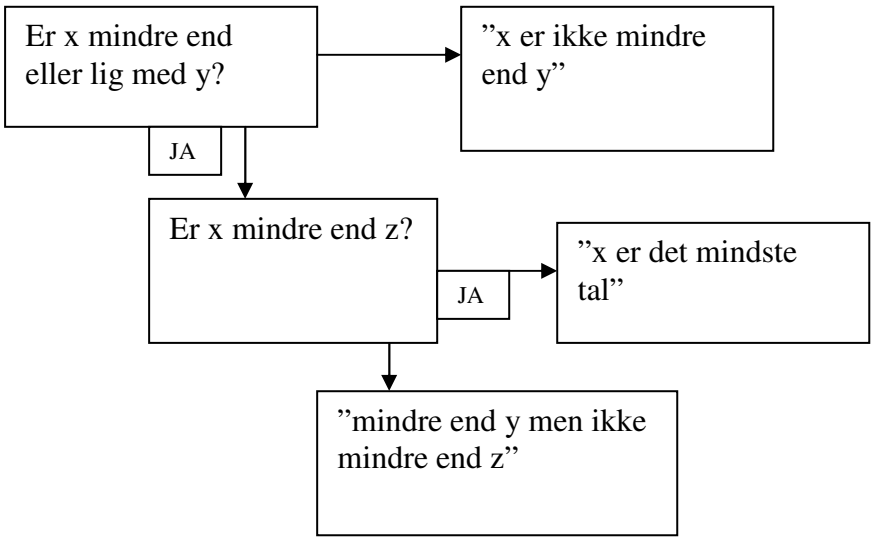
```
Console.WriteLine("{0} er ikke mindre end eller lig med {1}.",x, y);
```

Betyder at værdien af x skal indsættes i stedet for {0} og y i stedet for {1}.
(Metoden svarer til dels til printf() funktionen i C).

Programmet indeholder en **overordnet** if sætning (er x mindre end/lig med y) og en tilhørende else sætning – nemlig den **sidste** else.
Altså hvis x<=y gør det følgende ELLERS gå ned til **sidste** else.
Hvis x<=y gås videre til en ny **indlejret** if sætning der spørger om x (også) er lig med eller mindre end z. Også denne if sætning har en else.

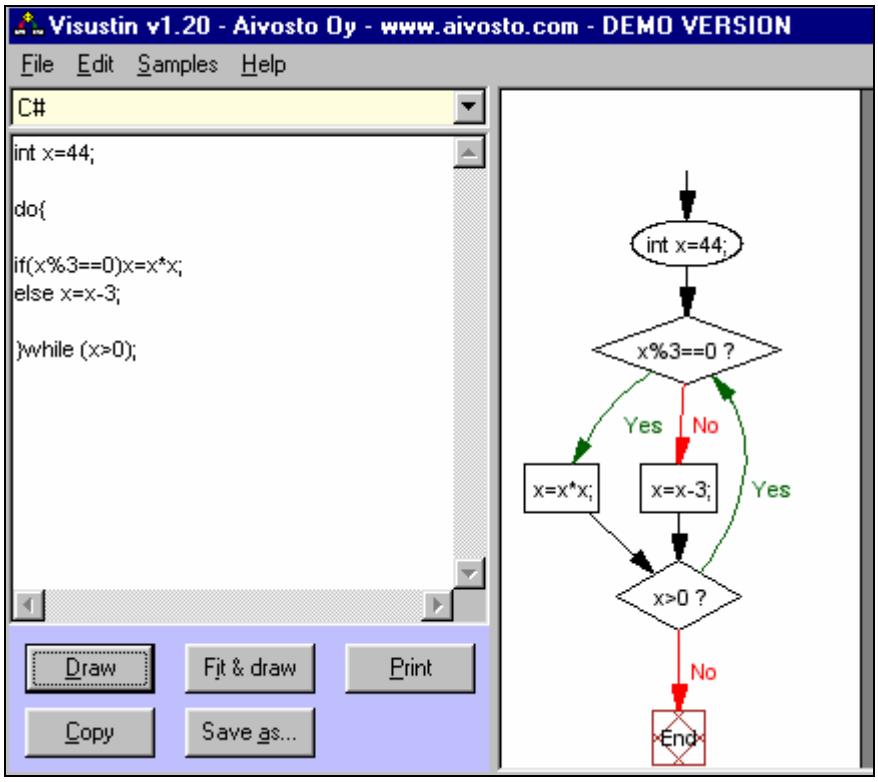
Det er vigtigt at holde hele **strukturen** i if/else meget klart ellers kan man let komme til at begå 'logiske fejl' – dvs programmet kommer til at gøre noget som man ikke ønsker. (Husk at et program med masser af logiske fejl sagtens kan kompileres – compileren kan **aldrig** finde en logisk fejl!!).

I ovenstående er anvendt **fed** om den **ydre** if sætning og **kursiv** om den **indre** if sætning – for at gøre strukturen tydeligere. I selve cs-koden skal disse linjer IKKE skrives med kursiv eller fed!!



Kontrol flow diagram:

Det er en god ide manuelt at tegne et kontrol flow diagram over programmets selektionen, if sætninger osv. På Internettet kan også hentes sharewareprogrammer som kan tegne automatiske diagrammer ud fra C# kode – jvf senere. Her er et lille eksempel med if og else sætninger:



Logiske operatorer:

I en if sætning kan betingelsen i parentesen består af et udtryk. Hertil bruges 2 logiske operatorer nemlig AND (&&) og OR (||).

For eksempel:

if(sex==”mand” && alder >=50){…så motioner!
if(land==”DK”||land==”No”||land==”Sv”){ …så er NN skandinav

Hvis en betingelse består af && skal BEGGE delbetingelser være sande for at det hele er sandt.

Hvis en betingelse består af || skal blot en af delbetingelserne være sande for at det hele er sandt.

Et mere kompliceret eksempel ville fx være:

if(x%3==0&&(x%4==0||x%5==0)){…

Betingelsen kræver for at være opfyldt følgende:

Tallet x skal både kunne deles med 3 uden rest og kunne deles med 4 uden rest eller med 5 uden rest.
Hvis x er lig 15: er betingelsen sand.
Hvis x er lig 20: er betingelsen falsk.

If sætninger og relationelle operatorer er et sted hvor der begås mange logiske fejl. Det er derfor en god ide at tegne kontrol flowet i et kontrol flow diagram eller data flow diagram som vist ovenfor.

Binære Tal: Operatorerne &, | og ^:

Det er meget vigtigt at **skelne** mellem operatoren && og &!! Operatorerne | og & bruges fx om 2 binære tal som kan AND'es eller OR'es (regning med bits!).

Fx er det binære tal: 0010 lig med vores decimal tal 2 og 0011 lig med decimal tal 3.

Udtrykket 0010&0011 (tal1 AND tal2) kan vises således:

Bit 3 (antal ottere)	Bit 2 (antal firere)	Bit 1 (antal toere)	Bit 0 (enere)
0	0	1	0
0	0	1	1
0	**0**	**1**	**0**

En **AND** af de to tal giver: **0010** (altså decimaltal: 2) fordi en AND kræver et et-tal samme sted (i samme bit) i begge tal.

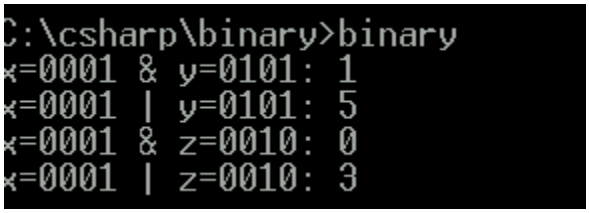
En **OR** kræver derimod at blot en af pladserne har et et-tal – f.eks. sådan:
1010|0101 giver 1111 (eller decimal tal: 15).

(1010^0101 (**XOR** eller **Eksklusiv OR**) betyder at resultatet giver et et-tal hvis den ene og **kun** den ene er et et-tal! I dette tilfælde giver det resultatet 0000).

Følgende lille program illustrerer regning med operatorerne | (OR) og & (AND) og bits:

```
// DEMO af binære operatorer /binære tal
//AND og OR mellem 2 binære tal:

using System;

class app
{
        public static void Main(string[] args)
        {
                byte x=1,y=5,z=2;
                //AND:
                Console.WriteLine("x=0001 & y=0101: {0}",x&y);
                //OR
```

```
                Console.WriteLine("x=0001 | y=0101: {0}",x|y);
                //AND:
                Console.WriteLine("x=0001 & z=0010: {0}",x&z);
                //OR:
                Console.WriteLine("x=0001 | z=0010: {0}",x|z);

                Console.Read();
            }
}
```



På http://csharpkursus.subnet.dk ligger der også et eksempel på et program som arbejder med at oversætte fra binære tal til decimaltal og omvendt! I System.Collections ligger en basis klasse BitArray som kan anvendes til at arbejde med bits og bytes!

Kontrol strukturer:

Faste kontrol strukturer bruges til at sikre en struktureret kode og undgå uoverskuelig spaghetti kode.
Ud over if-sætninger som lige er nævnt kan anvendes en række andre kontrolstrukturer i C#:

for:

Mange programmer løber igennem en løkke (loop) et antal gange. Sådanne kontrol strukturer kaldes **'iterationer'**. f**or-sætninger** er et eksempel her på. Modsat fx foreach sætninger løber for sætninger gennem løkken et **bestemt** antal gange fastsat på forhånd:

Eksempel:

```
for(int i=0;i<10;i++){
Console.WriteLine("Dette er C Sharp – {0}",i);
}
```

Denne program stump udskriver strengen "Dette er…" 10 gange. NB variablen i er kun synlig inden i løkken - ikke uden for (jvf tidligere afsnit om en variabels **scope** eller levetid).

Initializer:
int i=0; betyder at tælleren i startsættes til værdien 0.

Condition:
i<10; betyder at så længe i er mindre end 10 gentages blokken (mellem { og }). Dette led svarer fuldstændigt til en if-sætning.

Incrementer:
i++ betyder at hver gang løkken er nået **til ende** lægges 1 til værdien af i (og derefter chekkes om betingelsen/Condition stadig er sand).

Ved første gennemløb er i lig med 0, ved sidste er i lig med 9.
For strukturen anvendes til mange forskellige problemløsninger (algoritmer) fx sortering og søgning.
Ikke alle 3 led behøver altid at være til stede i en for-sætning og man kan sagtens konstruere mere indviklede for sætninger som fx:

```
for(int i=0,j=100;i<100,j>0;i++,j--){
```

Denne sætning inkrementerer (øger) i og dekrementerer (nedsætter) j for hvert gennemløb. Flere led mellem hvert semikolon skal adskilles med komma.

Et andet eksempel:

```
for(int i=0;i<101;i=i+2){
```

Dette eksempel henter kun de lige tal fra 0 til og med 100.

Senere i afsnittet om **switch** gives et praktisk eksempel på en for sætning.

Opgaver:

1. Opret en to dimensionel tabel og udskriv den med 2 for-sætninger inden i hinanden

2. Skriv et program hvor brugeren skal indtaste 10 bogstaver hvorefter programmet udskriver det 'mindste' bogstav (med det mindste nummer i ASCII/Unicodetegn koden)
3. Skriv et program som udskriver to beskeder afhængig af brugerens indtastning – fx spørgsmålet "Hvilket operativ system bruger du lige nu?".
4. Opret en tabel af int på 10 pladser og udskriv hvert tal med tallet selv, tallet i 2. potens og tallet i 3.potens
5. Udskriv en liste over tallene fra 1 til 64 og det tilsvarende hexadecimale tal. De hexadecimale tal går fra 0 til F idet 10=hex A, 11=hex B osv. For at få vist et tal som hex tal skal det formatteres som hex tal efter følgende formel:

   Console.WriteLine("{0} = {0:X}",tal);

   Hvilket betyder at tal vises som decimaltal og derefter som hexadecimalt tal.

Eksempel på løkker: Binære tal og decimaltal:

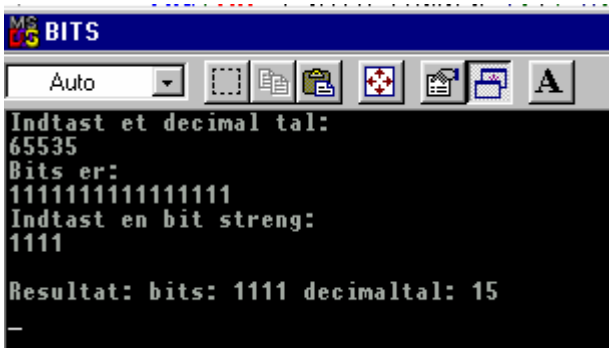
At analysere en streng eller et tal er eksempler på anvendelse af løkker. Eksemplet her viser hvordan man kan **konvertere** fra binære til decimal tal. Vi har tidligere været inde på binære tal. Prøv at løse følgende opgaver ved at bruge kode eksemplerne nedenfor. Her anvender vi en anden løkke do..while som vi snart skal se nærmere på:

Opgaver:

1. Skriv et program hvor brugeren indtaster et decimaltal (fx 211) og programmet udskriver tallet som en bit streng. Brug kode eksemplet nedenfor!
2. Skriv et program hvor brugeren indtaster en bit streng (fx '010101') og programmet udskriver tallet som decimaltal. Brug kode eksemplet nedenfor!
3. Forklar hvordan man regner frem og tilbage mellem binære tal og decimaltal!

```
string bits=null;

do{
bits+=(char)tal%2;//find rest, del med 2, find rest osv:
tal=tal/2;
}while(tal>=0.5);
```

```
int resultat=0;
//bit 0 har værdien 1*1 eller 1*0:
int faktor=1;

//start med bit 0 yderst til hoejre:
//NB ingen exception handling - hvad hvis bit er sat til 3!?
for(int i=bits.Length-1;i>=0;i--){
        //vi ganger med den binære faktor (1, 2, 4 osv):

        if(bits[i]=='1')resultat+=faktor*1;

        //Vi rykker en plads til venstre derfor:
        faktor=faktor*2;
}
```



C# format koder:

I C# kan anvendes disse og en del andre formatterings koder i stil med eksemplet ovenfor:

Formatterings tegn	Eksempel
C c	Formatteres som valuta (’Kr. 33.222,00’)
D d	Formatteres som decimaltal - d9 betyder altid 9 cifre
E e	Eksponentiel formattering
F f	Kommatal – F3: med 3 decimaler
X x	Formatteres som hexadecimale tal

Hvis vi bruger denne program stump:

```
double x=123.456;
int x1=123456;
double x2=0.05;
int cpr=0101646767;
int telefon=44449090;

Console.WriteLine("C: {0:C}",x);
Console.WriteLine("C0: {0:C0}",x);
Console.WriteLine("D: {0:D}",x1);
Console.WriteLine("E: {0:E}",x);
Console.WriteLine("F: {0:F}",x);
Console.WriteLine("F1: {0:F1}",x);
//N er numerisk uden decimaler:
Console.WriteLine("N: {0:N}",x1);
Console.WriteLine("X: {0:X}",x1);
//G er generelt format:
Console.WriteLine("G: {0:G}",x);
//R Roundtrip bevarer tallets præcision:
Console.WriteLine("R: {0:R}",x);
//Vis som procent:
Console.WriteLine("%: {0:0%}",x2);

//Eksempel på at man selv kan skrive et format:
```

```
Console.WriteLine("TELEFON: {0:00' ’00' '00' '00}",telefon);
Console.WriteLine("CPR: {0:000000'-'0000}",cpr);
```

- kan vi få følgende udskrift:

Opgaver:

1. Skriv et program hvor brugeren indtaster en liste af varer og priser, hvorefter programmet udskriver en faktura med varenavn og kronebeløb og sum af alle varer.
2. Skriv et program hvor brugeren kan indtaste et **bogstav** og hvor programmet så udskriver alle de **fonts** eller skrifttyper som er installeret på maskinen og som starter med bogstavet! Brug nedenstående kode som udgangspunkt:

```
//fil: fonts.cs

//Eksempel: arbejde med de installerede fonts/skrifttyper
//Brugeren kan søge om en font er installeret:

using System;
using System.Drawing.Text;

public class VisFonts
{


        public static void Main(string[] args)
        {

                //Find alle fonts på maskinen:
                InstalledFontCollection fonts=new InstalledFontCollection();
                for(int i=0;i<fonts.Families.Length;i++){
                                        Console.WriteLine(fonts.Families[i].Name);

                }

        }
}
```

Du skal bruge en sammenlignende if sætning af denne type:

```
if(fonts.Families[i].Name.StartsWith(...<tekst>...)
```

foreach:

er et alternativ til almindelige for-sætninger. **foreach** er en **iterations** kontrol struktur (som ikke findes i C(++) eller Java, men som C# har lånt fra Visual Basic).

Dens **fordel** er at man kan gennemløbe en liste uden at vide hvor mange elementer listen indeholder.

En **ulempe** ved foreach er at den er **readonly** – man kan ikke ændre værdier mens man løber igennem (som man kan i en **for** sætning).

Syntaksen er:

foreach (object o **in** collection){}, hvor **collection** er en eller anden **liste** og o er denne listes **type**.

Et **eksempel**, som udskriver en liste over drev på computeren:

```
//fil: foreach.cs

//postcondition: outputter en liste over computerens drev til skærmen:

using System;

public class app{
  public static void Main(string[] args){

                        //Environment er en klasse som beskriver denne maskines styresystem:
                        //NB: tabellen drev angives ikke med nogen størrelse
                        //tabellen initialiseres med det samme:

                        string[] drev=Environment.GetLogicalDrives();

                        //foreach gennemløber et måske ukendt antal drev
                        //modsat en for sætning som skal kende antallet af pladser:

                        foreach(string s in drev){
                          Console.WriteLine("Drev: {0}", s);
                        }
                        Console.Read();//teknisk af hensyn til Windows
  }

}
```

Det følgende kode eksempel illustrerer at foreach er praktisk netop når antallet af objekter som skal gennemløbes er ukendt.

Programmet anvender klasser fra System.IO.dll og udskriver en liste over filer i en mappe:

```
//fil: findfiler.cs
```

```
//eks på kommando linje parametre: findfiler c:\csharp exe
//eks: findfiler . * ville vise ALLE filer i nuværende mappe

//postcondition: udskriver liste over filer i en mappe:

using System;
using System.IO;//i dette namespace fides alle fil klasser

public class app{
  public static void Main(string[] args){

                    //dir sættes til argument 0 - et . betyder nuværende mappe:
                    DirectoryInfo dir=new DirectoryInfo(args[0]);

                    //chek om brugerens indtast overhovedet er et gyldigt bibliotek:
                    if(dir.Exists){

                    //filer er en tabel af FileInfo objekter dvs af filer
                    //GetFiles() kaldes med en fil type som parameter:

                    FileInfo[] filer=dir.GetFiles("*."+args[1]);

                    //her kender vi jo netop IKKE antallet af filer:

                    foreach(FileInfo f in filer){
                      Console.WriteLine("Navn: {0}",f.Name);
                      Console.WriteLine("Antal bytes: {0}",f.Length);
                      Console.WriteLine("Oprettet: {0}",f.CreationTime);
                      Console.WriteLine("Attributter: {0}",f.Attributes.ToString());
                      Console.WriteLine("Modificeret: {0}",f.LastWriteTime);
                      Console.WriteLine();
                    }
                    }else Console.WriteLine("{0} mappen findes IKKE!",args[0]);



                    Console.Read();//teknisk af hensyn til Windows
  }

}
```

En streng-liste over filer i det aktuelle bibliotek (applikationens mappe) kan også fås ved:

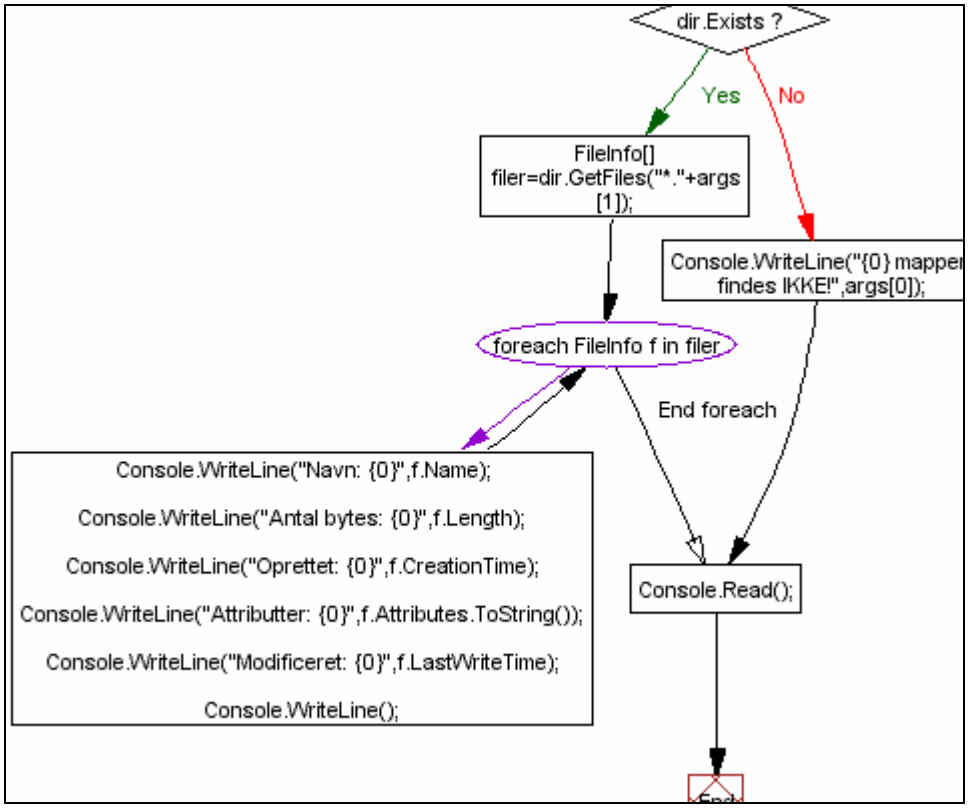
```
string[] files = Directory.GetFiles(Directory.GetCurrentDirectory());
```

Undermapper kan oprettes og omdøbes således:

```
DirectoryInfo di = new DirectoryInfo(Directory.GetCurrentDirectory());
di.CreateSubdirectory("Bitmaps");
DirectoryInfo jpgdir = di.CreateSubdirectory("jpg");
jpgdir.MoveTo("Billeder");
```

Her oprettes en under og underunder mappe og den sidste omdøbes til 'Billeder'.

Et kontrol flow diagram over ovennævnte kode ser nogenlunde sådan ud:



Klassen er forsynet med en if..else konstruktion der sikrer at det indtastede bibliotek findes og evt udskriver en fejl meddelelse. Dette er et lille eksempel på et **meget stort** problem:

Et godt råd er: **Forudse** hvad der kan gå galt og skriv koden så den tager højde for mulige 'fejl'!

Målet er at koden skal være så **'robust'** som muligt! Vi skal senere se på dette problem i forbindelse med **exceptions** i C#.

```
C:\csharp>findfiler c:\windows bin
Navn: SUBACK.BIN
Antal bytes: 229680
Oprettet: 01-01-1601 01:00:00
Attributter: Archive
Modificeret: 15-05-1998 21:01:00

Navn: W98SETUP.BIN
Antal bytes: 169003
Oprettet: 01-01-1601 01:00:00
Attributter: Archive
Modificeret: 15-05-1998 21:01:00
```

Eksempel: Programmet køres med parametrene **c:\windows** og **bin** (filerne oprettet i 1601!).

Et andet eksempel på foreach er databehandling af **args** – kommando linje parametre til programmet. Disse er som nævnt altid strings men følgende kode eksempel viser hvordan de meget let kan konverteres til fx int (heltal):

```
//fil: args_int.cs
//demo af foreach og konvertering af string til int

using System;

class app
{
        public static void Main(string[] args)
        {
                foreach(string s in args){
                        //metoden int.Parse("....") returnerer et heltal:
                        //tilsvarende findes: double.Parse(), float.parse() osv:

                        Console.WriteLine("Tal: {0}, Kvadratet:
                        {1}",int.Parse(s),(int.Parse(s)*int.Parse(s)));

                }
                Console.Read();
        }
}
```

Programmet udskriver de indtastede strings som heltal og kvadratet af dette tal. En streng parses altså på denne måde:

```
string s=”1234”;
int x=int.Parse(s);
double d=double.Parse(s);
```

Osv. **Alle** de indbyggede typer kan konverteres på denne måde også fx **bool**.

Tilsvarende har klassen System.**Convert** en række konverteringsmetoder f.eks:

```
int n = Convert.ToInt32(Console.ReadLine());
double d = Convert.ToDouble(Console.ReadLine());
```

Resultatet af int.Parse() og Convert metoderne er det samme.

Opgaver:

1. Skriv et program hvor brugeren bliver bedt om at indtaste et tal som derefter udskrives som tallet selv, tallet i 2.potens og kvadratroden af tallet. Husk at det skal indtastes i en ReadLine() er en streng ikke et tal! Til kvadratroden skal du bruge: **System.Math.Sqrt(tal). NB Sqrt()** returnerer en double, ikke en int!

2. Skriv et program hvor bruger indtaster de to korte sider i en ret vinklet trekant og programmet udskriver den lange side (hypotenusen)! (Pythagoras sætning).
3. Skriv et program hvor bruger indtaster en radius og programmet udskriver cirklens areal. Du kan her bruge areal = radius *2 *Math.PI. **PI** er defineret som en konstant i System.Math.

switch:

Hvad der kan opnås gennem en switch struktur kan **også** opnås med if..else, **men** switch strukturen er nemmere at forstå og derfor mere **struktureret**.

Et **eksempel**, der simulerer kast med terning og optælling af værdierne:

```
//fil: switch.cs

//eksempel på switch strukturen og random tal
//simulerer kast med terning
//postcondition: udskriver data om 100000 'tilfældige' tal:

using System;

public class app{

  public static void Main(string[] args){

                    //et_kast rummer værdien 1..6 af hvert kast med 'terning':
                    int et_kast=0;

                    //disse variable er 'tællere' dvs hvor mange ettere? osv:
                    int et=0,to=0,tre=0,fire=0,fem=0,seks=0,fejl=0;

                    //der kastes 100.000 gange med en 'terning':
                    for(int i=0;i<100000;i++){

                      Random rnd=new Random();
                      //tallet skal være mindst 1 og altid mindre end 7:
                      et_kast=(int)rnd.Next(1,7);

                    switch(et_kast){
                     case 1:et++;break;//hver gang kastet giver 1 lægges 1 til variablen et
                     case 2:to++;break;
                     case 3:tre++;break;
                     case 4:fire++;break;
                     case 5:fem++;break;
                     case 6:seks++;break;
                     default:fejl++;break;//forhåbentligt helt overflødig!!
                    }

                    }
                    //løkken (for sætningen) slutter her.

                    Console.WriteLine("Antal af 1: {0}",et);
                    Console.WriteLine("Antal af 2: {0}",to);
                    Console.WriteLine("Antal af 3: {0}",tre);
```

```
                Console.WriteLine("Antal af 4: {0}",fire);
                Console.WriteLine("Antal af 5: {0}",fem);
                Console.WriteLine("Antal af 6: {0}",seks);
                Console.WriteLine("\nAntal af fejl-tal: {0}",fejl);



                Console.Read();//teknisk af hensyn til Windows
 }

}
```

Kommentarer:

En af C#'s kerne klasser System.Random instantieres med new Random(), dvs der oprettes et Random-objekt ved navn 'rnd', dvs et 'tilfældigt' tal.. (Se mere herom senere i forbindelse med objektorienteret programmering).
Eftersom metoden rnd.Next() returnerer et kommatal (en double) laves en 'cast' gennem int. Cast betyder her at decimalerne bortkastes. Parametrene til rnd.Next betyder at tallet er under 7 og mindst 1 – hvad vi ønsker hvis vi skal simulere en terning.

```
Random rnd=new Random();
//tallet skal være mindst 1 og altid mindre end 7:
et_kast=(int)rnd.Next(1,7);
```

I forsætningen starter i med at være 0 og slutter med at være 99.999. I det øjeblik at i er 100.000 er betingelse i forsætningen ikke længere sand og løkken stopper.

Switch strukturen tester på værdien af et_kast og opdaterer de 6 variable hver gang.
Hvis dette skulle have været opnået med if..else sætninger var koden blevet meget mere kompliceret og sværere at overskue.
Der kan i C# også switches på en streng! (Modsat fx C++).

Derfor er switch ofte et godt valg hvor der skal vælges 'sti' i programmets kontrol flow.

Hver case afsluttes med en **break**. **break** betyder at kontrol flowet springer de følgende cases over og går op til næste kast.

I C# **skal** hver case afsluttes med en **break**!! Ellers fås fejl ved kompileringen! (Modsat fx C++).

```
C:\csharp>switch
Antal af 1: 18470
Antal af 2: 15647
Antal af 3: 14765
Antal af 4: 15716
Antal af 5: 17724
Antal af 6: 17678

Antal af fejl-tal: 0
```

while:

er en iterations struktur lige som **for**. Så længe en betingelse er sand kører while-løkken.

Et eksempel:

```
//fil: while.cs
//brugeren skal gætte en kode på 3 tegn
//postcondition: outputter feedback til brugeren

using System;

public class app{
  public static void Main(string[] args){
                    string kode="cba";

                    //variabel til at opsamle brugerens gæt:
                    string bruger_kode="";

                    //CompareTo() returnerer 0 hvis de to strenge er identiske:
                    //løkken kører indtil dette er tilfældet:

                    while(bruger_kode.CompareTo(kode)!=0){

                    Console.WriteLine("Min kode bestaar af a, b, c i en kombination: Hvad tror du?");
                    bruger_kode=Console.ReadLine();
                    }
                    Console.WriteLine("OK, min kode er: {0}",kode);

                    Console.Read();//teknisk af hensyn til Windows
  }

}
```

Kommentar:

Brugeren skal gætte en streng som 'abc' eller 'bca' eller 'cba'. While kører indtil metoden CompareTo() returnerer 0 – dvs indtil at koden er gættet.

Metoden CompareTo() returner –1 hvis den første streng er større, +1 hvis den sidste streng er større og 0 hvis de er ens. Metoden kan bruges til at sortere strenge. En 'mindre' streng er en streng som alfabetisk kommer før en 'større' streng. Eksempel: 'lise' er mindre end 'viktor'.

do..while:

fungerer lige som while, blot køres løkken altid **mindst** en gang:

```
                    do{
                    Console.WriteLine("Min kode bestaar af a, b, c i en kombination: Hvad tror du?");
                    bruger_kode=Console.ReadLine();
                    }
                    while(bruger_kode.CompareTo(kode)!=0);
```

I modsætning hertil kan en while sætning eventuelt ikke engang køre en gang – hvis betingelsen ikke er opfyldt fra starten.
Valget mellem de to kontrol strukturer afhænger altså af om man ønsker kørt løkken mindst en gang.

goto:

Der er tidligere givet et eksempel på kontrol strukturen goto. Det er ikke anbefalelsesværdigt at anvende **goto** hvis det kan undgås – og det kan det.
Alle iterationer/loops kan dannes med if og goto (sådan gør man også som regel i 'assembler programmering' som er programmering på et lavere niveau end høj niveau programmering som fx C#):

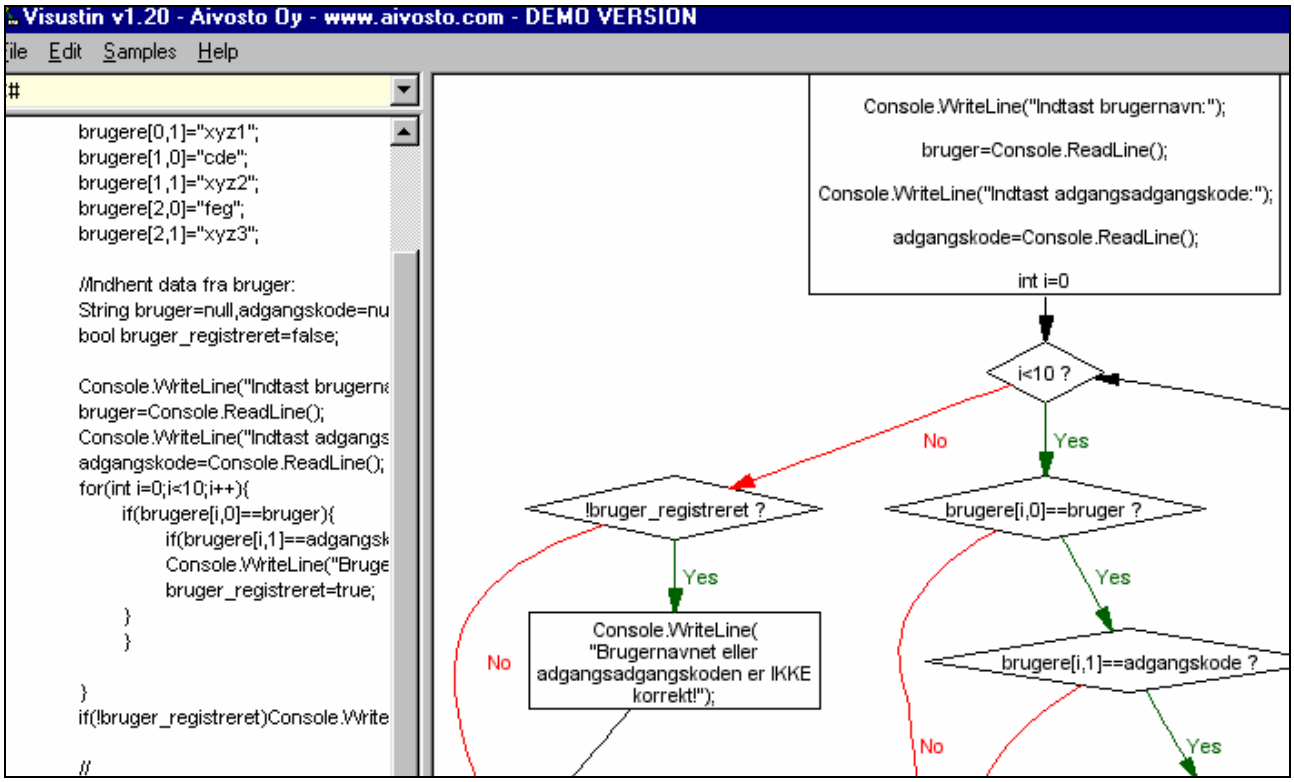
```
int x=0;
Igen:
Console.WriteLine("Hello!");
x++;
if(x<10) goto Igen;
```

Denne program stump viser hvordan en do..while kan konstrueres med if og goto. "Hello!" udskrives 10 gange. Også de andre loops kan dannes på denne måde. Men det er **ikke** anbefalelsesværdigt!

Kontrol Flow Diagrammer:

Det er en god ide – gerne med papir og blyant – at regne det kontrol flow der findes i et C# program På den måde kan man se svagheder og muligheder i programmet!

Der findes også en del shareware programmer på Internettet som kan tegne forskellige slags kontrol flow diagrammer. Her er vist et eksempel fra http://www.aivosto.com hvorfra kan downloades programmet Visustin. I dette program kan man indtaste (indkopiere) en kodetekst og automatisk få tegnet et diagram (!):



Koden indeholder et lille program som tester om et brugernavn og en adgangskode er korrekte!

Metoder eller funktioner:

Et vigtigt element i struktureret programmering er at opdele koden i metoder som hver især udfører en vel defineret del af det samlede arbejde (som i fællesskab bidrager til programmets algoritme). På denne måde **'modulariseres'** programmet – en metode er et afgrænset **'modul'** – i modsætning til 'spaghetti koden'.

Metoder vil blive grundigt gennemgået i forbindelse med objekt orienteret programmering senere. Et eksempel på kald af metoder i et almindeligt Main() program er følgende, hvor det kan ses at Main() 'uddelegerer' arbejdet til metoder uden for Main(). Det specielle ved dette eksempel er at alle metoder er **static,** hvilket normalt ikke er tilfældet med metoder, men her er krævet fordi de skal kaldes direkte af Main().

Programmet simulerer et program som indhenter en række data om en person. I Main() kaldes en metode ved simpelt hen at skrive metodens navn med parenteser og evt parametre (ingen af metoderne her har parametre, da de er get-metoder):

```
//fil: metoder.cs

//illustrerer metoder og struktureret programmering
//simulerer indhentning af data om person

//postcondition: outputter de indtastede data til skærmen

using System;
```

```
public class app{

 public static void Main(string[] args){

   get_fornavn();
   get_efternavn();
   get_email();
   get_telefon();
   get_mobil();
   Console.WriteLine("Efternavn: {0}, Fornavn: {1}, Email: {2}.",efternavn,fornavn,email);

   Console.Read();//teknisk af hensyn til Windows

 }

 //NB kun disse 3 metoder er implementeret:
 //alle metoder som kaldes af Main() skal være static:

 public static void get_fornavn(){
   Console.WriteLine("Indtast fornavn:");
   fornavn=Console.ReadLine();
 }
 public static void get_efternavn(){
   Console.WriteLine("Indtast efternavn:");
   efternavn=Console.ReadLine();
 }
 public static void get_email(){
   Console.WriteLine("Indtast email adresse:");
   email=Console.ReadLine();
 }

 //disse 2 metoder er ikke implementeret men tages med for at vise strukturen:
 //det er en god ide at skrive sådanne 'tomme metoder' i første omgang:

 public static void get_telefon(){}
 public static void get_mobil(){}

 //variable skal også være static i dette tilfælde
 //de skal kunne bruges både af Main() og af metoderne:
 public static string fornavn,efternavn,email;


}
```

En **metode** er altid bygger op efter følgende model hvis vi kun ser på metodens **signatur**:

Metodens **access** dvs Kan den kaldes udefra?	Evt. static?	Hvad metoden **returnerer**	Metodens **navn**	**Parametre**
public eller **private** eller **protected** eller **internal**	en **static** metode kan kaldes med klassenavnet punktum metodenavnet fx: Math.Sqrt()	int, string eller objekt... eller **void** (metoden returnerer ingenting)	(En metode kan kaldes hvad som helst blot kan keywords ej bruges)	EKS: string[] args eller int eller string Der kun være 0, 1 eller mange

				parametre

Metoder er oftest **public**. En metode som er **private** vil oftest fungere som en hjælpe metode til en public metode. En **protected** metode kan kun bruges af sub klasser. En **internal** metode kan kun bruges i denne assembly. (jvf senere om OOP).

Metoder kan også være **virtual** eller **override** og meget andet (jvf senere om OOP).

Typisk for metoder er altså at de **kaldes,** at de kaldes med evt. **parametre**, at de udfører et stykke **arbejde** og derefter **returnerer** til 'hovedprogrammet' eller 'hoved tråden**'**. Programmets flow springer til metoden og vender derefter **tilbage** til linjen efter hvor metoden blev kaldt (**kontrol flow**).

Det er en gammel anerkendt metode i struktureret programmering at designe sit program ved at starte med en Main() og derefter angive hvilke metoder Main() skal **kalde** for at 'løse problemet'. Først derefter skrives selve koden til disse funktioner/metoder.
På denne måde bevares overblikket og det overordnede design af programmet – hvilket er helt afgørende efterhånden som programmet vokser i størrelse!

Et teoretisk eksempel herpå ville være følgende, som er en 'applikation' der kan bage brød (!):

```
….Main(){
        hent_ingredienser();
        bland_og_ælt_ingredienser_til_dej();
        bag();
}
```

Ved at analysere problemet viser det sig måske at en eller flere af metoderne skal deles i under-metoder. Fx kunne bland_og_ælt passende deles i to metoder: bland() og ælt_til_dej()!
Princippet er at en metode skal udføre en **begrænset specifik** opgave og ikke fylde for mange kodelinjer. (Nogle vil sige: ikke mere end en halv side).

Metoden arbejder normalt med *lokale* variable:

En simpel metode som dividerer to tal kan se sådan ud:

```
//Illustrerer metoder i C#:

using System;

class Metoder
{
        public static void Main(string[] args)
        {
                double x=44;
                double y=6.7;
                double retur=divider(x,y);
                Console.WriteLine("x: {0}, y: {1}, retur: {2}",x,y,retur);

                //Giver compiler fejl: tal1, tal2, resultat eksisterer ikke!

                Console.WriteLine("\ntal1: {0}, tal2: {1}, resultat: {2}",tal1,tal2,resultat);
```

```
                Console.Read();

        }
        private static double divider(double tal1,double tal2){
                double resultat=tal1/tal2;
                Console.WriteLine("\ntal1: {0}, tal2: {1}, resultat: {2}",tal1,tal2,resultat);

                return resultat;
        }
}
```

Main kalder metoden divider.

NB når metoden er defineret sådan at den tager 2 double som parametre er det vigtigt at den også kaldes med to double ellers fås fejl! **Dog** kan metoden divider() faktisk godt kaldes med et **int** x – fordi et int automatisk kan **castes** til en double af C#! Populært sagt fordi en int (et heltal) er en 'delmængde' af en double: Et int kan godt 'være' en double!

Metoden kopierer tallene 44 og 6.7 (eller om man vil: x og y) over i to **andre** navne tal1 og tal2. Tal1 og tal2 **kunne** kaldes hvad som helst – de er noget helt **andet** end x og y – selv om de har de samme værdier. De 2 metode variable **'lever' kun** så længe metoden kører. De er **'lokale'** variable. De **'dør'** når program flowet vender tilbage til Main() lige før linjen:

```
                Console.WriteLine("x: {0}, y: {1}, retur: {2}",x,y,retur);
```

Hvis vi prøver at udskrive tal1 osv i Main() melder compileren fejl:

```
C:\csharp\metoder>csc main.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

Main.cs(16,61): error CS0103: The name 'tal1' does not exist in the c
        namespace 'Metoder'
Main.cs(16,66): error CS0103: The name 'tal2' does not exist in the c
        namespace 'Metoder'
Main.cs(16,71): error CS0103: The name 'resultat' does not exist in t
        namespace 'Metoder'
```

De findes simpelt hen ikke mere. Da metoden divider() blev kaldt, blev der oprettet et særskilt område i **RAM** hvor disse variable 'levede'. Når metoden slutter bliver dette RAM område (metodens 'stack') slettet.

Vi kunne også have skrevet metoden med et andet navn for resultatet sådan:

```
        private static double divider(double tal1,double tal2){
                double x=tal1/tal2;
                Console.WriteLine("\ntal1: {0}, tal2: {1}, resultat: {2}",tal1,tal2,x);

                return x;
        }
```

Vi opretter nu en variabel double x som hedder **fuldstændigt** det samme som den vi har i forvejen i Main(). Men de 2 x'er har egentligt **intet** til fælles (bortset fra de **tilfældigvis** hedder det samme) ! Metoden sætter x til resultatet af divisionen – **men** når Main() udskriver x, bruger Main() 'sin **egen** x' og udskriver x som **44** (og ikke 6,567...) !! Man kan også sige x i metoden 'skjuler' eller 'overskygger' x i Main().

Parametre har attributter – implicit eller eksplicit:

En parameter i en metode har teoretisk følgende format:

Attribut	Type	Navn ('identifier')
ref, out (eller implicit)	ex: double	ex: tal1

Normalt angives ikke nogen attribut fordi den er underforstået. I vores eksempel er attributten til fx. tal1 at den er en 'ind' parameter og en 'value' parameter dvs. den er en kopi af det x som findes i Main().

En **out** parameter bruges for at metoden kan returnere **flere** værdier – for 'almindelige' metoder kan jo kun returnere een værdi! **Alternativet** til out er altså at skrive **flere** metoder som hver returnerer en værdi!

Et eksempel er en metode med 2 out parametre, som producerer et tilfældigt brugernavn og en tilfældig adgangskode:

```
//Illustrerer metoder i C#:

using System;

class Metoder
{
		public static void Main(string[] args)
		{
				//Skal erklæres men ikke initialiseres:
				string brugernavn,adgangskode;

				//Skal kaldes med out attributten:
				opret_bruger(out brugernavn, out adgangskode);

				Console.WriteLine("BRUGERNAVN: {0}",brugernavn);
				Console.WriteLine("ADGANGSKODE: {0}",adgangskode);

				Console.Read();

		}
		private static void opret_bruger(out string bruger, out string kode){
				Random r=new Random();
				int tal=r.Next(65,98);
				bruger="abc-";
				bruger+=(char)tal;
				kode=(tal*tal).ToString();
```

```
        }
}
```

Metoden opret_bruger() er defineret med 2 out parametre. Den returnerer **void** (!) men pga. out attributten producerer metoden alligevel to strenge, som kan bruges i Main().

Metoden vælger et tilfældigt tal, bruger det som kode og som en del af brugernavnet. (**char**) tal giver ASCII værdien af tallet. (char) 97 er således 'a'.

Normalt arbejder en metode med **kopier** af variable som int, double eller string. Det kan vi ændre ved at give parameteren en attribut **ref.** Nu arbejder metoden **ikke** længere med en **kopi** men med selve **originalen**:

```
//Illustrerer metoder med attributten ref i C#:

using System;

class Metoder
{
        public static void Main(string[] args)
        {
                Console.WriteLine("Indtast et ord:");
                string ind=Console.ReadLine();
                modificer(ref ind);

                //NB strengen 'ind' er nu ændret!:
                Console.WriteLine("Ordet efter metoden modificer(): {0}",ind);

                Console.Read();

        }
        //Metoden ændrer i selve referencen ikke i en kopi!!

        private static void modificer(ref string s){
                s=s.ToLower();
        }
}
```

Hvis attributten ref **fjernes** de 2 steder sker følgende:

Som det ses arbejder modificer() nu kun med en **kopi** af strengen!

De indbyggede C# typer som int, byte, char og string er implicit value parametre. Det er altså nødvendigt at brug **ref** hvis en metode skal arbejde med selve variablen (referencen). Hvis ref bruges arbejder metoden med den **samme adresse** i RAM (i **heap'en** hvor alle ref objekter gemmes) som variablen ligger på. Parameteren er en 'pointer'.

Som vi siden skal se er **klasser** implicit **ref** parametre. Hvis vi har skrveet en klasse **Hus** og en metode som tager et hus som parameter – opfattes parameteren som en **ref** parameter **automatisk**. Metoden ændrer altså i selve objektet hus. Årsagen er at **klassen** Hus er en **reference** type, mens **int** er en **value** type!

Rekursive funktioner/metoder:

Der er skrevet tykke bøger om 'rekursive' funktioner dvs funktioner der kalder sig selv. Rekursive funktioner bliver brugt i mange forskellige sammenhænge fx til søgning i store data mængder.

Rekursive funktioner kan løse problemer som er **meget** vanskelige at løse på en ikke-rekursiv måde fx problemet om Hanois tårne hvor fx 111 ringe på en pind skal flyttes til en anden pind via en tredje pind - uden at en større ring på noget tidspunkt kommer til at ligge oven på en mindre ring.

Rekursive funktioner bygger på det princip at hvis problemet gøres **mindre** er det lettere at løse. Fx er 2 i 9. potens det samme som 2 gange 2 i 8. potens og det er igen lig med 2 gange 2 i 7. osv!!

På samme måde er Hanois tårne ulige nemmere at løse hvis der er 2 ringe end hvis der er 112 ringe!

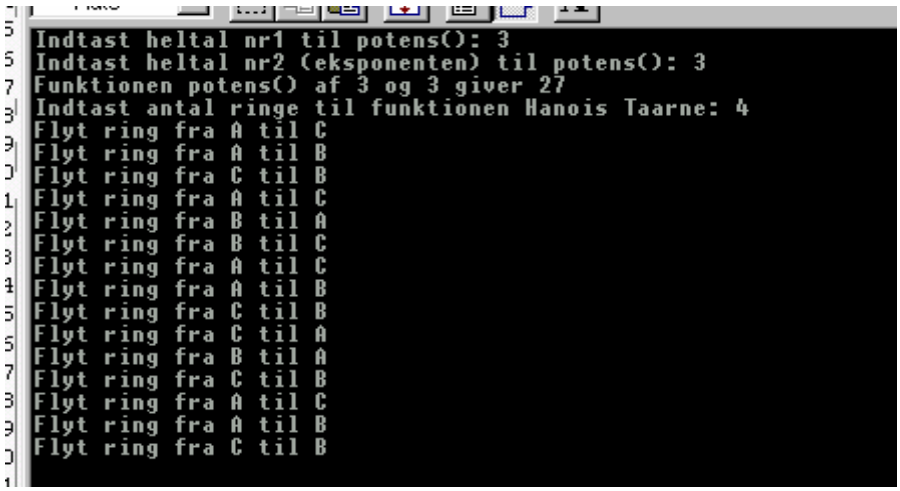
Følgende viser to eksempler på rekursive funktioner. Den første funktion potens() kunne have været beregnet på en ikke rekursiv måde men funktionen hanoi() kan - næsten - kun løses på en rekursiv måde:

```
// Eksempel paa rekursive funktioner:
//potens() og Hanois taarne:


using System;

class app
{
                public static void Main(string[] args)
                {
                                Console.Write("Indtast heltal nr1 til potens(): ");
                                int tal1=int.Parse(Console.ReadLine());
                                Console.Write("Indtast heltal nr2 (eksponenten) til potens(): ");
                                int tal2=int.Parse(Console.ReadLine());

                                Console.WriteLine("Funktionen potens() af {0} og {1} giver
{2}",tal1,tal2,potens(tal1,tal2));

                                Console.Write("Indtast antal ringe til funktionen Hanois Taarne: ");
```

```
                int n=int.Parse(Console.ReadLine());
                hanoi(n,"A","B","C");

                Console.Read();
        }

        //rekursiv funktion der kalder sig selv:

        public static int potens(int x,int y){

                //stop betingelsen er naar y er lig 0 – og det sker paa et eller andet tidspunkt!!:
                if(y==0)return 1;

                //Hvis y ikke er 0 kaldes potens() igen blot med y-1:
                else return x*potens(x,y-1);
        }

        //rekursiv funktion der løser det berømte problem Hanois taarne:
        //de 3 holdere hedder A, B og C
        //der flyttes fra fx A over C til B osv:

        public static void hanoi(int n,string init,string end,string temp){
                if(n==1){
                        Console.WriteLine("Flyt ring fra {0} til {1}",init,end);
                }
                else{
                        hanoi(n-1,init,temp,end);
                        Console.WriteLine("Flyt ring fra {0} til {1}",init,end);
                        hanoi(n-1,temp,end,init);
                }
        }
}
```

Som det ses i potens() bliver funktionen ved med at kalde sig selv indtil der nås en STOP betingelse dvs indtil tallet y er 0.

Alle rekursive funktioner **skal** have en stop betingelse ellers ville de kalde sig selv uden ende som spejle der spejler hinanden uendeligt!



NB **antallet** af flytninger i Hanois tårne er altid (antal ringe*antal ringe) –1! Her er 15 flytninger ((4*4) –1) minimum.

Rekursivt: Drev, mapper og filer

Et godt eksempel på en **rekursiv struktur** er computerens opdeling i mapper, undermapper og filer. Nedenstående program udskriver alle filer på C drevet (Se i øvrigt under **Opgaver**!):

```
//Rekursiv funktion som gennemløber mapper og undermapper:
//og finder filer:

using System;
using System.IO;

class MainClass
{
        public static void Main(string[] args)
        {
                string drev;

                //NB drev skal være et drev eller en mappe som HAR undermapper!:
                drev="C:\\";
                Console.WriteLine();

                find_mapper(drev);


                //Console.Read();
        }
        private static void find_mapper(string etdrev){
                DirectoryInfo dir=new DirectoryInfo(etdrev);
                DirectoryInfo[] mapper=dir.GetDirectories();

                foreach(DirectoryInfo d in mapper){
                                        //Udskriv mappens navn:
                                        Console.WriteLine(d.FullName);

                                        //Rekursivt kald hvis d er en mappe:
                                        find_mapper(d.FullName);

                                        //For hver mappe uden undermapper:
                                        FileInfo[] info=d.GetFiles();
                                        foreach(FileInfo f in info){
                                        Console.WriteLine("    "+f.FullName);
                                        }



                }
        }
}
```

På http://csharpkursus.subnet.dk ligger en Set Time cs-fil som bruger lignende metoder og nulstiller alle filers oprettelses dato mv!

Opgaver:

1. Omskriv programmet om at finde hypotenusen i en ret vinklet trekant så alle de enkelte funktioner **flyttes** over i **hver** sin metode: spørg om tal 1, spørg om tal 1, beregn og udskriv!
2. Hvad er fordelen ved en sådan struktur?
3. Forskellige metoder kan have samme navn hvis de har forskellige parametre! Skriv et program med metoder med forskellige parametre men med **samme** navn: fx beregn(int i), beregn(double d), beregn(byte b) osv. Hvad kan det bruges til?
4. Skriv en **public** metode beregn(), som kalder en **private** metode beregn_side1().
5. Skriv et program som kalder en metode med 4 **out** parametre.
6. Omskriv **mappe** programmet så brugeren kan indtaste en bestemt mappe – fx "C:\\windows"!
7. Omskriv **mappe** programmet så brugeren kan indtaste et filnavn og programmet søger efter filen! Eller: Indtaste en fil type og programmet finder alle filer af den type!

Enumerationer:

I System.Windows.Forms.dll findes utallige 'enumerationer' som definerer forskellige tilstande som et objekt, en kontrol (fx en knap) kan være i eller en farve. Enumerationer er en gammel foreteelse i programmeringen. De bruges til at 'mappe' en talværdi (et heltal) med et symbolsk navn.

Enumerationer er blevet brugt på 2 måder især: For det første er det lettere at huske og arbejde med en symbolsk værdi/et navn end en talværdi. For det andet kan enumerationer bruges til gennemløb.

Et konkret eksempel på en enumeration ville være:

```
public enum Skrift{
        Lille,
        Mellem,
        Stor
}
```

Hvis intet andet angives har 'Lille' værdien 0, 'Mellem' værdien 1 osv. Dette kunne bruges i et tekstprogram til at sætte skriftens størrelse og i stedet for at sætte skriftstørrelsen til 0 (og være i tvivl om hvad '0' nu egentligt var!) kan man med denne enumeration sætte skriften til:

Skrift skr=Skrift.Lille;

Dette kunne forbedres en smule ved at omskrive koden sådan:

```
public enum Skrift{
        Lille=9,
        Mellem=12,
```

```
        Stor=18
}
```

Nu har 'Lille' eksplicit fået værdien 9 og programmet kan så sætte skriften til font størrelse punkt 9 direkte. Stadig er Skrift.Lille meget nemmere at huske.

Som det andet eksempel vil vi se på hvordan en **enum** kan bruges til gennemløb:
Vi vil først definere 2 **enums**:

```
public enum Ugedag{
Mandag=1,Tirsdag,Onsdag,Torsdag,Fredag,Lørdag,Søndag
}
public enum Maaned{
        Januar,Februar,Marts,April,Maj,Juni
}
```

Det typiske for disse er jo, at kalenderen hele tiden gennemløber disse enums – af nemheds grunde er her kun medtaget de første måneder.

I enumerationen Ugedag har vi eksplicit givet Mandag værdien 1 – det betyder at de følgende retter sig efter nr 1 dvs Tirsdag får værdien 2 osv.

I Maaned har Januar derimod værdien 0 - da intet andet er defineret.

I nedenstående program vises en kalender som starter mandag den 13. januar og går 70 dage frem.
NB de 2 enums må IKKE oprettes inden i Main() idet de er selvstændige objekter.
Kalenderen **starter** ved at vi opretter 2 konkrete eksempler (instanser af) på de 2 enums:

```
Ugedag u=(Ugedag)1;
Maaned m=(Maaned)0;
```

Dette **betyder** at u **er** en 'Ugedag' som har værdien 1 altså 'Mandag' og m **er** en 'Maaned'som har værdien 0 altså Januar! Vi skal se dette skema senere i forbindelse med objekt orienteret programmering – enumerationer (og strukturer) er **forløbere** for de objekt orienterede klasser.

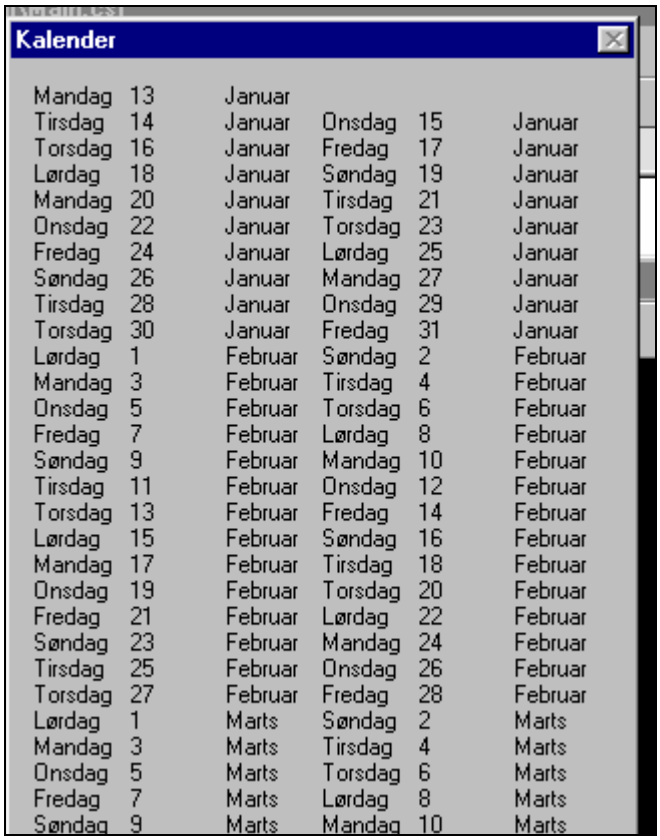
Udtrykket **(Ugedag)1** er en **'cast'** – tallet 1 bliver casted gennem Ugedag og **gjort** til en Ugedag med værdien 1.

NB nedenstående kode er desværre ikke helt tilstrækkelig – måneder på 29, 29, 30 og 31 giver visse problemer!

```
//fil:enum.cs
//illustrerer enums i C# med en kalender:

using System;
using System.Windows.Forms;

class Kalender
{
        public enum Ugedag{
```

```
Mandag=1,Tirsdag,Onsdag,Torsdag,Fredag,Lørdag,Søndag
}
public enum Maaned{
		Januar,Februar,Marts,April,Maj,Juni
}

public static void Main(string[] args)
{
		Ugedag u=(Ugedag)1;
		Maaned m=(Maaned)0;

		//udskriften: Mandag Januar:
		//Console.WriteLine("Ugedag er: {0}, Maaned er: {1}",u,m);

		string kalender=null;
		int divisor;
		int dato=13;
		//kalender skal vise 70 dage:
		for(int i=0;i<70;i++){


				kalender+=(u)+"\t"+dato+"\t"+m+"\t";
				if(i%2==0)kalender+="\n";
				if(u==(Ugedag)7)u=(Ugedag)1;
				else u++;
				if(m==(Maaned)0||m==(Maaned)2)divisor=32;
				else divisor=29;
				dato=++dato%divisor;
				if(dato%(divisor)==0){m++;++dato;}
		}
		MessageBox.Show(kalender,"Kalender");



}
}
```

Resultatet bliver en **MessageBox** der viser datoer, ugedage og måneder:



Flags og enums:

En enum kan erklæres således:

```
[Flags]
enum person {
hunkøn,
hankøn,
statsborger,
ikkestatsborger,
myndig,
ikkemyndig
}
```

Lige før erklæringen er anbragt en 'attribut' Flags som definerer at en Person kan have flere af de nævnte værdier samtidigt! I det kokrete tilfælde er det let at se, at en person kan være både kvinde, myndig og statsborger f.eks.!

To personer kan så oprettes fx sådan:

```
person p1=person.hunkøn|person.myndig|person.statsborger;
person p2=person.hankøn|person.myndig;
```

Opgaver:

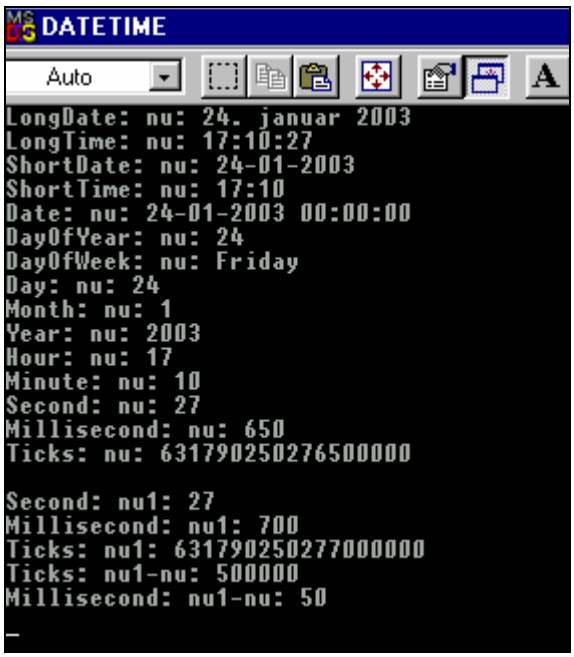
1. Enumerationer bruges om objekter som kun KAN have visse værdier – som fx et spil **kort**. Opret en enum som hedder **Farve** og som rummer de 4 farver i et kortspil
2. Opret en enum **Værdi** med værdier fra 1 til Es
3. Skriv et program som udskriver alle kortene: 'Hjerter Es','Hjerter Konge' osv
4. Skriv et program som giver brugeren et tilfældigt kort som fx 'Ruder Ni'.
5. Skriv et BlackJack spil (her det nemmest at nøjes med værdier fra 1 til 11 !).

DateTime:

C# har en indbygget klasse **DateTime** med en mængde forskellige egenskaber. Mange programmer anvender **klokkeslet** og **datoer** på en eller anden måde. Nedenstående eksempel viser en række af mulighederne i DateTime og hvordan man fx kan måle **hvor lang tid** programmet har kørt og hvor lang tid en bestemt kode blok eller funktion tager målt i tid (se kommentarer i koden!):

```
//DateTime eksempel:
//Beregn hvor lang tid en funktion tager i C#:

using System;

class Tider
{
		public static void Main(string[] args)
		{
				DateTime nu=DateTime.Now;
				Console.WriteLine("LongDate: nu: {0}",nu.ToLongDateString());
				Console.WriteLine("LongTime: nu: {0}",nu.ToLongTimeString());
				Console.WriteLine("ShortDate: nu: {0}",nu.ToShortDateString());
				Console.WriteLine("ShortTime: nu: {0}",nu.ToShortTimeString());
				Console.WriteLine("Date: nu: {0}",nu.Date);
				Console.WriteLine("DayOfYear: nu: {0}",nu.DayOfYear);
				Console.WriteLine("DayOfWeek: nu: {0}",nu.DayOfWeek);
				Console.WriteLine("Day: nu: {0}",nu.Day);
				Console.WriteLine("Month: nu: {0}",nu.Month);
				Console.WriteLine("Year: nu: {0}",nu.Year);
				Console.WriteLine("Hour: nu: {0}",nu.Hour);
				Console.WriteLine("Minute: nu: {0}",nu.Minute);
				Console.WriteLine("Second: nu: {0}",nu.Second);
				Console.WriteLine("Millisecond: nu: {0}",nu.Millisecond);

				//1 millisekund = 10000 ticks
				//Viser antal Ticks siden programmet startede:
				Console.WriteLine("Ticks: nu: {0}",nu.Ticks);

				//tom procedure:
				int t=1;
				for(int i=0;i<1000;i++){
```

```
                    t+=i*44/67;
                }
                //Vi måler igen tiden i et nyt nu:
                DateTime nu1=DateTime.Now;

                Console.WriteLine("\nSecond: nu1: {0}",nu1.Second);
                Console.WriteLine("Millisecond: nu1: {0}",nu1.Millisecond);
                Console.WriteLine("Ticks: nu1: {0}",nu1.Ticks);

                //Hvor lang tid er der gaaet?
                Console.WriteLine("Ticks: nu1-nu: {0}",nu1.Ticks-nu.Ticks);
                Console.WriteLine("Millisecond: nu1-nu: {0}",nu1.Millisecond-nu.Millisecond);

                Console.Read();
            }
}
```



Forskellen mellem de 2 tidsmålinger er altså 50 milli sekunder (1 millisekund er 1/1000 sekund) eller 500.000 Ticks.

En bestemt dato kan oprettes fx sådan:

```
                DateTime dato=DateTime.Parse("24-12-2003 12:00:00");
                Console.WriteLine("LongDate: dato: {0}",dato.ToLongDateString());
                Console.WriteLine("LongDate: dato: {0}",dato.ToLongTimeString());
```

En ny dato kan oprettes sådan:

```
                //Gaa en maaned frem:
                DateTime ny=dato.AddMonths(1);

                //To timer senere:
```

```
ny=ny.AddHours(2);
Console.WriteLine("LongDate: dato: {0}",ny.ToLongDateString());
Console.WriteLine("LongDate: dato: {0}",ny.ToLongTimeString());
```

```
LongDate: dato: 24. december 2003
LongDate: dato: 12:00:00
LongDate: dato: 24. januar 2004
LongDate: dato: 14:00:00
```

En DateTime kan også – lidt nemmere – oprettes sådan:

DateTime en_dato=new DateTime(2002, 10, 28);

Hvilket er datoen den 28. oktober 2002!

Dato og Tids format koder:

Følgende kode fragment viser hvordan en dato (eller et klokkeslet) kan **formatteres** med en række forskellige C# format kode (Vi har tidligere set eksempler med format koder i f.eks. Console.WriteLine()):

```
DateTime d=DateTime.Now;
Console.WriteLine("d: {0:d}",d);
Console.WriteLine("D: {0:D}",d);
Console.WriteLine("F: {0:F}",d);
Console.WriteLine("f: {0:f}",d);
Console.WriteLine("g: {0:g}",d);
Console.WriteLine("G: {0:G}",d);
Console.WriteLine("R: {0:R}",d);
Console.WriteLine("s: {0:s}",d);
Console.WriteLine("t: {0:t}",d);
Console.WriteLine("T: {0:T}",d);
```

Hvis denne kode bruges vil den vise noget i stil med følgende:

```
d: 27-01-2003
D: 27. januar 2003
F: 27. januar 2003 13:26:59
f: 27. januar 2003 13:26
g: 27-01-2003 13:26
G: 27-01-2003 13:26:59
R: Mon, 27 Jan 2003 13:26:59 GMT
s: 2003-01-27T13:26:59
t: 13:26
T: 13:26:59
```

Datoer og klokkeslet i C# kan formatteres på en masse forskellige måder. I øvrigt: **s** betegner en **sortable** dato (kan fx anvendes i en database tabel) og **R** betegner det datoformat som er det officielle i HTTP-protokollen på **internettet**.

Opgaver:

1. Skriv et program der udskriver alle mandage i et år. Brug metoden **AddDays**(int)!
2. Skriv et program hvor du foretager **tidsmålinger** ved hjælp af Ticks og Milliseconds!
3. Skriv et program efter egne ideer der bruger klassen DateTime!

På http://csharpkursus.subnet.dk ligger en fil Vis enumerationer som er et Windows program der kan vise alle enums i .NET. Programmet skal ligge i .NET mappen – sammen med System.dll osv!

Jvf. senere under emnet Reflection!

XML dokumentation af programmer:

I C# er det muligt at dokumentere sine programmer (omtrent som i Java: javadoc).

Vi vil her lave et konkret eksempel på hvordan dette kan gøres. Dokumentationen gemmes ikke i HTML (som i Java) men i **XML** som populært sagt er en slags forbedret generaliseret HTML, men som har langt videre muligheder. F.eks. kan en klasse erklæring gemmes i en XML fil og indlæses siden i stedet for en C# kildekode!!

Følgende eksempel bygger på **Objekt Orienteret Programmering** men kan sikkert let forstås selv om vi ikke har talt om det endnu. Der oprettes simpelthen nogle klasser som skal dokumenteres.

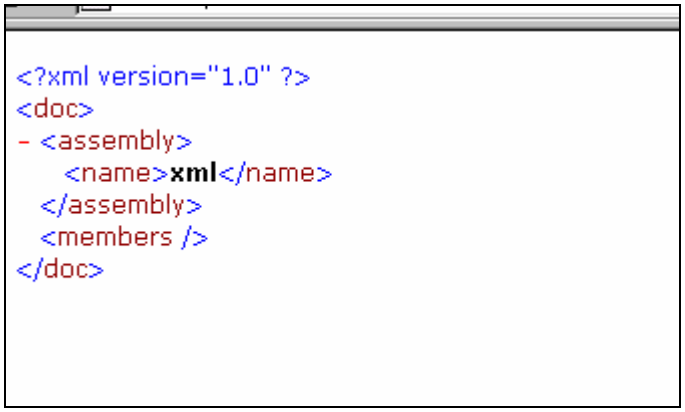
Hvis vi skriver følgende C# fil:

```
//fil: eks.cs
//Demo af dokumentation af fil i XML:
//kør: csc /doc:eks.xml eks.cs

public class XML{
public class Bil{
                public string maerke;
                public int alder;
}
public class Vaskemaskine{
                public string maerke;
                public int kode;
}
public static void Main(){}
}
```

og kompilerer filen med:

csc /doc:eks.xml eks.cs

Nu fås et **magert** resultat idet filen **eks.xml** viser følgende:



Vi kan dog med C# indføre **kommentarer** i filen, som så vil blive vist i XML filen.

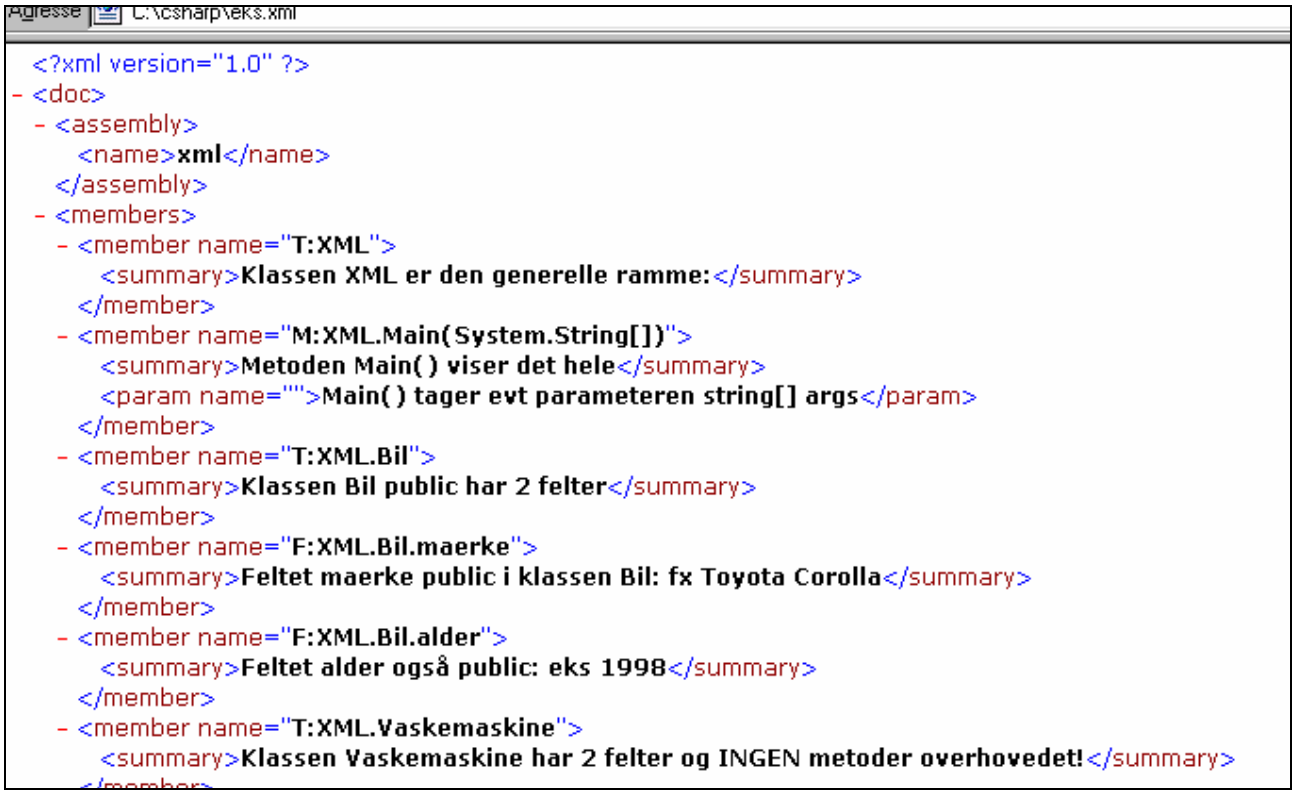
XML kommentarer starter med **///** og varer linjen ud – de er **altid** indelukket i mærker (< >).

```
//fil: eks.cs
//Demo af dokumentation af fil i XML:
//kør: csc /doc:eks.xml eks.cs

///<summary>Klassen XML er den generelle ramme:</summary>
public class XML{
///<summary>Klassen Bil public har 2 felter</summary>

public class Bil{
                ///<summary>Feltet maerke public i klassen Bil: fx Toyota Corolla</summary>

                public string maerke;
                ///<summary>Feltet alder også public: eks 1998</summary>

                public int alder;
}
///<summary>Klassen Vaskemaskine har 2 felter og INGEN metoder overhovedet!</summary>

public class Vaskemaskine{
                ///<summary>Feltet maerke som er string: eks Bosch</summary>

                public string maerke;
                ///<summary>Feltet kode som er public int</summary>

                public int kode;
}
///<summary>Metoden Main() viser det hele</summary>
///<param name="">Main() tager evt parameteren string[] args</param>

public static void Main(string[] args){}
}
```

Hvis denne C# kode køres med den samme kommando linje:

csc /doc:eks.xml eks.cs

fås følgende XML fil dokumentation af CS programmet:



NB:

T betyder klasse eller type
N namespace
F et felt
P en property
M en metode

Læg mærke til at XML filer kan vises i en **browser** som Internet Explorer og at browseren gør det muligt at **udvide** og **sammenklappe** felter i XML filen (med + og -)!

Et **summary** (eller et andet mærke) **skal** anbringes foran en klasse, metode eller felt – hvis den anbringes midt i en metode melder compileren fejl!

I C# kan bl.a. anvendes følgende **mærker** eller **tags** i forbindelse med csc /doc:

Mærke	**Betydning**
<c>	Kode
<example>	Kode eksempel
<list>	En liste
<param>	En parameter til metode
<returns>	Hvad metoden returnerer
<summary>	Forklaring

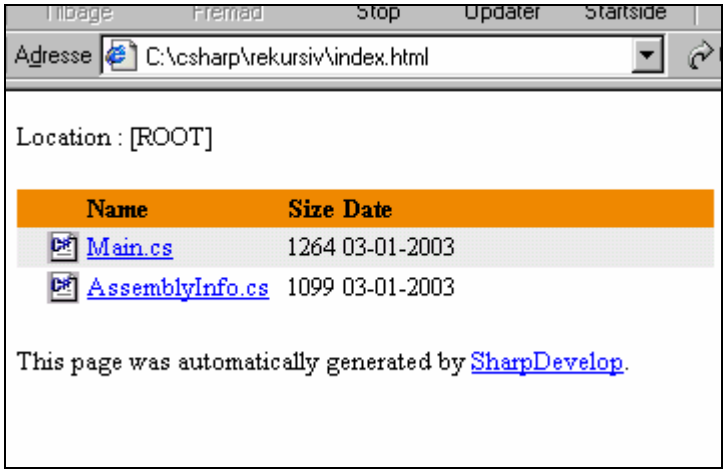
<remarks>	Bemærkninger som summary

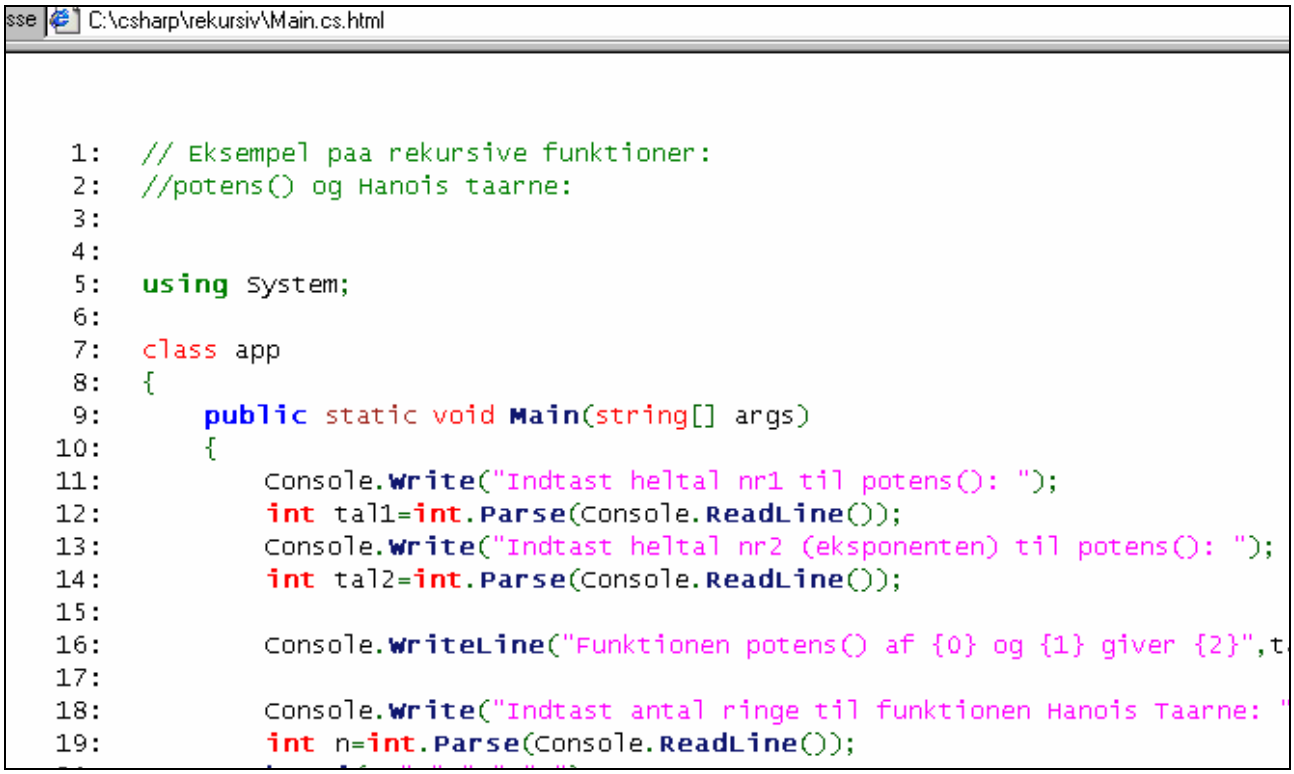
NB - i kommandoen:

csc /doc:**et_eller_andet**.xml eks.cs

kan XML filen hedde hvad som helst!

I **SharpDevelop** kan Project -> Options stilles til at producere en XML dokumentation af klassen/projektet. I SharpDevelop kan dokumentation også ske som **HTML** fil:





I dette tilfælde bliver hele kilde koden simpelthen oversat til HTML.

På adressen http://csharpkursus.subnet.dk ligger et eksempel på hvordan xml doc filer kan vises med XSL stylesheets sådan:

TYPE:XML	
Klassen XML er den generelle ramme:	
METODE:XML.Main(System.String[])	Metoden Main() viser det hele
TYPE:XML.Bil	
Klassen Bil public har 2 felter	
FELT:XML.Bil.maerke	Feltet maerke public i klassen Bil: fx Toyota Corolla
FELT:XML.Bil.alder	Feltet alder også public: eks 1998
TYPE:XML.Vaskemaskine	
Klassen Vaskemaskine har 2 felter og INGEN metoder overhovedet!	
FELT:XML.Vaskemaskine.maerke	Feltet maerke som er string: eks Bosch
FELT:XML.Vaskemaskine.kode	Feltet kode som er public int

Collections:

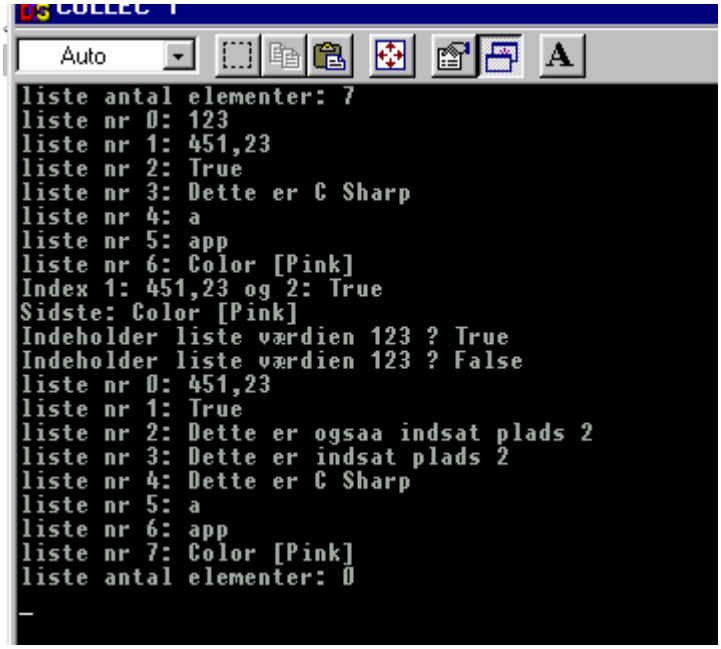
I C# findes en række indbyggede **collections** eller **container** klasser. De findes i namespace **System.Collections**. En collection kan rumme en række af objekter i en liste. Vi har indtil nu brugt en tabel eller array til dette, men C#'s collections er nemmere og mere fleksible i mange tilfælde.

Det følgende er et eksempel på en collection **ArrayList**:

```
//eksemplet viser ArrayList, en af C#s indbyggede collections:

using System;
using System.Drawing;
using System.Collections;//uden denne får du en kompiler fejl

class app
{
		public static void Main(string[] args)
		{
				ArrayList liste=new ArrayList();

				liste.Add(123);//int
				liste.Add(451.23);//float eller double
				liste.Add(true);//et Farve objekt!

				liste.Add("Dette er C Sharp");//string
				liste.Add('a');//char
				liste.Add(new app());//et objekt nemlig denne klasse selv!!
				liste.Add(Color.Pink);//et Farve objekt!

				Console.WriteLine("liste antal elementer: {0}",liste.Count);
```

```
                int i=0;
                foreach(object o in liste){
                        Console.WriteLine("liste nr {0}: {1}",i++,o.ToString());
                }
                Console.WriteLine("Index 1: {0} og 2: {1}",liste[1],liste[2]);
                Console.WriteLine("Sidste: {0}",liste[liste.Count-1]);

                Console.WriteLine("Indeholder liste værdien 123 ? {0}",liste.Contains(123));
                liste.RemoveAt(0);
                Console.WriteLine("Indeholder liste værdien 123 ? {0}",liste.Contains(123));
                liste.Insert(2,"Dette er indsat plads 2");
                liste.Insert(2,"Dette er ogsaa indsat plads 2");
                i=0;
                foreach(object o in liste){
                        Console.WriteLine("liste nr {0}: {1}",i++,o.ToString());
                }
                liste.Clear();
                Console.WriteLine("liste antal elementer: {0}",liste.Count);


        Console.Read();
        }
}
```

En kørsel giver dette resultat:



Kommentar:

Som det ses er ArrayList meget mere fleksibel end en almindelig tabel eller array. De metoder som vises her på ArrayList findes i de fleste collections i C#. En ArrayList har IKKE nogen fast max **størrelse** (som et array) og er ikke **type** begrænset. Som det ses kan forskellige typer lægges i den

samme liste. En ArrayList kan endda føjes til en ArrayList så at et element i listen består af en ny liste.

En ArrayList er **'dynamisk'** allokeret dvs den tilpasser sig hele tiden det størrelses behov som opstår. En sådan data-collection kaldes også en **'vektor'**. Når listen åbnes har den en 'Capacity' på 2 elementer – hvis nødvendigt øges Capacity først til 4, så til 8, så til 16 osv!

Læg også mærke til hvad der sker når vi sletter 1. element og hvad der sker med hele listen når vi indsætter nye elementer på en bestemt plads! Hvis disse manipulationer skulle foretages på et almindeligt array ville det kræve en hel del manuel kode. ArrayList er altså også et eksempel på objekt orienteret 'indkapsling' (jvf senere) – vi ser ikke den mekanik som ligger bagved metoderne – vi **bruger** dem blot!

En ArrayList kan også gennemløbes med en **Enumerator** således:

```
IEnumerator enumerator = liste.GetEnumerator();
   while (enumerator.MoveNext())
     Console.WriteLine(enumerator.Current);
```

Idet Current er det aktuelle element. Metoden MoveNext() returnerer sand eller falsk.

Eksempel på strenge og collections: En sorteret ordliste:

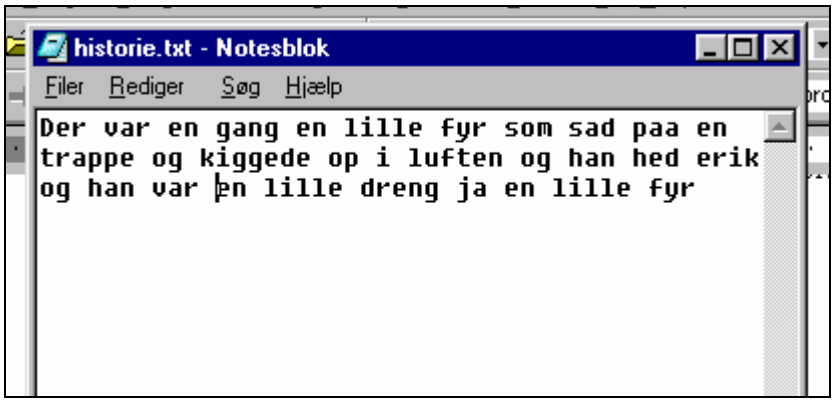
Følgende eksempel bruger også en collection og er et eksempel på **parsing** dvs databehandling af en streng – noget der spiller en stor rolle i databehandling. Eksemplet anvender også **filer** – som vi ikke rigtigt har set på endnu – men er meget enkelt.

Det vi gerne vil er at læse en hel fil, finde antallet af ord, finde de enkelte ord, opstille en ordliste hvor hvert ord kun forekommer een gang og sortere den **alfabetisk**!

En samling eller collection hvor hvert element **kun** forekommer **een** gang kaldes et **set** og bruges ofte i program løsninger.

C# klasserne gør alt dette **overraskende** nemt som det kan ses nedenfor.

Vi har først skrevet en lille dum tekst, som har den fordel, at den er let at overskue:



Når programmet kører får vi denne sorterede **'set'** liste:

C# **koden** ser således ud:

```
//Eksempel paa parsing af strenge:
//Metoder i System.String:

//konstruerer en alfabetisk sorteret liste over ord i fil:

using System;
using System.IO;
using System.Collections;

class Ordtabel
{
		public static void Main(string[] args)
		{
				//hent fil navn fra DOS linjen:
```

```
                    string s=null;

                    //Åbn en fil og læs den med en StreamReader:
                    StreamReader r=null;

                                try{
                    r = File.OpenText (args[0]);
                    s = r.ReadToEnd();
                                }finally{
                                            r.Close();
                                }


                    //Split splitter her ved mellemrum ' ' og betyder at hvert ord er en plads i tabellen:
                    string[] tabel = s.Split (' ');

                    //find fx antal ord i filen:
                    Console.WriteLine ("Ord ialt: {0}", tabel.Length);

                    //ArrayList bruges som et Set: kun eet eksemplar af hvert ord!
                    ArrayList liste=new ArrayList();
                    for(int i=0;i<tabel.Length;i++){
                                if(!liste.Contains(tabel[i]))liste.Add(tabel[i]);
                    }

                    //at sortere er en standardmetode i ArrayList!
                    liste.Sort();

                    //udskriv resultatet:
                    foreach(string str in liste){
                                Console.WriteLine(str);
                    }


                    //Console.Read();
          }
}
```

Hash-tabeller:

Collections eller containerklasser findes i forskellige **basistyper** som fx List eller Hashtable. En Hashtable anvender et fænomen som kaldes 'hashing' som går ud på at finde et heltal (integer) for hvert objekt som gemmes i en liste. Ideen med hashing er at systemet meget hurtigere skal kunne finde en værdi som er gemt i en liste. Hvis vi havde gemt værdien i et almindeligt Array, skulle programmet jo løbe alle elementerne igennem fra nummer 0 for at chekke om elementet fandtes og hvor det var gemt!

Meget simplificeret kan vi forklare **hashing** således: Hvis "erik jensen", "ulla hansen" og "direktør Pedersen" skal gemmes i en Hashtable beregner vi et heltal for hver af disse strenge ved hjælp af en hash **algoritme** (metode). Hashværdierne ville måske blive 155, 22 og 956. (Hvis disse pladser allerede er 'optaget' lægges værdien på den første frie plads – f.eks.). De tre værdier ville så blive gemt i en tabel på tabel[155], tabel[22] og tabel[956]! Når systemet siden bliver spurgt: Findes

værdien ”erik jensen”? beregner systemet hashkoden og går direkte til index 155!! På den måde fungerer en Hashtable meget hurtigere end en almindelig tabel!!

Modsat en almindelig tabel gemmes data i en hashtable altid som et par af en **key** og en **value** (lige som en **ordbog** eller en **telefonliste** med navn (key) og telefonnummer (value)). Som value kan anvendes alle objekter også fx en ArrayList! En hashtable er altså meget fleksibel og anvendelig til at gemme data systematisk!

Følgende eksempel på anvendelsen af en hash tabel anvender metoder fra System.IO (filer) som vi vil gennemgå på et senere tidspunkt. Programmet opretter en række keys (fil navne) og tilsvarende values (filernes tekstindhold) – hvilket giver en meget **hurtig** tilgang til disse filtekster bagefter!:

```
// eksempel paa en Hashtable

using System;
using System.Collections;
using System.IO;

class MainClass
{
		private static Hashtable tabel;

		public static void Main(string[] args)
		{
				tabel=new Hashtable();
				DirectoryInfo dir=new DirectoryInfo(@"C:\csharp\csfiler");
				FileInfo[] filer=dir.GetFiles();
				StreamReader reader=null;
				string indhold=null;
				foreach(FileInfo fil in filer){
						reader=File.OpenText(fil.FullName);
						indhold=reader.ReadToEnd();
						reader.Close();

						//læg data ind i vores hashtabel: key – value:
						tabel.Add(fil.Name,indhold);
				}
				Console.WriteLine("Indtast filnavn:");
				string sv=Console.ReadLine();

				//tabel[sv] udskriver den value som svarer til  key 'sv':
				if(tabel.Contains(sv)){
				Console.WriteLine(tabel[sv]);
				}
				else Console.WriteLine("Filen findes ikke i hash tabellen!");
		}
}
```

HashTable indeholder også nyttige metoder og properties som:

if(tabel.ContainsValue(”...”))
tabel.Remove(key-streng)
tabel.Count
tabel.IsReadOnly=true

En HashTable kan f.eks. gennemsøges efter en bestemt værdi (Value) således hvor koden finder alle de par af key-value, hvor værdien er ”Marie-Louise”:

```
string name = "Marie-Louise";

   if (table.ContainsValue(name))
   {
    IDictionaryEnumerator enumerator = table.GetEnumerator();
    while (enumerator.MoveNext())
    {
     Console.WriteLine( "{0} = {1}", enumerator.Key, enumerator.Value);
    }
   }
```

Man kan let oprette sine egne ’typede’ collections ved at arve fra en bestående collection - evt fra **CollectionBase** fx på denne måde:

```
public class ImageCollection : CollectionBase
{
 public ImageCollection()
 {
 }

 public void Add(Metafile value)
 {
  InnerList.Add(value);
 }

 public void Remove(Metafile value)
 {
  InnerList.Remove(value);
 }
}
```

Arv vil blive gennemgået lidt senere i kurset.

Opgaver:

1. Skriv et program hvor computeren finder 100 **tilfældige** tal mellem 0 og 100 og som chekker hvor mange ’gengangere’ der er! Prøv at køre programmet nogle gange og undersøg resultatet!
2. Skriv et program som indlæser en ordliste fra en fil, udskriver hvor mange ord der er i alt og hvad det gennemsnitlige **antal** tegn i hvert ord er! (Et ’LIX’ tal!).
3. Klassen System.String har en metode: str.IndexOf(”NET”) som returnerer –1 hvis ordet ”NET” **ikke** forekommer i str og ellers returnerer det index hvor ordet befinder sig. Skriv et program hvor man kan **søge** efter om et ord findes i teksten. Indlæs teksten fra en fil.
4. Skriv et program finder 10000 tilfældige tal mellem 0 og 100.000 og sorterer dem **stigende** og i **faldende** rækkefølge! ArrayList har metoden Reverse().

5. Skriv et 'database' program med en hashtabel hvor man kan gemme data systematisk og søge effektivt!

Reflection (og om hvordan man kommer videre):

Begrebet **reflection** bruges om C#'s evne til at finde information om 'sig selv', sine egne klasser, DLL filer og assemblies. Hvis denne information skaffes 'run time' kaldes den RTTI Runtime Type Identification. Vi skal se mange eksempler på det i forbindelse med OOP.

Reflection bruges ellers til at skaffe information om indholdet i en C# DLL eller EXE. .NET installeres med en række faste DLL filer som System.dll og System.Data.dll osv, der indeholder C#'s **basisklasser (BCL)**.

Når man skal finde rundt i disse basisklasser skal man være opmærksom på at namespaces (der er **logiske** strukturer) og DLL filer (som er **fysiske** virkeligt eksisterende strukturer) **ikke** altid svarer til hinanden. Fx findes klassen **System**.**Array ikke** i System.dll (hvad man ville tro!) men i **mscorlib**.dll!

Reflection er også helt afgørende når C# skal kommunikere med andre fremmede objekter fx COM objekter fra den traditionelle Windows platform.

Namespace **System**.**Reflection** indeholder de DLL filer som rummer denne funktionalitet. Programmet nedenfor anvender derfor en: **using System.Reflection** – for at inddrage dette namespace.

Som nævnt tidligere indeholder dette namespace også klassen Assembly:

Assembly.Load("System.Windows.Forms");

Denne sætning loader DLL filen System.Windows.Forms I RAM hukommelsen således at programmet kan 'referere' klasserne i Forms.dll.

Reflection gør det altså muligt **dynamisk** dvs mens programmet kører at inddrage, ændre og skrive til DLL filer og assemblies!

Reflection gør det også muligt at loade en bestemt version af en DLL fil – således at mange forskellige versioner af den samme DLL fil kan eksistere på samme maskine (noget der før .NET ikke var muligt).

Reflection vil her blive brugt til at udvikle nogle programmer der kan give os information om C#'s basisklasser.
Uden en sådan viden er det **ikke muligt** at arbejde fornuftigt med C#!

Udgangspunktet for den følgende kode - programmet **dll_reflection**.**exe** - er:

Hvilke klasser mv ligger der i en bestemt **DLL**-fil som f.eks. System.Windows.Forms.dll eller – hvilket er det samme – hvilke typer indeholder den **assembly** som hedder 'System.Windows.Forms'? (Navnet på assembly'en er altid 'uden .dll'!).

Det følgende program er et nyttigt program til at finde de indbyggede klasser i C#. Lidt senere skal præsenteres et program som kan vise data for den **enkelte** klasse.

dll_reflection.exe:

```
//fil: dll_reflection.cs

//eks på reflection, få viden om en class i C# som inputtes som kommando linje parameter

//eks: dll_reflection System.Web eller dll_reflection person (lokal DLL fil)
//navne som 'System.Web' eller 'person' kaldes i C# jargon'en for 'friendly names' (modsat kodede navne)

//programmet skal ligge i .NET mappen sammen med csc o.a. ellers kan det ikke 'finde' C#-kerne-DLL'erne
//programmet producerer en HTML fil med data om DLL filen:

using System.Reflection;
using System;
using System.IO;

public class app{
        public static void Main(string[] args){
                //try .. catch fanger evt fejl fx at der indtastes en ikke eksisterende DLL fil!
                Assembly ass=null;
                try{
                //assembly'en skal loades I RAM for at kunne analyseres:
                ass=Assembly.Load(args[0]);

                }catch{
                Console.WriteLine("{0} er ikke en gyldig DLL!",args[0]);
                }

                //GetTypes() returnerer en tabel af typer eller klasser:
                Type[] types=ass.GetTypes();
                string str=null;

                //foreach anvendes fordi antallet af klasser I DLL filen ikke kendes på forhånd
                foreach(Type t in types){
                  str+=t+"<br>";
                }

                //CreateText() returnerer en Writer:
                //hvis filen findes i forvejen overskrives den:
                //filen navngives med DLL navnet:
                StreamWriter writer=File.CreateText(args[0]+".html");

                //der skrives HTML koder til filen:
                writer.WriteLine("<html><head><title>{0}</title></head>",ass);

                writer.WriteLine("<body><h1>{0}</h1>",ass);
                writer.WriteLine("<font size=4>{0}</font></body></html>",str);
```

```
                                //NB en stream skal altid åbnes og lukkes som minimum!
                                writer.Close();
                                }
}
```

Programmet foretager chek på om det indtastede argument er en gyldig DLL fil. Dette gøres med en **try**..**catch** konstruktion. Vi vil senere vende tilbage til hvordan **exceptions** kodes i C#.
Det er afgørende vigtigt at evt. kommende bruger 'fejl' opfanges i koden - hvis det er muligt.

Den kode som skriver til HTML filen minder meget om et kommende eksempel på håndtering af filer og vil blive gennemgået i afsnittet om **filer**. Som det ses er metoderne Write() og WriteLine() de **samme** uanset om der skrives til skærmen eller til en fil! (Dette er faktisk et eksempel på objekt orienteret **polymorfisme** (samme metoder men forskellige sammenhænge/objekter) – som det vil blive gennemgået i det senere afsnit om objekt orienteret programmering).

Programmet producerer en HTML fil (en meget nyttig DLL-dokumentationsfil) som f.eks. i det følgende eksempel hvor programmet er kørt med parameteren **mscorlib**:

HTML filen opregner alle **klasser** (typer) i C# kerne-biblioteket mscorlib.dll.

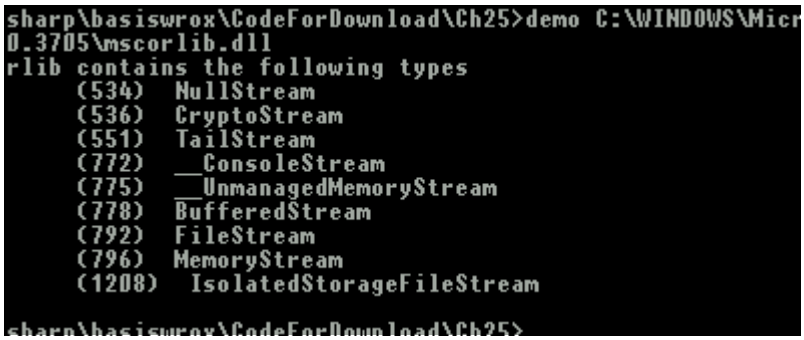
Hvis man i stedet for Assembly.Load() bruger Assembly.**LoadFrom**() – som tager en sti! – kan man referere til en hvilken som helst DLL eller EXE:

```
Assembly a=Assembly.LoadFrom("C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705\mscorlib.dll");
```

Vi kan også ved hjælp af reflection og metoden **IsSubclassOf()** fx finde alle de structs (en struct eller struktur arver fra System.ValueType), der ligger i en bestemt DLL på denne måde:

```
Type[] types = assembly.GetTypes();
   for (i=0; i<types.GetLength(0); ++i)
   {
     if ( types[i].IsSubclassOf(typeof(System.ValueType)) ){

    Console.WriteLine("\t ("+i+")  "+types[i].Name);
     }
   }
```

Eller man kan søge efter typer/klasser som er subklasser af System.IO.Stream:



På adressen http://csharpkursus.subnet.dk ligger der andre eksempler på C# programmer der anvender reflection.

reflection.exe:

Det efterfølgende program udfører et tilsvarende arbejde som dll_reflection idet det udskriver alle 'medlemmer' af en bestemt klasse I C# (dvs klassens egenskaber og metoder).

```
//fil: reflection.cs

//eks på reflection, få viden om en class i C#

//eks: reflection System.IO.File
//giver en liste paa 27 medlemmer af klassen File i namespace System.IO
//inkl nedarvede metoder fx fra System.Object

//output sendes i HTML fil med klassens navn

using System.Reflection;
using System;
using System.Windows.Forms;
using System.IO;

public class app{

                public static void Main(string[] args){
```

```
                        string klasse=null;
                        if(args.Length>0) klasse=args[0];
                        else klasse="System.Reflection.MemberInfo";//valgt som eksempel:

                        Type type=Type.GetType(klasse);
                        string tekst=null;
                        MemberInfo[] info=type.GetMembers();
                        int i=0;
                        for(;i<info.GetLength(0);i++){
                          tekst+=i+1+". "+info[i].Name+" : "+info[i].MemberType.ToString()+"<br>";
                        }

                        //CreateText() returnerer en Writer:
                        //OBS - hvis filen findes i forvejen overskrives den:

                        //HTML filen kaldes klassenavnet + html fx 'System.IO.StreamReader.html':
                        StreamWriter writer=File.CreateText(type.Name+".html");

                        //der skrives HTML koder til filen:
                        writer.WriteLine("<html><head><title>{0}</title></head>",type);

                        writer.WriteLine("<body><h1>{0}</h1>",type);
                        writer.WriteLine("<font size=3>{0}</font></body></html>",tekst);

                        //NB en stream skal altid både åbnes og lukkes … som minimum!
                        writer.Close();

                }
}
```

Opgaver:

1. **Omskriv** ovenstående kode så programmet viser klassens metoder, deres parametre, parametertyper osv. Du skal bruge følgende kode formler til at få denne effekt (men selve opskriften er den samme som i reflection.cs):

   ```
   MethodInfo[] info=type.GetMethods();
   ---
   tekst+="<hr>"+i+". METODE NAVN:"+info[i].Name+"<br>";
   ---
   ParameterInfo[] param=info[i].GetParameters();
   ---
   foreach(ParameterInfo pi in param){
   ```

2. På adressen http://csharpkursus.subnet.dk kan du finde et andet program Reflection som kan producere en tekstfil fx over alle klasser, metoder, properties, events osv i System.dll (eller en anden dll). **Download** dette program og **tilpas** det! Den klart mest effektive metode når man skal **søge** er at have en fuldstændig **tekstfil** over en DLL og så bruge Søg i et **tekstbehandlings** program som Word eller Wordpad!! Man kan anvende Search fx i SharpDevelop – men det fungerer **ikke** optimalt! Med denne metode kan du producere tekstfiler der effektivt kan bruges til opslag i C#'s basisklasser!

Reflection i SharpDevelop:

Ved at bruge **IDE'en** SharpDevelop kan man - uden alt for meget arbejde - se på DLL filer og klasse erklæringer.

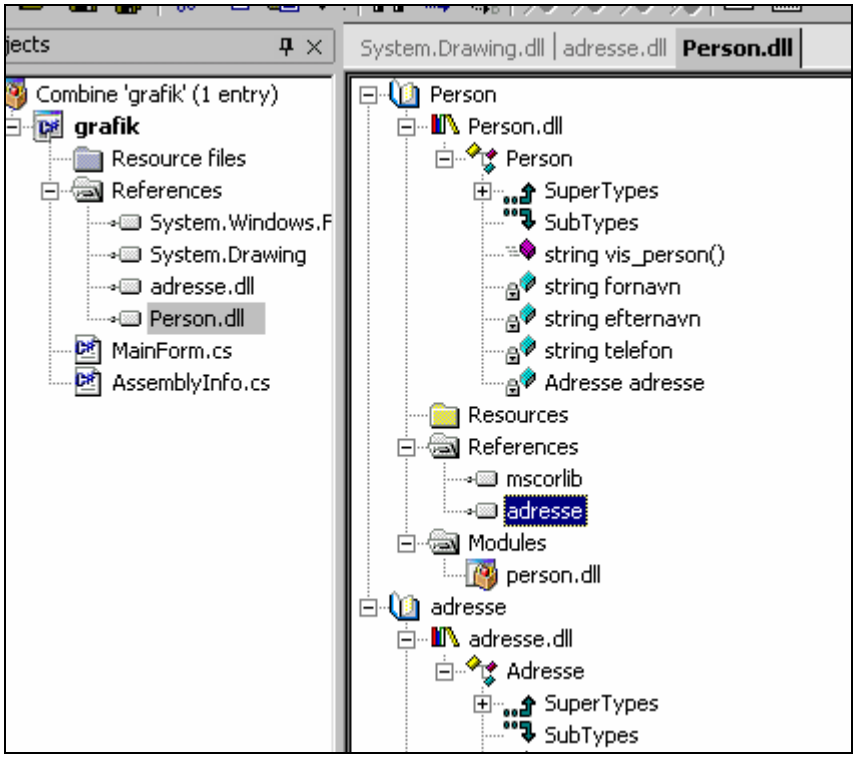
I SharpDevelop vises klasser med den såkaldte **UML** notation fx er private felter påsat en hængelås og en property vises som en hånd der peger på en liste.

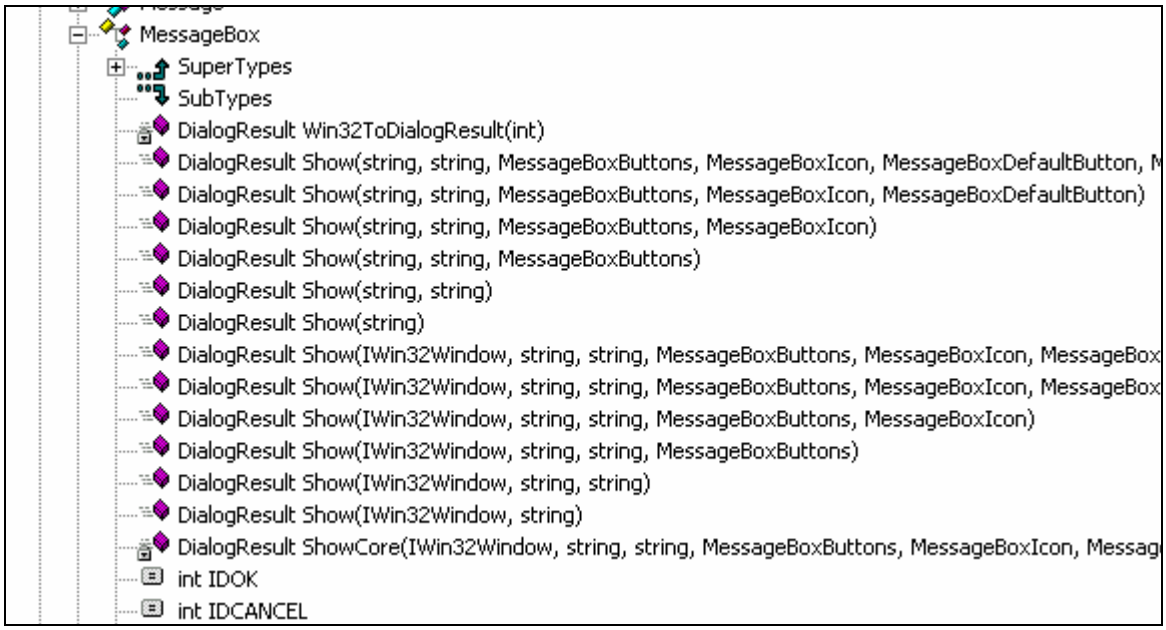
SharpDevelop er **meget nyttig** når man skal finde rundt i C#'s basis klasser – og sine egne 'hjemmelavede' !

Gør sådan: Opret et projekt i SharpDevelop ved at vælge File->New Combine->C# og Windows Form Project. Vælg View->Projects og Projekt vinduet vises. Højre klik References, vælg Add Reference og tilføj en DLL (fx System.Drawing) til projektet. Du kan nu dobbeltklikke på den nye reference i Projekt vinduet og få indholdet af DLL filen vist i et nyt vindue.

Nedenstående er et eksempel herpå.

I første tilfælde viser SharpDevelop to 'user defined' klasser Person og Adresse og deres indbyrdes relation og medlemmer:





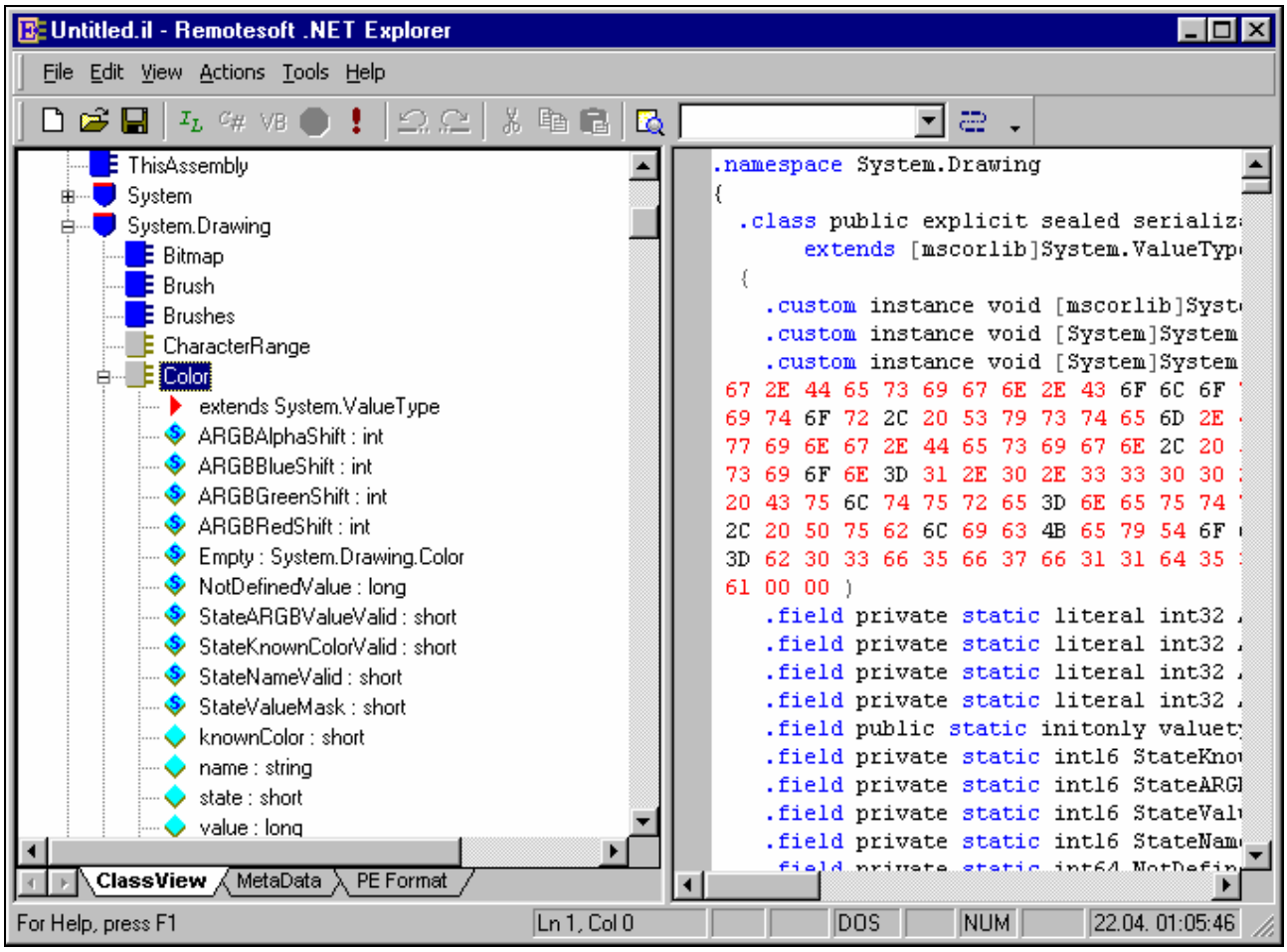
Det ovenstående eksempel viser et opslag under Klassen **MessageBox** og dens 13 (!) forskellige Show metoder! NB DialogResult er den type som boksen returnerer (fx DialogResult.Cancel hvis brugeren klikker Fortryd eller Annuller). Læg mærke til at en metodes parametre altid er angivet med deres type.

På denne måde – ved hjælp af C# reflection – kan du selv **'komme videre' på egen hånd**! Du finder klassen og dens medlemmer, metoder og egenskaber – og begynder at eksperimentere!

Du vil også normalt kunne 'arve' fra C# basis klasserne – som vi skal se senere: Dvs. du kan udbygge og videreudvikle en C# basisklasse med dine egne metoder og medlemmer.

Du vil **typisk** anvende klassens metoder, events og properties (Jvf senere om Objektorienteret Programmering).

Et andet **meget** nyttigt værktøj er .NET Explorer fra http://www.remotesoft.com som kan vise en lang række data om C# klasser:



Opgaver:

1. Find klassen System.Drawing.**Color** ved at åbne DLL filen i SharpDevelop. Hvilke **medlemmer** har denne basisklasse?
2. På hvilken måde kunne disse **bruges**?
3. Hvilke farver er på forhånd **defineret**? Hvordan kan de bruges?
4. Skriv et lille **program** hvor du tester klassen Color efter nedenstående skema (Vi vil ikke i dette kursus gå nøjere ind i Windows programmeringen).

```
//colordemo.cs

using System;
using System.Drawing;
using System.Windows.Forms;

public class ColorDemo {
		//verdens mindste windows program
		//KUN til demo af klassen Color:

		public static void Main(){

			Form form=new Form();
			Color color=Color.FromArgb(100,200,100);
```

```
                    //sætter formens baggrunds farve:
                    form.BackColor=color;
                    form.ShowDialog();

                }
}
```

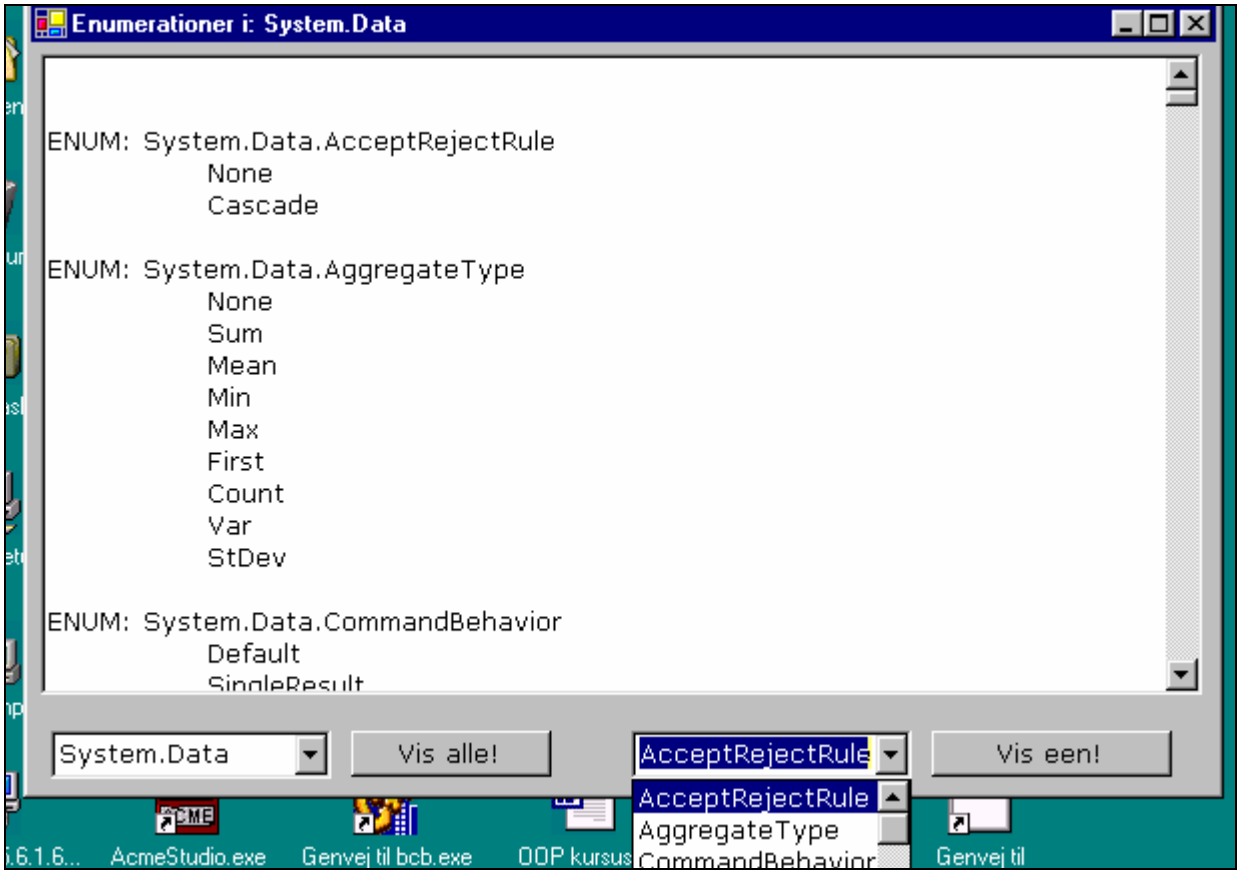
Ved at **eksperimentere** videre på egen hånd kan man komme videre. Egentligt **er** der ingen anden metode!

Enum reflection i Windows program:

På http://csharpkursus.subnet.dk ligger en fil Vis **enumerationer** som er et Windows program der kan vise alle enums i .NET. Prøv at se på koden og tilpas programmet til dit eget brug. For at kunne finde DLL filerne skal programmet ligge i .NET mappen – i **samme** mappe som System.dll osv!

Som vi vil se – er disse **enums** især vigtige i Windows programmeringen, og de kan være svære at finde rundt i!

Når programmet kører ser det sådan ud:



Filer:

At kunne **læse** fra og **skrive** til filer er i de fleste programmer en afgørende del af programmet. Dette gælder ikke bare tekstbehandlingsprogrammer, men også grafiske og database programmer.

En **fil** er en række af bits og bytes der ligger gemt et sted på computeren (i det 'sekundære' lager: på en CD, diskette, en harddisk osv). En fil har desuden et navn, en størrelse, forskellige datostemplinger (ex sidst modificeret) og typisk en filtype (i al fald i Windows).
Der findes især 2 helt forskellige slags filer: **tekstfiler** og **binære** filer (binære filer er fx bitmap eller lydfiler).

Vi vil lige nu koncentrere os om tekstfiler (Senere i forbindelse med objekt orienteret programmering vil vi se på binære filer):

En **'stream'** er i C# en åben forbindelse (et rør, en pipe) hvorigennem kan strømme bytes den ene eller den anden vej (Read = bytes ind, Write = bytes ud gennem røret).

En stream skal altid først åbnes og siden (efter læsning/skrivning) lukkes. Det er vigtigt at en stream lukkes korrekt.

I C# findes en række Reader og Writer klasser som er indbyggede klasser i C#.

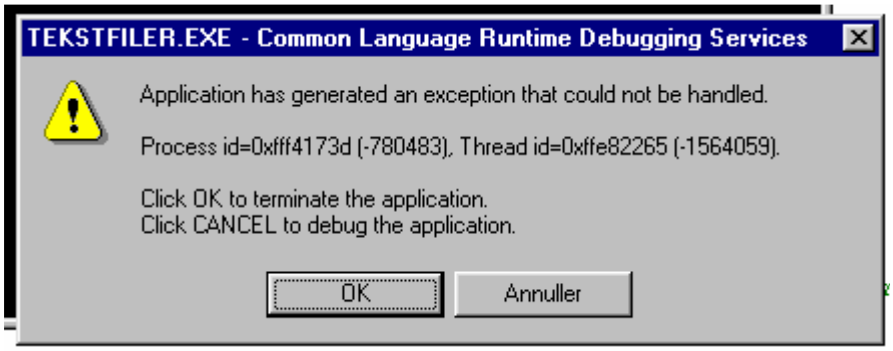
Klassen **StreamReader** læser tekstfiler som er gemt i Unicode (eller ASCII) format – dvs tekstfiler i bred forstand (inklusive TXT HTML DOC filer osv). Metoderne i StreamReader er især 3 forskellige Read metoder.
Klassen **File** opretter filobjekter og har en række **static** metoder. En static metode kaldes ikke på et objekt men på selve klassen, altså på klassenavnet.

Dette eksempel illustrerer en static metode i File nemlig File.OpenText() og StreamReader:

```
//fil: tekstfiler.cs
//programmet afvikles fx: tekstfiler tekstfiler.cs – med 1 kommandolinje parameter
//postcondition: outputter filens indhold til skærmen

using System;
using System.IO;//i dette namespace fides alle fil klasser

public class app{
  public static void Main(string[] args){

                    //OpenText() returnerer en Reader:
                    StreamReader reader=File.OpenText(args[0]);

                    //ReadToEnd(): læs hele filen
                    string fil=reader.ReadToEnd();
                    Console.WriteLine("FIL: {0}:\n{1}",args[0],fil);

                    //NB en stream skal altid åbnes og lukkes som minimum!
```

```
                reader.Close();

                Console.Read();//teknisk af hensyn til Windows
  }

}
```

Det er et typisk problem med denne applikation: Hvis filen 'args[0]' ikke eksisterer opstår en fejl – en såkaldt **exception** og resultatet kan være at programmet crasher og en boks som denne vises:



Programmet kan simpelthen ikke fortsætte. Filer er et typisk område hvor en exception kan opstå. Men exceptions kan opstå i alle programmer. I første omgang vil vi løse dette problem på den nemmeste måde nemlig med en simpel **try**..**catch.**

Her er et eksempel på hvordan koden **kan** ændres:

```
public static int Main(string[] args){

                StreamReader reader=null;
                //OpenText() returnerer en Reader:
                //forsøg følgende - hvis noget går galt gå ned til catch:
                try{
                reader=File.OpenText(args[0]);
                }
                //alarmer bruger og stop programmet:
                catch{
                                Console.WriteLine("Filen {0} kan ikke findes.",args[0]);
                                Console.Read();
                                return 1;//1 betyder traditionelt fejl
                }
                finally{ if(reader!=null)reader.Close(); }

                //Hvis alt går godt: ReadToEnd(): læs hele filen
                string fil=reader.ReadToEnd();
                Console.WriteLine("FIL: {0}:\n{1}",args[0],fil);


                //NB en stream skal altid åbnes og lukkes som minimum!
                reader.Close();

                Console.Read();//teknisk af hensyn til Windows
                return 0;
}
```

Som det fremgår af kommentarerne i koden, løses problemet med en **try** blok og en efterfølgende **catch** blok. Hvis noget går galt – spring **straks** frem til catch! Her lader vi programmet terminere/slutte i catch blokken – derfor er **Main** skrevet om til **public static int** Main(). Denne form muliggør at programmet kan afsluttes på 2 (eller 27 forskellige!) måder som vist i eksemplet.

Det vigtige i forbindelse med **exceptions** er at prøve at forudsige alle de ting som kan gå galt og så kode med try catch finally! I forbindelse med Objekt orienteret programmering vil vi komme meget nøjere ind på **exception handling** og exception **klasser**.

Vær også opmærksom på at mere kunne gå galt i eksemplet ovenfor: metoden ReadToEnd() kunne også fejle – hvis programmet skulle være mere perfekt skulle også denne Read omgærdes af en try catch!

Som det kan ses er det - i øvrigt - rimeligt enkelt at læse en fils indhold og udskrive indholdet til skærmen. I programmet angives filnavnet som parameter til programmet og der oprettes en Reader. I stedet for ReadToEnd() kunne have været anvendt metoderne: Read() som læser et tegn eller metoden ReadLine() som returnerer en string og læser en linje. Hvis det ønskes at læse en linje ad gangen (og evt nummerere linjerne) kunne koden se sådan ud:

```
string linje=null;
while((linje=reader.ReadLine())!=null)Console.WriteLine(linje);
```

En **Writer** skriver til en ny fil eller til en allerede eksisterende fil. Det følgende illustrerer anvendelsen af den indbyggede klasse StreamWriter:

```
//fil: skrivfiler.cs
//postcondition: opretter en html fil med brugerens input

using System;
using System.IO;//i dette namespace fides alle fil klasser

public class app{
  public static void Main(string[] args){

                    //CreateText() returnerer en Writer:
                    //hvis filen findes i forvejen overskrives den:
                    StreamWriter writer=File.CreateText(args[0]);

                    //der skrives HTML koder til filen (titelinje, overskrift, afsnit)
                    writer.WriteLine("<html><head><title>{0}</title></head>",args[0]);

                    //data fås fra brugeren om indholdet af HTML filen:
                    Console.WriteLine("Skriv en overskrift til HTML filen:");
                    string s=Console.ReadLine();
                    writer.WriteLine("<body><h1>{0}</h1>",s);
                    Console.WriteLine("Skriv et afsnit til HTML filen og tast ENTER:");
                    s=Console.ReadLine();
                    writer.WriteLine("<font size=4>{0}</font></body></html>",s);

                    //NB en stream skal altid åbnes og lukkes som minimum!
                    writer.Close();
```

```
                Console.Read();//teknisk af hensyn til Windows
  }

}
```

Kommentar:

En Writer kan bl.a. opnås ved metoden CreateText(). En anden metode: File.AppendText() ville returnere en Writer som appender til en fil, dvs som skriver videre på en fil.

En Writer kan også oprettes således med metoden Create():

```
string fil=”gemt.txt”;
StreamWriter sw = new StreamWriter(File.Create(fil));
```

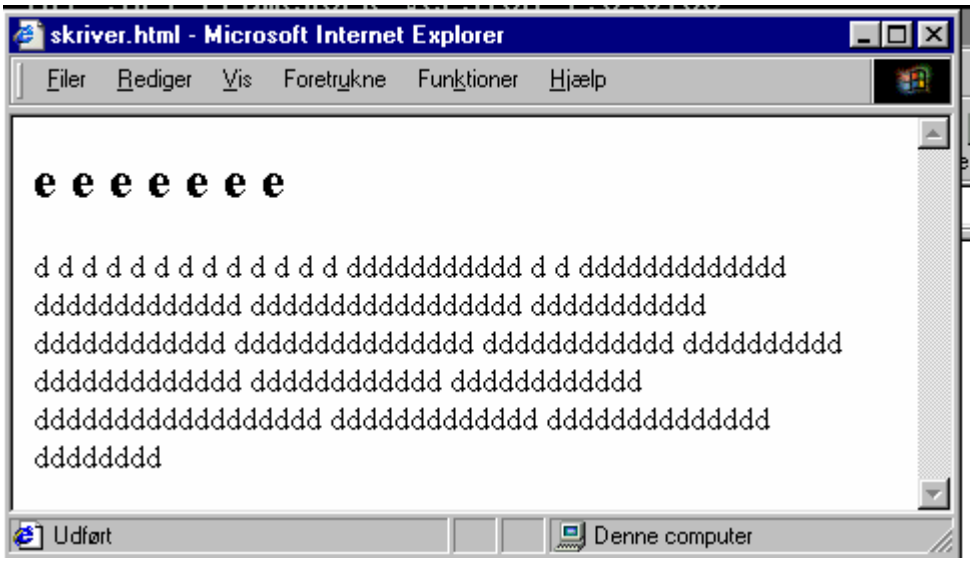
Det er væsentligt at se at metoderne WriteLine() osv egentligt er de samme metoder som anvendes på Console og på en Writer!! At skrive i C# kan være til en fil, til særmen, til en RAM adresse eller til en socket forbindelse på et netværk: Processen Write() er den samme!

Programmet opretter en primitiv HTML fil med de data som brugeren indtaster. Filen kan åbnes i Internet Explorer fx Prøv at chekke den HTML kode som findes i HTML filen! Parameteren til programmet bestemmer også hvor filen gemmes:

skrivfiler c:\csharp\html\eksempler\skriv.html

Denne kommando opretter en fil **skriv**.**html** i **mappen** c:\csharp\html\eksempler (**hvis** denne mappe ellers eksisterer!). Hvis der ikke angives nogen sti gemmes filen i nuværende bibliotek.

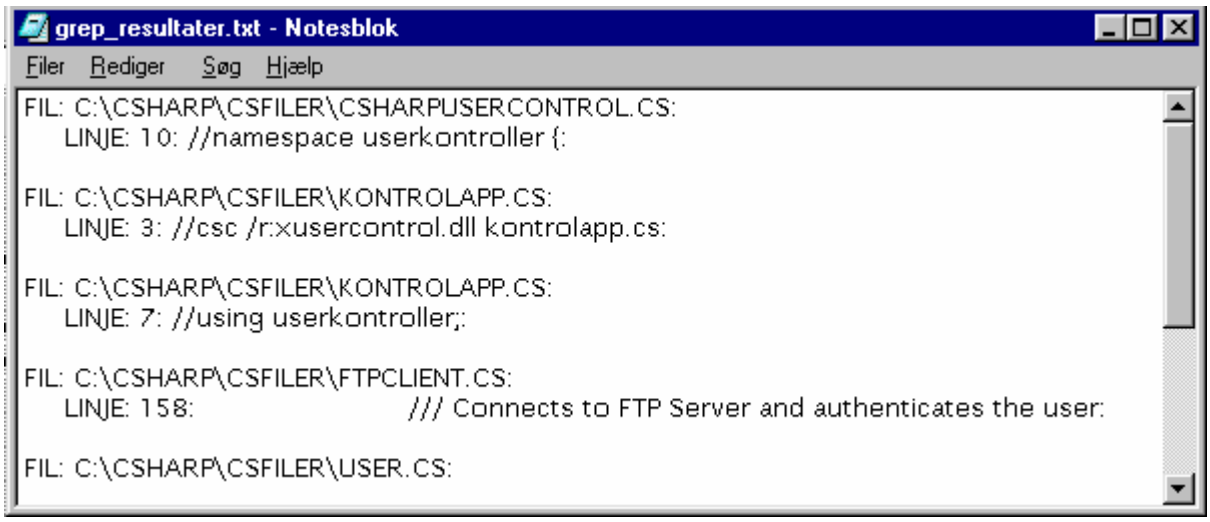
En kørsel af programmet **skrivfiler** - med parameteren **skriver**.**html** og en tåbelig mængde tegn som overskrift og tekst - giver dette resultat (NB mængden af tegn i afsnittet er det maximale antal tegn i en ReadLine()):



GREP – søg i filer:

GREP – Global Regular Expression Print – blev i sin tid opfundet i UNIX. GREP kan søge i en eller et antal filer efter et bestemt udtryk. På http://csharpkursus.subnet.dk ligger et grep program skrevet i C# med anvendelse af klasserne i System.Text.RegularExpressions. Programmer af denne art er meget nyttige når man har glemt hvor man egentligt har brugt en bestemt metode eller klasse!

Programmet producerer en resultat tekst. Her er søgt på 'user' i et antal filer:



Som det ses finder programmet også de steder hvor 'user' indgår som en del af et ord. Programmet udskriver filens navn og sti og linjenummeret hvor en 'Match' er fundet.

På grund af indkapslingen i C# er koden rimeligt simpel – her er den centrale kode i fragment:

```
                while(linje!=null){
                        Match match=regex.Match(linje);
                        if(match.Success){
                                writer.WriteLine("FIL: {0}:",
fil.FullName.ToUpper());
                                writer.WriteLine("    LINJE: {0}: {1}:", linjenummer,
linje);
                                writer.WriteLine();

                        }
                        linje=reader.ReadLine();
                        linjenummer++;
                }
```

Her er nogle få eksempler på hvordan GREP kan bruges:

Udtryk	**Betydning**
grep "Indtast side 1"	find de filer hvor strengen optræder
grep "user|User"	hvor enten 'user' eller 'User' optræder
grep "^int"	find alle linjer der starter med 'int' !

grep ”{0:*}”	linjer hvor format koden er anvendt - * betyder at hvad som helst kan følge efter kolon

Regulære udtryk kan bruges til søg og **erstat**. Et lille eksempel på det ville være følgende:

```
String input = "ted wanted the other ted to do it";
String pattern = @"\b(ted)\b";
String replace = "Ted";
Regex regex = new Regex(pattern);
String newString = regex.Replace(input, replace);
Console.WriteLine(newString);
```

Koden \b betegner at ‘ted’ skal forekomme som et ord med mellemrum før og efter!

I følgende eksempel søges efter et ord som starter med ’k’ eller ’K’ og derefter indeholder mindst 1 tegn:

String pattern = @"\b([kK]\S+)";
String str = "A kangaroo kicked a can to Kenya";

Dette giver altså 3 matches!

Strengen:

string pattern = @"\b(\S+)://(\S+)\b";

søger efter et afgrænset ord som består af et eller flere tegn (ikke whitespace), et kolon, to // og et eller flere tegn!

I denne streng ville der så være 2 matches:

```
String str = "Information about other wonderful "+
    "Wrox books can be found at http://www.wrox.com. "+
    "Or for you spotted toad aficionados, look to "+
    "https://www.spottedtoad.net for wonderful "+
    "photos of these magnificent creatures";
```

Følgende kode udnytter så en del af de muligheder som findes i RegEx og i klasserne MatchCollection og CaptureCollection:

```
MatchCollection matches = Regex.Matches(str,pattern);
Console.WriteLine("number of matches is {0}",matches.Count);
IEnumerator matchEnum = matches.GetEnumerator();
while ( matchEnum.MoveNext() )
{
  Console.WriteLine(matchEnum.Current);
}
Console.WriteLine();
for (int i=0; i<matches.Count; ++i)
{
```

```
    GroupCollection groups = matches[i].Groups;
    Console.WriteLine("In match {0}, "+
        "there are {1} groups",i,groups.Count);
    IEnumerator groupEnum = groups.GetEnumerator();
    while ( groupEnum.MoveNext() )
    {
      Console.WriteLine(groupEnum.Current);
    }
    Console.WriteLine();
    for (int j=0; j<groups.Count; ++j)
    {

      CaptureCollection captures = groups[j].Captures;
      Console.WriteLine("Group {0}, "+
        "has {1} capture",j,captures.Count);
      for (int k=0; k<captures.Count; ++k)
      {
        Console.WriteLine(captures[k].Value);
      }
      Console.WriteLine();
    }
```

Bruge 'gammel' Windows kode:

Man kan fra et C# program kalde EXE filer i det underliggende operativ system. Et C# program kan altså starte en browser eller et e-mail program eller en tekstbehandling!

Følgende kode viser et eksempel på dette:

```
// Eksempel: starter et Windows program
// Anvender klassen Process i System.Diagnostics:
//Fx: startprogram http://localhost
//starter Internet Explorer med den givne adresse
//'Verdens korteste internet program'!

//startprogram c:/windows/notepad.exe starter Notesblok


using System;


class StartProgram{

        public static void Main(string[] args){

                //Hvis ingen kommandolinje argumenter:
                string s=null;
                if(args.Length<1)s="c:/windows/notepad.exe";
                else s=args[0];

                System.Diagnostics.Process.Start(s);
        }
```

}

Som det ses er linjen

System.Diagnostics.Process.Start(s);

den eneste, der betyder noget væsentligt. Denne linje kan altså let lægges ind i et andet program og mulighederne kan udnyttes.

Metoden Start() kan også kaldes med 2 argumenter fx således:

startprogram c:/windows/notepad.exe startprogram.cs

I dette tilfælde åbner Notesblok med filen startprogram.cs!

En **browser** startes dog altid med: startprogram <...internetadresse...>
fx startprogram **http://www.dr.dk**

En alternativ – lidt mere fuldstændig kode – kan skrives således:

```
System.Diagnostics.Process proc = new System.Diagnostics.Process();
proc.EnableRaisingEvents=false;
proc.StartInfo.FileName="calc";
proc.Start();
```

Microsoft Word kan – også - åbnes med en bestemt fil således:

```
System.Diagnostics.Process proc = new System.Diagnostics.Process();
proc.EnableRaisingEvents=false;
proc.StartInfo.FileName="winword";
proc.StartInfo.Arguments="C:\\Dotnetstuff\\TestWordDoc.doc";
proc.Start();
```

En **Process** er populært sagt et program, som afvikles af computeren og som har sin egen del af RAM (med en stack, en kode og data sektion):

Opgaver:

1. Skriv et program med en menu hvor brugeren kan vælge mellem at starte forskelliige programmer (konsol og Windows programmer)!
2. Hvordan mener du at disse muligheder i øvrigt kan udnyttes – skriv et eksempel herpå!

Debugging eller at finde de logiske fejl:

Udtrykket **'debugging'** kommer af at en computer engang i 1950'erne brød sammen fordi en myg ('bug' betyder insekt) havde sat sig fast i computerens hardware. En glad amerikaner blev engang millionær i 'Hvem vil være millionær?' på at kende dette svar!

Nu bruges debugging om metoder til at finde logiske fejl i computerens software/programmer.

Debugging kan forekomme i 3 slags situationer:

1. mens programmet skrives og udvikles
2. når det siden testes
3. efter at programmet er 'færdigt' og leveret til en kunde

I C# kan debugging ske ved at anvende 'pre processor **directives'**. I første omgang vil vi koncentrere os om de to første situationer.

Ved at indsætte sådanne direktiver i kodeteksten kan man skabe to versioner af programmet: en 'debug version' (som bruges af udvikleren til at chekke programmet) og en 'release' som er det program brugeren skal have. Men ved at indsætte disse direktiver kan man altså slippe for at skrive programmet i 2 versioner.

Dette system har rigtigt mange muligheder men vi vil se på et konkret eksempel nemlig et program som **sorterer** en række ord som indtastes som kommando linje parametre til programmet:

sorter google cigar abe

Sortering foregår i **løkker. L**økker er typiske steder, hvor det kan være svært at gennemskue hvad programmet egentligt udfører! De 'logiske' fejl (som compileren jo ikke kan se) opstår meget tit i løkker.

Med debugging kan man få vist hvad der foregår rent faktisk:

```
C:\csharp\sorter>sorter google cigar abe
Ser nu paa Argument nr: 0 og Argument nr: 1
  Bytter Argument nr: 0 'google' med Argument nr: 1 'cigar'
  Argument 'cigar' rykkes op
Ser nu paa Argument nr: 0 og Argument nr: 2
  Bytter Argument nr: 0 'cigar' med Argument nr: 2 'abe'
  Argument 'abe' rykkes op
    Listen lige nu: abe google cigar
Ser nu paa Argument nr: 1 og Argument nr: 2
  Bytter Argument nr: 1 'google' med Argument nr: 2 'cigar'
  Argument 'cigar' rykkes op
    Listen lige nu: abe cigar google
Den sorterede tabel:
0: abe
1: cigar
2: google
```

Ovenstående viser udskriften, hvis programmet køres med debugging. Det første ord som brugeren indtaster er argument 0, det næste argument 1 osv. Programmet kører altså ved først at tage

argument 0 og **sammenligne** det med resten. **Hvis** en af disse er mindre end argument 0 byttes der. Derefter fortsættes med argument 1 og det sammenlignes med resten og der byttes hvis det er nødvendigt.
Denne sorterings metode **'Bubble Sort'** er berømt, men **ikke** særlig **effektiv,** hvis der er tale om at sortere fx 900.000 ord!! Algoritmen BubbleSort kræver alt for mange bytninger og tager urimeligt lang tid. Men den virker OK.

(Bubble sort er en **'kvadratisk'** algoritme dvs: Hvis 4 ord skal sorteres kræver det **16** operationer, hvis 16 ord skal sorteres kræver det **256** operationer – dvs at hvis vi skal sortere 4 gange så mange ord kræver det 16 gange så mange operationer. Derfor går det helt 'galt' når vi skal sortere fx 991.234 ord!!).

Her følger så koden til, hvordan denne debugging er etableret:

```
//programmet sorterer en række ord indgivet som kommandolinjeparametre
//der vises eksempler på debugging og pre processor directives som #if og #define

//hvis TEST ikke defineres i selve koden skal programmet kompileres med csc /define:TEST sorter.cs

//#define TEST

using System;

class MainClass
{
		public static void Main(string[] args)
		{

				for(int i=0;i<args.Length-1;i++){
						for(int j=i+1;j<args.Length;j++){

						//kun hvis TEST er defineret udføres dette:
						#if TEST
						Console.WriteLine("Ser nu paa Argument nr: {0} og Argument nr:
						{1}",i,j);
						#endif

						if(args[i].CompareTo(args[j])>0){
						#if TEST
						Console.WriteLine(@"  Bytter Argument nr: {0} '{1}' med Argument
						nr: {2} '{3}'",i,args[i],j,args[j]);
						Console.WriteLine(@"  Argument '{0}' rykkes op",args[j]);
						#endif

						string temp=args[i];
						args[i]=args[j];
						args[j]=temp;

						}

						}
						#if TEST
						Console.Write("    Listen lige nu: ");
						for(int ii=0;ii<args.Length;ii++){
						Console.Write(args[ii]+" ");
```

```
                                        }
                                        Console.WriteLine("");
                                        #endif

                                }
                                Console.WriteLine("Den sorterede tabel:");

                                for(int i=0;i<args.Length;i++){
                                        Console.WriteLine("{0}: {1}",i,args[i]);
                                }
                        }
}
```

Forklaring:

Hvis programmet indledes med en #define TEST (her er det kommenteret ud), vil programmet køre med TEST 'defineret', dvs resultatet bliver som vi så ovenfor med alle debug kontrol oplysningerne.
Hvis programmet kører uden at TEST er 'defineret', vil det blot udskrive den sorterede tabel.
Det mest effektive er at kompilere programmet på en af to måder:

csc /define:TEST sorter.cs

eller:

csc sorter.cs

I det første tilfælde defineres så TEST, og kommandoen sorter viser alle debug koderne. I det sidste tilfælde kompileres programmet til en færdig 'release' udgave.

#if ... #endif fungerer lige som if i C#.

Der kan anvendes mange forskellige define udtryk samtidigt og der kan anvendes en række forskellige direktiver:

pre processor directive:	**Betydning**:
#warning	udskriver en advarsel (lige som csc kompileren)
#error	udskriver en fejl, stopper kompilering – kan bruges hvis 2 uforenelige /defines er sat i kommando linjen!
#if #else #elif #endif	som i C#
#region #endregion	markerer en blok kode som kan klappes sammen med + - knapper
#undef X	fjerner X som defineret værdi i resten af koden

Define sætninger kan bruges til konfiguration:

Define formlen i C# kan bruges til at opsætte alle mulige konfigurationer af et program. Også using sætninger kan gøres betingede! Ligesom visse metoder kan gøres betingede af, at en værdi er defineret!

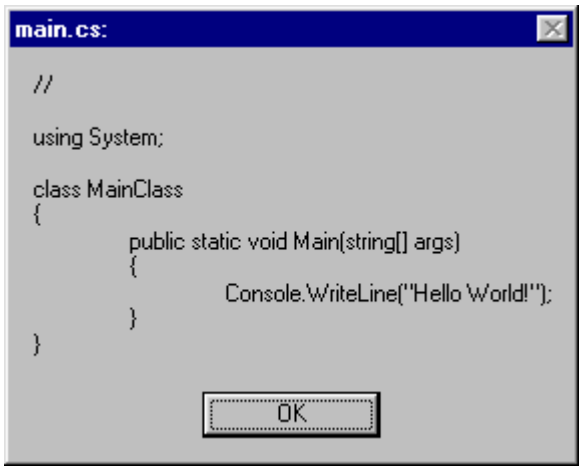
Her følger et meget simpelt eksempel på et 'to-i-een' program:

```
// eksempel paa brug af #define: to udgaver af samme program!

#if WIN
using System.Windows.Forms;
#endif
using System;
using System.IO;

class X {

		public static void Main(){
				StreamReader r=File.OpenText("main.cs");
				string s=r.ReadToEnd();
				#if WIN
				MessageBox.Show(s,"main.cs:");
				#else
				Console.WriteLine(s);
				#endif

		}
}
```

Hvis programmet kompileres med csc program.cs fås dette resultat:

```
//

using System;

class MainClass
{
        public static void Main(string[] args)
        {
                Console.WriteLine("Hello World!");
        }
}
```

Hvis det kompileres med: csc /define:WIN (eller /d:WIN) program.cs fås dette resultat:



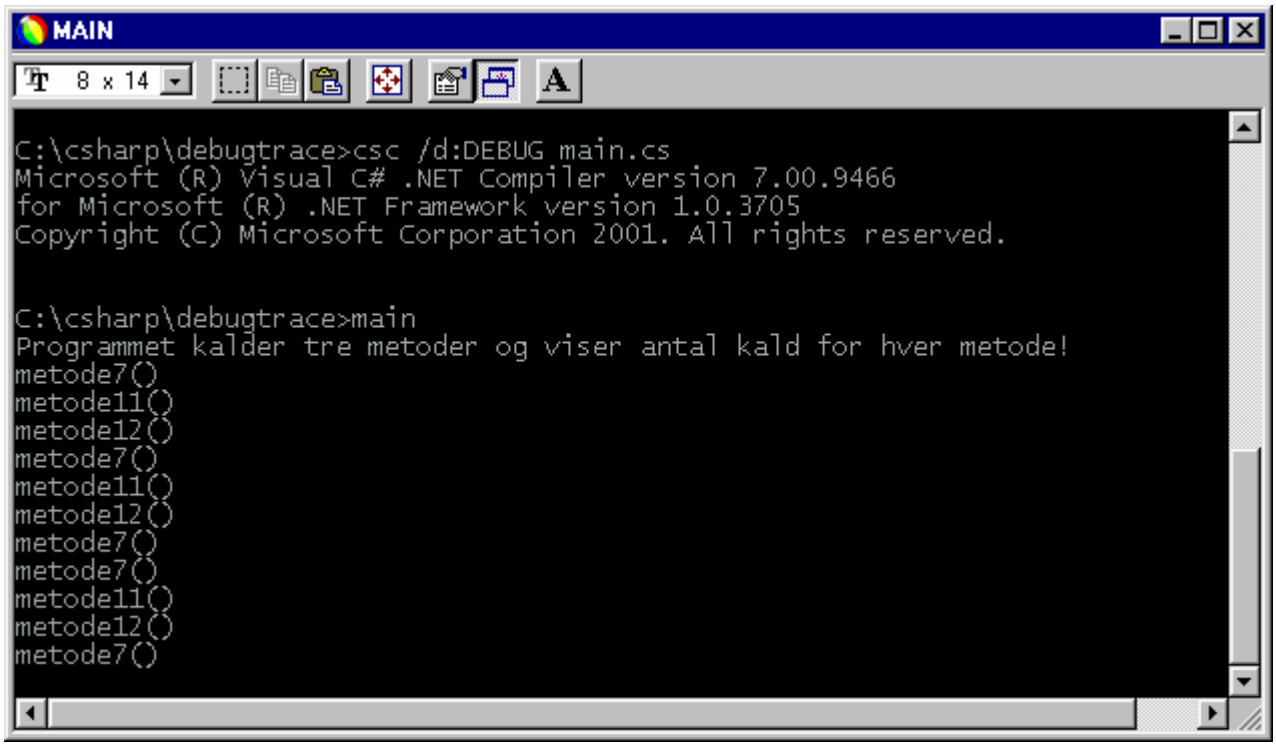
Debugging med klasserne i System.Diagnostics:

I System.Diagnostics ligger en del klasser som kan anvendes til debugging. Følgende eksempel viser en anvendelse af klassen Debug. Programmet kalder simpelhen nogle metoder og debug oplysningerne viser hvilke metoder som nu kaldes:

```
// debug og trace eksempler:

//'DEBUG' med stort er et reserveret udtryk!:
//#def DEBUG

using System;
using System.Diagnostics;

class MainClass
{
		public static void Main(string[] args)
		{
				//i stedet for Console.Out kan saettes en file stream!:
				TextWriterTraceListener listener=new TextWriterTraceListener(Console.Out);
				Debug.Listeners.Add(listener);
				Console.WriteLine("Programmet kalder tre metoder og viser antal kald for hver
metode!");

				int x=31;
				for(int i=0;i<x;i++){
						if(i%7==0)metode7();
						if(i%11==0)metode11();
						if(i%12==0)metode12();

				}

				Console.Read();
		}
		private static void metode7(){
				Debug.WriteLine("metode7()");
		}
		private static void metode11(){
				Debug.WriteLine("metode11()");
		}
		private static void metode12(){
				Debug.WriteLine("metode12()");
		}

}
```

Hvis koden kompileres som csc program.cs fås dette resultat:

Hvis vi ønsker debug data vist skal vi kompilere: csc /d:DEBUG program.cs (NB **'DEBUG'** er en **standard** udtryk – ikke en værdi vi selv definerer - som altid aktiverer **klassen** Debug!):

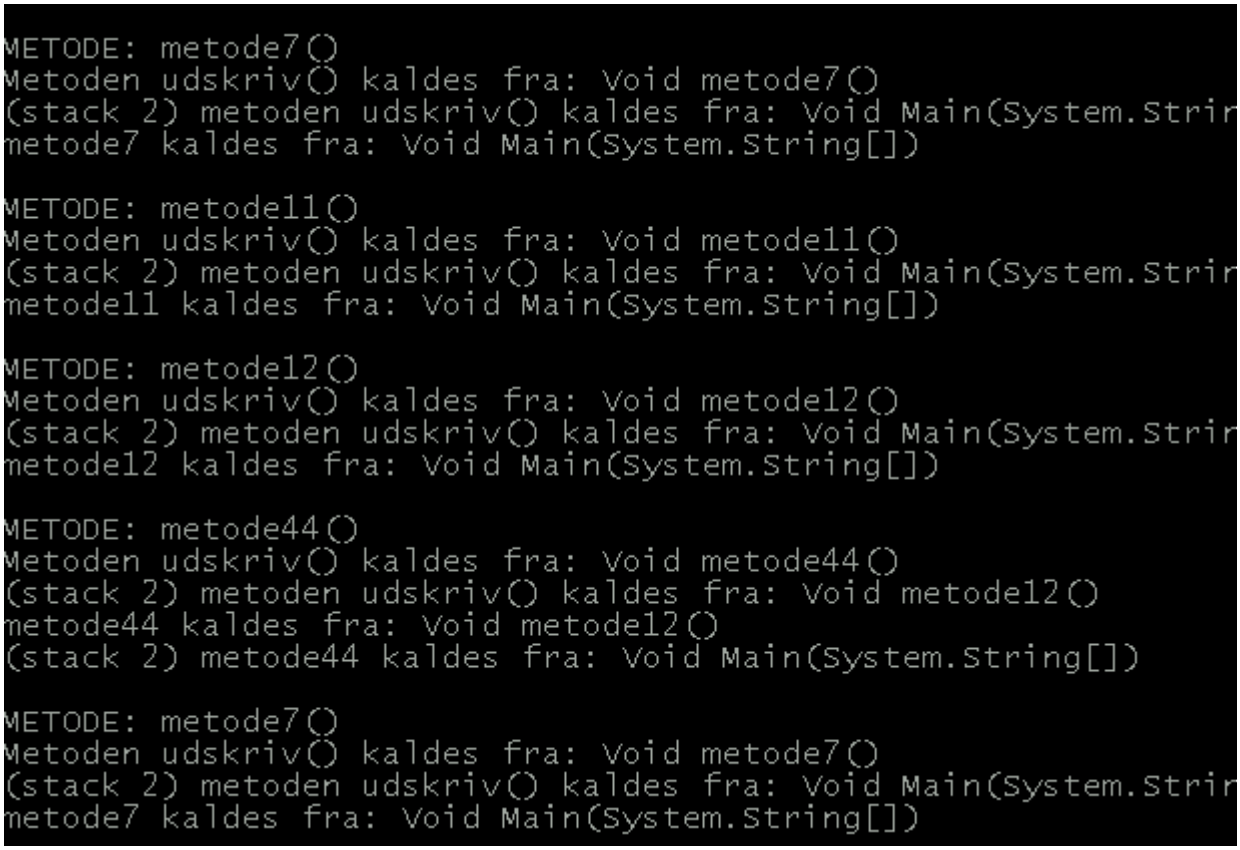


Som det ses har Debug klassen en samling af Listeners og vi kan tilføje nye Listeners hertil. En listener kunne også have skrevet til en FileStream i stedet. Console.Out er den stream som går til skærmen.

Metodens stack eller StackFrame:

Hver gang en metode kaldes åbner den med en ny stack. ved at undersøge metodens stack (i System.Diagnostics findes klassen StackFrame) kan vi se hvilken metode som har kaldt denne metode og så videre bagud!

Det viste program kan skrives om, så det viser, hvordan metoderne kalder hinanden – direkte og indirekte:



De forskellige debugger metoder kalder nu en ny metode udskriv(), og der er kommet en ny metode44() som kaldes af metode12() !

Linjen '(stack 2)' viser den metode som **indirekte** har kaldt metoden – nemlig den metode som ligger længere nede i stacken! 'stack 0' betegner den **nuværende** metode og 'stack 1' den metode, som **direkte** har kaldt denne metode!

Koden er også her kompileret med csc /d:DEBUG <program.cs>.

Koden til dette reviderede program ligger på http://csharpkursus.subnet.dk.

Attributten [Conditional()]:

DEBUG effekten kan også opnås ved at sætte en attribut over en metode, som opstarter debugging således:

```
		public static void Main(string[] args)
		{
			debug_init();
			Console.WriteLine("Programmet kalder tre metoder og viser antal kald for hver
metode!");

			int x=31;
			for(int i=0;i<x;i++){
				if(i%7==0)metode7();
```

```
                    if(i%11==0)metode11();
                    if(i%12==0)metode12();

                }

                Console.Read();
        }
        [Conditional("DEBUG")]
        private static void debug_init(){
                //i stedet for Console.Out kan saettes en file stream!:
                TextWriterTraceListener listener=new TextWriterTraceListener(Console.Out);
                Debug.Listeners.Add(listener);


        }
```

Her har vi flyttet initialiseringen/opstarten af Debug klassen til en metode og sat en [**Conditional**] attribut foran metoden. Dette bevirker, at **kun** hvis der er defineret en DEBUG vil metoden blive udført – programmet skal altså kompileres med csc /d:DEBUG (og **ikke** /d:debug!!) program.cs eller koden skal indledes med en #def DEBUG.

Løbende kontrol: Debug.Assert():

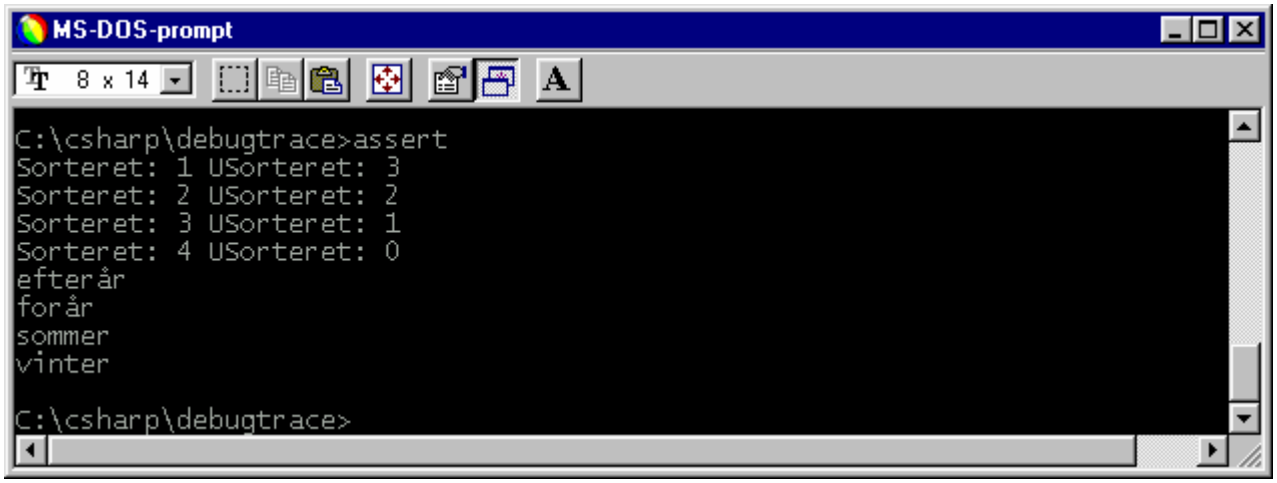
Et program eller en metode (en **algoritme**) kan **verificeres** ved at man på passende steder indlægger et **checkpoint** hvor man kontrollerer at en bestemt variabel har den korrekte værdi. Dette har en helt afgørende betydning i forbindelse med at bevise at algoritmer er **korrekte**. Til dette formål kan bruges en Debug.Assert(), som tager to parametre: Et boolsk testudtryk som – helst - skal være sandt og en besked som systemet viser i en messagebox, hvis testbetingelsen viser sig at være falsk!

Nedenstående eksempel bruger igen en simpel **sorteringsmekanisme**. Vi starter med 4 ord og sorterer forfra. Når vi har fundet det 'mindste' ord, flytter vi det over i en sorteret liste. Hver gang vi flytter et ord over i den sorterede liste, skal **summen** af elementerne i den sorterede liste og i rest-listen (de usorterede ord) være konstant – nemlig 4 hele tiden. Hvis det ikke er tilfældet, er algoritmen garanteret forkert.

```
// Debug.Assert() eksempel:

using System;
using System.Diagnostics;
using System.Collections;

class Assert {

        public static void Main(){
                string[] ord={"sommer","vinter","forår","efterår"};
                ArrayList sorteret=new ArrayList();
                for(int i=0;i<ord.Length;i++){
                        for(int j=0;j<ord.Length;j++){
                                if((String.Compare(ord[j],ord[i])>0)){
                                        string temp=ord[i];
                                        ord[i]=ord[j];
```

```
                                                ord[j]=temp;
                                        }
                                }
                                sorteret.Add(ord[i]);
                                int tilbage=ord.Length-i;

                                //foerste gang kontrolleres det at 0+4 er 4!:
                                Debug.Assert(tilbage+i==ord.Length,"Summen af de to grupper
(Sorteret og Usorteret) er forkert!");
                                Console.WriteLine("Sorteret: {0} USorteret:
{1}",sorteret.Count,tilbage-1);

                        }
                        for(int i=0;i<ord.Length;i++){
                                Console.WriteLine(ord[i]);
                        }



        }
}
```

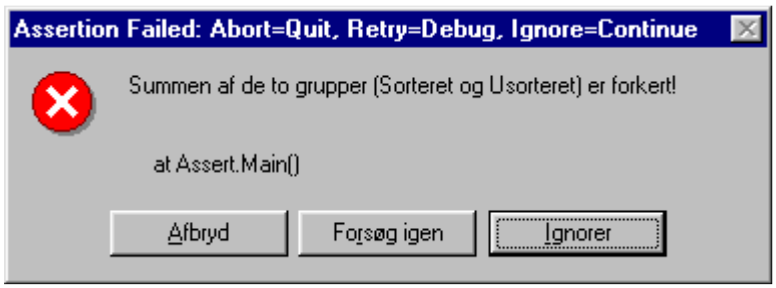
Koden **skal** kompileres med en csc /define:DEBUG program.cs.



Hvis vi laver en bevidst **fejl** i koden:

```
                                int tilbage=ord.Length-i-1;
```

Får vi adskillige messagebokse, hver gang Assertion.Fail() forekommer:



Debugging og tracing efter release:

Som nævnt er det muligt i C# at producere programmer som kan debugges efter at de er afleveret færdige – altså **uden** at programmet kompileres igen. Men filen SKAL oprindeligt være kompileret med /d:TRACE! På denne måde kan man finde bugs eller fejl efter at programmet er leveret.

Debug eller Trace oplysninger kan så slås til eller fra i en XML config fil. Dette sker ved at der oprettes en **TraceSwitch**, som kan sættes til 4 forskellige værdier. Koden ser nu sådan ud idet Debug er afløst af brug af Trace klassen i System.Diagnostics:

```
// debug og trace eksempler: TRACE klassen:
//Oprindeligt SKAL filen kompileres med csc /d:TRACE program.cs
//Trace kan saa siden aktiveres/de-aktiveres i config fil!

using System;
using System.Diagnostics;

class MainClass
{
		//Denne switch konfigureres i en XML config fil:

		//Navnet "TRACE" bruges i config filen!:
		public static TraceSwitch trace=new TraceSwitch("TRACE","Trace demo");

		public static void Main(string[] args)
		{
				trace_init();
				Console.WriteLine("Programmet kalder tre metoder og viser antal kald for hver
metode!");

				int x=31;
				for(int i=0;i<x;i++){
					if(i%7==0)metode7();
					if(i%11==0)metode11();
					if(i%12==0)metode12();

				}

				Console.Read();
		}
		//opretter en trace listener til release program:
		private static void trace_init(){

				//i stedet for Console.Out kan saettes en file stream!:
				TextWriterTraceListener listener=new TextWriterTraceListener(Console.Out);
				Trace.Listeners.Add(listener);


		}
		private static void metode7(){

				//dvs hvis trace.TraceInfo ikke er lig 0:
				Trace.WriteLineIf(trace.TraceInfo,"Trace: metode7()");
```

```
        }
        private static void metode11(){
                Trace.WriteLineIf(trace.TraceInfo,"Trace: metode11()");

        }
        private static void metode12(){
                Trace.WriteLineIf(trace.TraceInfo,"Trace: metode12()");

        }

}
```

Sætningen:

```
Trace.WriteLineIf(trace.TraceInfo,"Trace: metode7()");
```

kontrollerer værdien af TraceInfo dvs den værdi, som er sat i XML filen program.exe.config (XML filen SKAL benævnes med exe navnet + 'config'!):

```
<configuration>
<system.diagnostics>
<switches>
<add name="TRACE" value="4"></add>
</switches>
</system.diagnostics>
</configuration>
```

Her er værdien sat til 4 – de mulige værdier er:

Værdi for TraceInfo	Virkning
0 TraceLevel.Off	Ingen data udskrives
1 TraceLevel.Error	Kun fejl udskrives
2 TraceLevel.Warning	Fejl og advarsler udskrives
3 TraceLevel.Info	Info mv
4 TraceLevel.Verbose	Alle data udskrives

Opgaver:

1. Hvad sker der hvis du indtaster **sorter 111 44 5** ? Hvorfor??
2. Revider programmet sorter, så det sorterer tal i stedet for ord.
3. Sæt debug og trace koder ind i nogle af de C# programmer du allerede har skrevet.
4. Prøv at skabe flere 'konfigurationer' af et program med #define sætninger!
5. Skriv en Debug.Assert() fx i nogle af dine tidligere programmer, der kontrollerer om noget er sandt!

Objekt Orienteret Programmering (OOP):

Struktureret Programmering og Design (SD) opstod omkring 1970. Objekt Orienteret programmering (OOP) opstod som et paradigme omkring 1990 og findes i sprog som Java, SmallTalk, C++ og Visual Basic.

Struktureret programmering er en **delmængde** af OOP paradigmet.

Et meget enkelt eksempel på OOP og erklæring af en klasse er (eksemplet bliver udbygget senere!):

//fil: person.cs

```
//primitivt eks på OOP
//kompileres som DLL: csc /t:library person.cs

//koden har ikke noget 'entry point' og er IKKE en 'applikation':

using System;



public class Person{
                private string fornavn, efternavn, telefon;
}
```

OOP har en række overordnede typiske træk:

Indkapsling:

En klasse skal fungere som en '**black box'**. Omverdenen skal kunne bruge klassens 'interface' dvs dens metoder, men ikke kunne gribe ind i de indre mekanismer i klassen ('data hiding', objektets interne tilstand er skjult set udefra).

I eksemplet ovenfor: en persons egenskaber er interne dvs 'private'. Den omgivende verden skal ikke kunne ændre en persons egenskaber direkte. I det anførte eksempel findes (endnu) ingen metoder – så foreløbigt kan fornavn, efternavn og telefon slet ikke hentes ud af klassen! Person klassen er en black box.

Klassen har 3 **datamedlemmer**. Et datamedlem er noget andet og mere end en variabel: Et datamedlem rummer **information** om klassens/objektets 'tilstand'. Et datamedlem er **medlem** af klassen og 2 objekter – 2 personer – har antageligt forskellige værdier i deres **tilstandsvariable** eller datamedlemmer!

Der kan sammenlignes med at køre bil: det kræves ikke at man har nogen som helst forstand på bilens mekanik for at kunne køre bilen. Hvordan bilen egentligt fungerer er sikkert helt uvist for de fleste. Og egentligt også uvæsentligt for billisten (men ikke for automekanikeren!). Formålet med indkapsling eller 'encapsulation' er altså også at gøre det nemmere for brugeren! Se fx dette kode eksempel:

StreamReader r=File.OpenText("abc.txt");

Hele den indviklede procedure med at åbne en fil og åbne en stream til denne fil så at der kan læses fra filen bliver **indkapslet** I C# klassen StreamReader.
Hvad StreamReader egentligt gør i detaljer – er 'irrelevant' blot klassen gør det den skal!
Indkapsling betyder også at klassen kan skrives om mht. den **indre** mekanik blot den stadig har det samme **interface** eller de samme metoder **udadtil**!

Arv:

Klasser skal kunne arve egenskaber fra hinanden lige som vi I sproget har mere almene begreber og mere specialiserede begreber. Fx er 'person' et alment begreb og 'mand' et mere specielt begreb. Men egenskaber i 'person' findes også i 'mand'. 'mand' har blot fået nogle nye egenskaber sammenlignet med basis 'person'.

I OOP tales om en basis-klasse som kan have mange sub-klasser.

I forhold til eksemplet kunne man altså erklære en ny klasse 'mand' som arver fra 'person' og overtager alle egenskaber fra 'person' men tilføjer nogle flere specielle.

Arv gør kodningen nemmere, mere overskuelig og muliggør **genbrug**.

Alle klasser arver i C# fra klassen System.**Object** – direkte eller indirekte. Ovenstående erklæring kunne altså lige så godt skrives:

public class person : Object{}

idet kolon i C# betyder 'arver fra'. De **metoder** som findes i Object kan altså bruges i ALLE klasser uanset om de arver direkte eller indirekte fra Object.

Metoder i System.Object:

Metode	**Eksempel**	**Betydning**
Equals()	If(objekt.Equals(objekt2))	Er de identiske?
GetHashCode()	int h=objekt1.GetHashCode()	Returnerer et heltal som er objektets nummer eller hashkode – bruges i hashtabeller som index
GetType()	Obj.GetType()	Returnerer obj's type fx string
ToString()	Console.WriteLine(Obj.ToString());	Alle objekter kan udskrives også farver, windows osv
MemberwiseClone()	Obj.MemberwiseClone()	Opretter en klon eller kopi af objektet

GUID:

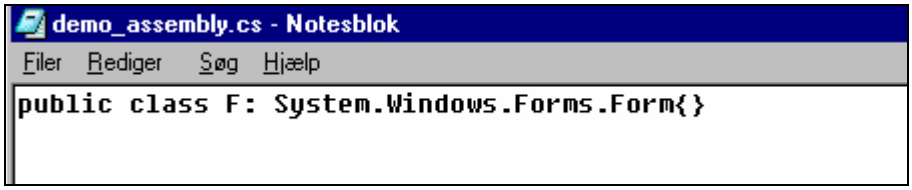
Alle objekter kan gives en unik hexadecimal kode, en **GUID,** således at det altid med sikkerhed kan afgøres om to objekter er 'ens' dvs om de referer til det samme objekt (samme adresse). Alle COM objekter har således en GUID kode. En GUID kan fx tildeles således:

```
public class MyObject
{
   private String name;
   private Guid guid;

  // NewGuid() generer et unikt GUID for hvert objekt:
  public MyObject(string str)
  {
    name = str;
    guid = Guid.NewGuid();
  }

  public Guid GetGuid() { return guid; }
  public String GetName() { return name; }
}
```

Et eksempel på en GUID kunne være:

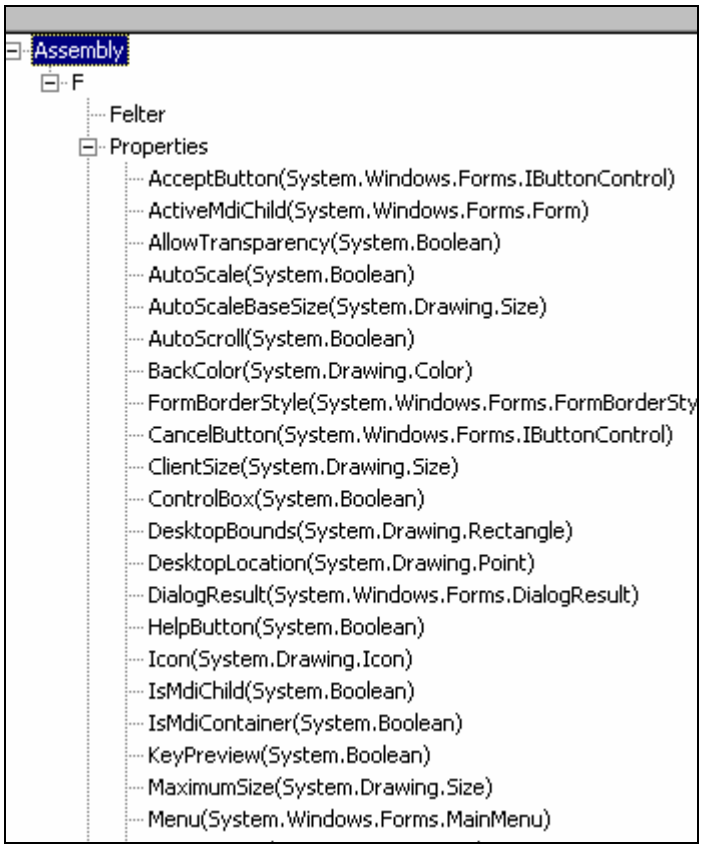
Lille eksempel på arv: En tom klasse som arver fra Form:

For at illustrere hvad arv betyder kan vi skrive følgende cs fil:



Vi opretter en ny klasse F som **arver** fra Form (som er den klasse i C# som definerer en Windows form eller et **vindue**). Men vores klasse er fuldstændigt **tom** og formel!

Denne fil kan kompileres til demo_assembly.**dll** og hvis vi undersøger denne binære fil med metoderne fra System.**Reflection** opdager vi til vores store overraskelse følgende:



Vi opdager nu at vores egentlig **'tomme'** klasse F blandt andet indeholder:

1. 105 properties !
2. 395 metoder !
3. 71 forskellige slags events !!

Klassen F har både en BackColor, en FormBorderStyle og en AcceptButton – **selv** om vi **ikke** har skrevet een eneste linje kode selv!

Alle disse egenskaber overtager vores egen klasse F fra basis klassen System.Windows.Forms.Form. (F får dem så at sige 'gratis'!). Den enorme styrke i arv kan ses her i dette enkle eksempel!

Alt dette handler om **genbrug** og om **IKKE** en gang til at 'opfinde den dybe tallerken'!

Såkaldt 'har en' arv (eller komposition):

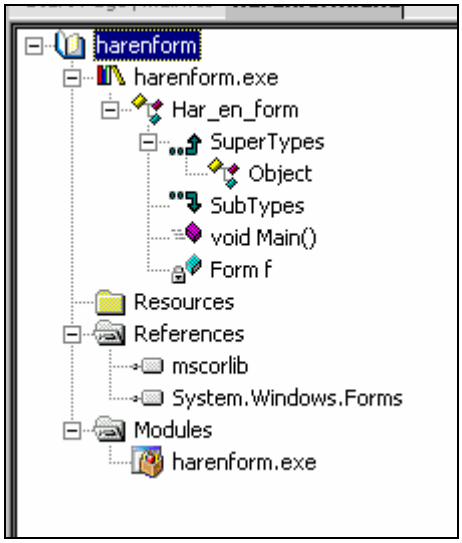
Der tales også om en anden slags arv – en **'har en'** defineret arv som **ikke** er arv i egentlig forstand. Dette betegner at en klasse har en anden klasse som et medlem. Dette kaldes så **'komposition'** dvs at klassen er **sammensat** af eller har en komponent som datamedlem.

Fx kan vi bruge dette eksempel:

```
using System;
using System.Windows.Forms;

public class Har_en_form : Object {
		private static Form f;

		public static void Main(){
				f=new Form();
				f.ShowDialog();
		}
}
```

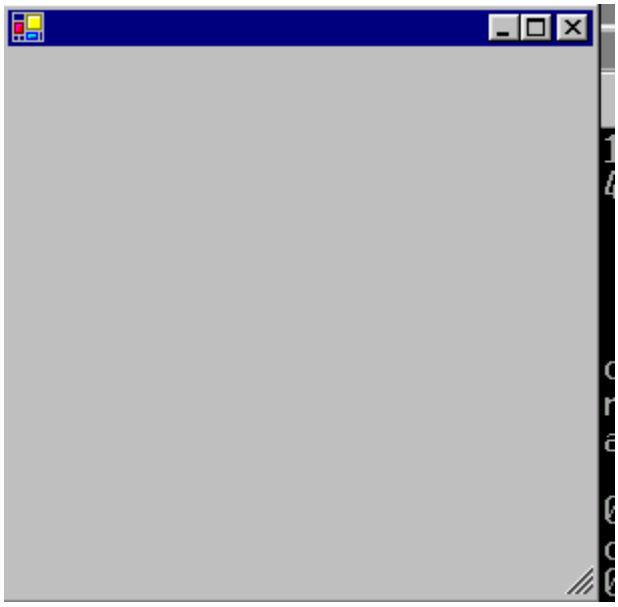
Den nye klasse har en **'har en'** relation til Form, fordi den har en Form som et datamedlem! Men den nye klasse **er ikke** nogen Form - dvs den arver **ikke** fra Form. Den arver fra System.Object!

Dette kan ses hvis vi åbner klassen i f.eks. SharpDevelop:



Som det ses har klassen kun een metode (Main()) og eet datamedlem (f) – **intet** er arvet fra Form!!

Alligevel er resultatet af at køre programmet stort det samme som hvis vi have arvet fra Form:



Polymorfisme:

Polymorfisme (’mange former’) beskriver at den samme metode – eller det samme metode navn - kan anvendes på mange forskellige objekter som stammer fra en fælles ‘forælder’.

Hvis man antager en klasse ’hus’, kan man oprette subklasser som ’byhus’, ’landhus’ og ’fritidshus’. Polymorfisme betyder så at den samme metode fx ’byg_huset()’ kan anvendes på alle objekter af (eksempler på) et ’hus’.

Eller: en metode ’vis_person()’ som kan anvendes på alle sub-klasser af ’person’ (fx ’mand’, ’kvinde’). Vi vil vende tilbage til polymorfisme mange gange i det følgende.

Genbrug:

Kode som er skrevet en gang skal **ikke** gentages, men gemmes i en klasse, som siden kan **genbruges** mange gange. Basis klasserne i C# som fx i System.Windows.Forms.dll er et eksempel herpå.

Programmeringen bliver på denne måde **’inkremental’** dvs den bygger hele tiden videre på det som er produceret i forvejen af mig selv eller – som regel – af andre. (Prøv i din egen kode hele tiden at genbruge kode, som du allerede **har** skrevet – det gør det hele **meget** nemmere!).

Dette princip kaldes også **komponent baseret** eller **package baseret** programmering.

Objekter og klasser:

En klasse i OOP er et skema eller en ’opskrift’ på hvordan objekter kan oprettes. (Klasse erklæringen er ’kageformen’ og ’kagen’ er objektet). En klasse **eksisterer ikke** nogen steder i

RAM, men et objekt har en adresse i RAM når programmet kører. Objektet 'eksisterer'. Det er oprettet med en **'new'**:

Jvf et tidligere eksempel:

Random rnd=**new** Random();
Random ny_rnd=**new** Random();

De 2 sætninger opretter to objekter af typen Random. Operativ systemet allokerer (sørger for) plads i RAM hukommelsen til hvert objekt og det findes på en bestemt adresse. De to objekter har samme struktur men kan have helt forskellige værdier i deres datamedlemmer (forskellige 'tilstands værdier').

Lige som to **objekter** af klassen 'person' antageligt har forskellige fornavn, efternavn og telefonnumre!

En **klasse** er derimod altid ens og den samme!

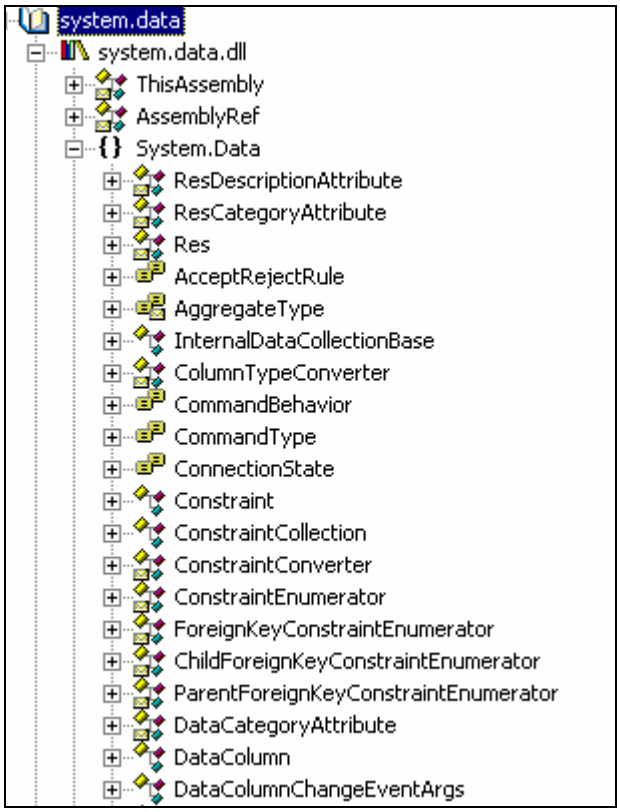
SharpDevelop og C# basis klasserne:

Det er besværligt at finde rundt i alle de klasser osv som C# er født med – dvs som .NET er født med (idet C# ikke har sine egne basis klasser).

Som tidligere nævnt bliver arbejdet nemmere med et IDE som SharpDevelop som anvender Reflection som tidligere omtalt. Du kan altid få information om en bestemt klasse ved at gøre sådan:

Opret et nyt projekt i SharpDevelop ved at vælge File->New Combine->C# og f.eks. et Windows Form Project.
Vælg Show->Projects og Projekt vinduet vises.
Højre klik References, vælg Add Reference og tilføj en DLL (en af fx System.Drawing) til projektet.
Du kan nu dobbeltklikke på den nye reference i Projekt vinduet og få indholdet af DLL filen vist i et nyt vindue.
Du kan også højre klikke på den nye reference og vælge Open.

F.eks.:



Arrays og operatorer på objekter:

Objekter kan samles i et array eller tabel helt som et **array** af heltal eller strenge. Hvis vi har en klasse **Rektangel** kan vi altså oprette en samling af rektangler således:

```
//opret en tabel med 10 pladser til 10 rektangler:
Rektangel[] objekter=new Rektangel[10];
```

Disse rektangler kaldes så med **index** operatoren: fx er det første rektangel lig med **objekter[0]**.

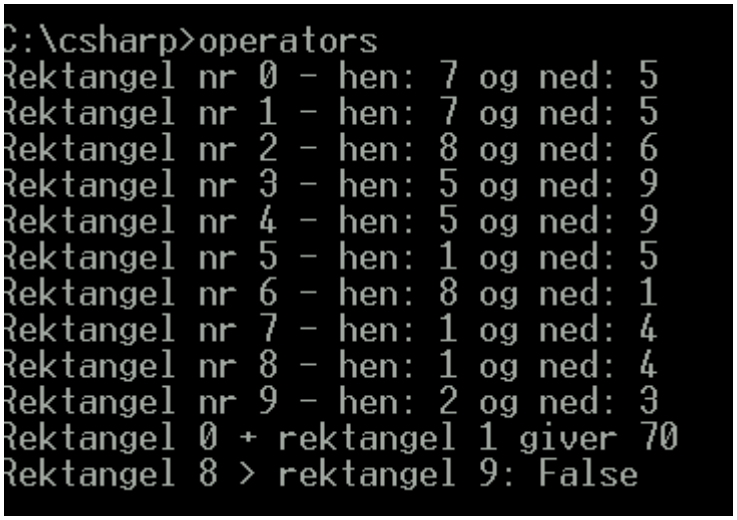
Men det er også muligt at sammenligne, sortere og manipulere rektangler med de almindelige operatorer som +, -, *, < og >.

I C# (og i C++) kaldes dette for **operator overloading**: den almindelige operator fx + bliver **overloaded** - dvs omdefineret til at kunne bruges i en ny sammenhæng.
Fx giver det mening at tale om at et rektangel er mindre end et andet hvis deres areal sammenlignes.

Følgende kode eksempel belyser dette:

```
//fil:operators.cs

//viser operator overloading på et object - et rektangel
//de almindelige operatorer +, -, *, /, >, >=, <, == kan anvendes på objekter også!

using System;
```

```
public class Rektangel{

		//for at gøre koden lettere er disse variable public!
		public int hen, ned;

		//constructor til Rektangel tager de 2 sider:
		public Rektangel(int h, int n){
		hen=h;
		ned=n;
		}

		//+ operator 'overloades' dvs 2 rektangler kan adderes: r1+r2:
		public static int operator +(Rektangel nr1,Rektangel nr2){
		return (nr1.hen*nr1.ned)+(nr2.hen*nr2.ned);
		}

		//de følgende metoder returnerer true hvis det første objekt er mindre/større:
		//< operator 'overloades' dvs 2 rektangler kan sammenlignes: if(r1<r2):
		public static bool operator <(Rektangel nr1,Rektangel nr2){
		return (nr1.hen*nr1.ned)<(nr2.hen*nr2.ned);
		}
		//> operator 'overloades' dvs 2 rektangler kan sammenlignes: if(r1>r2):
		public static bool operator >(Rektangel nr1,Rektangel nr2){
		return (nr1.hen*nr1.ned)>(nr2.hen*nr2.ned);
		}


}


class app{
public static void Main(){

		//opret en tabel med 10 pladser til 10 rektangler:
		Rektangel[] objekter=new Rektangel[10];

		//skab tilfældige rektangler og vis dem:
		for(int i=0;i<10;i++){

		Random rnd=new Random();
		int x=(int)rnd.Next(1,11);
		int y=(int)rnd.Next(1,11);
		objekter[i]=new Rektangel(x,y);
		Console.WriteLine("Rektangel nr {0} - hen: {1} og ned: {2}",i,objekter[i].hen,objekter[i].ned);
		}

		//hvad er summen af de 2 rektangler?
		Console.WriteLine("Rektangel 0 + rektangel 1 giver {0}",objekter[0]+objekter[1]);

		//er det ene rektangel støre end det andet?
		Console.WriteLine("Rektangel 8 > rektangel 9: {0}",objekter[8]>objekter[9]);

}
}
```



I C# anvendes altid dette skema (skabelon) til at overloade en operator:

```
public static int operator +(Rektangel nr1,Rektangel nr2){
```

Metoden skal være **static**, den kan returnere 'hvad som helst' (bool, int, string eller et objekt fx et nyt Rektangel objekt!), ordet 'operator' skal angives foran operatoren som skal overloades.
Disse metoder har altså typisk det format som er angivet i kode eksemplet.

NB: det er **muligt fuldstændigt** at omdefinere de kendte operatorer i et C# program. Vi er vant til at 4+5 giver 9 men med operator overloading er det **muligt** at overloade + operatoren så at 4+5 f.eks. giver strengen "Bla bla bla!"!! Denne form for overloading er selvfølgelig **ikke** nogen speciel god ide!!

Som det ses er det rimeligt nemt at konstruerer overloadede metoder til +, <, > osv.
Vigtigt er at afgøre om disse operatorer giver **mening** på en bestemt klasse. Det giver ikke megen mening fx at lægge to personer sammen – i al fald ikke på computeren! Hvis vi har et Hus objekt kan hus1 og hus2 måske adderes ved at deres arealer adderes. Men dette skal vurderes i det konkrete tilfælde.
Operator overloading kan bruges til at **sortere** objekter.

Opgaver:

1. skriv kode til at 2 rektangler kan subtraheres således: r1 – r2
2. skriv kode til at + operatoren returnerer et nyt Rektangel hvis areal er summen af to rektangler
3. skriv en metode sorter() som sorterer et antal rektangler efter areal
4. skriv en metode sorter() som sorterer et antal rektangler efter den længste side
5. skriv en ny klasse Bil og find ud af hvordan man kan kode at bil1 **er mindre end** bil2! Du må her vælge en egenskab ved bilen som kan bruges som målestok

Et praktisk eksempel på OOP: Person og Adresse klassen:

Person klassen har en indre egenskab – personens adresse. Vi starter derfor med at oprette en ny klasse som skal bruges til at holde styr på adresser eller 'adresse objekter':

```
//fil:adresse.cs
//kompileres: csc /target:library adresse.cs (giver adresse.dll):
//klassen i sin egen fil af hensyn til genbrug:

using System;

        public class Adresse{

                //contructor:
                public Adresse(string g, string n, string p, string b){
                        gade=g;nr=n;postnr=p;by=b;
                }

                //metode, medl funktion, interface:
                public string vis_adresse(){
                return "\n"+gade+" "+nr+"\n"+postnr+" "+by;
                }

                //felt, datamedlem, instans var: NB private pga OOP indkapsling hiding:
                private string gade, nr, postnr, by;
        }
```

Kommentar til Adresse klassen:

Adresse klassen gemmes i sin egen fil adresse.cs – for at den kan genbruges af andre klasse filer eller applikationer.
Den er IKKE nogen 'applikation' – den har ikke nogen Main() metode og kan altså ikke køres som et program.
Den skal kompileres således:
csc /target:library adresse.cs
eller:
csc /t:library adresse.cs
Resultatet bliver adresse.dll som er en DLL fil akkurat som den tidligere nævnte System.Windows.Forms.dll der indeholder definitioner på Windows komponenter.

Klassen Adresse indeholder **private** variable (gade, gadenr osv) – et eksempel på 'data hiding' eller indkapsling. Adresse er en black box som man kun har adgang til via de **public** metoder som klassen tilbyder (dens **interface**).
Der er en vis svag tradition for at benævne klasser i C# med stort begyndelsesbogstav (som fx **A**dresse) – men et klassenavn med kun små bogstaver er helt OK (modsat Java). Klassen Adresse **behøver** heller ikke at blive gemt i en fil der hedder **Adresse**.cs (igen modsat Java!) – CS filen kan hedde hvad som helst og indeholde mange klasse erklæringer.

Klassen har en **constructor/konstruktør** som opretter et nyt adresse objekt ved at modtage 4 parametre.
I en applikation kan således skrives:

Adresse adr=new Adresse(”Byvej”,”22”,”2730”,”Herlev”);

En **constructor** er en speciel metode hvis navn ALTID er identisk med klassenavnet. Hvis klassen hedder **public class ABC** – hedder constructoren ALTID: **public ABC**() osv.
Sætningen:

ABC abc=new ABC();

kalder i virkeligheden en **metode** (jvf parenteserne der viser det er et **metode kald**!), som kreerer et nyt objekt af typen ABC.

Alle klasser er ’født’ med en default (standard) constructor uden parametre. Man kan selv skrive de ’konstruktører’, der er brug for med 0,1,2 eller flere parametre. Formålet med at have flere konstruktører til en klasse er at brugen af klassen bliver mere fleksibel.

En constructor **returnerer** ikke noget - modsat en almindelig metode som altid skal returnere noget - evt returnere void.

Klassen har også en metode:

```
public string vis_adresse(){
        return "\n"+gade+" "+nr+"\n"+postnr+" "+by;
        }
```

Metoden vis_adreese() tager ingen parametre, er **public** (den kan kaldes udefra på et adresse objekt) og returnerer en **string (**som er en formatteret udskrift af adressen).

En applikation (eller en anden klasse) ville kunne indeholde følgende linjer:

Adresse adr=new Adresse(”Byvej”,”22”,”2730”,”Herlev”);
string adresse=adr.vis_adresse();
Console.WriteLine(adresse);

Person klassen, som ’har en’ Adresse:

Med udgangspunkt i klassen Adresse oprettes en klasse Person som indeholder et Adresse objekt:

```
//fil: person.cs
//primitivt eks på OOP
//kompileres som DLL: csc /t:library /reference:adresse.dll person.cs

//koden har ikke noget 'entry point' og er IKKE en 'applikation':


using System;
```

```
public class Person{

		//constructor:
		public Person(string f,string e,string t,Adresse a){
				fornavn=f;efternavn=e;telefon=t;adresse=a;
		}

		//metode, medl funktion, interface

		public string vis_person(){

		//vis_person() kalder en anden funktion i en anden klasse: vis_adresse()
		//NB begge metoder returnerer en 'streng' el tekst:

		return fornavn+" "+efternavn+" "+telefon+adresse.vis_adresse();
		}

		//felt, datamedlem, instans var: NB private pga OOP indkapsling hiding:
		//NB disse felter er IKKE properties, men 'gammeldags' variable:

		private string fornavn,efternavn,telefon;

		//eks på 'containment' el 'har en' relation mellem to klasser:

		private Adresse adresse;
}
```

Kommentar til klassen Person:

Klassen Person minder en del om klassen Adresse. Også Person gemmes i sin egen fil for at kunne genbruges. Den kompileres på samme måde som Adresse men med tilføjelse af en reference til adresse.dll:
csc /t:library /reference:adresse.dll person.cs
hvorved dannes person.dll.
Prøv som et forsøg at kompilere filen som:
csc /t:library person.cs

Du vil nu få en compiler fejl, der betyder at kompileren ikke kender noget til klassen Adresse:

```
Person.cs(15,44): error CS0246: The type or namespace name 'Adresse' could not be found (are you missing a using directive or an assembly reference?)
Person.cs(33,11): error CS0246: The type or namespace name 'Adresse' could not be found (are you missing a using directive or an assembly reference?)
```

Klassen har en constructor således at en ny person kan oprettes således:

Person p1=new Person(”Erik”,”Hansen”, ”56564534”,new Adresse(”Storevej”,”44”,”2000”,”Frederiksberg”));

Klassen er et eksempel på 'containment' eller 'har en' relation mellem to klasser. En person 'har en' adresse. Klassen Person har en indre egenskab som hedder 'adresse' som er af typen Adresse! Klassen Adresse optræder som en del af klassen Person.
Eksemplet viser også hvordan klasser i C# kommer til at fungere som de indbyggede typer som findes (fx int, float, char). Man kan erklære en variabel ved:
Adresse a;
På samme måde som:
int x;
Vores klasser Person og Adresse er **typer** der kan bruges i koden som alle andre typer.
Person har en vis metode som minder meget om metoden i klassen Adresse. Det specielle er at metoden vis_person() kalder metoden vis_adresse()! Formålet med dette er bl.a. at undgå at skrive koden to gange – dvs udnytte de metoder/'interface' som i forvejen ligger i klassen Adresse.

En applikation som anvender klasserne Adresse og Person:

Følgende kode samler trådene til et program som kan køres.
Koden kan nu også kompileres som:

csc /out:personapp.exe person.cs adresse.cs personapp.cs

da alle filerne nu er tilgængelige. Formlen **/out:abc.exe** SKAL anvendes hvis flere cs filer kompileres samtidigt for at angive navnet på EXE filen som skal produceres.
(abc.exe er et eksempel – men husk at skrive .**exe** ellers bliver resultatet ikke nogen EXE fil!).
Koden opretter 2 person objekter som derefter udskrives:

```
//fil:personapp.cs

//en applikation som anvender klasserne Person og Adresse

//kompileres: csc /r:adresse.dll;person.dll personapp.cs

//eller: csc /out:personapp.exe adresse.cs personapp.cs Person.cs
//

using System;


public class app
{
		public static void Main(string[] args)
		{
				//instantiering og oprettelse af 2 obj:

				Person p1=new Person("Jens","Jensen","67675656",new
Adresse("Byvej","44","2730","Herlev"));
				Person p2=new Person("Lise","Jensen","45455656",new Adresse("De riges
Alle","99","2000","Frederiksberg"));

				Console.WriteLine("PERSON: {0}",p1.vis_person());
				Console.WriteLine("PERSON: {0}",p2.vis_person());

		}
}
```

Proporties i C# klasser:

Vi har hidtil erklæret en klasses 'egenskaber' som typisk private variable eller instans variable eller data medlemmer (mange ord for det samme) f.eks. sådan:

private string fornavn;

Dette betyder at fx klassen Person har en egenskab eller et felt som hedder 'fornavn'. Hvis man skal have fat i denne indre private egenskab skal der skrives en get og set metode: get_fornavn() og set_fornavn() ellers bliver 'fornavn' aldrig offentligt tilgængeligt.

Da det kan være lidt besvær nogle gange at skrive disse to metoder er tingene forenklet i C# så at der kan kodes med en **'Property'**. En property er en slags sluse ind til den **private** variabel. Samtidigt er en property nemmere at anvende end to metoder. Klassen får nu **både** en private variabel (felt, egenskab) 'fornavn' **og** en public property 'Fornavn' (det er en god ide at skelne og skrive en property med **stort** begyndelsesbogstav – selv om 'fornavn' og 'Fornavn' i dette eksempel egentligt er det samme – har det samme indhold).

Følgende er et eksempel på at anvende properties i stedet for metoder i en **ny** klasse kaldet **Hus**, som har fire egenskaber dvs properties:

```
//fil: hus.cs
//enkelt eksempel på properties i klasse
//i stedet for properties kan anvendes public metoder - se eksempel i koden:

//properties spiller en enorm rolle i komponenter i Windows programmeringen
//F.eks. har en Button en property der hedder Text
//derfor kan man i C# skrive: minbutton.Text="Klik";

using System;


public class Hus{

		//NB der er FORSKEL på disse private felter/egenskaber og deres
		//tilsvarende properties - derfor kaldes de IKKE det samme!

		private string id;
		private int areal;
		private int etager;
		private int bygget;//ex 1945
		public Hus(){}
		public void udskriv_data(){
				Console.WriteLine("Husets ID er: {0}",id);
				Console.WriteLine("Huset er bygget {0}, er {1} etager med areal {2}
kvm",bygget,etager,areal);
		}

		//en property - skrives altid med denne signatur og struktur:
		//NB klassens indre egenskaber er stadig private!
		//Der er blot lettere adgang til dem med en property:
		public string ID{
				get{return id;}
				set{id=value;}
		}
		//OBS: i stedet kunne skrives 2 metoder:
		//Forskellen er i simple tilfælde ikke så stor:

		//public string get_id(){return id;} og
		//public void set_id(string i){id=i;}

		public int Areal{
				get{return areal;}
				set{areal=value;}
		}
		public int Etager{
				get{return etager;}
				set{etager=value;}
		}
		public int Bygget{
				get{return bygget;}
				set{bygget=value;}
		}

}
class app
{
		public static void Main(string[] args)
		{
```

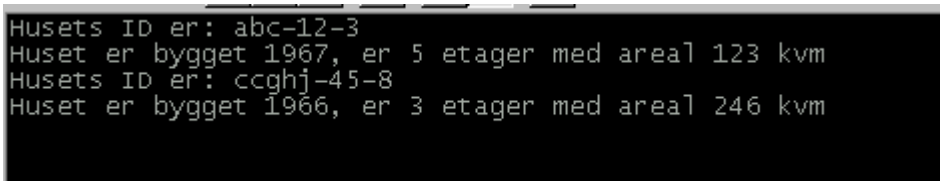
```
                    //det ses her at en property
                    //er nemmere og mere fleksibel end en metode:

                    Hus hus=new Hus();
                    hus.ID="abc-12-3";
                    hus.Areal=123;
                    hus.Etager=5;
                    hus.Bygget=1967;
                    hus.udskriv_data();

                    Hus hus1=new Hus();
                    hus1.ID="ccghj-45-8";
                    hus1.Areal=hus.Areal+123;
                    hus1.Etager=3;
                    hus1.Bygget=hus.Bygget-1;
                    hus1.udskriv_data();


                    Console.Read();//stopper vinduet!
            }
}
```

Programmet erklærer en klasse Hus og definerer 4 properties på den klasse.
En property erklæring ligner en blanding af en klasse erklæring og en metode erklæring:

```
public int Areal{
        get{return areal;}
        set{areal=value;}
}
```

Det reserverede ord 'value' anvendes ALTID på denne måde i en property. NB Der anvendes krøllede parenteser ikke almindelige parenteser! I en set{} kan skrives megen kode som kontrollerer eller databehandler den værdi (value) som propertyen sættes til. Get og set behøver altså slet ikke at være så simple som i disse eksempler!!
Det fordelagtige ved properties er at de er dejligt nemme at arbejde med:

```
hus1.ID="ccghj-45-8";
hus1.Areal=hus.Areal+123;
hus1.Etager=3;
hus1.Bygget=hus.Bygget-1;
```



Opgaver:

1. skriv en ny klasse Borger med felterne alder, navn, cpr.
2. skriv en property til feltet alder – i set delen skal du skrive kode som sætter alder til en rimelig værdi – hvis alder er sat til –22 eller 1234 skal koden reagere i set delen
3. skriv lignende properties til navn og CPR

4. skriv en set del i propertyen CPR som kontrollerer/reagerer på det som er tastet ind/ det som propertyen er sat til! Fx ved vi hvor mange tegn der er i CPR nummer!

Eksempler på Polymorfisme og Arv:

Formålet med at en klasse 'arver' fra en anden er at genbruge den funktionalitet som allerede ligger i en basis-klasse og at skabe hierarkier af klasser som er logiske og overskuelige. I C# kan er klasse kun arve fra en klasse, en klasse kan kun have en enkelt **base class**. (Men kan implementere mange **interfaces** som vi skal se). Men en basis klasse kan have mange efterkommere – nedarvede klasser eller 'børn'.
Arv erklæres meget nemt således i C#:

```
public class Rektangel: Figur{}
```

Kolon angiver altid arv – Rektangel arver fra Figur eller sagt i almindeligt sprog: 'Et rektangel **er** en figur!'.

Her er et meget enkelt eksempel på hvordan **arv** kan kodes i C#:

```
//fil: arv.cs

//illustrerer på enkel måde arv i C#
//af nemheds grunde er alle klasser her anbragt i samme fil arv.cs

//postcondition: outputter data om objekter

using System;

public class Figur{

                //protected og ej private da de skal bruges af arve-klasser:
                protected string navn;
                protected string id;

                public Figur(string n){navn=n;id="Figur";}

                //netode som nedarves:
                public void vis(){
                Console.WriteLine("Objektet er {0} og navnet er {1}",id,navn);
                }
}

public class Rektangel: Figur{
                //NB navn og id er nedarvede automatisk
                private int side1,side2;

                public Rektangel(string n,int s1,int s2):base(n){
                id="Rektangel";
                side1=s1;side2=s2;
                }
                //NB vis() er nedarvet automatisk
}
```

```
public class app{
  public static void Main(string[] args){

                    //instantier objekter af typerne
                    Figur f1=new Figur("Lise");
                    f1.vis();
                    Rektangel r1=new Rektangel("Thomas",200,300);
                    r1.vis();
                    Rektangel r2=new Rektangel("Mikkel",150,700);
                    r2.vis();

                    Console.Read();//teknisk af hensyn til Windows

  }

}
```

Hvis arv køres bliver resultatet:

```
C:\csharp>arv
Objektet er Figur og navnet er Lise
Objektet er Rektangel og navnet er Thomas
Objektet er Rektangel og navnet er Mikkel
```

Eksemplet viser at den nedarvede klasse **ikke behøver** at erklære metoden vis() eller de andre medlemmer (id, navn) som basis klassen har. Vi har altså dels gjort det nemmere for os selv og dels har vi skabt et logisk **hierarki**: Et Rektangel **ER** en Figur, dvs har alle de egenskaber som en Figur også har!
Metoden vis() kan altså uden videre bruges på klassen Rektangel, men det vi er interesseret i er snarere at vis() viser **alle** de egenskaber som et Rektangel har og ikke kun dem som er **fælles** med Figur!
Metoden i OOP og i C# er derfor at beholde metoden vis() fra basis klassen men at bygge videre på den – at **'override'** basis eller super klassens metode!

Et eksempel herpå:

```
//fil: arv.cs

//illustrerer på enkel måde arv i C#
//af nemheds grunde er alle klasser her anbragt i samme fil arv.cs

//postcondition: outputter data om objekter

using System;

public class Figur{

                    //protected og ej private da de skal bruges af arve-klasser:
                    protected string navn;
                    protected string id;

                    public Figur(string n){navn=n;id="Figur";}
```

```
                //netode som nedarves erklæres virtual så den kan overrides:
                public virtual void vis(){
                Console.WriteLine("Objektet er {0} og navnet er {1}",id,navn);
                }
}

public class Rektangel: Figur{
                //NB navn og id er nedarvede automatisk
                private int side1,side2;

                //constructor kalder base dvs kalder Figurs constructor:
                public Rektangel(string n,int s1,int s2) : base(n){
                id="Rektangel";
                side1=s1;side2=s2;
                }
                //NB vis() er nedarvet automatisk

                //NB der findes nu 2 metoder vis(): superklassens/basis' metode er = base.vis():
                //override angiver at vi omdefinerer en nedarvet metode:

                public override void vis(){
                base.vis();
                Console.WriteLine("Side 1 er {0} og Side 2 er {1} og arealet er {2}",side1,side2,side1*side2);
                }

}

public class app{
  public static void Main(string[] args){

                //instantier objekter af typerne
                Figur f1=new Figur("Lise");
                f1.vis();
                Rektangel r1=new Rektangel("Thomas",200,300);
                r1.vis();
                Rektangel r2=new Rektangel("Mikkel",150,700);
                r2.vis();

                Console.Read();//teknisk af hensyn til Windows

  }
}
```

De metoder i basis klassen som skal/kan overrides angives som: **public virtual…** og 'barne' klassen erklærer metoden som **public override** …
På denne måde beholdes de gode ting i basis klassen samtidigt med at den nedarvede klasse bygger videre.
Egenskaber i en basis klasse der skal nedarves skal være **public** eller **protected** – hvis de er private kan de ikke 'overtages' af en anden klasse - og hvis du prøver giver det en kompiler fejl!

Override metoder i System.Object:

Alle klasser arver i C# fra System.Object og metoden ToString() (som er erklæret virtual!) i Object kan fx overrides på denne måde:

```
//Demo af hvordan metoder i System.Object kan overrides:

using System;

public class Bil{
		private string navn;
		private byte alder;
		private int kilometer;

		public Bil(string n,byte a,int k){
				navn=n;alder=a;kilometer=k;
		}

		//OVERRIDE en metode i Object:
		public override string ToString(){
				return "Navn: "+navn+"\tAlder: "+alder+"\tKilometer: "+kilometer;
		}
}

class app
{
		public static void Main(string[] args)
		{
				Bil bil1=new Bil("Ford Fiesta",3,23000);
				Bil bil2=new Bil("Toyota Corolla",11,123000);

				//NB nu kan bilerne 'udskrives direkte' !!:
				Console.WriteLine(bil1);
				Console.WriteLine(bil2);

				Console.Read();

		}
}
```

Opgaver:

1. Skriv en ny klasse Trekant der arver fra Figur
2. skriv en metode vis() for klassen Trekant
3. Hvad bliver udskriften i eksemplet med Bil hvis ToString() IKKE overrides? Hvorfor?
4. En anden metode i Object som ofte overrides er: **Equals**() som kan overrides efter samme model som ToString():

```
public override bool Equals(object o){
		Bil temp=(Bil)o;
		return navn==temp.navn;
}
```

Skriv programmet om så det opretter nogle flere bil-objekter og tester om de er 'lig med' hinanden (du behøver jo ikke definere 'er lig med' som i dette eksempel!!). NB du vil få en

advarsel fra kompileren fordi du ikke også override metoden GetHashCode() – hvis du vil rette denne fejl kan du skrive endnu en override:

```
public override int GetHashCode(){
        return kilometer.GetHashCode();
}
```

Metoden GetHashCode() returnerer et heltal som er objektets 'nummer' eller hash kode (nummer kode) i RAM – metoden kan altså kaldes på hele bilen eller på fx dens navn eller et andet felt. Hash kode bruges i hashtabeller til at finde et index til et objekt.

Eksempel: Polymorfisme og dynamisk binding

Der vil nu blive præsenteret et lille eksempel på styrken i arv og **polymorfisme**:
For at give flere muligheder er der her oprettet en ny klasse Cirkel som arver fra Figur. Klassen cirkel overrider også metoden vis().
Desuden er skrevet en applikation som opretter et **array** af Figurer som vælges **tilfældigt**. Det mest interessante er at når disse tilfældige objekter er oprettet og indsat i tabellen kan systemet finde den rigtige metode (**run time** – dvs mens programmet afvikles) til det rigtige objekt. Den rigtige metode findes **dynamisk** ('dynamic binding') og ikke **statisk** (under kompileringen). Dette kaldes **polymorfisme**. Styrken i dette er også at det er muligt at oprette en tabel af Figurer som ikke blot kan rumme Figurer men også alle sub klasser af Figur!!

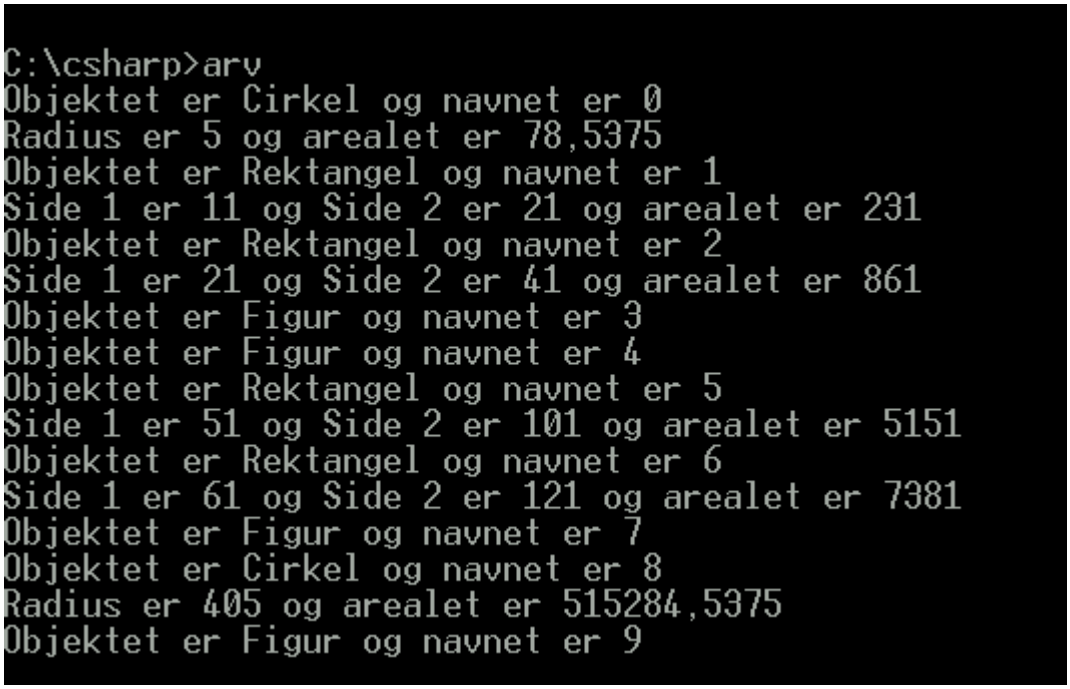
Kode eksempel på arv og polymorfisme:

```
//fil: arv.cs

//illustrerer på enkel måde arv og polymorfisme i C#
//af nemheds grunde er alle klasser her anbragt i samme fil arv.cs

//postcondition: outputter data om objekter

using System;

public class Figur{

        //protected og ej private da de skal bruges af arve-klasser:
        protected string navn;
        protected string id;

        public Figur(string n){navn=n;id="Figur";}

        //netode som nedarves erklæres virtual så den kan overrides:
        public virtual void vis(){
        Console.WriteLine("Objektet er {0} og navnet er {1}",id,navn);
        }
}

public class Rektangel: Figur{
        //NB navn og id er nedarvede automatisk
        private int side1,side2;

        //constructor kalder base dvs kalder Figurs constructor:
```

```
                public Rektangel(string n,int s1,int s2) : base(n){
                id="Rektangel";
                side1=s1;side2=s2;
                }
                //NB vis() er nedarvet automatisk

                //NB der findes nu 2 metoder vis(): superklassens/basis' metode er = base.vis():
                //override angiver at vi omdefinerer en nedarvet metode:

                public override void vis(){
                base.vis();
                Console.WriteLine("Side 1 er {0} og Side 2 er {1} og arealet er {2}",side1,side2,side1*side2);
                }

}

public class Cirkel: Figur{
                //NB navn og id er nedarvede automatisk
                private int radius;

                //constructor kalder base dvs kalder Figurs constructor:
                public Cirkel(string n,int r) : base(n){
                id="Cirkel";
                radius=r;
                }
                //NB vis() er nedarvet automatisk

                //NB der findes nu 2 metoder vis():
                //override angiver at vi omdefinerer en nedarvet metode:

                public override void vis(){
                base.vis();
                Console.WriteLine("Radius er {0} og arealet er {1}",radius,3.1415*(radius*radius));
                }

}

public class app{
  public static void Main(string[] args){

                //opret en tabel af Figurer - kan også rumme alle nedarvede figurer!
                Figur[] figurer=new Figur[10];

                for(int i=0;i<10;i++){
                Random rnd=new Random();
                int tal=(int)rnd.Next(1,4);

                switch(tal){

                //der oprettes tilfældige figurer
                //polymorfisme:
                //systemet kan run time se hvilken type der oprettes
                // og henter den rigtige metode!!

                case 1:figurer[i]=new Figur(i.ToString());break;
                case 2:figurer[i]=new Rektangel(i.ToString(),i*10+1,i*20+1);break;
                case 3:figurer[i]=new Cirkel(i.ToString(),i*50+5); break;
                }
```

```
            //tabellen udskrives i samme loop:
            figurer[i].vis();
            }


            Console.Read();//teknisk af hensyn til Windows

  }

}
```

Resultat af en kørsel kunne være dette:



Opgaver:

1. Skriv en basisklasse Menneske og to klasser Mand og Kvinde som arver fra Menneske
2. skriv virtuelle metoder og override dem
3. skriv en metode som kun findes i Mand
4. og en som kun findes i Kvinde
5. skriv en applikation som arbejder med et array af Mennesker

Polymorfisme og Streams:

Klassen System.IO.Stream (i mscorlib.dll) er en abstrakt klasse som er parent til en række forskellige Streams. De metoder og egenskaber som defineres i Stream arves. Vi kan altså anvende de samme formler uanset hvad for en Stream vi arbejder med. Det gør det hele mere overskueligt!

Stream definerer en række metoder som: ReadByte(), WriteByte(), Seek() og en række properties som CanRead, CanWrite, CanSeek, Length og Position.

Der findes grundlæggende 3 slags Streams (altså klasser som arver fra Stream):

1. NetworkStream (en Stream til og fra Internettet/et netværk)
2. FileStream (til og fra en fysisk fil som er gemt et bestemt sted)
3. MemoryStream (til og fra et område i RAM hukommelsen, ikke fysisk gemt fil)

Det følgende meget lange kode eksempel anvender præcist de **samme metoder** på eksemplarer af de 3 streams. Når man har opdaget det, er koden slet ikke så kompliceret! Vi kan her tydeligt se C#'s **polymorfisme – forskellige** objekter, men stadig de **samme metoder**:

```
//Streams.cs - System.IO.Stream:
//DEMO af 3 forskellige streams:

using System;
using System.IO;
using System.Net.Sockets;
using System.Net;
using System.Text;

public class XStreams{


        public static void Main(){

//1. eksempel System.Net.NetworkStream:

                TcpClient tcp=new TcpClient("localhost",80);
                //GetStream():
                NetworkStream net=tcp.GetStream();

                Console.WriteLine("\nnet:");
                Console.WriteLine("CanRead: {0}",net.CanRead);
                Console.WriteLine("CanWrite: {0}",net.CanWrite);
                Console.WriteLine("CanSeek: {0}",net.CanSeek);
                //Denne Stream kan ikke 'seeke'!:
                //Console.WriteLine("Length: {0}",net.Length);
                //Console.WriteLine("Position: {0}",net.Position);
                string svar="ja";

                while(svar=="ja"){

                Console.WriteLine("\nIndtast HTML side:");
                string side=Console.ReadLine();
                string r="GET /"+side+" \r\n";
                //WriteByte(): skriv een byte ad gangen paa Stream:
                for(int i=0;i<r.Length;i++){
                        net.WriteByte((byte)r[i]);
                }
                int ind;
                //Standard ReadByte() i alle Streams:
                //ReadByte(): modtag een byte ad gangen fra Stream:
                //Returnerer -1 ved EOF End Of File:
                while((ind=net.ReadByte())!=-1){
                        Console.Write((char)ind);
                }
```

```
                        net.Close();
                        Console.WriteLine("\n\nIgen - ja?");
                        svar=Console.ReadLine();
                        if(svar=="ja"){
                                tcp=new TcpClient("localhost",80);
                                //GetStream():
                                net=tcp.GetStream();
                        }

                        }

//2. eksempel: System.IO.FileStream:
                        //Create: ny, overskriv evt eksisterende, filen skabes nu:
                        //ReadWrite: der kan reades og writes
                        //FileShare.Read: Flere kan reade samtidigt:

                        FileStream fil=new FileStream("eksempel.dat", FileMode.Create,
FileAccess.ReadWrite, FileShare.Read);

                        Console.WriteLine("\nfil:");

                        Console.WriteLine("CanRead: {0}",fil.CanRead);
                        Console.WriteLine("CanWrite: {0}",fil.CanWrite);

                        //Modsat NetworkStream kan der seekes paa en FileStream:
                        Console.WriteLine("CanSeek: {0}",fil.CanSeek);
                        Console.WriteLine("Length: {0}",fil.Length);

                        //Position er den byte som fil-pointeren nu peger paa:
                        Console.WriteLine("Position: {0}",fil.Position);

                        //teksten er 76 bytes i alt:
                        string s="I programmeringssproget C Sharp kan man skrive mange interessante
programmer";

                        //WriteByte(): skriv een byte ad gangen paa Stream:
                        for(int i=0;i<s.Length;i++){
                                fil.WriteByte((byte)s[i]);
                        }

                        //Begge er nu 76!:
                        Console.WriteLine("Position: {0}",fil.Position);
                        Console.WriteLine("Length: {0}",fil.Length);

                        //Pointeren skal nu pege paa den 14. byte forfra
                        fil.Seek(14,0);
                        for(int i=0;i<25;i++){
                                Console.Write((char)fil.ReadByte());
                        }

                        //Peger nu paa byte nr 39:
                        Console.WriteLine("\nPosition: {0}",fil.Position);
                        //Stadig 76:
                        Console.WriteLine("Length: {0}",fil.Length);

                        //Videre: Fil pointeren husker sin gamle vaerdi og fortsaetter:
                        for(int i=0;i<25;i++){
                                Console.Write((char)fil.ReadByte());
```

```
                }

                //Peg paa start eller byte nr 1:
                fil.Seek(0,0);

                Console.WriteLine();
                //Laes hele filen:
                int indbyte=fil.ReadByte();
                while(indbyte!=-1){
                                Console.Write((char)indbyte);
                                indbyte=fil.ReadByte();

                }

                fil.Seek(0,0);
                Console.WriteLine();
                int linje=1;
                //Laes hele filen og vis den med HEX tal som binaer fil:
                indbyte=fil.ReadByte();
                while(indbyte!=-1){
                                Console.Write("{0:X} ",indbyte);
                                indbyte=fil.ReadByte();
                                //Skift linje efter 10 HEX tal:
                                if((linje++)%10==0)Console.WriteLine();
                }
                fil.Close();

//3. eksempel: MemoryStream: at gemme i et RAM omraade:

                MemoryStream mem=new MemoryStream();

                //Der allokeres 256 bytes et sted i heap'en:
                mem.Capacity=256;

                //Samme metoder som ovenfor:
                Console.WriteLine("\nmem:");
                Console.WriteLine("CanRead: {0}",mem.CanRead);
                Console.WriteLine("CanWrite: {0}",mem.CanWrite);
                //Modsat NetworkStream kan der seekes paa en FileStream:
                Console.WriteLine("CanSeek: {0}",mem.CanSeek);
                Console.WriteLine("Length: {0}",mem.Length);
                //Position er den byte som fil-pointeren nu peger paa:
                Console.WriteLine("Position: {0}",mem.Position);

                Random rnd=new Random();
                //Skriv tilfaeldige tal (ASCII for store bogstaver) til RAM:
                for(int i=0;i<256;i++){
                                mem.WriteByte((byte)(rnd.Next(65,91)));
                }
                FileStream ny_fil=new FileStream("dump.dat",FileMode.Create);
                //Gem direkte i fil!:
                mem.WriteTo(ny_fil);

                Console.WriteLine("\nmem:");
                Console.WriteLine("Length: {0}",mem.Length);
                //Position er den byte som fil-pointeren nu peger paa:
                Console.WriteLine("Position: {0}",mem.Position);
```

```
                mem.Close();
        }

}
```

NB Det første eksempel: **NetworkStream forudsætter** at man har installeret en server på sin egen maskine så at localhost kan kaldes. I stedet kunne være valgt en **internetadresse**!

Som det ses kan man komme langt med de samme få metoder i helt **forskellige** sammenhænge som at gemme i RAM, skrive til en fil eller sende en request til en server!

Der er **ikke** nogen principiel forskel på at sende bits og bytes til en fysisk fil, til en række RAM adresser eller til en adresse på Internettet! Det **afgørende** er, at C# programmet skriver/sender en række af bytes. **Adressen** er forsåvidt underordnet! At sende bytes til **skærmen** eller konsollen er egentligt også det samme – derfor bruges også de **samme** metoder som f.eks. Console.Write()!

Hver af **Stream** klasserne har dog sine egne **specielle** metoder der gør det nemmere at arbejde med klassen. Her har vi dog koncentreret os om de **fælles** metoder som går tilbage til super klassen System.IO.Stream.

Det ses også at en Stream ikke altid **oprettes** på samme måde.

Vigtigt for alle Streams er at de bliver lukket igen med **Close**()!

Vi skal siden vende tilbage til Streams på netværk.

Abstrakte klasser:

En abstrakt klasse er en klasse som **ikke** kan instantieres. Vi har just set et eksempel på en abstrakt klasse nemlig System.IO.Stream. Interfaces (se næste afsnit) og abstrakte klasser ligger tæt på hinanden i anvendelsesområde.

Abstrakte klasser bruges også i C# til at etablere **'globale** konstanter' som i dette eksempel hvor vi forestiller os et firma som ønsker at gemme konstante data på eet sted i en 'global' datasamling:

```
//Eksempel på abstrakt klasse til at gemme firmaets 'globale' data:

using System;

public abstract class Firma {
                public const string NAVN="Computer Her og Nu Aps";
                public const string ADRESSE="Storevej 44, 2000 Frederiksberg";
                public const string TELEFON="38 99 22 00";

}

class app
{
```

```
        public static void Main(string[] args)
        {
                //instantiering giver kompiler fejl !!

                //Firma f=new Firma();
                Console.WriteLine("Firma Data:");
                Console.WriteLine(Firma.NAVN);
                Console.WriteLine(Firma.ADRESSE);
                Console.WriteLine(Firma.TELEFON);

                Console.Read();

        }
}
```

Hvis klassen Firma forsøges instantieret med new, fås følgende fejl meddelelse:

```
16,11): error CS0144: Cannot create an instance of the abstract class or
interface 'Firma'
```

Arrays og operatorer på objekter (operator overloading):

Objekter kan samles i et array eller tabel helt som et **array** af heltal eller strenge. Hvis vi har en klasse **Rektangel** kan vi altså oprette en samling af rektangler således:

```
//opret en tabel med 10 pladser til 10 rektangler:
Rektangel[] objekter=new Rektangel[10];
```

Disse rektangler kaldes så med **index** operatoren: fx er det første rektangel lig med **objekter[0]**.

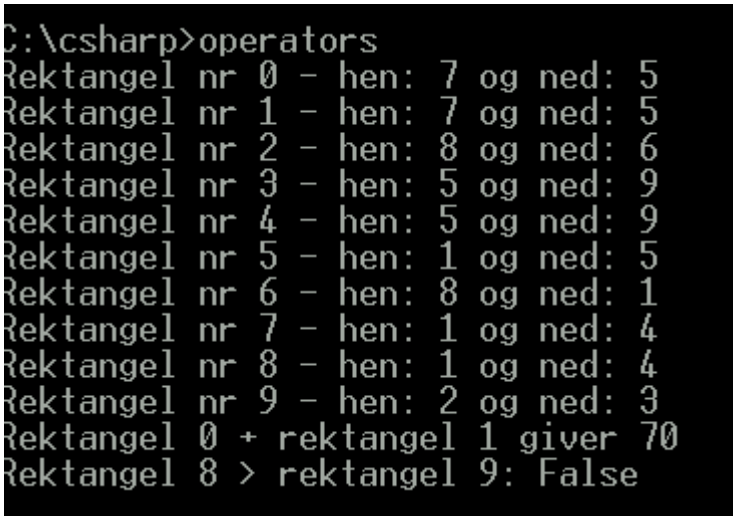
Men det er også muligt at sammenligne, sortere og manipulere rektangler med de almindelige operatorer som +, -, *, < og >.

I C# (og i C++) kaldes dette for **operator overloading**: den almindelige operator fx + bliver **overloaded** - dvs omdefineret til at kunne bruges i en ny sammenhæng.
Fx giver det mening at tale om at et rektangel er mindre end et andet hvis deres areal sammenlignes.

Følgende kode eksempel belyser dette:

```
//fil:operators.cs

//viser operator overloading på et object - et rektangel
//de almindelige operatorer +, -, *, /, >, >=, <, == kan anvendes på objekter også!

using System;

public class Rektangel{
```

```
		//for at gøre koden lettere er disse variable public!
		public int hen, ned;

		//constructor til Rektangel tager de 2 sider:
		public Rektangel(int h, int n){
		hen=h;
		ned=n;
		}

		//+ operator 'overloades' dvs 2 rektangler kan adderes: r1+r2:
		public static int operator +(Rektangel nr1,Rektangel nr2){
		return (nr1.hen*nr1.ned)+(nr2.hen*nr2.ned);
		}

		//de følgende metoder returnerer true hvis det første objekt er mindre/større:
		//< operator 'overloades' dvs 2 rektangler kan sammenlignes: if(r1<r2):
		public static bool operator <(Rektangel nr1,Rektangel nr2){
		return (nr1.hen*nr1.ned)<(nr2.hen*nr2.ned);
		}
		//> operator 'overloades' dvs 2 rektangler kan sammenlignes: if(r1>r2):
		public static bool operator >(Rektangel nr1,Rektangel nr2){
		return (nr1.hen*nr1.ned)>(nr2.hen*nr2.ned);
		}


}


class app{
public static void Main(){

		//opret en tabel med 10 pladser til 10 rektangler:
		Rektangel[] objekter=new Rektangel[10];

		//skab tilfældige rektangler og vis dem:
		for(int i=0;i<10;i++){

		Random rnd=new Random();
		int x=(int)rnd.Next(1,11);
		int y=(int)rnd.Next(1,11);
		objekter[i]=new Rektangel(x,y);
		Console.WriteLine("Rektangel nr {0} - hen: {1} og ned: {2}",i,objekter[i].hen,objekter[i].ned);
		}

		//hvad er summen af de 2 rektangler?
		Console.WriteLine("Rektangel 0 + rektangel 1 giver {0}",objekter[0]+objekter[1]);

		//er det ene rektangel støre end det andet?
		Console.WriteLine("Rektangel 8 > rektangel 9: {0}",objekter[8]>objekter[9]);

}
}
```



I C# anvendes **altid** dette skema (skabelon) til at overloade en operator:

```
public static int operator +(Rektangel nr1,Rektangel nr2){
```

Metoden skal være **static**, den kan returnere 'hvad som helst' (bool, int, string eller et objekt fx et nyt Rektangel objekt!), ordet **'operator'** skal angives foran operatoren som skal overloades. Disse metoder har altså typisk det **format** som er angivet i kode eksemplet.

NB: det er **muligt fuldstændigt** at omdefinere de kendte operatorer i et C# program. Vi er vant til at 4+5 giver 9, men med operator overloading er det **muligt** at overloade + operatoren så at 4+5 f.eks. giver strengen "Bla bla bla!"!! Denne form for overloading er selvfølgelig **ikke** nogen speciel god ide!!

Som det ses er det rimeligt nemt at konstruere overloadede metoder til +, <, > osv.
Vigtigt er at afgøre om disse operatorer giver **mening** på en bestemt klasse. Det giver ikke megen mening fx at lægge to personer sammen – i al fald ikke på computeren! Hvis vi har et Hus objekt kan hus1 og hus2 måske adderes ved at deres arealer adderes. Men dette skal vurderes i det konkrete tilfælde.

Operator overloading kan bruges til at **sortere** objekter.

Opgaver:

6. skriv kode til at 2 rektangler kan subtraheres således: r1 – r2
7. skriv kode til at + operatoren returnerer et nyt Rektangel hvis areal er summen af to rektangler
8. skriv en metode sorter() som sorterer et antal rektangler efter areal
9. skriv en metode sorter() som sorterer et antal rektangler efter den længste side
10. skriv en ny klasse Bil og find ud af hvordan man kan kode at bil1 **er mindre end** bil2! Du må her vælge en egenskab ved bilen som kan bruges som målestok

Interfaces – en anden slags arv:

Et interface erklæres som en klasse, men indeholder kun **abstrakte** metoder (inklusive properties, indexers og events – som alle opfattes som en slags metoder). Et interface kan IKKE indeholde felter!

```
public interface IBog {

        //properties:
        string Forfatter{get;set;}
        string Titel{get;set;}
        int Sider{get;set;}

        //en abstrakt metode:
        void vis_bog();
}
```

En klasse kan **implementere** et interface hvilket betyder at den har 'pligt' til at definere de metoder og properties som er indeholdt i dette interface. En **abstract** klasse **kan** indeholde metoder som er defineret og implementeret, **men** et interface kan **kun** rumme ikke-implementerede medlemmer!! Bortset herfra minder abstrakte klasser om interfaces!

På en vis måde 'arver' klassen fra dette interface. I en vis forstand kan man opfatte et interface som en basisklasse. Hvis klassen **Ejerlejlighed** implementerer et interface **ILejlighed** kan man sige at en ejerlejlighed **'er'** en lejlighed!

Et 'interface' (en **grænseflade** til verden uden om) er egentligt de metoder osv som klassen stiller til rådighed for 'den ydre verden' eller andre objekter eller 'klienten'. (Sådan fungerer interfaces typisk i Windows COM objekter). Jvf begrebet GUI – grafisk user interface! Alle medlemmer af et interface er derfor public – logisk nok.

Ved at bruge interfaces kan man igen sikre et **logisk hierarki**. Hvis man vil designe et system – kan man meget logisk starte med at definere en række interfaces – systemets **'skelet'**!

En klasse kan **både** arve fra en basisklasse **og** implementere et eller evt mange interfaces. (I selve kode **notationen**: class X:Y{} kan man faktisk **IKKE** se om Y er en basisklasse eller et interface!!).

En klasse kan implementere to interfaces som har den 'samme' metode (dvs **signaturen** er ens: samme navn, samme returtype, samme parametre!) – fx således:

```
public interface IKlasser{ string vis(); }
public interface IObjekter{ string vis(); }
```

I klassen kan man så skelne mellem de to implementeringer ved at skrive 'den fulde sti':

```
IKlasser.vis(){...}
```

IObjekter.vis(){...}

Arv fra flere klasser er **ikke** tilladt i C# (det er tilladt i fx C++), men i C# kan en klasse altså i stedet 'arve' fra mange interfaces. Effekten er stort set den samme – men man undgår de problemer som 'multiple inheritance' skaber i fx C++!

Hvis vi antager en basisklasse **Bog** og et interface **IBog** (interfaces starter – pr tradition - med 'I'), kan man altså skrive en ny klasse sådan:

public class Roman:Bog, Ibog{}

'Bog' efter kolon er **basis** klassen og 'IBog' er et **interface**.

Hvis en klasse både 'arver' fra en klasse og et interface, **skal** basisklassen stå først efter kolon!!

Et mere konkret eksempel på anvendelse af interfaces er følgende kode eksempel, som er kommenteret løbende:

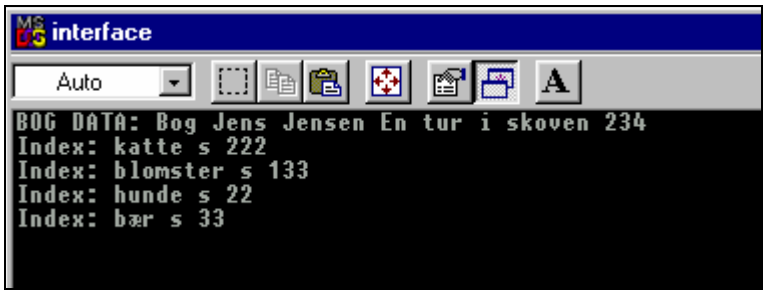
```
//interface.cs

using System;

//Alle bøger 'arver' fra IAbstraktBog (dvs det som er fælles for alle bøger):

public interface IAbstraktBog{

                //har 4 properties og en metode (her alle abstrakte uden implementering):

                string Forfatter{get;set;}
                string Titel{get;set;}
                int Sider{get;set;}

                //eks på property der kun er readonly: Bogens type:

                string Type {get;}

                void vis_bog();
}

//Endnu et  interface – 'arver fra' abstrakt bog: Index findes kun i en fagbog:
//Et interface kan implementere et andet interface!:

public interface IAbstraktFagBog : IAbstraktBog{

                //Denne property sættes altså med en streng tabel eller ord liste:

                string[] Index{get;set;}
}

//Klassen Bog implementerer dvs 'udfylder' interfacet IAbstraktBog - dvs 'er' en (abstrakt) bog:

public class Bog:IAbstraktBog{
                string forfatter,titel,type="Bog";
```

```
		int sider;

		public string Forfatter{get{return forfatter;}set{forfatter=value;}}
		public string Titel{get{return titel;}set{titel=value;}}
		public int Sider {get{return sider;}set{sider=value;}}

		//eks på property der kun er readonly - logisk!
		public string Type {get{return type;}}

		public void vis_bog(){}

}

//ny klasse Fagbog arver fra Bog - alle Bogs medlemmer arves:

//klassen fagbog implementerer interfacet IAbstraktFagbog:

public class FagBog:Bog,IAbstraktFagBog{

		//ArrayList kun bruges i stedet:

		string[] index=new string[300];

		public string[] Index{get{return index;}set{index=value;}}
}
```

Denne klasse kan så anvendes i en konkret applikation på denne måde:

```
class app
{
		public static void Main(string[] args)
		{
				FagBog bog1=new FagBog();
				bog1.Forfatter="Jens Jensen";
				bog1.Sider=234;
				bog1.Titel="En tur i skoven";
				bog1.Index=new string[]{"katte s 222","blomster s 133","hunde s 22","bær s 33"};
				Console.WriteLine("BOG DATA: {0} {1} {2}
{3}",bog1.Type,bog1.Forfatter,bog1.Titel,bog1.Sider);
				for(int i=0;i<bog1.Index.Length;i++){
						Console.WriteLine("Index: {0}",bog1.Index[i]);
				}
				Console.Read();
		}
}
```

Når programmet kører ses:



Implementere standard interfaces – IComparable:

Der findes i .NET og C# mange hundrede definerede interfaces, som kan (eller skal) implementeres i en given sammenhæng.

Vi skal se på et konkret eksempel: Det interface som hedder **IComparable** som findes i mscorlib.dll.

Formålet med dette interface er at muliggøre sortering af og sammenligninger mellem objekter. Det indeholder kun een abstrakt metode (som vi altså er tvunget til at implementere): **CompareTo**().

Koden for Bog skrives derfor om så den **implementerer** IComparable sådan (Bog implementerer nu **to** interfaces):

```
public class Bog : IAbstraktBog, IComparable {
                    string forfatter,titel,type="Bog";
                    int sider;

                    public string Forfatter{get{return forfatter;}set{forfatter=value;}}
                    public string Titel{get{return titel;}set{titel=value;}}
                    public int Sider {get{return sider;}set{sider=value;}}

                    //eks på property der kun er readonly - logisk!
                    public string Type {get{return type;}}

                    public void vis_bog(){}

                    //Her implementeres metoden i IComparable:
                    //Når 2 objekter sammenlignes er der 3 muligheder:

                    int IComparable.CompareTo(object o){
                                        Bog bog=(Bog)o;
                                        if(this.sider>bog.sider)return 1;
                                        if(this.sider<bog.sider)return -1;
                                        else return 0;
                    }
}
```

For at gøre det enkelt vælger vi at sammenligne to bøger med deres **sidetal** som kriterium – men vi kunne have valgt forfatter eller noget tredie i stedet (se Opgaver senere)! **Sammenligningskriteriet** er noget vi selv suverænt bestemmer!

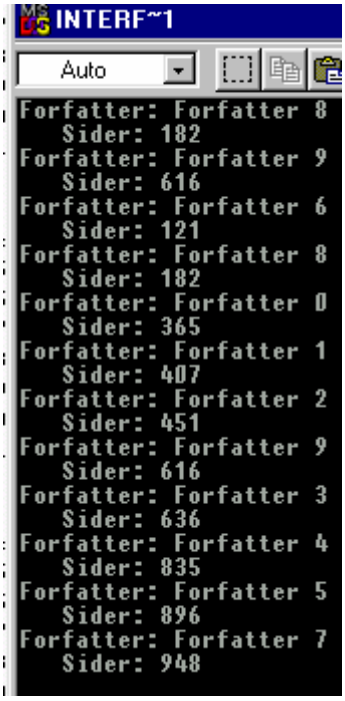
Vi kan nu skrive sætninger som fx:

if(bog1.CompareTo(bog2)==-1){ ...

og vi kan bruge **Array**.**Sort**() og **foreach** gennemløb! **CompareTo**() metoden kan forklares således: Hvis bog1 er størst returneres 1, hvis bog2 er størst returneres –1, hvis de er lige store returneres 0.

Applikations klassen kan så skrives sådan fx:

```
class app
{
        public static void Main(string[] args)
        {
                Bog[] liste=new Bog[10];
                Random r=new Random();

                //Opret 10 bøger:
                for(int i=0;i<10;i++){
                        liste[i]=new Bog();
                        liste[i].Forfatter="Forfatter "+i;

                        //Tilfældigt antal sider:
                        liste[i].Sider=r.Next(100,1000);
                }

                //Vis den usorterede liste:
                for(int i=0;i<10;i++){
                        Console.WriteLine("Forfatter: {0}",liste[i].Forfatter);
                        Console.WriteLine("   Sider: {0}",liste[i].Sider);
                }

                //Pga IComparable kan listen sorteres!
                Array.Sort(liste);

                //Vis sorteret:
                for(int i=0;i<10;i++){
                        Console.WriteLine("Forfatter: {0}",liste[i].Forfatter);
                        Console.WriteLine("   Sider: {0}",liste[i].Sider);
                }

                //OBS vi kan nu ogsaa anvende foreach!
                foreach(Bog b in liste){
                        Console.WriteLine("Forfatter: {0}",b.Forfatter);

                }


                Console.Read();
        }
}
```



Dette udpluk af en kørsel viser, at bøgerne bliver pænt sorteret efter sidetal – startende med forfatter nr 6!

Ovenstående sortering kunne også kodes sådan:

```
Comparer comparer = Comparer.Default;
Array.Sort(liste, comparer);
```

Dette kunne være relevant, hvis man ville anvende **forskellige** sorteringsmetoder i samme program.

Opgaver:

1. **Omskriv** interfacet IAbstraktBog så det **både** rummer en property Fornavn **og** en property Efternavn i stedet for Forfatter!
2. Omskriv derefter klassen Bog og Fagbog så de også har disse 2 egenskaber.
3. Omskriv disse klasser så de sorteres efter forfatterens **efternavn** og skriv en applikation, som anvender denne sortering!

Strukturer:

Strukturer eller **structs** har historisk været **forløbere** for OO klasser. En struct er en gruppering af datamedlemmer i en struktur. Strukturer går tilbage til C og Pascal (**records**) og kan minde om poster i en tabel.

I C# er der **næsten** ingen forskel på en struktur og en klasse. Derfor bruges som regel klasser – og sjældent strukturer i C#. Dog findes en række præ definerede strukturer som fx Size og Point.

Den vigtigste **forskel** er teknisk: Et klasseobjekt gemmes som et **referenceobjekt** i heap'en, det frit tilgængelige RAM område. Hvis vi skriver således:

XClass x;

har vi erklæret en **pointer** eller **reference** til et XClass objekt, **men** vi har ikke oprettet objektet endnu! Vi opretter det nye objekt sådan:

x=**new** XClass();

På den RAM **adresse** (fx 1000) hvor x gemmes gemmes en anden RAM adresse (fx 4444) hvor **selve** objektet gemmes. Klasse objekter er altså mere 'indviklede' end fx en int eller en struct. Et klasse obejkt kræver så at sige 2 RAM adresser: en pointer og en destination.

En struct er en **value** type og gemmes i programmets **stack**, dets eget RAM område. En struct svarer til en int eller char eller double. Derfor skal en struct heller **ikke** – nødvendigvis - instantieres med new som et klasse objekt! (Men den **kan** godt instantieres med new!).

Hvis vi har defineret en struct ved navn Punkt, instantierer vi et nyt punkt sådan:

Punkt p1;

Akkurat som vi ville skrive:

int x;

Når strukturer stadig bruges skyldes det at de er meget egnede til at gemme **små** blokke af data. En struct må helst ikke fylde mere end **16** bytes. Hvis en struct ikke fylder ret mange bytes, er den mere effektiv og hurtigere end en klasse. Dette skyldes at structs gemmes på stacken og derfor er der meget hurtigere access til disse data og en struct kræver ikke så megen 'overhead' og 'memory management' som en klasse.

I nedenstående program er derfor brugt operatoren **sizeof**() for at vise hvor mange bytes structen rummer. Programmet erklærer 2 strukturer Dato og Tid og bruger dem til at 'administrere' datoer og klokkeslet:

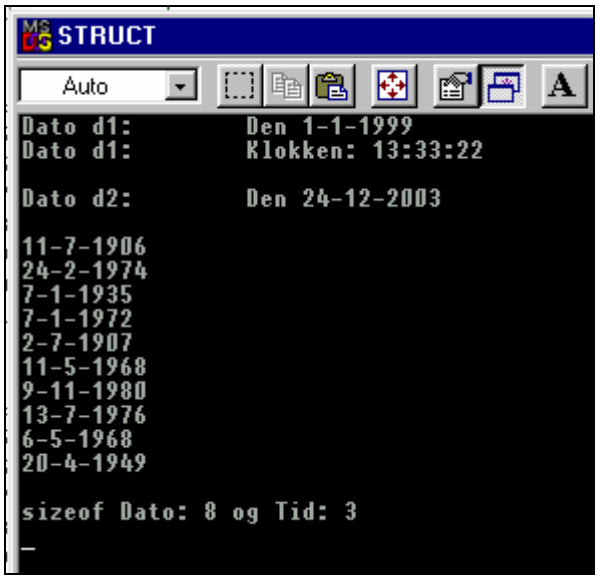
```
//Eksempel paa anvendelse af structs eller strukturer:

//NB en struct skal helst fylde mindre end 16 bytes!
//structs er value typer som int, char, double.
//structs er en gruppering af data i en 'post' eller 'record'.

using System;

class Strukturer
{
        public static void Main(string[] args)
        {
                //Instantier 2 Dato objekter: NB ingen 'new':
                //strukturens medlemmer kan sættes direkte da de er public:
                Dato d1;
```

```
                                d1.dag=1;
                                d1.maaned=1;
                                d1.aar=1999;

                                Dato d2;
                                d2.dag=24;
                                d2.maaned=12;
                                d2.aar=2003;

                                //Instantier en Tid:
                                Tid klokken;
                                klokken.minut=33;
                                klokken.sekund=22;
                                klokken.time=13;

                                d1.tid=klokken;

                                Console.WriteLine("Dato d1: \tDen {0}-{1}-{2}",d1.dag,d1.maaned,d1.aar);
                                Console.WriteLine("Dato d1: \tKlokken: {0}:{1}:{2}",d1.tid.time,d1.tid.minut,
d1.tid.sekund);
                                Console.WriteLine("\nDato d2: \tDen {0}-{1}-{2}",d2.dag,d2.maaned,d2.aar);

                                Dato[] datoer=new Dato[10];
                                Random r=new Random();

                                //Opret 10 tilfaeldige datoer:

                                for(int i=0;i<10;i++){
                                        datoer[i].dag=(byte)r.Next(1,29);
                                        datoer[i].maaned=(byte)r.Next(1,13);
                                        datoer[i].aar=(short)r.Next(1900,2001);

                                }

                                //Udskriv de 10 datoer:

                                Console.WriteLine();
                                for(int i=0;i<10;i++){
                                        Console.Write(datoer[i].dag);
                                        Console.Write("-"+datoer[i].maaned);
                                        Console.WriteLine("-"+datoer[i].aar);
                                }



                                //kompiler: csc /unsafe main.cs
                                unsafe{
                                Console.WriteLine("\nsizeof Dato: {0} og Tid: {1}",sizeof(Dato),sizeof(Tid));

                                Console.Read();
                                }
                }
}

//I alt 7 bytes:
public struct Dato {
                public byte dag;
                public byte maaned;
```

```
                public short aar;
                public Tid tid;
}
//Ialt 3 bytes:
public struct Tid {
                public byte time;
                public byte minut;
                public byte sekund;
}
```

Som det ses kan **sizeof** kun kompileres, hvis koden er markeret med **unsafe**{} **og** hvis der kompileres med csc /unsafe !! (Unsafe kan også sættes i SharpDevelop: Project Options). Årsagen er at sizeof direkte refererer til det underliggende operativ system og dets memory management.

De to strukturer opfylder 'kravene' til en struct: at den helst skal være under ca **16** bytes. Her er også valgt typer (**byte**, **short**) som fylder mindst muligt i RAM. Brugt sådan kan structs være meget **effektive** – også mere effektive end klasser! (I eksemplet kunne jo i stedet være erklæret 2 **klasser** Dato og Tid på næsten samme måde!).



Brug af unsafe:

En række kodekonstruktioner som kendes fra C og C++ kan også bruges i C# hvis koden markeres som unsafe! Dette er ikke specielt anbefalelsesværdigt – for det går egentligt imod det som var ideen med C# - men det kan altså lade sig gøre!

Man kan på den måde fx læse direkte i memory'en eller RAM. Man kan anvende pointere – som dette lille eksempel viser:

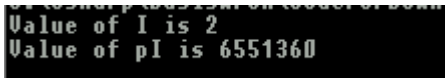
```
using System;

public class PointerDemo
{
```

```
  public unsafe static void Main()
  {
    int I = 4;
    int* pI = &I;

    *pI = 2;
    Console.WriteLine("Value of I is {0}", I);
  }
}
```

Her er hele Main() markeret som unsafe.

&I betegner adressen på variablen I og er derfor en **pointer** til I.
*pI betegner **værdien** af en pointer pI.

Hvis vi ændrer koden til at udskrive værdien af pointeren pI i stedet for værdien af variablen, vil vi få udskrevet den RAM adresse, som I ligger på:



Ved hjælp af unsafe kan man skrive direkte til memory'en – hvilket kan give anledning til mange crash af maskinen!

Strukturer modsat klasser:

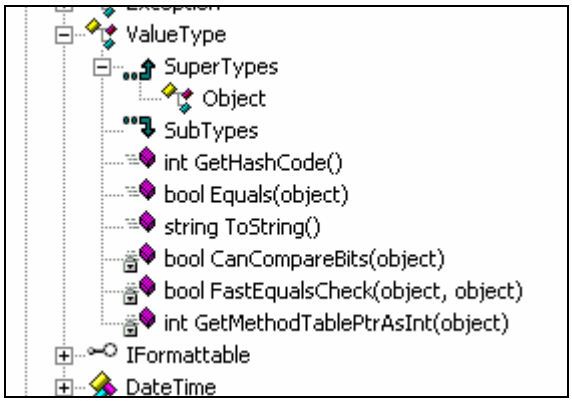
En struct kan rumme stort set de samme **medlemmer** som en klasse:

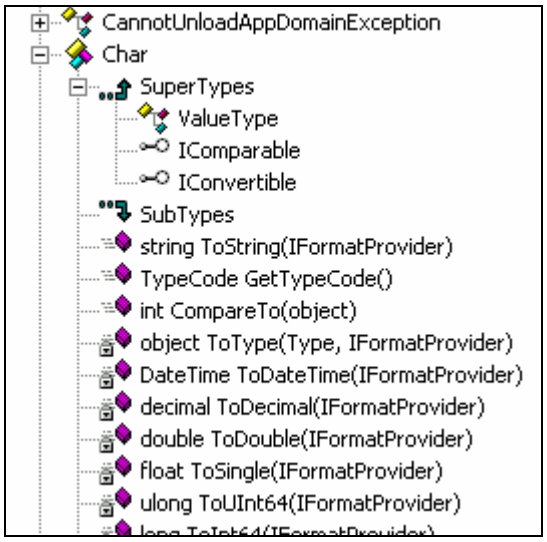
1. felter
2. metoder
3. properties
4. indexers
5. events
6. delegater
7. enums
8. andre structs

Men en struct kan ikke indeholde en default constructor – uden parametre - som fx public Dato(). En struct er **sealed** dvs kan ikke nedarves og kan ikke arve fra andre structs – men kan implementere interfaces. En struct kan ikke være abstract eller protected (pga at en struct ikke kan bruges til arv). En struct kan ikke have en destructor.

Selv om en struct instantieres med new (hvilket egentligt ikke er nødvendigt) allokeres (gemmes) den stadig på programmets **stack** – IKKE i heap'en!

Alle structs arver fra System.ValueType (direkte) men da System.ValueType arver fra System.Object er alle strukturer altså også objekter i C#:





System.Char er altså en ValueType eller en struct!

Typer og hvad der *kan* stå i en C# fil (*.cs fil):

C# er **defineret** sådan at det, som 'må' stå i en cs fil, er en **type**, dvs et eksemplar af eller en instans af **System**.**Type**.

Nedenfor følger en række eksempler på, hvad en cs **'minimums'** fil kan rumme:

// OK typer i cs fil: Typerne behøver IKKE at være 'public'!!

//fil: x.cs:

```
public class X{}
```

//fil: x.cs: Filen består KUN af en enum!

```
public enum X{}
```

//fil: x.cs: KUN af en struct!

public **struct** X{}

//fil: x.cs: KUN af en eventhandler jvf senere om Windows programmeringen:

public **delegate** event_handler(object o);

//fil: x.cs: KUN af et interface:

public **interface** X{}

Alle disse små filer kan kompileres med csc **/target**:library – uden problemer. De produceres som DLL filer.

Følgende er derimod **IKKE** gyldigt i C#:

//fil: x.cs:

public string vis_fornavn(){return "ole";}

Nu fås en kompilerfejl:

```
oft (R) Visual C# .NET Compiler version 7.00.9466
crosoft (R) .NET Framework version 1.0.3705
ght (C) Microsoft Corporation 2001. All rights reserved.

cs(5,9): error CS1518: Expected class, delegate, enum, interface, or
  struct
```

I **modsætning** til andre programmerings sprog som f.eks. C++ kan man **IKKE** i C# kompilere en fil som kun består af en metode. En sådan metode skulle **'wrappes'** ind i en klasse eller en anden System.Type!

Som det ses accepterer kompileren kun de 5 nævnte typer!

Lidt forvirrende er det så måske, at man i IL kode (assembler kode) GODT kan skrive og kompilere en il fil som **kun** indeholder en metode!!

Namespace:

Et namespace i C# er en samling af klasser under en **overskrift**. Eksempler på forud definerede namespaces er: System, System.Drawing, System.IO. Hvis jeg selv skriver en klasse 'trekant' kan jeg samtidigt definere at 'trekant' tilhører et namespace 'figurer', som måske også rummer klasser som 'firkant' og 'cirkel'!

Et namespace tjener mest til at etablere et logisk **hierarki** så man kan finde rundt i tusindvis af klasser. I 'System.Windows.Forms' er **System** det ydre namespace, **Windows** et under namespace under System og Forms et under namespace under Windows.

Sætningen **using** System tjener til at koden kan skrives **nemmere** – og til at kompileren kan vide **hvilket** namespace dvs **hvilke** klasser og metoder jeg taler om.

Et namespace kan være defineret i en enkelt fil eller – hvis der er tale om mere professionelle applikationer - i **mange** filer.

Et namespace er en **logisk** struktur, IKKE en fysisk struktur. Fx findes der ikke noget namespace mscorlib eller 'using mscorlib' men der findes en DLL fil (en **fysisk** fil) mscorlib.dll! Det logiske og fysiske dækker **ikke** altid hinanden.

Namespaces er kun 'noget som vi ser' – set fra compilerens synspunkt er der ikke noget som hedder namespaces. Fx klassen 'Form': Set fra .NET systemets synspunkt **hedder** denne klasse simpelthen 'System.Windows.Forms.Form' !!! Det er kun os, der opfatter det sådan, at klassen hedder Form men ligger i det og det namespace!

En 'using System;' er et 'præ processor direktiv' som gør at systemet **indsætter** ordet 'System.' foran vores linje: 'Console.WriteLine()' så at den linje kompileren rent faktisk arbejder med kommer til at lyde: 'System.Console.WriteLine();'.

Det er vigtigt at huske, at en **'using** System.Xml;' er en 'typografisk' genvej – for selve DLL filen bliver **IKKE** refereret (inddraget) fordi jeg skriver en using!!

På en vis måde svarer et namespace til en mappe på harddisken og en klasse til en fil – forstået på den måde at to filer godt kan have det samme navn, HVIS de ligger i to forkellige mapper! Men bortset fra denne analogi har et namespace ikke noget at gøre med en fysisk struktur.

Java anvender også namespaces, **men** i Java er et namespace bundet – som en **fysisk** struktur - til en mappe på computeren! Namespaces findes også i C++.

Namespaces gør **genbrug** og anvendelse af **komponenter** meget nemmere i koden. Fx kan jeg have anskaffet mig en 'fremmed' DLL fil som indeholder en klasse ArrayList. På grund af namespaces kan jeg godt **både** bruge denne 'fremmede' ArrayList **og** .NETs egen ArrayList samtidigt! Det eneste jeg skal gøre er at angive den fulde sti til klassen fx sådan – hvis vi forudsætter et firmanavn som namespace:

```
System.Collections.ArrayList dotnetliste;
SuperDigital.ArrayList sdListe;
```

Namespaces skal dels skabe en logisk orden i klasserne, dels løse det problem at vi kan have **forskellige** klasser med præcist **samme** navn. Hvis følgende kode forsøges kompileres, giver det en fejl og koden kan **ikke** kompileres:

```
// illustrerer name clash og namespace: fx Timer:
```

```
using System;
using System.Windows.Forms;

class N {

		Timer timer;

}
```

```
C:\csharp\namespace>csc clash.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

clash.cs(8,2): error CS0104: 'Timer' is an ambiguous reference
C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705\mscorlib.dll: (Location of symb
        related to previous error)
C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705\System.Windows.Forms.dll: (Loca
        of symbol related to previous error)

C:\csharp\namespace>
```

Da der findes flere (helt forskellige!) Timer klasser i .NETs basisklasser (BCL), er 'Timer' ikke nogen entydig betegnelse.

I stedet kan vi så skrive (præcist som mange builder-programmer gør!):

```
class N {

		System.Windows.Forms.Timer timer=new System.Windows.Forms.Timer();

}
```

Alias for et namespace eller for en klasse:

I C# kan også anvendes et **'alias'** som en genvej eller 'shorthand' - fx på denne måde:

```
using System;
using SWF=System.Windows.Forms;

class N {

		SWF.Timer timer=new SWF.Timer();

}
```

Eller:

```
//eks paa alias – her for en klasse!
```

```
using C=System.Console;

class MainClass
{
		public static void Main(string[] args)
		{
			C.WriteLine("Hello World!");

			C.Read();
		}
}
```

Systemet svarer tildels til #define sætninger i C og C++.

User Defined namespace:

Et eksempel på et **'user defined'** namespace kunne være følgende hvor alt er samlet summarisk i en enkelt fil:

```
//fil: namespace.cs
//demonstrerer nestede namespaces i 3 lag:

using System;

namespace Ydre{
using System;
			public class Y{}
			namespace Midterste{
				using System;
					public class Midter{}


					namespace Inderste{
						using System;
							public class Inderst{}
							public class Sidste{}

					}
			}
}



class app
{
		public static void Main(string[] args)
		{
			//Y y=new Y() - giver fejl:
			Ydre.Y y=new Ydre.Y();
```

```
                    Ydre.Midterste.Midter m=new Ydre.Midterste.Midter();
                    Ydre.Midterste.Inderste.Inderst i=new Ydre.Midterste.Inderste.Inderst();
                    Ydre.Midterste.Inderste.Sidste s=new Ydre.Midterste.Inderste.Sidste();


            }
}
```

Hvis man erklærer klassen Y som:

Y y=new Y();

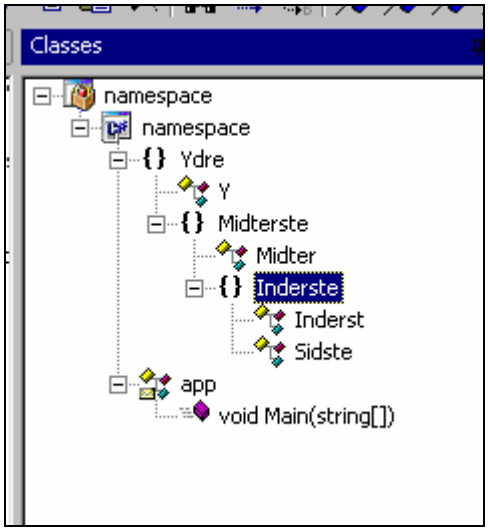
Fås en kompilerfejl sådan:

```
soft (R) .NET Framework version 1.0.3705
 (C) Microsoft Corporation 2001. All rights reserved.

7,3): error CS0246: The type or namespace name 'Y' could not be found
are you missing a using directive or an assembly reference?)
```

Hvis først en klasse er erklæret inden for et namespace SKAL dette namespace med som forstavelse til navnet!!
Dette kunne så undgås med en: **using Ydre**;

Kerneklasserne i C# er opbygget præcist som i dette eksempel med namespaces inden i hinanden – såkaldte 'nestede' namespaces. Der refereres til en klasse med **'punktum** operatoren' som ellers i OOP.

Eksemplet med 3 namespaces vist i SharpDevelop:



Opgaver:

1. Definer først **meget** kort et antal klasser inden for området (**namespace**) **Biler**.

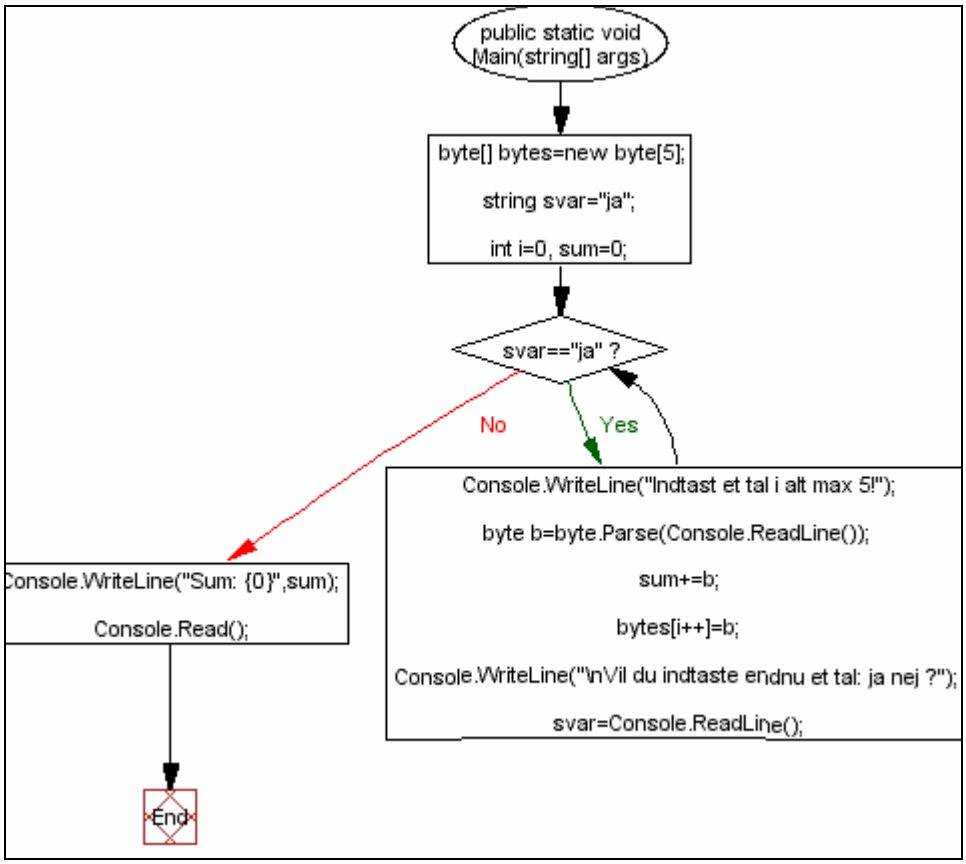
2. Opret en namespace Biler
3. Opdel derefter Biler i underafdelinger med hvert sit namespace og anbring klasserne i disse namespaces. Opbyg en logisk struktur/hierarki således at et under namespace repræsenterer en under kategori af biler.
4. Skriv et kort program hvor du tester at systemet med namespaces fungerer.

Exceptions i C# programmer:

Vi har tidligere set på exceptions i forbindelse med at åbne eller læse fra filer. En **exception** er i **snæver** betydning en **begivenhed** som opstår **runtime** dvs under et programs kørsel og som fører til at programmet crasher eller stopper brat med en **system** fejl meddelelse. Men i videre betydning kan vi selv definere hvad vi vil forstå ved en exception og vi kan under alle omstændigheder selv bestemme hvad der skal ske hvis der opstår en exception.
Exceptions er altså et eksempel på at der kan vise sig en 'fejl' i et program som ellers er kompileret OK og på den måde godkendt af systemet!
Kompileren kan **ikke** forudse 'events' under programmets kørsel – lige som den ikke kan afsløre eller finde 'logiske' fejl i et program.

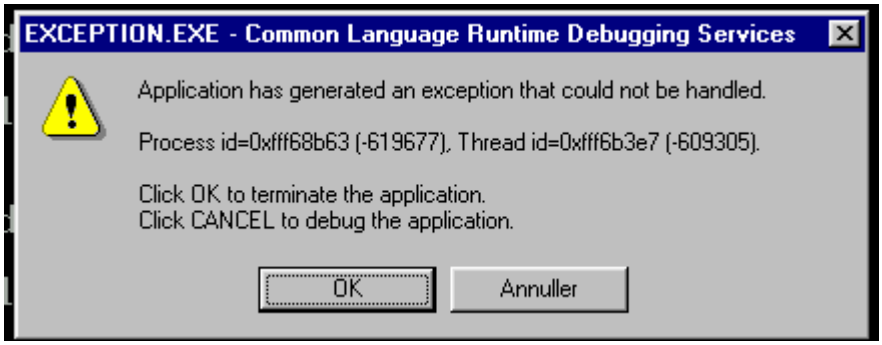
En exception vil således let kunne opstå i dette program:

```
public static void Main(string[] args)
{
        byte[] bytes=new byte[5];
        string svar="ja";
        int i=0, sum=0;
        while(svar=="ja")
        {
                Console.WriteLine("Indtast et tal i alt max 5!");
                byte b=byte.Parse(Console.ReadLine());
                sum+=b;
                bytes[i++]=b;
                Console.WriteLine("\nVil du indtaste endnu et tal: ja nej ?");
                svar=Console.ReadLine();
        }
        Console.WriteLine("Sum: {0}",sum);

        Console.Read();


}
```



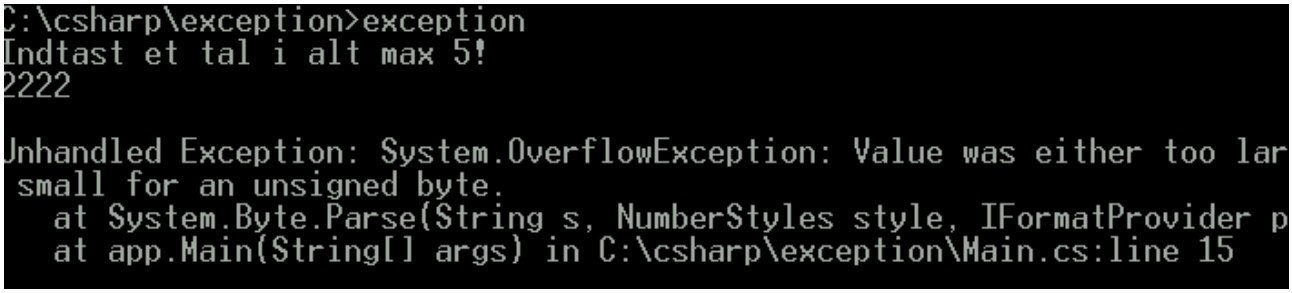
Programmet er bestemt **ikke** 'robust' eller 'brugervenligt' – idet programmet crasher hvis brugeren glemmer advarslen og indtaster flere end 5 tal!

Brugeren får så blot en system-meddelelse i hovedet! Emnet exception handling drejer sig **dels** om at undgå alt for mange crash eller mere eller mindre forståelig system fejl bokse **dels** om at sikre sig at fejl events ikke **kan** opstå! At gøre programmerne 'idiot sikre'.



Ikke mange brugere vil have megen gavn af denne boks!

Problemet med programmet er at **flere** ting kan 'gå galt'. Hvis jeg indtaster tallene 2, 22, 222, **2222** crasher programmet når det sidste tal nås – fordi en **byte** kun kan rumme tallene fra 0 til og med 255! Der opstår et overflow som det ses ved denne fejl meddelelse i konsollen:



I C# er alle exceptions delt i et stort antal **klasser** – her har vi et eksempel på klassen System.OverflowException. Hvis et int var blevet tildelt værdien 10.000.000.000 ville der også være opstået et overflow for max værdien for en int er ca 2,1 milliard. Hvis vi 'kommer til' at dele med 0 opstår en **DivideByZeroException**! osv.

Hovedsagen er dog at normalt ønsker vi ikke at programmet skal crashe og blot vise en systemboks! Læg også mærke til at udskriften af summen aldrig bliver til noget hvis en exception opstår – det hele stopper midt i processen. Hovedsagen er at vi normalt ikke ønsker at systemet skal gå ind i programmet 'på egen hånd' – vi ønsker selv at 'catche' de uheld der kan ske.

Den nemmeste løsning er at anvende en try .. catch .. finally konstruktion således:

```
try{
while(svar=="ja")
{
                Console.WriteLine("Indtast et tal i alt max 5!");
                byte b=byte.Parse(Console.ReadLine());
                sum+=b;
                bytes[i++]=b;
                Console.WriteLine("\nVil du indtaste endnu et tal: ja nej ?");
                svar=Console.ReadLine();

}
}
catch{
                Console.WriteLine("Der opstod fejl ved indtastningen!");
}
finally{
                Console.WriteLine("Sum: {0}",sum);
}
```

Programmet fungerer nu sådan at det som står i den sidste blok (**finally** blokken) **ALTID** udføres **uanset** hvad der sker – der udskrives altså altid en sum så programmet slutter på en for os bedre måde trods alt!
Hvis der opstår en eller anden exception i **try** blokken springer programmet **straks** ned til 1. linje i **catch** blokken (og fortsætter derefter i **finally** blokken). Hvis der **ikke** opstår 'fejl' gennemløbes try blokken og derefter finally blokken.

Vi undgår **fuldstændigt** de fæle system meddelelser! Alle programmer som 'kan gå galt' bør have en konstruktion der ligner denne. Finally blokken bruges som ofte til 'at rydde op' – hvis en fil er åbnet og når går galt skal den lukkes igen og det skal ske under **alle** omstændigheder.

I de **fleste** tilfælde kan man klare sig med en sådan exception handling men der er i C# mange flere muligheder som vi her kort skal illustrere.

For det første kan man skrive sine egne nye Exception **klasser** med flere nye **egenskaber**.

For det andet er det almindeligt at anbringe 'risikabel' kode i en selvstændig **metode** som kan **'kaste'** exceptions på passende steder – med formlen **throw new Exception()**.

Et eksempel herpå er følgende, hvor vi har omstruktureret programmet:

```
// Eksempel på Exception handling i C#:

using System;

//Eksempel på en ny Exception klasse med flere egenskaber:

public class EException : Exception{
        public int linje;
        public string metode;
        public EException(string e,int l,string m):base(e){
                linje=l;
                metode=m;

        }
}

class app{

        public static int sum=0;

        //OBS her er oprettet en ny metode som kaldes i Main() med en try catch finally:

        public static void find_tal(){
        byte[] bytes=new byte[5];
        string svar="ja";
        int i=0;
        while(svar=="ja")
                {
                        Console.WriteLine("Indtast et tal i alt max 5!");
                        byte b=0;
                        try{
                        b=byte.Parse(Console.ReadLine());
                        }catch{
                        throw new Exception("Tallet er enten over 255 eller under 0!");
                        }
                        sum+=b;
                        bytes[i++]=b;
                        if(i>4)throw new EException("Der er indtastet for mange tal! Der
        er kun 5 pladser i tabellen!",37,"find_tal()");

                        Console.WriteLine("\nVil du indtaste endnu et tal: ja nej ?");
                        svar=Console.ReadLine();
                }
        }

        public static void Main(string[] args)
```

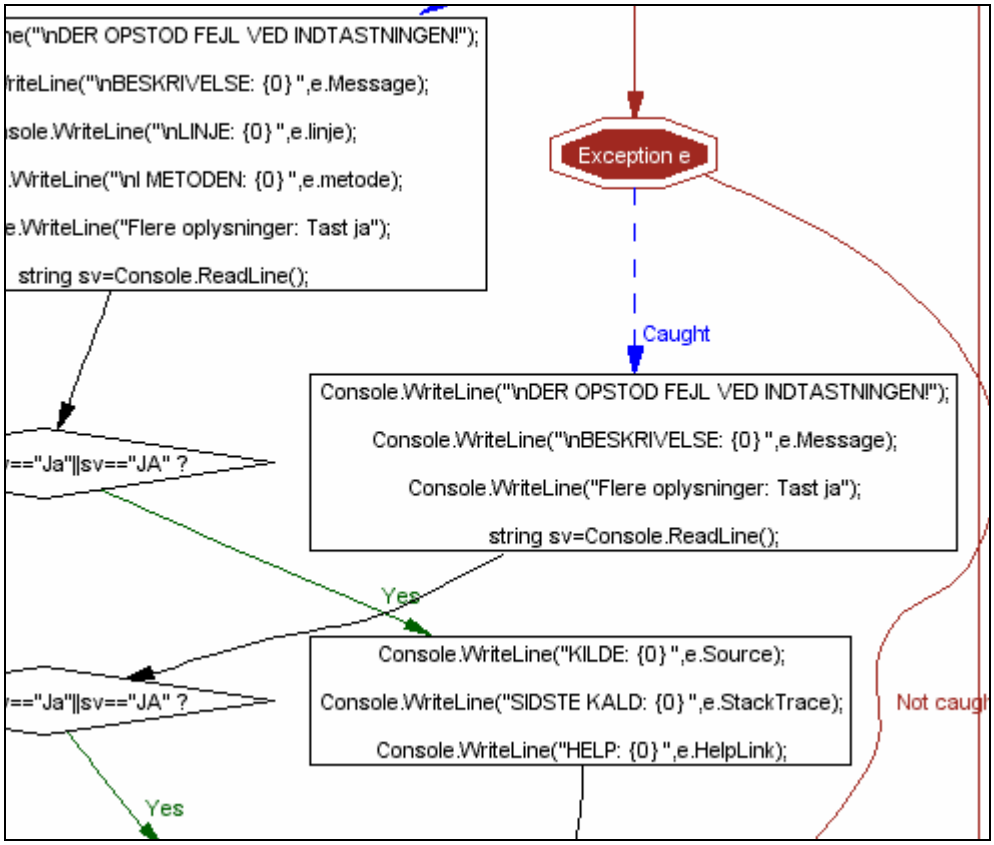
```
        {
                try{find_tal();}

                //den mest specifikke Exception skal altid stå først
                catch(EException e){
                        Console.WriteLine("\nDER OPSTOD FEJL VED
                        INDTASTNINGEN!");

                        //e.Message osv er de faste egenskaber i klassen Exception:

                        Console.WriteLine("\nBESKRIVELSE: {0} ",e.Message);
                        Console.WriteLine("\nLINJE: {0} ",e.linje);
                        Console.WriteLine("\nI METODEN: {0} ",e.metode);

                        Console.WriteLine("Flere oplysninger: Tast ja");
                        string sv=Console.ReadLine();
                        if(sv=="ja"||sv=="Ja"||sv=="JA"){
                        Console.WriteLine("KILDE: {0} ",e.Source);
                        Console.WriteLine("SIDSTE KALD: {0} ",e.StackTrace);
                        Console.WriteLine("HELP: {0} ",e.HelpLink);
                        }

                }
                //Exception er altid den mest generelle og skal stå sidst:
                catch(Exception e){
                        Console.WriteLine("\nDER OPSTOD FEJL VED
                        INDTASTNINGEN!");

                        Console.WriteLine("\nBESKRIVELSE: {0} ",e.Message);
                        Console.WriteLine("Flere oplysninger: Tast ja");
                        string sv=Console.ReadLine();
                        if(sv=="ja"||sv=="Ja"||sv=="JA"){
                        Console.WriteLine("KILDE: {0} ",e.Source);
                        Console.WriteLine("SIDSTE KALD: {0} ",e.StackTrace);
                        Console.WriteLine("HELP: {0} ",e.HelpLink);
                        }

                }
                finally{
                        Console.WriteLine("\nSummen er: {0}",sum);
                        Console.Read();

                }



        }
}
```

Vores program kan nu **opfange** 2 slags Exceptions (en mere specifik **EException** og en helt generel **Exception** som opfanger **ALLE øvrige** 'fejl'). Derfor bliver der 2 catch blokke til sidst i Main(). Metoden find_tal() kaldes af Main() og kan flere steder **'kaste'** exceptions.

De væsentlige ændringer er markeret med fed i eksemplet.

Vi skal også senere vende tilbage til det generelle spørgsmål om hvordan sikrer at et program er **brugervenligt** og **'sikkert'**.
Ovenstående kode eksempel er selvfølgelig på mange måder **kunstigt** for nogle af problemerne fx at der kun er plads til 5 tal – kunne let løses på anden måde. Eksemplet er mest en illustration af hvordan evt problemer kan løses.

Kontrolflowet i forbindelse med exceptions vises også fint programmet Visustin fra http://www.aivosto.com/ (omtalt tidligere) – her er et lille uddrag af det diagram som ovenstående kode producerer:



Opgaver:

1. Hvilke forhold kan 'gå galt' når man åbner, lukker, læser fra, skriver til filer? Lav en liste og skitser hvordan man med exception handling kan undgå 'crash'!
2. Skriv et kort program der skal indtastes navne på nøjagtigt 4 tegn og hvor exceptions vises i en MessageBox.
3. Skriv et program hvor bruger skal indtaste en række personer med navn og telefonnummer og hvor samme navn ikke må forekomme 2 gange – brug metoderne i exception handling.
4. Skriv en ny Exception klasse som skal bruges ved de 'crash' der kan ske hvis man lægger flere elementer ind i et array end der er plads til!

Serialisering af objekter – gem objekter:

At **serialisere** et objekt er at **gemme** det og hele dets **tilstand** i en fil (typisk). At **deserialisere** er den modsatte proces: at indlæse et objekt fra en fil.

Et objekt kan dog også serialiseres til en netværksforbindelse (socket), en CD, til Windows Clipboard (udklipsholder) mv.

I C# bliver ikke blot objektets tilstand (tilstandsvariable) men også dets **referencer** (relation til andre objekter) gemt. Det som gemmes er en 'objekt **graf'**.

En serialisering kan ske på to forskellige måder i C#: i en *.dat fil (med en **binær** formattering) eller i en *.xml fil (med en **SOAP** formattering).

Disse ting er defineret i System.Runtime.Serialization.Formatters.dll.

En klasse skal mærkes med en attribut [**Serializable**] for at kunne serialiseres. I det følgende eksempel er vist de to måder at gemme på:

```
// Demo program som gemmer objekter i XML og binært dat format
// Demo af C# serialisering og deserialisering

//se filen huse.xml og huse.dat!

using System.IO;
using System.Runtime.Serialization.Formatters.Soap;
using System.Runtime.Serialization.Formatters.Binary;
using System;
using System.Collections;


//NB hvis ikke klassen får denne 'attribut' kan den ikke serialiseres!

[Serializable]
public class Hus{
                private string kode;
                private int byggeaar;
                public Hus(string k,int a){kode=k;byggeaar=a;}
                public string getKode(){return kode;}
                public int getAar(){return byggeaar;}
}
class app
{
                public static void Main(string[] args)
                {
                                Hus hus1=new Hus("abc-22",1977);
                                Hus hus2=new Hus("xz-1234",1999);
                                Hus hus3=new Hus("ab-77",2001);

                                //alle objekter gemmes i ArrayList som serialiseres samlet
                                //hvert objekt kunne selvfølgelig gemmes hver for sig - hvis ønsket

                                ArrayList liste=new ArrayList();
```

```
                    liste.Add(hus1);
                    liste.Add(hus2);
                    liste.Add(hus3);

                    //gem de 3 objekter i XML format med en SOAP Formatter
                    //'Simple Object Access Protocol':

                    FileStream fil=File.Create("huse.xml");
                    SoapFormatter soap=new SoapFormatter();
                    soap.Serialize(fil,liste);
                    fil.Close();

                    //læs tilbage fra filen huse.xml med metoden Deserialize:
                    fil=File.OpenRead("huse.xml");
                    liste=(ArrayList)soap.Deserialize(fil);
                    fil.Close();
                    Console.WriteLine("Soap Formatter: Data fra xml filen:");
                    foreach(Hus h in liste){
                                Console.WriteLine("Hus: {0} - {1}",h.getKode(),h.getAar());
                    }

                    //ANDEN METODE: BINARY FORMATTER:
                    //serialiserer listen i binær fil - som bytes:

                    fil=File.Create("huse.dat");
                    BinaryFormatter bin=new BinaryFormatter();
                    bin.Serialize(fil,liste);
                    fil.Close();

                    fil=File.OpenRead("huse.dat");
                    liste=(ArrayList)bin.Deserialize(fil);
                    fil.Close();
                    Console.WriteLine("Binary Formatter: Data fra dat filen:");

                    foreach(Hus h in liste){
                                Console.WriteLine("Hus: {0} - {1}",h.getKode(),h.getAar());
                    }


          }
}
```

Som det ses er det rimeligt nemt at gemme fx en liste over objekter. Den centrale kode (i eksemplet med SOAP) i programmet er:

```
FileStream fil=File.Create("huse.xml");
SoapFormatter soap=new SoapFormatter();
soap.Serialize(fil,liste);
fil.Close();
```

Hele listen gemmes med formlen: enformatter.**Serialize**(). Om man vælger den ene eller anden metode er for så vidt ligegyldigt – men SOAP eller XML formatteringen giver – som vi skal se – mange fordele. FX kan en sådan SOAP formatteringen let gemmes på Internettet og hentes ned gennem en almindelig browser. .NET har derfor satset hårdt på at XML formatet bliver dominerende. Når objekterne skal deserialiseres skal de castes gennem den klasse de blev gemt igennem her: ArrayList for at filens indhold kan lægges ind i en ny ArrayList:

```
fil=File.OpenRead("huse.xml");
liste=(ArrayList)soap.Deserialize(fil);
fil.Close();
Console.WriteLine("Soap Formatter: Data fra xml filen:");
foreach(Hus h in liste){
        Console.WriteLine("Hus: {0} - {1}",h.getKode(),h.getAar());
}
```

I den XML fil som her bliver resultatet kan man se at alle objekternes referencer ('#ref 4' osv) gemmes og at objektets meta-data også gemmes. Fx kan nedenfor ses at programmets version er '1.0.1090.20971'. XML filen rummer en **fuldstændig dokumentation** af objekterne:

```
    <_items href="#ref-2" />
    <_size>3</_size>
    <_version>3</_version>
  </a1:ArrayList>
- <SOAP-ENC:Array id="ref-2" SOAP-ENC:arrayType="xsd:anyType[16]">
    <item href="#ref-3" />
    <item href="#ref-4" />
    <item href="#ref-5" />
  </SOAP-ENC:Array>
- <a3:Hus id="ref-3" xmlns:a3="http://schemas.microsoft.com/clr/assem/binfiler%2C%20Version%
    3D1.0.1090.20971%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
    <kode id="ref-7">abc-22</kode>
    <byggeaar>1977</byggeaar>
  </a3:Hus>
- <a3:Hus id="ref-4" xmlns:a3="http://schemas.microsoft.com/clr/assem/binfiler%2C%20Version%
    3D1.0.1090.20971%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
    <kode id="ref-8">xz-1234</kode>
    <byggeaar>1999</byggeaar>
```

Programmet udskriver desuden til skærmen:

```
Soap Formatter: Data fra xml filen:
Hus: abc-22 - 1977
Hus: xz-1234 - 1999
Hus: ab-77 - 2001
Binary Formatter: Data fra dat filen:
Hus: abc-22 - 1977
Hus: xz-1234 - 1999
Hus: ab-77 - 2001
```

På http://csharpkursus.subnet.dk ligger et C# program som viser hvordan man kan serialisere en Windows form, dens tekster, titler, billeder, størrelse osv. I dette eksempel serialiseres også med SOAP til en XML fil. En ny form kan så hente sine data fra XML filen. Dette svarer til at et program i C# kan konfigureres fra en XML fil.

Opgaver:

1. Skriv et program som erklærer en klasse **Trekant** og opretter en række af trekanter og serialiserer dem i XML.

2. Opret en **telefonliste** og serialiser den som en *.dat fil
3. Skriv et program som **deserialiserer** fra huse.xml som ligger på adressen: http://csharpkursus.subnet.dk/huse.xml (Desværre kan man ikke i dette tilfælde direkte deserialisere fra internet adressen – man bliver nødt til først at downloade filen!).

.NET eller DotNet – Hvad er det egentligt?

Efter at vi foreløbigt har set eksempler på objekt orienteret programmering vil vi i dette afsnit se mere detaljeret på hvad Microsoft .NET eller 'DotNet' egentligt er for en størrelse.
Vi vil ikke her gå ned i hver eneste detalje, men se på så mange træk i .NET at man får en vis grundlæggende forståelse for denne 'arkitektur'.

Helt overordnet består DotNet af: CLR - en **Common Language Runtime** (som muliggør at forskellige sprogs programmer kan køre – fx C# eller Visual Basic) - en samling af **basisklasser** (flere end 1000 i alt), en definition af fælles typer (int, float, char …) i **Common Type System** (CTS) og en definition af minimums krav til alle de sprog som vil programmere til .NET nemlig **Common Language Specification** (CLS). Nogle af de ting som fx er OK i C# er ikke OK ifølge CLS. Hvis jeg ønsker at fastholde at min kode skal holde sig til det fælles minimum kan jeg forsyne min kode med en attribut således:

```
[CLSCompliant]
public class X{}
```

DotNet er populært sagt ligeglad med hvad programmer i forskellige sprog som Visual Basic eller JScript laver **internt** hvis de blot **eksternt** overholder bestemte krav. Som i objektorienteret programmering opfattes klassen som en 'black box'!

NB C# har IKKE sit eget klasse bibliotek, men bruger direkte basisklasserne (BCL) i .NET.

C# er decideret designet til at fungere sammen med .NET.

DotNet består helt centralt af 2 DLL filer: **mscorlib**.dll som rummer alle de centrale klasser og metoder og **mscoree**.dll som er den 'exe-maskine' (.NET execution engine) som afvikler programmet i .NET.

Ideen med .NET ligner ideen bag sproget **Java** som opstod i midten af 1990'erne. Et Java program skrives som en tekstfil (lige som et C# program) og kompileres af en Java kompiler.
Resultatet af Java kompileringen er en såkaldt **byte kode**. En fil 'Mitprogram.java' kompileres til en byte kode 'Mitprogram.class'. Denne class fil kan ikke uden videre køre fx på en Windows maskine, men skal oversættes en gang til 'native' kode som Windows kan forstå. Men det gode ved Java løsningen er, at den **samme** Java kode kan køre på **alle** operativ systemer og **alle** processorer (CPU'er). Det er altså muligt at skrive **eet** program som kan køre på en Windows 98, Mac eller en Linux maskine!

Forudsætningen er at den pågældende maskine har installeret 'Java Virtual Machine' som er et lag som lægges **oven på** maskinens operativ system (styre system).

IL – Intermediate Language:

Microsoft .NET fra år 2000 bygger på samme princip. En C# fil 'Nyt_program.cs' kompileres i virkeligheden **ikke** til en færdig EXE fil (som man kunne få indtryk af!) – men til en mellemliggende kode i sproget **Microsoft Intermediate Language** (kaldet **IL**). Denne IL kode – en såkaldt **'assembly'** - består af koder som er uafhængige af platform (operativ system) og programmeringssprog. IL er - i .NET jargonen - 'platforms/sprog agnostisk'.

IL består af assembler koder som er abstrakte og som denne konkrete maskine/CPU ikke kan forstå. (En CPU eller processor som fx Intel forstår kun et **bestemt** sæt af simple instruktioner som fx add eller gem i stacken). En IL kode er altså **ikke** 'maskin kode' (kode som CPU'en direkte kan udføre). IL svarer til Java byte kode – men den store forskel er at IL kode også er tvær-**sprogligt**. Dvs. at en klasse eller en funktion skrevet i fx Visual Basic og en klasse skrevet i C# resulterer i den **samme** IL kode!! Vi skal straks se et eksempel herpå. I .NET kan man altså programmere i mange forskellige sprog på kryds og tværs efter smag og behag! (Det kan man selvfølgelig ikke i Java).

Der er skrevet kompilere til 20-30 forskellige sprog og alle disse kan altså programmere til .NET!

Et **eksempel** på Microsoft **IL** kode kunne være følgende, som kan kompileres til en DLL fil sådan:

ilasm /DLL IL2.il

```
// En klasse med en constructor og en public metode:

//ilasm /DLL IL2.il

//Dette er egentligt overfloedigt - sker automatisk!:
.assembly extern mscorlib{}

//Vi bestemmer selv versionen:
.assembly IL2 {.ver 1:0:1:0}

//Module er fil navnet:
.module IL2.dll

//Opret en klasse 'Y':

.class public auto ansi Y extends [mscorlib]System.Object{

    .method public static string Message() cil managed{

        //push paa stacken:
        ldstr "Dette er en enkelt linje fra IL metoden Message()!"

        //pop fra stacken:
        ret
    }

    //NB klassen skal have en constructor for at kunne instantieres:

    .method public specialname rtspecialname instance void .ctor() cil managed{
```

```
        .maxstack 1

        //argument 0 er 'this'
        ldarg.0

        //kald basis klassens constructor:
        call instance void [mscorlib]System.Object::.ctor()
        ret
    }

}
```

Denne IL kode minder en del om C# kode. Der oprettes en assembly med en version, et modul, en klasse og en public metode som returnerer en streng! Klassen har også en constructor så den ville kunne instantieres – fx i et C# program – med new Y()!

Som det ses er IL koden noget mere besværlig og langtrukken end C# kode.

Fordelen ved at arbejde med IL kode er bl.a.:

1. Det er muligt at skrive mere effektiv og hurtigere kode i IL
2. Forståelse for IL koden giver en bedre forståelse for C# og .NET!

Det væsentlige er at uanset om klasser og metoder oprindeligt er skrevet i C#, VB, C++ eller JScript, er resultatet altid den **samme** assembler kode (IL kode) – eller næsten den samme kode!

Managed/Unmanaged kode:

Kode skrevet i C#, Jscript eller Visual Basic kaldes i jargon'en '**managed'** kode – dvs kode som umiddelbart forstås af .NET Runtime.

Gamle DLL filer i Windows (typisk skrevet i C) er derimod **'unmanaged'** kode, og de kan kun bruges hvis der etableres et mellemled (fx med en **InterOp** i .NET).

Vi skal se et eksempel på at et C# program **direkte** anvender klasser og metoder skrevet i JScript og VB.

En klasse i C# kan også direkte **arve** fra en Jscript eller VB klasse!

COM:

Den 'traditionelle' Windows kode kaldes COM – **Common Object Model**. DotNet er et forsøg på at løse problemer i COM. Der er mange forskelle på COM og .NET – her er nogle eksempler:

1. COM har ikke et fælles type system

2. COM klasser anvender meget specifikke og ejendommelige typer som BSTR, VARIANT osv som ikke passer godt sammen med en bred vifte af sprog
3. COM bygger stadig på programmeringssproget C i høj grad – og er derfor ikke altid lige Objekt Orienteret!
4. Alle COM klasser skal registreres i Windows System Registrerings Databasen
5. COM har ikke et fælles hjem (mappe) som .NETs Globale Cache.
6. COM klasser allokeres ikke på heapen men i programmets stack
7. COM klasser har ingen meta data og er ikke selv beskrivende
8. COM består af CPU afhængig maskinkode – modsat IL
9. I COM kan der ikke være 2 eller flere DLL filer med samme navn men med forskellige versioner eller kulturer
10. COM klasser identificeres med en CLSID (en række af HEX tal)
11. Hvis en COM klasse flyttes – bryder systemet sammen og den skal registreres igen
12. COM klasser er svære at bruge i distribuerede systemer/over netværk
13. COM klasser kan ikke køre på noget andet styresystem end Windows

JIT kompiler:

IL koden bliver oversat (kompileret) af en Just In Time kompiler (**JIT** kompiler) for at programmet skal kunne køre på denne konkrete maskine. (Akkurat som hvis vi havde haft en Java class fil).

Først nu opstår en EXE fil (maskin kode), der kan fx kan køre i Windows 98 helt konkret. Første gang dette sker tager det længere **tid** at opstarte programmet – men hvis samme program køres igen er maskinkoden (EXE filen) gemt og C# programmet kører – stort set - lige så hurtigt som fx et C++ program.

IFølge Microsoft vil .NET programmer på længere sigt komme til at køre **hurtigere** end traditionel Windows kode (fordi JIT kompileren bedre kan optimere koden) – men ifølge de **flestes** opfattelse kører .NET programmer for øjeblikket 5-10 % langsommere!

Man kan producere et 'native image' af sine C# programmer med **ngen**, som findes i .NET mappen i Windows. Teoretisk skulle det producere en 'native' EXE der så skulle kunne starte jurtigere. ngen kompilerer koden til almindelig Windows kode.

Microsofts ide er at IL koden skal kunne afvikles på **alle** platforms også på en Linux eller Mac maskine. (Dette er allerede implementeret på forsøgsniveau med Linux).

En EXE eller DLL er ikke en EXE eller DLL!

For at alt dette skal kunne lade sig gøre skal Microsofts .NET selvfølgelig **installeres** på maskinen (svarende til Javas virtuelle maskine). Når du har kompileret et program i C# og fået en EXE fil som kører fint på din maskine – kan du prøve at køre EXE filen på din nabos maskine. Du vil så opdage at resultatet er en sværm af **fejl** meddelelser!!

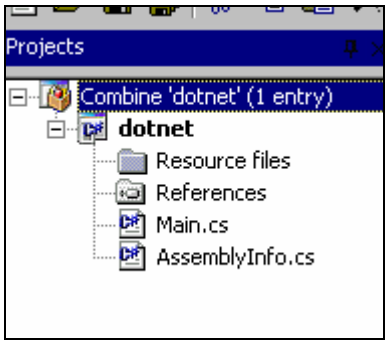
Det som ser ud som en ganske 'almindelig' exe fil på **din** maskine er i virkeligheden **IKKE** en 'almindelig' Windows exe fil – for inde hos naboen (som ikke har .NET) er der **intet** af det som

virker! (Dette er den afgørende forskel i forhold til hvis du havde skrevet i program fx i C – for den exe **ville** have virket inde hos naboen!).

Eksempel: Projekt i SharpDevelop:

Vi vil nu se på et konkret eksempel på hvad DotNet egentligt er for noget:

Åbn et nyt konsol projekt i SharpDevelop ved navn 'dotnet'. Vælg View->Projects. (NB ved at højre klikke på de forskellige objekter fås genveje).



Et projekt (eller 'Combine' der kan bestå af mange projekter og filer) består fra starten af to **CS** filer fra starten: Main.cs og en AssemblyInfo.cs (som også kompileres når programmet kompileres).

AssemblyInfo:

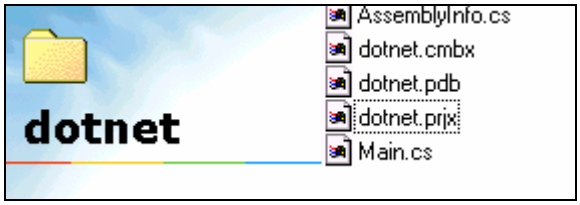
Assembly Info består af en række **'attributter'** som er tomme fra starten, men som kan sættes til at beskrive denne assembly:

```
using System.Reflection;
using System.Runtime.CompilerServices;

// Information about this assembly is defined by the following
// attributes.
//
// change them to the information which is associated with the assembly
// you compile.

[assembly: AssemblyTitle("")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("")]
[assembly: AssemblyCopyright("")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

// The assembly version has following format :
//
// Major.Minor.Build.Revision
```

```
//
// You can specify all values by your own or you can build default build and revision
// numbers with the '*' character (the default):

[assembly: AssemblyVersion("1.0.*")]

// The following attributes specify the key for the sign of your assembly. See the
// .NET Framework documentation for more information about signing.
// This is not required, if you don't want signing let these attributes like they're.
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile("")]
```

Enhver assembly har altså en Version, der består af 4 tal fx 2.4.111.222 – heraf er det første tal (2) Major (dvs hovedversion 2) og det andet tal Minor (underversion 4). De 2 sidste tal står for build nummeret og revisions nummeret. Hvis jeg opgraderer min assembly retter jeg kun i de to første tal. I modsætning til det gamle Windows COM system er det altså let at have flere udgaver/versioner af den samme assembly på maskinen. Modsat COM skal .NET klasser IKKE registreres i System registrerings Databasen i Windows. DotNet arkitekturen er på den måde langt mere fleksibel og nemmere at administrere.

Der oprettes følgende **filer** i mappen 'dotnet' (hvis du har gemt projektet i en mappe 'dotnet'):

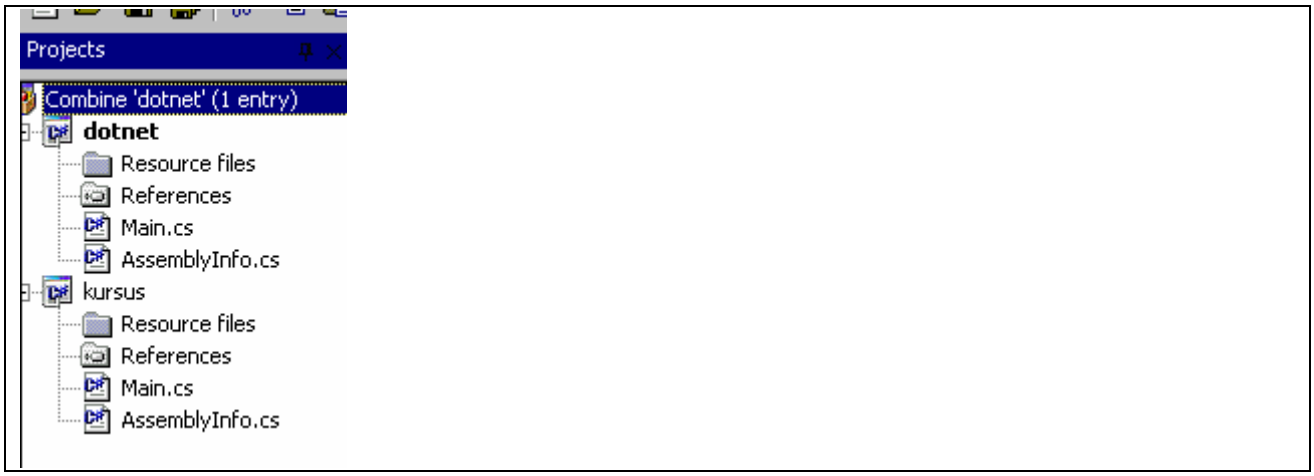


CMBX filen er en XML fil som indeholder en række konfigurations oplysninger om det samlede projekt/Combine (dotnet).
PRJX filen er en tilsvarende XML fil for det 'under projekt' dotnet som findes inden i combine dotnet (!). Vi vil vende tilbage til **konfigurations** filer senere.

Eksempel fra **PRJX** filen:

```
<Project name="dotnet" description="creates a console C# project" newfilesearch="None" enableviewstate="True"
version="1.1" projecttype="C#">
  <Contents>
    <File name=".\Main.cs" subtype="Code" buildaction="Compile" dependson="" data="" />
    <File name=".\AssemblyInfo.cs" subtype="Code" buildaction="Compile" dependson="" data="" />
  </Contents>

…

</Project>
```

Tilføj et **nyt** projekt til Dotnet ved at højre klikke Combine og vælge Add new projekt. Opret et nyt konsol projekt kursus. Dette tilføjes så som en del af det større projekt/Combine:



Slet indholdet i den automatisk oprettede Main.cs og skriv i stedet følgende **klasse** erklæring i Main.cs i det nye projekt **kursus**:

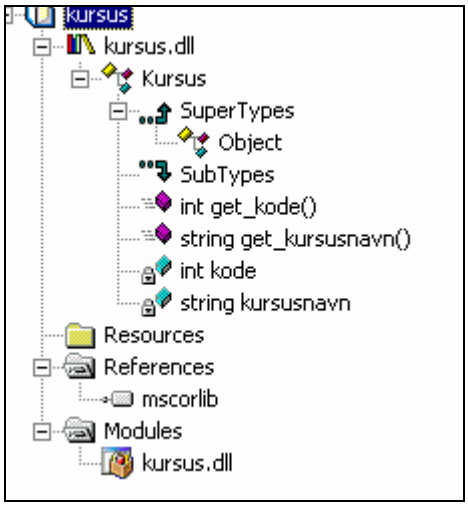
```
//dotnet demo klassen Kursus kompileres som DLL/Library

using System;

public class Kursus{

private int kode;
private string kursusnavn;

public Kursus(int k,string s){kode=k;kursusnavn=s;}

public int get_kode(){return kode;}
public string get_kursusnavn(){return kursusnavn;}
}
```

Vælg at kompilere kursus som Library (altså DLL fil) ved at højre klikke 'kursus' vælge Options – vælge Configuration og sæt target til Library. I stedet for at gøre dette i SharpDevelop kan du også gøre dette manuelt som tidligere vist – ved at kompilere filen Main.cs i kursus sådan:

csc /t:library /out:kursus.dll Main.cs

Nu oprettes en ny fil **kursus.dll**, en binær fil der rummer vores nye klasse. (Du kunne også forinden **omdøbe** Main.cs til det mere passende navn: kursus.cs så slipper du for /out:kursus.dll).

Klassen Kursus:



Du kan nu redigere i filen kursus AssemblyInfo og gemme data om denne klasse – fx:

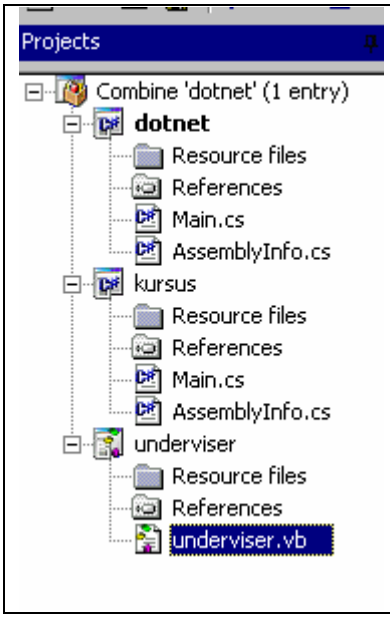
```
[assembly: AssemblyTitle("Kursus")]
[assembly: AssemblyDescription("Klassen Kursus til registrering af kurser med ID og tekst")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("")]
[assembly: AssemblyCopyright("Computer APS 2002")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("DK")]
```

Disse data er fra nu af en integreret del af selve DLL filen – de kompileres sammen med selve klassen. På denne måde kan man i .NET og C# have forskellige udgaver af den samme DLL fil på samme maskine. Systemet kan skelne mellem forskellige udgaver ud fra deres attributter fx Culture! En assembly er 'selvbeskrivende'.

En klasse i Visual Basic:

Vi vil nu se et lille eksempel på hvordan .NET kan håndtere kode på kryds og tværs af programmeringssprog – vi vil tilføje en ny klasse til dotnet projektet skrevet i Visual Basic:

Til føj et nyt projekt til dotnet – Visual Basic/VBNet – og tilføj en ny fil underviser.vb til projektet. Vores projekt ser nu således ud i SharpDevelop:



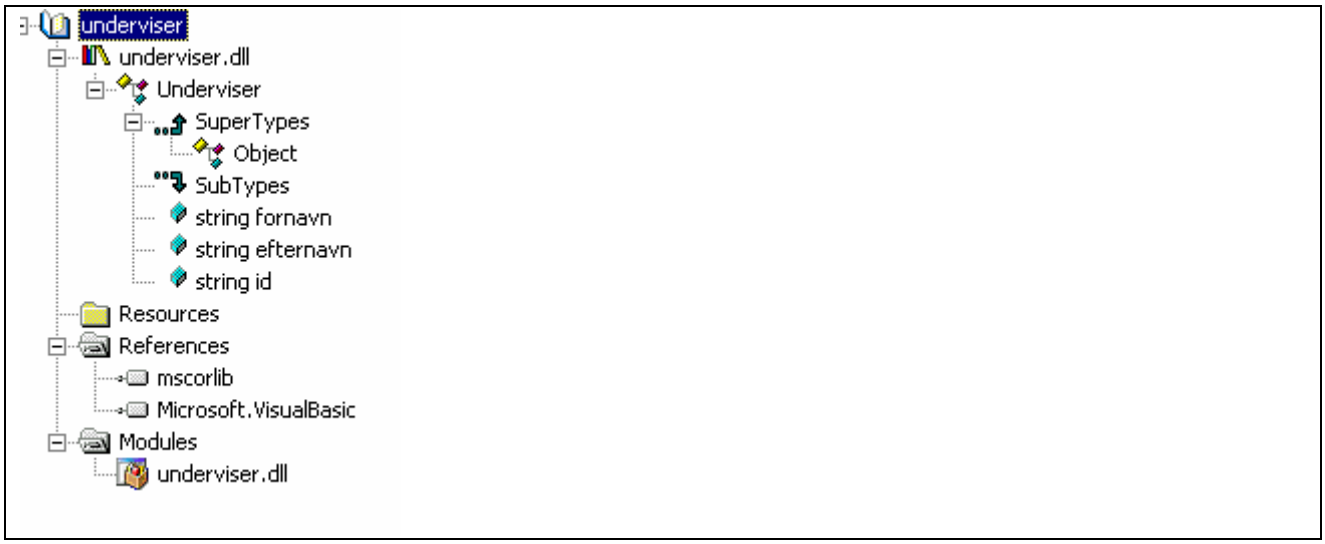
Skriv følgende Visual Basic kode i underviser.vb:

```
'VB klasse Underviser til brug i demo af dotnet:
'


public class Underviser
                'opret 3 public felter (det nemmeste - kun derfor lige nu!):

                public fornavn as string
                public efternavn as string
                public id as string

End Class
```

Sæt Options i underviser til at output bliver en DLL fil – lige som ovenfor i eksemplet kursus! Når underviser kompileres fås underviser.dll.

Vi vil ikke her gå nærmere ind på koden i Visual Basic. Som det ses er Visual Basic på mange måder et simplere sprog end fx C#. Der er ikke de samme strenge krav til syntaks i Visual Basic som i C#. (Tegnet ’ markerer en kommentar i VB).

Klassen Underviser:



Klasse i JScript:

Vi vil – for eksemplets skyld! – også skrive en klasse i Microsoft Jscript:
Skriv blot denne klasse erklæring i en ny fil **pris.js**:

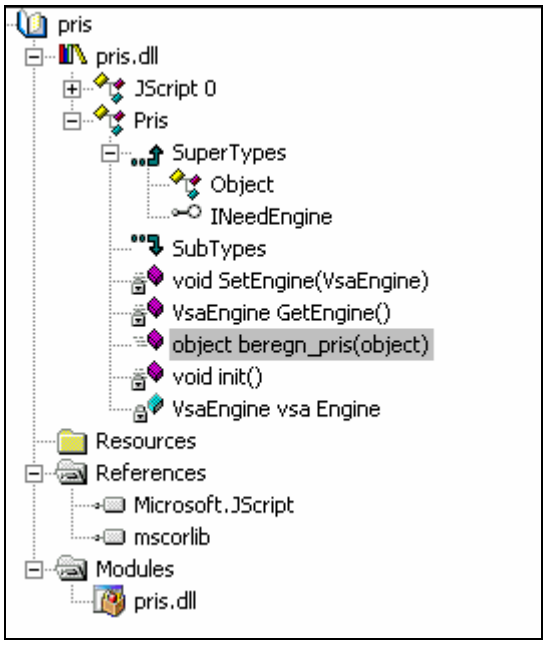
```
//JScript klasse Pris til demo projekt Dotnet:

//beregn_pris kaldes som static metode direket på klassen: Pris.beregn_pris():

public class Pris{

public static function beregn_pris(tal){
return tal*500;
}

}
```

JS filen kompileres med:

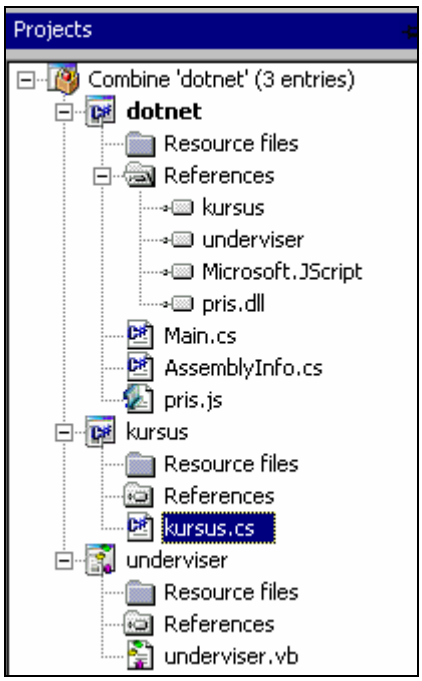
jsc /t:library pris.js

Funktionen er en ren demo funktion som 'beregner' en kursus pris blot ved at gange et tal med 500!

Klassen Pris:
(NB som det ses arver JScript klassen Pris også fra Object!):



jsc er .NET's jscript kompiler. Resultatet er en pris.dll fil. Tilføj filen pris.js til projektet dotnet og tilføj en reference til **alle** dll filerne således:



dotnet refererer altså til de 3 klasse filer: **kursus**.dll (en C# klasse), **underviser**.dll (en VB klasse) og **pris**.dll (en JScript klasse) – og desuden til Microsoft.Jscript (hvilket er nødvendigt for at anvende Jscript).

Vi vil nu vise hvordan de 3 'fremmede' klasser kan anvendes:

En .NET applikation der bruger klasserne:

Omskriv **Main**.cs i projektet **dotnet** (hovedprojektet) som vist i følgende eksempel – den nye kode er forklaret i kommentar linjerne:

```
//DEMO af DotNet
//klasser fra VB, JScript og C#:


using System;

//Der erklæres en ny klasse der anvender de oprettede klasser ('containment' eller 'har en' relation):
public class Kursussted{

		//Klassen Kursus er en C# klasse defineret i kursus.dll:
		private Kursus[] kurser;

		//Klassen Underviser er en VIsual Basic klasse defineret i underviser.dll:
		private Underviser[] undervisere;

		//Klassen Kursussted er en C# klasse defineret her:
		public Kursussted(Kursus[]k,Underviser[]u){
				kurser=k;
				undervisere=u;
		}
		public string vis_kurser(){
				string s=null;
				for(int i=0;i<kurser.Length;i++){
						s+="KURSUS: "+kurser[i].get_kode()+" :
"+kurser[i].get_kursusnavn()+"\n";
				}
				return s;
		}
		public string vis_undervisere(){
				string s=null;
				for(int i=0;i<undervisere.Length;i++){
						s+="UNDERVISER: "+undervisere[i].efternavn+",
"+undervisere[i].fornavn+" ("+undervisere[i].id+")"+"\n";
				}
				return s;
		}



}

//Den endelige applikation:
class app
{
		public static void Main(string[] args)
		{
				Kursus k1=new Kursus(1,"Internet");
				Kursus k2=new Kursus(2,"Regneark");
				Kursus k3=new Kursus(4,"Email   ");
				Kursus k4=new Kursus(5,"Database");
```

```
                Kursus[] k=new Kursus[4];
                k[0]=k1;
                k[1]=k2;
                k[2]=k3;
                k[3]=k4;

                Underviser u1=new Underviser();
                u1.fornavn="Jens";
                u1.efternavn="Jensen";
                u1.id="jj";

                Underviser u2=new Underviser();
                u2.fornavn="Lise";
                u2.efternavn="Jensen";
                u2.id="lj";

                Underviser[] u=new Underviser[2];
                u[0]=u1;
                u[1]=u2;

                Kursussted ks=new Kursussted(k,u);

                Console.WriteLine("DotNet DEMO:\n\n{0}",ks.vis_kurser());
                Console.WriteLine("DotNet DEMO:\n\n{0}",ks.vis_undervisere());
                for(int i=0;i<k.Length;i++){

                        //Klassen Pris er en jScript klasse defineret i pris.dll:
                        //anvend static metode i JScript klassen Pris:

                        //NB: JScript kræver en reference til Microsoft.JScript:

                        Console.WriteLine("Kursus:
\t{0}\t{1:C}",k[i].get_kursusnavn(),Pris.beregn_pris(k[i].get_kode()));


                }

                Console.Read();
        }
}
```

NB: Forudsætningen for at applikationen kan køre er (i al fald lige nu) at de refererede dll filer (underviser.dll osv) **SKAL** ligge i **samme** mappe som applikationen!! Koden kan godt **kompileres** uden at dette er tilfældet (!!), men IKKE **køre**!

Flyt derfor de omtalte DLL filer så de ligger sammen med dotnet.exe (applikationen)!

Når programmet afvikles fås følgende resultat:

```
C:\csharp\dotnet>dotnet
DotNet DEMO:

KURSUS: 1 : Internet
KURSUS: 2 : Regneark
KURSUS: 4 : Email
KURSUS: 5 : Database

DotNet DEMO:

UNDERVISER: Jensen, Jens (jj)
UNDERVISER: Jensen, Lise (lj)

Kursus:         Internet        kr 500,00
Kursus:         Regneark        kr 1.000,00
Kursus:         Email           kr 2.000,00
Kursus:         Database        kr 2.500,00
```

Kravet om at DLL filerne (underviser.dll osv) SKAL ligge i samme mappe som applikationen for at .NET Runtime 'resolveren' kan finde DLL filerne gælder med visse undtagelser:

Hvis kursus.dll og underviser.dll fx er anbragt i **undermapper** med de **rette** navne (kursus.dll skal ligge i undermappen kursus osv) - behøver de ikke længere at ligge i samme mappe som applikationen f.eks sådan:

Nu kan .NET 'resolve' referencerne – altså finde DLL filerne!

XML config-filer:

En anden mulighed er at anvende en **konfigurations** fil – som vi tidligere så eksempler på:

Skriv følgende **XML** kode i en fil der ALTID skal kaldes efter applikationen plus '.config' – i dette tilfælde **SKAL** config filen altså hedde **dotnet.exe.config**:

```
<configuration>
 <runtime>
  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
                <probing privatePath="dllfiler">
                </probing>
  </assemblyBinding>
 </runtime>
</configuration>
```

XML filen dotnet.exe.config angiver i hvilken **undermappe** (her er undermappen kaldet: 'dllfiler') hvor Runtime skal kigge efter DLL filer. Med denne konfiguration kan alle nødvendige DLL filer så placeres i dotnet\dllfiler:

Konfigurationsfilen **skal** ligge i samme mappe som exe filen!

Mange forskellige undermapper kan angives i linjen '<probing …' ved at skrive mapperne med semikolon imellem! (Runtime kan **kun** søge i undermapper – en sti som 'C:\dokumenter' er ikke gyldig!).

En config fil kan indeholde mange forskellige parametre for programmets opstart. I filen **machine.config (**i mappen Microsoft.NET\Framework\v1.0.3705) defineres en lang række konstanter for hele DotNet.

I **security.config** en række forhold omkring .NET og security.

NB det er **ikke** tilstrækkeligt blot at kopiere DLL filer til den mappe hvor de øvrige kerne .NET dll filer ligger (fx C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705)! I det normale tilfælde hvor der arbejdes med 'private' assemblies har dette ingen værdi eller betydning!!

Vi vil til sidst i dette afsnit om .NET se lidt på de filer vi omtalte i starten:
Filen **dotnet.prjx** indeholder en række data om projektet fx:

```
<Project name="dotnet" description="creates a console C# project" newfilesearch="None" enableviewstate="True"
version="1.1" projecttype="C#">
  <Contents>
    <File name=".\Main.cs" subtype="Code" buildaction="Compile" dependson="" data="" />
    <File name=".\AssemblyInfo.cs" subtype="Code" buildaction="Compile" dependson="" data="" />
    <File name=".\pris.js" subtype="Code" buildaction="Nothing" dependson="" data="" />
  </Contents>

  <References>
    <Reference type="Project" refto="kursus" />
    <Reference type="Project" refto="underviser" />
    <Reference type="Gac" refto="Microsoft.JScript, Version=7.0.3300.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a" />
    <Reference type="Assembly" refto=".\pris.dll" />
  </References>
```

Det ses, at projektet nu består 3 filer Main, AssemblyInfo og pris.js. Projektet har 4 referencer nemlig til 2 af vores projekter kursus.dll og underviser.dll og en ren assembly reference pris.dll (JScript er jo ikke et projekt).

Desuden refererer projektet til Microsoft.JScript.dll som findes i "GAC". **GAC** – den globale assembly cache – er samlingen af kerne .NET assemblies som System.dll, System.IO.dll osv.

Det er muligt selv at skrive assemblies og gemme dem globalt i GAC så de får samme status som de oprindelige kerne klasser. Det normale er dog at anvende 'private' assemblies som kun gemmes i forbindelse med en applikation.

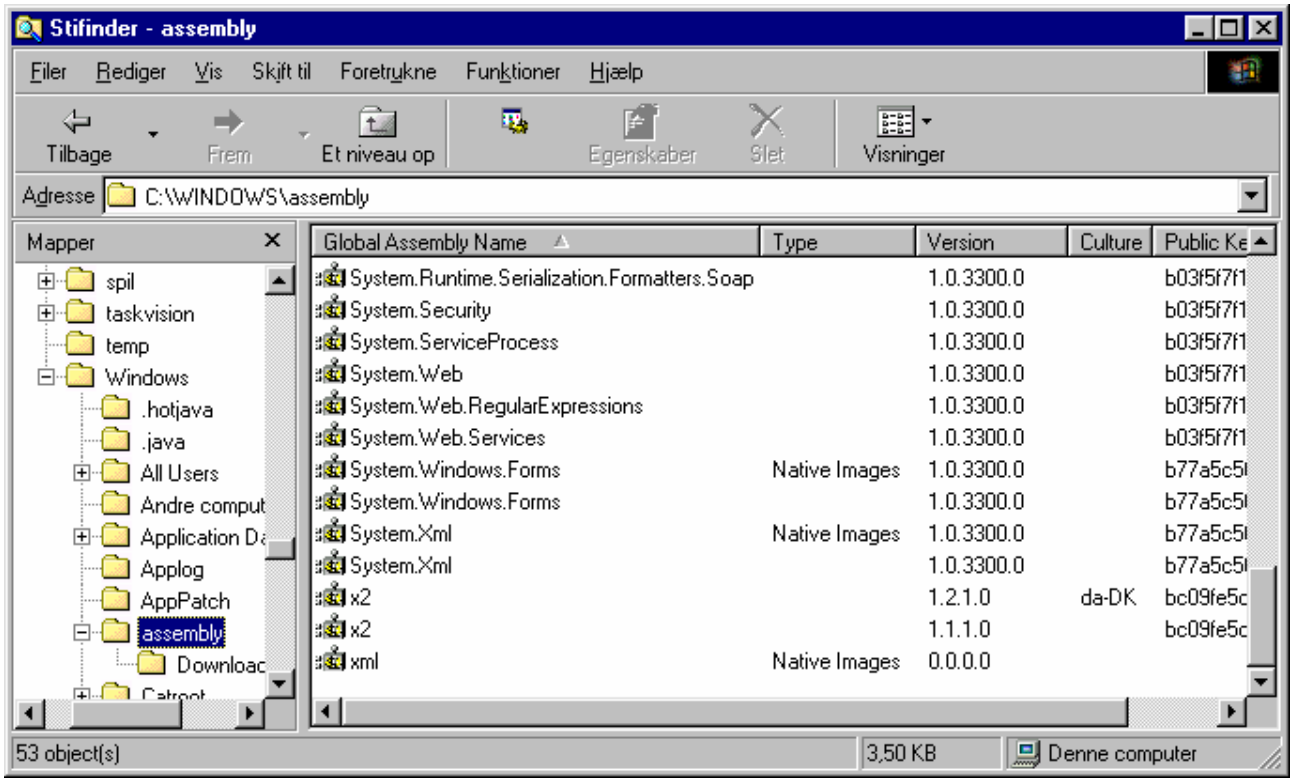
Som det ses kræver en global assembly at den kodes med en public **key** (et såkaldt **'strong name'** som ID**).**

På http://csharpkursus.subnet.dk ligger dels et eksempel på en 'strong name' fil (snk fil) som kan bruges til at signere en assembly dels et lille eksempel på hvordan en assembly kan installleres i GAC med version og kultur(det nemmeste er i øvrigt at anvende drag-and-drop/kopier – sæt ind i mappen C:\Windows\Assembly og i .NET mappen i Windows).

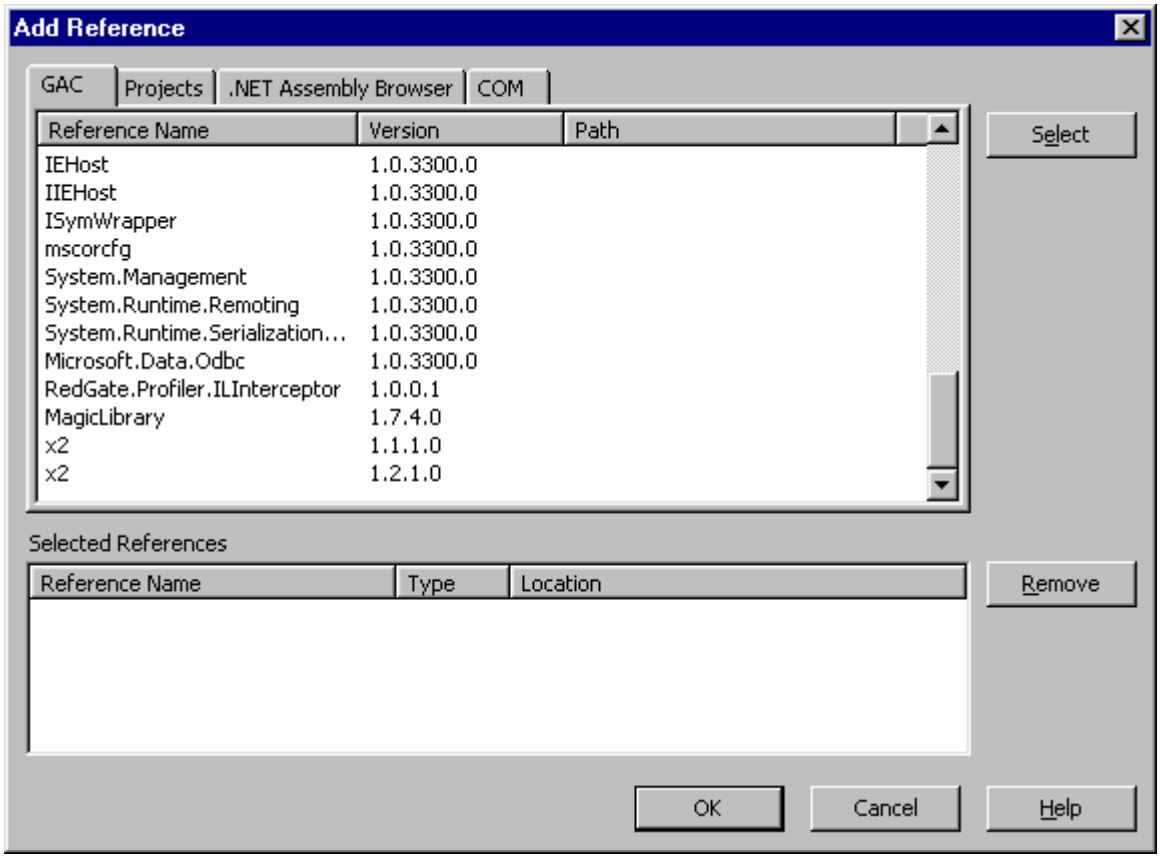
Når en assembly er installeret i Cachen kan den refereres **globalt** – den skal altså ikke ligge i applikations mappen og skal ikke kopieres til applikations mappen.

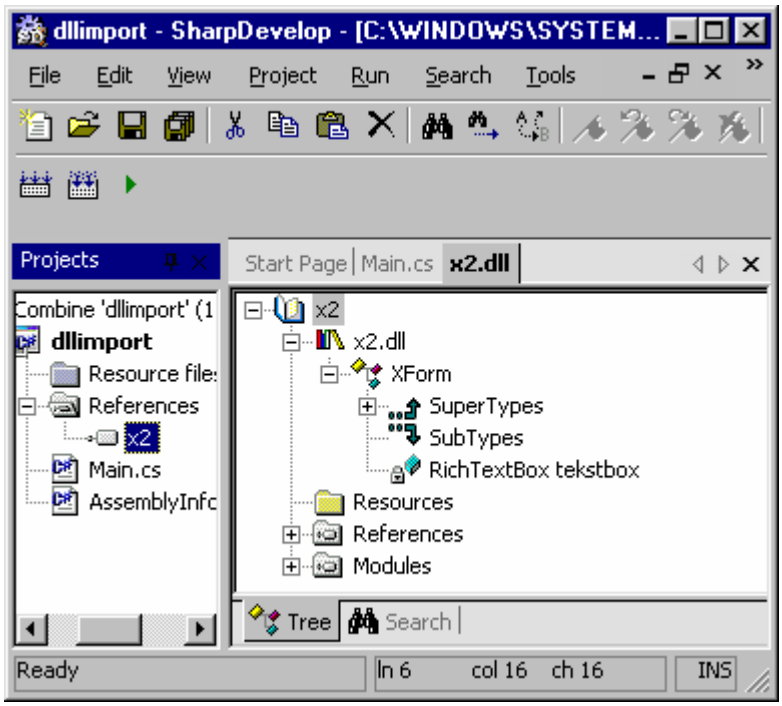
I GAC kan ligge **flere** versioner af den samme DLL og forskellige 'kultur (sprog)' udgaver af den samme assembly.

Et eksempel på den samme assembly x2 i to udgaver installeret i den globale cache:



Et projekt i Sharpdevelop kan så referere til x2 via GAC:





Vi vender nu tilbage til det tidligere DotNet eksempel:

NB når Version er 1.1 skyldes det vi bevidst **satte** den til den værdi tidligere. Version har vi altså selv fuld kontrol over.

AssemblyInfo.cs ser – bortset fra kommentarer - sådan ud:

```
 [assembly: AssemblyTitle("DOTNET")]
[assembly: AssemblyDescription("Demo om DOTNET")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("")]
[assembly: AssemblyCopyright("Computer Aps 2002")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

[assembly: AssemblyVersion("1.1.*")]

 [assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile("")]
```

Hvis denne assembly skulle have været gemt som global assembly (i GAC) skulle der i nederste linje have været en fil med en public key.

Denne * som findes i Version markerer at de 2 sidste tal (Build og Revision) automatisk opdateres af Runtime når vi reviderer og builder/kompiler filen. Dette kan også konstateres hvis vi undersøger filens egenskaber fx i Windows Stifinder!

To versioner af en assembly med de samme Major og Minor tal betragtes som **kompatible**, også selv om de 2 sidste tal ikke er identiske. (Forudsat er her at de har den samme kultur – ellers er de IKKE kompatible!).

AssemblyInfo filen til klassen **Kursus** ser sådan ud:

```
[assembly: AssemblyTitle("Kursus")]
[assembly: AssemblyDescription("Klassen Kursus til registrering af kurser")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("")]
[assembly: AssemblyCopyright("Computer Aps 2002")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("DK")]
[assembly: AssemblyVersion("1.1.*")]
```

At producere IL kode dynamisk:

Som sagt er EXE og DLL filer i .NET egentlig **ikke** 'exe' eller 'dll' filer som de normalt forstås i Windows. De er IL (Intermediate Language) tekst filer som består af **assembler** instruktioner.

For bedre at forstå det vil vi se på følgende kode eksempel som viser hvordan man kan producere en assembly og en ny klasse **dynamisk** eller 'run time'. Dette sker ved at bruge klasser og metoder i det namespace som hedder **System**.**Reflection**.**Emit** (i mscorlib.dll).

Den assembly vi producerer er egentligt dynamisk (dvs den eksistrerer kun i RAM), men vi vil gemme den i en fil – for at kunne bruge og vise den bagefter.

De metoder som vises her bruges af builder værktøjer som Visual Studio og SharpDevelop!

Følgende C# kode producerer en dynamisk assembly og gemmer den i en fil NyAssembly.dll:
OBS Koden er forklaret løbende:

```
//Eksempel paa en dynamisk assembly
//Programmet er et slags 'builder' program:
//JScript i .NET fungerer efter disse metoder
//naar JScript builder/skaber en klasse:
//Med System.Reflection.Emit kan man skabe sine klasser i ren IL kode med metoden Emit():

using System;
using System.Reflection;
using System.Reflection.Emit;
using System.Threading;

class MainClass
{
		public static void Main(string[] args)
		{


				//Opret ny assembly der skal rumme en klasse:
				AssemblyName ny_assembly=new AssemblyName();
				ny_assembly.Name="NyAssembly";
				ny_assembly.Version=new Version("1.0.0.0");

				//Den centrale klasse er AssemblyBuilder som skaber den nye klasse:
				AppDomain domain=Thread.GetDomain();
				AssemblyBuilder assembly_builder = domain.DefineDynamicAssembly
(ny_assembly,AssemblyBuilderAccess.Save);

				//En modul builder opretter det nødvendige modul
				//Een assembly kan bestaa af mange moduler - men her kun et:
				ModuleBuilder modul_builder = assembly_builder.DefineDynamicModule
("NyAssembly","NyAssembly.dll");

				//Dette er minimum - der skal oprettes mindst en klasse i hvert modul:
				//En TypeBuilder skal anvendes. Dette er en public class:
				//Metoden nedenfor (MethodBuilder) kunne altså droppes - og alt ville fungere:
				TypeBuilder klasse_builder = modul_builder.DefineType
("NyAssembly.Nyklasse",TypeAttributes.Public);

				//Eneste metode i klassen Nyklasse:
				//En metode oprettes med en MethodBuilder
				//Flere metoder kunne skabes blot ved at gentage disse 4 linjer!:

				MethodBuilder metode_builder = klasse_builder.DefineMethod
("Skriv",MethodAttributes.Public,null,null);

				//OBS - en ILGenerator skriver IL kode:
```

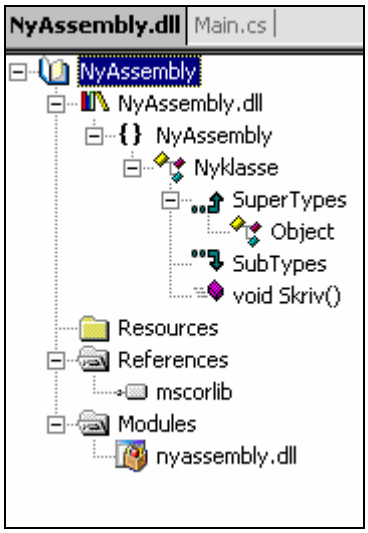
```
                    ILGenerator skriv_metode_IL=metode_builder.GetILGenerator();
                    skriv_metode_IL.EmitWriteLine ("Dette er en linje fra en dynamisk oprettet
assembly!");
                    //Metoden Emit() skriver IL op-koder fx 'Ret' = return (fra metode):
                    skriv_metode_IL.Emit(OpCodes.Ret);

                    //OBS OBS: Her buildes og gemmes den dynamiske (runtime) assembly-type:

                    klasse_builder.CreateType();
                    assembly_builder.Save("NyAssembly.dll");


            }
}
```

Hele historien handler altså om at anvende en række 'buildere' i den rigtige rækkefølge:

1. AssemblyBuilder
2. ModuleBuilder
3. TypeBuilder
4. MethodBuilder

og derefter bruge en ILGenerator til at 'emitte' dvs skrive IL kode!

Resultatet af at køre programmet er at der oprettes en fil **NyAssembly**.dll i den aktuelle mappe (her vist i UML notation):



Vores ny klasse ser helt 'normal' ud. Den har ikke noget 'Entry Point' dvs ingen Main(), derfor skal den gemmes som DLL. Klassen arver fra System.Object – det har vi godt nok ikke selv skrevet! Klassen har en metode Skriv(). Den gemmes i modulet 'nyassembly.dll' (Assembly'en kunne have været fordelt på mange moduler!).

Vi kan nu skrive et lille applikations program som bruger klassen:

```
//Brug vores egen NyAssembly.dll
//fil: ny_app.cs:

using NyAssembly;

class app{

                        public static void Main(){
                            Nyklasse ny=new Nyklasse();
                            ny.Skriv();

                            System.Console.Read();
                        }
}
```

Dette skal så **kompileres** sådan:

csc /r:nyassembly.dll ny_app.cs

Der skal være en explicit **reference** til nyassembly.dll - ellers kan programmet ikke kompileres. **Hvis** vi ikke havde brugt en **using** skulle vi i stedet have skrevet:

NyAssembly.Nyklasse ny=new NyAssembly.Nyklasse();

Hvis vi **ikke** havde gemt vores assembly i en fysisk fil (dvs hvis den **kun** havde ligget i RAM hukommelsen) kunne vi have brugt den med følgende kode:

```
Assembly en_assembly=Assembly.Load("NyAssembly");
Type type=en_assembly.GetType("NyAssembly.Nyklasse");
MethodInfo en_metode=type.GetMethod("Skriv");
object obj=Activator.CreateInstance(type);
en_metode.Invoke(obj,null);
```

Det er denne metode som **professionelle** builder-programmer anvender. Vi kan altså godt **loade** en assembly, **selv om** den **ikke** er gemt i en fil!! Vi kan instantiere en klasse, som **ikke** fysisk er gemt nogen steder, og vi kan få dens metoder til at køre!!

Opgaver:

1. Skriv flere metoder til Nyklasse efter det givne mønster!
2. Lad være med at gemme assembly'en i en fil og prøv at kalde metoderne fra RAM hukommelsen!
3. Skriv en 'klasse builder', hvor brugeren **indtaster** nogle ønsker og hvor programmet så opretter en klasse og en assembly efter disse ønsker!

At skrive IL (Intermediate Language) kode i hånden:

Det er sjældent nødvendigt at skrive IL kode i hånden. Desuden er det besværligt, langsomt og det er let at lave fejl i koden!

Det følgende er derfor mest ment som en illustration af hvad IL egentligt er for noget.

IL eller Microsoft Intermediate Language er et 'lav niveau' programmerings sprog (et assembler sprog) der ligger neden under sprog som C# eller Visual Basic eller C++. Det vil sige at den kode som skrives i IL er nøjagtigt den samme uanset om man 'starter' med C# eller et andet sprog som er kompatibelt med .NET.

IL kode består af helt almindelige tekstfiler fx skrevet i Notesblok. Som et enkelt eksempel er dette en IL fil som opretter 'det mest nødvendige':

```
.class public auto ansi import DanskParcelHus extends [mscorlib] System.Object {}
.class public auto ansi import DanskLilleParcelHus extends [mscorlib] System.Object {}
.class public auto ansi import DanskStortParcelHus extends [mscorlib] System.Object {}
.class public auto ansi import JyskParcelHus extends [mscorlib] System.Object {}

.assembly parcelhuse{
.ver 1:0:0:0
}
.assembly extern mscorlib{
.ver 1:0:2411:0
}
.module parcelhuse.dll
```

Filen gemmes som parcelhuse.il – idet filtypen 'il' betegner IL filer.

Filen opretter 4 klasser, som er public og som arver fra System.Object. mscorlib betegner den DLL fil hvor System.Object er defineret. De 4 klaser er 'tomme' – har ingen metoder (ud over dem de arver) og ingen constructor eller andre medlemmer.

Filen opretter også en **assembly** ved navn 'parcelhuse' med version 1:0:0:0 som har et eneste **modul** (som også hedder parcelhuse). En assembly kunne evt bestå af mange moduler.

Det som det nu drejer sig om er at få produceret en **binær kompileret** fil ved navn **parcelhuse**.**dll**. Det gøres med den IL **kompiler** som følger med .NET nemlig **ilasm**.exe sådan:

ilasm /DLL parcelhuse.il

OBS ilasm har en række 'flag' som kan ses hvis der tastes:

ilasm –help

Som **standard** kompilerer ilasm til en EXE fil – derfor er /DLL nødvendigt. **ilasm** er en **kompiler** lige som csc (for C#) eller vbc (for Visual Basic) – men den kompilerer blot IL filer i stedet for.

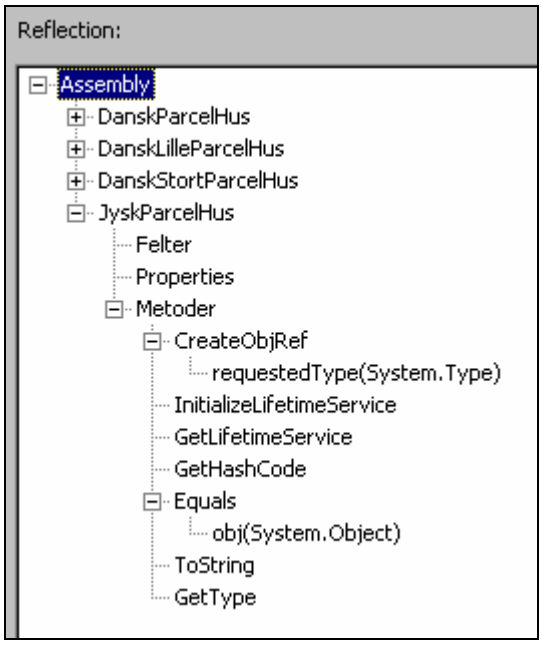
Resultatet af at kompilere en CS fil med csc og at kompilere en tilsvarende IL fil med ilasm er imidlertid **fuldstændigt** det **samme**!! Der kommer **præcist** den samme EXE eller DLL ud af det!!

Den afgørende forskel er – som sagt – at ilasm er fuldstændig ligeglad med høj niveau programmeringen!

```
C:\csharp\exe>ilasm /DLL parcelhuse.il

Microsoft (R) .NET Framework IL Assembler.  Version 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
Assembling 'parcelhuse.il' , no listing file, to DLL --> 'parcelhuse.
Source file is ANSI

Creating PE file

Emitting members:
Global
Class 1
Class 2
Class 3
Class 4
Writing PE file
Operation completed successfully
```

Som det ses 'emitter' (opretter .NET koden til) kompileren 4 klasser nemlig de 4 parcel hus klasser vi har defineret.

Den oprettede DLL fil kan nu ses ved hjælp af C# reflection (her vist i det Reflection program som ligger på http://csharpkursus.subnet.dk):



De 4 mere eller mindre velbegavede husklasser ses her. Klasserne har kun de metoder de arver fra System.Object. De kan altså ikke - som de er nu - bruges til noget fornuftigt! De er kun ment som en **illustration** af hvad der egentligt foregår i .NET!

For alligevel at bevise at klasserne faktisk ER oprettet i systemet, kan skrives dette lille program i C#:

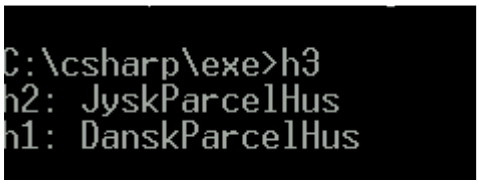
```
using System;

class HUS{
                                public static void Main(){
                                      DanskParcelHus h1;
                                      JyskParcelHus h2;
                                      Console.WriteLine("h2: {0}",typeof(JyskParcelHus));
                                      Console.WriteLine("h1: {0}",typeof(DanskParcelHus));
                                }
}
```

Denne fil kan kompileres og de 2 objekter (dvs pointere egentligt) kan altså oprettes.
Filen skal kompileres med csc /r:parcelhuse.dll ...
Kompileren giver dog en advarsel:

```
Copyright (C) Microsoft Corporation 2001. All rights reserved.

h3.cs(5,18): warning CS0168: The variable 'h1' is declared but never us
h3.cs(6,17): warning CS0168: The variable 'h2' is declared but never us
```

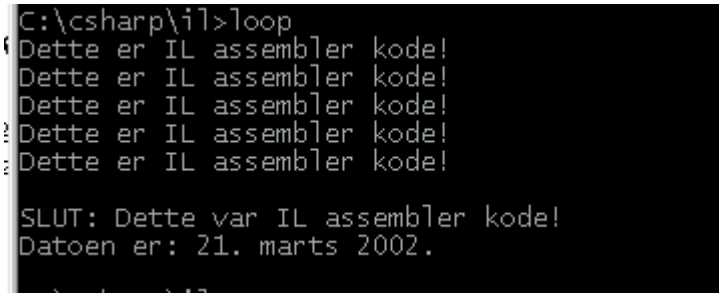
De 2 objekter bliver faktisk aldrig brugt til noget – og det giver en advarsel – det må man ikke! Men kompileren kan altså godt **finde** de 2 klasser!!

NB Når en klasse **IKKE** har en **constructor** (og det har vores 4 husklasser **IKKE** – **heller** ikke en standard eller default constructor!) kan de **ikke** oprettes med **new**. Man kan altså ikke instantiere disse klasser - det eneste man kan - stort set - er at undersøge deres **type** – som vist her.



Et konkret eksempel på IL:

Uden at gå i for mange detaljer skal her vises et eksempel på et færdigt, 'levedygtigt' IL program som gennemfører en løkke. Programmet udskriver:



Den følgende kode viser hvordan en sådan loop kan skrives i IL – koden er forklaret løbende:

```
//IL assembler program
//assembles med ilasm <programnavn>
//evt ilasm /DLL <programnavn>

//svarer til brug af using i C#:
.assembly extern mscorlib{}

//en assembly skal have et navn
.assembly Loop{

//her kunne tilføjes mange andre egenskaber:
.ver 1:0:1:0
}

//en assembly indeholder mindst eet modul:
.module loop.exe

//Der oprettes en klasse uden om metoden:

.class public auto ansi ProgramKlasse extends [mscorlib]System.Object{

.method public static void Main() cil managed{

//aldrig mere end 2 variable i stacken ad gangen!
//dette direktiv kan overspringes!
```

```
//.maxstack 2

//alle exe SKAL have en entrypoint:
.entrypoint

//opret 3 anonyme lokale variable a la int x=0:

//init betyder at int saettes til 0 og en streng til null:
.locals init(int32,int32,string)

//var 0 = 0

//push:
ldc.i4 0

//pop:
stloc.0

//var 1 = 4

//push
ldc.i4 4

//pop:
stloc.1

//var 2 = en streng/tekst:

//push streng til stack
ldstr "Datoen er: 21. marts 2002."

//pop strengen fra stacken:
stloc.2

//loop:
Igen:

//ldloc.0 betyder push local variabel til stacken:
ldloc.0
ldloc.1

//Er var 0 stoerre end var 1? bgt = branch if greater than:
bgt Slut

//dvs var 0 er stadig mindre end var 1 - inkrementer med 1:
ldloc.0
ldc.i4 1
add
stloc.0
ldstr "Dette er IL assembler kode!"
call void [mscorlib] System.Console::WriteLine(string)

//br betyder branch altid/spring altid tilbage til Igen:
br Igen

Slut:
ldstr "\nSLUT: Dette var IL assembler kode!"
call void [mscorlib] System.Console::WriteLine(string)
```

```
//dvs push strengen i den lokale variabel paa stacken:
ldloc.2
call void [mscorlib] System.Console::WriteLine(string)

ret
}
}
```

At programmere i IL betyder at man hele tiden lægger værdier i og henter værdier fra metodens stack! Et eksempel er slutningen:

```
ldstr "\nSLUT: Dette var IL assembler kode!"
call void [mscorlib] System.Console::WriteLine(string)

//dvs push strengen i den lokale variabel paa stacken:
ldloc.2
call void [mscorlib] System.Console::WriteLine(string)
```

ldstr lægger strengen oven i stacken. Når metoden WriteLine() kaldes tager den simpelthen den værdi der ligger øverst i stacken og udskriver den!

Ved at studere IL kode får man altså et mere præcist billede af, hvad der rent faktisk foregår i et C# program. Vi kan også se, at løkker (som i C# hedder for, while, do osv) alle sammen i IL implementeres som branch som fx:

```
br Igen
```

der betyder: Hop tilbage til den label der hedder 'Igen:'! Dette svarer til goto sætninger i C#!

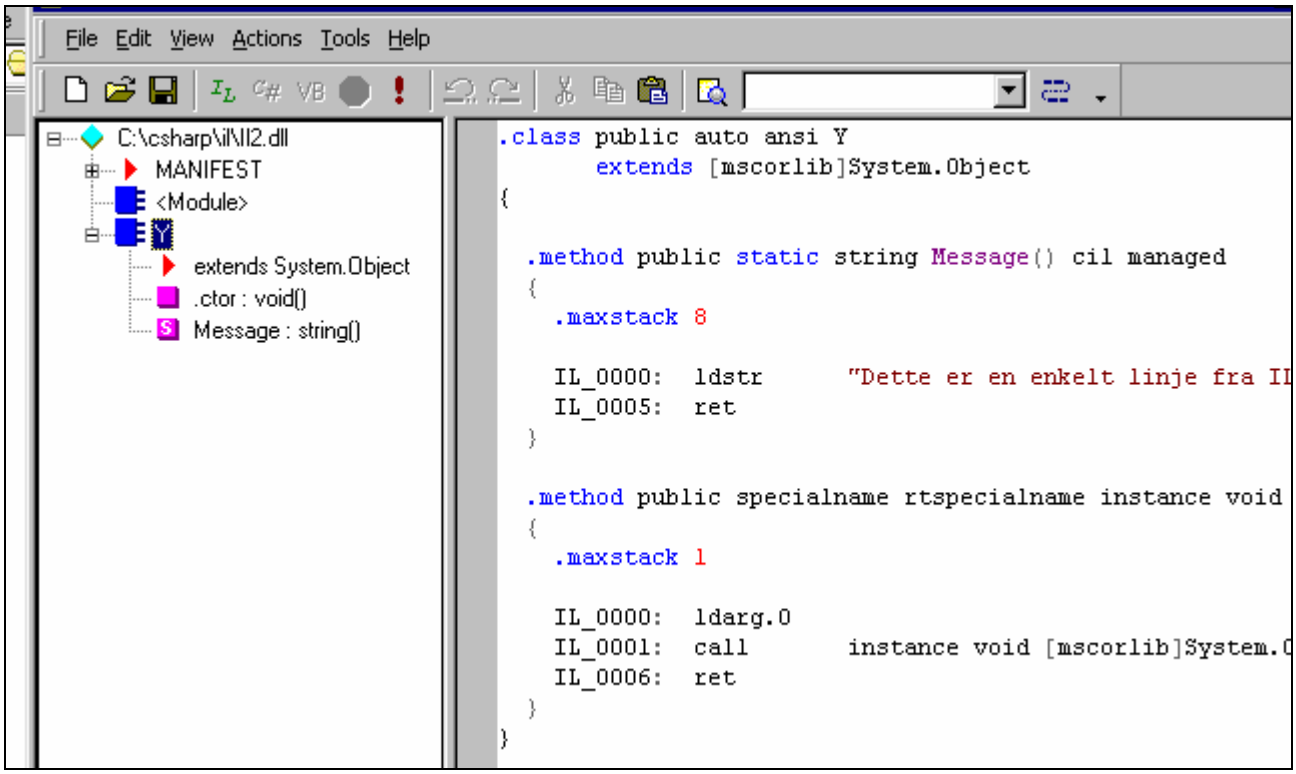
Variable i en metode i IL er typisk uden navn og identificeres kun ved deres nummer i en række! Fx opererer metoden ovenfor med 2 int variable og en streng variabel.

Programmering i IL ligner meget anden assembler programmering – som stort set altid foregår efter samme model.

Dis-assembling, utilities:

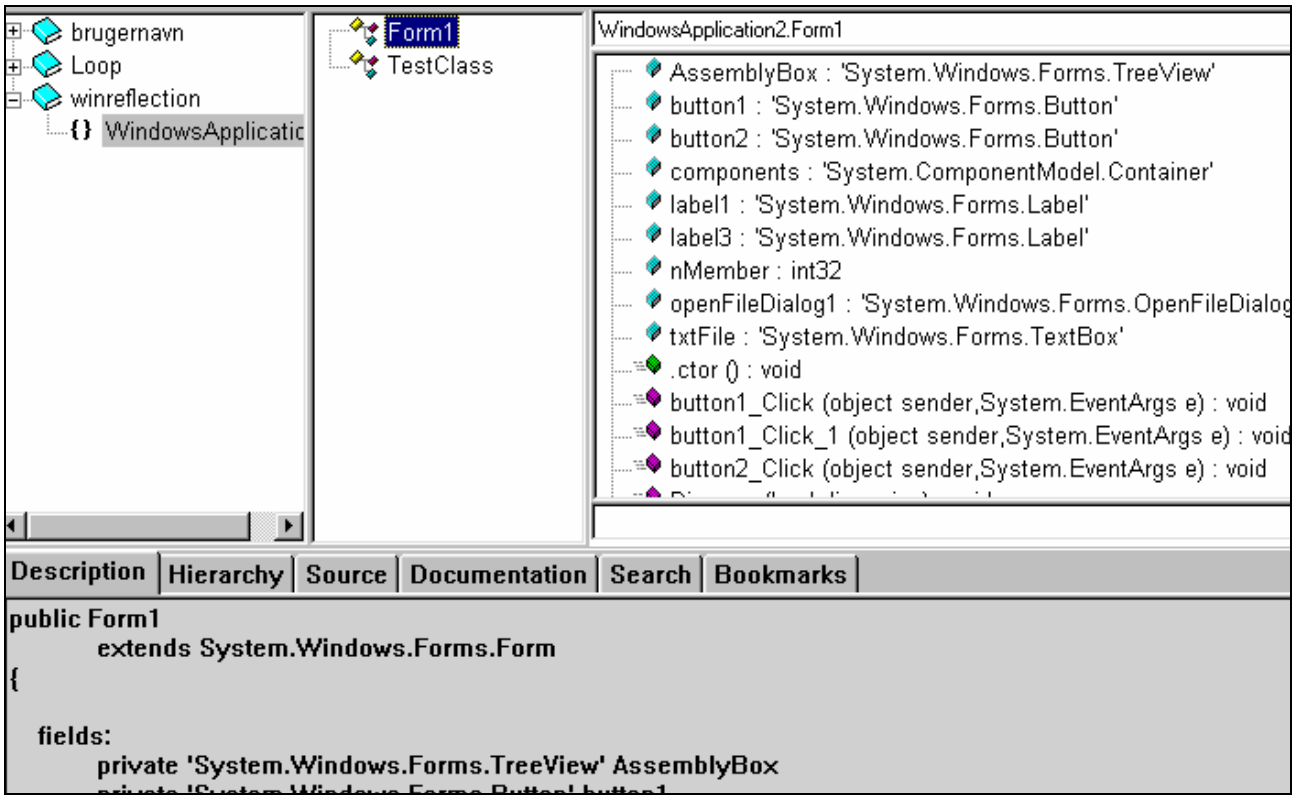
Der kan downloades shareware programmer som kan arbejde med IL kode, producere IL kode fra EXE filer (**disassemble**) osv. Ved at bruge sådanne programmer kan man – i et vist omfang – lave 'reverse engineering' (konstruere kode fra binære filer) og i al fald få en bedre forståelse for mekanismer i C# og .NET!

Her er et eksempel, hvor en DLL fil vises i programmet .NET Explorer fra http://www.remotesoft.com:



NB: Denne slags programmer viser ofte IL kode med en **label** foran hver linje – her fx 'IL_0001:' osv. Disse labels er egentlig slet ikke en del af selve IL koden! De viser hvor mange bits og bytes hver linje fylder. Som det ses er det almindeligt at en IL instruktion i alt fylder 5 bytes – nemlig 1 byte til selve instruktionen (fx ldstr) og 4 bytes til argumentet (fx "Dette er ...").

Et andet eksempel er **Lesser Software** .NET Reflection Browser - her følger et eksempel på hvad dette program kan producere (programmet kan hentes som en evaluation). I dette eksempel er også loadet en binær exe fil winreflection.exe:


brugernavn
Loop
winreflection
{} WindowsApplicatic
Form1
TestClass
WindowsApplication2.Form1
AssemblyBox : 'System.Windows.Forms.TreeView'
button1 : 'System.Windows.Forms.Button'
button2 : 'System.Windows.Forms.Button'
components : 'System.ComponentModel.Container'
label1 : 'System.Windows.Forms.Label'
label3 : 'System.Windows.Forms.Label'
nMember : int32
openFileDialog1 : 'System.Windows.Forms.OpenFileDialog
txtFile : 'System.Windows.Forms.TextBox'
.ctor () : void
button1_Click (object sender,System.EventArgs e) : void
button1_Click_1 (object sender,System.EventArgs e) : void
button2_Click (object sender,System.EventArgs e) : void
Description
Hierarchy
Source
Documentation
Search
Bookmarks
public Form1
extends System.Windows.Forms.Form
{
fields:
private 'System.Windows.Forms.TreeView' AssemblyBox