

Gert Hansen / Hadsten højskole

Mange tilhørende videoer på
www.bogenomcsharp.dk

BOGEN OM

C#

2021-UDGAVE (C# 9.0)

MICHELL CRONBERG

Gert Hansen / Hadsten højskole

Bogen om C# 9.0

Michell Cronberg

Gert Hansen / Hadsten højskole

Copyright © 2021 Konsulentfirmaet M. Cronberg Aps

Michell Cronberg

Bogen om C# 9.0

1. udgave - februar 2021

Forlag:

Konsulentfirmaet M. Cronberg Aps

Allingtonvænget 32

5270 Odense N

Mail: michell@cronberg.dk

Forfatter: Michell Cronberg

Redaktør og korrektur: Henrik Trojaborg

Omslag: Claus Dalgaard

Tryk: Scandinavian Book A/S

ISBN: 978-87-993382-3-8

Bogen må ikke kopieres eller efterlignes helt eller delvist uden skriftlig tilladelse fra forlaget. Hverken forlag, redaktør eller forfatter kan holdes økonomisk ansvarlig for eventuelle fejl og mangler i bogen.

Indholdsfortegnelse

Indledning	2
Introduktion til .NET.....	3
Udviklingsmiljøer	14
Introduktion til C#.....	19
En konsol-applikation i C#.....	35
Praktisk brug af VS og VSC	47
Simple variabler	60
Tekster	87
Konstanter	99
Programflow	105
Metoder	119
Fejlhåndtering	138
Arrays.....	153
Samlinger	162
Dine egne typer.....	170
Klasser	183
Grundlæggende hukommelsesteori	201
Indkapsling	214
Arv.....	228
Polymorfi.....	241
Interface.....	257
Delegates	266
Hændelser.....	286
Avancerede typer.....	295
LINQ	320
Asynkron programmering.....	338
Efterskrift	354
Figurer	355
Tabeller	357
Specificeret indholdsfortegnelse	358
Indeks.....	364

Indledning

Så er jeg i gang med endnu en indledning til endnu en bog om C#.

Jeg har tidligere udgivet en bog om C# 2.0 med min gamle ven Niels Hilmar Madsen i 2003, en bog om C# 5.0 i 2012, en bog om C# 8.0 i 2020 – og nu en opdateret version af bogen fra 2020 med nyt indhold relateret til C# 9.0.

Jeg har skrevet bogen med det formål, at den kan benyttes på uddannelsesinstitutionerne som en dansk grundbog i programmering med C#. Den indeholder alle de emner, som du bør kende til for at kunne benytte C# til udvikling af mange forskellige typer af applikationer, men jeg har dog begrænset og skåret en del i emnerne. Det skyldes dels, at jeg gerne vil ramme pensum så godt som muligt, dels at jeg ikke vil gøre bogen alt for tung. Den er havnet på lige under 400 sider, hvilket jeg synes passer meget godt. I slutningen af de fleste kapitler har jeg et, *Hvis du vil vide mere*-afsnit. Her kan du læse mine forslag til eventuel yderligere teori, du kan kaste dig over.

På <https://www.bogenomcsharp.dk> kan du finde yderligere materiale relateret til bogen. Du kan blandt andet finde en blog med nyheder relateret til bogen og en del videoer, som er optaget for at underbygge emner, som kan være svære at skrive og læse om. Nogle emner beskrives bedre med video end med ord.

God fornøjelse med bogen!

Odense, februar 2021

Michell Cronberg

Introduktion til .NET

I midten og slutningen af 1990'erne var der flere dominerende programmeringssprog herunder JavaScript, Python, Delphi, C++ og Java. Microsoft gjorde på det tidspunkt mest i C, C++, Visual Basic (VB) og komponenter relateret til Java og JavaScript.

I 1996 blev danske Anders Hejlsberg (født 1960) ansat hos Microsoft for i første omgang at hjælpe med udviklingen af Microsofts Java-komponent kaldet J++, men han blev hurtigt den drivende kraft bag et nyt programmeringssprog kaldet C# og en tilhørende runtime.

Se eventuelt et interview med Anders Hejlsberg, hvor han fortæller om de mange projekter, han har arbejdet på. Jeg har skrevet et blogindlæg om det i oktober 2020 (se <https://blog.cronberg.dk/>).

Anders Hejlsberg havde erfaringer med programmeringssprog fra sin uddannelse og sin tidligere ansættelse hos Borland, hvor han arbejdede med sprog som Turbo Pascal og Delphi. Han ville udvikle C# som et sprog, der lå mellem det meget produktive og simple Visual Basic, og det komplekse, men hurtige og effektive C++.

År	Version
2002	1
2005	2
2007	3
2010	4
2013	5
2015	6
2017	7
2019	8
2020	9

Tabel 1 C# versioner

Omkring 2002 blev den første version af C# og tilhørende runtime frigivet, og i skrivende stund (2021) er version 9 frigivet.

Microsoft kaldte sproget for C# for at skabe en reference til C/C++ og for at indikere, at C# er skridtet efter C++ (inden for musikken er # symbolet for at hæve en tone med en halvtone).

.NET

C# (og andre sprog som VB.NET og F#) er en integreret del af det såkaldte .NET økosystem. Det er en samling af komponenter, som bruges til både at afvikle og udvikle applikationer. Disse komponenter kaldes tilsammen for en runtime.

I november 2020 blev .NET 5 frigivet og sammen med den også C# version 9. Der eksisterer en del ældre .NET platforme og standarder herunder:

- .NET Framework
- .NET Core
- Xamarin / Mono
- .NET Standard (fælles standard for ældre runtimes),

men .NET 5 er første spadestik til sammenlægning af de forskellige runtimes til en fælles runtime. I næste version (.NET 6 – ultimo 2021) forventes det, at sammenlægningen er komplet.

Fra ultimo 2020 skal du fokusere på .NET 5 men blot være opmærksom på, at .NET 5 i virkeligheden er næste version af den runtime, som hedder .NET Core. Derfor vil nogle projekter i Visual Studio og andre steder måske stadig have rester af navngivningen fra .NET Core.

Du behøver ikke fokusere på de andre ældre runtimes i standard C# udvikling, men på at skabe .NET 5 applikationer. Det vil dels sikre at de

applikationer, du skaber, er optimeret mest muligt, samt at du benytter den seneste version af C# (version 9).

Generelt typebibliotek

I .NET 5 vil du kunne finde en masse forskellige komponenter, herunder komponenter til kompilering, afvikling, fejlfinding og oprydning, men den mest synlige komponent er en meget stor samling af typer med tilhørende generel funktionalitet.

Således vil det eksempelvis være nemt at skrive kode til at arbejde med simple og komplekse variabler, filer, databaser, internettet, xml- eller json-filer, tilfældige tal, kryptering og meget andet. Den kode har Microsoft allerede skrevet og stiller til rådighed enten som en del af runtime eller som en pakke (i .NET hedder sådan en pakke en NuGet-pakke), du frit kan hente og benytte. De fleste typer er organiseret i en logisk struktur, og det er relativt simpelt at finde den funktionalitet, man ønsker.

Faktisk er det, at lære disse medfølgende typer at kende, en stor del af at blive en dygtig C#-udvikler. Dels skal man lære, hvilke typer der er tilgængelige, dels hvordan man benytter dem bedst. Der er ingen grund til, at du opfinder den dybe tallerken igen, når nu Microsoft har skrevet og testet en masse kode for dig.

Som du senere vil lære, så er dette generelle bibliotek struktureret i et stort namespace (samling af typer og andre namespaces) kaldet *System*. Dette bibliotek består af tusindvis af typer, du kan benytte frit i din kode.

Kompilering af kode i .NET

Traditionelle sprog som eksempelvis C++ oversætter typisk kildekode (source code) til assembler eller direkte til binære instruktioner:



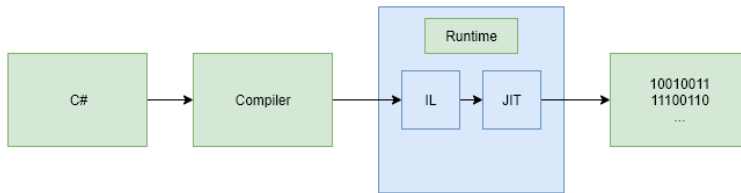
Figur 1 Fra C++ til binær eksekverbar kode

Det kræver typisk en kompiler til hver platform, applikationen skal afvikles på, men til gengæld kan applikationen afvikles uden yderligere installation fra brugerens side.

En kompiler (compiler) og en transpiler er begge programmer, som har til formål at oversætte en given syntaks til kode. Forskellen på en kompiler og en transpiler er typisk, at en kompiler producerer eksekverbar kode, medens en transpiler producerer kode i en anden syntaks. Helt overordnet kan du blot se begge som et program som oversætter syntaks til noget andet, og så ikke gå så meget op i om det kaldes en kompiler eller transpiler.

I C# oversættes kildekode derimod til en såkaldt mellemkode kaldet IL kode (Intermediate Language), som så ved afvikling oversættes til binær eksekverbar kode af den på brugerens maskine installerede runtime.

Denne oversættelse kaldes blandt andet for JIT (just in time) transpilering:



Figur 2 Fra C# til mellemkode til binær eksekverbar kode

Således er en runtime nødvendig for at afvikle en .NET applikation, og den skal enten installeres eller være en del af den eksekverbare fil, inden applikationer kan afvikles. Fordelen ved det er, at C# transpileren oversætter til en fælles sprogstandard, og at den efterfølgende oversættelse fra selve runtime kan optimere oversættelsen til at tage hensyn til forskellige platforme og ressourcer.

```
.maxstack 2
.locals init (
    [0] int32,
    [1] bool
)

IL_0000: nop
IL_0001: ldstr "Program start"
IL_0006: call void [mscorlib]System.Console::WriteLine(string)
IL_000b: nop
IL_000c: ldc.i4.1
IL_000d: stloc.0
// sequence point: hidden
IL_000e: br.s IL_001d
// loop start (head: IL_001d)
IL_0010: nop
IL_0011: ldloc.0
```

Figur 3 Eksempel på IL kode som er langt mere kompleks end C#

Som C# udvikler behøver du ikke forstå IL kode.

Når du læser om .NET vil du måske falde over begrebet "virtuel maskine". Det skal du forstå således, at Microsoft har skabt en specifikation til, hvordan en generel computer og tilhørende operativsystem fungerer, og IL-koden er skrevet til denne virtuelle

enhed. Opgaven for runtime er så at oversætte IL-koden til kode, der kan afvikles på den pågældende platform.

Som begynder i C# programmering er det ikke så vigtigt at du kender så meget til oversættelse af kode eller den underliggende platform og runtime. Men du bør kende begreberne .NET, runtime og mellemkode (IL-kode). Når du er blevet mere erfaren, eller skal distribuere applikationer professionelt, bør du læse mere om dette.

Afvikling af .NET applikationer

For at kunne afvikle .NET applikationer skal der altså enten installeres en runtime, eller også skal den kompilerede eksekverbare fil indeholde en runtime. Fordelen ved en at skabe en fil som indeholder alt det som skal bruges til afvikling (kaldes på engelsk en "self contained application") er, at brugerne ikke behøver have noget installeret. Ulempen er at den eksekverbare fil hurtigt kan fylde over 10-20 Mb.

Den sidste nye version af .NET runtime kan altid hentes fra

<https://dotnet.microsoft.com/download>

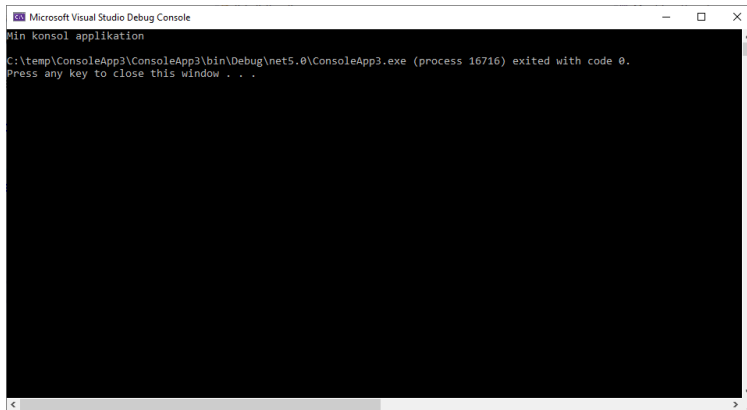
På siden skal du tage stilling til, om du ønsker at hente en ren runtime (til afvikling) eller et SDK (Software Development Kit) til afvikling og udvikling.

Hvis du udvikler med Visual Studio (som er fuldt opdateret), er runtime allerede installeret på din maskine. Hvis du udvikler med Visual Studio Code, skal du selv sørge for at installere SDK.

Type af applikationer

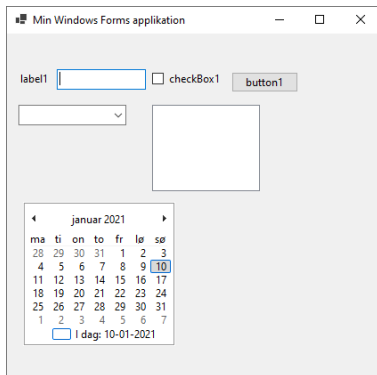
Hvis først du kender til et .NET sprog (C#, F#, VB.NET), har du mulighed for at skabe mange forskellige typer af applikationer.

Denne bog tager udgangspunkt i en konsolapplikation for at kunne holde fokus på teorien og syntaksen. Det er den mest simple type af applikation, men bliver faktisk benyttet en del til værktøjer og administrative applikationer. Og som en af de få applikationstyper (bortset fra webapplikationer) kan den afvikles på alle supportede platforme.



Figur 4 Konsol applikation

Hvis du ønsker at skabe en desktop applikation har du forskellige muligheder. Du kan vælge at udvikle en traditionel Windows Forms-applikation, som er bundet til interne komponenter i Windows. Denne type applikation har levet i mange år, og er opgraderet til forskellige runtimes. I skrivende stund kan den findes på listen over mulige projekter i Visual Studio som "Windows Forms (.NET)".

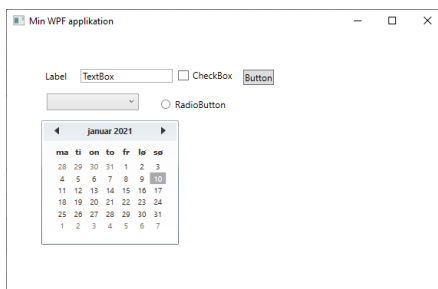


Figur 5 Windows Forms applikation

Med Windows Forms kan du meget hurtigt skabe en brugerflade med alle mulige former for kontroller, og så binde det hele sammen med C# kode.

Du kan også vælge en WPF (Windows Presentation Foundation) applikation som er noget nyere end Windows Forms, og ikke helt så bundet til interne komponenter i Windows. Der er nogen flere muligheder i en WPF applikation, men den er fortsat bundet til Windows-plattformen.

I skrivende stund kan den findes på listen over mulige projekter i Visual Studio som "WPF App (.NET)".



Figur 6 WPF applikation

I skrivende stund er Microsoft ved at færdiggøre et projekt kaldet WinUI¹, og det vil sikkert blive den anbefalede måde (måske fra .NET 6) at skabe desktop-applikationer på.

Kig på WinUI hvis du skal skabe nye desktop-applikationer.
Det burde være klar i løbet af 2021.

Du kan også skabe forskellige typer af mobile applikationer til Android og iOS. Microsoft købte for nogle år siden et firma, der hed Xamarin, som havde skabt et framework til at udvikle mobile applikationer. Navnet Xamarin fremgår stadig af projektyperne i Visual Studio, og det er i skrivende stund måden at skabe mobile applikationer på med C#.

Sammen med .NET 6 (slutningen af 2021) forventes det, at Microsoft frigiver et projekt kaldet MAUI² (Multi-platform App UI). Det er en videreudvikling af Xamarin Forms, og nye projekter bør måske kigge på MAUI fremfor Xamarin.

Kig på MAUI hvis du skal skabe en ny mobil applikation.
Det burde være klar i løbet af 2021.

Hvis du ønsker at skabe web-applikationer har Microsoft tre forskellige projektyper, du kan vælge imellem.

Det mest solide og komplette framework hedder ASP.NET MVC (står for Model View Controller), og det benyttes til de traditionelle serverbaserede web-applikationer. Alternativet er det mindre, men også meget nemmere, framework kaldet ASP.NET Web Pages. Hvis du blot skal skabe et par websider, er det uden tvivl nemmere at komme i gang med.

¹ <https://microsoft.github.io/microsoft-ui-xaml/>

² <https://github.com/dotnet/maui>

Som noget nyt kan du også vælge at skabe SPA-applikationer (Single Page Applications) med et nyt framework kaldet Blazor³. Det benytter helt nye standardiserede måder at afvikle IL kode (og dermed C#) direkte i browseren. I skrivende stund er dette framework meget nyt, men den teknologi, det er bygget på (WebAssembly forkortet WASM), er på vej frem i webudvikling, og det bliver spændende at se, om Blazor bider sig fast.

Kig på Blazor hvis du skal skabe en ny webapplikation.

Hvis du ønsker at skabe en servicebaseret applikation, som kan være kernen i samling af forskellige applikationer, bør du se nærmere på gRPC⁴. Det er oprindeligt Google's framework til RPC (Remote Procedure Call), og er nu tæt på industristandard for den type af applikationer på mange platforme og i mange programmeringssprog. Projekttyper relateret til gRPC er i skrivende stund netop tilføjet .NET (og Visual Studio).

Sluttelig kan du næsten ikke skabe en applikation uden at involvere en database. Der findes masser af muligheder for at tale med alle mulige former for databaser i .NET, men de fleste vælger at benytte et ORM-produkt (Object–Relational Mapping). Det giver mulighed for at abstrahere meget af databasekoden væk, og i stedet fokusere på forretningslogik.

Microsofts eget ORM-produkt hedder Entity Framework⁵ (EF), og kan spare en frygtelig masse tid ved udvikling af en applikation.

³ <https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor>

⁴ <https://docs.microsoft.com/en-us/aspnet/core/grpc/>

⁵ <https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/overview>

Kig på Entity Framework (EF) hvis du skal skabe en applikation, som kommunikerer med en database.

EF er en integreret del .NET og udviklingsmiljøerne.

NuGet

Selv om det generelle typebibliotek i .NET er kæmpestort og består af tusinder af forskellige typer, som kan hjælpe dig med at udvikle applikationer, findes der yderligere tusinder af generiske komponenter som du (typisk gratis) kan benytte dig af. De kan findes i en service kaldet NuGet⁶ drevet af Microsoft, og i skrivende stund er der over 230.000 forskellige pakker med kode relateret til næsten alle former for problemstillinger.

NuGet indeholder tusinder af pakker med kode, du kan benytte i din applikation.

Hvis du udvikler en applikation, og lige står og mangler en metode der kan hjælpe dig med et eller andet konkret, bør du lige kigge i NuGet og se, om du ikke kan finde det, du mangler.

Brug af pakker hentet fra NuGet er en integreret del af alle store C# applikationer, og de fleste pakker er i øvrigt typisk et open source-projekt fra GitHub.

⁶ <https://www.nuget.org/>

Udviklingsmiljøer

Alle programmeringssprog opbevarer kildekode i almindelige tekstfiler, og du kan derfor i princippet benytte en ganske almindelig teksteditor som Notepad til at redigere kildefiler med kode og så compilere på kommandoprompt. I den virkelige verden vil det tage alt for lang tid og blive alt for tungt, så alle benytter et program (udviklingsmiljø) skabt til at skrive kode.

Når du skal udvikle i C#, har du flere miljøer at vælge imellem. Visual Studio (forkortet VS) er Microsofts store udviklingsmiljø og benyttes af de fleste C# udviklere. VS kan installeres på Windows og på Mac.

Inden for de seneste par år er Visual Studio Code (forkortet VSC) dukket op som et alternativ. Det fylder ikke så meget som VS, men giver til gengæld heller ikke helt de samme muligheder. Flere og flere benytter dog efterhånden VSC, fordi det er hurtigt, effektivt og kan installeres på alle platforme.

Valg af udviklingsmiljø

Du kan frit vælge, om du ønsker at benytte Visual Studio eller Visual Studio Code, men skal jeg anbefale dig noget, må det være Visual Studio. Den fylder en del på disken, men er bedre at arbejde med for begyndere, og der er en del flere muligheder for at udvikle forskellige typer af applikationer. Og skal du ud og arbejde som C# udvikler, vil du nok ende i Visual Studio.

Men du kan altså uden problemer benytte Visual Studio Code til C# udvikling, og samtlige eksempler i denne bog (bortset fra et enkelt som er relateret til en Windows-brugerflade) kan udvikles og afvikles med VSC. Yderligere har VSC den fordel, at den kan installeres på alle platforme – så arbejder du på en Mac eller Linux, er VSC det bedste valg. Samtidig fylder den ingenting sammenlignet med VS og kan i øvrigt benyttes til alt muligt andet end C# udvikling.

Så det er op til dig – VS eller VSC. Begge er meget effektive udviklingsmiljøer, og begge kan benyttes til at skrive kode i den applikationstype, som bogen benytter.

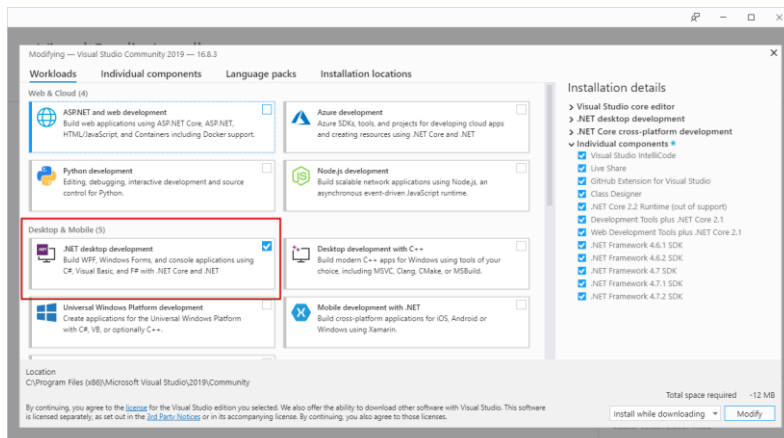
Installation af Visual Studio

Du kan finde Visual Studio på

<https://visualstudio.microsoft.com/vs>

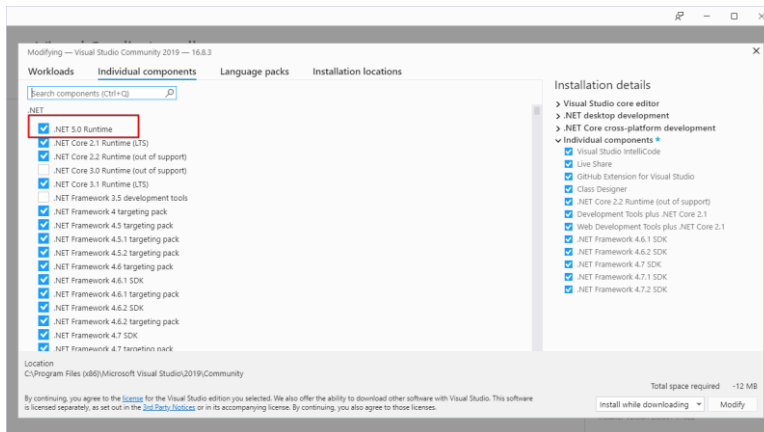
i forskellige versioner og til både Windows og til macOS. Hvis du er privat eller et mindre firma (læs licensbetingelser) er Visual Studio Community et gratis program, du blot kan hente og installere. Alternativt kan du købe både Visual Studio Professional og Visual Studio Enterprise, men ingen af disse versioner er nødvendige i forbindelse med denne bog.

Når du installerer Visual Studio skal du som minimum sætte kryds i det *workload* som hedder *.NET desktop development*:



Figur 7 Installation af Visual Studio 2019

Og du bør lige checke, at *.NET 5* er valgt under "Individual components":



Figur 8 Installation af Visual Studio 2019 (.NET 5)

Når installationen er færdig, kan du starte VS, og i skrivende stund skal du ved første start vælge, om du ønsker at oprette en profil. Det giver mulighed for, at indstillinger kan synkroniseres over flere maskiner hvis du har behov for det – ellers blot ignorer det.

Visual Studio

Welcome!

Connect to all your developer services.

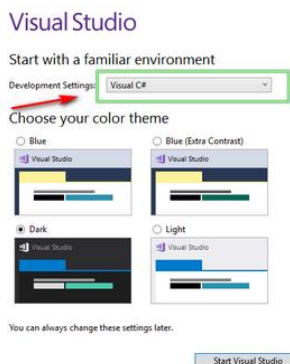
Sign in to start using your Azure credits, publish code to a private Git repository, sync your settings, and unlock the IDE.

[Learn more](#)



Figur 9 Profil ved start af Visual Studio

Herefter bør du vælge *Visual C#* som *Developer settings*, samt et farveskema (mange bruger det mørke tema – det skulle være bedst for øjnene).



Figur 10 Valg af settings og tema

Herefter skulle Visual Studio være klar til udvikling af applikationer.

Bemærk, at denne beskrivelse af installation er skrevet i starten af 2021 med udgangspunkt i Visual Studio 2019 version 16.8.3. Når du læser dette, kan der være frigivet en ny version af Visual Studio, og processen ved opstart kan være anderledes.

Installation af Visual Studio Code

Hvis du vælger at bruge Visual Studio Code, skal du først hente .NET Core SDK (Software Development Kit), som indeholder kompilator og andre værktøjer, du skal bruge til udvikling. Det kan du finde på

<https://dotnet.microsoft.com/download>

hvor du blot skal vælge den rigtige platform (Windows, Mac eller Linux). Men husk at hente SDK-filen og sørg for at vælge .NET 5.x.

Herefter kan du installere Visual Studio Code (VSC) fra

<https://code.visualstudio.com/>

til din platform.

Udover selve VSC skal du installere nogle udvidelser (*extensions*). Du finder de mulige udvidelser ved at vælge *Extensions* fra *View*-menuen. I søgefeltet skal du lede efter:

- C# (eller ms-dotnettools.csharp)
- Visual Studio IntelliCode (eller visualstudio-exptteam.vscodintellicode)

Når de er installeret, er du klar til at udvikle C# applikationer med VSC.



Du kan finde videoer på bogenomcsharp.dk, der viser, hvordan du installerer extensions i Visual Studio Code. Klik på "Video" i menuen.

Introduktion til C#

Alle traditionelle programmeringssprog gemmer kildekode (source code på engelsk) i ganske almindelige tekstfiler – typisk med et sigende filtypenavn. C# filer gemmes som .cs-filer – og i et tegnsæt, som håndterer alle typer tegn og bogstaver. Du må således gerne bruge danske bogstaver, men typisk prøver man at holde sig til engelsk i kildekode. I denne bog ”kodes” der på dansk, men det er af pædagogiske årsager.

Selve syntaksen har nogle få, men vigtige regler, som du skal kende til.

Tuborgklammer

C# er et såkaldt tuborgklammesprog – ligesom Java, C++, C og andre sprog. Hvis du har erfaring med lignende syntaks, er det nemt at komme i gang. Hvis ikke så bøvler du sikkert en del i starten med at finde tuborgklammerne på et dansk tastatur (Alt Gr + 7 og Alt Gr + 0):

```
using System;

namespace MinTest
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello world");
        }
    }
}
```

Bemærk, at tuborgklammer bruges til at markere blokke af kode. Indrykning er kompiler ligeglad med, men det bruges normalt for at gøre koden så nem at læse som muligt.

Semikolon afslutter en instruktion

Kompilatoren er også ligeglad med linjeskift, mellemrum, tabuleringer og lignende. Det, som afslutter en instruktion, er semikolon. Således er alle følgende kodelinjer lovlige:

```
int a = 10;
int b =      10;
int c

      = 10;
```

De fleste vil jo skrive koden som den første linje, men de efterfølgende er helt lovlige.

Følgende er også fuldstændig lovlig kode:

```
bool d = true;
if (d == true) { Console.WriteLine(d); }
if (d == true) {
    Console.WriteLine(d);
}
if (d == true)
{
    Console.WriteLine(d);
}
```

Bemærk de forskellige måder at sætte tuborgklammerne.

I virkeligheden koder man typisk efter en standard, så det ser pænt ud, især hvis man er flere om at skrive koden – men det er helt op til dig som udvikler. Kompilatoren er ligeglad, så længe du har styr på dine semikoloner og tuborgklammer.

Kommentarer

Kommentarer i koden kan være ret brugbart som dokumentation, og i C# kan man enten skrive en kommentar på en enkelt linje som:

```
// Dette er en kommentar
```

Eller som et afsnit:

```
/*  
    Dette er et afsnit der  
    kan bestå af flere linjer  
*/
```

Jo flere kommentarer, man skriver i koden, jo nemmere bliver den at forstå og vedligeholde på et senere tidspunkt.

Store og små bogstaver

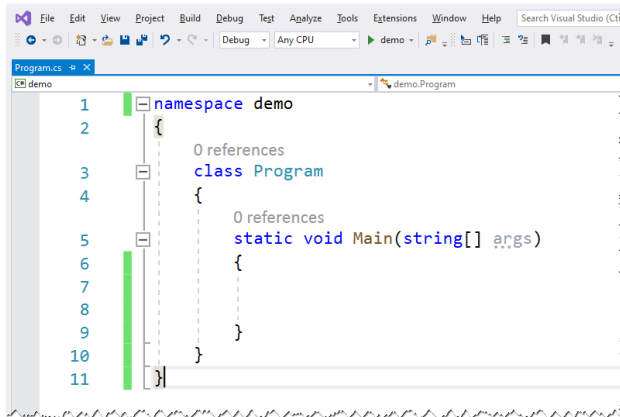
Ligesom i de fleste (men ikke alle) moderne programmeringssprog er der i C# forskel på store og små bogstaver. Det gælder både ved definering af typer samt erklæring af variabler med videre. Således er dette eksempelvis forskellige typer, variabler og metoder:

```
// der er forskel på a og A  
int a = 10;  
int A = 20;  
  
// der er forskel på metode og Metode  
void metode() { }  
void Metode() { }  
  
// Der er forskel på a og A  
class a { }  
class A { }
```

Linjenumre

Har du erfaring fra ældre programmeringssprog kender du sikkert brugen af linjenummerering som en del af syntaksen, men det bruger man ikke i moderne programmeringssprog, herunder C#, fordi man sjældent benytter jump-kommandoer (goto).

Dermed ikke sagt, at en kompiler og udviklingsmiljø ikke refererer til et linjenummer, men det er ikke noget, du skriver som en del af syntaksen.



Figur 11 Automatisk linjenummerering i Visual Studio

Programmeringsparadigmer

På de højere uddannelser inden for IT-udvikling lærer de studerende forskellige programmeringssprog – typisk 2-3 stykker. Det handler ikke om, at de skal have et dybt kendskab til, hvordan man skriver kode i konkrete sprog, men mere at de får en forståelse for, at der fundamentalt, og på et overordnet niveau, kan være forskel på, hvordan man skriver kode. Det kaldes også med et fancy ord *programmeringsparadigmer*, som vel kan oversættes til *måden at skrive kode på*.

Det *imperative* programmeringsparadigme er den gamle måde at skrive kode på – afvikling af én instruktion ad gangen og en masse hop fra den ene blok af instruktioner til den anden. Det skaber en noget uoverskuelig kode, som er svær at vedligeholde, men kæmpestore kodebaser er skabt med imperativ kode. Hele styresystemet bag Apollo 11 i 1968 er eksempelvis skrevet på den måde, og endda i Assembler, hvilket er en meget imponerende præstation. Teamet bag var ledet af Margaret Hamilton, og der findes et ikonisk billede, hvor hun står med en udskrift af kildekoden der er højere end hende selv. I oktober 2020 har jeg skrevet et indlæg om hende på min blog (<https://blog.cronberg.dk>). Her kan du også se det forrygende billede.

Procedural kode er videreudviklingen af imperativ kode, hvor blandt andet funktioner erstatter de klassiske goto-instruktioner med funktionskald.

Objektorienteret kode er en helt anden måde at skrive kode på.

I stedet for at hoppe rundt i koden kan du i den objektorienterede programmering skabe skabeloner for typer, som definerer elementer (kaldes også entiteter) i koden. Disse skabeloner kan så benyttes til at skabe objekter, der repræsenterer de konkrete elementer.

Skal du eksempelvis kode et Yatzy-spil, ville det være oplagt med typer som en terning, et terningebæger, en spilleplade, en spiller og så videre. Herefter kan du sammensætte de enkelte typer (et terningebæger består eksempelvis af terninger) og skabe objekter af disse. På den måde kan koden både skrives og læses noget nemmere end traditionel procedural kode.

I objektorienteret programmering har man ligeledes mulighed for at skabe genbrug af kode ved hjælp arv, og simplificere og optimere afvikling ved hjælp af polymorfi. Meget mere om dette senere i bogen.

Objektorienteret programmering har rod i sprog som (norske) Simula og Smalltalk, og C++ (skabt af danske Bjarne Stroustrup) er stadig højt på listen over populære programmeringssprog⁷.

Funktionsorienteret programmering tager udgangspunkt i matematiske funktioner samt en noget mere stram styring af data i hukommelsen, og er helt anderledes end både procedural og objektorienteret programmering. Det har rod i Lambdakalkyle (Lambda calculus) fra 1930'erne (opfundet af Alonzo Church, som i øvrigt var PhD supervisor for Alan Turing – begge bør du læse mere om, hvis du ikke kender dem) og er kendt fra sprog som Lisp, Scheme og F# (del af .NET).

⁷ Se StackOverflows årlige undersøgelse blandt sine brugere (<https://insights.stackoverflow.com/survey>)

C# er et objektorienteret sprog, hvor alt er baseret på typer, mens logik og flow er imperativt og proceduralt. Der er samtidigt en del features relateret til funktionsorienteret kode, og sproget favner dermed meget bredt med grene ud i alle typer af programmeringsparadigmer.

Men det er op til dig at vælge, hvordan du vil skrive kode.

Alt er typer i C#

C# er altså baseret på brugen af typer, og du vil falde over det begreb hele tiden.

Der findes seks forskellige typer:

- Klasser (class)
- Poster (records)
- Strukturer (struct)
- Relaterede konstanter (enumerations)
- Interfaces (interface)
- Referencer til metoder (delegate).

Du skal se en type som en skabelon for, hvordan instanser (kaldes også objekter) vil se ud.

Microsoft har eksempelvis defineret en masse strukturer, vi kan bruge til at skabe simple variabler. En af de meget benyttede er strukturen `System.Int32` (kaldes også en `int`), som er Microsofts bud på, hvordan et heltal skal fungere. I strukturen er der defineret, hvilke værdier instanser af et heltal består af, og hvad de fylder i hukommelsen. I `System.Int32` bliver der udelukkende opbevaret et 32-bit tal, men i andre typer kan der gemmes mange forskellige værdier. Udover værdier kan der defineres forskellige metoder, som typisk er relateret til værdien. I `System.Int32` findes eksempelvis metoden `ToString`, som kan bruges til at konvertere heltalsværdien til tekst.

Så `System.Int32`-strukturer repræsenterer altså et heltal, og hvis du ønsker at benytte et heltal i din kode, skal du blot skabe en instans:

```
// bemærk - skrives normalt på en anden og hurtigere måde  
// dette er blot et eksempel  
System.Int32 a = new System.Int32();
```

Den første linje kan du læse som:

- Skab en variabel kaldet a, som kan indeholde en instans af strukturen (typen) System.Int32
- Opret en ny instans af System.Int32 (bemærk new-kodeordet)
- Tildel den nye instans til variabelen a

Alle instanser af System.Int32 er fuldstændig ens bortset fra værdien (i dette tilfælde heltalsværdien). De har samme metoder, fungerer præcis på samme måde, og fylder det samme i hukommelsen.

Her oprettes to heltal:

```
System.Int32 a = new System.Int32();  
a = 10;  
System.Int32 b = new System.Int32();  
b = 20;
```

De er begge instanser af System.Int32-typen, men har hver sin værdi.

Et andet eksempel på en type er en klasse, du selv skaber til at repræsentere en terning i et Yatzy-spil. I typedefinitionen fortæller du kompileren, hvilke værdier en instans vil have (måske blot en værdi af terningens antal øjne), og hvilke metoder, som arbejder på instansens værdier. Måske kunne det være smart med en metode, der udskriver værdien eller en metode, der ryster terningen ved at tildele en tilfældig værdi.

Når du har defineret typen, kan du selv, eller andre der må benytte din klasse, skabe instanser (også kaldet objekter) af din skabelon, og benytte dem i eksempelvis et Yatzy-spil:

```
Terning t = new Terning();  
t.Værdi = 4;  
t.Print();  
t.Ryst();
```

```
t.Print();
```

Bemærk brugen af punktum-notationen, som fortæller kompileren, at man ønsker at tildele værdi til et konkret felt og afvikle en konkret metode på en konkret instans.

Der findes et hav af typer i runtime, som du kan benytte, og du vil sjældent skabe en C# applikation uden at skabe dine egne typer.

Vi kommer tilbage til strukturer, klasser og de andre typer, samt opdeling af hukommelse, men lige nu er det vigtigt, at du forstår, at alt er baseret på dine og andres typedefinitioner (skabeloner).

Hierarki af typer

Alle definitioner af typer – både dine egne og dem fra eksempelvis Microsoft selv – er placeret i et såkaldt *namespace*. Et namespace kan bestå af mange typer samt andre namespaces, og man kan derfor opbygge et hierarki af typer. Det sikrer logisk struktur og indkapsling, og samtidigt undgår man et eventuelt navnesammenfald mellem typer. Slutteligt gør det typerne nemme at finde i Visual Studio og Visual Studio Code.

Det bedste eksempel er Microsofts egne typer, som er placeret i eller under System-namespacet. Her finder man typer relateret til konsol, matematik, tilfældige tal, men også andre namespaces som opbevarer typer relateret til filer (System.IO), XML (System.Xml), databaser (System.Data), sikkerhed (System.Security) og så videre.

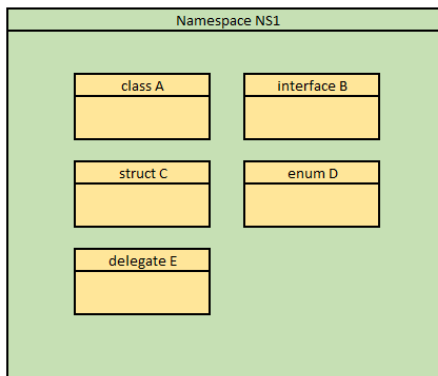
Under System-namespaces findes et kæmpe hierarki af typer, som du bruger hele tiden, og da man i C# benytter punktumnotation til at adskille namespaces og typer, er det ret nemt at finde rundt.

Eksempelvis skal du senere arbejde med samlinger, og de er placeret i et namespace et stykke nede i træet:

```
System.Collections.Generic.List<T>
```

Det kan du læse som typen `List<T>`, som ligger under `Generic-namespaces`, som ligger under `Collections-namespaces`, som ligger under `System-namespacet`.

Når du skriver kode, kan du selv opbygge en struktur og et hierarki, som passer dig, ved at benytte `namespace`-kodeordet. Du kunne eksempelvis ønske en struktur, hvor alle typer ligger under et samlet `namespace`:

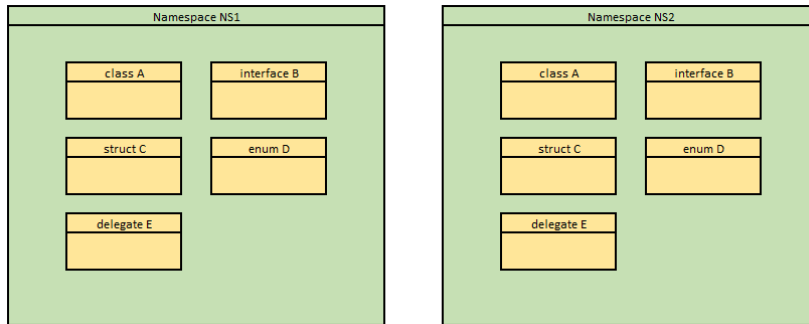


Figur 12 Typer i ét namespace

I så fald placerer du typer i samme namespace (NS1 – men det kan kaldes, hvad du har lyst til). Det vil se således ud i kode:

```
namespace NS1
{
    public class A { }
    public class B { }
    public struct C { }
    public enum D { }
    public delegate Action E();
}
```

Du kan også vælge at benytte to namespaces:



Figur 13 Typer i hvert sit namespace

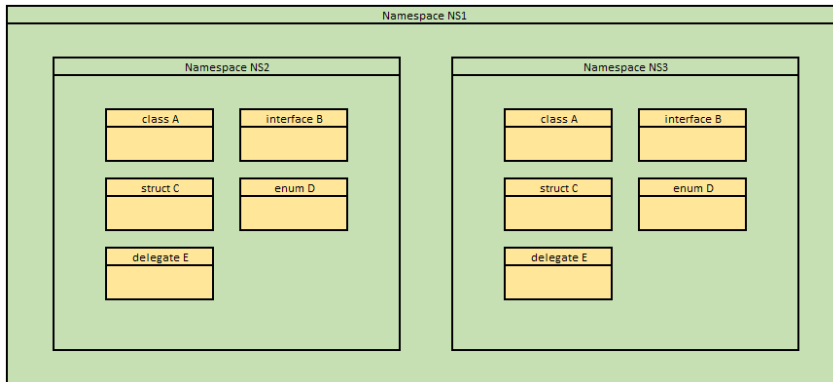
Hvilket kodes som:

```
namespace NS1
{
    public class A { }
    public class B { }
    public struct C { }
    public enum D { }
    public delegate Action E();
}
```

```
namespace NS2
{
    public class A { }
    public class B { }
    public struct C { }
    public enum D { }
    public delegate Action E();
}
```

Ved at benytte flere namespaces kan du skabe en struktur og undgå navnesammenfald. Selv om der er to klasser med navnet A, så er de placeret i hvert sit namespace og hedder i virkeligheden NS1.A og NS2.A – det er to forskellige typer.

Sluttelig kan du vælge en mere kompleks struktur som eksempelvis:



Figur 14 Namespace i namespace

Hvilket kodes som:

```
namespace NS1
{
    namespace NS2
    {
        public class A { }
        public class B { }
        public struct C { }
        public enum D { }
        public delegate Action E();
    }

    namespace NS3
    {
        public class A { }
        public class B { }
        public struct C { }
        public enum D { }
        public delegate Action E();
    }
}
```

Her to forskellige klasser A i både NS2 og NS3, og hedder i virkeligheden NS1.NS2.A og NS1.NS3.A.

Den minder om den Microsoft benytter, hvor System-namespacet svarer til NS1-namespacet.

Det er helt op til dig, hvor du ønsker at placere dine typer. Hvis du ikke angiver et namespace, vil kompilatoren gøre det for dig.

Som begynder behøver du ikke gå så meget op i en fin strukturel opbygning. Placer alle typer i ét namespace til at starte med. Når du skaber en applikation, vil navnet på det overordnede namespace automatisk blive sat til det navn, du har angivet til applikationen.

Helt grundlæggende om tekster

Tekster (kaldet strenge) er en samling af tegn og findes i alle programmeringssprog i en eller anden form. Der kommer meget senere mere om strenge senere, men du vil falde over typen allerede i de første kodeeksempler så her er en kort introduktion.

En streng (datatypen hedder en string) i C# er omkranset af dobbeltplinger (""). Her erklæres en variabel til at indeholde en kort tekst:

```
string tekst = "Dette er tekst";
```

Du kan sammenlægge strenge med + operatoren:

```
string fornavn = "Mathias";  
string efternavn = "Cronberg";  
string navn = fornavn + " " + efternavn;    // Mathias Cronberg
```

Yderligere vil du se brugen af såkaldte string-templates i en masse eksempler på nettet. Det er en simpel og effektiv måde at sammenlægge strenge, og sker ved at danne en skabelon med \$" hvor udtryk og variable kan indsættes med tuborgklammer:

```
string fornavn = "Mathias";  
string efternavn = "Cronberg";
```

```
string navn = $"Mit navn er {fornavn} {efternavn}";  
// Mit navn er Mathias Cronberg
```

Der er meget mere du skal vide om strenge - dette var blot en introduktion så du lige kender lidt til typen.

Helt grundlæggende om metoder

Metoder er i alle programmeringssprog en måde at genbruge kode, og dermed gøre koden mere læsbar og nem at vedligeholde. Du kan se det som en blok kode, der kan afvikles (kaldes) en eller flere gange.

I C# er metoder integreret i en type, og kan enten bestå af kode, der skal afvikles uden en returnværdi (kaldes en void-metode), eller returnere et resultat af en konkret type (tekst, tal, sand/falsk med videre). Yderligere kan en metode have argumenter, ligesom en matematisk funktion, og de angives altid i parentes.

Du skal senere lære at skabe dine egne metoder, men her er for en god ordens skyld et par eksempler på brug af allerede eksisterende metoder, så du kan genkende syntaksen, når du ser den:

```
// Kald til metoden WriteLine på Console-klassen uden argumenter.  
// Metoden har ingen returnværdi (void-metode)  
System.Console.WriteLine();  
  
// Kald til metoden Delete på File-klassen  
// med et enkelt argument. Metoden  
// har ingen returnværdi (void-metode)  
System.IO.File.Delete(@"c:\temp\data.txt");  
  
// kald til metoden Delete på Directory-klassen  
// med to argumenter. Metoden  
// har ingen returnværdi (void-metode)  
System.IO.Directory.Delete(@"c:\temp\", true);  
  
// kald til metoden ReadAllText på File-klassen.  
// Metoden har et argument og returnerer en streng (tekst)  
string a = System.IO.File.ReadAllText(@"c:\temp\data.txt");
```

```
// kald til metoden Max på Math-klassen.  
// Metoden har to argumenter  
// og returnerer en double (kommatal)  
double b = System.Math.Max(5, 4);
```

Bemærk at hvis en metode ikke har nogen argumenter, skal du angive dette med (), og eventuelle argumenter adskilles med komma.

I nogle programmeringssprog kaldes metoder for funktioner eller procedurer, men i C# hedder det metoder, fordi C# er objektorienteret af natur, og alle metoder er en del af en typedefinition. Det dækker over samme begreb – en blok kode der kan kaldes med eventuelle argumenter som kan returnere et resultat, hvis du ønsker det.

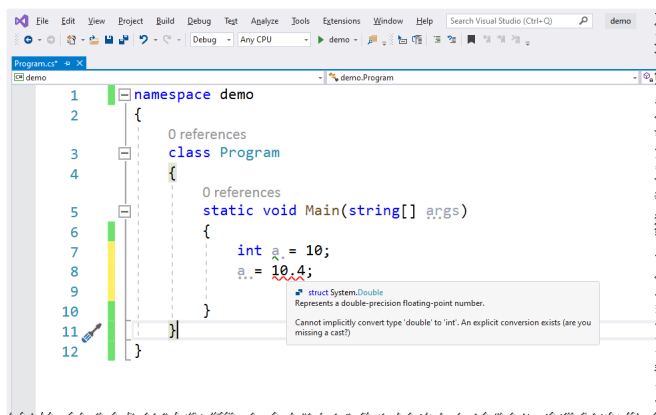
C# er typestærkt

I C# er alt som nævnt baseret på typer – både dem runtime stiller til rådighed, som eksempelvis simple variabler eller typer der bruges i daglig kode, samt dine egne typer. At C# er typestærkt betyder blandt andet, at når du erklærer variabler og felter (variabler i typer), skal du angive en konkret type. Det gælder også for argumenter til metoder samt eventuelle returverdier. Alt er baseret på konkrete typer.

Det kan lyde besværligt, især hvis man kommer fra typesvage sprog som JavaScript eller Python, men det betyder dels at runtime kan optimere afvikling og hukommelsesforbrug, men også at et udviklingsmiljø som Visual Studio kan give en masse hjælp i forbindelse med udvikling:

```
int a = 10;           // erklæring af et heltal  
a = 10.4;             // fejl (kommatal er ikke et heltal)  
a = "test";          // fejl (streng er ikke et heltal)
```

Det er således svært at skrive direkte fejlagtig kode i C#. Fejlen bliver fanget i forbindelse med kompilering – og de store udviklingsmiljøer foretager hele tiden en baggrundskompilering og viser eventuelle fejl.



Figur 15 C# er et typesækket sprog

C# er typesikkert

I C# skal du sjældent bøvle med at skrive kode til at rydde op i hukommelsen – det klarer runtime typisk. Men det betyder også, at runtime skal være meget skrap i styring af tilgang til hukommelsen.

Derfor kan du ikke tilgå værdier i hukommelsen på steder, hvor du ikke har noget at gøre, og anden kode kan heller ikke pille ved dine værdier. Det giver ikke blot sikkerhed for dine data i hukommelsen, det giver også mulighed for at skrive kode, som udnytter den feature:

```
{  
    int a = 10;  
    // her må man gerne tilgå værdien af a  
    a = a + 1;  
}
```

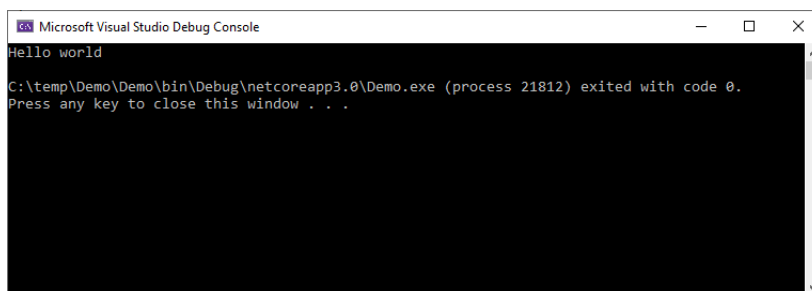
`a = a + 1; // fejl - a findes ikke her`

I koden benyttes et *virkefelt* skabt af tuborgklammer til at fortælle kompileren, at de variabler, der er erklæret i en konkret blok af kode, kun lever der.

Du kan se en blok omkranset af tuborgklammer som en lille sandkasse. Alt, hvad der foregår der, bliver i den sandkasse og kun i den – med mindre du som udvikler giver andre steder i koden en *reference* til en konkret variabel. Mere om det senere i bogen.

En konsol-applikation i C#

For at holde fokus på kode – og ikke brugerflade – har jeg valgt, at hele bogen tager udgangspunkt i en .NET konsol-applikation. Så er der ikke særlig meget brugerflade at skulle holde styr på. Du kan skrive ud til konsol, hente indtastninger fra brugeren, og hvis det går helt vildt for sig, kan du sågar ændre forgrunds- og baggrundsfarve:

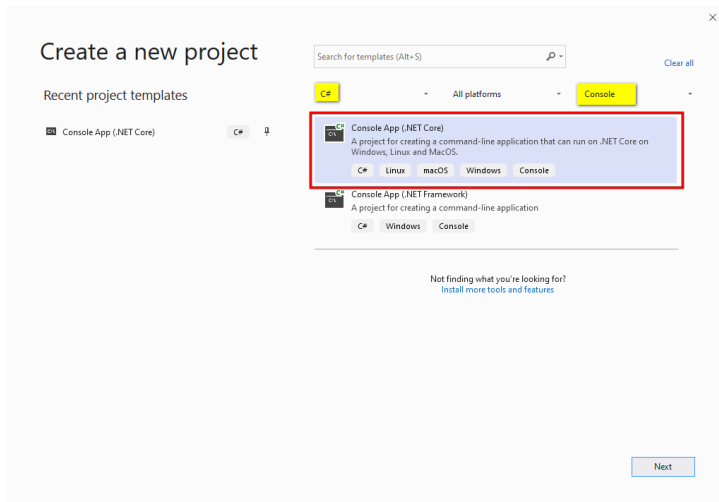


Figur 16 En konsol-applikation

Applikationen er også nem at oprette i både Visual Studio og Visual Studio Code.

En konsol-applikation i Visual Studio

Hvis du benytter Visual Studio, kan du skabe en tom konsol-applikation ved at benytte den i Visual Studio indbyggede skabelon Console App (.NET Core), som du kan finde, når du opretter et nyt projekt (New -> Project på File-menuen):



Figur 17 En konsol-applikation i Visual Studio

Nu oprettes applikationen i Visual Studio og du er klar til at skrive kode.

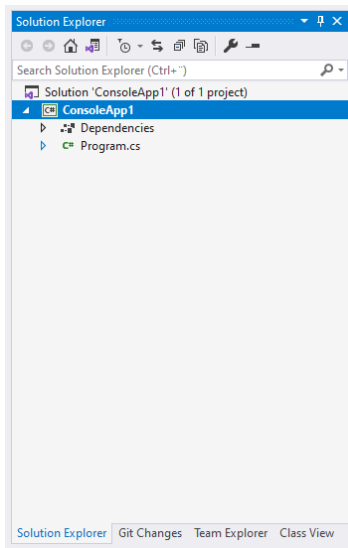
Bemærk, at nævnte beskrivelse er skrevet i starten af 2021 baseret på Visual Studio 2019 version 16.8.3. Det kan være, at skabelonerne i Visual Studio ændrer sig rent navngivningsmæssigt. Følg med på bogens blog for yderligere informationer

<https://www.bogenomcsharp.dk>



Du kan finde en video på [bogenomcsharp.dk](https://www.bogenomcsharp.dk), der viser, hvordan du opretter en konsol-applikation i Visual Studio. Klik på "Video" i menuen.

Når du har oprettet en konsolapplikation, bør du lige kontrollere, at du benytter .NET version 5.x. Det kan du nemmest gøre i Visual Studio ved at dobbeltklikke på selve projektfilen i Solution Explorer-vinduet:



Figur 18 Klik på projektfilen for at tilrette runtime

Her skal du sørge for, at TargetFramework står til "net5.0" eller senere:

```
<Project Sdk="Microsoft.NET.Sdk">
```

```
  <PropertyGroup>
```

```
    <OutputType>Exe</OutputType>
```

```
    <TargetFramework>net5.0</TargetFramework>
```

```
  </PropertyGroup>
```

```
</Project>
```

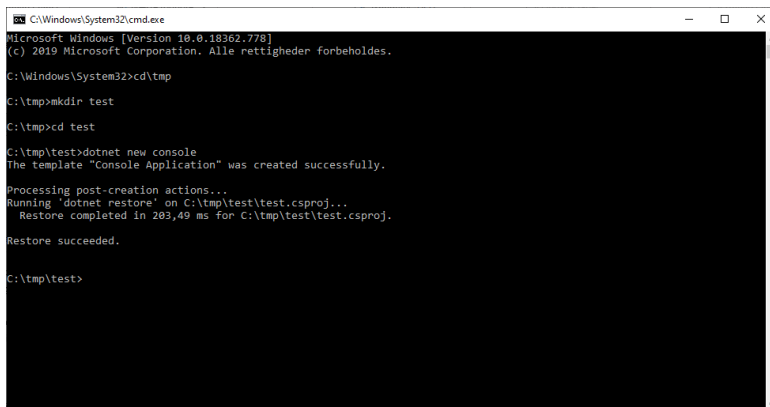
Mon ikke Microsoft tilretter skabelonerne så .NET 5 angives automatisk – men du må hellere lige kontrollere det.

En konsol-applikation i Visual Studio Code

Hvis du benytter Visual Studio Code, skal du åbne en kommando-prompt og navigere til den mappe, du ønsker, applikationen skal placeres i. Herefter skal du på kommandoprompt skrive:

```
dotnet new console
```

og trykke Enter.



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.18362.778]
(c) 2019 Microsoft Corporation. Alle rettigheder forbeholdes.

C:\Windows\System32>cd \tmp
C:\tmp>mkdir test
C:\tmp>cd test
C:\tmp\test>dotnet new console
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on C:\tmp\test\test.csproj...
  Restore completed in 203,49 ms for C:\tmp\test\test.csproj.
Restore succeeded.

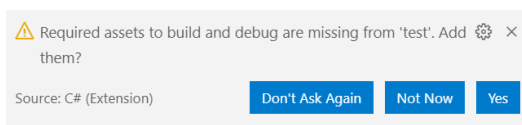
C:\tmp\test>
```

Figur 19 En konsol-applikation i Visual Studio Code (på Windows)

Så oprettes applikationen med de filer, der skal benyttes, og du kan starte Visual Studio Code fra samme konsol ved at skrive:

```
code .
```

Når Visual Studio Code er startet, skal du vente et par sekunder indtil den C# extension, du har installeret, har luret, at der er tale om en C# applikation. Så dukker der en dialogboks op som denne:



Figur 20 En C# konsol-applikation i Visual Studio Code

Når du klikker på Yes-knappen, tilføjes filer til projektet, som er nødvendige i forbindelse med fejlfinding.

Bemærk, at nævnte beskrivelse er skrevet i starten af 2021 baseret på version 5.0 af .NET og version 1.52 af Visual Studio Code. Det kan være ændret, når du læser dette.



Du kan finde en video på bogenomcsharp.dk der viser, hvordan du opretter en konsol-applikation i Visual Studio Code. Klik på "Video" i menuen.

For en god ordens skyld bør du lige kontrollere, at projektfilen (.csproj-filen) henviser til den korrekte version af runtime. Den skal henvise til ".net5.0" eller senere:

```
<Project Sdk="Microsoft.NET.Sdk">
```

```
  <PropertyGroup>
```

```
    <OutputType>Exe</OutputType>
```

```
    <TargetFramework>net5.0</TargetFramework>
```

```
  </PropertyGroup>
```

```
</Project>
```

Det kan være, at det er sket automatisk i den version af .NET SDK, du arbejder med, men primo 2021 skal man tilrette versionen manuelt.

Skabelon til en konsol-applikation

Den kode, der skabes automatisk, når du opretter en .NET konsol-applikation, er i skrivende stund (primo 2021) som følger:

```
using System;
```

```
namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Koden er placeret i filen program.cs, som indeholder et overordnet namespace (navngivet efter navnet på applikationen – her "ConsoleApp1"), som er omkranset af klassen Program. Denne klasse består udelukkende af den statiske metode Main, og koden i denne udgør hele applikationen. Kompileren vil lede efter denne Main-metode, og hvis den ikke kan findes, kan applikationen ikke kompileres. Så du kan altså ikke ændre navnet på metoden.

I .NET 5.x er der dog også en alternativ måde at skabe en konsol-applikation. Du kan vælge at slette både namespace, Program-klasse og Main-metode, og i stedet blot skrive:

```
using System;

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

Det giver præcis samme resultat, idet kompileren vil sørge for at tilføje den manglende kode.

Du vælger selv, om du vil bruge en konsolapplikation med den korte eller lange syntaks. I starten af 2021 benyttes den lange syntaks i den indbyggede skabelon fra Microsoft, men det kan godt ændres med tiden.

En C# løsning

En C# *løsning* (solution på engelsk – løsning på dansk er en lidt skidt oversættelse) består typisk af et eller flere *projekter*, og et enkelt projekt kan være en konsol-applikation (ligesom du oprettede i forrige afsnit), en web-applikation, en Windows-applikation, mobil-applikation og meget andet. En løsning er repræsenteret af en .sln-fil, og du kan finde en visuel repræsentation i Solution Explorer-vinduet. Der kan kun være én løsning ad gangen i VS/VSC, men du må gerne have flere udviklingsmiljøer åbne på samme tid.

Hvis du arbejder med VSC, er der ikke nødvendigvis en .sln-fil, der repræsenterer en løsning. Men den kan tilføjes med kommandoen "dotnet new sln".

Hvert projekt kan være opdelt i flere filer, som kompileren vil sørge for at kompilere sammen. Et projekt er repræsenteret af en .csproj-fil, og består typisk af en eller flere .cs-filer med kode samt eventuelle konfigurationsfiler. Der kan være mange projekter under hver løsning.

Prøv at finde det projekt du oprettede i forrige afsnit i en Stifinder. Hvis det er oprettet med VS, burde du kunne se en filstruktur som følger:

```
c:\temp\MinTest
MinTest.sln
\MinTest
  MinTest.csproj
  Program.cs
```

Hvis du åbner .sln-filen eller csproj-filen i Notepad, vil du kunne se, at det er filer VS bruger til at holde styr på løsning og tilhørende projekter. Du vil sjældent tosse rundt i disse filer, men nu ved du, hvad de består af.

Du må gerne have mange cs-filer (kodefiler) i et projekt, og du kan organisere disse, som du ønsker i mapper. Typisk vil du have en .cs-fil

til hver typedefinition, og dermed ende med mange filer, men du kan også vælge at have flere typer i samme fil – kompilereren er ligeglad.

Using

I toppen af program.cs fra en standard konsol-applikation vil du finde en using-instruktion, og denne instruktion vil du også bruge i alle dine egne filer.

Den fortæller kompileren, i hvilke namespaces den skal lede efter typer, inden den må opgive med en kompileringsfejl. I standard skabelonen er:

```
using System;
```

eksempelvis angivet, og det betyder, at du ikke behøver angive den fulde reference til en type, der ligger lige under System-namespacet.

Uden nødvendigvis at forstå følgende kode til fulde så se dette eksempel:

```
namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            // Beregner 2 opløftet i anden
            System.Double a = System.Math.Pow(2, 2);
            System.Console.WriteLine(a);

            // Henter indhold fra c:\temp\data.txt
            System.String b = System.IO.File
                .ReadAllText(@"c:\temp\data.txt");
            System.Console.WriteLine(b);

            // Henter html fra nettet
            System.Net.WebClient c = new System.Net.WebClient();
            System.String d =
                c.DownloadString("http://www.bogenomsharp.dk");
            System.Console.WriteLine(d);
        }
    }
}
```

```
    }  
  }  
}
```

Koden benytter forskellige typer (Math, File og WebClient) til at foretage forskellige operationer, og der henvises til typernes fulde navn. Eksempelvis er File-klassen placeret i IO-namespacet, som igen er placeret i System-namespacet, og det fulde navn er dermed System.IO.File.

Denne måde at tilgå typer er fuld lovlig, men kan forkortes en del, hvis du i stedet benytter tre using-instruktioner i toppen af koden:

```
using System;  
using System.IO;  
using System.Net;  
  
namespace ConsoleApp1  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            // Beregner 2 opløftet i anden  
            Double a = Math.Pow(2, 2);  
            Console.WriteLine(a);  
  
            // Henter indhold fra c:\temp\data.txt  
            String b = File.ReadAllText(@"c:\temp\data.txt");  
            Console.WriteLine(b);  
  
            // Henter html fra nettet  
            WebClient c = new WebClient();  
            String d = c.DownloadString("http://www.bogenomcsharp.dk");  
            Console.WriteLine(d);  
        }  
    }  
}
```

De tre using-instruktioner betyder, at kompileren eksempelvis leder efter Fil-klassen i de tre angivne namespaces, inden den fejler.

Du må have så mange using-instruktioner, du ønsker, men de skal være placeret i starten af en fil eller i starten af et namespace.

Brug af Console-klassen

Du kommer til at skrive værdier ud på konsollen en del gange, så det er vigtigt, du kender til de metoder, du kan benytte i den sammenhæng.

De er alle fra System.Console-klassen, og den vigtigste er metoden WriteLine. Den skriver blot forskellige værdier (tekster, tal, datoer med videre) ud på konsollen. Den kan eventuelt kombineres med Write-metoden, som også udskriver, blot uden linjeskift:

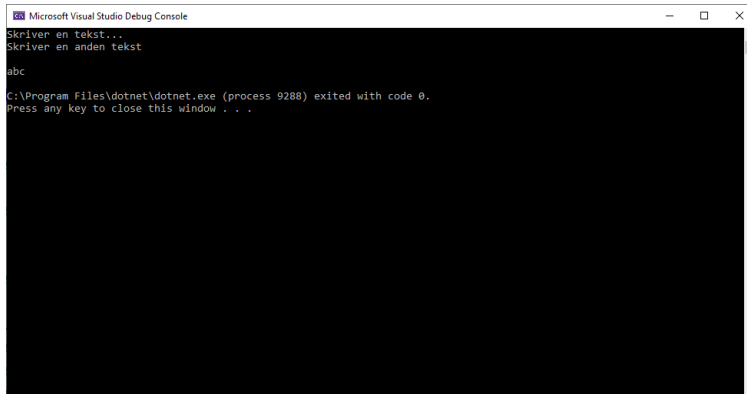
```
using System;

namespace Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            // Udskriver tekst til konsol
            Console.WriteLine("Skriver en tekst...");
            string a = "Skriver en anden tekst";
            Console.WriteLine(a);

            // Tom linje
            Console.WriteLine();

            // Skriver uden linjeskift
            Console.Write("a");
            Console.Write("b");
            Console.Write("c");
            Console.WriteLine();
        }
    }
}
```

Koden vil resultere i følgende:



Figur 21 Udskrift på konsol med `Console.WriteLine`

Metoderne `WriteLine` og `Write` er egentlig alt, hvad du behøver at benytte, men nogle gange kan det være praktisk at kunne hente en værdi fra brugeren, skifte farve og måske endda få computeren til at sige *beep*:

```
using System;
namespace Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            // Læs fra linje
            string input = Console.ReadLine();

            // Afvent at brugeren trykker på en tast
            ConsoleKeyInfo k = Console.ReadKey();
            if (k.Key == ConsoleKey.A) { }
            if (k.Key == ConsoleKey.Escape) { }

            // Farve
            Console.ForegroundColor = ConsoleColor.Red;
```



```
Console.WriteLine("Rød");  
Console.ForegroundColor = ConsoleColor.Gray;  
  
// "beep"  
Console.Beep();  
}  
}
```

Hvis du benytter Console.ReadLine gennem Visual Studio Code bør du tilrette launch.json således, at der benyttes en integreret terminal. Det kan gøres ved at ændre console-egenskaben fra internalConsole til integrated-Terminal.

Bemærk brugen af using – det er nemmere at skrive Console.Beep end System.Console.Beep.



Du kan finde en video på bogenomcsharp.dk der viser, hvordan du kan lege lidt med Console-klassen i en konsolapplikation. Klik på "Video" i menuen.

Hvis du vil vide mere

Når du bliver lidt mere øvet, kan du måske lede efter yderligere information relateret til dette kapitel. Søg eksempelvis efter:

- Alias-kodeordet ved brug af using
- Static-kodeordet ved brug af using
- Asynkrone konsolapplikationer (se også senere i bogen)
- Argumenter til afvikling fra konsol (args-array).

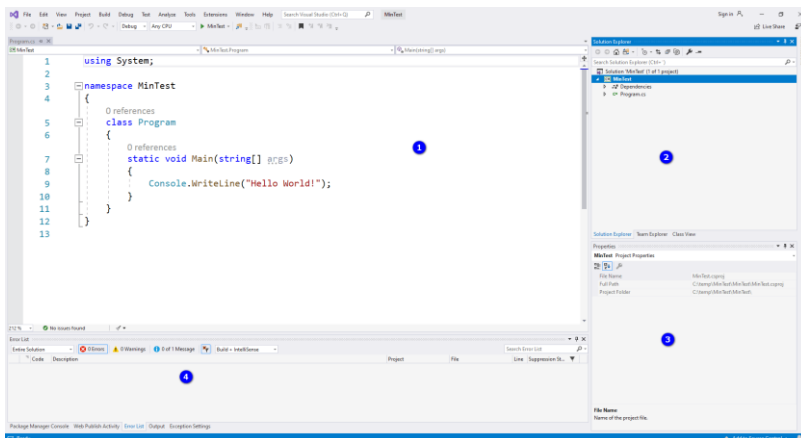
Praktisk brug af VS og VSC

Når du benytter VS (Visual Studio) eller VSC (Visual Studio Code), er det vigtigt, at du udnytter, at et moderne udviklingsmiljø kan hjælpe dig med en masse ting. Derfor er dette kapitel helliget nogle af de features, som begge programmer stiller til rådighed. Det er svært at beskrive i tekst og billeder, så derfor er der referencer til en del videoer, som du også bør bruge lidt tid på.

En tur rundt i Visual Studio

Hvis du er helt ny i brugen af Visual Studio, er det vigtigt, at du kender de basale vinduer og features – så må du bygge på din viden hen ad vejen.

Når du har oprettet en konsol-applikation, som vist i forrige kapitel, vil VS sikkert se nogenlunde således ud (bortset fra mine numre):



Figur 22 De mest benyttede vinduer i VS (version 16.8.3)

Punkt nr. 1 er placeret i vinduet, hvor du kan skrive kode.

Punkt nr. 2 er placeret i Solution Explorer-vinduet, som viser hvilke filer, der kan arbejdes med. I dette simple projekt er der kun program.cs (som ses i kode-vinduet).

Punkt nr. 3 er Properties-vinduet, hvor egenskaber relateret til det valgte element i Solution-vinduet kan ændres.

Punkt nr. 4 er placeret i Error List-vinduet, hvor eventuelle fejl og advarsler i koden kan aflæses. Måske skal du grave lidt efter Error List-vinduet, eller måske finde det på View-menuen.

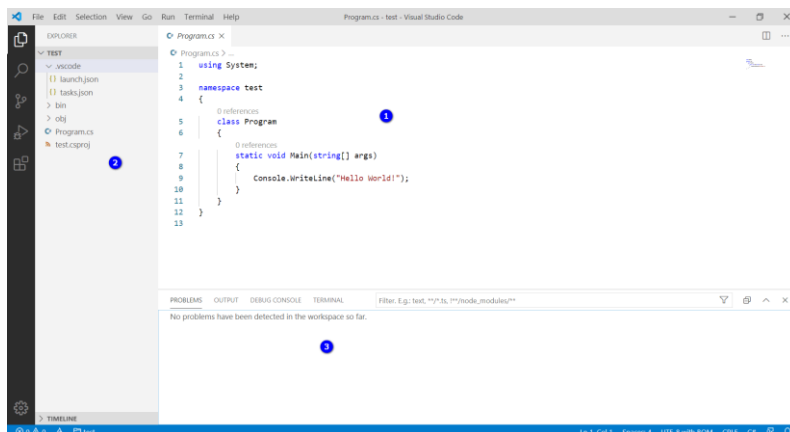
Der er rigtig mange vinduer i VS, men nu har du styr på de mest benyttede.



Du kan finde en video på bogenomcsharp.dk der viser, hvordan du finder rundt i Visual Studio. Klik på "Video" i menuen.

En tur rundt i Visual Studio Code

Hvis du benytter VSC, og har skabt en konsol-applikation som vist i forrige kapitel, vil VSC se nogenlunde således ud:



Figur 23 De mest benyttede vinduer i VSC (version 1.52.1)

Punkt nr. 1 er placeret i vinduet, hvor du kan skrive kode.

Punkt nr. 2 er placeret i Explorer-vinduet, som viser, hvilke filer der kan arbejdes med.

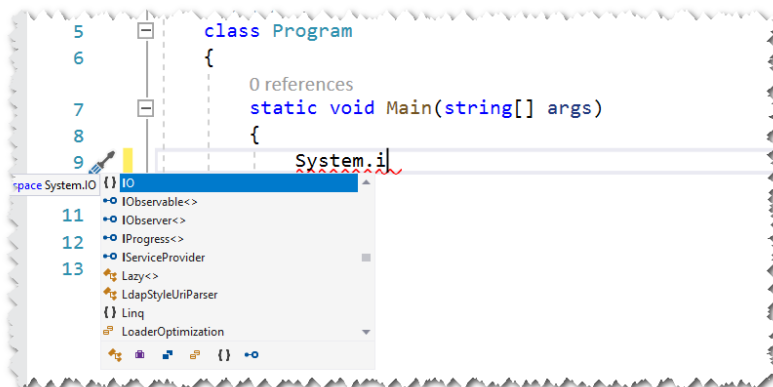
Punkt nr. 3 er placeret i Problems-vinduet, hvor eventuelle fejl og advarsler i koden kan aflæses. Hvis du ikke kan finde vinduet, kan du åbne det fra View-menuen.



Du kan finde en video på bogenomcsharp.dk der viser hvordan dig rundt i Visual Studio Code. Klik på "Video" i menuen.

IntelliSense

Når du skriver kode i VS eller VSC, skal du lære at bruge *IntelliSense*. Det er en feature, som gør det muligt at vælge elementer på en liste, der automatisk dukker op, når du har brug for det:

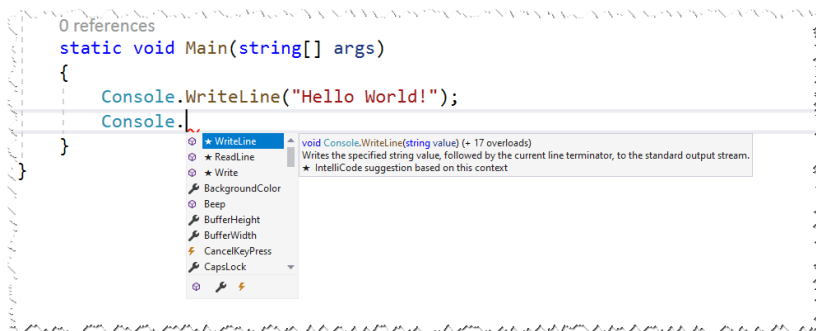


Figur 24 *Intellisense i VS (også tilgængeligt i VCS)*

Det er en af fordelene ved at arbejde med et typestærkt sprog som C# – udviklingsmiljøet ved, hvad en konkret type indeholder, og kan dermed stille en masse hjælp til rådighed.

Når du skriver kode, er det derfor oplagt at lade VS/VSC gøre arbejdet – så kig på skærmen og vælg på listen i stedet for at skrive et ord fuldt ud. Brug piletasterne til at vælge, og tryk Enter eller Tabulering for at indsætte.

Begge udviklingsmiljøer har sågar en feature (*IntelliCode*), hvor den ud fra en indbygget ML (Machine Learning) motor prøver at gætte hvilken kode, du skal bruge:



Figur 25 Machine Learning i VS

Elementerne på listen er alle de medlemmer, som er tilgængelige på Console-klassen, men de tre øverste (med stjerner) er et kvalificeret gæt fra VS.

Hvis IntelliCode ikke er automatisk installeret, kan du søge efter en extension kaldet "Visual Studio IntelliCode".

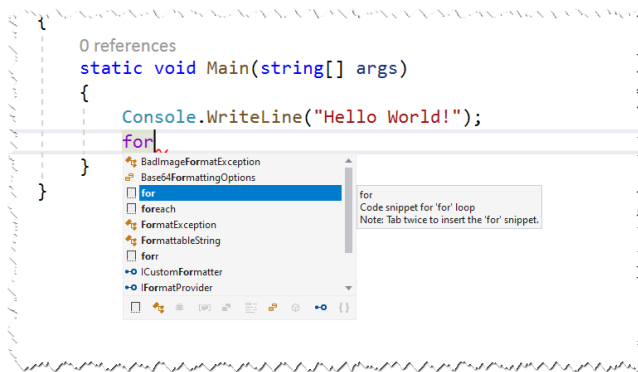


Du kan finde en video på bogenomcsharp.dk, der viser IntelliSense og IntelliCode i VS og VSC. Klik på "Video" i menuen.

Snippets

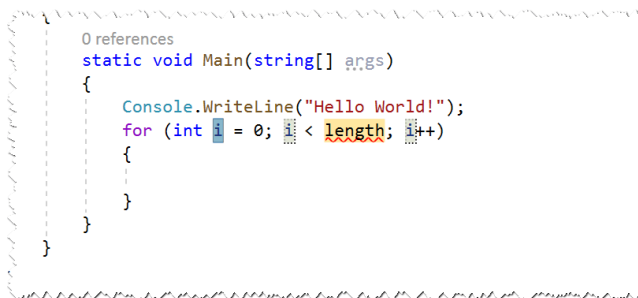
En anden feature, du bliver nødt til at lære at udnytte, er *snippets*. Det er en nem og effektiv mulighed for at indsætte og efterfølgende redigere kode, og hvis du først lærer de mest benyttede C# snippets, kan du skrive kode meget hurtigt.

I kodevinduet kan du eksempelvis skrive "for" og trykke på tabulering to gange:



Figur 26 Brug af en snippet i Visual Studio (1)

Efter at have trykket tabulering to gange indsættes kode, og du kan tabulere mellem de områder i den indsatte kode, som kan redigeres:



Figur 27 Brug af en snippet i Visual Studio (2)

I VS kan du finde de tilgængelige snippets ved at højreklikke et sted i koden og vælge *Insert snippet* fra Snippet-menuen. Der er sågar mulighed for at omkrænse eksisterende kode med en snippet.

Her er nogle af de mere kendte snippets til C#:

Genvej	Funktionalitet
<i>for</i>	<i>For loop</i>
<i>try</i>	<i>try/catch (fejlhåndtering)</i>
<i>do</i>	<i>løkke</i>
<i>if</i>	<i>forgrening</i>
<i>cw</i>	<i>Console.WriteLine (skriv på konsol)</i>

Tabel 2 De mest benyttede snippets i VS/VSC

Når du er blevet lidt mere øvet i VS/VSC og C#, vil du sikkert savne snippets til den kode, du bruger tit, og så kan du oprette dine egne snippets. Søg efter "snippet Visual Studio" eller "snippet Visual Studio Code" for mere information.



Du kan finde en video på bogenomcsharp.dk, der viser snippets i VS og VSC. Klik på "Video" i menuen.

Genvejstaster

Både VS og VSC består af et hav af genvejstaster, og jo flere du bliver bekendt med, jo hurtigere kan du naturligvis skrive kode.

Her er en tabel over de mest benyttede genvejstaster til Visual Studio. Efterhånden som du bliver mere erfaren med at bruge VS, må du komme tilbage til tabellen og prøve de forskellige genveje af i praksis:

Genvej	Funktion
<i>Ctrl + K + D</i>	<i>Formater kode</i>
<i>Ctrl + K + C</i>	<i>Kommentar</i>
<i>Ctrl + K + U</i>	<i>Fjern kommentar</i>
<i>Ctrl + K + X</i>	<i>Indsæt snippet</i>
<i>Ctrl + K + S</i>	<i>Omkrans med (snippet)</i>
<i>Ctrl + K + R</i>	<i>Find alle referencer</i>
<i>Ctrl + .</i>	<i>Quick actions</i>
<i>Ctrl + [space]</i>	<i>IntelliSense</i>
<i>F2</i>	<i>Omdøb</i>
<i>F5</i>	<i>Start med debugger</i>
<i>Ctrl + F5</i>	<i>Start uden debugger</i>
<i>Shift + F5</i>	<i>End debugger session</i>
<i>F6</i>	<i>Build / byg løsning</i>
<i>Shift + F6</i>	<i>Build / byg projekt</i>
<i>F9</i>	<i>Sæt breakpoint</i>
<i>F10</i>	<i>Step over</i>
<i>F11</i>	<i>Step into</i>
<i>F12</i>	<i>Goto definition</i>
<i>Ctrl + Shift + F9</i>	<i>Slet alle breakpoints</i>
<i>Alt + op/ned pil</i>	<i>Flyt kode</i>
<i>Ctrl + Shift + B</i>	<i>Byg (Build)</i>
<i>Ctrl + ,</i>	<i>Find</i>
<i>Ctrl + Tab</i>	<i>Skift mellem vinduer</i>

Tabel 3 Nogle af de mange genveje i VS

Her er en tabel over de mest benyttede genveje i Visual Studio Code:

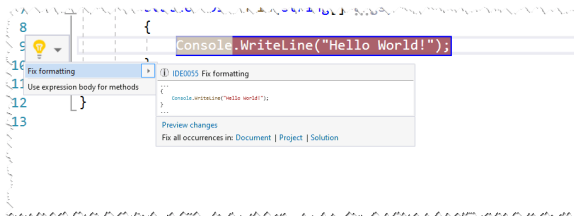
<i>Genvej</i>	<i>Funktion</i>
F1 + "format"	Formater kode
Ctrl + K + C	Kommentar
Ctrl + K + U	Fjern kommentar
F1 + "Insert snippet"	Indsæt snippet
Ctrl + .	Quick actions
Ctrl + [space]	IntelliSense
F2	Omdøb
F5	Start med debugger
Ctrl + F5	Start uden debugger
Shift + F5	End debugger session
F9	Sæt breakpoint
F10	Step over
F11	Step into
F12	Goto definition
Alt + op/ned pil	Flyt kode
Ctrl + Shift + B	Byg (Build)
Ctrl + Tab	Skift mellem vinduer

Tabel 4 Nogle af de mange genveje i Visual Studio Code

Du kan naturligvis også tilrette og oprette dine egne genveje.

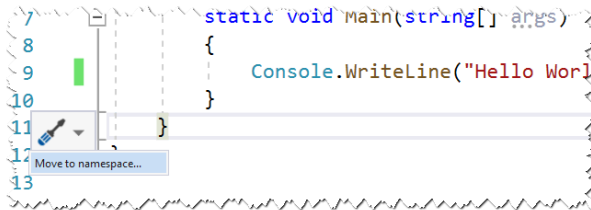
Hjælp fra VS og VSC

Når du koder, vil Visual Studio og Visual Studio Code forsøge at hjælpe dig, så meget den kan. Det sker eksempelvis ved hjælp af et ikon i form af en lysende pære:



Figur 28 Hjælp i Visual Studio (pære-ikon)

Det kan også ske ved hjælp af et ikon i form af en skruenøgle:



Figur 29 Hjælp i Visual Studio (skruenøgle-ikon)

Hvis du klikker på ikonet, vil du få mulighed for at vælge forskellige funktioner.

I starten vil du nok ikke benytte den hjælp, Visual Studio tilbyder, men efterhånden som du bliver mere erfaren så prøv at være lidt nysgerrig og lad Visual Studio ændre koden. Det er en god måde at blive lidt klogere på C# syntaks, og du kan altid fortryde en ændring.



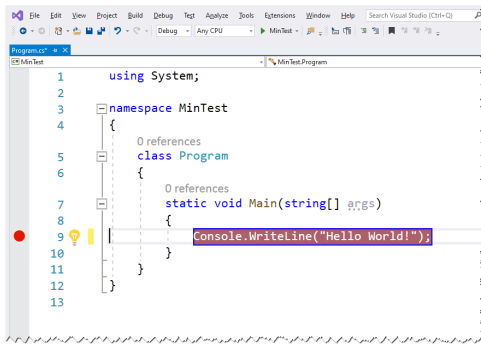
Du kan finde en video på bogenomcsharp.dk, der viser kodehjælp i VS og VSC. Klik på "Video" i menuen.

Debugging

Den måske vigtigste ting i forbindelse med udvikling af applikationer, er brugen af en *debugger*. Så det er vigtigt, at du kender til den .NET og både VS og VSC stiller til rådighed.

Debugging kan være meget kompleks, men i sin simple form handler det om at stoppe afvikling af en applikation, for at kunne aflæse værdier af variable, og så *steppe* sig gennem koden instruktion for instruktion.

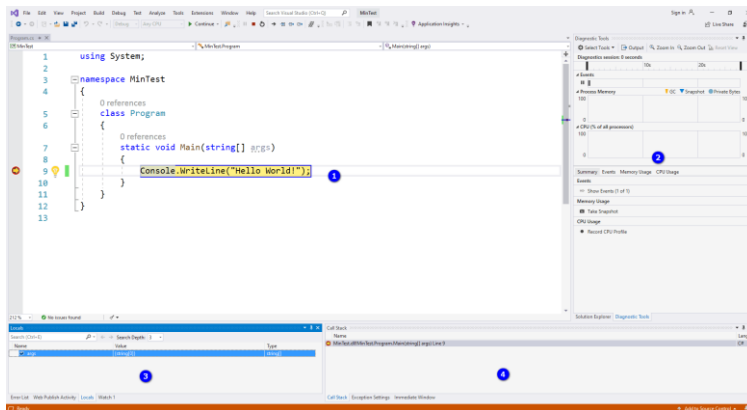
For at stoppe koden skal du sætte et eller flere *breakpoints*, som i VS og VSC kan ses som en rødt markeret linje med en rød cirkel i venstre margin:



Figur 30 Breakpoint i Visual Studio

Et breakpoint er besked om, at afvikling skal stoppe før afvikling af den angivne instruktion. Du må gerne sætte mange breakpoints i en applikation, og i VS kan du gøre det på mange måder. Det nemmeste er at trykke på F9, når du står på en konkret kodelinje. Det vil sætte eller fjerne et breakpoint. På Debug-menuen i VS eller Run-menuen i VSC kan du finde flere muligheder.

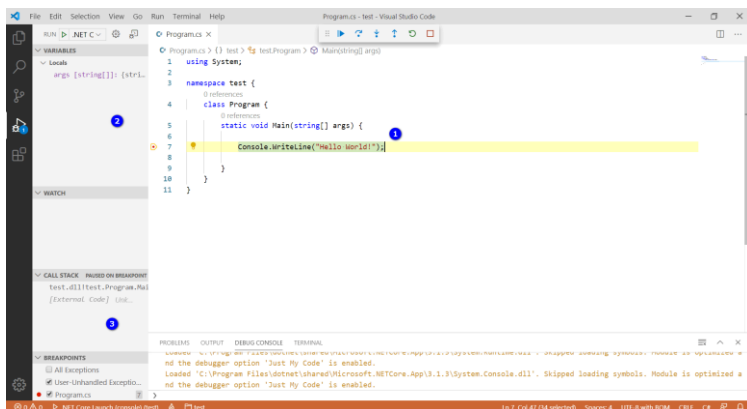
Når du starter en applikation med debuggeren tilknyttet (F5 som nævnt tidligere) vil afvikling stoppe før et breakpoint, og det kan ses i VS/VSC ved at linjen bliver gul:



Figur 31 Visual Studio i breakmode

Når Visual Studio er i breakmode – altså har ramt et breakpoint (punkt 1) – dukker der nye vinduer op. Diagnostics tools-vinduet (punkt 2) giver en masse informationer om hukommelses- og CPU-forbrug, Locals-vinduet (punkt 3) viser information om de variable, som er tilgængelige på det givne sted i koden, og Call Stack-vinduet (punkt 4) giver information om, hvilke metoder der er blevet kaldt for at ende i den aktuelle metode. Der er en masse andre vinduer, som kan være interessante, men Locals- og Call Stack-vinduet er de vigtigste i grundlæggende brug af Visual Studio.

Her er samme vindue i Visual Studio Code:



Figur 32 Visual Studio Code i breakmode

Når Visual Studio Code er i breakmode via et breakpoint (punkt 1), dukker der nye vinduer op. Locals-vinduet (punkt 2) viser information om de variabler, som er tilgængelige på det givne sted i koden, og Call Stack-vinduet (punkt 3) giver information om, hvilke metoder der er blevet kaldt for at ende i den aktuelle metode.

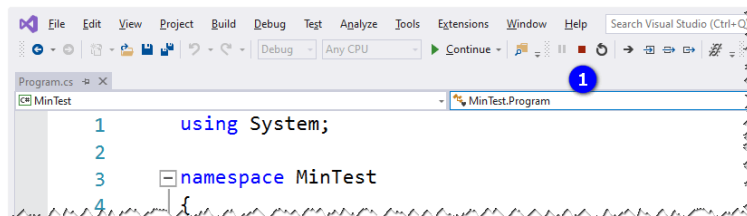
Når afviklingen har ramt et breakpoint, og du eksempelvis har fået aflæst værdierne af variabler, kan du vælge at *steppe* dig videre i koden.

Det kan du gøre med funktioner fra Debug-menuen:

Genvej	Funktion	Beskrivelse
F10	Step over	Afvikler en metode uden at følge de enkelte instruktioner i metoden
F11	Step Into	Afvikler en metode og følger de enkelte instruktioner i metoden
Shift + F11	Step Out	Hopper ud af en metode

Tabel 5 Styring af afvikling under breakmode

Når du er færdig med en debug-session, kan du blot lade applikationen køre færdig (tryk F5 til du er forbi det sidste breakpoint), eller stoppe afviklingen med Shift+F5 (se også Debug-menuen).



Figur 33 Stop breakmode (her VS)

Du kan også klikke på den lille røde firkant (punkt 1) under menuen.



Du kan finde en video på bogenomcsharp.dk, der viser debugging i VS og VSC. Det er vigtigt, at du ser den video, hvis du ikke kender til debugging i forvejen. Klik på "Video" i menuen.

Kompilering af kode

Hvis du blot ønsker at kompilere programmet til senere afvikling, skal du finde *Build solution* (genvejen er F6) eller *Build project* (genvejen er Shift + F6) på Build-menuen i Visual Studio. I Visual Studio Code skal du trykke F1 og skrive build eller trykke Ctrl + Shift + B. Det vil påbegynde en kompilering, og resultatet vil du kunne finde i projektmappen i en debug- eller release-mappe.

Da en kompilering på denne måde typisk sker i forbindelse med en distribuering af applikationen, vil du nok ikke benytte en ren kompilering så meget, men i stedet afvikle applikationen gennem Visual Studio (jævnfør tidligere afsnit).

Hvis du vil vide mere

Når du bliver lidt mere øvet, kan du måske lede efter yderligere information relateret til dette kapitel. Eksempelvis er der mange områder inden for debugging, som er meget avancerede, og som du bør vide noget om som erfaren C# udvikler. Der er masser af interessante videoer på nettet – søg eksempelvis efter *C# advanced debugging* på YouTube.

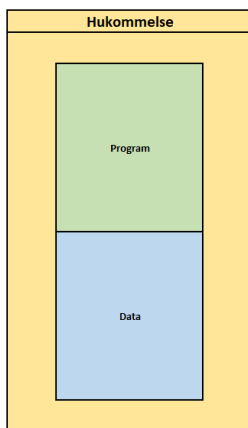
Simple variabler

Alle moderne programmeringssprog giver mulighed for at opbevare midlertidige værdier i hukommelsen, mens en applikation afvikles, og typisk kan man få adgang til disse værdier gennem selvvalgte navne. Det kaldes variabler og dækker i virkeligheden over en ret kompleks funktionalitet – herunder hvor og hvordan værdier skal placeres i hukommelsen, sammenhæng mellem variabelnavne og adresser i hukommelsen, hvilken kode har adgang til hvilke variabler, hvordan og hvornår skal der ryddes op, optimering og meget andet.

Men heldigvis skal du i C# ikke bekymre dig om særlig meget i relation til variabler – i hvert fald ikke i den grundlæggende C#. Det klarer kompiler og runtime for dig.

Hukommelse

Når en C# applikation startes, vil den blive tildelt plads i hukommelsen af runtime og efterfølgende af operativsystemet:



Figur 34 Hukommelse

Der vil både blive afsat (allokeret med et fint ord) plads til de binære instruktioner og de midlertidige data. I C# vil du primært befinde dig i

data-delen, men som du skal se senere, kan du også skabe referencer til metoder (brug af Delegate-typen), som er placeret i program-delen.

I modsætning til nogle andre programmeringssprog vil du i C# typisk ikke have nogen adgang til hukommelsen, medmindre det sker gennem variabler. I andre sprog kan du tilgå hukommelsen gennem pointere (pegepinde), men det er i grundlæggende C# (heldigvis) ikke en mulighed.

C# er jo som tidligere nævnt typestærk og det betyder, at alle variabler skal oprettes (erklæres) af en konkret type. Det sker blandt andet for at runtime ved, hvor meget plads, der skal allokeres, og for at det er nemmere at optimere.

Så når du skriver kode, der anmoder om en variabel, der kan opbevare et heltal, skal du tage stilling til, hvilken type du vil benytte. Den mest benyttede heltalstype i C# er en Int32 (int) og den fylder 32 bit.

Hukommelsen er jo i virkeligheden en masse små registre, som hver kan indeholde 1 bit (og dermed værdien 0 eller 1). Med 8 af disse små registre (også kaldet en byte) kan du opbevare et heltal med værdier fra -127 til 128. På 16 bit kan du opbevare et heltal med værdier fra -32.768 til 32.767. På 32 bit omkring -2.1 milliard til +2.1 milliard, og på 64 bit er der tale om et meget stort negativt eller positivt tal. Søg på Google efter "hukommelse byte" for artikler og videoer, hvis du vil vide mere.

Udover typen skal du også tage stilling til det navn, du gerne vil benytte som tilgang til værdien.

Så når du skriver kode som:

```
int a = 0;
```


anmoder du runtime om at allokere 32 bit i hukommelsen og gemme værdien 0, og hver gang du benytter variabelen *a*, vil du have adgang til det område i hukommelsen. C# er (som også tidligere nævnt) type-sikker, og det betyder, at den allokerede plads, du kan tilgå gennem variabelen, er din, og du kan være sikker på, at intet andet ændrer værdien medmindre, du selv giver mulighed for det.

Hvis du skriver kode som:

```
bool a = false;
double b = 200.23;
DateTime c = new DateTime(2019, 9, 21);
```

anmoder du runtime om at allokere 8 bit til *a* (sand/falsk) samt tildele værdien false, allokere 64 bit til *b* (kommatal) samt tildele værdien 200,23 og allokere 64 bit til *c* (dato og tid) samt tildele værdien 21/9-2019.

Blot til orientering findes der avancerede muligheder i C# for at tilgå hukommelsen direkte gennem en adresse – men det kræver brug af såkaldt *unsafe* kode og bruges meget sjældent. Bare navnet unsafe indikerer jo også, at det ikke er noget, du bør bruge ;)

Variabelnavne

Du kan navngive dine variabler, som du har lyst – med nogle få undtagelser:

- Navnet skal begynde med et bogstav eller en underscore (`_`)
- Navnet må ikke kun bestå af underscores
- Navnet må kun bestå af bogstaver, tal eller underscores
- Man må gerne benytte Æ, Ø, Å og andre tegn.

De fleste udviklere benytter en navngivningsstandard, så der er en eller anden form for fælles standard, men det er helt op til dig. Jeg vil dog

anbefale dig at bruge Microsofts navngivningsstandard⁸, som kan forkortes ned til, at variabler i metoder starter med et lille bogstav, og benytter *CamelCasing* (hvert ord i en variabel skrives med stort bortset fra det første).

Her er et par eksempler:

```
a
antal
månedPrÅr
gennemsnitLøn
gadeOgBy
```

Om du vil *kode* på dansk eller engelsk er helt op til dig. Du må gerne bruge danske bogstaver, men spørgsmålet er, hvor smart det er. I dit helt eget projekt, og i denne bog, er det ligegyldigt, men i et projekt hvor koden måske skal ses af udlændinge, er danske bogstaver ikke så fikst. Men kompileren er altså ligeglad.

Virkefelter

C# er et rigtigt semikolon- og tuborgklammesprog, og tuborgklammerne bruges til at definere en blok kode. Det kaldes også et *virkefelt* (på engelsk *scope*).

Når du erklærer variabler i en metode, kan du antage, at de lever og kan tilgås i denne metode – samt eventuelle indre virkefelter (andre tuborgklammer). Kompileren skal nok sørge for at blokere adgang andre steder fra.

Derfor er det vigtigt, at du har styr på tuborgklammerne. De *definerer* et område i koden, hvor variabler lever.

Se følgende kode:

```
using System;

namespace MinTest
```

⁸ <https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/naming-guidelines>

```
{
    class Program
    {
        static void Main(string[] args)
        {
            // Et virkefelt

            bool a = true;
            if (a == true)
            {
                // Et virkefelt
            }

            for (int i = 0; i < 10; i++)
            {
                // Et virkefelt
            }

            {
                // Et virkefelt
            }
        }
    }
}
```

Bemærk, at metoden Main er et virkefelt, men inde i Main-metoden er der tre andre virkefelter på samme niveau.

Variabler kan kun tilgås i samme eller indre virkefelter:

```
using System;

namespace MinTest
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 1;
            // her kan a tilgås
            {
```

```

    int b = 2;
    // her kan a og b tilgås
    {
        int c = 3;
        // her kan a, b og c tilgås
    }
    // her kan a og b tilgås
}
// her kan a tilgås
}
}
}

```

Hvis du forsøger at få fat på en variabel uden for et virkefelt, vil Visual Studio og kompileren ikke give dig lov.

Det står dig jo frit for at benytte tabuleringer, men som du kan se, giver det et godt overblik over kode og virkefelter. Du kan selv formatere koden, men du vil kunne spare tid ved at lade Visual Studio gøre det. Forudsat at der ikke er nogen fejl i koden, kan du bruge *Format document* fra Advanced-menuen under Edit-menuen. Genvejstasten er Ctrl + E + D eller Ctrl + K + D i Visual Studio. I Visual Studio Code kan du trykke F1 og skrive format.

Heltal

Der findes fire forskellige typer, du kan bruge til at skabe heltal i C#. De kan enten tilgås gennem et reelt typenavn eller et genvejsnavn, som de fleste benytter:

Typenavn	Genvej	Forklaring	Spænd
<i>System.Byte</i>	<i>byte</i>	<i>8-bit uden fortegn</i>	<i>0-255</i>
<i>System.Int16</i>	<i>short</i>	<i>16-bit med fortegn</i>	<i>-32.768 til 32.767</i>
<i>System.Int32</i>	<i>int</i>	<i>32-bit med fortegn</i>	<i>-2.147.483.648 til 2.147.483.647</i>
<i>System.Int64</i>	<i>long</i>	<i>64-bit med fortegn</i>	<i>meget lille tal til meget stort tal</i>

Tabel 6 Heltal i C#

Når du skriver en heltalskonstant (altså blot et tal) i koden vil kompileren opfatte det som en `int`, og i virkeligheden vil du sjældent bruge de andre variabeltyper.

Du undrer dig måske over, at du ikke skal spare så meget plads som muligt og eksempelvis bruge en byte, hvis du blot skal tælle til 10. Men du skal mere kigge på den CPU, som applikationen afvikles på. Hvis der er tale om en 32 bit eller 64 bit arkitektur giver det ikke den store (om nogen) forskel at flytte 32 bit rundt i stedet for 8 bit. Med moderne maskiner kan man faktisk undre sig over, at det ikke er et 64 bit heltal (`long`) som er default i stedet for et 32 bit heltal (`int`). Men det skyldes historik og kompatibilitet.

Hvis du vil erklære en variabel af typen `int`, kan du gøre det på flere måder. Det nemmeste at forstå, og det mest pædagogiske, er således:

```
System.Int32 a = new System.Int32();
```

Koden er logisk og pædagogisk, fordi det er tydeligt, at du fortæller kompileren, at du ønsker en variabel af typen `System.Int32` kaldet *a*, og derefter skaber en ny instans af denne type og gemmer den i variabelen. Variabelen *a* vil blive tildelt en defaultværdi på 0.

Samme kode kunne skrives ved brug af `int`, da `int` er genvejsnavnet til `System.Int32`:

```
int b = new int();
```

Det giver præcis samme resultat – skab variabel kaldet *b* af typen `System.Int32`, skab en ny instans og gem værdien. Variabelen *b* vil også blive tildelt en defaultværdi på 0.

Du må gerne skrive kode således, men Microsoft har givet mulighed for, at man kan skabe variabler af de fleste strukturer på en nemmere måde – uden brug af `new`:

```
System.Int32 a;  
int b;
```

Det er ikke helt så pædagogisk, men nemt og giver samme resultat dog med den forskel, at både *a* og *b* ikke er initialiserede, og skal tildeles en værdi inden brug.

Men prøv at læse koden:

```
int a;
```

som:

```
System.Int32 a = new System.Int32();
```

og tænk: erklær en variabel af typen `System.Int32` kaldet *a*, skab en ny (new) instans og bind variabelen og instansen sammen.

Du kan dog vælge at initialisere variabelen sammen med erklæringen:

```
System.Int32 a = 0;
```

```
int b = 0;
```

De fleste vælger (naturligvis) at bruge genvejsnavnet og initialisere med det samme:

```
int a = 1;
```

Så længe du arbejder med samme type, må du gerne erklære flere variabler i samme instruktion:

```
int a = 0, b = 0, c = 0, d = 0;
```

De fire nævnte datatyper er alle strukturer, hvilket har betydning for, hvor værdier placeres i hukommelsen, og hvad der reelt gemmes. Når der er tale om strukturer, kalder man variabler for værdibaserede variabler, fordi der opbevares værdier, og variabler (og dermed værdier) gemmes i et område i hukommelsen, der kaldes en stak. Det kommer vi tilbage til senere – lige nu skal du bare notere, at strukturer er værdibaserede typer.

Hvis du er i tvivl, om du har fat i en struktur eller en klasse, kan du altid holde musen over datatypen i Visual Studio eller Visual Studio Code. Så fremgår det meget tydeligt, hvilken type der er tale om.

Kommatal

Der findes grundlæggende tre forskellige typer til at håndtere kommatal:

Typenavn	Genvej	Forklaring	Betydende cifre
<i>System.Single</i>	<i>float</i>	<i>32-bit reelt tal</i>	<i>Omkring 7</i>
<i>System.Double</i>	<i>double</i>	<i>64-bit reelt tal</i>	<i>Omkring 15</i>
<i>System.Decimal</i>	<i>decimal</i>	<i>128-bit reelt tal</i>	<i>Omkring 28</i>

Tabel 7 Kommatal i C#

Både float og double benytter den såkaldte floating-point standard, som i de fleste programmeringssprog benyttes til at repræsentere reelle tal. De betydende cifre (før eller efter komma) er afhængig af størrelsen, men generelt kan en double indeholde et meget stort (få decimaler) eller meget lille (mange decimaler) tal. Kommatal skrives med punktum, når du benytter konstanter i koden:

```
float a = 0;  
double b = 0;  
decimal c = 0;
```

```
b = 233.2341;
```

Typen double er mest benyttet og er også standard i C#, men decimal kan bruges, hvis beregninger kræver en præcis håndtering af decimaler og styring af afrunding. En double kan i nogen situationer være udfordrende grundet den tilhørende algoritme, men til gengæld er den superhurtig sammenlignet med en decimal. Det klassiske eksempel på en double-afrundingsfejl er:

```
double a = .1 + .1 + .1 + .1 + .1 + .1 + .1 + .1 + .1 + .1;
```

```
// a = 0.9999999999999999
```

Foretager man samme beregning med en decimal, er resultatet 1.

Du behøver i langt de fleste tilfælde ikke være så bevidst om eventuelle afrundingsproblemer på 16. eller 17. decimal og blot benytte en double til de dine kommatil.

Operatorer relateret til tal

Når først du har fat i heltal eller reelle tal, kan du naturligvis foretage diverse beregninger ved hjælp af indbyggede operatorer:

Operator	Forklaring
+	<i>Plus</i>
-	<i>Minus</i>
*	<i>Gange</i>
/	<i>Division</i>
%	<i>Modulus (returnerer resten ved en division)</i>
+=	<i>Adderer og tildeler værdien af en variabel med en værdi</i>
-=	<i>Subtraherer og tildeler værdien af en variabel med en værdi</i>
*=	<i>Multiplikerer og tildeler værdien af en variabel med en værdi</i>
/=	<i>Dividerer og tildeler værdien af en variabel med en værdi</i>
++	<i>Forøger en variabel med én</i>
--	<i>Formindsker en variabel med én</i>

Tabel 8 Operatorer relateret til tal

Her er et par eksempler på brug af operatorerne:

```
int a = 10; // a = 10
a = a + 10; // a = 20
a += 10;    // a = 30
```

```
int b = 50; // b = 50
b = b - 10; // b = 40
b -= 10;    // b = 30
```

```
int c = 10; // c = 10
c = c * 2;  // c = 20
c *= 2;     // c = 40
```



```
c /= 4;    // c = 10
```

```
int d = 10; // d = 10
```

```
d++;      // d = 11
```

```
d--;      // d = 10
```

De fleste er logiske og nemme at arbejde med. Det eneste, du skal være opmærksom på, er division. To heltal divideret med hinanden giver et nyt heltal, men hvis det ene er et reelt tal, returneres et reelt tal:

```
int a = 10;
```

```
int b = 3;
```

```
int c = a / b; // c = 3
```

```
double d = 3.0;
```

```
// fejl - udtryk returnerer en double og passer ikke ind i en int
```

```
// int e = a / d;
```

```
double e = a / d; // e = 3.333333333
```

Bemærk fejlen! Udtrykket `a / d` (int/double) returnerer en double, og du kan ikke bare putte et 64-bit (floating point) tal ind i et 32-bit heltal. Det vil kræve en typekonvertering.

Hvis du gerne vil vide lidt mere om tal og operatorer i C#, kan du søge efter info om "overflow/checked", som dækker over hvad der sker, når en variabel eksempelvis rammer loftet af sin maksimale værdi. Der findes også mere avancerede datatyper som `UInt16`, `UInt64`, store heltal i `BigInteger`-typen samt avancerede komplekse tal i `Complex`-typen.

Formatering af tal

Du har tit behov for at gemme eller udskrive et tal i formateret form – med eller uden separator, i et givet antal decimaler og i en given kultur (dansk, tysk, amerikansk med videre).

Flere metoder i C# tager såkaldte formateringstegn som argument, og dem kan du bruge for at slippe for selv at formatere tal. Her er nogle af de vigtigste:

Tegn	Forklaring	Eksempel (DK)
N (antal decimaler)	Brug separator for tusinde	<i>N2 = 100.000,00</i>
F (antal decimaler)	Brug ikke separator for tusinde	<i>F3 = 100000,000</i>
C (antal decimaler)	Formater som valuta	<i>C2 = 100.000,00 kr.</i>
P (antal decimaler)	Formater som procent	<i>P4 = 3,1415 %</i>

Tabel 9 Tegn til formatering af tal

En af metoderne, der benyttes meget i forbindelse med formatering, er ToString-metoden, som findes på alle typer:

```
int a = 25123;
double b = 232345.3426;
double c = 0.25;

Console.WriteLine(a.ToString("N2")); // Udskriver 25.125,00
Console.WriteLine(a.ToString("N3")); // Udskriver 25.125,000
Console.WriteLine(a.ToString("F1")); // Udskriver 25125,0
Console.WriteLine(a.ToString("C2")); // Udskriver 25.125,00 kr.

Console.WriteLine(b.ToString("N2")); // Udskriver 232.345,34
Console.WriteLine(b.ToString("N3")); // Udskriver 232.345,343
Console.WriteLine(b.ToString("F5")); // Udskriver 232345,34260

Console.WriteLine(c.ToString("P0")); // 25 %
Console.WriteLine(c.ToString("P2")); // 25,00 %
```

Samme metode kan også benyttes til forskellige kulturer:

```
int a = 25123;
double b = 232345.3426;
double c = 0.25;
```

```
System.Globalization.CultureInfo culture = new
    System.Globalization.CultureInfo("en-US");

Console.WriteLine(a.ToString("N2", culture));    // Udskriver 25,123.00
Console.WriteLine(a.ToString("N3", culture));    // Udskriver 25,123.000
Console.WriteLine(a.ToString("F1", culture));    // Udskriver 25123.0
Console.WriteLine(a.ToString("C2", culture));    // Udskriver $25,123.00

Console.WriteLine(b.ToString("N2", culture));    // Udskriver 232,345.34
Console.WriteLine(b.ToString("N3", culture));
// Udskriver 232,345.343
Console.WriteLine(b.ToString("F5", culture));
// Udskriver 232345.34260

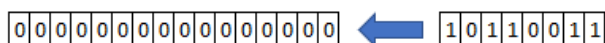
Console.WriteLine(c.ToString("P0", culture));    // 25%
Console.WriteLine(c.ToString("P2", culture));    // 25,00%
```

Du bør angive en kultur, når du formaterer, for du kan ikke styre, hvilken maskine din applikation afvikles på, og det kan give forskellige udfordringer. Koden for den danske kultur er "da-DK", og du kan finde alle de andre hos Microsoft⁹. Der findes også en del andre formateringssteg, som du ligeledes kan finde i dokumentationen¹⁰.

Typekonvertering af tal

Du har tit brug for at konvertere én datatype til en anden, og det kan ske implicit eller eksplicit.

Implicit datakonvertering betyder konvertering af tal fra en lille datatype (eksempelvis en 8-bit byte) til en stor datatype (eksempelvis en 32-bit int):



Figur 35 Implicit datakonvertering

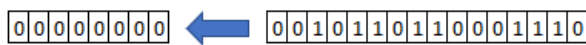
⁹ <https://docs.microsoft.com/en-us/dotnet/api/system.globalization.cultureinfo>

¹⁰ <https://docs.microsoft.com/en-us/dotnet/standard/base-types/standard-numeric-format-strings>

Her behøver du ikke foretage dig andet end at tildele den store variabel værdien af den lille:

```
byte a = 10;  
int b = a;
```

Det er straks værre den anden vej – fra en stor datatype til en lille datatype, fordi et tal repræsenteret på eksempelvis 32 bit kan risikere ikke at kunne være på 8-bit. Det hedder eksplicit datakonvertering:



Figur 36 Eksplicit datakonvertering

Her eksempelvis fra int til byte:

```
int a = 10;  
// byte b = a; FEJL
```

Ved eksplicit datakonvertering bliver du nødt til at hjælpe kompileren, og det kan du gøre på flere måder. Det anbefalede er at benytte metoder fra klassen `System.Convert` således, at afrunding og fejlhåndtering sker på den rigtige måde. På klassen findes der eksempelvis metoder som `ToByte`, `ToInt32`, `ToDouble` og så videre, og disse metoder kan typisk kaldes på mange forskellige måder med mange forskellige datatyper.

Her er et par eksempler fra int til byte med brug af `ToByte`:

```
int a = 10;  
byte b = System.Convert.ToByte(a);  
// det går fint - 10 er nu placeret i en byte  
int c = 300;  
byte d = System.Convert.ToByte(c);  
// vil fejle - 300 ikke kan være i en byte
```

Der er rigtig mange metoder på klassen, du kan bruge til at komme fra en datatype til en anden:

```
byte a = 0;
```

```
short b = 0;
int c = 0;
long d = 0;
a = System.Convert.ToByte(b);
a = System.Convert.ToByte(c);
a = System.Convert.ToByte(d);

double e = 3434.45;
float f = System.Convert.ToSingle(e);
```

og hvis du forsøger at konvertere tal, der kræver afrunding, vil Convert-metoderne også klare det:

```
double a = 100.96;
int b = System.Convert.ToInt32(a); // b = 101
```

Sluttelig vil du nogle gange skulle hjælpe kompileren med at konvertere konstanter. Her kan du benytte et bogstav i slutningen af konstanten for at fortælle kompileren, at den skal opfatte tallet som en konkret type:

Kode	Type
<i>L eller l</i>	<i>Konstanten er en long</i>
<i>F eller f</i>	<i>Konstanten er en float</i>
<i>D eller d</i>	<i>Konstanten er en double</i>
<i>M eller m</i>	<i>Konstanten er en decimal</i>

Tabel 10 Kode for datatype til brug i konstanter

Det kan eksempelvis bruges som følger:

```
long a = 1001L; // L = long
float b = 1002.34F; // F = float
```

Hvis du ikke angiver et bogstav, vil kompileren opfatte heltal som int og reelle tal som double.

Brugen af disse *konstantkoder* er især brugbart ved metodekald, hvor du skal sende en konkret type med, samt ved erklæring af variabler med *var*-kodeordet, som vi kigger på senere:

```
float res = LægSammen(10.4F, 20.7F);
```

```
float LægSammen(float a, float b) {  
    return a + b;  
}
```

I koden kaldes en metode, der har argumenter af typen float. Hvis metoden skal kaldes med konstanter og ikke variabler, bliver du nødt til at fortælle kompileren, at 10,4 og 20,7 ikke er af typen double (default), men derimod float.

Der findes ligeledes konstantkoder til at konvertere binære (0b) og hexadecimaler (0x) konstanter. Se mere i dokumentationen på MSDN¹¹.

Sand eller falsk

Hvis du skal bruge en variabel, der kan få værdien sand eller falsk, kan du benytte 8-bit typen System.Boolean. De fleste benytter dog genvejsnavnet bool, og en variabel af denne type kaldes typisk for en boolsk variabel:

<i>Typenavn</i>	<i>Genvej</i>	<i>Størrelse</i>
<i>System.Boolean</i>	<i>bool</i>	<i>8-bit</i>

Tabel 11 Type til at indeholde en sand eller falsk værdi

Den er simpel i brug, fordi den kun kan tildeles værdien sand eller falsk og ikke som i nogen andre sprog andre værdier, som så typekonverteres automatisk.

Som i de fleste simple variabeltyper kan du vælge, om du vil benytte typenavnet (System.Boolean) eller genvejsnavnet (bool):

```
System.Boolean a;  
a = true;
```

```
bool b;
```

¹¹ <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/>

```
b = false;
```

```
bool c = true;
```

```
bool d, e, f;
```

Husk, at når du ser koden

```
bool a = true;
```

står der egentlig

```
System.Boolean a = new System.Boolean();  
a = true;
```

Det er lidt mere logisk at læse (giv mig en variabel a, der kan indeholde en System.Boolean – og skab en ny instans og tildel den til variabelen).

Ved du hvorfor en sand/falsk variabeltype hedder en bool eller boolean i de fleste sprog? Fra 1815 til 1864 levede en engelsk matematiker og filosof ved navn George Boole. Han var professor ved universitetet Queen's College i Cork i Irland og er ophavsmand til en speciel form for algebra, der benytter operatoren og variabler med de to logiske værdier – sand eller falsk. Hans tanker og teorier ligger til grund for udvikling af logiske porte, som igen er grundlaget for CPU-konstruktion. Hvis du ikke kender ham, så læs om ham på nettet¹², eller se dokumentarudsendelsen "The Genius of George Boole - How to be a Genius" på YouTube¹³.

Boolske operatoren

En boolsk variabel benyttes især ved styring og kontrol af programflow. Kode som *hvis dette er sandt så gør dette ellers så gør dette*, er

¹² https://en.wikipedia.org/wiki/George_Boole

¹³ https://www.youtube.com/watch?v=Hljir_TyTEw (link kontrolleret ultimo 2019)

grundlaget for al programmering. Der findes derfor et par logiske operatører, som du skal kende til:

Operator	Forklaring
==	<i>Lig med</i>
!=	<i>Forskellig fra</i>
&&	<i>And (og) – begge værdier skal være sande for at returnere sand</i>
 	<i>Or (eller) – kun en værdi skal være sand for at returnere sand</i>
!	<i>Not (modsat værdi)</i>
>	<i>Større end</i>
<	<i>Mindre end</i>
>=	<i>Større end eller lig med</i>
<=	<i>Mindre end eller lig med</i>

Tabel 12 Logiske operatører

De benyttes typisk i if-instruktioner, som du skal se senere, men kan også benyttes til at tildele værdier til variabler:

```
bool a = true;      // a = true
bool b = false;     // b = false

bool c = a == b;    // c = false
bool d = a != b;    // d = true
bool e = a && b;     // e = false
bool f = a || b;    // f = true
bool g = !b;        // g = true
```

Du kan naturligvis kombinere operatørerne, som du vil:

```
bool a = true;      // a = true
bool b = false;     // b = false
bool c = !(a == b) && true; // c = true
```

Den sidste linje ser teknisk ud, men skal læses som *den modsatte værdi af true=false OG true* kan forkortes til *den modsatte værdi af false OG true*, som kan forkortes til *true OG true*, som er true.

Husk at operatoren AND (&&) returnerer sand, hvis begge (alle) værdier er sande, og OR (||) returnerer sand, hvis kun én værdi er sand. NOT (!) operatoren returnerer den modsatte værdi.

Dato

Opbevaring, og især beregning, af dato og tid kan være lidt af en udfordring, og derfor stiller .NET flere strukturer til rådighed, der kan hjælpe.

System.DateTime er en 64 bit struktur, som repræsenterer dato og tid fra 01-01-0001 til 31-12-9999, og består i virkeligheden af et meget stort tal, der dækker antallet af såkaldte ticks (100 nanosekunder). Du arbejder dog sjældent med ticks, men benytter instanser af strukturen som en repræsentation af dato og tid:

Typenavn	Genvej	Størrelse
System.DateTime	-	64-bit

Tabel 13 System.DateTime

System.DateTime har mystisk nok ikke noget genvejsnavn (som int eller bool), og der er heller ikke nogen måde at skrive en dato og tid som en konstant. Du er for det meste tvunget til at initialisere værdien ved hjælp af strukturens konstruktør (den kode der afvikles, når der skabes en ny instans), og dem er der en del af. De meste brugte er som følger:

```
// Ikke initialiseret
System.DateTime a;

// Samme som ovenfor - men kræver "using System;"
DateTime b;

// Initialiseret til 1-1-1 0:0:0
DateTime c = new DateTime();

// d initialiseret til 15/10-2019
```

```
DateTime d = new DateTime(2019, 10, 15);

// e initialiseret til 15/10-2019 kl. 8:15
DateTime e = new DateTime(2019, 10, 15, 8, 15, 0);
```

Der findes en del andre måder at initialisere en DateTime-variabel på, men mange er relateret til brug af forskellige tidszoner og kalendere og ligger uden for denne bogs rammer.

DateTime-typen har ligeledes nogle statiske medlemmer, som kan benyttes ved initialisering:

```
// Lokal systemtid (dato og tid)
DateTime a = DateTime.Now;

// Lokal systemdato
DateTime b = DateTime.Today;
```

Når først du har fat i en DateTime-variabel, er der en masse praktiske egenskaber og metoder på selve instansen til rådighed:

```
// Lokal systemtid (dato og tid)
DateTime a = DateTime.Now;
int dagIMåned = a.Day;
DayOfWeek ugedag = a.DayOfWeek;
int time = a.Hour;
int minut = a.Minute;
int month = a.Month;
int sekund = a.Second;
int år = a.Year;
```

Nogle metoder returnerer en ny DateTime-variabel:

```
// Lokal systemtid (dato og tid)
DateTime a = DateTime.Now;
DateTime b = a.Date;           // datodelen (tid fjernes)
DateTime c = a.AddDays(1);
DateTime d = a.AddDays(-1);
DateTime e = a.AddHours(10);
DateTime f = a.AddMinutes(100);
DateTime g = a.AddSeconds(30000);
```

Som det fremgår, kan man trække enheder fra ved at benytte Addxxx-metoderne med et negativt tal.

Det er vigtigt at huske på, at metoder som Addxxx returnerer en ny DateTime variabel, som du skal håndtere på en eller anden måde:

```
DateTime a = new DateTime(2019, 10, 17);  
Console.WriteLine(a.ToShortDateString()); // 17-10-2019  
a.AddDays(1);  
Console.WriteLine(a.ToShortDateString()); // 17-10-2019  
  
a = a.AddDays(1);  
Console.WriteLine(a.ToShortDateString()); // 18-10-2019
```

Du kan måske undre dig over, at linjen:

```
a.AddDays(1);
```

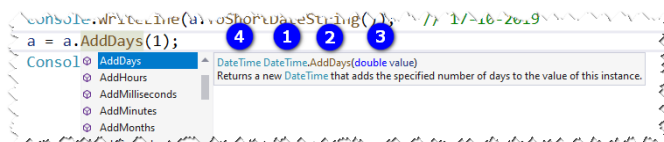
ikke tilretter variabelen a, men det skyldes, at AddDays-metoden ikke tilretter den eksisterende værdi, men returnerer en helt ny, og at denne nye værdi ignoreres.

På linjen:

```
a = a.AddDays(1);
```

gemmes værdien i en variabel. Det kunne være en helt ny variabel, men hvis du ikke har noget at bruge den gamle værdi til, kan du lige så godt genbruge variabelen.

Hvis du er i tvivl om, hvad en metode returnerer, bliver du nødt til at se i dokumentationen, eller være opmærksom på hvad Visual Studio fortæller dig:



Figur 37 AddDays-metoden returnerer en ny DateTime

Bemærk, at Visual Studio fortæller dig, at der på instanser af `DateTime` (1) findes metoden `AddDays` (2), der som argumenter tager en `double` (3), og at den returnerer en `DateTime` (4). Læs også beskrivelsen af metoden.

I programmeringsteori kalder man dette for *immutable* data fordi den underliggende værdi ikke kan rettes, men skal tildeles en ny. `DateTime`-typen er, ligesom mange andre simple variabeltyper, en immutabel datatype, og årsagerne til, at nogle datatyper er immutable og andre er mutable, er relateret til sikkerhed, performance og optimering.

At arbejde med datoer fra den gregorianske kalender i samme tidszone er nogenlunde simpelt, men at arbejde med datoer hen over tidszoner kan være noget fnidder. `DateTime`-strukturen har medlemmer relateret til UTC, men måske skulle du i stedet se på `System.DateTimeOffset`¹⁴, som er bygget til at håndtere beregninger med tidszoner.

Tid

Hvis du vil repræsentere tid, kan du bruge `System.TimeSpan`-strukturen – enten for blot at repræsentere tid som en værdi eller som resultatet af en beregning:

```
// uinitialiseret
TimeSpan a;

// initialiseret til 0:00
TimeSpan b = new TimeSpan();

// initialiseret til 10 timer, 15 minutter og 25 sekunder
TimeSpan c = new TimeSpan(10, 15, 25);

// initialiseret til 1 dag, 10 timer, 15 minutter og 25 sekunder
```

¹⁴ <https://docs.microsoft.com/en-us/dotnet/api/system.datetimeoffset>

```
TimeSpan d = new TimeSpan(1, 10, 15, 25);
```

Og ligesom DateTime er der flere statiske medlemmer, du kan bruge til initialisering:

```
TimeSpan a = TimeSpan.FromSeconds(1000);
TimeSpan b = TimeSpan.FromMinutes(100);
TimeSpan c = TimeSpan.FromDays(2);
TimeSpan d = TimeSpan.FromHours(3);
TimeSpan e = DateTime.Now.TimeOfDay;           // system tid
```

Et TimeSpan-objekt kan også komme fra en beregning mellem to DateTime-variabler:

```
DateTime a = new DateTime(2019, 1, 1);
DateTime b = new DateTime(2019, 8, 28);
TimeSpan c = b.Subtract(a);
TimeSpan d = b - a; // samme som Subtract
```

Da TimeSpan repræsenterer tid, kan den jo opfattes på forskellig måde (minutter, timer, dage, måneder med videre), og derfor indeholder TimeSpan instanser forskellige egenskaber relateret til, hvordan du ønsker at repræsentere tid:

```
DateTime a = new DateTime(2019, 1, 1, 15, 20, 0);
DateTime b = new DateTime(2019, 8, 28, 8, 0, 0);
TimeSpan c = b - a;

int antalMinutter = c.Minutes;
// 40 (antal minutter mellem 0 og 20 minutter)

double totalAntalMinutter = c.TotalMinutes;
// 343.780 (total antal minutter mellem a og b)

int antalTimer = c.Hours;           // 16
double totalAntalTimeDouble = c.TotalHours; // 5.728,66

int antalDage = c.Days;             // 238
double TotalAntalDage = c.TotalDays; // 238,69
```

Og sluttelig en masse muligheder for at regne på tid.

```
TimeSpan a = new TimeSpan(8, 0, 0);
```

```
TimeSpan b = a * 5;
```

```
// 8 timer * 5
```

```
TimeSpan c = a / 2;
```

```
// 8 timer / 2
```

```
TimeSpan d = a.Add(new TimeSpan(0, 90, 0));
```

```
// 8 timer + 90 minutter
```

```
TimeSpan e = a.Subtract(new TimeSpan(0, 90, 0));
```

```
// 8 timer - 90 minutter
```

Formatering af dato og tid

En instans af en `DateTime` indeholder metoderne `ToShortDateString`, `ToLongDateString`, `ToShortTimeString` og `ToLongTimeString`, som du kan bruge til en hurtig formatering, men ligesom tal kan dato og tid også udskrives eller gemmes i formateret form ved brug af formateringstegn.

Der er rigtig mange at vælge imellem, men her er de vigtigste:

Tegn	Forklaring
<i>dd</i>	<i>Dag med eventuelt foranstillet 0</i>
<i>ddd</i>	<i>Kort navn på dag (ma, ti ...)</i>
<i>dddd</i>	<i>Langt navn på dag (mandag, tirsdag, ...)</i>
<i>MM</i>	<i>Måned med eventuelt foranstillet 0</i>
<i>MMM</i>	<i>Kort navn på måned</i>
<i>MMMM</i>	<i>Langt navn på måned</i>
<i>y</i>	<i>År (9)</i>
<i>yy</i>	<i>År med eventuelt foranstillet nul (09)</i>
<i>yyyy</i>	<i>Langt år (2009)</i>
<i>mm</i>	<i>Minut med eventuelt foranstillet 0</i>
<i>HH</i>	<i>Time med eventuelt foranstillet 0</i>
<i>ss</i>	<i>Sekund med eventuelt foranstillet 0</i>

Tabel 14 Tegn til formatering af dato og tid

Du kan eventuelt benytte ToString-metoden til formatering:

```
// Forudsætter afvikling på en dansk maskine med dansk (da-DK) kultur
DateTime a = new DateTime(2019, 10, 17, 13, 37, 25);
```

```
// brug af indbyggede metoder
Console.WriteLine(a.ToShortDateString()); // 17-10-2019
Console.WriteLine(a.ToLongDateString()); // 17. oktober 2019
Console.WriteLine(a.ToShortTimeString()); // 13:37
Console.WriteLine(a.ToLongTimeString()); // 13:37:25

// brug af ToString og formateringsstegn
Console.WriteLine(a.ToString()); // 17-10-2019 13:37:25
Console.WriteLine(a.ToString("dd")); // 17
Console.WriteLine(a.ToString("ddd")); // to
Console.WriteLine(a.ToString("dddd")); // torsdag
Console.WriteLine(a.ToString("MM")); // 10
Console.WriteLine(a.ToString("MMM")); // okt
Console.WriteLine(a.ToString("yy")); // 19
Console.WriteLine(a.ToString("yyyy")); // 2019

Console.WriteLine(a.ToString("HH")); // 13
Console.WriteLine(a.ToString("mm")); // 37
Console.WriteLine(a.ToString("ss")); // 25

Console.WriteLine(a.ToString("dd-MM-yyyy")); // 17-10-2019
Console.WriteLine(a.ToString("ddMMyyyy")); // 17102019
Console.WriteLine(a.ToString("yyyyMMdd")); // 20191017
```

Men ligesom ved formatering af tal bør du måske angive en kultur for at sikre, at formatering er ligegyldig, uanset hvilken maskine der afvikles på. Det kan gøres i de enkelte metoder (som ved formatering af tal på side 72) eller ved at sætte kultur en gang for alle:

```
System.Globalization.CultureInfo c = new
System.Globalization.CultureInfo("en-US");
System.Threading.Thread.CurrentThread.CurrentCulture = c;

DateTime a = new DateTime(2019, 10, 17, 13, 37, 25);
Console.WriteLine(a.ToShortDateString()); // 10/17/2019
Console.WriteLine(a.ToLongDateString());
```

```
// Thursday, October 17, 2019
Console.WriteLine(a.ToShortTimeString()); // 1:37 PM
Console.WriteLine(a.ToLongTimeString()); // 1:37:25 PM

Console.WriteLine(a.ToString()); // 10/17/2019 1:37:25 PM
Console.WriteLine(a.ToString("dd")); // 17
Console.WriteLine(a.ToString("ddd")); // Thu
Console.WriteLine(a.ToString("dddd")); // Thursday
Console.WriteLine(a.ToString("MM")); // 10
Console.WriteLine(a.ToString("MMM")); // okt
Console.WriteLine(a.ToString("yy")); // 19
Console.WriteLine(a.ToString("yyyy")); // 2019

Console.WriteLine(a.ToString("HH")); // 13
Console.WriteLine(a.ToString("mm")); // 37
Console.WriteLine(a.ToString("ss")); // 25

Console.WriteLine(a.ToString("dd-MM-yyyy")); // 17-10-2019
Console.WriteLine(a.ToString("ddMMyyyy")); // 17102019
Console.WriteLine(a.ToString("yyyyMMdd")); // 20191017
```

Typekonvertering af dato og tid

Såvel `DateTime` som `TimeSpan` kan typekonverteres fra strenge af mange forskellige formater, og der er mange måder at gøre det på. Du kan vælge at benytte metoder fra `Convert`-klassen, men det nemmeste er nok den statiske `Parse`-metode fra `DateTime`-strukturen selv:

```
DateTime a = DateTime.Parse("2019-10-15");
DateTime b = DateTime.Parse("2019-10-15T20:15:30");
DateTime c = DateTime.Parse("15. Oktober 2019");
TimeSpan d = TimeSpan.Parse("15:25");
```

Du skal dog være opmærksom på, hvilket format der konverteres fra. Førnævnte kode forventer en dansk kultur, men det anbefales enten at angive kulturen:

```
DateTime a = DateTime.Parse("15. Oktober 2019 20:15", new
    System.Globalization.CultureInfo("da-DK"));
```


eller eventuelt at benytte ParseExact-metoden

```
DateTime a = DateTime.ParseExact("15102019", "ddMMyyyy", new  
    System.Globalization.CultureInfo("da-DK"));
```

Formatet på strengen (ddMMyyyy) angives med specielle tegn, som er generelle for både konvertering og formatering.

Hvis du vil vide mere

Når du bliver lidt mere øvet, kan du måske lede efter yderligere information relateret til dette kapitel. Se kapitlet om Avancerede typer på side 295 og søg på nettet efter eksempelvis:

- Binære og hexadecimale konstanter
- Forskellen på double- og decimal-typerne
- Andre datatyper

Tekster

Du har tit behov for at opbevare tegn og samlinger af tegn (tekster eller, som det hedder i de fleste programmeringssprog, *streng*), og derfor stiller C# naturligvis også typer relateret til tegn og strenge til rådighed.

Tegn

Hvis du gerne vil gemme et enkelt tegn, kan du benytte datatypen `System.Char` eller blot `char`:

Typenavn	Genvej	Størrelse
<code>System.Char</code>	<code>char</code>	16-bit

Tabel 15 `System.Char`

Den kan indeholde et enkelt tegn, som en konstant kan den tildeles værdi med enkeltplinger ('):

```
// uinitialiseret
System.char a; // eller blot: char a;
```

```
// initialiseret med A
char b = 'A';
```

```
// initialiseret med *
char c = '*';
```

```
// værdien fra c kopieres til d
char d = c;
```

Et tegn gemmes i virkeligheden som et tal defineret i en standard kaldet Unicode (læs mere hos [WikiPedia](https://en.wikipedia.org/wiki/List_of_Unicode_characters)¹⁵). Således er et stort A i virkeligheden nummer 65, et stort B er nummer 66, et Ø er nummer 216, en stjerne (*) er 42 og så videre.

¹⁵ https://en.wikipedia.org/wiki/List_of_Unicode_characters

Hvis du gerne vil finde et tegns nummer, eller gerne vil finde et tegn ud fra et nummer, kan du foretage en simpel typekonvertering:

```
char a = '*';  
Console.WriteLine(Convert.ToInt32(a));      // 42  
int svar = 42;  
Console.WriteLine(Convert.ToChar(svar));    // *
```

Der er flere måder at foretage typekonverteringen på, men metoder fra System.Convert er nemmest at forstå.

Måske har du læst The Hitchhiker's Guide to the Galaxy af Douglas Adams. I bogen henvises til super computeren Deep Thought, som har brugt 7,5 million år på at beregne The answer to the Ultimate Question of Life, The Universe, and Everything, og fundet frem til, at svaret er 42. Fans over hele verden har fremført alle mulige teorier om, hvorfor svaret lige præcis skulle være 42, og en af dem er, at 42 svarer til en * i Unicode, og at en stjerne repræsenterer *alt* i mange kommandoer og instruktioner relateret til IT. I virkeligheden er svaret på spørgsmålet "Hvorfor lige 42" noget helt andet – spørg Google hvis du har tid og lyst.

Streng

En streng i C# er en samling af tegn og er repræsenteret med datatypen System.String – eller blot string:

Typenavn	Genvej	Forklaring
System.String	string	Unicode tekst

Tabel 16 System.String

En string-variabel kan tildeles værdier, hvor tekster er omkranset af dobbeltplinger ("):

```
// uinitialiseret
```

```
System.String a;    // eller blot string a;
a = "abcde";        // tildelt en værdi

// initialiseret med det samme
string b = "a";
string c = "Kort sætning";
string d = "Noget længere sætning - i øvrigt med danske tegn";
```

En string-variabel kan indeholde en meget stor mængde tegn, og du vil næppe ramme loftet.

Du kan eventuelt også benytte new-operatoren til at oprette en streng, men det er typisk kun, når du gerne vil starte med en streng med et antal ens tegn:

```
string a = new string('*', 5);
Console.WriteLine(a);           // "*****"
```

Operatorer relateret til strenge

Når du benytter strenge, kan du benytte operatorer, du kender fra andre datatyper:

Operator	Forklaring
=	<i>Tildeling</i>
==	<i>Sammenligning</i>
+	<i>Lægger strenge sammen</i>
+=	<i>Samme som + med tildeler resultat til samme variabel</i>

Tabel 17 Operatorer relateret til System.String

Her et par eksempler:

```
// Tildeling
string a;
a = "a";
Console.WriteLine(a);           // "a"

// Sammenligning
bool test;
test = a == "a";
Console.WriteLine(test);        // true
```

```
Console.WriteLine(a == "b");    // false

// Sammenlægning
a = a + " b";
Console.WriteLine(a);           // "a b"
a += " c";
Console.WriteLine(a);           // "a b c"
```

Metoder relateret til strenge

Når du har skabt en instans af `System.String` kan du benytte en masse forskellige metoder – her nogle af de mest benyttede:

```
string a;
a = " Dette er en længere sætning ";
Console.WriteLine(a.Length);      // 31
Console.WriteLine(a.Contains("sætning")); // true
Console.WriteLine(a.EndsWith("ning ")); // true
Console.WriteLine(a.StartsWith("Test")); // false
Console.WriteLine(a.ToUpper());
// " DETTE ER EN LÆNGERE SÆTNING "

Console.WriteLine(a.ToLower());
// " dette er en længere sætning "

Console.WriteLine(a.Trim());
// "Dette er en længere sætning"

Console.WriteLine(a.Substring(2, 2));
// "De" (fra pos 2 og 2 frem)
```

Der findes en del flere du bør prøve af i Visual Studio.

Selve `System.String`-klassen har også nogle interessante statiske metoder:

```
string a = "a", b = "b";
Console.WriteLine(System.String.IsNullOrEmpty(a));
// false

Console.WriteLine(System.String.Join(" ", "*", a, b, "*"));
```

```
// "* a b *"
```

```
Console.WriteLine(System.String.Compare(a, b));  
// -1
```

Her benyttes eksempelvis IsNullOrEmpty-metoden til at finde ud af, om a er en tom streng, Join-metoden til at sammenlægge strenge med en given separator og Compare-metoden til at sammenligne strenge (metoden returnerer -1, 0 eller 1).

Specialtegn

Nogle gange har du behov for at benytte specialtegn som tabulering, linjeskift, anførselstegn med videre, og disse kan benyttes, hvis du benytter en omvendt slash (\ – en backslash):

Specialtegn	Forklaring
<code>\b</code>	<i>Backspace</i>
<code>\f</code>	<i>Formfeed</i>
<code>\n</code>	<i>New line</i>
<code>\r</code>	<i>Carriage return</i>
<code>\t</code>	<i>Tabulering</i>
<code>\'</code>	<i>Enkelt anførselstegn</i>
<code>\"</code>	<i>Dobbelt anførselstegn</i>
<code>\\</code>	<i>Backslash</i>
<code>\x hhhh</code>	<i>Specielt Unicode tegn (hhhh = nummer)</i>

Tabel 18 Specialtegn

Det klassiske eksempel på manglende brug af specialtegn er dette:

```
string sti = "c:\temp\test.txt";
```

```
Console.WriteLine(sti);  
// "c: emp est.txt"
```

Bemærk, at \t bliver konverteret til en tabulering, og dermed bliver stien til filen helt forkert. Strengen bør i stedet skrives som:

```
string sti = "c:\\temp\\test.txt";
```

```
Console.WriteLine(sti);
```

```
// "c:\temp\test.txt"
```

Du kan også benytte en @ foran strenge, som fortæller kompilatoren, at den skal ignorere alle specialtegn:

```
string sti = @"c:\temp\test.txt";

Console.WriteLine(sti);
// "c:\temp\test.txt"
```

Her er et par eksempler på brug af andre specialtegn:

```
string a;

a = "Linje1\r\nLinje2\r\nLinje3\r\n";
Console.WriteLine(a);
// skriver tre linjer

a = "a\tb";
Console.WriteLine(a);
// a {tab} b

a = "abc\ndef";
Console.WriteLine(a);
// abc"def"
```

Interpolerede strenge

I C# har du mulighed for at skabe såkaldte string templates – også kaldt interpolerede strenge. Det kan gøre det meget nemmere at oprette en streng, der består af værdier fra andre variabler, som måske endda ønskes formateret.

En interpoleret streng kan skabes ved at sætte \$ foran strenge og tilføje variabler i turogklammer:

```
string navn = "Mathias";
int alder = 13;
string a = $"Jeg hedder {navn} og er {alder} år gammel.";
Console.WriteLine(a); // Jeg hedder Mathias og er 13 år gammel.
```

```
// tuborgklammer definerer et udtryk så
// der kan benyttes beregninger, metoder mv
a = $"Jeg hedder {navn.ToUpper()} og er født i {DateTime.Now.Year -
alder}.";
Console.WriteLine(a); // Jeg hedder MATHIAS og er født i 2006.
```

Ved at benytte et kolon i tuborgklammerne kan du sågar benytte de formaterings tegn, du blev introduceret for tidligere:

```
double pris = 2300.2523;
DateTime dato = new DateTime(2020, 1, 1);
string a = $"Prisen på varen er {pris:N2}, og har udløb i måned {dato:M-
yy}";
Console.WriteLine(a);
// Prisen på varen er 2.300,25, og har udløb i måned 1-20
```

Der er flere muligheder ved brug af interpolerede strenge, men det kan du eventuelt læse om i dokumentationen¹⁶.

Strenge er en reference-type

Indtil videre har du lært om såkaldte strukturer (på engelsk struct). Alle de simple variabler, vi har kigget på, har været af denne konkrete type. Hvis du er i tvivl om en type er struct eller en af de andre typedefinitioner, man kan benytte i C# (class, interface, enum eller delegate), kan du altid holde musen hen over typen i Visual Studio eller Visual Studio Code:

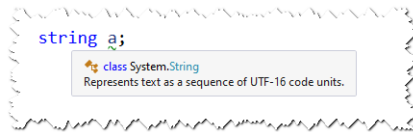


Figur 38 En int (System.Int32) er en struct

Microsoft har valgt at definere de fleste simple variabeltyper som structs for at opnå så høj performance som muligt, og som du skal se senere, kan du også vælge at benytte strukturer til dine egne typer.

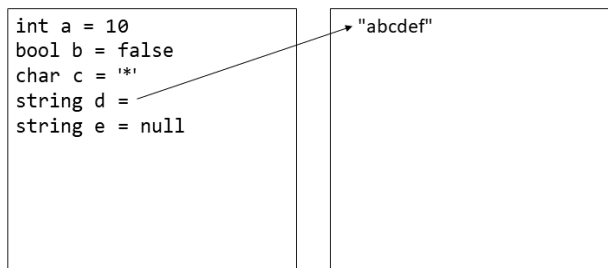
¹⁶ <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/tokens/interpolated>

Men en string (System.String) er ikke en struktur – det er en klasse:



Figur 39 En string er en klasse

Det betyder blandt andet, at variablen ikke indeholder værdien, men derimod en reference til et sted i hukommelsen, hvor værdien findes. Der er mange årsager til, at man gerne vil benytte klasser i stedet for strukturer, og du vil lære meget mere om dette senere i bogen. Lige nu skal du blot bide mærke i, at en variabel af typen string er en klasse, og dermed indeholder en reference og ikke en værdi. Derfor kalder man også variabler af klasser for referencevariabler:



Figur 40 string er en reference-variabel

Fordi en reference-variabel kan indeholde en reference (i virkeligheden et nummer svarende til en adresse i hukommelsen), kan den også indeholde en værdi, der indikerer, at den ikke refererer til noget. I C# hedder denne værdi *null*, og alle reference-variabler kan have denne værdi:

```
string a = null;
```

At du kan arbejde med referencer i stedet for værdier, kan være super-smart og meget effektivt, men det betyder også, at du kan have variabler, som ikke refererer til noget, og det kan være et problem:

```
string a = "mikkel";
Console.WriteLine(a.ToUpper()); // MIKKEL

string b = null;
Console.WriteLine(b.ToUpper());
// Fejl (exception) - b peger ikke på noget
```

Som C# udvikler vil du tit høre om (og bøvle med) såkaldte *null reference exceptions*, som opstår, når du tror, du arbejder med en konkret værdi, men i virkeligheden har fat i null-værdi. Derfor bør du være sikker på, at du har fat i noget konkret:

```
string a = "mikkel";
Console.WriteLine(a.ToUpper()); // MIKKEL

string b = null;
// hvis b er forskellig fra null så...
if (b != null) {
    Console.WriteLine(b.ToUpper());
    // Fejl (exception) - b peger ikke på noget
}
```

C# indeholder et par operatorer, som kan hjælpe med at kontrollere, om en variabel har værdien null:

Operator	Forklaring
?.	Tester om variabel er null og fortsætter kun ved værdi
??	Tester om variabel er null - ellers returneres værdi

Tabel 19 Operatorer relateret til null

De kan bruges til at forkorte koden lidt:

```
string a = null;

if (a != null) {
    Console.WriteLine(a.ToUpper());
}
```

```
// eller
Console.WriteLine(a?.ToUpper());

// eller - hvis a == null så sæt b til "" (en konkret værdi)
string b = a ?? "";
Console.WriteLine(b.ToUpper());
```

Du behøver ikke selv benytte disse operatorer til at starte med, men det er vigtigt, at du ved, at en string er en klasse, og variabler dermed er reference-variabler, der kan have værdien null.

På side 298 kan du læse om, hvordan kompileren kan hjælpe dig med at fange fejl relateret til null.

Du vil senere få meget mere at vide om reference-variabler, og hvilken konsekvens det har for din kode.

Strengene er immutable

Slutteligt er strenge immutable ligesom de fleste simple variabler.

En *immutable* datatype er en speciel type, hvor data ikke kan tilrettes, efter de er tildelt værdier ved initialisering.

Derfor vil alle metoder returnere en ny instans og ikke tilrette den eksisterende:

```
string a = "mathias";
Console.WriteLine(a);           // mathias

a.ToUpper();                   // Returværdi ignoreres
Console.WriteLine(a);           // mathias

a = a.ToUpper();               // Returværdi gemmes
Console.WriteLine(a);           // MATHIAS
```

Det har Microsoft valgt af både performance- og sikkerhedsmæssige hensyn, men det kan give udfordringer.

Prøv følgende kode:

```
string a = "";
for (int i = 0; i < 500000; i++)
{
    a = a + "*";
}
```

Det er et simpelt stykke kode, der opretter en streng med 500.000 stjerner, og burde afvikles inden for få millisekunder. Men hvis du prøver, vil du konstatere, at det tager over et minut (afhængig af din maskine). Det skyldes, at en sammenlægning af strenge skaber en ny kopi, som bliver sammenlagt til en ny kopi, som bliver sammenlagt til en ny værdi, som ...

Faktum er, at koden:

```
a = a + "*";
```

resulterer i, at data flyttes rundt i hukommelsen hele tiden – og at det tager en frygtelig masse tid.

Husk at du aldrig må manipulere strenge i løkker, som kan risikere at tælle til et ukendt antal. Du kan meget hurtigt løbe ind i et performance-problem.

Hvis du tæller til 10 eller 100, er det ligegyldigt, men der skal ikke meget til, før det tager lang tid at afvikle, og det er nemt at forestille sig situationer, hvor du i udvikling måske tæller til 100, men i produktion til 100.000 – eksempelvis fordi du henter nogle data fra en database og skaber en csv-fil.

Strengene og gentagne ændringer kan være et stort performancemæssigt problem, og `System.String` (string) er ikke skabt til det. Det findes der heldigvis andre typer som er – herunder `System.Text.StringBuilder` som netop er skabt til effektiv manipulering af strenge. Læs mere om klassen `StringBuilder`¹⁷ i dokumentationen.

Hvis du vil vide mere

Når du bliver lidt mere øvet, kan du måske lede efter yderligere information relateret til dette kapitel. Det kunne eksempelvis være mere viden om `StringBuilder`-klassen og mere teori om mutable og immutable data.

¹⁷ <https://docs.microsoft.com/en-us/dotnet/api/system.text.stringbuilder>

Konstanter

En konstant er en variabel, som tildeles en værdi ved erklæring, og som efterfølgende ikke kan rettes igen.

Simple konstanter

Erklæring af en enkelt konstant sker ligesom erklæring af en almindelig variabel med tilføjelse af const-kodeordet, samt en tildeling:

```
const int antalMåneder = 12;  
const int juleDag = 24;  
const int juleMåned = 12;
```

```
const double moms = 0.25;
```

```
const string farve = "Rød";
```

Herefter kan konstanterne benyttes, som var det almindelige variabler, men kan blot ikke tildeles en ny værdi:

```
const int juleDag = 24;  
const int juleMåned = 12;  
DateTime jul2019 = new DateTime(2019, juleMåned, juleDag);
```

```
const double moms = 0.25;  
double prisFørMoms = 200;  
double prisEfterMoms = 200 * (1 + moms);
```

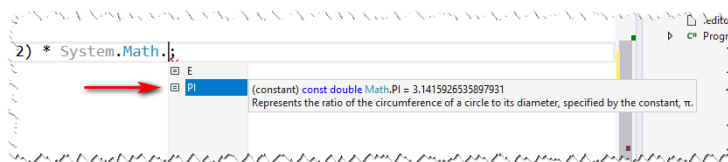
```
moms = 0.15;    // FEJL - kan ikke kompilere
```

Indbyggede simple konstanter

Du vil uden tvivl falde over konstanter, som Microsoft stiller til rådighed. Det klassiske eksempel er konstanten for Pi, som kan findes under System.Math:

```
double radius = 7;  
// radius opløftet i anden x PI  
double areal = System.Math.Pow(radius, 2) * System.Math.PI;
```

Når du ser en konstant i Visual Studio, har de et specielt ikon:



Figur 41 Konstant i Visual Studio

Relaterede konstanter

En anden type konstant er en decideret typedefinition, som gør det muligt at gruppere relaterede konstanter. Det gør koden langt mere læsbar, samt nem at både skrive og vedligeholde.

Forestil dig, at du skal skrive kode, der repræsenterer en person med navn, alder og køn:

```
string navn = "Mikkel";  
int alder = 16;  
int køn = 0;           // 0 = mand, 1 = kvinde
```

Bemærk, at køn er en int og dermed i princippet kan have værdier fra -2.1 milliard til +2.1 milliard. Vi vedtager dog, at værdien 0 repræsenterer en mand, og værdien 1 en kvinde. Det er helt ok og fuldt lovligt, men vi skal sørge for, at vi tydeligt dokumenterer hvad 0 og 1 betyder for dem, der skriver og læser koden. Det kan naturligvis gøres i koden som kommentarer, men det bliver hurtigt svært at styre.

Derfor har du mulighed for at definere relaterede konstanter i en typedefinition en gang for alle, og så lade kompilatoren og Visual Studio (Code) hjælpe med at benytte konstanterne.

Denne type hedder en enum (forkortet fra enumeration) og er en af de typer, du kan definere i C# (enum, class, struct, interface, poster og delegate). Når du definerer en enum, skaber du en skabelon for,

hvordan instanser skal se ud, og en enum består udelukkende af konstanter med navne og værdier:

```
enum [navn] [:type]
{
    [Konstantnavn] = værdi,
    [Konstantnavn] = værdi,
    [Konstantnavn] = værdi,
    ...
}
```

Hvis du skal definere en enum, der indeholder konstanter til kønnet på en person, kan den eksempelvis skrives således:

```
public enum PersonKøn : int
{
    Mand = 0,
    Kvinde = 1
}
```

Du behøver dog ikke angive typen (int er default):

```
public enum PersonKøn
{
    Mand = 0,
    Kvinde = 1
}
```

Og hvis du ikke angiver et nummer, vil kompileren nummerere konstanterne fra 0:

```
public enum PersonKøn
{
    Mand,
    Kvinde
}
```

En enum bør placeres på namespace-niveau og typisk i en fil for sig selv. Men du må gerne placere den i eksempelvis program.cs, så længe den er placeret på namespace-niveau:


```
using System;

namespace MinTest
{
    class Program
    {
        static void Main(string[] args)
        {

        }

    }

    public enum PersonKøn
    {
        Mand = 0,
        Kvinde = 1
    }
}
```

Brug af enum

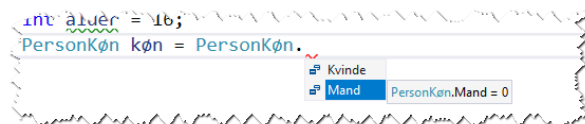
Når du har defineret en enumeration, kan den benyttes på samme måde som en int eller en anden type – så koden fra tidligere:

```
string navn = "Mikkel";
int alder = 16;
int køn = 0;           // 0 = mand, 1 = kvinde
```

kan nu erstattes med det langt mere logiske og læsbare:

```
string navn = "Mikkel";
int alder = 16;
PersonKøn køn = PersonKøn.Mand;
```

Det er ligeledes langt nemmere at skrive i Visual Studio (Code), fordi den godt ved, at der er tale om en relateret konstant:



Figur 42 Brug af en enum i Visual Studio

Typekonvertering

Hvis du ønsker at tildele en enum-variabel en værdi fra et nummer (eksempelvis et nummer fra en database eller fil), skal du foretage en typekonvertering:

```
// hvis nummeret er et heltal
int nr1 = 1;
PersonKøn køn1 = (PersonKøn)nr1;
```

```
// hvis nummeret er en streng er det lidt mere komplekst
string nr2 = "1";
PersonKøn køn2 = (PersonKøn)Enum.Parse(typeof(PersonKøn), nr2);
```

Hvis du ønsker nummeret fra en enum-variabel, kan du konvertere til en int:

```
PersonKøn køn = PersonKøn.Kvinde;
int k = (int)køn; // eller brug Convert.ToInt32
```

Så du kan godt komme fra tal til enum og retur igen.

Indbyggede relaterede konstanter

Microsoft har en del forskellige enums i frameworket, som du kan benytte, som du vil. Her er et par eksempler, men der er mange flere:

```
// ugedag
System.DayOfWeek dag = DayOfWeek.Saturday;
// farve til konsol
System.ConsoleColor farve = ConsoleColor.Cyan;
// drev type
System.IO.DriveType t = System.IO.DriveType.Network;
```

Hvis du vil vide mere

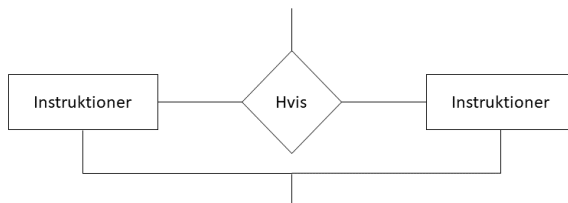
Når du bliver lidt mere øvet, kan du måske lede efter yderligere information relateret til dette kapitel. Søg eksempelvis efter mere information om Enum-typen og hvordan du ved hjælp af attributter (attributes på engelsk) kan tilføje yderligere metadata.

Programflow

Et af de vigtigste redskaber i programmering er muligheden for at styre, hvilke instruktioner der skal afvikles på et givet tidspunkt, og C# indeholder naturligvis kommandoer relateret til dette. Set i lyset af vigtigheden af disse kommandoer, er det bemærkelsesværdigt, at der i virkeligheden ikke er så mange af dem.

If-betingelsen

Det vigtigste kodeord i C# er uden tvivl if-kodeordet, som du kan benytte til styre afvikling af instruktioner baseret på et boolsk udtryk (et udtryk der returnerer sandt eller falsk):



Figur 43 If-strukturen

I C# ser en if-struktur således ud:

```
if([udtryk])
{
    // instruktioner
}
else
{
    // instruktioner
}
```

Hvis der kun er én sand-blok, kan else-blokken udelades:

```
if([udtryk])
{
```

```
// instruktioner  
}
```

De instruktioner, der skal afvikles, placeres i tuborgklammerne, som samtidigt definerer et virkefelt således, at variabler defineret i virkefeltet kun lever her.

Hvis der kun er én instruktion, der skal afvikles, hvis udtryk er sandt, kan tuborgklammer eventuelt undlades:

```
if([udtryk])  
    // instruktion
```

Men de fleste vælger at benytte tuborgklammer, selv om der kun er én instruktion.

Selve udtrykket kan gemmes i en boolsk variabel eller angives direkte i if-strukturen, og det kan være både simpelt og ret komplekst med mange operatorer:

```
int i = 1, j = 2;  
  
// Simpelt  
bool b1 = i == 1;  
if (b1)  
{  
}  
  
// Komplekst  
bool b2 = i == 1 || j > 1 && DateTime.Now.DayOfWeek == DayOfWeek.Monday;  
if (b2)  
{  
}  
  
// eller direkte i if  
if (i==1)  
{  
}  
  
if (i == 1 || j > 1 && DateTime.Now.DayOfWeek == DayOfWeek.Monday)
```

```
{  
}
```

Hvis der er mange forgreninger, kan du eventuelt tilføje nogle else if-kommandoer. Når først en blok er afviklet, springes resten over:

```
int i = DateTime.Now.Second;
```

```
if (i <= 10)  
{  
  
}  
else if (i <= 20)  
{  
  
}  
else if (i <= 30)  
{  
  
}  
else  
{  
  
}
```

Den bedste måde at lære forgreninger at kende er at sætte breakpoints, og så steppe igennem koden. Prøv det hvis du er i tvivl.

Husk at bruge snippets så meget du kan.
Du kan skabe en if-struktur med if+tab+tab.

I nyere C# versioner har du mulighed for at skrive udtryk på en lidt mere logisk måde ved hjælp af mønstergenkendelse (pattern matching), samt brugen af kodeordene is, and og or. Således kan følgende:

```
int antal = 30;  
if (antal > 24 && antal < 40)
```

```
Console.WriteLine("Antal er større end 24 og mindre end 40");
```

også skrives som

```
int antal = 30;  
if(antal is > 24 and < 40)  
    Console.WriteLine("Antal er større end 24 og mindre end 40");
```

Bemærk brugen af *is* og *and*. Det er mere logisk for os mennesker at tænke: "hvis antal er større end 24 og mindre end 40" end "hvis antal er større end 24 og antal er mindre end 40". De nyere features relateret til mønstergenkendelse gør det muligt.

Led efter "C# pattern matching" hvis du vil vide mere om avanceret mønstergenkendelse i C#.

Switch-betingelsen

Som alternativ til if-kommandoen kan du eventuelt benytte switch-kommandoen, som nogle gange kan være lidt nemmere at overskue end en masse else if-kommandoer. Den fungerer ved at kontrollere en konkret værdi, som skal være af typen char, string, int, long eller enum, og så afvikle en given blok-kode. Syntaksen er som følger:

```
switch([variabel])  
{  
    case [værdi]:  
        // instruktion(er)  
        break;  
    case [værdi]:  
        // instruktion(er)  
        break;  
  
    // andre case ...  
  
    default:  
        // instruktion(er)  
        break;
```

```
}
```

Bemærk, at for hver case-blok benyttes break-kodeordet for at bryde helt ud af strukturen, hvorefter afvikling fortsætter med første instruktion efter strukturen. Du kan eventuelt benytte en default-blok som afvikles, hvis ingen af de andre case-blokke afvikles – men den er ikke nødvendig.

Her er et simpelt eksempel, hvor der kontrolleres en int:

```
// Tilfældigt tal mellem 1 og 3 (inklusiv begge)
int i = new System.Random().Next(1, 4);

switch (i)
{
    case 1:
        // instruktioner
        break;
    case 2:
        // instruktioner
        break;
    case 3:
        // instruktioner
        break;
}
```

Du kan eventuelt flytte break-kodeordet for at kontrollere på flere værdier:

```
int i = DateTime.Now.Month;

switch (i)
{
    case 1:
    case 2:
    case 3:
        Console.WriteLine("vinter");
        break;

    case 4:
```



```
case 5:
case 6:
    Console.WriteLine("forår");
    break;

case 7:
case 8:
case 9:
    Console.WriteLine("sommer");
    break;

case 10:
case 11:
case 12:
    Console.WriteLine("efterår");
    break;

default:
    Console.WriteLine("Forkert værdi!!");
    break;
}
```

Strukturen kan også benyttes mere avanceret ved at benytte et when-kodeord, men det kræver, at der angives en ny variabel af en konkret type, der så kontrolleres:

```
int i = DateTime.Now.Month;

switch (i)
{

    case int x when x >= 1 && x <= 3:
        Console.WriteLine("vinter");
        break;

    case int x when x >= 4 && x <= 6:
        Console.WriteLine("forår");
        break;
```

```
case int x when x >= 7 && x <= 9:
    Console.WriteLine("sommer");
    break;

case int x when x >= 10 && x <= 12:
    Console.WriteLine("efterår");
    break;

default:
    Console.WriteLine("Forkert værdi!!");
    break;
}
```

Det kan se lidt komplekst ud, men kan være ret brugbart i mere avanceret (og objektorienteret) kode, fordi der nu ikke blot kontrolleres en værdi, men også en type.

Brug *switch+tab+tab* for at indsætte kode ved hjælp af en snippet.

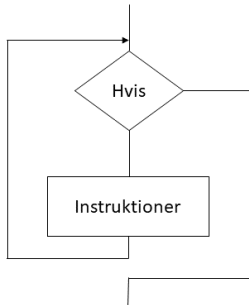
I nyere C# versioner kan du også benytte mønstergenkendelse (pattern matching) i forbindelse med switch. Som begynder skal du ikke gå så meget op i denne form for switch, for det kræver viden om både lambda-funktioner og delegates, men for en god ordens skyld kan du se følgende eksempel:

```
int antal = 30;
Console.WriteLine(antal switch
{
    < 24 => "Antal er mindre end 24",
    > 24 and < 40 => "Antal er større end 24 og mindre end 40",
    _ => "Antal er større end 39"
});
// Antal er større end 24 og mindre end 40
```

Koden giver noget mere mening, når du har læst om lambda (=> operatoren) på side 280.

For-løkken

En løkke kan bruges til at afvikle en blok-kode et givet antal gange (eller så længe et udtryk returnerer sandt), og C# har flere typer. Den mest benyttede er dog nok for-løkken:



Figur 44 En løkke-struktur

Den består i sin grundlæggende form af tre instruktioner – erklæring af en tællevariabel, kontrol af værdi samt opskrivning eller nedskrivning af tællevariablen.

I sin mest simple syntaks ser det ud som følger:

```
for ([erklæring og tildeling af tællevariabel];  
    [kontroludtryk];  
    [opskrivning/nedskrivning af tællevariabel])  
{  
    // kode  
}
```

Her er eksempelvis en løkke, der tæller fra 0 til 9:

```
for (int i = 0; i < 10; i++)  
{  
    Console.WriteLine(i);  
}
```

Hvis der blot er en enkelt instruktion, kan tuborgklammer eventuelt udelades:

```
for (int i = 0; i < 10; i++)  
    Console.WriteLine(i);
```

Da det er helt op til dig, hvordan udtryk og påvirkning af tællevariabel skal ske, kan du skabe en masse forskellige løkker. Her er et par eksempler:

```
// Hver anden (0, 2, 4, ...)  
for (int i = 0; i < 10; i = i + 2)  
{  
    Console.WriteLine(i);  
}  
  
// Bagfra (9, 8, 7, 6, ...)  
for (int i = 10; i > 0; i--)  
{  
    Console.WriteLine(i);  
}
```

Da tællevariablen erklæres i løkkestrukturen, kan den kun tilgås i selve blokken:

```
for (int i = 0; i < 3; i++)  
{  
    // her kan i tilgås  
}  
// her kan i ikke tilgås
```

I Visual Studio kan en for-løkke nemt skabes med en snippet. Brug "for + tab + tab".

Do- og while-løkken

Nogle gange ved du ikke, hvor meget du skal arbejde i en løkkestruktur. Måske vil du eksempelvis afvikle kode så længe, der er linjer i en fil,

eller data i en tabel. Her kan do-strukturen, eller den mere rene while-struktur, være brugbar.

En do-løkkestruktur ser således ud:

```
do
{
    // kode
} while([udtryk])
```

I denne struktur afvikles kode i en løkke, så længe et udtryk er sandt, og du er sikker på at komme ind i strukturen mindst én gang.

En while-struktur er vendt om:

```
while([udtryk])
{
    // kode
}
```

Også her afvikles kode i en løkke, så længe udtrykket returnerer sandt, men da udtrykket er placeret i toppen, behøver du ikke komme ind i strukturen overhovedet.

Her er et par eksempler:

```
// Tæller til tre (0, 1, 2) med en tællevariabel
int i = 0;
do
{
    // kode
    i++;
} while (i < 3);
```

```
DateTime tid = DateTime.Now.AddSeconds(5);
// Løber i 5 sekunder
while (DateTime.Now < tid)
{
    // kode
}
```

I Visual Studio kan du indsætte kode med en snippet til både en do (do + tab + tab) og en while (while + tab + tab).

Brug af continue

I løkkestrukturer kan du eventuelt benytte continue-kodeordet for at fortælle compileren, at du ønsker at starte næste iteration omgående, og dermed springe resten af koden i en løkke over. Det kan eksempelvis benyttes således:

```
// Tæller 1, 2, 4, 5 fordi 3 springes over
for (int i = 1; i < 6; i++)
{
    if (i == 3)
        continue;
    Console.WriteLine(i);
}
```

Kodeordet kan også benyttes i do- eller while-løkker.

Brug af break

Nogle gange ønsker du at hoppe helt ud af et loop, og her kan du bruge break-koden (ligesom i en switch-struktur). Hvis compileren ser et break, vil næste instruktion til afvikling blive første instruktion efter løkken:

```
// Tæller 1, 2 fordi break hopper helt ud af løkken
for (int i = 1; i < 6; i++)
{
    if (i == 3)
        break;
    Console.WriteLine(i);
}
```

Løkker i løkker

Du må naturligvis gerne placere løkker inde i andre løkker. Det kan være ret brugbart, hvis du eksempelvis ønsker at gennemløbe en

datastruktur med flere dimensioner. Jo flere løkker, du placerer i hinanden, jo sværere bliver det at overskue, hvad der foregår, men her er især debuggeren guld værd. Hvis du sætter et breakpoint og stepper dig igennem koden instruktion for instruktion, er du ikke i tvivl.

En anden måde er at udskrive indeks – eksempelvis:

```
for (int i = 0; i < 3; i++)
{
    for (int x = 0; x < 3; x++)
    {
        Console.WriteLine($"i: {i} x: {x}");
    }
}

/*
i: 0 x: 0
i: 0 x: 1
i: 0 x: 2
i: 1 x: 0
i: 1 x: 1
i: 1 x: 2
i: 2 x: 0
i: 2 x: 1
i: 2 x: 2
*/
```

Hvis du følger logikken, vil du se, at den ydre løkke tæller en gang, mens den indre tæller tre gange.

Du kan kombinere så mange løkker i hinanden, du måtte få brug for, men det kan være svært at holde tungen lige i munden nogle gange.

ForEach-løkken

Den sidste løkke-struktur er ForEach-løkken. Den er lidt speciel i forhold til de andre, fordi den arbejder med arrays og samlinger helt uden tællevariabler. Senere i bogen vil du blive introduceret til arrays og samlinger, og der kommer vi retur til ForEach-løkken.

Goto

Slutteligt kan du styre programpointeren med en goto-instruktion, men det skal du prøve at lade være med. Det skaber en kode, som er svær at gennemskue og vedligeholde, og de fleste C# udviklere går langt uden om goto, hvis de kan. Det er en instruktion, der typisk hører hjemme i de helt gamle iterative sprog og i lavniveau-sprog som assembler (hvor der slet ikke findes løkker, men en masse *jump* instruktioner).

Men der *kan* være situationer, hvor goto kan være praktisk – eksempelvis hvis du skal hoppe ud af løkker, som er placeret inde i hinanden (break-kodeordet hopper kun ud af den løkke, den er placeret i).

En goto-instruktion hænger sammen med en navngivet etiket:

```
goto [navn]

// etiket
[navn]:
// kode
```

Her er et eksempel på brug af goto:

```
Console.WriteLine("Start");
for (int i = 1; i < 11; i++)
{
    for (int x = 1; x < 11; x++)
    {
        if (x == 5 && i == 5)
            goto slut;
    }
}
slut:
Console.WriteLine("Slut");
```

Her vil goto sørge for at hoppe ud af begge løkke-strukturer og fortsætte afvikling.

Kodeordet goto er en fuldt lovlig instruktion, men prøv at begrænse brugen af det eksempelvis at hoppe ud af dybe løkker – ellers kan det skabe noget frygtelig kode at vedligeholde på et senere tidspunkt.

Hvis du vil vide mere

Når du bliver lidt mere øvet, kan du måske lede efter yderligere information relateret til dette kapitel. Der findes eksempelvis en mere avanceret måde at benytte switch-flowstrukturen. Søg efter *Pattern Matching*.

Metoder

Metoder, som i nogle sprog også benævnes funktioner, kan gøre koden mere genbrugelig og overskuelig, og giver mulighed for at skabe blokke af instruktioner som kan kaldes fra forskellige steder i koden. En metode kan enten være en metode, som blot afvikler instruktioner uden en returnværdi, eller en samling af instruktioner, som returnerer en værdi ligesom en matematisk funktion. Yderligere kan en metode kaldes med en samling værdier (kaldet argumenter), som kan benyttes i metoden.

Hvorfor benytte metoder

Som eksempel på brug af metoder for at gøre koden både nemmere at genbruge og læse, kan du forestille dig en opgave, hvor du skal tælle fra 1 til 5, udskrive værdien, og gentage dette tre gange.

Det kunne jo kodes således:

```
for (int i = 1; i < 6; i++)  
    Console.WriteLine(i);
```

```
for (int i = 1; i < 6; i++)  
    Console.WriteLine(i);
```

```
for (int i = 1; i < 6; i++)  
    Console.WriteLine(i);
```

men gentaget kode er roden til alt ondt – hvad nu hvis opgaven i fremtiden ændres, så du skal tælle til 10. Så skal du ændre koden tre steder.

Måske kunne det i stedet kodes som:

```
for (int x = 0; x < 3; x++)  
    for (int i = 1; i < 6; i++)  
        Console.WriteLine(i);
```

Det ser noget bedre ud og giver samme resultat. Men det kan ikke genbruges uden at kopiere koden – og så er vi tilbage i gentaget kode.

Du kunne også rode dig ud i noget goto-fnidder:

```
int programLinjeTæller = 0;
start:
if (programLinjeTæller < 3)
    goto tæl;
else
    goto slut;
tæl:
for (int i = 1; i < 6; i++)
    Console.WriteLine(i);
programLinjeTæller++;
goto start;
slut:
```

Nu kan koden i princippet genbruges, men sikke en gang rod.

Opgaven løses langt bedre med en metode:

```
for (int x = 0; x < 3; x++)
{
    Tæl();
}
void Tæl()
{
    for (int i = 1; i < 6; i++)
        Console.WriteLine(i);
}
```

Nu tælles der til tre og hver gang kaldes en metode, som udskriver værdierne fra 1-6.

Koden kan gøres endnu mere genbrugelig ved at tilføje et argument til Tæl-metoden, så det antal, der skal tælles til, er angivet i kaldet:

```
Console.WriteLine();
for (int x = 0; x < 3; x++)
{
```

```
Tæl(5);  
}  
  
void Tæl(int tælTil)  
{  
    for (int i = 1; i < tælTil+1; i++)  
        Console.WriteLine(i);  
}
```

Sluttelig kunne hele opgaven løses i en enkelt metode:

```
Tæl(5, 3);  
  
void Tæl(int tælTil, int gentag)  
{  
    for (int x = 1; x < gentag + 1; x++)  
        for (int i = 1; i < tælTil + 1; i++)  
            Console.WriteLine(i);  
}
```

Nu kan Tæl-metoden genbruges overalt, hvor der skal tælles et tal, og eventuelt gentages et antal gange.

For at få et par begreber på plads, så betyder

- et *kald* til en metode, at metoden afvikles
- *void*-kodeordet foran metoden, at metoden ikke *returnerer* noget (void betyder *tom* – se senere)
- *navnet* på metoden er Tæl
- de to *argumenter* tælTil og gentag er begge heltal, og skal angives i kaldet til metoden.

Programpointeren og metoder

Noget af det, en begynder i programmering (ikke bare C#) kan have svært ved, er at se, hvilke instruktioner som afvikles ved kald til metode og hvornår. Det kaldes også at følge *programpointeren*, der er den pegepind som kompiler og runtime benytter til at afvikle instruktioner i den rette rækkefølge.

Uden metoder er det nemt nok – du er sikkert ikke tvivl om rækkefølgen, når følgende instruktioner afvikles:

```
int i = DateTime.Now.Second; // find aktuelt sekund
if (i % 2 == 0) // Hvis det er et lige sekund
{
    Console.WriteLine("Lige");
}
else
{
    Console.WriteLine("Ulige");
}
```

Koden kan naturligvis kompliceres med en masse løkker og yderligere betingelsesstrukturer, men grundlæggende er det nemt at følge programpointeren. Og du kan jo altid benytte debuggeren og steppe dig igennem koden, hvis du er i tvivl.

Anderledes er det ved metodekald, fordi kompiler og runtime automatisk sørger for at vende retur til instruktionen lige efter et metodekald. Se følgende kode (prøv den gerne selv):

```
Console.WriteLine("Start");
MinMetode();
Console.WriteLine("Slut");

void MinMetode() {
    Console.WriteLine("I metode");
}
```

Når du afvikler koden, vil du se:

```
Start
I metode
Slut
```

Det er et tydeligt bevis på, at runtime husker, hvilken instruktion der blev afviklet inden metodekald og automatisk vender retur til den efterfølgende instruktion.

Lidt mere kompleks er følgende kode:

```
Console.WriteLine("Start");
MinMetode1();
Console.WriteLine("Slut");

void MinMetode1()
{
    Console.WriteLine("I metode 1");
    MinMetode2();
}
void MinMetode2()
{
    Console.WriteLine("I metode 2");
    MinMetode3();
}
void MinMetode3()
{
    Console.WriteLine("I metode 3");
}
```

Bemærk, at MinMetode1 kalder MinMetode2 som kalder MinMetode3, og når runtime har afviklet denne vendes retur til MinMetode2, som dermed er færdigafviklet, og så vendes retur til MinMetode1, og slutteligt vendes der retur til instruktionen lige efter kaldet til MinMetode1. Resultatet bliver dermed:

```
Start
I metode 1
I metode 2
I metode 3
Slut
```

Så ved metoder, der kalder andre metoder, husker runtime altså, hvor programpointeren kom fra, og skal nok selv vende retur.

Hvis du er helt ny i programmering, er det en god ide at kopiere ovennævnte kode, og så bruge debuggeren til at steppe dig igennem. Så kan du tydeligt se rækkefølgen. Du skal blot huske at bruge F11 (Step Into), når du stepper igennem koden.

Man kunne måske fristes til at kalde MinMetode2 i slutningen af MinMetode3 for at *komme retur*, men det vil give en runtime fejl, fordi du ender i et uendeligt loop. Det kaldes også et uendeligt rekursivt loop, og ender med at applikationen opbruger al tildelt hukommelse.

Definition af metoder

En metode har følgende definition:

```
[synlighed] [[static]] [returtype] Navn ([argumenter])
{
    // kode
    // return ...
}
```

Synlighed fortæller, hvor metoden kan tilgås og kan i sin grundlæggende form have værdierne public, private samt protected. Indtil videre kan du blot erklære dine metoder offentlige (public).

En metode kan være enten en statisk (static) eller en instans-metode, og det fortæller runtime, om metoden er placeret på et objekt, som først skal oprettes, inden metoden kan benyttes, eller om metoden kan tilgås direkte på selve typen. Hvis du vil lege med forskellige typer af metoder i en standard konsolapplikation, er det nemmeste at definere statiske metoder i Program-klassen, og så kalde dem fra Main-metoden (som i sig selv er statisk). Vi kommer senere mere ind på statiske og instans medlemmer.

Returtypen definerer, hvilken type metoden returnerer. Hvis den ikke returnerer noget, angives typen som *void* (tom), men ellers kan enhver type angives. Du kan således skabe en metode, der returnerer en int, double, bool, DateTime, File og meget andet.

Navnet på metoden er selvvalgt og skal blot overholde få regler omkring blandt andet brug af specialtegn. De fleste vil navngive metoder med et stort bogstav til at starte med, og herefter benytte

camel-casing, som du typisk også ser variabler benytte. Således kan eksempler på metodenavne være Beregn, BeregnAntalDage, KørProgram og så videre. Men der er frit valg ved navngivning.

Du kan angive så mange argumenter til en metode, du ønsker. De skal angives med komma imellem, og alle argumenter skal være omkranset af en parentes. Hvis der ikke er nogle argumenter, skal der angives en tom parentes. De enkelte argumenter skal angives med en type og et navn ligesom ved erklæring af variabler og er typisk navngivet på samme måde. Der er dog helt frit valg.

Selve metodekroppen er omkranset af tuborgklammer, og hvis metoden returnerer andet end void, så skal der angives en (eller flere) *return*-instruktioner. Return-kodeordet fortæller runtime, hvad metoden konkret returnerer, og hvis kodeordet mangler, kan koden ikke kompileres. Det gælder ikke ved en void-metode, som blot returnerer efter den sidste instruktion. Man må dog gerne benytte et eller flere return-kodeord i en void-metode for at afslutte før tid.

Placering af metoder

Som tidligere nævnt er alt i C# baseret på de fem typer – klasser, strukturer, interfaces, enumerations og delegates. En metode skal placeres enten i en struktur eller i en klasse, og en af disse typer repræsenterer typisk et element i en applikation. Microsoft bruger eksempelvis strukturer og klasser til at repræsentere variabeltyper som `int` (struktur) og `string` (klasse), og disse typer har forskellige metoder. På samme måde kan du oprette klasser og strukturer, og her kan du placere metoder.

I en standard konsol-applikation har du allerede klassen `Program`, og den kan du godt udvide med metoder:

```
namespace Demo
{
    class Program
    {
        static void Main(string[] args)
```



```
{  
    MinMetode();  
}  
  
static void MinMetode() { }  
}  
}
```

Hvis du udvider Program-klassen, er det nemmest at skabe statiske metoder (med static kodeordet) – primært fordi Main i sig selv er statisk og dermed nemt kan tilgå andre statiske metoder.

Metoder der ikke returnerer en værdi

Her er en samling af metoder, du kan bruge som skabelon, indtil du har syntaksen på plads. Først void-metoder – altså metoder som ikke returnerer noget:

```
using System;  
  
namespace Demo  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            // eksempler på kald til alle metode  
            MinMetode1();  
            MinMetode2(4);  
            MinMetode3(4, "z");  
            MinMetode4(4, "z");  
            MinMetode5(4, "z");  
        }  
  
        // void-metode uden argumenter  
        static void MinMetode1()  
        {  
            Console.WriteLine("I MinMetode1");  
        }  
    }  
}
```

```
// void-metode med et enkelt argument
static void MinMetode2(int a)
{
    Console.WriteLine($"I MinMetode2 med a={a}");
}

// void-metode med to argumenter
static void MinMetode3(int a, string b)
{
    Console.WriteLine($"I MinMetode3 med a={a} og b={b}");
}

// void-metode med argumenter som kun afvikler kode hvis a >= 5
static void MinMetode4(int a, string b)
{
    if (a >= 5)
    {
        Console.WriteLine($"I MinMetode4 med a={a} og b={b}");
    }
}

// void-metode med argumenter som kun afvikler kode hvis a >= 5
// ved hjælp af return
static void MinMetode5(int a, string b)
{
    if (a < 5)
    {
        return;
    }
    Console.WriteLine($"I MinMetode5 med a={a} og b={b}");
}
}
```

Bemærk de forskellige måder en void-metode kan oprettes:

- uden argumenter
- med argumenter
- brug af return for at afbryde før tid.

Disse eksempler kan du benytte som skabelon til alle typer af void-metoder.

Du kan altså opfatte en void-metode som en mulighed for at genbruge en samling instruktioner – med eller uden argumenter. Der kan ikke returneres noget fra en void-metode.

Metoder der returnerer en værdi

Hvis du ønsker at en metode skal returnere noget, skal du blot erstatte void-kodeordet med den givne type. Her er eksempler på metoder, der returnerer simple datatyper:

```
using System;

namespace Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            // kald til metode hvor returværdi ikke benyttes
            Metode1();
            // kald til metode hvor returværdi gemmes
            bool r1 = Metode1();
            // kald til metode hvor resultat blot udskrives (true)
            Console.WriteLine(Metode1());

            // kald til metode med argument
            bool r2 = Metode2(1);
            // kald til metode med argumenter
            int r3 = Metode3(5, 5);
            // 10
            Console.WriteLine(r3);

            // kald til metode med et enkelt argument
            string r4 = Metode4("mathias");
```

```
// Mathias
Console.WriteLine(r4);
// kald til metode hvor resultat blot udskrives (Mikkel)
Console.WriteLine(Metode4("mikkel"));

// Kald til metode med flere returns (")
Console.WriteLine(Metode5(null));
// Kald til metode med flere returns (")
Console.WriteLine(Metode5(""));
// Kald til metode med flere returns (Michell)
Console.WriteLine(Metode5("mIcheLl"));

}

// Metode der returnerer en bool uden argumenter
static bool Metode1()
{
    return true;
}

// Metode der returnerer en bool med argument
static bool Metode2(int a)
{
    bool res = a > 10;
    return res;    // eller blot return a > 10;
}

// Metode der returnerer en int to argumenter
static int Metode3(int a, int b)
{
    return a + b;
}

// Metode der returnerer en string med et argument
static string Metode4(string navn)
{
    string lille = navn.ToLower();
    string førsteBogstav = lille.Substring(0, 1).ToUpper();
    string resten = lille.Substring(1);
    return førsteBogstav + resten;
}
```

```
}

// Metode der returnerer en string med et argument
// med flere return
static string Metode5(string navn)
{
    if (navn == null || navn == "")
        return "";
    string lille = navn.ToLower();
    string førsteBogstav = lille.Substring(0, 1).ToUpper();
    string resten = lille.Substring(1);
    return førsteBogstav + resten;
}

}

}
```

Bemærk de forskellige typer, som metoderne returnerer, samt brugen af return i den sidste metode. Du må gerne have flere returns, men en metode med en returtype skal returnere en værdi – ellers vil du få en *not all code paths returns a value* fejl.

En metode kan returnere alle typer – herunder også dine egne typer (klasser eller strukturer). Det ser vi på senere.

Argumenter og variabler

Et klassisk begynder spørgsmål tager udgangspunkt i kode som følger:

```
using System;

namespace Demo
{
    class Program
    {
        static void Main(string[] args)
        {

            Metode1(1, 1);
            int a = 1, b = 1;
```

```
Metode1(a, b);  
// hvad er værdien af a og b her  
int x = 1, y = 1;  
Metode1(x, y);  
// hvad er værdien af x og y her  
  
}  
  
public static void Metode1(int a, int b)  
{  
    Console.WriteLine($"I Metode1 med a={a} og b={b}");  
    a++;  
    Console.WriteLine($"a har nu værdien {a}");  
}  
}  
}
```

Metoden Metode1 tager to argumenter (a og b) og kaldes tre gange – først med konstanter, så med variablerne kaldet a og b, og slutteligt med variablerne x og y.

Spørgsmålet er, om variablerne a, b, x og y ændrer værdi i Main efter at metoden er kaldt? Svaret er nej – de ændrer ikke værdi.

I Main vil både a, b, x og y have værdien 1 både før og efter kaldet til Metode1. Det gælder altså også variablerne a og b, som jo har samme navn som argumenter a og b i metoden.

En standard C# metode har sit eget lille virkefelt, og det der sker i metoden, har ikke påvirkning i andre virkefelter i programmet. Når en metode kaldes, vil værdier (fra konstanter eller variabler) kopieres ind i metoden som argumenter, og du kan se på argumenterne, som var det variabler i metoden. Hvis du ændrer værdierne af argumenterne i metoden, har det ingen konsekvens for variabler i det virkefelt, som kalder metoden.

I den viste kode er argumenterne simple værdibaserede typer, så her kopieres værdierne ind som argumenter. I mere komplekse situationer kan argumenterne være referencebaserede, og så kan en ændring have konsekvens i det kaldende virkefelt. Det ændrer dog ikke på, at værdien kopieres ind i argumenterne. Når der er tale om referencebaserede variabler, er værdien blot en reference. Mere om referencebaserede variabler senere.

Argumenter som ikke behøves angivet

I nogle situationer giver det mening at argumenter har default værdier, og dermed ikke behøves angivet ved kald. Det klares nemt i C# ved at bruge lig med operatoren efter et argument. Eneste regel er, at argumenter med default værdier skal stå til sidst – ellers kan kompileren ikke regne ud, hvilke værdier der skal tildeles hvilke argumenter.

Her er et eksempel med et enkelt argument:

```
using System;

namespace Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            // Kald til metoden med momsPct
            double res1 = BeregnPrisMedMoms(100, 0.25);
            // Kald til metoden uden momsPct
            // (dermed benyttes 0.25 som værdi)
            double res2 = BeregnPrisMedMoms(100);

            Console.WriteLine(res1);    // 125
            Console.WriteLine(res2);    // 125
        }
    }
}
```

```
static double BeregnPrisMedMoms(double pris,  
    double momsPct = 0.25) {  
    return pris * (1 + momsPct);  
}  
}
```

Bemærk, at argumentet momsPct sættes til 0.25 i definitionen af metoden, og det betyder at metoden kan kaldes på to måder:

```
BeregnPrisMedMoms(100, 0.25);  
BeregnPrisMedMoms(100);
```

Du må benytte så mange defaultværdier til argumenter, du har lyst til, men argumenterne skal placeres til sidst.

Navngivne argumenter

Nogle metoder kan have mange argumenter, og det kan gøre det svært at skrive og læse koden, når metoden kaldes. Derfor kan du vælge at angive navne på argumenter ved kald til metoden. Se følgende eksempel:

```
using System;  
  
namespace Demo  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            // standard kald af metoden  
            // værdier skal placeres i korrekt rækkefølge  
            Metode1("Mikkel", 16, true, "DK");  
  
            // kald af metoden med navngivne argumenter  
            Metode1(navn: "Mikkel", alder: 16,  
                erSmart: true, land: "DK");  
  
            // kald af metoden med navngivne argumenter
```



```
// og nu er rækkefølgen ikke længere vigtig
Metode1(land: "DK", erSmart: true,
        alder: 16, navn: "Mikkel");

// kald af metoden med både unavngivne
// og navngivne argumenter
Metode1("Mikkel", 16, land: "DK", erSmart: true);
}

static void Metode1(string navn, int alder,
                    bool erSmart, string land)
{
}
}
```

Bemærk, at metoden kan kaldes så det tydeligt fremgår, hvilke værdier som skal kopieres til hvilke argumenter, og at unavngivne og navngivne argumenter kan kombineres (med her har rækkefølgen betydning igen).

Der er lidt blandede meninger om, hvorvidt man bør benytte navngivne argumenter – flere mener, det er noget pjat og kan bedre lide at angive argumenter i den rigtige rækkefølge. Det er helt op til dig – hvis du mener, det er nemmere at skrive og læse navngivne argumenter, gør du det. Det "koster" ikke noget ved afvikling af applikationen.

Metoder med samme navn (overload)

For at gøre kald til relaterede metoder så effektive som muligt, kan du vælge at have flere metoder med samme navn. Det kræver dog, at kompilatoren kan adskille metoderne fra hinanden ved hjælp af metodens *signatur* (returværdi og argumenter).

Se følgende eksempel, hvor der findes tre metoder til at beregne moms ud fra forskellige variabeltyper:

```
using System;

namespace Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            double a = 100;
            decimal b = 100;
            int c = 100;

            Console.WriteLine(BeregnMomsDouble(a, 0.25));        // 125
            Console.WriteLine(BeregnMomsDecimal(b, 0.25));       // 125
            Console.WriteLine(BeregnMomsDouble(c, 0.25));        // 125
        }

        static decimal BeregnMomsDouble(double pris, double momspct) {
            return Convert.ToDecimal(pris * (1 + momspct));
        }

        static decimal BeregnMomsDecimal(decimal pris, double momspct)
        {
            return pris * (1 + Convert.ToDecimal(momspct));
        }

        static decimal BeregnMomsInt32(int pris, double momspct)
        {
            return pris * (1 + Convert.ToDecimal(momspct));
        }
    }
}
```

Der er tale om tre metoder, som beregner en samlet pris, men gør det på basis af forskellige variabeltyper – derfor er de navngivet:

BeregnMomsDouble
BeregnMomsDecimal
BeregnMomsInt32

Fordi metoderne i virkeligheden har forskellig signatur, kan de godt hedde det samme, og det gør koden mere logisk og læsbar:

```
using System;

namespace Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            double a = 100;
            decimal b = 100;
            int c = 100;

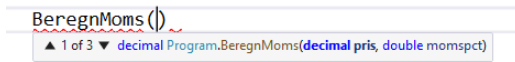
            Console.WriteLine(BeregnMoms(a, 0.25));           // 125
            Console.WriteLine(BeregnMoms(b, 0.25));           // 125
            Console.WriteLine(BeregnMoms((double)c, 0.25));   // 125
        }

        static decimal BeregnMoms(double pris, double momspct) {
            return Convert.ToDecimal(pris * (1 + momspct));
        }

        static decimal BeregnMoms(decimal pris, double momspct)
        {
            return pris * (1 + Convert.ToDecimal(momspct));
        }

        static decimal BeregnMoms(int pris, double momspct)
        {
            return pris * (1 + Convert.ToDecimal(momspct));
        }
    }
}
```

Bemærk, at alle tre metoder nu hedder BeregnMoms, og når du angiver kaldet i eksempelvis Visual Studio, er det meget tydeligt:



Figur 45 Kald af metoder med samme navn.

Metoder med samme navn hedder også et *overload*, og der findes et hav af overloadede metoder i frameworket. Se eksempelvis hvor mange gange `Console.WriteLine`-metoden er overloaded.

Hvis du vil vide mere

Vi ser på brug af lambda-udtryk senere i bogen, men når du bliver lidt mere øvet, kan du måske lede efter yderligere information relateret til dette kapitel. Søg eksempelvis efter information om:

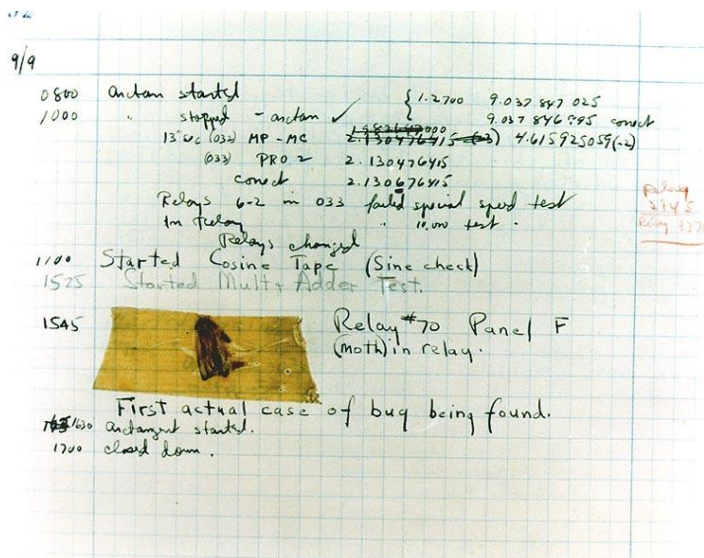
- Brug af params-kodeordet
- Metoder i metoder
- Brug af ref og out ved metoder
- Expression-bodied members
- Extension methods

Fejlhåndtering

Der vil altid kunne opstå fejl i dine applikationer!

Det kan enten skyldes, at du skriver kode, der fejler, eller udefrakommende påvirkninger som en bruger, der indtaster noget forkert, en database, der pludselig ikke svarer, eller en fil der ikke længere eksisterer. Derfor er det vigtigt at kunne fange og håndtere fejl.

Når du hører om softwarefejl, vil du tit høre dem omtalt som *bugs* (engelsk for insekter), hvilket kan lyde lidt mystisk. Men begrebet skulle komme fra en reel hardwarefejl i den gamle Mark II relæcomputer fra 1946, hvor et møl var kommet i klemme i et relæ og dermed skabte en fejl. Insektet blev fundet, klistret ind i en log og er i dag udstillet på National Museum of American History:



Figur 46 Billede af den første bug¹⁸

¹⁸ Billede er fra Naval Surface Warfare Center – se mere på https://en.wikipedia.org/wiki/Software_bug

I C# hedder en fejl en *exception*, og den kan enten skabes af runtime, 3. parts kode eller din egen kode. I daglig tale vil du høre, at der *bliver smidt en fejl*, og det skal du forstå, som at applikationen afbrydes, og årsagen kan aflæses i et Exception-objekt. Din opgave er så at fange Exception-objektet og eventuelt håndtere fejlen på en eller anden måde. Måske vil du blot fortælle brugeren på en pæn måde, at der er sket en fejl, måske vil du gentage nogle instruktioner, eller måske vil du logge fejlen i en database eller fil. Der er typisk tale om en kombination, men det er helt op til dig.

I C# teori vil du høre om begreberne *handled* og *unhandled exceptions*. Det skal du forstå som håndterede (du skriver kode til at håndtere fejlen) og uhåndterede (runtime vil blot afbryde programmet) fejl.

Eksempler på fejl

For at du kan få en idé om forskellige typer af fejl, og eventuelt prøve dem af selv, er her et par eksempler:

```
using System;

namespace Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            // Forskellige typer af fejl

            // Fejl1();
            // Fejl2();
            // Fejl3();
            // Fejl4();

        }
    }
}
```

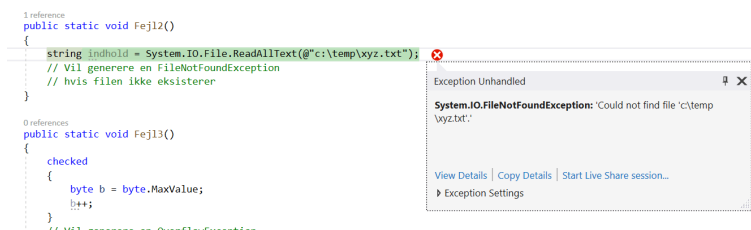
```
public static void Fejl1()
{
    string tekst = null;
    // Vil generere en NullReferenceException
    // fordi tekst variabelen ikke har en reference
    // til noget
    Console.WriteLine(tekst.ToUpper());
}

public static void Fejl2()
{
    string indhold =
        System.IO.File.ReadAllText(@"c:\temp\xyz.txt");
    // Vil generere en FileNotFoundException
    // hvis filen ikke eksisterer
}

public static void Fejl3()
{
    checked
    {
        byte b = byte.MaxValue;
        b++;
    }
    // Vil generere en OverflowException
    // fordi 255 + 1 ikke "kan være" i en
    // byte (og checked sørger for at smide
    // en fejl i så fald)
}

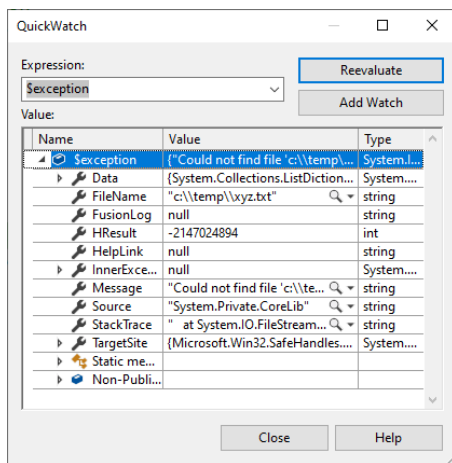
public static void Fejl4()
{
    int a = 10;
    int b = 0;
    int res = a / b;
    // DivideByZeroException
}
}
}
```

Du bør prøve at skabe en ny konsol-applikation og prøve et par af fejlene af i praksis – både med og uden debugger. Hvis fejlen opstår med debuggeren tilknyttet, vil du opleve, at Visual Studio stopper, hvor fejlen er opstået, med information om et konkret Exception-objekt, som beskriver fejlen:



Figur 47 Exception i Visual Studio

Ved at klikke på *View details* kan du få oplysninger om den konkrete fejl:



Figur 48 Informationer om en Exception

Exception-objektet indeholder en masse brugbare informationer som eksempelvis:

Navn	Forklaring
<i>Message</i>	<i>Den konkrete fejlttekst</i>
<i>StackTrace</i>	<i>I hvilken metode fejlen er opstået, og hvordan programpointeren er kommet derhen</i>
<i>Source</i>	<i>I hvilket assembly fejlen er opstået</i>
<i>InnerException</i>	<i>Hvis fejlen er sket grundet en anden fejl, kan den oprindelige fejl aflæses her</i>

Tabel 20 Medlemmer på Exception-klassen.

Brug af try/catch

Fejlhåndtering i C# sker ved at omkrænse kode med en try/catch struktur og dermed *prøve* noget kode af. Skulle der ske en fejl i koden, kan den fanges i catch-delen af strukturen:

```
try
{
    // kode der skal testes for fejl
}
catch(Exception ex)
{
    // skulle der opstå en fejl "fanges" den her
    // og oplysninger om fejlen kan findes i
    // exception-objektet.
}
```

Koden, der testes for fejl, kan både være enkeltstående instruktioner eller metodekald ud af try-strukturen. Skulle der ske en fejl i en metode (eller én metode der kalder en anden) vil fejlen også blive fanget.

Her er et eksempel på enkeltstående instruktioner og en fejlhåndtering, som blot udskriver en fejlttekst:

```
try
{
    Console.WriteLine("Indtast tal");
    string talTekst = Console.ReadLine();
    double tal = Convert.ToDouble(talTekst);
    Console.WriteLine($"Du har indtastet {tal:N2}");
}
```

```
catch (Exception ex)
{
    Console.WriteLine($"Ups - følgende fejl er opstået: {ex.Message}");
}
```

Koden henter en tekst fra brugeren, og teksten konverteres til et tal. Det kan jo gå fint, hvis brugeren indtaster et tal som forventet, men det kan også gå galt, hvis brugeren indtaster noget vrøvl. I så fald smider ToDouble-metoden en fejl, som fanges i catch og udskriver en tekst:

```
Indtast tal
abc
Ups - følgende fejl er opstået: Input string was not in a correct
format.
```

Du bestemmer naturligvis selv, hvor detaljeret du vil informere brugeren.

Her er et andet eksempel, som benytter et kald til en metode:

```
using System;

namespace Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Console.WriteLine("Indtast tal");
                string talTekst = Console.ReadLine();
                int resultat = LægEnTil(talTekst);
                Console.WriteLine($"Resultatet er {resultat}");
            }
            catch (Exception ex)
            {
                Console.WriteLine(
                    $"Ups - følgende fejl er opstået: {ex.Message}");
            }
        }
    }
}
```

```
static int LægEnTil(string talTekst) {  
    int tal = Convert.ToInt32(talTekst);  
    tal++;  
    return tal;  
}  
}
```

Hvis brugeren indtaster noget, som ikke kan konverteres til et tal, fanges fejlen, og det sker altså på trods af, at fejlen opstår i en separat metode. Såfremt metoden er kaldt i en try-struktur, vil eventuelle fejl altid blive fanget.

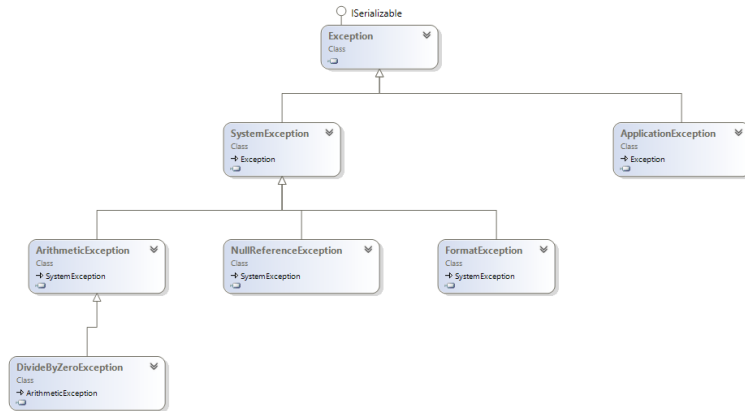
Flere catch-blokke

I grundlæggende C# kan du nøjes med en simpel try/catch struktur med en enkelt catch-blok:

```
try  
{  
}  
catch(Exception ex)  
{  
}
```

Men det er muligt at udvide antallet af catch-blokke med andre exception-typer således, at du kan skrive kode til at håndtere forskellige typer af fejl. Det gør det muligt at skrive kode til eksempelvis fejl relateret til filsystemet (måske vent et par millisekunder og prøv igen), eller fejl relateret til typekonvertering (måske bede brugeren om at indtaste data igen).

Den overordnede klasse er Exception-klassen, men i frameworket findes der en masse forskellige klasser, der arver fra Exception – herunder eksempelvis ArithmeticException, FormatException, IOException og mange flere:



Figur 49 Lille udsnit af forskellige Exception-klasser

Hvis du tilføjer flere catch-blokke, vil runtime sørge for at afvikle den korrekte blok kode afhængig af typen af fejl:

```
try
{
    // kald en metode hvor flere fejl kan ske
    MuligFejl();
}
catch (System.IO.IOException ex)
{
    // Gør noget
}
catch (NullReferenceException ex)
{
    // Gør noget
}
catch (ArithmeticException ex)
{
    // Gør noget
}
catch (Exception ex)
{
    // Gør noget
}
```

Du skal blot sørge for, at catch-blokken er placeret i en rækkefølge, der sikrer, at de mest specifikke typer står først. Hvis en fejl ikke passer på en specifik catch-blok, vil den altid passe ind i den sidste Exception-blok. Således vil du altid fange en fejl, hvis den sidste blok håndterer et Exception-objekt.

Du behøver ikke have mange catch-blokke i din kode. Du kan bare nøjes med en catch-blok, som er relateret til Exception-klassen. Så bliver alle fejl fanget.

Brug af finally

I nogen situationer kan det være praktisk at være sikker på, at instruktioner altid afvikles – både når der sker en fejl, og når der ikke sker en fejl. Derfor kan try/catch strukturen udvides med en finally-blok, som kan indeholde instruktioner til afvikling:

```
try
{
    // instruktioner
}
catch(Exception ex)
{
    // instruktioner til at afvikling ved fejl
}
finally
{
    // instruktioner til afvikling hvad enten der sker en fejl eller ikke
}
```

Grunden til, at du skal benytte en finally blok, i stedet for blot at placere instruktioner efter try/catch strukturen er, at du er sikret at instruktioner i finally altid afvikles – også selv om instruktioner tvinger afvikling ud af try-blokken (eksempelvis return-kodeordet) eller der skulle ske en ny fejl i catch-blokken.

Man benytter tit en finally-blok, hvis der arbejdes med ressourcer, hvor det er vigtigt, at der altid ryddes op efter brug. Det kunne eksempelvis være brug af filer eller databaser, hvor det ikke er smart, at en forbindelse kan risikere at blive efterladt åben.

Her er et kort eksempel, hvor der læses et antal bytes fra en fil. Selve IO koden er ikke væsentlig, men bemærk finally-blokken:

```
int index = 0;
string sti = @"c:\temp\test.txt";
System.IO.StreamReader file = new System.IO.StreamReader(sti);
char[] buffer = new char[10];
try
{
    file.ReadBlock(buffer, index, buffer.Length);
}
catch (Exception ex)
{
    // log fejl
    Console.WriteLine("Der er opstået en fejl!");
}
finally
{
    if (file != null)
        file.Close();
}
```

Ved at placere kald til Close-metoden i finally-blokken er du sikker på, at den altid bliver afviklet.

Skab dine egne fejl

At *smide en Exception* er en god måde at fortælle de udviklere, som benytter din kode (herunder dig selv), at der er sket en fejl, og det kan du gøre ved hjælp af *throw*-kodeordet.

Antag at du skal skabe en metode, der lægger to heltal tal sammen:

```
using System;
```

```
namespace Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            // dette er ok
            Console.WriteLine(LægSammen(10,10));
            // dette er ikke ok
            Console.WriteLine(LægSammen(1500, 10));
        }

        static int LægSammen(int a, int b) {
            return a + b;
        }
    }
}
```

Du skal nu sørge for, at metoden kun må lægge tal sammen, der er større end 0 og mindre end 1.000 – hvordan vil du løse den opgave?

Du kan ikke være sikker på, at den der kalder LægSammen-metoden, kender restriktionerne, så det er i selve metoden, du skal tilføje kode til at sikre, at logikken bliver overholdt. Du kunne måske gøre noget som eksempelvis:

```
static int LægSammen(int a, int b) {
    if ((a < 0 || a > 1000) || (b < 0 || b > 1000))
        return -1;
    else
        return a + b;
}
```

Nu vil metoden returnere -1, hvis a eller b har en forkert værdi, men det er en skidt løsning, fordi udvikleren, der kalder metoden, skal være bevidst om, at -1 er lig med en fejl og skal teste for en konkret fejl ved hvert kald.

Bedre var det, hvis udvikleren, som kalder metoden, blot kan bruge en try/catch struktur og dermed fange eventuelle fejl. Det kræver dog, at

LægSammen-metoden smider en fejl med en tilhørende Exception, og det kan som nævnt ske ved hjælp af throw-kodeordet:

```
using System;

namespace Demo
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                // dette er ok
                Console.WriteLine(LægSammen(10, 10));
                // dette er ikke ok
                Console.WriteLine(LægSammen(1500, 10));
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }

        static int LægSammen(int a, int b)
        {
            if ((a < 0 || a > 1000) || (b < 0 || b > 1000))
                throw new Exception("Argumenter har en forkert værdi");
            else
                return a + b;
        }
    }
}
```

Bemærk brugen af throw-kodeordet som smider en ny Exception med en sigende besked, og bemærk brugen af try/catch som fanger eventuelle fejl – herunder fejlen der opstår, når metoden kaldes med argumenterne 1500 og 10.

Brugen af throw kan sidestilles med brugen af return på den måde, at det også er en måde at fortælle kompileren, at afvikling af metoden ønskes afbrudt.

Når du udvikler metoder, skal du have to kasketter på – én når du udvikler metoden, og én når du kalder metoden. Hvis du tænker på den måde, giver det lidt mere mening, at du smider en fejl i en metode, og tester for fejl ved kald.

I den mere avancerede C# kan du smide en fejl baseret på andre typer end Exception-klassen, men det er ikke så væsentligt på dette niveau.

Brug en snippet (try + to gange tabulering) for at få hjælp til at indsætte en try/catch struktur. Du kan også omkranse kode med en try/catch ved at bruge ctrl+K+S i Visual Studio og vælge try på listen over mulige snippets.

Log

Typisk vil fejlhåndtering involvere en eller anden form for log således, at der findes data til at genskabe fejlen. Du skal ikke selv kode et log-system – for det første er det ret komplekst at gøre rigtigt, og for det andet er det gjort mange gange. Du kan eventuelt benytte (open source) systemer som NLog eller Log4Net, men det ligger uden for denne bogs rammer at komme nærmere ind på det. Hvis du gerne vil have en smule log, kan du eventuelt tage udgangspunkt i følgende metode – men du må ikke bruge den i produktion:

```
using System;

namespace Demo
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
        Log("Test #1");
        Log("Test #2");

    }

    // Simpel log-metode der skriver til en fil.
    // Metoden tilføjer tid og fjerner eventuelle linjeskift i tekst
    static void Log(string tekst) {
        System.IO.File.AppendAllText(
            @"c:\temp\log.txt",
            $"{DateTime.Now:dd-MM-yy HH:mm:ss:ffff} " +
            $"{tekst.Replace("\r\n", "")} \r\n");
    }
}
```

Koden vil skabe en log-fil i c:\temp (som skal eksistere) kaldet log.txt med følgende indhold:

```
13-01-20 16:43:37:5347 Test #1
13-01-20 16:43:37:5620 Test #2
```

Med denne simple log-metode vil en typisk try/catch struktur se således ud:

```
try
{
    Console.WriteLine("Indtast tal");
    string talTekst = Console.ReadLine();
    int resultat = LægEnTil(talTekst);
    Console.WriteLine($"Resultatet er {resultat}");
}
catch (Exception ex)
{
    // ex.ToString() vil returnere al info til en streng
    Log(ex.ToString());
    Console.WriteLine($"Ups - følgende fejl er opstået: {ex.Message}");
}
```

Nu vil fejl blive fanget, der udskrives en besked til brugeren, og fejlen bliver gemt i en log-fil til eventuelt senere gennemsyn.

Brug altid et gennemtestet Log-system til log – søg efter NLog, Log4Net eller Serilog.

Hvis du vil vide mere

Når du bliver lidt mere øvet, bør du prøve at skabe dine egne Exception-klasser ved at arve fra Exception-klassen. Så kan du skabe dit eget hierarki af typer, der repræsenterer fejl og på den måde gøre det nemmere at fejlfinde og logge.

Du bør også kigge nærmere på et af de nævnte logsystemer og måske et af de mange frameworks, der kan hjælpe med fejl relateret til eksempelvis timeout og fejlhåndtering efter flere forsøg. Prøv at se nærmere på Polly¹⁹.

¹⁹ <https://github.com/App-vNext/Polly>

Arrays

Et array er et begreb i de fleste programmeringssprog, og det dækker over muligheden for at gemme data i struktureret form i stedet for i enkelte variabler.

Forestil dig, at du skal beregne en sum og gennemsnit af nedbør over nogle måneder. Det kunne eksempelvis ske således:

```
double nedbørMåned1 = 4;
double nedbørMåned2 = 12;
double nedbørMåned3 = 2;
double nedbørMåned4 = 8;
double sum = nedbørMåned1 + nedbørMåned2 + nedbørMåned3 + nedbørMåned4;
Console.WriteLine($"Sum {sum:N2}");
double gns = sum / 4;
Console.WriteLine($"Gennemsnit {gns:N2}");
```

Det fungerer ganske udmærket og beregner både sum og gennemsnit helt korrekt. Problemet opstår, når der skal flere måneder med i beregningen, for det vil kræve ændring i både sum- og gennemsnitsberegning. Det var nemmere, hvis nedbør kunne gemmes i en variabel, som kan indeholde flere værdier, og at disse værdier kunne tilgås ved hjælp af et indeks. Så kan der benyttes løkkestrukturer til at løbe alle data igennem:

nedbør				
Indeks	0	1	2	3
Værdi	4	12	2	8

Figur 50 Array over nedbør

En sådan struktur hedder et array og giver en masse muligheder for at arbejde med strukturerede data. Et array i C# er nulpaseret forstået således, at det første element skal tilgås med indeks 0.

Koden fra tidligere kan nu skrives således:

```
double[] nedbør = { 4, 12, 2, 8 };
```

```
double sum = 0;
for (int i = 0; i < nedbør.Length; i++)
    sum += nedbør[i];
Console.WriteLine($"Sum {sum:N2}");
double gns = sum / nedbør.Length;
Console.WriteLine($"Gennemsnit {gns:N2}");
```

Du behøver ikke gå op i syntaks, men blot bemærke, at koden nu nemt kan udvides med flere måneders nedbør ved blot at udvide arrayet i første linje.

Oprettelse af et array

Som i de fleste programmeringssprog er et array i syntaks defineret ved hjælp af firkantede parenteser []. Yderligere er C# jo et typestærkt sprog, hvilket betyder, at variabler skal erklæres af en konkret type. Det gælder også for et array.

Således skal en variabel, der kan indeholde referencen til et array, erklæres som følger:

```
int[] heltalsArray;
double[] doubleArray;
string[] stringArray;
bool[] boolArray;
```

Bemærk, at det er brugen af firkantede parenteser, der fortæller kompileren, at der er tale om et array.

Men dette opretter blot en variabel, som kan pege på et array af en konkret type – der er ikke oprettet en instans i hukommelsen. Det skal ske ved hjælp af *new*-operatoren samt angivelse af hvor mange elementer, du ønsker.

Således opretter følgende kode et heltalsarray med fem elementer, som automatisk bliver initialiseret til defaultværdien (0):

```
int[] heltalsArray = new int[5];
```

Koden opretter en variabel kaldet `heltalsArray`, der kan pege på et array med fem elementer (heltal) i hukommelsen:

heltalsArray					
Indeks	0	1	2	3	4
Værdi	0	0	0	0	0

Figur 51 Array af heltal

Og følgende kode opretter en variabel kaldet `boolArray` med tre elementer:

```
bool[] boolArray = new bool[3];
```

Da defaultværdien til en `bool` er `false`, bliver alle elementerne initialiseret til `false`:

boolArray			
Indeks	0	1	2
Værdi	false	false	false

Figur 52 Array af bool

Du kan også vælge at splitte erklæring og oprettelse af array over to instruktioner:

```
bool boolArray;  
boolArray = new bool[3];
```

Du skal læse koden således, at første instruktion erklærer en variabel, der kan indeholde referencen til et `bool`-array, og næste instruktion opretter et array med tre elementer og tildeler referencen af arrayet i hukommelsen til variabelen.

Hvis du kender værdierne, når du opretter arrayet, kan du vælge at benytte lidt syntaks sukker:

```
double[] nedbør = { 4, 12, 2, 8 };
```

Her erklæres en variabel nedbør som peger på et array med fire elementer, der med det samme har fået en værdi.

Lige som en streng er et array baseret på en klasse og er dermed en reference-type. Det betyder, at der gemmes en reference til et array i hukommelsen. I basal C# programmering er det ikke så væsentligt, men det vil hurtigt blive noget, du skal være opmærksom på.

Tilgang til elementer

Når først du har et array tilgængeligt, kan du tildele og aflæse data på flere måder. De fleste vælger at angive et indeks i firkantede parenteser:

```
double[] array = new double[4];  
// Tildeler data  
array[0] = 4;  
array[1] = 12;  
array[2] = 2;  
array[3] = 8;  
  
// aflæser data  
double a = array[0]; // 4
```

Hvis du forsøger at tilgå et element uden for det antal elementer, der er angivet ved oprettelse, vil runtime smide en fejl (husk at et array er nulbaseret, så det første indeks er nul).

Gennemløb af arrays

Fordi data kan tilgås gennem et indeks, og fordi alle instanser af et array har en egenskab kaldet `Length`, der returnerer antallet af elementer i arrayet, kan du benytte en for-løkke til at løbe elementerne igennem:

```
double[] nedbør = { 4, 12, 2, 8 };  
for (int i = 0; i < nedbør.Length; i++)
```

```
{  
    Console.WriteLine($"Indeks={i} Værdi={nedbør[i]}");  
}
```

Det vil resultere i følgende:

```
Indeks=0 Værdi=4  
Indeks=1 Værdi=12  
Indeks=2 Værdi=2  
Indeks=3 Værdi=8
```

Alternativt kan du benytte for ForEach-løkke, som ikke benytter sig af en tællevariabel, men blot løber alle elementer igennem, og tildeler den aktuelle værdi til variablen angivet i løkken:

```
double[] nedbør = { 4, 12, 2, 8 };  
foreach (double item in nedbør)  
{  
    Console.WriteLine(item);  
}
```

Det resulterer i følgende:

```
4  
12  
2  
8
```

Hvis du ikke har brug for en tællevariabel, kan ForEach-løkken være lidt hurtigere at skrive.

Manipulering af arrays

Ethvert array er baseret på klassen System.Array, og denne klasse indeholder en del praktiske instans-metoder (metoder tilgængelig på instansen) og statiske metoder (metoder tilgængelige på typen).

På selve instansen er metoderne Clone og CopyTo interessante. Clone-metoder kopierer et array til et nyt array, og CopyTo kopierer værdier. Se følgende eksempler:


```
int[] a = new int[] { 10, 5, 1, 7, 1, 6 };

// Kopiering af array til et nyt array (typekonvering er nødvendig)
int[] b = a.Clone() as int[];
// b er nu
// [10, 5, 1, 7, 1, 6]

int[] c = new int[8];
c[0] = 20;
c[1] = 30;
a.CopyTo(c, 2);
// c er nu
// [ 20, 30, 10, 5, 1, 7, 1, 6]
```

Af statiske metoder findes især Resize, som ændrer størrelsen på et array, samt Sort og Reverse som sorterer værdier. Se følgende eksempler:

```
// Et array med 6 elementer
int[] a = new int[] { 10, 5, 1, 7, 1, 6 };
```

```
System.Array.Sort(a);
// a er nu
// [1, 1, 5, 6, 7, 10]
```

```
System.Array.Reverse(a);
// a er nu
// [10, 7, 6, 5, 1, 1]
```

```
// Nu har det 10 elementer
System.Array.Resize(ref a, 10);
// a er nu
// [10, 7, 6, 5, 1, 1, 0, 0, 0, 0]
```


Lige nu kan du se bort fra ref-kodeordet i kaldet til Resize, men du skal være bevidst om, at alle metoder tilretter værdier i det eksisterende array.

Flere dimensioner

Et array har som udgangspunkt kun én dimension forstået således, at det kan ses som en enkelt kolonne i et regneark med mange rækker. Så følgende kode:

```
int[] a = { 5, 1, 3, 7, 8 };
```

kunne ansues som et regneark som følger:



Indeks	Værdi
0	5
1	1
2	3
4	7
5	8

Figur 53 Et endimensionelt array i et regneark

Men det står dig frit for at oprette et array med flere dimensioner ved at benytte et eller flere kommaer i erklæringen:

```
int[,] a = new int[3, 3];
a[0, 0] = 1;
a[0, 1] = 2;
a[0, 2] = 3;
a[1, 0] = 4;
a[1, 1] = 5;
a[1, 2] = 6;
a[2, 0] = 7;
a[2, 1] = 8;
a[2, 2] = 9;
```

Det kan jo se noget teknisk ud, men prøv at se det som data i et regneark:

	0	1	2
Indeks	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

Figur 54 Et todimensionelt array i et regneark

Nu er det lidt nemmere at gennemskue, hvorfor `a[1, 2]` indeholder værdien 6.

Her er et lidt mere komplekst eksempel på brug af et todimensionelt array:

```
string[,] skakBræt = new string[8, 8];
skakBræt[0, 0] = "Ts";
skakBræt[0, 1] = "Hs";
skakBræt[0, 2] = "Ls";
skakBræt[0, 3] = "Ds";
skakBræt[0, 4] = "Ks";
skakBræt[0, 5] = "Ls";
skakBræt[0, 6] = "Hs";
skakBræt[0, 7] = "Ts";

for (int i = 0; i < 8; i++)
{
    skakBræt[1, i] = "Bs";
    skakBræt[6, i] = "Bh";
    skakBræt[7, i] = skakBræt[0, i].Replace("s", "h");
}
```

Koden skaber et skakbræt med 64 felter og placerer samtidig brikker (Ts = tårn sort, Hh = Hest hvid og så videre). Skakbrættet kan udskrives med:

```
for (int række = 0; række < 8; række++)
{
    for (int kolonne = 0; kolonne < 8; kolonne++)
    {
```

```
        Console.Write(skakBræt[række, kolonne] + " ");  
    }  
    Console.WriteLine();  
}
```

Hvilket giver følgende resultat:

```
Ts Hs Ls Ds Ks Ls Hs Ts  
Bs Bs Bs Bs Bs Bs Bs Bs
```

```
Bh Bh Bh Bh Bh Bh Bh Bh  
Th Hh Lh Dh Kh Lh Hh Th
```

Hvis du er ny i programmering, så se om du kan følge logikken – både i koden, der skaber skatbrættet, og koden der udskriver det.

Du må gerne skabe arrays med mange dimensioner, men efter 3. dimension begynder det hele at blive noget teknisk at overskue.

Hvis du vil vide mere

Når du bliver lidt mere øvet, kan du måske lede efter information om `Span<T>` og `Memory<T>`, som benyttes i forbindelse med optimering. Det er absolut ikke et emne for begyndere, men måske skulle du læse lidt om, hvad typerne kan bruges til.

Samlinger

Et traditionelt array er en meget statisk datastruktur forstået således, at når det først er oprettet med en type og en størrelse, koster det en del at ændre størrelsen, hvis man ønsker flere data i strukturen.

Heldigvis findes der andre strukturtyper, du kan benytte som alternativ – herunder lister, stakke, køer og nøgle-/værdi-strukturer, og der findes både typesvage (System.Collections) og typestærke versioner (System.Collections.Generic).

Du skal fokusere på de typestærke i System.Collections.Generic, fordi de er langt mere effektive i både udvikling og afvikling.

Generiske samlinger

Inden vi kigger på de konkrete samlinger, er det dog vigtigt, at du er introduceret til et begreb, du vil falde over mange gange i C# – nemlig begrebet *generics*. Det kan vel oversættes til *generel* og er relateret til brugen af konkrete typer.

Det er en avanceret feature i C#, som giver mulighed for at fortælle kompileren, at du på en type (eller en metode) ønsker at arbejde med en konkret variabeltype. I generiske typer er logik og funktionalitet således ens for alle typer (som dog kan filtreres), og ved brug skal den type, du ønsker, angives. Hertil benyttes en speciel syntaks med mindre og større end tegn samt en angivelse af typen. Tit ser du generiske typer angivet med en variabeltype kaldet T (forkortelse for type), og T kan du mentalt erstatte med den type, du ønsker at benytte:

```
// Forskellige eksempler på generiske typer
List<T> = Liste af MANGE FORSKELLIGE typer
Queue<T> = Kø af MANGE FORSKELLIGE typer
Point<T> = Point af MANGE FORSKELLIGE typer
```

Når du i kode ser en konkret datatype omkranset af < > skal du mentalt tilføje af:

```
List<int> = Liste AF heltal  
Queue<bool> = Kø AF boolske værdier  
Stack<string> = Stak AF strenge
```

Det smarte ved generiske typer er, at både Visual Studio og kompileren kan optimere brug og afvikling, fordi du fortæller, hvilken konkret datatype, du ønsker at arbejde med.

Liste

I langt de fleste tilfælde vil du arbejde med `List<T>`, som er en simpel og effektiv dynamisk liste af instanser af en konkret datatype. Følgende opretter eksempelvis en liste af heltal i hukommelsen og tildeler referencen til en variabel:

```
System.Collections.Generic.List<int> a = new  
    System.Collections.Generic.List<int>();
```

Hvis du har:

```
using System.Collections.Generic;
```

i toppen af filen, kan du nøjes med den noget mere spiselige syntaks:

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

En `List` kan også bruges på alle andre typer – eksempelvis:

```
List<bool> a = new List<bool>();  
List<DateTime> b = new List<DateTime>();  
List<string> a = new List<string>();
```

Den kan ligeledes benyttes med dine egne typer (se senere i bogen).

Ligesom arrays og andre typer kan en liste initialiseres med data direkte ved oprettelse:

```
List<int> tal = new List<int> { 4, 12, 2, 8 };  
List<string> strenge = new List<string> { "a", "x", "y", "b" };  
List<DateTime> datoer = new List<DateTime> {  
    new DateTime(2020, 1, 1),  
    new DateTime(2021, 1, 1)}
```

```
};
```

Når først instansen er oprettet, består den af en masse medlemmer til at indsætte, fjerne, sortere, filtrere, gennemløbe og meget mere. Her er eksempler på de mest benyttede metoder. Der tages udgangspunkt i en liste af strenge, men husk at du kan skabe en liste af mange forskellige typer:

```
// Liste af strenge
List<string> lst = new List<string>();

// Tilføj "a", "b" og "c"
lst.Add("a");
lst.Add("b");
string c = "c";
lst.Add(c);

// Hvor mange strenge er der i listen
int antal = lst.Count; // 3

// Indsæt "*" i position 1 (husk - alt er nulbaseret)
lst.Insert(1, "*"); // lst = a, *, b, c

// Indsæt data fra et array
lst.InsertRange(2, new string[] { "!", "@" });
// lst = a, *, !, @, b, c

// Fjern *
lst.Remove("*"); // lst = a, !, @, b, c

// Fjern streng i position 2
lst.RemoveAt(2); // lst = a, !, b, c

// finder der en ! i listen
bool test = lst.Contains("!"); // test = true

// Løb alle igennem og skriv ud
Console.WriteLine("Vis alle:");
foreach (var item in lst) // item = string
{
```

```
        Console.WriteLine(item);
    }

    // Sorter listen
    lst.Sort();

    // Sorter listen (omvendt rækkefølge)
    lst.Reverse();
```

Her er et konkret eksempel på brug af en liste til at beregne en sum og et gennemsnit af nedbør (samme eksempel som blev gennemgået i kapitlet om arrays):

```
List<int> nedbør = new List<int> { 4, 12, 2, 8 };
int sum = 0;
foreach (int tal in nedbør)
    sum += tal;

// eller
// for (int i = 0; i < nedbør.Count; i++)
//     sum += nedbør[i];

double gns = sum / nedbør.Count;    // sum = 26, gns = 6
```

Her er samme eksempel hvor sum og gennemsnit beregnes i metoder:

```
List<int> nedbør = new List<int> { 4, 12, 2, 8 };
int sum = BeregnSum(nedbør);           // 26
double gns = BeregnGennemsnit(nedbør); // 6

int BeregnSum(List<int> l) {
    int sum = 0;
    foreach (int tal in nedbør)
        sum += tal;
    return sum;
}

double BeregnGennemsnit(List<int> l)
{
    int sum = BeregnSum(l);
```



```
double gns = sum / l.Count;  
return gns;  
}
```

Brug en liste (List<T>) alle de steder, hvor du skal gemme værdier af samme type – det er meget nemmere end et array.

Stak

En anden klassisk datastruktur er en stak (stack på engelsk), som også findes under System.Collections.Generic. Du kan tænke på en stak, som en bunke af spillekort med bagsiden opad. Du kan lægge kort på bunken (Push-metoden), og tage kort af bunken (Pop-metoden), og det kort, du har lagt på bunken sidst, er også det kort, du tager af bunken først. Derfor kaldes en stak en LIFO-struktur (last in – first out).

Strukturen kan være brugbar i mange situationer – herunder i kortspil, hvor den naturligvis er meget brugt.

En Stack<T> kan ligesom List<T> oprettes ud fra en konkret type. Det gælder både de simple variabeltyper samt dine egne typer. Her er eksempelvis en stak af strenge – men husk, det kan være hvad som helst:

```
Stack<string> s = new Stack<string>();  
  
// Tre kort på bunken  
s.Push("Ruder konge");  
s.Push("Spar es");  
s.Push("Hjerter to");  
  
// Kig på det øverste kort - men fjern det ikke  
string a = s.Peek();  
  
// Fjern et kort fra bunken og gem det i en variabel  
string b = "";  
b = s.Pop(); // nu består s af "Ruder konge" og "Spar es"  
// "Hjerter to" kan findes i b  
  
// Hvor mange kort er der i bunken
```

```
int antal = s.Count; // 2

// Udskriv alle kort
foreach (var item in s)
    Console.WriteLine(item);    // Spar es, Ruder konge
```

Kø

At benytte en kø som datastruktur er også meget brugt i programmering – blandt andet til at afvikle funktionalitet i den korrekte rækkefølge. Du kan tænke på en kø, som en kø til kassen i et supermarked. Kunde X står først, så kunde Y, så kunde Z og så videre. Efterhånden som kunderne bliver ekspederet, rykker de andre frem i køen. En kø kaldes derfor for en FIFO-struktur (First In – First Out).

I C# kan du finde en FIFO-datastruktur i klassen `Queue<T>`:

```
Queue<string> kunder = new Queue<string>();

// Sæt kunder i kø
kunder.Enqueue("Y");
kunder.Enqueue("Z");
kunder.Enqueue("W");

// hvor mange er i kø
int antal = kunder.Count;    // 3

// Hvem står først (kigger uden at fjerne)
string a = kunder.Peek();    // a = "Y"

// Ekspeder kunde
string b = kunder.Dequeue();  // b = "Y" og kunder = "Z", "W"

// hvor mange er i kø
antal = kunder.Count;    // 2

// Skriv alle ud
foreach (var kunde in kunder)
{
    Console.WriteLine(kunde);    // "Z", "W"
```

```
}
```

Da Queue er generisk, kan du bruge den til alle typer.

Nøgle og værdi

Nogle gange giver det mening at benytte en datastruktur, som gemmer værdier ud fra en nøgle, og her kan du måske benytte Dictionary-klassen (fra System.Collections.Generic). Ved oprettelse skal du angive, hvilken type du ønsker til nøgle, og hvilken type du ønsker til værdi:

```
// Nøgle = string, Værdi = int
Dictionary<string, int> dic = new Dictionary<string, int>();

// indsæt værdier - først nøgle så værdi
dic.Add("a", 100);
dic.Add("b", 200);
dic.Add("c", 300);

// værdier kan så findes ud fra nøglen
Console.WriteLine(dic["b"]);    // 200

// Findes der en nøgle
Console.WriteLine(dic.ContainsKey("a"));    // true

// Samlingen kan også gennemløbes
foreach (var item in dic)
    Console.WriteLine($"{item.Key} {item.Value}");
```

Der er en masse interessante metoder på klassen, som du selv kan kigge nærmere på, og husk at klassen er generisk (Dictionary<TKey, T>), så både nøgle og værdi kan være hvilken som helst type.

Der findes en del andre klasser under System.Collections.Generic, som du kan kigge nærmere på, men i langt de fleste tilfælde vil du benytte en List<T>.

Hvis du vil vide mere

Når du bliver lidt mere øvet, kan du måske lede efter yderligere information relateret til dette kapitel. Søg eksempelvis efter:

- Implementering af IEnumerable (se senere om interface)
- Brug af indexer og initializers
- Brug af yield.

Dine egne typer

Microsoft stiller en masse typer til rådighed, så du har nemt ved at skrive applikationer. Du skal ikke tænke på variabeltyper, fordi Microsoft har skrevet koden bag `System.Int32`, `System.DateTime`, `System.Boolean` og alle de andre typer. Du skal heller ikke tænke på, hvordan du opretter tilfældige tal (`System.Random`), eller hvordan du tilgår en fil (`System.IO.File`), for det har Microsoft også tilføjet til runtime.

Men Microsoft ved ikke hvilke applikationer, du skal udvikle, eller hvordan du vil strukturere koden, så derfor har du mulighed for at skabe dine helt egne typer.

Hvad er objektorienteret programmering

Lad os antage, at du skal skabe en Yatzy-applikation.

I de traditionelle programmeringsparadigmer som iterativ og procedural programmering skal du fokusere på *flowet* i applikationen. Når spillet afvikles, skal du løbende opbevare data i variable og arrays med spillernes navne, deres point, hvis tur det er og værdien af et slag med terningerne. Og du vil sikkert skabe metoder som `SpillerMedFlestPoint`, `ErDerFuldtHus`, `ErDerYatzy`, `ErSpilletSlut` og så videre for at gøre koden så nem at udvikle og vedligeholde som muligt. Du vil uden de store problemer, kunne skabe et Yatzy-spil på den måde. Sådan er millioner af applikationer skrevet siden 1950'erne og bliver det uden tvivl også i skrivende stund.

I objektorienteret programmering, eksempelvis ved hjælp af et sprog som C#, kan du vælge (du behøver ikke – det er et valg fra din side) at benytte objektorienterede principper for at gøre koden nemmere at skrive, forstå, vedligeholde og teste – både for dig selv og andre. Du kan vælge at finde elementer (kaldes også entiteter) i applikationen, som du kan repræsentere med en definition. Denne definition kaldes også en skabelon, og i C# kan du vælge mellem en *klasse* (class), *post*

(record) eller en *struktur* (struct). I grundlæggende C# kan du se bort fra at oprette strukturer, og poster bliver gennemgået senere (side 316). Lige nu skal du blot fokusere på klasser.

En klasse er en skabelon, der ligger til grund for oprettelse af objekter, som lever i din applikation. Objekter bliver også kaldt for instanser, men der er tale om det samme. Disse objekter bliver oprettet i din kode og eksisterer i hukommelsen i det tidsrum, du ønsker. De mest simple klasser består blot af en enkelt variabel, der repræsenterer din entitet, men du kan tilføje så mange variabler, du ønsker. Du kan dermed se klasser som relaterede variabler, der repræsenterer et eller andet.

Hvis du vil skabe Yatzy-spil i objektorienteret kode, skal du starte med at lede efter entiteter, og du vil hurtigt kunne identificere entiteter som en terning, et bæger med terninger, en pointtavle, en spiller, selve spillet og så videre. Du skal prøve at finde så atomare entiteter som muligt forstået således, at en entitet ikke logisk kan opdeles yderligere. En terning er eksempelvis atomar fordi det næppe vil give mening at tænke på en terningside som en enkeltstående entitet, og en terning dermed består af en samling af terningsider. Men det er et arkitektonisk valg.

Du kunne også vælge at se en spilleplade som en entitet med et navn og en samling af point, men en mere erfaren udvikler vil måske se entiteter som spiller, point og en spilleplade som består af en spiller og en samling point. Det er netop det, der er svært i objektorienteret programmering – at identificere entiteter. Det kræver erfaring og øvelse.

Hvis du eksempelvis skulle skabe et ERP (Enterprise Resource Planning) system vil du kunne identificere entiteter som person, selskab, kunde, vare, lager, faktura, fakturalinje og så videre.

Tit er de entiteter, du finder, noget du kan relatere til fysiske ting – en person, en terning, en vare og så videre, men det kan også være

abstrakte ting som eksempelvis noget, der repræsenterer flowet i et Yatzy-spil, en risikovurdering på en kunde eller en algoritme til at beregne effektiv rente på en obligation.

Mange projekter starter med en brainstorm over entiteter, og et stort whiteboard, cola og pizza plejer at være en god måde at komme i gang på. Der findes også flere strukturerede måder at skabe sig et overblik over en applikation, og UML (Unified Modeling Language) er en af de mest kendte standarder. Nogle udviklere kan også godt lide blot at definere entiteter i koden efterhånden som de dukker op. Det bruges eksempelvis i moderne agil udvikling.

Terningen

Jeg vil bede dig om at gøre en ting, som du vil synes er underligt – måske endda tåbeligt. Men stol på mine mindst 15 års erfaring i undervisning i programmering og bare gør, hvad jeg beder om.

Du skal i en skuffe et sted finde en almindelig terning! Altså en terning fra et Yatzy- eller et brætspil, med seks sider og numrene 1-6. En ganske almindelig terning – størrelse og farve er underordnet, men hvis du kan vælge så tag den terning, du umiddelbart bedst kan lide.



Figur 55 En OOP-terning²⁰

²⁰ Billedet er fra Pixabay (<https://pixabay.com/vectors/dice-cube-die-game-gambling-luck-152179/>)

Det næste stykke tid vil jeg gerne have, at du har terningen i nærheden af dig – i hånden, i lommen, i tasken eller stående på bordet ved siden af dit tastatur. Terningen skal på en eller anden måde være i din bevidsthed et stykke tid, og du skal helst sørge for at have fingrene i den et par gange om dagen. Terningen skal minde dig om, hvordan objektorienteret kode fungerer og holde disse tanker friske i din hukommelse.

Vi skal bruge terningen til flere ting, men til at starte med skal vi se på, hvordan du kan skabe en klasse (skabelon), der repræsenterer en terning. Denne klasse kan ligge til grund for objekter, og allerede nu er du i gang med at tænke objektorienteret, fordi det som udgangspunkt er ligeegyldigt i hvilke terningespil, du skal bruge klassen. En terning er en terning, og du kan derfor genbruge klassen i mange forskellige applikationer.

Medlemstyper

I en klasse kan du skabe forskellige typer af funktionalitet - det kaldes *medlemstyper*, når det er relateret til strukturer og klasser. I C# har du mulighed for *felter*, der repræsenterer klassens data, *egenskaber* som typisk bruges til at beskytte felterne imod forkert tildeling og aflæsning, *metoder* der kan bruges mod klassens data, og *hændelser* som giver mulighed for at få afviklet kode, når en speciel hændelse sker. Sluttelig findes der kode, som afvikles, når der oprettes et objekt (kaldes en *konstruktør*), og kode som afvikles, når et objekt bliver fjernet fra hukommelsen af runtime (kaldes en *destruktør*). Du bestemmer helt selv, hvor mange felter, egenskaber, metoder, hændelser og konstruktører du ønsker, men der kan kun være én destruktør.

For god ordens skyld er her navnene på alle de mulige medlemstyper på både dansk og engelsk:

<i>Medlemstype på dansk</i>	<i>Medlemstype på engelsk</i>
<i>Felt/Felter</i>	<i>Field/Fields</i>
<i>Egenskaber</i>	<i>Property/Properties</i>
<i>Metode/Metoder</i>	<i>Method/Methods</i>
<i>Hændelse/Hændelser</i>	<i>Event/Events</i>
<i>Konstruktør</i>	<i>Constructor</i>
<i>Destruktør</i>	<i>Destructor</i>

Tabel 21 Medlemstyper

Synlighed

Alle medlemstyper kan have forskellig synlighed. For nemmest at kunne forstå dette begreb er det vigtigt, at du kan se en klasse fra to udvikleres synspunkt – den udvikler, der *skriver* koden til klassen, og den udvikler, som *benytter* klassen. Det kan naturligvis godt være en og samme person, men det er nemmere at forstå, hvis de to personer er adskilt.

En medlemstype kan i helt grundlæggende C# være enten *privat* eller *offentlig*. En privat medlemstype kan udelukkende benyttes af andre medlemmer i klassen selv, og kan slet ikke ses uden for klassen. Så den er altså tilgængelig for udvikleren, som skriver koden til klassen, men ikke for udvikleren, der benytter klassen. Derfor siger man, at medlemstypen er privat og ikke synlig *udefra*.

Et medlem i en klasse kan enten være offentlig eller privat.

En offentlig medlemstype kan benyttes af *alle* - både af medlemmer i klassen selv og af kode, der benytter klassen. Eller sagt på en anden måde. En offentlig medlemstype kan både benyttes af den udvikler, som skriver koden til klassen, og af den udvikler, som benytter klassen til at skabe objekter.

Du kan altså godt skabe en klasse med mange medlemstyper, hvoraf nogle er private og andre er offentlige. På den måde kan du *gemme* nogle medlemmer væk som er ligegyldige for brugeren af klassen.

Da du sikkert støder på synlighedsbegrebet i dokumentation og artikler er her den engelske oversættelse.

<i>Synlighed på dansk</i>	<i>Synlighed på engelsk</i>
<i>privat</i>	<i>private</i>
<i>offentlig</i>	<i>public</i>

Tabel 22 Synlighed

Alle disse forskellige medlemstyper og deres synlighed kan virke lidt abstrakt, men hvis du tænker på din terning, kan vi gøre det mere håndgribeligt.

Felter

Hvis du skal skabe noget, som skal repræsentere en terning, skal du i hvert fald have ét *felt* til at opbevare værdien af terningen. Den kan jo have værdien 1-6, så du skal finde en datatype som passer. Det kunne være en af de mange heltalstyper som `byte` eller `int`, men det er helt op til dig. Måske en streng eller en enumeration kunne bruges?

Kan du komme i tanke om andre felter? Måske et boolsk felt til at indeholde værdien sand eller falsk for at indikere om det er en snyd-terning, som kun kan få værdien 6? Måske et felt til at repræsentere den farve, der skal bruges, når der skrives ud? Måske et `DateTime`-felt som kan gemme information om, hvornår terningen sidst er rystet?

Felter er normalt en medlemstype, der gemmes væk for den, der benytter klassen ved at gøre dem `private`. På den måde kan du beskytte data, og selv sørge for at skabe tilgang gennem andre medlemstyper som eksempelvis metoder – eller som du skal se senere – egenskaber.

Felter repræsenterer klassens data.

Felter er altså terningens data, og hvert objekt baseret på klassen har disse data placeret i hukommelsen. Det er ikke noget, du tager dig af – det klarer runtime.

I relation til klassen Terning kan vi lige nu nøjes med ét felt kaldet værdi og gøre dette felt til et heltal. Feltet er privat, hvilket betyder, at det kun kan tilgås inde fra klassens andre medlemmer. Det indikerer jo, at klassen nu mangler nogle offentlige medlemmer, så man udefra kan tildele og aflæse en værdi.

Egenskaber

I relation til terningen har du dog et problem med feltet værdi. Det er jo af en heltalstype, og der findes ingen typer, som kun tillader værdien 1-6. Det kan vi jo blot dokumentere os ud af ved at skrive, at ingen, der benytter vores terning, må tildele feltet værdi under 1 eller over 6. Men det var bedre, hvis terningen selv kunne holde styr på det, og det er her, *egenskaber* kommer ind i billedet. De kan bruges til at beskytte et felt og afvikle kode ved tildeling og ved aflæsning af feltet. Således kan vi skabe en egenskab kaldet Værdi (stort V i modsætning til feltet som staves med lille v), og skrive kode som sikrer, at der kun kan tildeles 1-6 ved, at en for lille eller høj værdi resulterer i en fejl (exception). Du kunne også tilføje kode, der afvikles, når feltet værdi aflæses. Hvis der er tale om en snydeterning, skal den eksempelvis returnere værdien 6 og ikke værdien af feltet. I større systemer er koden i egenskaberne også tit relateret til eksempelvis sikkerhed og log.

Egenskaber beskytter klassens felter ved tildeling og aflæsning.

Egenskaberne beskytter altså typisk felterne, og giver en masse fordele som vi kommer nærmere ind på. I grundlæggende C# hænger et felt sammen med en tilhørende egenskab, og Microsoft foreslår en navngivningsstandard hvor feltet er stavet med lille og egenskaben med stort.

Metoder

I en klasse (eller struktur) kan du ligeledes tilføje metoder med samme syntaks, som du har set tidligere i bogen. Metoderne er typisk relateret til klassen felter (data), men behøver ikke at være det.

Metoder repræsenterer klassens funktionalitet
og er tit relateret til klassens felter.

I Terning-klassen er i hvert fald én metode nødvendig. Der skal være en mulighed for at ryste terningen, hvilket i kode svarer til at tildele feltet værdi et tilfældig tal mellem 1 og 6. Men der kunne godt være andre metoder. Hvad med en boolsk metode der fortæller, om terningen har en given værdi, en metode der udskriver værdien eller metoder, der gemmer/henter værdien fra en fil. Der er masser af muligheder.

Hændelser

Noget mere avanceret er den medlemstype, der kaldes *hændelser* (events). For at forstå denne medlemstype er det igen vigtigt, at du kan se klassen Terning fra to udviklers synspunkt – den udvikler, der *skriver* klassen Terning, og den udvikler, som *benytter* klassen Terning.

Hændelser er en mulighed for, at objekter af en klasse selv kan afvikle en metode når en eller anden hændelse sker, og denne metode er blevet tildelt af den, der benytter klassen Terning. Du kan med et godt programmeringsbegreb sige, at hændelser giver mulighed for at *afkoble* funktionalitet. Det er ikke udvikleren af klassen, der bestemmer, hvad der skal ske – men brugeren af klassen.

I relation til terningen kunne det måske være smart, at objekter af klassen terning kunne afvikle en metode, når værdien "rystes" til en 6'er. Så kunne en udvikler, der bruger terningen, måske få applikationen til at give lyd, når det bliver en 6'er, og en anden kunne måske gøre noget grønt på en brugerflade. Pointen er, at udvikleren af

terningen ikke skal skrive koden til en given hændelse – det er op til brugeren af terningen.

En hændelse giver brugeren af en klasse mulighed at få afviklet en eller flere metoder, når en given hændelse finder sted.

Du kan med rette tænke, at brugeren af terningen jo blot selv kan skrive koden til at kontrollere om værdien er en 6'er og så selv afvikle sin egen kode. Og det er helt korrekt, men du skal tænke på, at der kan være situationer, hvor det kan være svært at vide, hvor man skal kontrollere en værdi, og så er alternativet et uendeligt loop eller en timer, der hele tiden kontrollerer værdier. Der kan også ligge en kompliceret logik bag en hændelse, og det derfor giver god mening at have kodet dette i terningen selv.

Konstruktører

Det kan tit være nødvendigt at afvikle kode, når et objekt bliver oprettet.

Der kan både være tale om kode relateret til initialisering (alle felter bliver sat til default værdier, og det er måske ikke optimalt), sikkerhed, log og så videre. Dette kan ske i en classes konstruktører. Du kan vælge at tilføje konstruktører både med og uden argumenter således, at du kan oprette objekter på flere måder.

I relation til terningen kan man forestille sig, at klassen har to konstruktører.

Hvis brugeren af klassen Terning opretter et objekt uden argumenter, sørger en konstruktør for at kalde en metode, som tildeler et tilfældigt tal til feltet værdi. På den måde sikrer du, at feltet værdi ikke har en værdi på 0, hvilket jo ikke giver mening for en terning.

En konstruktør er en samling instruktioner, der afvikles, når der bliver skabt et objekt af en klasse.

Hvis brugeren af klassen `Terning` opretter et objekt med et heltal som argument, kan en konstruktør sørge for at kontrollere om argumentet er mellem 1 og 6, og så tildele argumentet til feltet værdi. Så kan brugeren oprette en terning uden, at den får en tilfældig værdi.

Destruktør

Det modsatte af en konstruktør er en destruktør, som afvikler kode, når objektet fjernes fra hukommelsen af runtime (reelt af en komponent i frameworket kaldet en *Garbage collector*).

Det er sjældent, man benytter destruktører i C#, fordi runtime er så god til at rydde op efter sig. Der kan være enkelte situationer, hvor det er nødvendigt, men i grundlæggende C# kan du se helt bort fra dem. De gør typisk mere skade end gavn, fordi de nemt kan benyttes forkert.

I relation til terningen giver det heller ikke nogen mening af tilføje en destruktør.

Destruktører benyttes ikke særlig meget i grundlæggende C# – blandt andet fordi runtime stiller andre features til rådighed, hvis du ønsker at afvikle oprydningsskode.

Eksempler på brug af medlemstyper

Indtil nu har kapitlet bestået af en masse teori, og det kan måske give mere mening, hvis du ser lidt kode. Du skal ikke fokusere på, hvordan klassen er skrevet her, det lærer du i de efterfølgende kapitler, men mere på brugen af de forskellige medlemmer.

Her er eksempelvis brugen af de to konstruktører:

```
// Her oprettes en terning, og den konstruktør uden
```

```
// argumenter bliver afviklet automatisk. Det betyder,  
// at Ryst-metoden bliver kaldt automatisk, og feltet  
// værdi dermed allerede ved oprettelse bliver tildelt  
// et tilfældigt tal mellem 1-6  
Terning t1 = new Terning();  
  
// Her oprettes en terning, og den konstruktør med et enkelt  
// argumentet bliver afviklet automatisk. Det betyder,  
// at feltet værdi dermed bliver tildelt værdien 2  
Terning t2 = new Terning(2);  
  
// Array af terninger som alle rystede  
Terning[] bøger = new Terning[5];  
for (int i = 0; i < 5; i++) {  
    bøger[i] = new Terning();  
}
```

Her er eksempler på brug af egenskaben Værdi:

```
// Her oprettes en terning med værdien 3  
Terning t1 = new Terning(3);  
// Her sættes terningens værdi til 4  
t1.Værdi = 4;  
// Her aflæses terningens værdi  
Console.WriteLine(t1.Værdi);  
  
// Dette vil skabe en fejl fordi  
// værdi skal være >=1 og <=6  
t1.Værdi = 7;
```

Her benyttes terningens metode:

```
Terning t1 = new Terning();  
// Her "rystes" terningen og får dermed en ny værdi  
t1.Ryst();  
Console.WriteLine(t1.Værdi);
```

Sluttelig er her et eksempel på brugen af hændelsen:

```
Terning t1 = new Terning();  
// Metoden Beep bindes til ErSekser-hændelsen  
t1.ErSekser += Beep;  
for (int i = 0; i < 10; i++)
```

```
{  
    Console.WriteLine(t1.Værdi);  
    t1.Ryst();  
}  
  
// Denne metode afvikles automatisk når  
// værdien bliver en sekser  
void Beep(object sender, EventArgs e)  
{  
    // Her kan være alt mulig kode  
    Console.Beep();  
}
```

Eksemplet er lidt mere avanceret, og du skal ikke forstå syntaksen. Det som er væsentligt er, at brugeren af klassen Terning bestemmer, hvad der skal ske, når der rystes til en sekser, og klassen sørger selv for at få det til at ske.

Konklusion

Det var en masse teori om medlemstyper og synlighed, og i de efterfølgende kapitler skal du lære at skrive koden. Men det er vigtigt, at du har en grundlæggende forståelse for teorien, for det er grundstammen i objektorienteret programmering.

Brug terningen til at huske på at:

- Du kan bruge klasser til at simulere, emulere, abstrahere og repræsentere en entitet. Klassen Terning er en måde at beskrive en terning i kode.
- En af styrkerne ved objektorienteret programmering er genbrug af kode. Hvis først du har skabt klassen Terning, kan den benyttes i alle mulige terningespil, og i den mere avancerede C# kan du sågar *arve* funktionalitet til andre klasser. Mere om det senere.
- Felter er objekternes data. I eksemplet med terningen gemmes værdien i et felt.

- Egenskaber bruges til at beskytte felterne mod forkerte værdier og forkert aflæsning. I eksemplet med terningen beskytter egenskaben kaldet Værdi feltet kaldet værdi ved, at egenskaben er offentlig (og dermed kan tilgås udefra), og feltet er privat.
- Metoder er en måde at arbejde med objekternes felter. I eksemplet med terningen bruges Ryst-metoden til at skabe et tilfældigt tal.
- Hændelser giver mulighed for, at objekter afvikler metoder, når en hændelse sker. I eksemplet med terningen giver klassen mulighed for at tilføje en reference til en metode, der afvikles, når der rystes en sekser.

Klasser

En klasse i C# kan skabes med *class*-kodeordet, og med definitionen angives samtidig klassens synlighed:

```
[synlighed] class [navn]
{
    // medlemmer
}
```

Synligheden kan enten være *internal*, som fortæller, at klassen kun er synlig i det projekt, den er defineret, eller *public* som fortæller, at andre projekter med en reference til projektet, hvor klassen er defineret, kan se og benytte klassen. Hvis du ikke angiver noget, er den som standard *internal*. Men til at starte med bør du angive synligheden, så den er tydelig for dig selv og andre, der arbejder med koden.

De fleste udviklere benytter Microsofts navngivningsstandard der blandt andet fortæller, at klassenavnet skal starte med stort og derefter benytter *camelcasing* (hvert ord starter med stort bortset fra det første), men det er helt op til dig.

Her er et par eksempler:

```
internal class Faktura
{
}
```

```
internal class Person
{
}
```

```
internal class Terning
{
}
```

```
internal class PointTavle
{
}
```

Klassen bør defineres på namespace-niveau, så hvis du tilføjer klasser til en konsol-applikation, bør den placeres på linje med Program-klassen:

```
using System;

namespace Demo
{
    internal class Program
    {
        public static void Main()
        {
        }
    }

    internal class Terning {

    }
}
```

Bemærk, at klassen er placeret i Demo-namespacet på samme niveau som Program-klassen.

De fleste vælger dog at placere en klasse (eller andre typer) i en fil for sig selv med samme navn som typen. Det kan du nemt gøre i Visual Studio ved at højreklikke på projektet i Solution Explorer-vinduet, vælge Add... og herefter Class. I Visual Studio Code skal du blot tilføje en cs-fil eller benytte en extension, der kan hjælpe.

Om du placerer alle typer i en og samme fil eller i hver sin fil er underordnet – kompileren vil sørge for at compilere alle filer sammen.

I VS og VSC kan du altid flytte en type til en ny fil ved at klikke på typenavnet, finde det lille lyspære-ikon og vælge "move type to ..."

Oprettelse af objekter

Oprettelse af objekter sker ved hjælp af new-kodeordet, hvilket vil oprette et objekt (kaldes også en instans) i hukommelsen. Da en klasse er en referencebaseret type, kan referencen gemmes i en variabel. Du kan enten benytte en eksisterende variabel eller erklære en ny. I grundlæggende C# kan du blot benytte en variabel af den konkrete type, men i mere avanceret kode kan du begynde at udnytte objekt-orienterede muligheder som eksempelvis arv eller interface – mere om det senere i bogen.

Lige nu skal du blot gemme et objekt i en variabel af samme type.

Her er et par eksempler, der tager udgangspunkt i en klasse Terning:

```
// Erklær en variabel der kan holde referencen
// til et objekt af typen Terning
Terning a;
```

```
// Opret et nyt objekt og gem referencen i a
a = new Terning();
```

```
// Erklær en variabel der kan holde referencen
// til et objekt af typen Terning, og opret
// objekt med det samme
Terning b = new Terning();
```

```
// Erklær en variabel der kan holde referencen
// til et objekt af typen Terning, og tildel
// variabelen referencen fra b
Terning c = b;
// Nu er reference i c og b ens - de peger
// på samme objekt i hukommelsen
```

I et senere kapitel kommer vi meget mere i dybden omkring hukommelsesteori, men det er vigtigt, at du ser oprettelse af et objekt som tre separate operationer:

```
Terning a = new Terning();
```

Denne linje skal du læse som

- Erklæring af en variabel
- Oprettelse af et nyt objekt med new
- Tildeling af reference fra det nye objekt til variabelen.

Hvis du læser det på den måde, er det lidt nemmere at forstå.

Bemærk, at brugen af new() er en C# 9 feature!

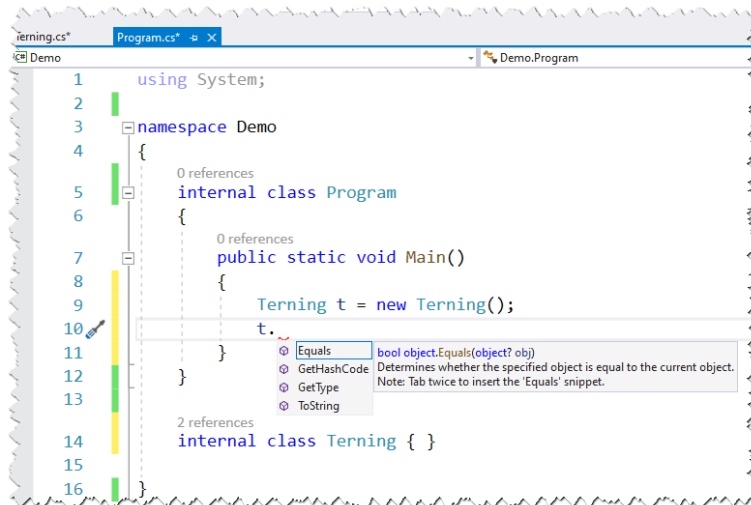
I nyere C# kan du vælge at bruge en lidt nemmere syntaks:

```
Terning a = new();
```

Kompileren er smart til at udlede, at a tildeles referencen til et nyt objekt af typen Terning, men som begynder kan det være en fordel at skrive, hvilken type du ønsker en ny instans af. Det er lidt mere logisk at skrive og læse.

Medlemmer fra System.Objekt

Selv om du skaber objekter af helt tomme klasser, vil du måske undre dig over, at der alligevel findes nogle metoder på objektet, hvis du kigger på det i Visual Studio.



Figur 56 Metoder fra System.Object

Det skyldes, at alle typer automatisk har fire metoder fra den øverste klasse i .NET kaldet Object. Det er sjældent, du i grundlæggende C# vil bruge andre metoder end metoden ToString, som har til formål at skabe en streng, der beskriver objektet bedst muligt. Senere i bogen vil du lære at skrive din egen implementering af denne metode, men indtil da vil metoden blot returnere typenavnet.

Felter

Du kan opfatte felter i typer (klasser og strukturer) som muligheden for at opbevare data. Du kan også kalde felter for typens variabler.

De er det eneste, der adskiller objekter af samme type fra hinanden idet alle andre medlemstyper (metoder, egenskaber og hændelser) er nøjagtig ens og i virkeligheden kun eksisterer et sted i hukommelsen. Objekterne indeholder blot en reference til dem.

Felter er objekternes data.

Felter kan blandt andet erklæres som private (private) eller offentlige (public). Offentlige felter kan tilgås både udefra og inde fra selve objektet, medens private blot kan tilgås inde fra objektets andre medlemmer.

Her er eksempler på forskellige klasser med offentlige felter:

```
internal class Terning {
    public int Værdi;
}

internal class TerningeBæger {
    public Terning[] Terninger;
    public DateTime SidstRystet;
}

internal class Person {
    public string Navn;
    public int Alder;
    public bool ErDansk;
}
```

Når der oprettes objekter af klasserne, bliver felterne automatisk tildelt typernes default værdier:

```
Person person1 = new Person();
// nu har
// person1.Alder værdien 0
// person1.Navn værdien null
// person1.ErDansk værdien False
```

Men du kan naturligvis tildele felterne andre værdier:

```
Terning terning = new Terning();
terning.Værdi = 5;

TerningeBæger bæger = new TerningeBæger();
bæger.SidstRystet = DateTime.Now;
bæger.Terninger = new Terning[5];

Person person1 = new Person();
```

```
person1.Alder = 14;  
person1.Navn = "Mathias";  
person1.ErDansk = true;
```

I nyere versioner af C# kan man både erklære, oprette og tildele i en og samme instruktion:

```
Person person2 = new Person()  
{  
    Navn = "Mikkel",  
    Alder = 16,  
    ErDansk = true  
};
```

Det kan naturligvis også gøres på én lang linje:

```
Person person2 = new Person() { Navn = "Mikkel", Alder = 16, ErDansk =  
true };
```

Det er helt op til din kodestandard – kompileren er ligeglad.

Offentlige data

Du vil i virkeligheden *aldrig* erklære felter som offentlige, fordi du dermed ikke kan afvikle instruktioner, når der tildeles og aflæses værdier. Du er nødt til at kunne *indkapsle* data og styre tilgangen, og som du skal se senere, kan dette ske ved at gøre felter *private*, og tilføje offentlige medlemmer, som blot skaber adgang til felterne.

Som eksempel kan du tænke på din Terning:

```
internal class Terning {  
    public int Værdi;  
}
```

Hvis der skabes en instans af terningen, kan værdi tildeles hvad som helst, så længe der er tale om lovlige værdier for en `Int32`:

```
Terning t = new Terning();  
t.Værdi = -1;
```


Som udvikler af klassen `Terning` er det ikke så smart, at du ikke ved, hvornår værdi tildeles og aflæses, og i eksemplet er terningens værdi blevet tildelt -1, hvilket jo ikke giver nogen mening. Men heldigvis kan vi sikre felterne på anden vis, som du skal se senere.

Jeg plejer at sige til mine kursister, at offentlige felter er den direkte vej til en medarbejdersamtale. Det er naturligvis ment i spøg, men lige nu skal du bare forstå begrebet felter, og må (indtil du har læst om egenskaber og indkapsling) gerne gøre brug af offentlige felter.

Brug af `this`

Når du fra medlemmer i en klasse tilgår andre medlemmer, kan du vælge, om du vil benytte *this*-kodeordet:

```
internal class Terning
{
    public int Værdi;

    public void Ryst()
    {
        System.Random rnd = new Random();
        // Tilfældig værdi fra 1-6
        this.Værdi = rnd.Next(1, 7);
    }
}
```

Koden bliver lidt mere logisk at læse, og Visual Studio kan hjælpe lidt mere ved at give dig forslag. Den ved jo, at der er tale om objektet selv, så brug af `this` giver en fin dropdown-liste med muligheder.

Men kompileren er ligeglad, så denne kode er lige så god:

```
internal class Terning
{
    public int Værdi;
```

```
public void Ryst()
{
    System.Random rnd = new Random();
    // Tilfældig værdi fra 1-6
    Værdi = rnd.Next(1, 7);
}
}
```

Det er op til den kodestandard, du benytter, men jeg vil anbefale brugen af `this`, hvis du er ny i C#.

Metoder

Instans metoder i klasser er som nævnt typisk relateret til klassens felter i en eller anden form, og syntaksen på metoderne er, som du tidligere har set. Du kan vælge om en metode skal returnere en konkret værdi eller blot er en samling instruktioner til afvikling (void-metode).

Her er et par eksempler på metoder i klassen `Terning`:

```
internal class Terning
{
    public int Værdi;

    public void Ryst()
    {
        Random rnd = new Random();
        this.Værdi = rnd.Next(1, 7);
    }

    public string VærdiTilPrint()
    {
        return $"[ {this.Værdi} ]";
    }

    public void SkrivTilConsole(ConsoleColor farve)
    {
        ConsoleColor gammelFarve = Console.ForegroundColor;
```

```
        Console.ForegroundColor = farve;
        Console.WriteLine(this.VærdiTilPrint());
        Console.ForegroundColor = gammelFarve;
    }

    public bool ErSekser()
    {
        return this.Værdi == 6;
    }
}
```

Bemærk, at de alle i en eller anden form har med feltet Værdi at gøre, men det behøver ikke være tilfældet. Her er eksempler på brug af metoderne:

```
Terning t = new Terning();
t.Ryst();
Console.WriteLine(t.VærdiTilPrint());
t.SkrivTilConsole(ConsoleColor.Red);
Console.WriteLine(t.ErSekser());
```

Der er ikke syntaks-mæssigt nogen forskel på disse metoder og de metoder, der er gennemgået tidligere. Du kan således bruge navngivne argumenter, argumenter med en standardværdi, overloads med videre.

Konceptuelt er der dog den forskel, at kapitlet om metoder benyttede statiske metoder, og de metoder, der nævnes her, er instans metoder. Forskellen er, om en metode skal have adgang til instansens data, eller om den skal tilgås fra typen.

Her er klassen Terning med en instans metode Ryst (den skal have adgang til feltet Værdi) og en statisk metode FindTilfældig-TerningVærdi, som blot er en hjælpemetode, som måske kunne være praktisk. Ved at kalde den, kan du finde en tilfældig værdi uden at skabe en instans:

```
internal class Terning
{
```

```
public int Værdi;

public void Ryst()
{
    Random rnd = new Random();
    this.Værdi = rnd.Next(1, 7);
}

public static int FindTilfældigTerningVærdi() {
    Random rnd = new Random();
    return rnd.Next(1, 7);
}
}
```

Her er begge metoder i brug:

```
// Instans metode
Terning t = new Terning();
t.Ryst();

// Statisk metode
int v = Terning.FindTilfældigTerningVærdi();
```

Bemærk at Ryst-metoden kan findes på instansen, men FindTilfældigTerningVærdi-metoden kan findes på klassen Terning.

Standard konstruktør

Som tidligere nævnt er en konstruktør (constructor) kode, der afvikles, når der oprettes et objekt med new operatoren. Det bruges blandt andet til initialisering, log, sikkerhed og logik. En konstruktør ligner en metode, men der er ikke angivet nogen returværdi, og navnet på en konstruktør er det samme som navnet på klassen.

Den mest simple konstruktør har ingen argumenter og kaldes for standard-konstruktøren (default constructor). Den kunne bruges i vores terning for at sikre, at feltet ikke har værdien 0, når der oprettes et objekt:

```
internal class Terning
{
    public int Værdi;

    // Konstruktør
    public Terning()
    {
        this.Værdi = 1;
    }
}
```

Når der oprettes et objekt, vil værdi automatisk blive tildelt 1, og dermed overskrive standardværdien på 0:

```
Terning t = new Terning();
Console.WriteLine(t.Værdi); // 1
```

Måske kunne det give mening, at terningens værdi blev tildelt et tilfældigt tal mellem 1 og 6, når den bliver oprettet:

```
internal class Terning
{
    public int Værdi;

    public Terning()
    {
        this.Ryst();
    }

    public void Ryst()
    {
        System.Random rnd = new Random();
        // Tilfældig værdi fra 1-6
        // det sidste argument - 7 - er rigtig nok
        this.Værdi = rnd.Next(1, 7);
    }
}
```

Bemærk, at konstruktøren vises med et navn svarende til klassen selv – ligesom i koden.

Nu er værdien tilfældig ved oprettelse:

```
Terning t = new Terning();  
Console.WriteLine(t.Værdi); // tilfældigt tal mellem 1-6
```

Her er et andet eksempel:

```
internal class TerningeBæger  
{  
    public Terning[] Terninger;  
  
    public TerningeBæger()  
    {  
        this.terninger = new Terning[5];  
        for (int i = 0; i < 5; i++)  
            this.Terninger[i] = new Terning();  
    }  
}
```

Når der skabes en instans af klassen, bliver der oprettet et array som tildeles nye terninger, og hvis terningerne automatisk får en tilfældig værdi i konstruktøren i klassen Terning, vil de jo automatisk få en tilfældig værdi:

```
TerningeBæger b = new TerningeBæger();  
// Fem terninger med tilfældige værdier  
foreach (Terning terning in b.Terninger)  
    Console.WriteLine(terning.Værdi);
```

En konstruktør består typisk af initialiseringskode, men det er helt op til dig. Du må gerne tilføje eksempelvis sikkerhed, som sikrer, at man kun kan oprette et objekt, hvis man har rettigheder til det.

Du kan nemt tilføje en konstruktør i Visual Studio ved at bruge en snippet. Placer cursoren i klassen og skriv *ctor* og tryk to gange tabulering.

Brugerdefineret konstruktør

En brugerdefineret konstruktør kan du opfatte som en konstruktør med argumenter, og kan således bruges til at oprette objekter på flere måder. Du bestemmer selv, hvor mange konstruktører, du ønsker, og om du udelukkende vil have brugerdefinerede konstruktører.

I eksemplet med terningen kunne det være smart at kunne oprette en terning både med og uden argument svarende til værdien på terningen:

```
internal class Terning
{
    public int Værdi;

    // Afvikles hvis terning oprettes uden
    // argumenter
    public Terning()
    {
        this.Ryst();
    }

    // Afvikles hvis terning oprettes med
    // en int som argument
    public Terning(int startVærdi)
    {
        this.Værdi = startVærdi;
    }

    public void Ryst()
    {
        System.Random rnd = new Random();
        this.Værdi = rnd.Next(1, 7);
    }
}
```

Nu kan terningen oprettes på to måder:

```
Terning t1 = new Terning();
Console.WriteLine(t1.Værdi);    // Tilfældig værdi
```

```
Terning t2 = new Terning(2);  
Console.WriteLine(t2.Værdi);    // 2
```

Terningen kunne også udvides med et par ekstra konstruktører, hvor man kan angive en *snydefaktor*, som betyder at Ryst-metoden altid giver en 6'er:

```
internal class Terning  
{  
    public int Værdi;  
    public bool SnydeFaktor;  
  
    public Terning()  
    {  
        this.Ryst();  
    }  
  
    public Terning(int startVærdi)  
    {  
        this.Værdi = startVærdi;  
    }  
  
    public Terning(bool snydeFaktor)  
    {  
        this.Værdi = 6;  
        this.SnydeFaktor = snydeFaktor;  
    }  
  
    public void Ryst()  
    {  
        if (this.SnydeFaktor == false)  
        {  
            System.Random rnd = new Random();  
            this.Værdi = rnd.Next(1, 7);  
        }  
        else  
        {  
            this.Værdi = 6;  
        }  
    }  
}
```



```
    }  
  }  
}
```

Nu kan du skabe en terning på tre måder:

```
Terning t1 = new Terning();  
Console.WriteLine(t1.Værdi);    // Tilfældigt tal  
t1.Ryst();  
Console.WriteLine(t1.Værdi);    // Tilfældigt tal  
  
Terning t2 = new Terning(3);  
Console.WriteLine(t2.Værdi);    // 3  
t2.Ryst();  
Console.WriteLine(t2.Værdi);    // Tilfældigt tal  
  
Terning t3 = new Terning(true);  
Console.WriteLine(t3.Værdi);    // 6  
t3.Ryst();  
Console.WriteLine(t3.Værdi);    // 6
```

Læg mærke til at klassen nu har et ekstra felt kaldet *SnydeFaktor*. Det skal gemme den værdi, der angives i den ene konstruktør og bruges i øvrigt i *Ryst*-metoden.

Hvis en klasse har en eller flere brugerdefinerede konstruktører, behøver du ikke også have en standard konstruktør. På den måde kan du tvinge brugeren af en klasse til at oprette objekter på en konkret måde.

Genbrug af konstruktører

Gentagelse af kode er roden til alt ondt, og du skal så vidt muligt forsøge at centralisere kode, så du kun skal rette ét sted. I relation til konstruktører kan der eksempelvis være log- eller sikkerhedskode, som altid skal køres. Du kan vælge at lade konstruktører kalde en fælles metode som eksempelvis:

```
internal class Terning
{
    public int Værdi;

    public Terning()
    {
        this.Værdi = 1;
        this.Check();
    }

    public Terning(int startVærdi)
    {
        this.Værdi = startVærdi;
        this.Check();
    }

    private void Check()
    {
        if (DateTime.Now.DayOfWeek == DayOfWeek.Sunday)
            throw new
                Exception("Du må ikke bruge terningen på en søndag");
    }
}
```

Her kalder begge konstruktører metoden Check, som sikrer, at terningen ikke kan benyttes på en søndag.

Alternativt kan du få konstruktører til at kalde hinanden ved hjælp af this-kodeordet:

```
internal class Terning
{
    public int Værdi;

    // Kalder først Terning(int)
    // inden resten af konstruktøren
    // afvikles
    public Terning() : this(1)
    {
        // Kode der evt skal afvikles
    }
}
```

```
}

public Terning(int startVærdi)
{
    if (DateTime.Now.DayOfWeek == DayOfWeek.Sunday)
        throw new Exception(
            "Du må ikke bruge terningen på en søndag");

    this.Værdi = startVærdi;
}
}
```

Terningen kan stadig oprettes på to måder, men hvis du benytter standardkonstruktøren, bliver den brugerdefinerede konstruktør kaldt først, og herefter afvikles eventuelle instruktioner i standardkonstruktøren.

Hvis du vil vide mere

Når du bliver lidt mere øvet, kan du måske lede efter yderligere information relateret til dette kapitel. Søg eksempelvis efter:

- Brug af en privat constructor
- Brug af partial-kodeordet
- Teori relateret til *boxing* og *unboxing*.

Grundlæggende hukommelsesteori

Inden du begynder at kode og benytte dine egne klasser, er du nødt til at kende til grundlæggende hukommelsesteori. Ikke fordi du på dette niveau skal være superbevidst om performance, men fordi den mest klassiske begynderfejl i objektorienteret programmering er manglende forståelse for forskellen på værdibaserede og referencebaserede typer.

Som nævnt flere gange i bogen har du seks typer af vælge i mellem i C#:

- Klasse
- Struktur
- Post
- Enumeration
- Delegate
- Interface.

De tre første – klasser, poster og strukturer – bruger du som også nævnt til at skabe skabeloner for instanser, og disse instanser kan indeholde værdier, som placeres i felter. I frameworket har Microsoft også gjort brug af klasser og strukturer til de fleste af de skabeloner, der er til rådighed under System-namespacet.

Af de mere kendte strukturer kan blandt andet nævnes System.Int32 (int) og System.DateTime. Af kendte klasser kan nævnes System.String (string), System.Random, System.Array og mange andre.

Du har frit valg og kan vælge at kode en Terning som en klasse:

```
internal class Terning
{
    public int Værdi;
```

```
public void Ryst()
{
    Random rnd = new Random();
    this.Værdi = rnd.Next(1, 7);
}
}
```

som kan benyttes således:

```
Terning t = new Terning();
t.Ryst();
Console.WriteLine(t.Værdi);
```

Du kan også vælge at benytte en struktur:

```
internal struct Terning
{

    public int Værdi;

    public void Ryst()
    {
        Random rnd = new Random();
        this.Værdi = rnd.Next(1, 7);
    }
}
```

som benyttes på præcis samme måde:

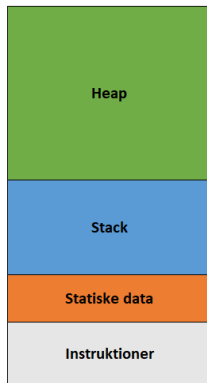
```
Terning t = new Terning();
t.Ryst();
Console.WriteLine(t.Værdi);
```

Du kan ikke se forskel i brugen af typen, men på typeniveau er der forskel.

Eksempelvis kan en struktur ikke indgå i et arvehierarki. Men den helt store forskel skal findes i, hvordan værdier opbevares i hukommelsen – og det er du nødt til at forstå, før du bliver god til objektorienteret programmering.

Diagram over hukommelsen

Når en applikation starter, bliver den tildelt et område af hukommelsen til at opbevare instruktioner og midlertidige data:



Figur 57 Hukommelse tildelt til instruktioner og data

Instruktioner (den kompilerede applikation) og statiske data kan du se bort fra lige nu, men *stack* og *heap* er vigtige begreber i mange programmeringssprog.

I virkeligheden er brugen af *stack* og *heap* meget kompleks og en del af det, man lærer i teori relateret til udvikling af kompilere, men i grundlæggende C# behøver du kun forstå det overordnet.

Stack

Helt overordnet og konceptuelt består en *stack* af et område i hukommelsen, hvor de variabler, du har defineret i en applikation, er placeret. Området er igen opdelt i mindre enheder kaldet en *stack-frame*, og hver *stack-frame* er relateret til en metode, der bliver kaldt, når programmet eksekveres. Du har tidligere lært om virkefelter, og som du sikkert kan huske, så har en metode (eller andre medlemmer) sit eget virkefelt med helt isolerede variabler. Disse variabler er kun

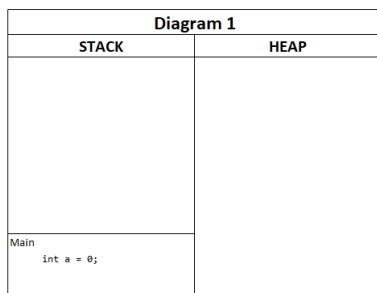
tilgængelige i denne metode. Hvis de skal benyttes i andre metoder, må de sendes med som argumenter.

Hvis du arbejder med en konsol-applikation, vil applikationen starte i Main-metoden, som runtime vil sørge for at afvikle. Applikationens stack har derfor en enkelt stack-frame, som vi kan relatere til Main-metoden. I denne stack-frame kan der angives de variabler, der er defineret og tildelt værdier eller referencer.

Forestil dig at du afvikler følgende applikation, og at du stopper afvikling ved diagram-kommentaren:

```
internal class Program
{
    private static void Main(string[] args)
    {
        int a = 0;
        // Diagram 1
    }
}
```

Så vil du kunne skabe et diagram som følger:

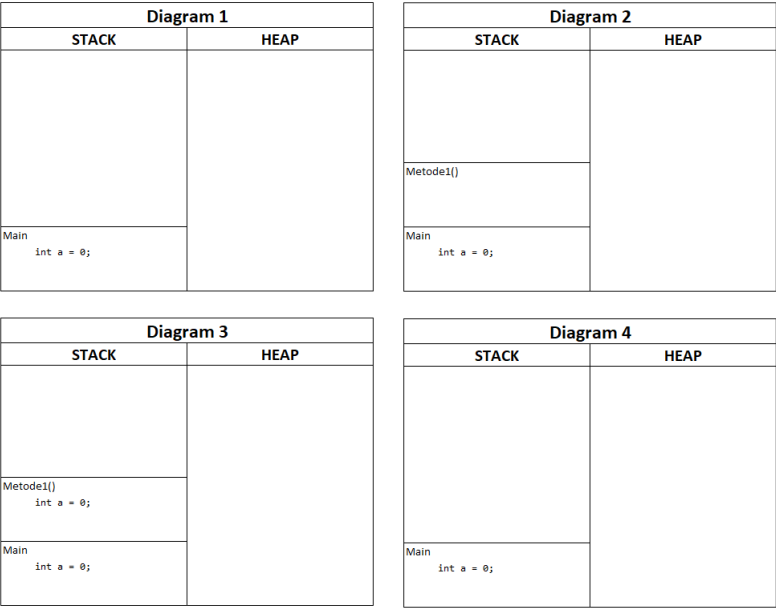


Navnet "stack" kommer fra det faktum, at en metode kan kalde en anden metode, som også har sin egen stack-frame, og denne (konceptuelt) lægges oven på den forrige. Se følgende eksempel:

```
internal class Program
{
    private static void Main(string[] args)
```

```
{
    int a = 0;
    // Diagram 1
    Metode1();
    // Diagram 4
}
private static void Metode1()
{
    // Diagram 2
    int a = 0;
    // Diagram 3
}
}
```

Diagrammerne ser således ud:

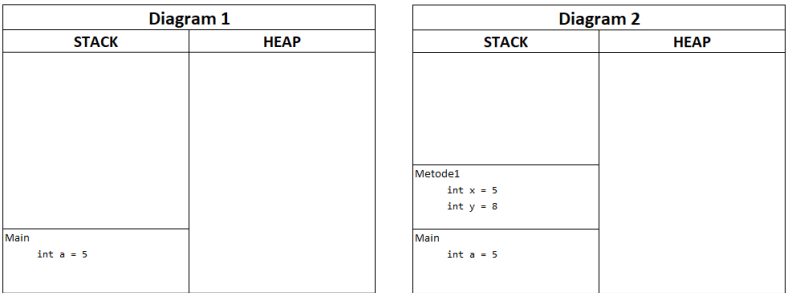


Som det fremgår, er hver metode indkapslet i sin egen lille sandkasse, og alt hvad der benyttes af variabler, lever kun her. Såfremt Metode1 kalder en anden metode, vil der blot dukke en ny stack-frame op som afvikles og forsvinder igen.

Hvis en metode har argumenter, kan du se dem som variabler i selve metoden. Værdierne fra den kaldende metode kopieres ind i den kaldte metode:

```
internal class Program
{
    private static void Main(string[] args)
    {
        int a = 5;
        // Diagram 1
        Metode1(a);
    }
    private static void Metode1(int x)
    {
        int y = 8;
        // Diagram 2
    }
}
```

Koden resulterer i følgende:



Bemærk, at værdien i a kopieres ind i den kaldende metode og lever sit helt eget liv i sin helt egen lille verden. Når Metode1 er afviklet, forsvinder x og y og de andre variabler i Metode1. Og blot for en god ordens skyld – variablerne x og y kunne lige så godt være kaldt a og b. Det har ingen betydning for den kaldende metode (Main).

Værdibaserede og referencebaserede typer

Som du tidligere har lært, så er en struktur (struct) en type, hvor variabler indeholder værdier, medens en klasse (class) er en type, hvor variabler indeholder reference til et sted i hukommelsen.

En struktur er en værdibaseret type, mens en klasse er en referencebaseret type!

Overordnet betyder det også, at variabler af strukturer opbevarer sine værdier på det område i hukommelsen, der kaldes en stack – ligesom de forrige diagrammer viser. Variabler af klasser er også placeret på den førnævnte stack, men indeholder reference til instanser, der er placeret et andet sted i hukommelsen kaldet en heap. Her vil runtime allokere (tildele) plads til de instanser, der ønskes, og sørge for at rydde op igen når instanserne ikke længere bliver brugt. Oprydningen sker med en såkaldt *garbage collector*.

De diagrammer, du har set indtil nu, har udelukkende bestået af Int32 variabler, og da Int32 er en struktur, opbevares værdier direkte på den førnævnte stack. Men som du så i starten af kapitlet, kan du skabe skabeloner af både strukturer og klasser. Enten:

```
struct Terning1 {  
    public int Værdi;  
}
```

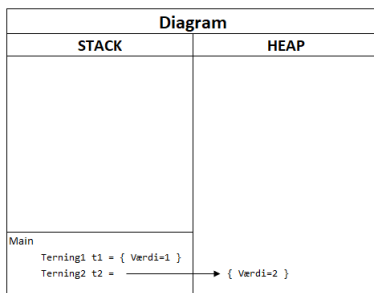
eller:

```
class Terning2 {  
    public int Værdi;  
}
```

Forskellen på brugen af de to typer kommer rigtig til syne, når du tegner diagrammer over, hvad der sker, når du skaber instanser. Med udgangspunkt i Terning1 og Terning2 kan der oprettes instanser som følger:

```
Terning1 t1 = new Terning1() { Værdi = 1 };
Terning2 t2 = new Terning2() { Værdi = 2 };
Console.WriteLine(t1.Værdi);    // 1
Console.WriteLine(t2.Værdi);    // 2
```

Et diagram, der viser, hvordan hukommelsen ser ud, når instanser er oprettet, ser således ud:



Som det fremgår, indeholder t1 en *værdi* og t2 en *reference*, og det kan give en stor forskel i brugen af variablerne.

Lad os se på det klassiske C# eksamensspørgsmål, der tager udgangspunkt i følgende kode:

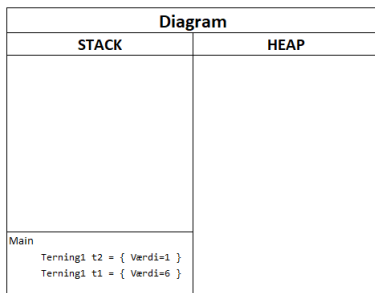
```
using System;
namespace Demo
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            Terning1 t1 = new Terning1() { Værdi = 1 };
            Terning1 t2 = t1;
            t1.Værdi = 6;
            // Hvad er værdien af t1.Værdi og t2.Værdi
        }
    }

    struct Terning1
    {
        public int Værdi;
    }
}
```

```
}  
  
class Terning2  
{  
    public int Værdi;  
}  
}
```

Hvad tror du, svaret er på spørgsmålet stillet som en kommentar i koden? Og inden du svarer – husk at t1 og t2 er af typen Terning1, som er en struktur.

Du bør skrive koden selv og prøve det af – men et diagram afslører tydeligt svaret:



Værdier fra t1 er kopieret over i t2, og når der efterfølgende rettes i t1, har det ikke nogen konsekvens på t2. Så svaret er, at t1.Værdi = 6 og t2.Værdi = 1.

Lad os så se på næste eksamensspørgsmål der tager udgangspunkt i følgende kode:

```
using System;  
namespace Demo  
{  
    internal class Program  
    {  
        private static void Main(string[] args)  
        {  
            Terning2 t1 = new Terning2() { Værdi = 1 };  
            Terning2 t2 = t1;  
        }  
    }  
}
```

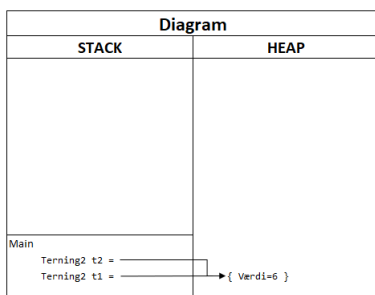
```
        t1.Værdi = 6;
        // Hvad er værdien af t1.Værdi og t2.Værdi
    }
}

struct Terning1
{
    public int Værdi;
}

class Terning2
{
    public int Værdi;
}
}
```

Hvad tror du, svaret er på spørgsmålet stillet som en kommentar i koden? Og igen, inden du svarer – husk at t1 og t2 er af typen Terning2, som er en klasse.

Diagrammet afslører svaret meget tydeligt:



Variablerne t1 og t2 indeholder referencer til et sted i hukommelsen, så instruktionen t2 = t1 kopierer referencen fra t1 til t2. Da både t1 og t2 dermed peger på den samme instans i hukommelsen, er svaret, at t1.Værdi er lig med 6 og t2.Værdi også er lig med 6.

Det er vigtigt, du forstår denne helt basale forskel på variabler af strukturer og variabler af klasser, så du bør prøve ovennævnte kode af selv og tegne et par diagrammer.

Argumenter til metoder

Viden om forskellen på værdibaserede og referencebaserede typer vil også komme dig til gode, når du skal kalde metoder med argumenter.

Her er endnu et klassisk eksamensspørgsmål, der tager udgangspunkt i følgende kode:

```
namespace Demo
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            Terning1 t1 = new Terning1() { Værdi = 1 };
            Terning2 t2 = new Terning2() { Værdi = 1 };
            Metode1(t1, t2);
            // Hvad er værdien af t1.Værdi og t2.Værdi
        }

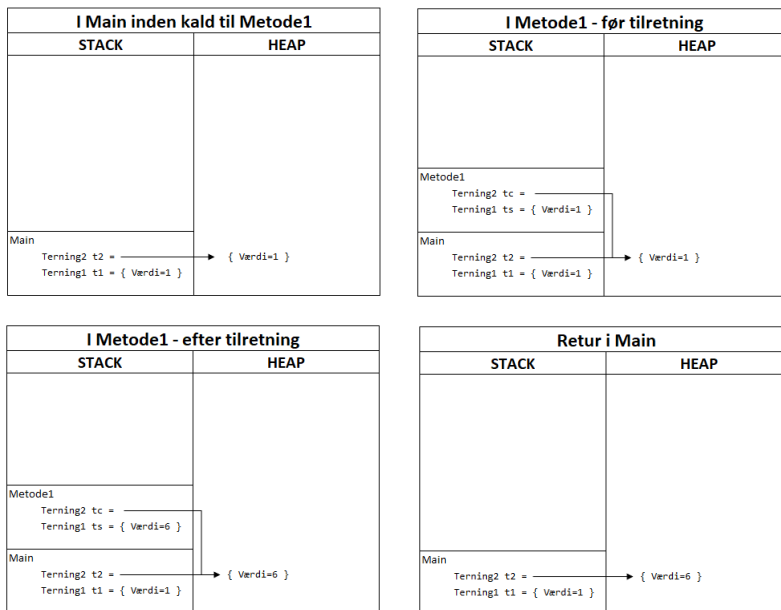
        private static void Metode1(Terning1 ts, Terning2 tc) {
            ts.Værdi = 6;
            tc.Værdi = 6;
        }
    }

    struct Terning1
    {
        public int Værdi;
    }

    class Terning2
    {
        public int Værdi;
    }
}
```

Der bliver skabt en instans af Terning1 (struktur) og en instans af Terning2 (klasse), og variablerne benyttes som argumenter i en metode, hvor værdien sættes til 6. Hvad er værdien af terning1.Værdi

og terning2.Værdi efter kaldet til metoden? Prøv at tegne diagrammet selv – det vil afsløre svaret:



Efter kaldet til Metode1 vil t1.Værdi have værdien 1 og t2.Værdi have værdien 6. Årsagen skal findes i forskellen på typerne – ved kaldet til Metode1 bliver *værdien* af t1 og *referencen* til t2 kopieret ind i metoden. Derfor vil en tilretning af t1 ikke have nogen konsekvens.

Det er vigtigt, at du kan svare på de tre klassiske eksamensspørgsmål, så skriv gerne koden selv og brug debuggeren – og tegn nogle diagrammer. Det er ikke svært, så længe du husker at:

struktur = værdi
klasse = reference



Du kan finde flere videoer på bogenomcsharp.dk, der viser grundlæggende hukommelsesteori relateret til C#. Klik på "Video" i menuen.

Hvis du vil vide mere

For en god ordens skyld skal det nævnes, at stack og heap er ret simplificeret i dette kapitel. I den virkelige verden, er det noget mere komplekst, men hvis du kan tegne de konceptuelle diagrammer i hovedet, når du koder, er du meget langt.

Der er masser af information på nettet omkring hukommelsesteori, og når du er blevet lidt mere øvet, bør du læse mere om emnet.

Indkapsling

Indkapsling er et vigtigt begreb i objektorienteret programmering. Det giver mulighed for at skjule både data (felter) og funktionalitet (metoder) for dem, der benytter typerne til at skabe objekter.

I C# er indkapsling så simpelt, at det blot handler om at skifte synlighed på medlemmer. Se følgende klasse:

```
internal class Terning
{
    public int Værdi;

    public void Ryst()
    {
        // kode
    }
}
```

Her kan både feltet og metoden ses og benyttes udefra fordi medlemmer er offentlige (public):

```
Terning t1 = new Terning();
t1.Værdi = 1;
t1.Ryst();
```

Men hvis medlemmernes synlighed rettes til privat (private):

```
internal class Terning
{
    private int Værdi;

    private void Ryst()
    {
        // kode
    }
}
```

kan hverken feltet Værdi eller metoden Ryst ses udefra, og et eventuelt forsøg på tilgang vil resultere i en kompileringsfejl. Private medlemmer kan udelukkende tilgås fra medlemmer i typen selv.

Umiddelbart kan du måske ikke se fordelene ved at skjule medlemmer, men du skal forstå det således, at du som udvikler af typen bestemmer, hvordan medlemmer skal tilgås ved at skabe en offentlig grænseflade. Hvad der så sker bag facaden er underordnet – det behøver brugeren af typen ikke at forstå.

Indkapsling af felter

Indkapsling bruges især til at styre tilgangen til felter, så du skriver kode relateret til validering, sikkerhed, log og så videre. I mange programmeringssprog benyttes såkaldte get- og set-metoder til at skabe tilgang til et privat felt. Get-metoden aflæser og Set-metoden tildeler.

Du vil typisk lade C# kompilatoren autogenerer get- og set-metoderne, men det er vigtigt, du lige har set og forstået teknikken. Hvis du skulle kode det manuelt, kunne det se således ud:

```
public class Terning
{
    private int værdi;

    public void SetVærdi(int value)
    {
        this.værdi = value;
    }

    public int GetVærdi()
    {
        return this.værdi;
    }
}
```

Bemærk, at feltet nu er stavet med lille v. Det har ingen praktisk betydning for kompilatoren, men C# udviklere er vant til denne navn-

givningsstandard. Nu er det private felt beskyttet (privat) og tilgang kan udelukkende ske gennem metoderne:

```
Terning t = new Terning();  
t.SetVærdi(1);  
Console.WriteLine(t.GetVærdi());    // 1
```

I metoderne kan du naturligvis placere de instruktioner, du ønsker afviklet, når der tildeles og aflæses – nu er du som udvikler af terningen i kontrol.

Her er eksempelvis koden, der sikrer, at terningen ikke kan få en forkert værdi, og at man kun kan benytte terningen i en weekend:

```
public class Terning  
{  
    private int værdi;  
  
    public void SetVærdi(int value)  
    {  
        if (!this.ErWeekend())  
            throw new Exception("Terning må kun bruges i weekenden");  
        if (value < 1 || value > 6)  
            throw new Exception("Forkert værdi");  
        this.værdi = value;  
    }  
  
    public int GetVærdi()  
    {  
        if (!this.ErWeekend())  
            throw new Exception("Terning må kun bruges i weekenden");  
        return this.værdi;  
    }  
  
    private bool ErWeekend()  
    {  
        DayOfWeek dag = DateTime.Now.DayOfWeek;  
        switch (dag)  
        {  
            case DayOfWeek.Sunday:
```

```
        case DayOfWeek.Saturday:
            return true;
        default:
            return false;
    }
}
```

Bemærk, at metoden `ErWeekend` også er privat og kun lever i klassen `Terning`. Den er dermed indkapslet og kan ikke tilgås udefra. Du kan også skabe private felter, som ikke har en `get/set` metode, og dermed holdes helt internt i klassen.

Her er en lidt mere komplet terning:

```
public class Terning
{
    private int værdi;
    private System.Random rnd = new System.Random();
    private string[] fejlTekst = {
        "Forkert værdi",
        "Terning må kun bruges i weekenden"
    };

    public void SetVærdi(int value)
    {
        if (!this.ErWeekend())
            throw new Exception(this.fejlTekst[1]);
        if (value < 1 || value > 6)
            throw new Exception(this.fejlTekst[0]);
        this.værdi = value;
    }

    public int GetVærdi()
    {
        if (!this.ErWeekend())
            throw new Exception(this.fejlTekst[1]);
        return this.værdi;
    }
}
```

```
public void Ryst() {
    this.værdi = this.rnd.Next(1, 7);
}

private bool ErWeekend()
{
    DayOfWeek dag = DateTime.Now.DayOfWeek;
    switch (dag)
    {
        case DayOfWeek.Sunday:
        case DayOfWeek.Saturday:
            return true;
        default:
            return false;
    }
}
}
```

Nu er både fejltekster (gentaget kode er roden til alt ondt) og felter, der refererer til en instans af System.Random, holdt privat.

De offentlige medlemmer er de medlemmer, som *brugeren af Terning* har adgang til. De offentlige og private medlemmer er de medlemmer, som *udvikleren af Terning* har adgang til.

Egenskaber

Du kan vælge, om du vil benytte get- og set-metoder til at skabe tilgang til et privat felt, men C# stiller en anden medlemstype til rådighed, som du bør benytte i stedet. Det kaldes en egenskab (eller på engelsk - property).

Det giver nogle fordele frem for at kode dine egne get/set metoder. For det første er det nemmere og hurtigere at skrive koden i klassen, for det andet er syntaksen bedre for dem, der benytter klassen, og sluttelig er det ikke *bare* almindelige metoder, men en speciel medlemstype. Det giver en del muligheder for at benytte klassen til eksempelvis automatisk databinding i brugerflade runtimes (Windows Forms, Windows Presentation Foundation eller Xamarin Mobile) eller

automatisk kodegenerering i Visual Studio (ASP.NET MVC bruger det eksempelvis meget).

Så du bør benytte egenskaber i stedet for at skabe dine egne get- og set-metoder – for det er jo bare metoder lige som alt andet. I virkeligheden bliver get- og set-metoder autogenereret af kompilatoren bag om ryggen på os, men det er en helt anden historie.

Komplette egenskaber

Hvis du først har forstået årsagen til brugen af get- og set-metoder, er syntaksen bag egenskaber meget simpel. Den ser således ud:

```
[datatype] [navn]
{
    get {
        return [værdi]
    }
    set {
        // value er implicit
    }
}
```

Som du kan se, går get og set notationen igen fra almindelige metoder, men er noget hurtigere at kode. Hvis klassen terning skal benytte en egenskab til værdi, ser det således ud:

```
public class Terning
{
    private int værdi;

    public int Værdi
    {
        get { return værdi; }
        set { værdi = value; }
    }
}
```

Du kan tilføje en egenskab med en snippet i Visual Studio
og i Visual Studio Code.
Brug "propfull" og to gange tabulering.

Bemærk, at i set-delen af egenskaben kan variablen value benyttes. Den bliver automatisk tildelt en værdi af runtime og behøver dermed ikke erklæres.

Den nye egenskab kan benyttes som følger:

```
Terning t = new Terning();  
t.Værdi = 4;  
Console.WriteLine(t.Værdi);
```

Bemærk, at syntaksen ved brug nu er meget pænere – der er ikke længere tale om metodekald. Når der bliver tildelt en værdi, bliver set-delen kaldt, og når der bliver aflæst, bliver get-delen kaldt.

Ligesom ved almindelige metoder kan man naturligvis tilføje den kode, man ønsker relateret til validering, sikkerhed eller lignende:

```
public class Terning  
{  
    private int værdi;  
  
    public int Værdi  
    {  
        get  
        {  
            if (!this.ErWeekend())  
                throw new  
                    Exception("Terning må kun bruges i weekenden");  
  
            return this.værdi;  
        }  
        set  
        {  
            if (value < 1 || value > 6)
```

```
        throw new Exception("Forkert værdi");

        this.værdi = value;
    }
}

private bool ErWeekend()
{
    DayOfWeek dag = DateTime.Now.DayOfWeek;
    switch (dag)
    {
        case DayOfWeek.Sunday:
        case DayOfWeek.Saturday:
            return true;
        default:
            return false;
    }
}
}
```

Du kan vælge, hvilken synlighed get- eller set-blokken skal have – eller simpelthen fjerne en af de to for at skabe en ren read-only eller write-only egenskab.

Her er et eksempel på en egenskab med en offentlig get (som er standard) og en privat get. Det betyder, at set-blokken kun kan tildeles internt i klassen, og dermed er der ikke nogen grund til at tildele en værdi direkte til feltet.

```
public class Terning
{
    private int værdi;

    public int Værdi
    {
        get
        {
            return this.værdi;
        }
    }
}
```



```
        private set
        {
            this.værdi = value;
        }
    }

    public Terning()
    {
        this.Ryst();
    }

    public void Ryst()
    {
        System.Random rnd = new Random();
        this.Værdi = rnd.Next(1, 7);
    }
}
```

Samme funktionalitet kunne opnås ved at fjerne set-blokken helt, og dermed skabe en read-only egenskab, men det er *best practice* altid at gå gennem en egenskab. Så kan eventuel kommende kode samles et sted.

Initialisering

Som alternativ til initialisering af data gennem en konstruktør kan man også i C# initialisere egenskaber ved oprettelse ved brug af tuborg-klammer.

Se følgende klasse, der for eksemplets skyld blot består af et felt og en tilhørende egenskab:

```
class Terning
{
    private int værdi;

    public int Værdi
    {
        get { return værdi; }
        set

```

```
{
    if (value < 1 || value > 6)
        throw new Exception("Forkert værdi");
    værdi = value;
}
}
```

Da klassen ikke indeholder en brugerdefineret konstruktør, kan værdien tildeles efter oprettelse som følger:

```
Terning t = new Terning();
t.Værdi = 6;
```

Men der findes en alternativ syntaks som er lidt hurtigere at skrive:

```
Terning t = new Terning { Værdi = 6 };
```

Bemærk, at det ikke længere er nødvendigt med parenteser, og at kompileren godt kan se, at værdien tilhørende instansen t. Hvis der er flere egenskaber, kan navn og værdi blot adskilles med komma.

Da kompileren jo er ligeglad med linjeskift vil du også se denne måde at initialisere objekter:

```
Terning t = new Terning
{
    Værdi = 6
};
```

Det kan gøre det lidt nemmere at læse, hvis der er mange egenskaber.

Init-egenskab

Som alternativ til en set-egenskab kan du også vælge at benytte en init-egenskab:

```
class Terning
{
    private int værdi;

    public int Værdi
```

```
{
    get { return værdi; }
    init
    {
        if (value < 1 || value > 6)
            throw new Exception("Forkert værdi");
        værdi = value;
    }
}
```

Det har den fordel at egenskaben nu kun kan tildeles en værdi ved initialisering med tuborgklammer. Efterfølgende kan egenskaben ikke tildeles en værdi:

```
Terning t = new Terning
{
    Værdi = 6
};
// t.Værdi = 1;    // fejl
```

Bemærk, at init-egenskaber er en C# 9 feature!

Denne init-del kan ses som alternativ til en klasse med en konstruktør og en read-only egenskab:

```
class Terning
{
    private int værdi;

    public int Værdi
    {
        get { return værdi; }
    }

    public Terning(int værdi)
    {
        if (værdi < 1 || værdi > 6)
```

```
        throw new Exception("Forkert værdi");
        this.værdi = værdi;
    }
}
```

Nu skal terningen tildeles en værdi i konstruktøren:

```
Terning t = new Terning(6);
// t.Værdi = 1;    // fejl
```

Både denne metode, samt brug af init-kodeordet, giver mulighed for at skabe immutable objekter som, når de først er tildelt en værdi, ikke efterfølgende kan ændres.

Automatiske egenskaber

Der er masser af situationer, hvor du ikke ønsker at tilføje kode, der afvikles ved get- eller set-blokken, og du kan med rette spørge, hvorfor du så ikke bare kan tilføje et offentligt felt. Men for det første bør alle felter være private, og for det andet kan det være, at du i en senere version af applikationer ønsker at tilføje eventuelt validerings- eller sikkerhedskode. Derfor bør du altid benytte en egenskab – også selv den som udgangspunkt er helt tom og blot fører værdier til og fra et felt.

Det kunne kodes således jævnfør forrige afsnit:

```
public class Terning
{
    private int værdi;

    public int Værdi
    {
        get
        {
            return this.værdi;
        }

        set
        {
```

```
        this.værdi = value;
    }
}
```

Men hvis det blot er den funktionalitet du ønsker, kan du få kompileren til at autogenerere hele strukturen for dig. Det hedder en automatisk egenskab og kan kodes således:

```
public class Terning
{
    public int Værdi { get; set; }
}
```

Bemærk at tuborgklammer mangler og det er dermed ikke muligt at tilføje kode til get- eller set-blokken. Til gengæld er den superhurtig at kode, og du overholder best practice ved at benytte en egenskab i stedet for et felt.

Kompileren vil sørge for at autogenerere både et private felt samt get- og set-metoder, men du kan altid selv på et senere tidspunkt erstatte den automatiske egenskab med en komplet egenskab. Så længe navn og synlighed er ens, vil du ikke ændre på interfacet til klassen.

Du kan stadig rette synligheden på en automatisk egenskab – eksempelvis kan egenskaben Værdi have en offentlig get-blok og en privat set-blok:

```
public class Terning
{
    public int Værdi { get; private set; }

    public Terning()
    {
        this.Ryst();
    }

    public void Ryst()
    {
        System.Random rnd = new Random();
    }
}
```

```
        this.Værdi = rnd.Next(1, 7);  
    }  
}
```

Nu kan egenskaben Værdi udelukkende aflæses og ikke tildeles en værdi udefra.

Du kan tilføje en automatisk egenskab med en snippet i Visual Studio – brug ”prop” og to gange tabulering.

Du kan også vælge at benytte init-kodeordet i stedet for set-kodeordet, og som tidligere vist (se side 222) har det den konsekvens, at data kun kan tildeles ved initialisering:

```
class Terning  
{  
    public int Værdi { get; init; }  
}
```

Nu skal du tildele en værdi ved oprettelse:

```
Terning t = new Terning { Værdi = 6 };  
// t.Værdi = 1;    // fejl
```

Det giver især mening i immutable typer.

Hvis du vil vide mere

Senere i bogen vil vi se nærmere på lambdaudtryk, og i moderne C# vil du måske se en egenskab kodet som:

```
public string Navn  
{  
    get => navn;  
    set => navn = value;  
}
```

Koden vil give langt mere mening senere, men måske bør du på et tidspunkt læse lidt om brugen af *Expression-bodied members*.

Arv

Et af de mest omtalte begreber i objektorienteret programmering er arv, hvilket dækker over muligheden for at genbruge og dermed vedligeholde kode i et hierarki af klasser.

Det er supersmart og i praksis ret nemt at implementere, men du skal være opmærksom på, at du *ikke behøver* benytte arv i din kode. Det er et værktøj i værktøjsskassen, du kan vælge at trække frem, og nogle applikationer egner sig bedre til at benytte arv end andre. Microsoft har i høj grad benyttet sig af arv ved design af frameworket, men du vælger, i hvilken grad du vil benytte det i din applikation. Det er helt op til dig.

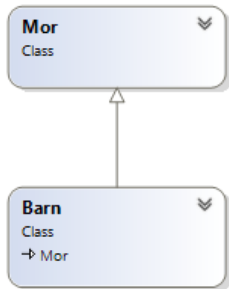
Hierarki af klasser

Ved hjælp af arv har du mulighed for at skabe en (og i C# kun én) klasse, som betragtes som mor for andre. Det kaldes også en superklasse. Alle medlemmer (felter, egenskaber, metoder og hændelser) med den korrekte synlighed i denne klasse er tilgængelige i alle underklasser, og det vigtigste budskab i den forbindelse er, at medlemmernes signatur ikke kan ændres i underklasser.

En signatur på en metode i et arvehierarki svarer til metodens navn, returværdi og argumenter.
--

Hvis eksempelvis en metode er gjort tilgængelig i underklasser, vil den ikke kunne fjernes i hele hierarkiet. Den kan muligvis blive tildelt en anden implementation (andre instruktioner) – men metoden med den samme signatur skal være der.

Det mest simple eksempel på et arvehierarki er følgende:



Figur 58 Et simpelt arvehierarki

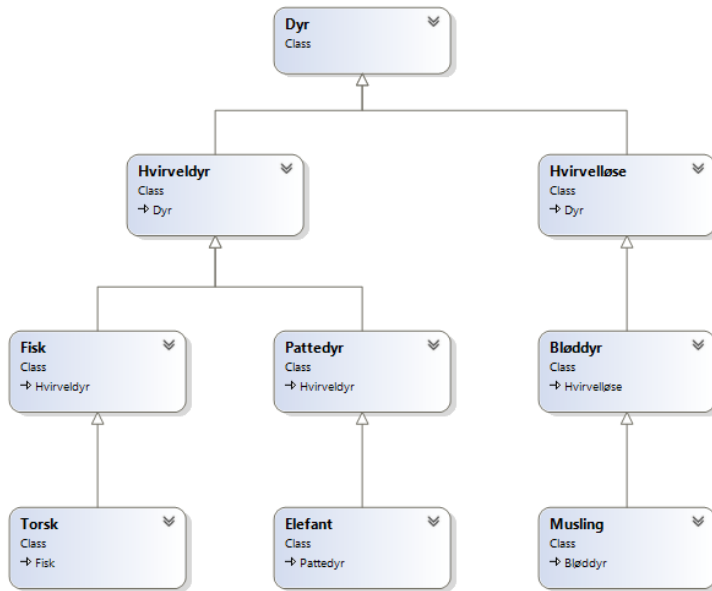
Det kan du læse som klassen Barn, der arver fra klassen Mor, og alt hvad der findes i mor findes også i barnet. Yderligere kan alle offentlige medlemmer i mor også tilgås i barnet. Derfor kan man altid i arvehierarkier sige, at et barn *er* en mor med eventuelle tilføjelser.

Hvis klassen Mor har en offentlig metode Skriv, så har barnet det også. Hvordan metoden er implementeret i barnet, kan være anderledes end i mor – men metoden findes og kan ikke fjernes på nogen måde.

Syntaksen for arv i C# er kolon, så ovennævnte hierarki kan skrives som:

```
class Mor { }
class Barn : Mor { }
```

Du kan også opfatte et hierarki som en definering af generelle offentlige medlemmer, som i underklasser bliver mere og mere specifikke. Her er et andet og lidt mere komplekst eksempel, der beskriver dette:



Figur 59 Et arvehierarki med dyr

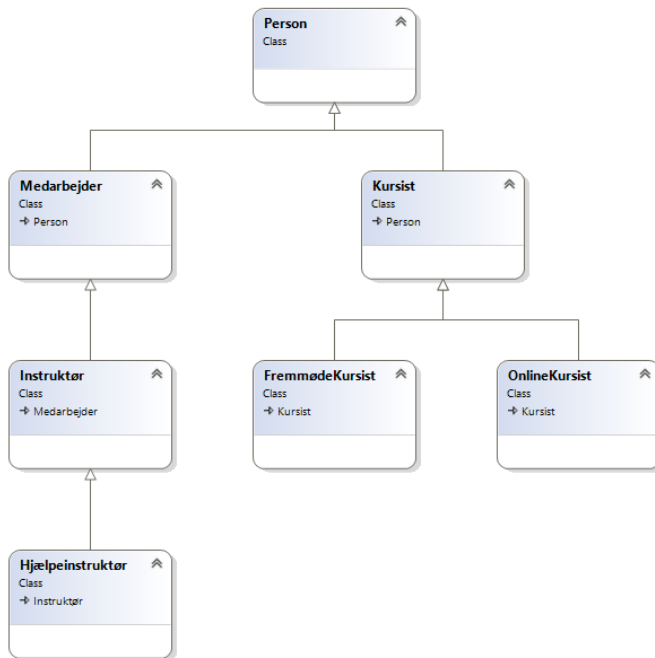
I toppen af hierarkiet findes klassen Dyr, og jo længere du kommer ned i hierarkiet, bliver klasserne mere og mere specifikke, men du (og runtime) kan være helt sikker på, at medlemmer, der findes i Dyr, også findes i alle underklasser. Du vil ligeledes altid kunne antage, at en underklasse *er* en overklasse. Eksempelvis er en torsk en fisk, som er et hvirveldyr, som et er dyr, og det samme gælder en musling, som er et bløddyr, som er et hvirvelløst dyr, som er et dyr.

I kode ser hierarkiet således ud:

```
class Dyr { }
class Hvirveldyr : Dyr { }
class Hvirvelløse : Dyr { }
class Fisk : Hvirveldyr { }
class Torsk : Fisk { }
class Pattedyr : Hvirveldyr { }
class Elefant : Pattedyr { }
class Bløddyr : Hvirvelløse { }
class Musling : Bløddyr { }
```

Sluttelig er her et mere applikationsorienteret hierarki.

Forestil dig, at du skal udvikle en applikation relateret til kurser. I så fald kunne følgende måske gøre koden mere genbrugelig:



Figur 60 Et klassehierarki relateret til kurser

Alle klasser arver fra Person og består dermed af alle de medlemmer, som en person måtte have (måske egenskaberne navn og cpr-nummer og tilhørende felter, samt metoderne skriv og gem), men klasserne bliver mere og mere specifikke.

Igen vil du altid kunne antage, at både en hjælpeinstruktør (fra venstre gren) og en kursist (fra højre gren) er en person.

Her er koden bag hierarkiet:

```
class Person { }
```

```
class Medarbejder : Person { }  
class Kursist : Person { }  
class Instruktør : Medarbejder { }  
class FremmødeKursist : Kursist { }  
class OnlineKursist : Kursist { }  
class HjælpeInstruktør : Instruktør { }
```

Rækkefølgen eller placering af klasserne har ingen betydning. Nogle udviklere vil gerne have klasser i et hierarki i samme fil, men de fleste kan bedst lide, at hver fil indeholder én klasse. Men det er helt op til dig.

I C# må en klasse kun have én *mor*, det kaldes som begreb *single inheritance*. I andre sprog er det muligt, at en klasse kan have flere mødre. Det kaldes *multiple inheritance*.

Brug af arv

I et arvehierarki vil medlemmer med offentlige (eller som vi skal se senere – beskyttede) medlemmer i mor altså være tilgængelig i alle børn. Du skal huske på, at et *barn* er en *mor*, og kompileren skal nok sørge for, at medlemmer er tilgængelige.

Se følgende eksempel:

```
internal class Person {  
    public string Navn { get; set; }  
    public void Skriv() {  
        Console.WriteLine($"Mit navn er {this.Navn}");  
    }  
}
```

Det er en almindelig klasse med en egenskab og en metode som tidligere gennemgået, og den kan benyttes som følger:

```
Person p = new Person();  
p.Navn = "Mathias";  
p.Skriv();    // Mit navn er Mathias
```

Hvis en klasse Elev arver fra Person ser det således ud:

```
internal class Person {
    public string Navn { get; set; }
    public void Skriv() {
        Console.WriteLine($"Mit navn er {this.Navn}");
    }
}
internal class Elev : Person { }
```

Da en Elev *er* en Person er følgende muligt:

```
Elev e = new Elev();
e.Navn = "Mathias";
e.Skriv();
```

Alle medlemmer er helt automatisk en del af Elev, og så har du jo ikke opnået særlig meget. Men du kan udvide klassen Elev med de medlemmer, du har lyst til - eksempelvis:

```
internal class Person
{
    public string Navn { get; set; }
    public void Skriv()
    {
        Console.WriteLine($"Mit navn er {this.Navn}");
    }
}

internal class Elev : Person
{
    public int ElevNummer { get; set; }
    public void Gem()
    {
        Console.WriteLine($"Gemmer {this.Navn}");
    }
}
```

Og nu kan Elev bruges således:

```
Elev e = new Elev();
```

```
e.Navn = "Mathias";  
e.ElevNummer = 1;  
e.Skriv(); // Mit navn er Mathias  
e.Gem(); // Gemmer Mathias
```

Nu har Elev både medlemmer fra mor (Person) og sine egne medlemmer, og en eventuel tilretning af Person vil også slå igennem i Elev. I et hierarki med få typer har det måske ikke så stor betydning, men i et dybt eller bredt hierarki med mange typer, kan det betyde en meget stor grad af genbrug.

Tilgang til medlemmer

Du har tidligere lært om synlighed i typer. Et medlem kan være enten offentlig (public) eller privat (private):

```
public class A {  
    private void Test1() { }  
    public void Test2() { }  
}
```

Her kan metoden Test1 kun tilgås internt i klassen (eksempelvis fra Test2), og metoden Test2 kan tilgås både internt og eksternt:

```
A a = new A();  
a.Test1(); // Fejl - Test1 er ikke synlig udefra  
a.Test2(); // Ok
```

I forbindelse med arv er der en ny mulighed kaldet beskyttet (protected), og med den åbnes der op for, at medlemmer kan tilgås i underklasser, men ikke udefra. Du kan se det som privat inden for hierarkiet.

Prøv at se følgende:

```
internal class Person  
{  
    public string Navn { get; set; }  
    protected int Alder { get; set; }  
    private string CprNummer { get; set; }  
}
```

```
public void Test1()
{
    // Navn kan godt tilgås
    // Alder kan godt tilgås
    // CprNummer kan godt tilgås
}
}
internal class Elev : Person
{
    public void Test2()
    {
        // Navn kan godt tilgås
        // Alder kan godt tilgås
        // CprNummer kan ikke tilgås
    }
}
```

Klassen Person har tre egenskaber – en privat, en beskyttet og en offentlig. I underklasser (her Elev) kan både Navn og Alder tilgås, men CprNummer kan ikke. Det betyder ikke, at en Elev ikke har et CprNummer – for en Elev er jo en Person – men det betyder, at egenskaben ikke kan tilgås direkte.

Udefra ser det således ud:

```
Person p = new Person();
// p.Navn kan godt tilgås
// p.Alder kan ikke tilgås
// p.CprNummer kan ikke tilgås

Elev e = new Elev();
// e.Navn kan godt tilgås
// e.Alder kan ikke tilgås
// e.CprNummer kan ikke tilgås
```

Den eneste egenskab, der kan tilgås, er Navn, for den er offentlig. Hverken den beskyttede eller den private kan tilgås udefra.

Så synligheden kan beskrives således:

Synlighed	Beskrivelse
<i>public (offentligt)</i>	<i>Medlemmer kan både tilgås internt og eksternt</i>
<i>protected (beskyttet)</i>	<i>Medlemmer kan kun tilgås internt i alle klasser i et hierarki</i>
<i>private (privat)</i>	<i>Medlemmer kan kun tilgås i klassen selv</i>

Tabel 23 Synlighed i et arvehierarki

Konstruktører

Som før nævnt er en konstruktør en metode, der afvikles, når der skabes en ny instans, og den fungerer lidt specielt i et arvehierarki.

En standard konstruktør (uden argumenter) afvikles i rækkefølge fra mor til barn. Se følgende kode:

```
class A
{
    public A()
    {
        Console.WriteLine("I A()");
    }
}

class B : A
{
    public B()
    {
        Console.WriteLine("I B()");
    }
}
```

Hvis der skabes en instans af B, bliver konstruktør i A kørt først, og herefter i B:

```
B b = new B();
// I A()
// I B()
```

Lidt anderledes er det med en brugerdefineret (custom) konstruktør, fordi den ikke bliver nedarvet til børn – det skal du selv sørge for:

```
class Person
{
    protected string navn;

    public Person()
    {
        this.navn = "";
    }

    public Person(string navn)
    {
        this.navn = navn;
    }
}

class Elev : Person
{
}
```

I klassen Elev kan den brugerdefinerede konstruktør fra Person ikke findes:

```
// Dette er ok
Elev e1 = new Elev();

// Dette er ikke ok fordi konstruktør ikke findes
// Elev e2 = new Elev("Mikkel");
```

Men du må naturligvis godt skabe en brugerdefineret konstruktør i Elev:

```
class Elev : Person
{
    public Elev(string navn)
    {
        this.navn = navn;
    }
}
```

Nu kan den brugerdefinerede konstruktør benyttes:


```
Elev e2 = new Elev("Mikkel"); // Dette er ok
```

Brug af base-kodeordet

I et arvehierarki kan man hurtigt komme til at gentage kode i konstruktører, og det går jo imod hele princippet ved nedarvning. Eksempelvis er kode som dette uheldigt:

```
class Person
{
    protected string navn;

    public Person(string navn)
    {
        this.navn = navn;
    }
}

class Elev : Person
{
    public Elev(string navn)
    {
        this.navn = navn;
    }
}
```

Bemærk, at koden i konstruktørerne er ens, og det duer ikke. Der kunne jo også være kode, som validerer feltet.

Men det kan undgås ved at lade den ene konstruktør kalde mors konstruktør ved hjælp af base-kodeordet:

```
class Person
{
    protected string navn;

    public Person(string navn)
    {
        this.navn = navn;
    }
}
```

```
}  
  
class Elev : Person  
{  
    public Elev(string navn) : base(navn)  
    {  
        // Eventuel yderligere kode  
    }  
}
```

I Elev sendes argumentet navn videre til *mor* som bruges til at initialisere feltet navn. På den måde slipper man for gentaget kode.

En ludoterning

Det er tid til at finde din terning frem igen. Her er en simpel terning i kode:

```
class Terning {  
    public int Værdi { get; protected set; }  
    public void Ryst() {  
        System.Random rnd = new System.Random();  
        this.Værdi = rnd.Next(1, 7);  
    }  
    public Terning()  
    {  
        this.Ryst();  
    }  
}
```

Forestil dig nu, at du skal bruge en anden type terning til et spil – eksempelvis en Ludo-terning. Det er jo en terning, men med metoder, som ikke har noget med en almindelig terning at gøre. I Ludo har en 3'er og en 5'er jo en anden værdi. Men ved brug af nedarvning er det meget simpelt at skabe en Ludo-terning:

```
class LudoTerning : Terning  
{  
    public bool ErStjerne()  
    {
```

```
        return this.Værdi == 3;
    }

    public bool ErGlobus()
    {
        return this.Værdi == 5;
    }
}
```

Du kan nøjes med at tilføje de medlemmer, som gør terningen til en Ludo-terning – alle de andre medlemmer har den allerede. Nu kan du nemt spille Ludo:

```
LudoTerning l = new LudoTerning();
Console.WriteLine(l.Værdi);
Console.WriteLine(l.ErStjerne());
Console.WriteLine(l.ErGlobus());
```

Du kan selv prøve at kode en anden type terning – hvad med en pakkeleg-terning (her er en 6'er jo speciel)?

Hvis du vil vide mere

Når du bliver lidt mere øvet, kan du måske lede efter yderligere information relateret til synlighed. Prøv at søge efter brug af *protected internal* og *private protected*.

Polymorfi

Polymorfi er et fint ord for flerformethed (mange former), og er måske det begreb som begyndere kæmper mest med i objektorienteret programmering. Det benyttes blandt andet i arvehierarkier og er i virkeligheden ikke så komplekst, hvis det underbygges med nogle eksempler.

Selve ordet polymorfi passer meget fint til den funktionalitet, der stilles til rådighed, fordi det dækker over muligheden for, at runtime kan arbejde med et objekt på flere måder. Ved at benytte typer i et arvehierarki kan et objekt således have mange grænseflader og dermed forskellig funktionalitet. Rent praktisk betyder det, at en variabel kan indeholde en reference til mange forskellige objekter af typer i hierarki, og det er jo meget anderledes, end jeg tidligere har beskrevet, hvor en variabel af typen String, Random, Terning og eller andre kun kan indeholde en reference til objekter af denne type.

Polymorfi kommer grundlæggende i to former i C#:

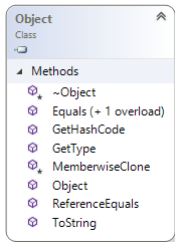
- Underklasser (børn) kan have deres egen implementation af metoder og egenskaber defineret i en overklasse (mor)
- En variabel defineret af en overklasse (mor) kan altid pege på objekter af underklasser (børn).

Begge former kigger vi på i dette kapitel.

System.Object

Du skal ikke særlig langt ned i frameworkets store klassehierarki, før du falder over de to former for polymorfi. Alle klasser arver nemlig helt automatisk fra klassen System.Object. Det gælder både andre klasser i frameworket samt dine egne.

System.Object har seks metoder, som dermed er tilgængelig i alle andre klasser, men i grundlæggende C# er det kun ToString-metoden, der er interessant.



Figur 61 System.Object.

ToString-metoden har til formål at returnere en streng, der repræsenterer objektet bedst muligt, men som udgangspunkt returnerer den blot navnet på klassen.

Som eksempel kan du kode en klasse Terning som følger:

```
class Terning { }
```

Af klassen er det muligt at oprette instanser og kalde ToString:

```
Terning t = new Terning();
Console.WriteLine(t.ToString());    // [Namespace].Terning

// Eller andre klasser
System.Random r = new Random();
Console.WriteLine(r.ToString());    // System.Random
```

Nogle klasser har dog *deres egen implementation* af ToString(), så metoden giver bedre mening:

```
string b = "x";
Console.WriteLine(b.ToString());    // x
```

Her returnerer ToString() ikke længere navnet på klassen, men værdien, og det indikerer jo, at metoden er ændret fra det oprindelige format i System.Object til koden i System.String. Man kalder dette en overskrivning (overwrite) af metoden, og det kræver at metoden, der ønskes overskrevet, er markeret med virtual-kodeordet og dermed *virtuel*.

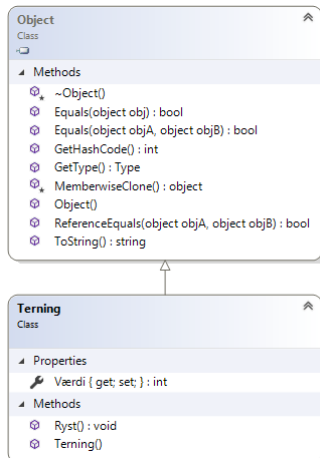
Den anden form for polymorfi (en mor kan altid pege på sine børn) er også meget tydelig, når man arbejder med `System.Object`. Forudsat at klassen `Terning` er defineret:

```
class Terning
{
    public int Værdi { get; private set; }
    public void Ryst()
    {
        System.Random rnd = new System.Random();
        this.Værdi = rnd.Next(1, 7);
    }
    public Terning()
    {
        this.Ryst();
    }
}
```

kan følgende kode kompilere uden problemer:

```
object o1 = new Terning();
```

Det giver jo umiddelbart ingen mening, før du ser arvehierarkiet:



Figur 62 Alt arver fra `System.Object`

Fordi Terning arver fra Object, har en terning alt, hvad Object har, og dermed kan en variabel af typen Object godt indeholde en reference til en Terning. Men variabelen kan ikke tilgå medlemmer andre end medlemmer defineret på Object:

```
object t1 = new Terning();
Terning t2 = new Terning();

// Dette er ok
Console.WriteLine(t1.ToString());
Console.WriteLine(t2.ToString());
Console.WriteLine(t2.Værdi);

// Dette kan ikke kompilere
// Console.WriteLine(t1.Værdi);
```

At skabe variabler af superklasser giver en del muligheder – især i kombination med den forrige form for polymorfi. Blandt andet kan denne samling indeholde hvad som helst, fordi alt arver fra objekt:

```
List<Object> objekter = new List<object>();
```

Det skal vi også se nærmere på i dette kapitel.

Virtuelle metoder

Hvis du ønsker, at børn skal have sin egen implementation af et medlem, skal metoden i mor være *virtual*. Det sker ved at markere metoden med virtual-kodeordet. Når det er sket, kan børn vælge, om de vil overskrive den eller lade være. Overskrivning af en metode sker med override-kodeordet:

```
class Mor {
    public virtual void Metode() { }
}

// Vælger ikke at overskrive metoden
class Barn1 : Mor { }

// Vælger at overskrive metoden
```

```
class Barn2 : Mor {  
    public override void Metode() { }  
}
```

I eksemplet vil kald til Metode i instanser af Barn1 afvikle Metode i Mor, medens kald til Metode i instanser af Barn2 vil afvikle Barn2's egen implementation.

Et praktisk eksempel er den virtuelle ToString-metode i System.Object-klassen, der som nævnt er mor for alt. Hvis du ikke overskriver metoden, vil den blot returnere en streng svarende til navnet på klassen:

```
using System;  
  
namespace Demo  
{  
    internal class Program  
    {  
        private static void Main(string[] args)  
        {  
            Terning terning = new Terning();  
            Console.WriteLine(terning.ToString()); // Demo.Terning  
        }  
    }  
  
    class Terning  
    {  
        public int Værdi { get; private set; }  
        public void Ryst()  
        {  
            System.Random rnd = new System.Random();  
            this.Værdi = rnd.Next(1, 7);  
        }  
        public Terning()  
        {  
            this.Ryst();  
        }  
    }  
}
```



```
}
```

Læg mærke til, at ToString-metoden blot returnerer navnet. Men du kan vælge at overskrive metoden, så den returnerer en mere logisk tekst:

```
using System;
```

```
namespace Demo
```

```
{
```

```
    internal class Program
```

```
    {
```

```
        private static void Main(string[] args)
```

```
        {
```

```
            Terning terning = new Terning();
```

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

```
            // Jeg er en terning med værdien [2]
```

```
            // (eller andet tilfældigt tal)
```

```
        }
```

```
    }
```

```
class Terning
```

```
{
```

```
    public int Værdi { get; private set; }
```

```
    public void Ryst()
```

```
    {
```

```
        System.Random rnd = new System.Random();
```

```
        this.Værdi = rnd.Next(1, 7);
```

```
    }
```

```
    public Terning()
```

```
    {
```

```
        this.Ryst();
```

```
    }
```

```
    public override string ToString()
```

```
    {
```

```
        return $"Jeg er en terning med værdien [{this.Værdi}]";
```

```
    }
```

```
}  
}
```

Bemærk overskrivning af ToString som betyder, at Terning har sin helt egen ToString.

Kodeordene *virtual* og *override* hører sammen.

Du kan også vælge at skabe dine egne virtuelle metoder ved brug af virtual-kodeordet. Her er et eksempel på et arvehierarki, som definerer en terning med en almindelig Ryst-metode, og en speciel terning, der har sin helt egen implementation og henter en tilfældig værdi fra random.org:

```
class Terning  
{  
    public int Værdi { get; protected set; }  
    public virtual void Ryst()  
    {  
        System.Random rnd = new System.Random();  
        this.Værdi = rnd.Next(1, 7);  
    }  
    public Terning()  
    {  
        this.Ryst();  
    }  
  
    public override string ToString()  
    {  
        return $"Jeg er en terning med værdien [{this.Værdi}]";  
    }  
}  
  
class SpecielTerning : Terning {  
    public override void Ryst()  
    {  
        using (WebClient w = new WebClient())  
        {
```

```
        string s =  
            w.DownloadString("https://www.random.org/integers/" +  
                "?num=1&min=1&max=6&col=1&base=10&format=plain&rnd=new");  
        this.Værdi = Convert.ToInt32(s);  
    }  
}  
}
```

Nu vil den specielle terning afvikle sin egen version af Ryst og finde et tilfældigt tal fra nettet i stedet for fra Random-klassen.

Base

Hvis du gerne vil afvikle en metode i mor fra en metode i et barn, kan du vælge at benytte base-kodeordet:

```
class Mor {  
    public override string ToString()  
    {  
        return "mor";  
    }  
}  
  
class Barn : Mor {  
    public override string ToString()  
    {  
        return "Jeg er et barn af " + base.ToString();  
    }  
}
```

Nu vil et kald til barnets ToString-metode returnere "Jeg er et barn af mor".

Abstrakte metoder

I nogle situationer ønsker du at skrive klasser, der ligger til grund for nedarvning, som *ikke* har sin egen implementation, men som *kræver*, at børn overskriver metoden.

Det kaldes en abstrakt metode og er helt speciel på den måde, at mor ikke har nogen implementation overhovedet. Men hvis en klasse ikke

har nogen implementation af en metode, må man heller ikke kunne skabe en instans af klassen. Derfor *skal* abstrakte metoder placeres i abstrakte klasser.

Abstrakte klasser og metoder kan defineres ved hjælp af `abstract`-kodeordet, og overskrives ved hjælp af `override`-kodeordet:

```
abstract class Mor
{
    public abstract void Metode1();
    public void Metode2() {
        // Må gerne have sin egen metode
    }
    public virtual void Metode3() {
        // Må også gerne være virtual
    }
}

class Barn : Mor
{
    public override void Metode1()
    {
        // kode...
    }
}
```

Bemærk, at der ikke er angivet nogen implementation i `Metode1` i modsætning til `Metode2` og `Metode3`. Kompileren vil sørge for, at underklasser skal overskrive `Metode1`, og underklasser kan *vælge* om de vil overskrive `Metode3`, og den vil også smide en fejl, såfremt du forsøger at skabe en instans af `Mor`. Det må du ikke, fordi klassen er abstrakt. `Mor` kan således kun benyttes til nedarvning.

Her er et andet eksempel på en abstrakt klasse `Terning`, som dermed skal ligge til grund for en nedarvet klasse. Klassen har en abstrakt metode `Ryst`, hvilket betyder, at underklasser skal implementere sin egen version af metoden. Men runtime kan være 100 % sikker på, at

metoden eksisterer i alle underklasser – det skal kompilatoren nok sørge for:

```
abstract class Terning
{
    public int Værdi { get; protected set; }
    public abstract void Ryst();
    public Terning()
    {
        this.Ryst();
    }
}

class YatzyTerning : Terning
{
    public override void Ryst()
    {
        System.Random rnd = new Random();
        this.Værdi = rnd.Next(1, 7);
    }
}
```

Brugen af abstrakte og virtuelle metoder giver dig altså mulighed for at skabe et hierarki af klasser med forskellige former for implementering.

Mor kan altid pege på sine børn

De regler som kompilatoren sørger for, at du overholder i et arvehierarki, giver en anden mulighed for polymorfi, som mange kæmper med at forstå i starten – nemlig det faktum at en variabel af en overklasse altid kan indeholde en reference til instanser af sig selv og af instanser af underklasser.

Det kan være svært at forstå, men i virkeligheden er det meget simpelt: et barn er jo en mor!

Eller sagt på en anden måde. De medlemmer, der findes i mor, findes *altid* i barnet og kan *ikke* fjernes. De kan overskrives, men aldrig

fjernes. Derfor kan runtime være sikker på, at en variabel af typen mor altid vil have sine egne medlemmer tilgængelig i sine børn.

Se følgende simple eksempel:

```
class Mor
{
    public void Metode1()
    {
        // kode...
    }
}

class Barn : Mor
{
    public void Metode2()
    {
        // kode...
    }
}
```

Der er tale om et simpelt arvehierarki med et barn, der arver fra mor, og følgende kode er derfor fuldt lovlig:

```
// Dette er helt logisk
Mor a = new Mor();
Barn b = new Barn();

// og ved nærmere eftertanke er dette
// også logisk fordi er barn ER jo en mor
Mor c = new Barn();

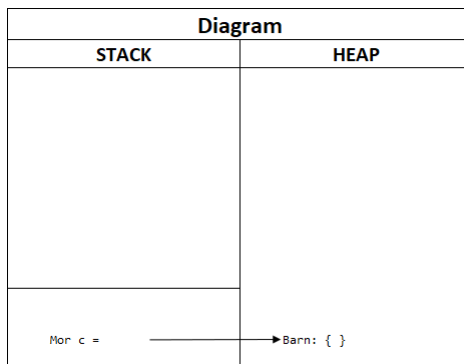
// Dette duer til gengæld ikke
// Barn d = new Mor();
```

Det er variablen c, som er interessant. Den er af typen Mor, men må gerne indeholde en reference til et barn i hukommelsen (heap) fordi runtime kan være helt sikker på, at de medlemmer, som er i Mor med 100 % sikkerhed også er i Barn – herunder Metode1:

```
c.Metode1();
```

I Barn findes jo også Metode2, men denne kan ikke tilgås af en variabel af type Mor, for her er kun Metode1 synlig. Der er et objekt (en instans) af Barn i hukommelsen, men medlemmer fra Barn kan blot ikke ses af en variabel af typen Mor.

Diagrammet for variabel c ser således ud:



Det er vigtigt at forstå, at objektet af Barn ikke på nogen måde er ændret til noget andet – det bliver blot refereret af en variabel af en højere type i samme arvehierarki.

Typecheck og typekonvertering

Så – endnu engang – følgende kode er fuldt lovlig:

```
Mor a = new Barn();  
a.Metode1();
```

og du ved, at objektet i hukommelsen i virkeligheden er af typen Barn og alt, hvad den nu består af, udover det den får fra Mor. Men da Mor kun kender til Metode1 – hvordan kan du så afvikle Metode2:

```
Mor a = new Barn();
// a.Metode2();
// "Mor does not contain a definition for Metode2" exception
```

Det kræver en typekonvertering, og dette kræver kode fra din side – for denne kode fejler:

```
// Barn b = a; // Cannot implicit convert Barn to Mor
```

Men du har forskellige muligheder:

```
Barn b = a as Barn;  
Barn c = (Barn)a;
```

Begge typekonverteringer vil kunne kompileres, men forskellen skal findes i eventuelle fejl. Ved brug af `as`-kodeordet vil variablen blive tildelt værdien `null` ved en eventuel fejl. Ved brug af `()` konverteringen vil der blive kastet en `Exception`, som du kan vælge at fange i en `try/catch` struktur.

Så hvis du er i tvivl, om en konvertering går godt, bør du kontrollere for en `null`-værdi:

```
Barn b = a as Barn;  
if (b != null) {  
  
}
```

Der findes ligeledes en `is`-operator, du kan benytte til at kontrollere, om et objekt er af den type, du forventer. Følgende kode beviser i øvrigt også, at et `Barn` reelt er en `Mor` (og alt arver fra `Object`):

```
Mor a = new Barn();  
Console.WriteLine(a is Object); // true  
Console.WriteLine(a is Mor);    // true  
Console.WriteLine(a is Barn);    // true  
  
Console.WriteLine(a is int);     // false  
Console.WriteLine(a is string);  // false
```

Virtuelle metoder (igen)

Du har tidligere set, hvordan virtuelle (eller abstrakte) metoder kan give underklasser sin egen implementation, og kombinationen af dette

samt det faktum at mor altid kan pege på sine børn, kan bruges til mange ting:

```
class Mor
{
    public virtual void Metode1()
    {
        Console.WriteLine("Jeg er mor");
    }
}

class Barn : Mor
{
    public override void Metode1()
    {
        Console.WriteLine("Jeg er barn");
    }
}
```

Med udgangspunkt i disse klasser kan følgende kode afvikles:

```
Mor a = new Mor();
a.Metode1();           // Jeg er mor
Barn b = new Barn();
b.Metode1();           // Jeg er barn
Mor c = new Barn();
c.Metode1();           // Jeg er barn
```

Det er variabel c, du skal fokusere på. Den er af typen Mor, men indeholder en reference til et objekt af typen Barn. Når den virtuelle og overskrevne Metode1 afvikles, er det barnets metode, der afvikles. Det er derfor denne feature i objektorienteret programmering hedder polymorfi (mange former), og den giver en masse fordele.

Polymorfi og typestærke samlinger

Hvis du kan skabe et arvehierarki med virtuelle eller abstrakte metoder, har du også mulighed for at udnytte polymorfi i typestærke samlinger:

```
class Mor
{
    public virtual void Metode1()
    {
        Console.WriteLine("Jeg er mor");
    }
}

class Barn1 : Mor
{
    public override void Metode1()
    {
        Console.WriteLine("Jeg er barn1");
    }
}

class Barn2 : Mor
{
    public override void Metode1()
    {
        Console.WriteLine("Jeg er barn2");
    }
}
```

I stedet for en samling (array, liste, stak, kø med videre) af konkrete typer kan du skabe en samling af en overklasse og tilføje objekter af underklasser – de er jo alle sammen en mor:

```
List<Mor> lst = new List<Mor>();
lst.Add(new Barn1());
lst.Add(new Barn2());
lst.Add(new Mor());
lst.Add(new Barn1());
lst.Add(new Barn2());

foreach (Mor i in lst)
    i.Metode1();

/*
    Jeg er barn1
```

Jeg er barn2

Jeg er mor

Jeg er barn1

Jeg er barn2

*/

Da metoden Metode1 er overskrevet i alle børn, vil den rigtige metode blive afviklet helt automatisk.

Med klasserne Mor, Barn1 og Barn2 er det hele lidt abstrakt, men tænk over hvordan det kan udnyttes i en rigtig applikation.

I et ERP-system (Enterprise Resource Planning), med et hierarki med klassen Virksomhed som mor, kan en samling af denne type indeholde alle andre typer i hierarkiet (debitor, kreditor, partner med videre). Hvis disse typer har overskrevne metoder som Hent, Gem, Print eller SendFaktura giver det en effektiv mulighed for at afvikle metoder på alle objekter. Hvis metoderne i Virksomhed er virtuelle, kan de overskrives, hvis det er nødvendigt, men Virksomhed kunne også være en abstrakt klasse, så man ikke kan skabe en instans af denne, men kun af underklasser. Så kunne metoderne også være abstrakte og dermed skabes et krav om, at underklasser skal have deres egen implementation af metoderne. Men runtime kan stadig være sikker på, at metoderne findes – og det er hele essensen i polymorfi.

Hvis du vil vide mere

Når du bliver lidt mere øvet, kan du måske lede efter yderligere information relateret til dette kapitel. Søg eksempelvis efter:

- *Pattern matching* (blandt andet i forbindelse med brug af switch)
- *Shadowing*, som giver mulighed for at gemme et medlem væk fra højere variabeltyper i et hierarki.

Interface

Du har tidligere i bogen lært om aktrakte klasser og aktrakte medlemmer, og ideen med dem er, at runtime er helt sikker på, at der findes et medlem i en klasse med en helt bestemt signatur (navn, returværdi og argumenter). Det kunne eksempelvis se således ud:

```
public abstract class Terning
{
    public int Værdi { get; protected set; }
    public abstract void Ryst();
}
```

Bemærk Ryst-metoden som er abstrakt og dermed uden nogen form for implementation. Den skal tilføjes i nedarvede klasser, og man kan heller ikke skabe en instans af klassen Terning. Den *skal* nedarves, og underklasser *skal* implementere Ryst:

```
public class YatzyTerning : Terning
{
    public override void Ryst()
    {
        this.Værdi = new Random().Next(1, 7);
    }
}
```

Dermed kan klassen YatzyTerning indgå i polymorfi jævnfør forrige kapitel, fordi kompileren *ved*, at der findes en Ryst:

```
Terning t = new YatzyTerning();
t.Ryst();
Console.WriteLine(t.Værdi);
```

Du kan skabe samme typesikkerhed ved hjælp af en anden overordnet type kaldet *interface*. I denne type definerer du, hvilke medlemmer som *skal* være tilgængelige i de klasser, som *implementerer* det konkrete interface.

Definition af et interface

Et interface placeres på namespace-niveau ligesom andre overordnede typer (klasser, strukturer med videre) og ser således ud:

```
public interface ITerning
{

}
```

Typisk vil du se, at et interface er navngivet med et stort I samt et navn – det er dog helt op til dig.

I et interface kan du definere, hvilke medlemmer der skal være tilgængelige, og du skal angive den fulde signatur, men uden definering af synlighed (skal under alle omstændigheder være offentlige) og uden nogen form for implementering. Eksempelvis som følger:

```
public interface ITerning
{
    int Værdi { get; set; }
    void Ryst();
}
```

Du kan tilføje så mange medlemmer (egenskaber, metoder eller hændelser), du ønsker. I eksemplet er der angivet en egenskab og en metode, og denne definition ligger til grund for klasser, som *implementerer* det konkrete interface.

Implementering af et interface

Et interface skal implementeres i en klasse eller struktur og det sker med samme syntaks som nedarvning:

```
class/struct [klasse navn] : [interface navn] { }
```

så vil kompileren sørge for, at medlemmer defineret i det konkrete interface implementeres i klassen eller strukturen:

```
public class YatzyTerning : ITerning
```

```
{
    public int Værdi { get; set; }

    public void Ryst()
    {
        this.Værdi = new Random().Next(1, 7);
    }

    public void Skriv()
    {
        Console.WriteLine($"[{this.Værdi}]");
    }
}
```

Du kan tilføje alle mulige andre medlemmer til en klasse, der implementerer et interface. I eksemplet har metoden `Skriv` ikke noget med `ITerning` at gøre, men pointen er, at kompileren *ved*, at der findes medlemmer, som er defineret i et interface.

Da et interface er en referencetype, giver det dermed følgende muligheder:

```
YatzyTerning a = new YatzyTerning();
a.Ryst();
a.Skriv();

ITerning b = new YatzyTerning();
b.Ryst();
Console.WriteLine(b.Værdi);
```

Som det fremgår, er `YatzyTerning` en helt normal klasse, men da den også er en `ITerning`, må en variabel af denne type gerne indeholde en reference til en `YatzyTerning` – og så er vi tilbage i polymorfi.

Så det er altså muligt at opbygge en samling af klasser, som implementerer et interface (eller flere), og referere til objekter af disse typer gennem en variabel af interface typen.

IComparable <T>

Som et lidt mere komplekst eksempel på brug af interface kan du se på følgende kode:

```
using System;
using System.Collections.Generic;

namespace Demo
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            List<Terning> terninger = new List<Terning>();
            for (int i = 0; i < 5; i++)
                terninger.Add(new Terning());
            terninger.Sort();    // FEJLER!!
            foreach (var terning in terninger)
                Console.WriteLine(terning.Værdi);
        }
    }

    public class Terning {
        public int Værdi { get; private set; }
        public void Ryst() {
            this.Værdi = new Random().Next(1, 7);
        }
        public Terning()
        {
            this.Ryst();
        }
    }
}
```

I koden bliver der oprettet en liste af fem terninger baseret på en almindelig klasse Terning, og herefter forsøges listen sorteret. Hvis du prøver koden, vil du opleve, at Sort-metoden fejler, hvilket jo i virkeligheden er logisk nok – hvor skulle den vide fra, hvordan en terning skal sorteres?

Men vi kan faktisk godt fortælle Sort-metoden, hvordan den skal sortere terninger ved at lade klassen implementere et indbygget interface kaldet `IComparable<Terning>`. Dette interface sikrer, at metoden `Compare` implementeres, og så kan runtime sortere terningerne:

```
using System;
using System.Collections.Generic;
namespace Demo
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            List<Terning> terninger = new List<Terning>();
            for (int i = 0; i < 5; i++)
                terninger.Add(new Terning());
            terninger.Sort();
            foreach (var terning in terninger)
                Console.WriteLine(terning.Værdi);
        }
    }

    public class Terning : IComparable<Terning> {
        public int Værdi { get; private set; }
        public void Ryst() {
            this.Værdi = new Random().Next(1, 7);
        }

        public int CompareTo([AllowNull] Terning other)
        {
            if (this.Værdi > other.Værdi)
                return 1;
            if (this.Værdi < other.Værdi)
                return -1;
            return 0;
        }

        public Terning()
        {

```



```
        this.Ryst();  
    }  
}  
}
```

Når du afvikler koden, vil du opleve, at terningerne bliver sorteret helt korrekt, og det skyldes at CompareTo-metoden returnerer et tal, der kan bruges til at sammenligne og dermed sortere terningerne. Metoden kan skrives på mange måder, men du skal blot returnere et heltal, som kan benyttes til sammenligning.

Afkobling

Du vil tit finde interface benyttet i forskellige former for software mønstre – herunder afkobling gennem et mønster kaldet *dependency injection* (forkortet DI). Det vil du falde over mange steder, og det er i øvrigt kun en af mange forskellige måder at skabe kode, som ikke er så hårdt bundet til konkrete klasser. Bare for at du får en ide om, hvad DI er, er her et kort eksempel.

Prøv at se følgende klasse:

```
public class Terning  
{  
    public int Værdi { get; private set; }  
    private Random rnd;  
    public void Ryst()  
    {  
        this.Værdi = rnd.Next(1, 7);  
    }  
    public Terning()  
    {  
        this.rnd = new Random();  
        this.Ryst();  
    }  
}
```

Det er jo en helt almindelig terning, men rent arkitektonisk er klassen *hårdt bundet* til System.Random, som bruges til at skabe tilfældige tal.

Det kan give problemer i senere versioner, hvis du (eller brugerne af klassen) ønsker at skifte tilfældighedsgenerator, fordi det vil kræve en rekompilering af koden.

Men hvis du nu går et skridt op i abstraktionsniveau og benytter et interface, kan du løse problemet. Her er et interface med en enkelt metode:

```
public interface ITilfældighedsGenerator {  
    int FindTilfældigtTal();  
}
```

Hvis klassen terning tilrettes således, at det ikke er System.Random, der kodes mod, men i stedet ITilfældighedsGenerator ser det således ud:

```
public class Terning  
{  
    public int Værdi { get; private set; }  
    private ITilfældighedsGenerator rnd;  
    public void Ryst()  
    {  
        this.Værdi = rnd.FindTilfældigtTal(); ;  
    }  
    public Terning(ITilfældighedsGenerator generator)  
    {  
        this.rnd = generator;  
        this.Ryst();  
    }  
}
```

Nu er System.Random skiftet ud med ITilfældighedsGenerator, og i konstruktøren skal en instans, der implementerer dette interface, medsendes.

Det kræver jo en klasse eller struktur, som implementerer dette interface – og der er helt frit valg i implementationen. Det eneste kompileren sikrer er, at metoden FindTilfældigtTal *skal* være til stede.

Her er et par forslag:

```
class TilfældighedsGeneratorRandom : ITilfældighedsGenerator
{
    public int FindTilfældigtTal()
    {
        return new System.Random().Next(1, 7);
    }
}

class TilfældighedsGeneratorSnyd : ITilfældighedsGenerator
{
    public int FindTilfældigtTal()
    {
        return 6;
    }
}

class TilfældighedsGeneratorRandomOrg : ITilfældighedsGenerator
{
    public int FindTilfældigtTal()
    {
        string url = "https://www.random.org/integers/" +
            "?num=1&min=1&max=6&col=1&base=10&format=plain&rnd=new";
        using (System.Net.WebClient w = new System.Net.WebClient())
        {
            string s = w.DownloadString(url);
            return Convert.ToInt32(s);
        }
    }
}
```

Den ene klasse benytter System.Random, en anden returnerer blot 6, og den sidste henter et tilfældigt tal fra random.org – så implementeringen er forskellig, men interface er det samme. Og det kan så udnyttes, når der skabes en instans af Terning:

```
Terning t1 = new Terning(new TilfældighedsGeneratorRandom());
Terning t2 = new Terning(new TilfældighedsGeneratorSnyd());
Terning t3 = new Terning(new TilfældighedsGeneratorRandomOrg());
```

Det er nu brugeren af Terning, som bestemmer hvilken tilfældigheds-generator, der skal bruges, og brugeren kan sågar skabe og benytte sin egen klasse, så længe den implementerer det nævnte interface.

I arkitektur kalder man mønsteret i dette eksempel for *constructor based dependency injection*, fordi brugeren tilføjer Terning den ønskede generator i konstruktøren. Det er den simpleste form for DI (også kaldet fattigmands DI), men beskriver meget godt, hvor effektivt brug af et interface kan være.

Software-arkitektur ligger et stykke uden for denne bogs rammer. Hvis du er mere interesseret, er bøger som *Patterns of enterprise application architecture* af Martin Fowler eller *Design Patterns: Elements of Reusable Object-Oriented Software* af Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (også kaldet *Gang of four*) måske værd at bruge lidt tid på. De er ikke relateret til C#, men til generel softwareudvikling.

Hvis du vil vide mere

Når du bliver lidt mere øvet, kan du måske lede efter yderligere information relateret til dette kapitel. Søg eksempelvis efter brug af `using` og `IDisposable` samt de helt nye *default interface methods*.

Delegates

Du har indtil nu lært om fire forskellige muligheder for skabe egne typer – klasse, struktur, enumeration og interface. I dette kapitel skal du lære om en af de mere avancerede typer kaldet en *delegate*.

En delegate, der, som navnet lidt antyder, giver mulighed for at delegere metodekald videre. I nogle programmeringssprog kaldes det en funktionspointer, men det er noget anderledes skruet sammen i C#, fordi sproget er så typestærkt og typesikkert.

I sin helt grundlæggende form er en delegate en type, du kan skabe et objekt af, og dette objekt indeholder en liste af referencer til metoder af samme signatur (returværdi og argumenter). Da delegate-typen er referencebaseret, betyder det, at du kan gemme eller videresende en reference til en eller flere metoder, og disse metoder kan så senere afvikles.

Min erfaring med at lære studerende, hvordan delegates virker, er, at det ikke det er så svært rent syntaksmæssigt, men at det er svært at se det overordnede formål, så lad mig vise dig et par eksempler på brugen af delegates, inden vi kigger på syntaks.

En terning med log

Lad os antage, at du har fået til opgave at skabe en terning. Det har vi gjort mange gange i løbet af bogen, men denne terning skal kodes således, at hver gang der rystes en ny værdi, så skal den skrive info til en log om, at er der rystet samt angive den nye værdi.

I første version af terningen skal du blot logge til konsol, og det kan du kode som følger:

```
public class Terning
{
    public int Værdi { get; private set; }
    private Random rnd = new Random();
```

```
public void Ryst()
{
    this.Værdi = rnd.Next(1, 7);
    this.Log($"Rystet til en {this.Værdi}'er");
}

private void Log(string tekst)
{
    Console.WriteLine(tekst);
}

public Terning()
{
    this.Ryst();
}
}
```

Bemærk, at når Ryst-metoden kaldes, så sørger den for at kalde den private Log-metode, som skriver til konsol. Prøv at kode det selv, så du kan se funktionaliteten.

Det varer dog ikke længe, før der på listen af opgaver (issues hedder det typisk i et source control-system) kræves en rettelse af log-funktionaliteten. Flere kunder vil gerne have mulighed for at logge til en fil (c:\temp\terning.log), til konsol, til begge dele eller slet ikke logge. Du må tilbage til koden, og kommer måske frem til følgende simple løsning:

```
public class Terning
{
    public int Værdi { get; private set; }
    private Random rnd = new Random();
    public bool LogTilFil { get;
        private set; }
    public bool LogTilKonsol { get;
        private set; }
    public void Ryst()
    {
        this.Værdi = rnd.Next(1, 7);
        this.Log($"Terning er rystet til en {this.Værdi}'er");
    }
}
```

```
private void Log(string tekst)
{
    if (this.LogTilKonsol)
        Console.WriteLine(tekst);
    if (this.LogTilFil)
        System.IO.File.AppendAllText(@"c:\temp\log.txt",
            tekst + "\r\n");
}
public Terning(bool logTilKonsol, bool logTilFil)
{
    this.LogTilFil = logTilFil;
    this.LogTilKonsol = logTilKonsol;
    this.Ryst();
}
}
```

Nu skal instanser af terninger skabes med angivelse af, hvordan man ønsker log:

```
// logger både til konsol og til fil
Terning t = new Terning(true, true);
```

Opgave løst – kompilér applikation og send til kunderne.

Der varer dog ikke længe, før nogle kunder også gerne vil kunne logge til forskellige databaser samt foretage http-kald. Det er jo skruen uden ende! Du skal gøre koden i klassen mere og mere kompleks for at imødekomme kundernes krav og vil blive nødt til at skuffe nogle kunder, fordi du ikke ønsker, at din Terning-applikation skal have reference til eksempelvis obskure databasedrivere.

Den endelige terning

Det er her, en delegate kan hjælpe dig!

En delegate er som nævnt en type, der kan skabes et objekt af, og i dette objekt er der en liste af reference til metoder med samme signatur. Hvis nu du aftaler med kunderne, at de selv kan kode deres egen log-metode med en konkret signatur (måske en void metode med

et string-argument), så skal du nok sørge for at afvikle metoden eller metoderne, når der rystes. På den måde lægger du jo hele log-funktionaliteten over på kunderne selv. De kan kode præcis, hvad de har lyst til og blot tilføje reference til metoderne til et delegate-objekt. Terningen indeholder en reference til objektet og sørger for at kalde metoderne, når der rystes.

Her er et eksempel på en sådan løsning. Du skal ikke forstå syntaksen, men ideen bag brugen af et delegate-objekt:

```
public class Terning
{
    public int Værdi { get; private set; }
    private Random rnd = new Random();
    public Action<string> LogMetoder { get;
        set; }

    public void Ryst()
    {
        this.Værdi = rnd.Next(1, 7);
        this.Log($"Terning er rystet til en {this.Værdi}'er");
    }

    private void Log(string tekst)
    {
        if (LogMetoder != null)
            LogMetoder.Invoke(tekst);
    }
    public Terning()
    {
        this.Ryst();
    }
}
```

Du skal altså se bort fra forståelse af syntaks lige nu, men i klassen er egenskaben LogMetoder en reference til et delegate objekt, der kan indeholde referencer til void-metoder med et enkelt string argument:

```
public Action<string> LogMetoder { get; set; }
```


og i Log-metode afvikles disse metoder (hvis der er nogen):

```
if (LogMetoder != null)
    LogMetoder.Invoke(tekst);
```

Så nu er al log-funktionalitet lagt i hænderne på brugerne af klassen (igen – se bort fra syntaks – det er forståelsen, det handler om):

```
// Ingen log
Terning t1 = new Terning();
t1.Ryst();

// Log til konsol
Terning t2 = new Terning();
t2.LogMetoder += tekst => Console.WriteLine(tekst);
t2.Ryst();

// Log til fil
Terning t3 = new Terning();
t3.LogMetoder += tekst =>
    File.AppendAllText(@"c:\temp\log.txt", tekst + "\r\n");
t3.Ryst();

// Log til fil og konsol
Terning t4 = new Terning();
t4.LogMetoder += tekst => Console.WriteLine(tekst);
t4.LogMetoder += tekst =>
    File.AppendAllText(@"c:\temp\log.txt", tekst + "\r\n");
t4.Ryst();

// Log til HTTP
Terning t5 = new Terning();
t5.LogMetoder += tekst => {
    using (System.Net.WebClient w = new System.Net.WebClient())
    {
        w.UploadString("http://www.minlog.dk/log/", "POST", tekst);
    }
};
t5.Ryst();
```

I koden benyttes noget helt nyt i forhold til oprettelse af metode – en såkaldt *lambda*-syntaks – som i virkeligheden blot er en simpel måde at skabe en metode på. Mere om dette senere i dette kapitel, men eksempelvis vil koden:

```
t2.LogMetoder += tekst => Console.WriteLine(tekst);
```

oprette et delegate objekt og tildele en enkelt metode på listen som udskriver på konsol. Denne metode vil så blive kaldt af koden i Terning-klassen. At der benyttes en Lambda har ikke noget med delegate objektet at gøre. En Lambda er bare en anden måde at skrive en metode på.

I seneste kapitel lærte du om brugen af interface, og at det blandt andet blev brugt til afkobling af *data* (dependency injection). Brugen af en delegate til log-funktionalitet kan du også formulere som afkobling af *funktionalitet*.

Hvad er en delegate

Du kan opfatte en delegate som en form for en klasse.

Den repræsenterer en skabelon, du kan skabe instanser af, men i modsætning til klasser, hvor du kan definere mange forskellige medlemmer (felter, egenskaber, metoder med videre), så kan du som udgangspunkt kun definere én ting i en delegate – nemlig hvilken metodesignatur (returværdi og argumenter) instanser af typen kan indeholde referencer til.

Du kan arbejde med delegates på to måder. I ældre C# kode vil du måske falde over en definition som eksempelvis:

```
public delegate void MinDelegate1();  
public delegate void MinDelegate2(int a);  
public delegate int MinDelegate3();  
public delegate bool MinDelegate4(string a, int b);
```

Koden definerer fire typer hvor:

- MinDelegate1 kan indeholde referencer til metoder uden returværdi og uden argumenter
- MinDelegate2 kan indeholde referencer til metoder uden returværdi og et enkelt int argument
- MinDelegate3 kan indeholde referencer til metoder, der returnerer en int uden argumenter
- MinDelegate3 kan indeholde referencer til metoder, der returnerer en bool med argumenterne string og int.

Definition er typisk placeret på namespace niveau ligesom klasser og kan benyttes på samme måde, som klasser benyttes til at skabe instanser.

Dog kan instanser oprettes på flere måder:

```
using System;

namespace Demo
{
    public delegate void MinDelegate1();

    internal class Program
    {
        private static void Main(string[] args)
        {
            // Brug af konstruktør
            MinDelegate1 d1 = new MinDelegate1(Metode1);
            // Genvej til oprettelse af instans (blot ref til metode)
            MinDelegate1 d2 = Metode1;
        }

        public static void Metode1()
        {
            Console.WriteLine("I Metode1");
        }
    }
}
```

Når først instanser af en delegate type er oprettet, kan de metoder, der er gemt referencer til, afvikles – igen på flere måder:

```
using System;

namespace Demo
{
    public delegate void MinDelegate1();

    internal class Program
    {
        private static void Main(string[] args)
        {

            // Direkte afvikling
            Metode1();                // I Metode1

            // Inddirekte afvikling
            MinDelegate1 d1 = Metode1;
            d1.Invoke();              // I Metode1
            // eller blot
            d1();                    // I Metode1

            // Som argument
            DelegateSomArgument(d1); // I Metode1

            // Som returværdi
            MinDelegate1 d2 = DelegateSomReturVærdi();
            d2();                    // I Metode1
            // eller blot
            DelegateSomReturVærdi()(); // I Metode1
        }

        public static void Metode1()
        {
            Console.WriteLine("I Metode1");
        }

        public static void DelegateSomArgument(MinDelegate1 d)
```

```
{
    d();
}

public static MinDelegate1 DelegateSomReturVærdi()
{
    return Metode1;
}
}
```

Bemærk, at referencer til metoder gemt i en instans af en delegate kan afvikles på flere måder – eksempelvis:

```
d1.Invoke();
```

eller blot:

```
d1();
```

Det er ligegyldigt, hvilken version du vælger, men normalt bør du teste om et delegate-objekt indeholder referencer, hvilket (lidt ulogisk) kan gøres ved at teste om objektet er null:

```
if(d1 != null)
    d1.Invoke();
```

```
// eller
```

```
if(d1 != null)
    d1();
```

```
// eller
```

```
d1?.Invoke();
```

Referencen til delegate-instansen kan naturligvis benyttes til både returværdi og argument, ligesom den kan gemmes i objekter.

Husk at kontrollere for en null værdi inden du kalder Invoke-metoden.

Eksemplet benytter udelukkende en delegate, der kan indeholde referencer til en void-metode uden argumenter, men prøv at kopiere koden til en ny konsolapplikation og leg lidt med at ændre typerne til at håndtere forskellige returværdier og argumenter.

Indbyggede delegates

I de nyere versioner af C# vil du dog sjældent falde over brugen af delegate kodeordet, fordi der findes en bedre og nemmere måde at definere en delegate-type. Det sker ved hjælp af de indbyggede typer *Action*, *Func* og *Predicate* som findes i forskellige variationer.

Type	Svarer til
<i>Action</i>	<i>void delegate uden argumenter</i>
<i>Action<T></i>	<i>void delegate med argumenter (angivet som en liste af T)</i>
<i>Func<T></i>	<i>delegate med returværdi og eventuelle argumenter (angivet som en liste af T)</i>
<i>Predicate<T></i>	<i>delegate der altid returnerer en bool og tager et argument af typen T</i>

Tabel 24 Indbyggede delegates

Du behøver ikke være så opmærksom på Predicate-typen, den benyttes ikke så meget mere i nyere C#, men ved at benytte de andre indbyggede delegates kan du slippe for at definere dine egne, så længe du holder dig til metoder med højst 15 argumenter.

Således kan de delegates, du så tidligere, helt undgås, fordi variabler kan erklæres således:

```
using System;

namespace Demo
{
    internal class Program
    {
```

```
// Brug af det gamle delegate-kodeord
public delegate void MinDelegate1();
public delegate void MinDelegate2(int a);
public delegate int MinDelegate3();
public delegate bool MinDelegate4(string a, int b);

private static void Main(string[] args)
{
    MinDelegate1 d1a;
    // Samme som
    Action d1b;

    MinDelegate2 d2a;
    // Samme som
    Action<int> d2b;

    MinDelegate3 d3a;
    // Samme som
    Func<int> d3b;

    MinDelegate4 d4a;
    // Samme som
    Func<string, int, bool> d4b;
}
}
```

Bemærk, at typen på eventuelle argumenter angives med den generiske definition <>, og når der er tale om en Func, svarer den sidste type til typen på returværdien.

Når du benytter en Func så husk, at den sidste angivne type repræsenterer returværdien.

Det forrige eksempel på brug af en void delegate kan nu omskrives til brug af en Action:

```
using System;
```

```
namespace Demo
{
    internal class Program
    {
        private static void Main(string[] args)
        {

            // Direkte afvikling
            Metode1();                // I Metode1

            // Inddirekte afvikling
            Action d1 = Metode1;
            d1.Invoke();               // I Metode1
            // eller blot
            d1();                     // I Metode1

            // Som argument
            DelegateSomArgument(d1);  // I Metode1

            // Som returværdi
            Action d2 =
                DelegateSomReturVærdi();
            d2();                     // I Metode1
            // eller blot
            DelegateSomReturVærdi()(); // I Metode1
        }

        public static void Metode1()
        {
            Console.WriteLine("I Metode1");
        }

        public static void DelegateSomArgument(Action d)
        {
            d();
        }

        public static Action DelegateSomReturVærdi()
        {

```



```
        return Metode1;
    }
}
```

Prøv at sammenligne koden med det forrige eksempel og du vil se, at delegate definitionen er væk, og typer er blot erstattet med en Action.

Under alle omstændigheder ender det med et objekt af en delegate, som benyttes på samme måde. Det er bare meget nemmere at skrive Action, Action<T> eller Func<T>.

Så når du ser eller skriver kode som dette:

```
using System;

namespace Demo
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            Func<int, int, int> d = LægSammen;
            Console.WriteLine(d(5, 5));           // 10
        }

        public static int LægSammen(int a, int b)
        {
            return a + b;
        }
    }
}
```

kan du oversætte det til:

- Erklær en variabel d, som kan indeholde en reference til et delegate objekt, der kan indeholde referencer til metoder, der returnerer et heltal og tager to heltal som argumenter.
- Opret et nyt delegate objekt, hvor der gemmes en reference til metoden LægSammen og gem referencen i d

- Afvikling af den metode, der er gemt som reference i delegate objektet.

Flere referencer i en delegate

Som tidligere nævnt består et delegate objekt af en *liste af referencer*, og et kald til Invoke-metoden vil sørge for at afvikle dem alle.

For at tilføje flere referencer kan du bruge += operatoren, og du kan fjerne referencer med -= operatoren. Det er best practice at fjerne referencer, når du er færdig med at bruge dem. Det kan ske ved enten at benytte -= operatoren eller ved at sætte variabelen til null. I nogle situationer har det ingen betydning, men især i brugerflade-applikationer kan der være *hængende* referencer, og det er derfor et af de steder, hvor du kan komme til at skabe *memory leaks* – en af de sværeste fejl at finde i en C# applikation.

Her er et kort eksempel på tilføjelse af flere referencer:

```
using System;

namespace Demo
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            Action<string> d = Metode1;
            d += Metode2;
            d += Console.WriteLine;

            d.Invoke("Test");
            // eller bare d("Test");

            /*
             I Metode1 med Test
             I Metode2 med Test
             Test
            */
        }
    }
}
```

```
// Ryd op  
d = null;  
  
}  
  
public static void Metode1(string a)  
{  
    Console.WriteLine($"I Metode1 med {a}");  
}  
  
public static void Metode2(string a)  
{  
    Console.WriteLine($"I Metode2 med {a}");  
}  
}  
}
```

Som du kan se, tilføjes tre metoder til et nyt (Action<string>) delegate objekt, og et kald til Invoke vil afvikle alle tre metoder på en gang.

Hvis du ser eksemplet i starten af kapitlet, vil du se et andet eksempel på et delegate objekt med flere referencer.

Lambda-udtryk

Nu har du lært om delegates og ved, at objekter af delegates kan indeholde referencer til metoder, og referencen til delegate objekterne kan sendes med som argumenter til andre metoder eller benyttes som returnværdier. Dermed har du viden nok til at kunne forstå lambda-udtryk, som bliver benyttet meget i moderne C#.

Lambda-udtryk er blot en nem måde at skrive en metode på.

Lambda-udtryk har fået sit navn fra den amerikanske matematiker Alonzo Church, som blandt andet arbejdede med teorierne bag logiske

lambda-kalkyler, og findes nu som en notation for anonyme funktioner i mange programmeringssprog – herunder C#.

I C# erklæres lambda-udtryk ved hjælp af operatoren => (fed pil eller fat arrow på engelsk). Så når du ser en => i koden, ved du, at du har fat i et lambda-udtryk/anonym metode.

I virkeligheden er lambda-udtryk blot en nem og hurtig måde at skrive en funktion på, og når du lige lærer syntaksen, er det meget nemt. Det som kan forvirre en del, når du starter med at lære lambda-udtryk er, at et udtryk kan forkortes ned til næsten ingenting, men når du har lært reglerne for forkortelse, ser du hurtigt lyset.

Lad os tage udgangspunkt i følgende kode:

```
using System;

namespace Demo
{
    internal class Program
    {
        private static void Main(string[] args)
        {

            Func<int, int> d = Dobbelt;
            Console.WriteLine(d(5));    // 10

        }

        public static int Dobbelt(int a)
        {
            return a * 2;
        }
    }
}
```

I koden erklæres en variabel d, som kan indeholde referencen til en delegate, som igen kan indeholde referencer til metoder, der returnerer et heltal og tager et heltal som argument. På samme linje oprettes et delegate objekt med en reference til metoden. Herefter

afvikles metoden indirekte gennem variablen *d*. Metoden *Dobbelt* er defineret som en helt almindelig metode, men kan ændres til et lambda-udtryk for at gøre koden nemmere at skrive og læse.

Når du skriver lambda-udtryk, er operatoren *=>* helt central. På venstre side af operatoren findes argumenterne, og på højre side selve metodekroppen.

[argumenter] => [metodekrop]

Du kan også se det som argumenter, *der bliver sendt ind* i en metode.

Så koden fra tidligere kan ændres til følgende:

```
using System;

namespace Demo
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            Func<int, int> d = (int a) => { return a * 2; };
            Console.WriteLine(d(5));    // 10
        }
    }
}
```

Allerede her kan du se, hvorfor et lambda-udtryk også kaldes en anonym metode. Vores *Plus*-metode har ikke længere noget navn, og selve udtrykket skaber blot en reference til et delegate objekt – præcis på samme måde som tidligere.

Men styrken ved lambda-udtryk er muligheden for at forkorte koden, og det sker efter tre simple regler. Her er først det komplette udtryk uden forkortelser fra eksemplet tidligere:

```
Func<int, int> d = (int a) => { return a * 2; };
```

Første regel er, at du *ikke behøver angive typen på argumenterne*. Kompileren kan blot kigge på erklæringen af delegate objektet og ud fra dette udledes, at argumentet er et heltal. Så koden kan forkortes til:

```
Func<int, int> d = (a) => { return a * 2; };
```

Kompileren kan også ud af delegate erklæringen udlede, at returværdien er et heltal, så det behøver du slet ikke nævne.

Anden regel er, at du *ikke behøver omkranse argumenter i en parentes, hvis der kun er ét argument*. Da vores metode kun har ét argument, kan koden forkortes yderligere til:

```
Func<int, int> d = a => { return a * 2; };
```

Hvis der er mere end et argument, skal du omkranse med parenteser. Hvis der ikke er nogen argumenter overhovedet, skal du angive tomme parenteser.

Tredje regel er, at du *ikke behøver angive tuborgklammer, hvis der kun er én instruktion*. Yderligere *hvis denne instruktion returnerer et resultat, skal du ikke benytte return-kodeordet*. Så kan koden forkortes yderligere:

```
Func<int, int> d = a => a * 2;
```

Det giver præcis samme resultat som tidligere, men koden er forkortet kraftigt og er nem og hurtig at både læse og skrive (når du lige har lurt syntaksen).

Så tricket bag at læse et lambda-udtryk er at fjerne forkortelserne fra:

```
Func<int, int> d = a => a * 2;
```

til:

```
Func<int, int> d = (int a) => { return a * 2; };
```

Mere er der ikke i lambda-udtryk!

Det er blot en nem måde at skrive metoder på – især når metoden tager et enkelt argument og returnerer noget i første instruktion. Men alle metoder kan skrives som lambda-udtryk – du skal bare huske reglerne. Her er et par eksempler, du kan lege lidt med:

```
using System;

namespace Demo
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            Action d1 = Test1;
            Action<string, int> d2 = Test2;
            Func<bool> d3 = Test3;
            Func<int, bool, int> d4 = Test4;

            // ELLER
            Action d1a = () =>
            {
                Console.WriteLine("I Test1 på linie 1");
                Console.WriteLine("I Test1 på linie 2");
            };

            Action<string, int> d2a = (a, b) =>
                Console.WriteLine($"I Test2 med {a} og {b}");
            Func<bool> d3a = () => true;
            Func<int, bool, int> d4a = (a, b) =>
            {
                if (b)
                    return a;
                else
                    return a - 1;
            };
        }

        public static void Test1()
        {
            Console.WriteLine("I Test1 på linie 1");
        }
    }
}
```

```
        Console.WriteLine("I Test1 på linie 2");
    }

    public static void Test2(string a, int b)
    {
        Console.WriteLine($"I Test2 med {a} og {b}");
    }

    public static bool Test3()
    {
        return true;
    }

    public static int Test4(int a, bool b)
    {
        if (b)
            return a;
        else
            return a - 1;
    }
}
}
```

Prøv at skrive koden selv og følg oversættelsen fra almindelige metoder til lambda-udtryk. Den eneste måde at blive god til at kode med lambda-udtryk er at skrive noget kode!

Nu har du viden nok til at forstå Log-eksemplet på side 268, så prøv også at skrive denne kode selv.

Hvis du vil vide mere

Når du bliver lidt mere øvet, kan du måske lede efter yderligere information relateret til dette kapitel. Søg eksempelvis efter:

- Brug af *DynamicInvoke*-metoden
- Brug af *Expression-bodied members* i relation til lambda-udtryk.

Hændelser

Du har nu kendskab nok til at kunne forstå og tilføje den sidste medlemstype til klasser og strukturer – nemlig hændelser. Efter dette kapitel er pladen fuld, og du har forståelse for samtlige medlemstyper i klasser:

- felter
- egenskaber
- metoder
- hændelser.

En hændelse i en klasse giver mulighed for, at objekter af klassen kan afvikle en eller flere metoder, når en eller anden hændelse sker. Du kan se det lidt som at *give objekterne liv*. Og med tanke på forrige kapitel sker afvikling af en eller flere metoder naturligvis ved hjælp af delegates.

Hvis du kender til brugerflade-applikationer, kender du allerede til hændelser, for mange af den type applikationer er bygget op omkring hændelsesorienteret kode. I en Windows- eller mobil-applikation er funktionalitet typisk bundet op på hændelser som eksempelvis et klik på en knap, valg på en liste, minimering af et vindue og så videre.

Brug af delegates

Den samme funktionalitet kan du indbygge i dine klasser.

I vores terning kunne du eksempelvis vælge at tilføje en Rystet-hændelse. Den afvikler eventuel tilknyttet kode, hver gang terningen bliver rystet. Du kan vælge at kode det ved brug af almindelig delegates, som du lærte i forrige kapitel – eksempelvis:

```
class Terning
{
    public int Værdi { get; private set; }
    public Action Rystet;
```

```
public void Ryst() {
    this.Værdi = new Random().Next(1, 7);
    if (Rystet != null)
        Rystet.Invoke();
}

public Terning()
{
    this.Værdi = 1;
}
}
```

Bemærk, at terningen har et offentligt felt af typen Action (void uden argumenter), og i Ryst-metoden kontrolleres der, om den er forskellig fra null, og hvis den er, så bliver de metoder, som er gemt i delegate objektet, afviklet.

Nu kan terningen benyttes som følger:

```
Terning t = new Terning();
t.Rystet = () => Console.WriteLine("Terning er rystet!");

t.Ryst(); // Terning er rystet!
Console.WriteLine(t.Værdi); // [Tilfældig værdi]
```

Når terningen bliver rystet, så bliver metoden (her et lambda-udtryk) gemt i delegate objektet afviklet, men pointen er, at der kan afvikles præcis den kode, der ønskes. Det er ikke længere udvikleren af terningen, der tager stilling til, hvad der skal ske, men brugeren af terningen som styrer en eventuel funktionalitet.

Hvis du som udvikler af terningen ønsker at give lidt mere information, kan du blot ændre delegate definitionen. Hvad med at fortælle, hvad terningen blev rystet til og tidspunktet:

```
class Terning
{
    public int Værdi { get; private set; }
    public Action<int, DateTime> Rystet;
```

```
public void Ryst()
{
    this.Værdi = new Random().Next(1, 7);
    if (Rystet != null)
        Rystet.Invoke(this.Værdi, DateTime.Now);
}

public Terning()
{
    this.Værdi = 1;
}
}
```

Nu er der lidt flere argumenter på hændelsen, så den skal benyttes som:

```
Terning t = new Terning();
t.Rystet = (v, t) => Console.WriteLine(
    $"Terning er rystet kl {t.ToLongTimeString()} til en {v}'er");

t.Ryst(); // Terning er rystet kl 18:49:17 til en 4'er
Console.WriteLine(t.Værdi); // 4
```

Brug af event-kodeordet

Koden i tidligere afsnit virker fint, og du må gerne skrive dine hændelser på den måde. Der er dog intet, der over for brugeren af klassen indikerer, at der er tale om en hændelse (det er jo blot en delegate), og samtidig gør rene delegate objekter klassen noget følsom over for eventuelle null referencer, samt yderligere uhensigtsmæssigheder. Yderligere er offentlige felter som tidligere nævnt ikke et godt valg.

Derfor har Microsoft tilføjet et event-kodeord, som dels markerer en delegate som en decideret hændelse, og dels autogenererer kompileren noget kode til beskyttelse af feltet.

Det eneste, du skal gøre, er at tilføje event-kodeordet:

```
class Terning
```

```
{
    public int Værdi { get; private set; }
    public event Action<int, DateTime> Rystet;

    public void Ryst()
    {
        this.Værdi = new Random().Next(1, 7);
        if (Rystet != null)
            Rystet.Invoke(this.Værdi, DateTime.Now);
    }

    public Terning()
    {
        this.Værdi = 1;
    }
}
```

Nu opfatter kompileren, at der er tale om en hændelse, og som det fremgår af klassediagrammet, så ved Visual Studio også, at feltet ikke er et almindeligt felt (bemærk lynet).

Set fra brugerne af klassen er der også en enkelt ændring. For at kunne tilføje referencer til metoder, skal operatoren += benyttes:

```
using System;

namespace Demo
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            Terning t = new Terning();
            t.Rystet += (v, t) =>
                Console.WriteLine($"Terning er rystet " +
                    $"kl {t.ToLongTimeString()} til en {v}'er");

            t.Ryst(); // Terning er rystet kl 18:55:17 til en 4'er
            Console.WriteLine(t.Værdi); // 4
        }
    }
}
```

```
}

class Terning
{
    public int Værdi { get; private set; }
    public event Action<int, DateTime> Rystet;

    public void Ryst()
    {
        this.Værdi = new Random().Next(1, 7);
        if (Rystet != null)
            Rystet.Invoke(this.Værdi, DateTime.Now);
    }

    public Terning()
    {
        this.Værdi = 1;
    }
}
}
```

Du bestemmer selv, hvor mange hændelser en klasse skal have, og hvilken signatur brugeren af klassen skal bruge. I terningen kan du måske overveje en BlevSekser-hændelse, fordi mange spil har særligt fokus på seksere. Måske kan hændelsen også i kaldet give besked om, hvor mange seksere der er slået.

Her er mit forslag, men prøv selv:

```
class Terning
{
    public int Værdi { get; private set; }
    public event Action<int, DateTime> Rystet;
    public event Action<int> BlevSekser;
    private int antalSeksere = 0;

    public void Ryst()
    {
        this.Værdi = new Random().Next(1, 7);
        if (Rystet != null)
```

```
        Rystet.Invoke(this.Værdi, DateTime.Now);
    if (this.Værdi == 6)
        antalSeksere++;
    if (BlevSeksere != null)
        BlevSeksere.Invoke(antalSeksere);
}

public Terning()
{
    this.Værdi = 1;
}
}
```

Prøv at tilføje koden til en applikation og se om du kan bruge terningen.

Så en hændelse er altså blot en almindelig delegate, som er dekoreret med event-kodeordet. Om du binder lambda-udtryk eller konkrete metoder til delegate-objektet er underordnet, men signaturen skal passe jævnfør kapitlet om delegates.

EventHandler

I mange af Microsofts applikationstyper benyttes en speciel signatur på en hændelse – void metode, der som argument tager et Object (typisk kaldt sender), der repræsenterer det objekt, som kalder metoden, samt et EventArgs-objekt (typisk kaldt e), der repræsenterer eventuelle yderligere informationer om hændelsen.

Microsoft har derfor indbyggede delegates kaldet EventHandler og EventHandler<T>, der repræsenterer denne signatur. Den finder du blandt andet i forskellige Windows-applikationer, og du må også gerne bruge den selv.

Her er terningen igen med en tilrettet Ryst-hændelse:

```
using System;

namespace MinTest
{
```

```
class Program
{
    static void Main(string[] args)
    {
        Terning t = new Terning();
        t.Rystet += (s, e) => Console.WriteLine("Rystet");
        t.Ryst();
    }
}

class Terning
{
    public int Værdi { get; private set; }
    public event EventHandler Rystet;

    public void Ryst()
    {
        this.Værdi = new Random().Next(1, 7);
        if (Rystet != null)
            Rystet.Invoke(this, new EventArgs());
    }

    public Terning()
    {
        this.Værdi = 1;
    }
}
```

Hændelsen benytter den indbyggede EventHandler, som ikke giver yderligere informationer om hændelsen.

Hvis du ønsker at oplyse mere i hændelsen, må du skabe en klasse, der arver fra EventArgs, og udvide denne. Så kan du tilføje yderligere informationer.

Her benyttes en klasse, som indeholder information om, hvornår terningen er rystet:

```
using System;
```

```
namespace MinTest
{
    class Program
    {
        static void Main(string[] args)
        {
            Terning t = new Terning();
            t.Rystet += (s, e) =>
                Console.WriteLine("Rystet " + e.TidForRyst);
            t.Ryst();
        }
    }

    class Terning
    {
        public int Værdi { get; private set; }
        public event EventHandler<MinEventArgs> Rystet;

        public void Ryst()
        {
            this.Værdi = new Random().Next(1, 7);
            if (Rystet != null)
                Rystet.Invoke(this,
                    new MinEventArgs() { TidForRyst = DateTime.Now });
        }

        public Terning()
        {
            this.Værdi = 1;
        }
    }

    class MinEventArgs : EventArgs {
        public DateTime TidForRyst { get; set; }
    }
}
```


Bemærk klassen `MinEventArgs`, som kan indeholde yderligere informationer. Men det er helt op til dig, om du ønsker at benytte Microsofts `EventHandler` eller skabe din helt egen delegate.

Avancerede typer

I dette kapitel ser vi på lidt flere og lidt mere avancerede features relateret til typer. Hvis du er ny i programmering, kan du vælge at springe dette kapitel over, indtil du er blevet lidt mere erfaren.

Værdibaserede variabler med null-værdier

Som du tidligere har lært, findes der overordnet tre måder at skabe typer, der kan ligge til grund for instanser – klasser, poster og strukturer. Klasser og poster (som i virkeligheden er en autogenereret klasse – se senere i kapitlet) er referencebaserede, og strukturer er værdibaserede. Det har betydning for hvor i hukommelsen værdier placeres. Se i kapitlet om hukommelsesteori på side 207, hvis du vil have en opsummering.

At en variabeltype er referencebaseret betyder, at variabler ikke indeholder værdier, men referencer. Variablen kan dermed tildeles værdien *null* for at indikere, at den ikke indeholder nogen reference. Det kan værdibaserede variabler ikke, fordi de indeholder konkrete værdier.

Her er et eksempel:

```
int i = 10;    // det er ok - 10 er værdien
// i = null;   // fejl - en int kan ikke få værdien null
```

En int er det samme som `System.Int32`, og da det er en struct, kan den udelukkende indeholde værdier.

Men der kan være situationer, hvor det kunne være smart, at en værdibaseret type kan blive tildelt værdien *null*.

Det kan eksempelvis være en bool, der repræsenterer et valg, en bruger har taget på en brugerflade. Værdien kan være true eller false eller null, hvor null indikerer, at brugeren ikke har foretaget et valg.

Det kunne også være et felt i en klasse, som repræsenterer et felt i en database. I de fleste databaser kan man godt kan oprette et tal-felt, der kan tildeles en null-værdi, og det skal nogle gange kunne beskrives i kode.

Derfor har du mulighed at benytte en anden type i C#, som giver mulighed for null-værdi - den generiske System.Nullable-type.

Her er et eksempel, der benytter typen til at gøre en int *nullable*:

```
Nullable<int> i;
```

Nu kan i benyttes, som var det en int:

```
i = 10;  
Console.WriteLine(i);
```

Men fordi det nu er en Nullable<int>, kan den også tildeles null:

```
i = null;
```

og den har fået et par ekstra metoder:

```
i = 10;  
Console.WriteLine(i.HasValue);           // true  
Console.WriteLine(i.GetValueOrDefault()); // 10
```

Egenskaben HasValue fortæller om instansen har en værdi eller er null, og metoden GetValueOrDefault returnerer en værdi, hvis den findes, og ellers returnerer en default værdi (som for en int er 0).

For at gøre det nemt at benytte Nullable variabler, kan du også benytte tegnet ? ved erklæring. Således er

```
Nullable<int> i;
```

det samme som

```
int? i;
```

Det er jo noget nemmere at skrive, men giver præcis den samme type.

Her er et par andre eksempler:

```
int? a = 10;
bool? b = true;
double? c = 1000.33;
char? d = '*';

a = null;
b = null;
c = null;
d = null;

int? e = LægSammen(10, 10);
Console.WriteLine(e);      // 20
e = LægSammen(null, 10);
Console.WriteLine(e);      // null

int? LægSammen(int? a, int? b)
{
    if (!a.HasValue || !b.HasValue)
        return null;
    else
        return a + b;
}
```

int? er det samme som System.Nullable<int>
--

Bemærk, at typen kan benyttes på alle værdibaserede typer, i argumenter, returværdier og også i andre typer:

```
class Person
{
    public int PersonId { get; set; }
    public string Navn { get; set; }
    public bool? ErDansk { get; set; }
}
```

I klassen benyttes en nullable bool til at beskrive, om en person er dansk (true), ikke dansk (false) eller det ikke er angivet (null).

Referencebaserede variabler med null-værdier

Den lidt erfarne C# udvikler vil undre sig lidt over overskriften "Referencebaserede variabler med null-værdier", fordi alle referencebaserede variabler (variabler skabt med udgangspunkt i en klasse og ikke en struktur) jo netop kan indeholde en null-værdi – eksempelvis:

```
class Person
{
    public string Navn { get; set; }
    public string NavnMedStort()
    {
        return Navn.ToUpper();
    }
}
```

Her er tale om en ganske almindelig klasse med en autogenerated egenskab Navn, og en metode som returnerer navnet med store bogstaver. Den kunne eksempelvis bruges som:

```
Person p = new Person();
p.Navn = "Mikkel";
Console.WriteLine(p.NavnMedStort()); // MIKKEL
```

Det fungerer jo fint i dette eksempel, men klassen indeholder en mulig, og faktisk meget udbredt, fejlmulighed. Hvad tror du, der sker, når denne kode afvikles:

```
Person p = new Person();
Console.WriteLine(p.NavnMedStort());
```

Den fejler med et hult drøn med en `NullReferenceException`, hvilket i virkeligheden er meget logisk. Metoden `NavnMedStort` forsøger at

afvikle en metode på et objekt, som ikke eksisterer. Navn er jo aldrig tildelt en værdi og har dermed værdien null.

Det kan undgås på flere måder – måske ved at ændre metoden så den returnerer null, hvis Navn er lig med null:

```
class Person
{
    public string Navn { get; set; }
    public string NavnMedStort()
    {
        return Navn?.ToUpper();
    }
}
```

Kan du huske hvad Navn?.ToUpper() betyder?
Hvis ikke, så se side 95

Eller ved at gøre Set-delen af Navn-egenskaben privat, og værdien tildeles gennem en konstruktør:

```
class Person
{
    public string Navn { get; private set; }
    public string NavnMedStort()
    {
        return Navn.ToUpper();
    }
    public Person(string navn)
    {
        Navn = navn ?? "";
    }
}
```

Problemet med mulige null værdier kan løses på mange måder, men det vigtige er, at du er bevidst om eventuelt mulige kommende fejl. I en simpel klasse, er det nemt nok at overskue, men i en kodebase på

mange tusinder linjer kan man hurtigt overse et muligt kommende problem.

Heldigvis kan man bede kompileringen om at være opmærksom på eventuelle null-problemer og oplyse om dette i advarsler eller fejl. Du kan enten dekorere koden med direkte besked til kompileringen (# direktiver) som følger:

```
#nullable enable
class Person
{
    public string Navn { get; set; }
    public string NavnMedStort()
    {
        return Navn.ToUpper();
    }
}
#nullable restore
```

Eller ved at fortælle kompileringen i .csproj-filen (projektfilen), at du ønsker hele projektet kontrolleret for eventuelle fejl:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

Du bør altid anmode kompileringen om at foretage kontrol af eventuelle fejl,, der kan opstå ved null-værdier.

Under alle omstændigheder vil kompileringen nu fortælle, at der er et problem med Navn.

```

1 reference
class Person
{
    1 reference
    public string Navn { get; set; }
    1 reference
    public string NavnMedStort()
    {
        return Navn;
    }
}

```

CS8618: Non-nullable property 'Navn' must contain a non-null value when exiting constructor. Consider declaring the property as nullable.
Show potential fixes (Alt+Enter or Ctrl+.)

Figur 63 Advarsel om mulig null-fejl

Advarslen fortæller, at Navn bør tildes en værdi for at undgå eventuelle fejl – eksempelvis:

```

class Person
{
    public string Navn { get; set; } = "";
    public string NavnMedStort()
    {
        return Navn.ToUpper();
    }
}

```

Det løser umiddelbart problemet, men kun indtil du opretter en instans af person og så selv tildeler null til Navn. Men den fanger kompilatoren også:

```

0 references
static void Main(string[] args)
{
    Person p = new Person();
    p.Navn = null;
}

```

class System.String
Represents text as a sequence of UTF-16 code units.
CS8625: Cannot convert null literal to non-nullable reference type.
Show potential fixes (Alt+Enter or Ctrl+.)

Figur 64 Ny mulighed for fejl ved Null værdi

Du kan også vælge at fortælle kompilatoren, at Navn gerne må få værdien null, og det gøres på samme måde, som når du erklærer en variabel af en nullable struktur:

```

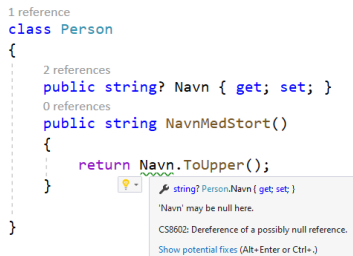
class Person
{

```



```
public string? Navn { get; set; }  
public string NavnMedStort()  
{  
    return Navn.ToUpper();  
}  
}
```

Nu ved kompilatoren, at Navn kan få en null-værdi, og det skaber en ny advarsel:



The screenshot shows the C# code from the previous block. A warning icon (yellow triangle with an exclamation mark) is placed over the `Navn` property access in the `return Navn.ToUpper();` line. A tooltip is displayed, showing the signature `string? Person.Navn { get; set; }` and the message: `'Navn' may be null here.` Below this, it says `CS8602: Dereference of a possibly null reference.` At the bottom of the tooltip, there is a link: `Show potential fixes (Alt+Enter or Ctrl+J)`.

Figur 65 Det er ikke nemt at snyde kompilatoren ved null check

Advarslen fortæller, at ToUpper-metoden kan risikere at fejle, fordi Navn kan have en null-værdi. For at få advarslen væk, er du nødt til at være meget konkret i koden – eksempelvis:

```
class Person  
{  
    public string? Navn { get; set; }  
    public string NavnMedStort()  
    {  
        return Navn == null ? "" : Navn.ToUpper();  
    }  
}
```

Du kan også fjerne advarslen ved at benytte udråbstegn-operatoren (som meget smukt også hedder ”null-forgiving operator”) som fortæller kompilatoren, at du tager ansvaret:

```
class Person  
{  
    public string? Navn { get; set; }  
}
```

```
public string NavnMedStart()  
{  
    return Navn!.ToUpper(); // Eget ansvar!!  
}  
}
```

Under alle omstændigheder tvinges du altså til at tage stilling til eventuelle null-relaterede problemer.

Hvis du ønsker at få deciderede kompileringsfejl (errors) og ikke blot advarsler (warnings), kan du tilføje `WarningsAsErrors` i `.csproj`-filen:

```
<Project Sdk="Microsoft.NET.Sdk">  
  <PropertyGroup>  
    <OutputType>Exe</OutputType>  
    <TargetFramework>net5.0</TargetFramework>  
    <Nullable>enable</Nullable>  
    <WarningsAsErrors>nullable</WarningsAsErrors>  
  </PropertyGroup>  
</Project>
```

Nu kan koden slet ikke kompilere, hvis kompileren opsnuser mulige kommende problemer relateret til null-værdier.

Tuples

C# understøtter som mange andre programmeringssprog en speciel datastruktur kaldet en **tuple**. Hvis du hører ordet "tuple" udtalt, vil du sikkert høre trykket lagt på både Tuple og tuPLE, men det helt korrekte er TUPle (tuh-pl)²¹.

Denne lidt specielle datastruktur giver en nem og effektiv mulighed for at opbevare værdier af forskellige typer, og du kan se det lidt som at slippe for at definere en klasse eller en struktur. De findes nemlig i to former – `System.Tuple<>` og `System.ValueTuple<>`. Førstnævnte er en referencebaseret type, og har været at finde i C# i flere versioner. Sidstnævnte er en værdibaseret type og blev tilføjet C# i version 7.

²¹ Søg efter "how to pronounce tuple" på Google hvis du vil høre ordet udtalt

System.Tuple

Som eksempel på brug af den referencebaserede Tuple-klasse må du forestille dig en type til opbevaring af data relateret til en person – eksempelvis:

```
class Person
{
    public string Navn { get; set; }
    public int Alder { get; set; }
    public bool ErDansk { get; set; }
}
```

Klassen er, som du kan se, en almindelig DTO (Data Transfer Object), og indeholder udelukkende data. Den kan ligge til grund for opbevaring af data relateret til en enkelt person:

```
Person p = new Person()
{
    Navn = "Mathias", Alder = 14, ErDansk = true
};
```

Eller en samling af personer:

```
List<Person> personer = new List<Person> {
    new Person() { Navn = "Mikkel", Alder = 17, ErDansk = true },
    new Person() { Navn = "Mathias", Alder = 14, ErDansk = true },
};
```

Eller som argument eller returværdi:

```
List<Person> personer = new List<Person> {
    new Person() { Navn = "Mikkel", Alder= 17, ErDansk = true },
    new Person() { Navn = "Mathias", Alder= 14, ErDansk = true },
};
```

```
Console.WriteLine(FindÆldstePerson(personer).Navn); // Mikkel
```

```
Person FindÆldstePerson(List<Person> personer)
{
    return personer.OrderBy(p => p.Alder).LastOrDefault();
}
```

```
}
```

Alle eksempler er helt korrekte og logiske, men kræver, at du opretter klassen Person. Ideen med tuples er, at samme kode kan skrives generisk uden definition af en klasse. Der kan angives op til otte forskellige typer i en tuple, og dermed kan vores person-klasse erstattes med følgende:

```
Tuple<string, int, bool> p;
```

Nu kan variablen p indeholde referencen til en (anonym) datastruktur bestående af en string, int og en bool:

```
Tuple<string, int, bool> p1;  
p1 = new Tuple<string, int, bool>("Mathias", 14, true);  
// eller  
Tuple<string, int, bool> p2 =  
    new Tuple<string, int, bool>("Mikkel", 17, true);
```

Da det jo ikke er navngivne egenskaber, må værdier hentes ud på en anden måde:

```
Tuple<string, int, bool> p =  
    new Tuple<string, int, bool>("Mikkel", 17, true);  
Console.WriteLine(p.Item1); // Mikkel  
Console.WriteLine(p.Item2); // 17  
Console.WriteLine(p.Item3); // true
```

Egenskaber kan nu tilgås som itemX, hvor X er den rækkefølge, de er erklæret.

Førnævnte metode, der finder den ældste person, kan nu omskrives til:

```
List<Tuple<string, int, bool>> personer =  
    new List<Tuple<string, int, bool>>() {  
        new Tuple<string, int, bool>("Mikkel", 17, true),  
        new Tuple<string, int, bool>("Mathias", 14, true)  
    };  
  
Console.WriteLine(FindÆldstePerson(personer).Item1); // Mikkel
```

```
Tuple<string, int, bool> FindÆldstePerson  
    (List<Tuple<string, int, bool>> personer)  
{  
    return personer.OrderBy(p => p.Item2).LastOrDefault();  
}
```

Og den fungerer præcis på samme måde. Forskellen er, at du ikke behøver erklære og benytte en decideret klasse. Koden er blevet generisk og kan benyttes på eksempelvis både en person, hund eller maskine.

System.ValueTuple

Endnu mere smart er den værdibaserede type (bemærk – ikke referencebaseret) ValueTuple. Den kan du sammenligne med en struktur, der består af værdier. Førnævnte Person klasse kunne omskrives til:

```
struct Person  
{  
    public string Navn { get; set; }  
    public int Alder { get; set; }  
    public bool ErDansk { get; set; }  
}
```

Det betyder, at værdier er placeret et andet sted i hukommelsen (stack) med (tit) tilhørende forbedring i performance, og eksempelvis kopiering af data resulterer i kopiering af værdier og ikke referencer:

```
Person p1 = new Person()  
    { Navn = "Mathias", Alder = 14, ErDansk = true };  
Person p2 = p1;  
p1.Navn = "Mikkel";  
Console.WriteLine(p1.Navn); // Mikkel  
Console.WriteLine(p2.Navn); // Mathias
```

Havde Person været en klasse ville p1 og p2 indeholde samme reference, og dermed udskrive 2 gange "Mikkel".

Hvis du ønsker en værdibaseret tuple kan du benytte `System.ValueTuple<>` med samme syntaks som `System.Tuple<>`:

```
ValueTuple<string, int, bool> p1
    = new ValueTuple<string, int, bool>("Mikkel", 17, true);
ValueTuple<string, int, bool> p2
    = new ValueTuple<string, int, bool>("Mathias", 14, true);
Console.WriteLine(p1.Item1);    // Mikkel
Console.WriteLine(p2.Item1);    // Mathias

List<ValueTuple<string, int, bool>> personer =
    new List<(string, int, bool)>
    {
        p1, p2
    };
```

Du kan sågar direkte sammenligne værdier:

```
ValueTuple<string, int, bool> p1
    = new ValueTuple<string, int, bool>("Mikkel", 17, true);
ValueTuple<string, int, bool> p2
    = new ValueTuple<string, int, bool>("Mathias", 14, true);
ValueTuple<string, int, bool> p3
    = new ValueTuple<string, int, bool>("Mikkel", 17, true);

Console.WriteLine(p1 == p2);    // false
Console.WriteLine(p1 == p3);    // true
```

Men `ValueTuple` indeholder også en del syntakssukker, som hurtigt kommer til at klistre på fingrene.

For det første kan værdier tildeles på en smart måde ved hjælp af parenteser:

```
ValueTuple<string, int, bool> p1
    = new ValueTuple<string, int, bool>("Mikkel", 17, true);
ValueTuple<string, int, bool> p2 = ("Mikkel", 17, true);
var p3 = ("Mikkel", 17, true);

Console.WriteLine(p1.Item1);    // Mikkel
Console.WriteLine(p2.Item1);    // Mikkel
```

```
Console.WriteLine(p3.Item1);    // Mikkel
```

Alle tre tildelinger giver det samme, og især

```
var p3 = ("Mikkel", 17, true);
```

er jo ret fiks, fordi kompilatoren er kvik nok til at udlede typer.

Yderligere kan egenskaber navngives således, at du ikke behøver tilgå egenskaber gennem ItemX:

```
var p1 = (Navn: "Mathias", Alder: 14, ErDansk: true);
```

```
Console.WriteLine(p1.Navn);    // Mathias
Console.WriteLine(p1.Alder);   // 14
Console.WriteLine(p1.ErDansk); // true
```

```
Console.WriteLine(p1.Item1);   // Mathias
Console.WriteLine(p1.Item2);   // 14
Console.WriteLine(p1.Item3);   // true
```

Som du kan se, kan egenskaber nu både tilgås gennem et logisk navn, samt gennem ItemX.

Til slut er her førnævnte metode FindÆldstePerson – nu med Value-Tuple:

```
List<(string, int, bool)> personer = new List<(string, int, bool)>()
{
    ("Mikkel", 17, true),
    ("Mathias", 14, true)
};
```

```
Console.WriteLine(FindÆldstePerson(personer).Navn);    // Mikkel
Console.WriteLine(FindÆldstePerson(personer).Item1);   // Mikkel
```

```
(string Navn, int Alder, bool ErDansk) FindÆldstePerson
(List<(string, int, bool)> personer)
{
    return personer.OrderBy(p => p.Item2).LastOrDefault();
}
```

Bemærk, at en tuple, der benyttes som returnværdi, navngives, som var det argumenter til en metode.

Du behøver ikke benytte tuples, men det kan gøre koden både generisk og nem at vedligeholde

Nedbrydning til variable

En anden feature som tit er relateret til tuples er nedbrydning (deconstruction). Det gør det muligt nemt at nedbryde en tuple til enkelte variable. Se eksempelvis denne tuple:

```
var a = ("Mikkel", 17);  
Console.WriteLine(a.Item1); // Mikkel  
Console.WriteLine(a.Item2); // 17
```

Den kan nedbrydes manuelt til variable som følger:

```
string navn = a.Item1;  
int alder = a.Item2;  
Console.WriteLine(navn);    // Mikkel  
Console.WriteLine(alder);   // 17
```

Men koden kan simplificeres en hel del – her på en enkelt linje:

```
(string navn, int alder) = a;  
Console.WriteLine(navn);    // Mikkel  
Console.WriteLine(alder);   // 17
```

og endnu nemmere:

```
(var navn, var alder) = a;  
Console.WriteLine(navn);    // Mikkel  
Console.WriteLine(alder);   // 17
```

Men nedbrydning er ikke bare relateret til tuples. Samme funktionalitet kan tilføjes dine egne klasser ved hjælp af en metode kaldet `Deconstruct`.

Her er et eksempel med klassen Person:

```
class Person
{
    public string Navn { get; set; }
    public int Alder { get; set; }
    public bool ErDansk { get; set; }

    public void Deconstruct(out string navn,
        out int alder, out bool erDansk)
    {
        navn = Navn;
        alder = Alder;
        erDansk = ErDansk;
    }
}
```

Bemærk, at metoden Deconstruct benytter såkaldte out-parametre, som blandt andet benyttes i avancerede metoder til at tildele værdier til variabler erklæret i den kaldende metode. Men her benyttes out-parametrene til automatisk nedbrydning:

```
Person p = new Person
{
    Navn = "Mathias",
    Alder = 14,
    ErDansk = true
};
Console.WriteLine(p.Navn);    // Mathias
Console.WriteLine(p.Alder);   // 14
Console.WriteLine(p.ErDansk); // true

// Nedbrydning
(var navn, var alder, var erDansk) = p;
Console.WriteLine(navn);      // Mathias
Console.WriteLine(alder);     // 14
Console.WriteLine(erDansk);   // true
```

Bemærk den nemme nedbrydning til enkelte variabler.

Metoden Deconstruct kan eventuelt overloades således, at nedbrydning kan ske med færre eller flere argumenter:

```
class Person
{
    public string Navn { get; set; }
    public int Alder { get; set; }
    public bool ErDansk { get; set; }

    public void Deconstruct(out string navn, out int alder,
                           out bool erDansk)
    {
        navn = Navn;
        alder = Alder;
        erDansk = ErDansk;
    }

    public void Deconstruct(out string navn, out int alder)
    {
        navn = Navn;
        alder = Alder;
    }
}
```

Nu kan klassen nedbrydes til både en string, int og bool samt en string og en int.

Overskrivning af operatorer

Der kan være situationer, hvor du ønsker at kunne sammenligne to objekter af klasser på forskellige måder. Umiddelbart giver C# kun mulighed for at kontrollere, om referencen er det samme. Med udgangspunkt i følgende klasse:

```
class Person
{
    public string Navn { get; set; }
    public int Alder { get; set; }
}
```

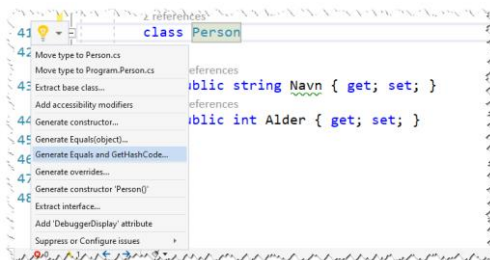
kan objekter sammenlignes som følger:

```
Person p1 = new Person
{
    Navn = "Mathias",
    Alder = 14,
};

Person p2 = new Person
{
    Navn = "Mathias",
    Alder = 14,
};

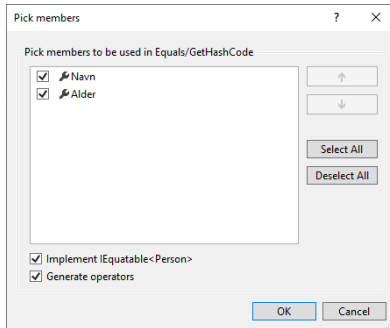
Console.WriteLine(p1 == p2);    // false
```

Selv om to objekter indeholder samme værdier, så vil en sammenligning fejle, fordi det er referencerne, der sammenlignes og ikke værdierne. Men det kan du ændre, hvis du ønsker, og du kan endda lade Visual Studio eller Visual Studio Code gøre det for dig.



Figur 66 Visual Studio kan hjælpe med at overskrive operatorer

Hvis du placerer cursoren på klassenavnet, vil du få mulighed for at få hjælp til at overskrive operatorer ved at vælge "Generate Equals and GetHashCode...". Det vil skabe denne dialogboks:



Figur 67 Autogenerering af kode til overskrivning

Et klik på OK-knappen vil ændre koden i Person-klassen til følgende:

```
class Person : IEquatable<Person?>
{
    public string Navn { get; set; }
    public int Alder { get; set; }

    public override bool Equals(object? obj)
    {
        return Equals(obj as Person);
    }

    public bool Equals(Person? other)
    {
        return other != null &&
            Navn == other.Navn &&
            Alder == other.Alder;
    }

    public override int GetHashCode()
    {
        return GetHashCode.Combine(Navn, Alder);
    }

    public static bool operator ==(Person? left, Person? right)
    {
        return EqualityComparer<Person>.Default.Equals(left, right);
    }
}
```

```
public static bool operator !=(Person? left, Person? right)
{
    return !(left == right);
}
}
```

Der tilføjes nu både en overskrivning af == og != operatorerne samt en overskrivning af metoderne Equals (som internt bruges til at sammenligne) og GetHashCode (som benyttes til at skabe en unik nøgle, der benyttes internt af runtime i eksempelvis samlinger af objekter). Det ligger uden for denne bogs rammer at komme nærmere ind på den autogenererede kode, men resultatet er meget tydeligt:

```
Person p1 = new Person
{
    Navn = "Mathias",
    Alder = 14,
};

Person p2 = new Person
{
    Navn = "Mathias",
    Alder = 14,
};

Console.WriteLine(p1 == p2);    // true
```

Nu sammenlignes på værdier og ikke på referencer, hvilket nogle gange kan være ret praktisk.

På samme måde kan du også vælge at overskrive eksempelvis < (mindre end) og > (større end) operatorerne. Det kunne være, at du ville have en forretningslogik, hvor det gav mening at sammenligne objekter på den måde. Her er eksempelvis en simpel terning:

```
class Terning
{
    public int Værdi { get; private set; }
    private Random random = new Random();
}
```

```
public Terning()
{
    Ryst();
}
public void Ryst()
{
    Værdi = random.Next(1, 7);
}
}
```

Og her er brugen af klassen med et par terninger:

```
Terning t1 = new Terning();
Console.WriteLine(t1.Værdi);    // Tilfældig værdi
Terning t2 = new Terning();
Console.WriteLine(t2.Værdi);    // Tilfældig værdi
```

Det kunne måske være smart, at kunne spørge om $t1 > t2$ eller omvendt, men det er jo kun dig, der ved, hvordan logikken skal være, og det kan du fortælle runtime ved at overskrive de to operatorer:

```
class Terning
{
    public int Værdi { get; private set; }
    private Random random = new Random();
    public Terning()
    {
        Ryst();
    }
    public void Ryst()
    {
        Værdi = random.Next(1, 7);
    }

    public static bool operator >(Terning? left, Terning? right)
    {
        return left.Værdi > right.Værdi;
    }

    public static bool operator <(Terning? left, Terning? right)
    {

```

```
        return left.Værdi < right.Værdi;
    }
}
```

Da klassen Terning ved hvordan instanser skal sammenlignes er følgende kode mulig:

```
Terning t1 = new Terning();
Terning t2 = new Terning();

bool mindre = t1 < t2;
bool større = t1 > t2;
```

Du kan naturligvis også vælge at overskrive andre operatorer – herunder == og != som før nævnt.

Søg på nettet efter ”operator overloading c#” for yderligere informationer.

Poster (records)

En post (record) i C# er en af de features, som du har svært ved at undvære, når du først lige har lært, hvad de kan bruges til.

I sin helt grundlæggende form kan du oprette brugen af record-kodeordet, som en hurtig måde at skabe en type til at opbevare data. Lad os antage, at du har behov for at opbevare data relateret til en person. Hvis du benytter en klasse eller en struktur, kunne du skrive kode som følger:

```
class Person
{
    public string Navn { get; init; }
    public int Alder { get; init; }
    public bool ErDansk { get; init; }

    public Person(string navn, int alder, bool erDansk)
    {
        Navn = navn;
        Alder = alder;
    }
}
```

```
        ErDansk = erDansk;  
    }  
}
```

Det er en simpel data-klasse, som kan benyttes som eksempelvis:

```
Person p1 = new Person("Mikkel", 17, true);
```

Bemærk, at klassen er immutable. Når egenskaber først er tildelt en værdi, kan de ikke tilrettes igen.

Du ser tit denne form for meget simple datatyper, som blot har til formål at transportere data.

Præcis samme type kan også oprettes som en post som følger:

```
record Person(string Navn, int Alder, bool ErDansk);
```

I stedet for 12-13 linjers kode kan du nøjes med én linje. Resten klarer kompileren for dig. Den vil nemlig sørge for at autogenerere en klasse med de nævnte egenskaber, en konstruktør og en del andre medlemmer.

Bemærk, at poster (records) er en C# 9 feature!

Oprettelse af et objekt sker på præcis samme måde som ved den førnævnte Person-klasse:

```
Person p1 = new Person("Mikkel", 17, true);
```

De autogenerede egenskaber Navn, Alder og ErDansk vil være init-egenskaber, og typen er dermed immutable:

```
Person p1 = new Person("Mikkel", 17, true);  
// p1.Navn = "Mathias"; // init egenskab - kan ikke tildeles en værdi
```

Men udover at der bliver autogenereret egenskaber og konstruktør bliver der ligeledes tilføjet andre medlemmer. Eksempelvis er ToString-

metoden overskrevet, så den returnerer en string, der beskriver værdierne:

```
Person p1 = new Person("Mikkel", 17, true);
Console.WriteLine(p1);
// Udskriver:
// Person { Navn = Mikkel, Alder = 17, ErDansk = True }
```

Hvis det er din egen klasse, skal du selv tilføje den kode. Ellers vil ToString-metoden blot returnere navnet på klassen.

Yderligere er der tilføjet kode, der overskriver != og == operatorerne samt metoderne GetHashCode og Equals. Det betyder, at objekter kan sammenlignes på værdier og ikke på referencer:

```
Person p1 = new Person("Mikkel", 17, true);
Person p2 = new Person("Mikkel", 17, true);
Console.WriteLine(p1 == p2);    // true
```

Hvis det er din egen klasse, skal du selv tilføje den kode (se side 311).

Der er ligeledes autogenereret en Deconstruct-metode således, at følgende kode er mulig:

```
Person p1 = new Person("Mikkel", 17, true);
(var navn, var alder, var erDansk) = p1;
Console.WriteLine(navn);    // Mikkel
```

Slutteligt er det nemt at få en kopi (husk – en post er immutabel) af objektet ved hjælp af with-kodeordet:

```
Person p1 = new Person("Mikkel", 17, true);
Console.WriteLine(p1);
// Person { Navn = Mikkel, Alder = 17, ErDansk = True }
Person p2 = p1 with { Alder = 18 };
Console.WriteLine(p2);
// Person { Navn = Mikkel, Alder = 18, ErDansk = True }
```

Bemærk, hvor nemt det er at kopiere data fra p1 til p2 og blot ændre alder.

Hvis du er nysgerrig på, hvordan den autogenererede klasse ser ud, så prøv at erklære en post på <https://sharplab.io>. Der kan du se både den dekompilede IL kode og tilhørende C# kode.

Du kan vælge at tilføje metoder og ekstra konstruktør samt egne egenskaber eller andre medlemmer, og en post kan også indgå i et arvehierarki med andre poster, men det ligger uden for denne bogs rammer at komme nærmere ind på det – bortset fra et simpelt eksempel:

```
record Person(string Navn, int Alder, bool ErDansk)
{
    public int EstimeretFødselsår()
    {
        return DateTime.Now.Year - this.Alder;
    }
}
```

Bemærk, at definitionen af typen nu benytter tuborgklammer for at gøre det muligt at tilføje en metode.

```
Person p1 = new Person("Mikkel", 17, true);
Console.WriteLine(p1);
// Person { Navn = Mikkel, Alder = 17, ErDansk = True }
Console.WriteLine(p1.EstimeretFødselsår());
// 2003
```

Helt overordnet skal du bare vide, at definition af en simpel datatype kan klares på en enkelt linje som en post, og samtidigt får du en masse yderligere funktionalitet foræret helt automatisk.

LINQ

LINQ er en af de features, du som udvikler vil få stor glæde af, men syntaks og teori kan virke lidt skræmmende, hvis du ikke har den helt store erfaring med C#. Det skyldes især brugen af funktionsorienteret kode og lambda-udtryk.

Men LINQ gør det muligt at skrive meget avanceret kode på meget få linjer og sikrer samtidig hurtig afvikling – så det er vigtigt, du bruger tid på at forstå, hvordan LINQ fungerer.

LINQ står for **L**anguage **I**Ntegrated **Q**uery, og giver overordnet en nem mulighed for at gennemløbe, filtrere, sortere og gruppere samlinger af data i arrays, lister og andre typer. Det er ligegyldigt, om der er tale om samlinger af simple strukturer som heltal eller komplekse samlinger af indbyggede eller egne objekter. I langt de fleste tilfælde vil du kunne benytte LINQ til at behandle samlingerne på mange forskellige måder.

Hvorfor LINQ

Bare for at du får en lille ide om, hvad LINQ stiller til rådighed så forestil dig en eksamensopgave, du skal løse, der lyder som følger:

Skriv en kode, som fra liste af heltal, samt et heltal der repræsenterer en maksimal værdi, finder en ny liste af tal, der består af unikke tal i sorteret form, og som er under den maksimale værdi. Som eksempel skal listen 2, 5, 5, 7, 3 returnere 2, 3, 5, hvis den maksimale værdi er 5.

En sådan opgave kunne en studerende uden kendskab til LINQ måske løse således:

```
using System.Collections.Generic;

namespace Demo
{
    internal class Program
    {
        private static void Main(string[] args)
```

```
{
    List<int> tal = new List<int> { 2, 5, 5, 7, 3 };
    List<int> res = FindUnikkeTal(tal, 5); // 2, 3, 5
}

private static List<int> FindUnikkeTal(List<int> tal, int max)
{
    List<int> tmp = new List<int>();
    for (int i = 0; i < tal.Count; i++)
    {
        if (tal[i] <= max && !tmp.Contains(tal[i]))
            tmp.Add(tal[i]);
    }
    tmp.Sort();
    return tmp;
}
}
```

Det kræver lidt kode til at gennemløbe, filtrere og sortere listen af tal, men løser jo opgaven.

En studerende med kendskab til LINQ ville måske løse opgaven således:

```
using System.Collections.Generic;
using System.Linq;

namespace Demo
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            List<int> tal = new List<int> { 2, 5, 5, 7, 3 };
            List<int> res = tal.Where(i => i <= 5)
                               .OrderBy(i => i).Distinct().ToList(); // 2, 3, 5
        }
    }
}
```

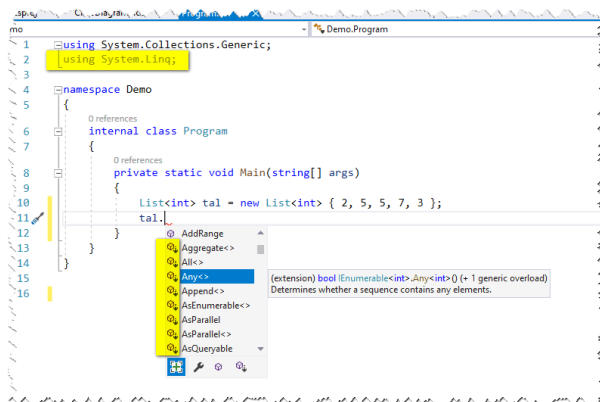
Hele filtreringen og sorteringen kan skrives på én linjes kode, men det kræver jo lidt viden om LINQ, delegates og lambda-udtryk.

Om LINQ

Hvis du søger på nettet efter LINQ, vil du muligvis finde flere versioner af teknologien, fordi den benyttes flere steder – eksempelvis i Entity Framework (kommunikation med databaser ved generering af SQL) og ved XML-serialisering.

Men den version, der er indbygget i C#, hedder *LINQ to Objects* og benyttes til at arbejde på samlinger af objekter af alle typer i hukommelsen. Du behøver altså ikke tilføje eksterne komponenter for at kunne benytte LINQ, og du kan benytte en almindelig .NET Core konsol-applikation til at afprøve dette kapitals eksempler.

Det eneste, du skal være opmærksom på, er, at LINQ primært fungerer ved hjælp af såkaldte extensionsmetoder. Det er metoder, som kan tilføjes eksisterende klasser, så de er nemme at finde og tilgå. Da LINQ arbejder på samlinger, er klasser som `System.Array`, `List<T>` med flere udvidet med nye metoder, men for at disse kan tilgås, skal du tilføje `System.Linq`-namespace som en `using`-instruktion.



Figur 68 Husk "using System.Linq"

Når du gør det, er der pludselig en masse nye metoder tilgængelige på de fleste typer af samlinger.

Syntaks

LINQ-udtryk kan skrives på to måder.

Enten kan du vælge at skrive udtryk i en SQL-lignende syntaks kaldet *query syntax*, eller benytte en funktionsorienteret syntaks kaldet *method syntax*. Sidstnævnte er den mest benyttede og i øvrigt også den nemmeste at læse og skrive, når du kender til delegates og lambda-udtryk, men du kan vælge den syntaks, der passer dig bedst. I dette kapitel vil du dog kun se den mest benyttede (metode) syntaks, men for en god ordens skyld er her brugen af begge syntakser til sammenligning.

Her er et simpelt udtryk vist i query syntaks:

```
List<int> tal = new List<int> { 3, 5, 5, 7, 2 };  
var res = from t in tal where t < 5 orderby t select t;
```

og her er det samme udtryk vist i method syntaks:

```
List<int> tal = new List<int> { 3, 5, 5, 7, 2 };  
var res = tal.Where(i => i < 5).OrderBy(i => i);
```

Begge udtryk kan afvikles til resultatet "2, 3" og kan læses som:

```
find alle tal mindre end 5 i sorteret form
```

men der er jo stor forskel i syntaksen.

Det første udtryk ligner SQL (Structured Query Language) bortset fra, at select-kodeordet er placeret til sidst. Det kan du vælge at bruge, hvis du ikke er bekendt med den funktionsorienterede tilgang til programmering.

Det sidste udtryk består af genkendelige metodekald, og de fleste metoder returnerer et nyt udtryk, som efterfølgende metoder kan arbejde videre på. Det gør koden meget kompakt og effektiv.

Som det fremgår, benytter syntaksen lambda-udtryk, og her skal du huske på reglerne jævnfør kapitlet om Delegates – du behøver ikke angive typer på argumenter og parenteser er ikke nødvendige, hvis der kun er ét argument. Hvis der kun er én instruktion, og denne returnerer en værdi, behøver du ikke angive tuborgklammer eller return-kodeordet. Så i virkeligheden kan følgende:

```
var res = tal.Where(i => i < 5).OrderBy(i => i);
```

skrives som:

```
var res = tal.Where((int i) => { return i < 5; }).OrderBy((int i) => {  
return i; });
```

eller for den sags skyld som:

```
Func<int, bool> w = i => i < 5;  
Func<int, int> o = i => i;  
var res = tal.Where(w).OrderBy(o);
```

eller pindet helt ud:

```
using System;  
using System.Collections.Generic;  
using System.Linq;
```

```
namespace Demo  
{  
    internal class Program  
    {  
        private static void Main(string[] args)  
        {  
            List<int> tal = new List<int> { 2, 5, 5, 7, 3 };  
            Func<int, bool> w = Find;  
            Func<int, int> o = Sort;  
            var res = tal.Where(w).OrderBy(o);  
        }  
  
        static bool Find(int i)  
        {  
            return i < 5;  
        }  
    }  
}
```

```
    }

    static int Sort(int i)
    {
        return i;
    }
}
```

Pointen er, at både Where og OrderBy-metoderne tager en reference til et delegate-objekt som argument, og kalder metoder fra denne ved afvikling af udtrykket. Hvordan du skaber delegate-objekterne, er helt op til dig, men som du sikkert har luret, er den forkortede anonyme lambda-version det nemmeste:

```
var res = tal.Where(i => i < 5).OrderBy(i => i);
```

Det er hurtigt at både læse og skrive.

Sen afvikling

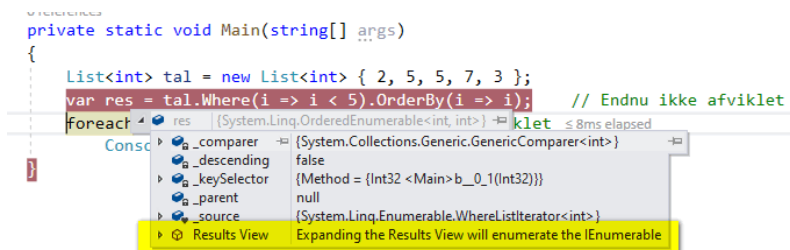
Når du skriver et LINQ-udtryk, er det reelt et *udtryk*. For at finde et resultat, skal udtrykket afvikles, og det sker automatisk, når du eksempelvis foretager et gennemløb:

```
List<int> tal = new List<int> { 2, 5, 5, 7, 3 };
var res = tal.Where(i => i < 5).OrderBy(i => i);
// Endnu ikke afviklet

foreach (var t in res) // HER bliver det afviklet
    Console.WriteLine(t);
```

Det kan virke lidt uforståeligt, men det, at der arbejdes med udtryk, giver en masse fordele relateret til datafriskhed, performance, hukommelsesforbrug samt mulighed for at andre teknologier kan pille et udtryk fra hinanden og kigge på de enkelte elementer. Det er blandt andet det, som LINQ to EF (Entity Framework) benytter til at skabe SQL kald mod en database ud fra LINQ-udtryk.

Det er måske ikke noget, du vil opdage, medmindre du i debuggeren vil se resultatet af et LINQ-udtryk, men her er det tydeligt, at LINQ ikke afvikler udtrykket, før du reelt beder om resultatet:



Figur 69 Sen afvikling i LINQ

Du vil også se det, når du kigger nærmere på returtypen af et LINQ-udtryk. Der kommer ikke en `List<int>` retur i førnævnte eksempel, men en liste af en speciel LINQ-type.

Du vil dog tit se, at udviklere gennemtvinger en afvikling og dermed en returværdi af en konkret og kendt type. Det kan ske ved eksempelvis at kalde metoderne `ToList` eller `ToArray`:

```
var res = tal.Where(i => i < 5).OrderBy(i => i).ToList();
```

Nu er udtrykket både skabt og afviklet, og variabelen `res` er af typen `List<int>`.

Men nogle gange giver det at arbejde med udtryk en masse muligheder – også relateret til genbrug:

```
List<int> tal = new List<int> { 2, 5, 5, 7, 3 };
```

```
var u = tal.Where(i => i < 5);
var res1 = u.OrderBy(i => i);
var res2 = u.OrderByDescending(i => i);
```

Her benyttes `Where`-udtrykket flere gange.

Til orientering er *sen afvikling* oversat fra *deferred execution*. Så har du nemmere ved at finde yderligere information på nettet, når du søger information om LINQ.

Filtrering

Når du arbejder med en samling af data, er filtrering nok en af de operationer, du har mest brug for, og her er Where-metoden tilgængelig.

Den tager som argument en reference til en delegate svarende til en såkaldt *predicate*. Det er en metode, der returnerer sand eller falsk, og som tager et enkelt argument svarende til typen på samlingen – altså en `Func<T, bool>` (der findes også en `Predicate<T>`, men den benyttes ikke så meget i nyere C# kode). Når LINQ gennemløber din samling, vil den for hvert element kalde din metode, og hvis metoden returnerer sand, kommer elementet med i resultatet – ellers forkastes det:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Demo
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            List<int> tal = new List<int> { 2, 5, 5, 4, 7, 3, 8, 1 };
            List<int> res;

            res = tal.Where(i => true).ToList();
            // alle
            Console.WriteLine(string.Join(' ', res));
            // 2 5 5 4 7 3 8 1

            res = tal.Where(i => false).ToList();
            // ingen
        }
    }
}
```

```
Console.WriteLine(string.Join(' ', res));
//

res = tal.Where(i => i < 7).ToList();
// < 7
Console.WriteLine(string.Join(' ', res));
// 2 5 5 4 3 1

res = tal.Where(i => i < 7 && i % 2 == 0).ToList();
// < 7 og lige tal
Console.WriteLine(string.Join(' ', res));
// 2 4

res = tal.Where(i => ErPrimtal(i)).ToList();
// primtal
Console.WriteLine(string.Join(' ', res));
// 2 5 5 7 3

res = tal.Where(ErPrimtal).ToList();
// primtal (igen - mere simpel syntaks)
Console.WriteLine(string.Join(' ', res));
// 2 5 5 7 3

}

static bool ErPrimtal(int nummer)
{
    if (nummer <= 1) return false;
    if (nummer == 2) return true;
    if (nummer % 2 == 0) return false;

    var til = (int)Math.Floor(Math.Sqrt(nummer));
    for (int tæller = 3; tæller <= til; tæller += 2)
        if (nummer % tæller == 0)
            return false;

    return true;
}
}
```

Som du kan se, kan du kode hvad som helst i metoden – den skal bare returnere sand eller falsk, og det behøver sågar ikke have noget med det argument, metoden kaldes med, at gøre.

Det er vigtigt at forstå, at LINQ arbejder med alle mulige typer af instanser – herunder også instanser af objekter fra frameworket eller dine egne. For nemt at kunne lege med LINQ på et mere komplekst objekt, kan du prøve følgende:

```
FileInfo[] filer = new DirectoryInfo(@"c:\tmp")
    .GetFiles("*.*", SearchOption.AllDirectories);
```

FileInfo-klassen kommer fra System.IO og repræsenterer en fil på disken. Ovennævnte kode vil skabe et array af FileInfo-objekter fra en mappe og dens underbiblioteker. De enkelte FileInfo-objekter har egenskaber som eksempelvis Name (string), FullName (string), Length (long), Exists (bool), Extension (string), CreationTime (DateTime) og mange flere. Et array af disse egner sig derfor godt til at prøve forskellige LINQ-metoder.

Hvad med eksempelvis:

```
FileInfo[] filer = new DirectoryInfo(@"c:\tmp")
    .GetFiles("*.*", SearchOption.AllDirectories);
FileInfo[] res;

// Alle filer under 1000 bytes
res = filer.Where(i => i.Length < 1000).ToArray();

// Alle txt filer
res = filer.Where(i => i.Extension == ".txt").ToArray();

// Alle readonly txt filer
res = filer.Where(i => i.Extension == ".txt" && i.IsReadOnly).ToArray();

// Alle filer hvor test indgår i navn
res = filer.Where(i => i.Name.Contains("test")).ToArray();
```

Bemærk, at typen `T` i `Func<T, bool>` nu er af typen `FileInfo`, og det er den, du kan spørge på i lambda-udtrykket.

Hvis du kun ønsker en fil ud af et udtryk, kan du bruge en af de mange *singleton*-metoder som eksempelvis `First` eller `FirstOrDefault`. Sidstnævnte returnerer en reference til et objekt eller null:

```
FileInfo[] filer = new DirectoryInfo(@"c:\tmp")
    .GetFiles("*.*", SearchOption.AllDirectories);
FileInfo res;

// Første fil der indeholder "test" i navnet
res = filer.FirstOrDefault(i => i.Name.Contains("test"));
if (res != null)
    Console.WriteLine(res.FullName);

// Første txt fil under 1000 bytes
res = filer.FirstOrDefault(i => i.Extension == ".txt"
    && i.Length < 1000);
if (res != null)
    Console.WriteLine(res.FullName);
```

Bemærk, at der nu returneres et enkelt objekt og ikke et array af objekter.

Metoden kan i øvrigt også kaldes uden argumenter direkte på resultatet af et udtryk:

```
FileInfo[] filer = new DirectoryInfo(@"c:\tmp")
    .GetFiles("*.*", SearchOption.AllDirectories);
FileInfo res;

// Første fil under 1000 bytes
res = filer.Where(i => i.Length < 1000).FirstOrDefault();
```

Der findes en del andre metoder, som returnerer et enkelt element, men `FirstOrDefault` er meget brugt. Du skal bare selv huske at kontrollere for en null værdi, som fortæller, at der ikke kunne findes nogle elementer.

Sortering

Sortering af data er naturligvis også meget brugt, og til det kan du benytte metoderne:

- `OrderBy`
- `OrderByDescending`
- `ThenBy`
- `ThenByDescending`.

Metoderne tager som argument en `Func<T, TKey>`, hvor `T` er typen og `TKey` er den egenskab, der ønskes sorteret efter.

Her er et par eksempler relateret til heltal, hvor `TKey` blot er værdien selv:

```
List<int> tal = new List<int> { 2, 5, 5, 7, 3 };
List<int> res;
res = tal.OrderBy(i => i).ToList();
res = tal.OrderByDescending(i => i).ToList();
```

Blot til orientering kan `TKey` jo være hvad som helst. Her sorteres efter et tilfældigt tal – svarende til at listen ”blandes”:

```
List<int> tal = new List<int> { 2, 5, 5, 7, 3 };
List<int> res;
Random rnd = new Random();
res = tal.OrderBy(i => rnd.Next(1, 1000)).ToList();
```

Sorteringsmetoderne kan naturligvis også benyttes på objekter – her eksempler på brug ved arrays af filer:

```
FileInfo[] filer = new DirectoryInfo(@"c:\tmp")
    .GetFiles("*.*", SearchOption.AllDirectories);

FileInfo[] res;

res = filer.OrderBy(i => i.Name).ToArray();
res = filer.OrderBy(i=>i.Length).ToArray();
res = filer.OrderBy(i => i.Extension).ThenBy(i => i.Name).ToArray();
```

```
res = filer.OrderByDescending(i => i.CreationTime).ThenBy(i =>
i.Name).ToArray();
```

Igen skal du huske, at TKey kan være hvad som helst.

Gruppering

Gruppering af data er ret komplekst, hvis du selv skal kode det, men metoden `GroupBy` gør det nemt. Den tager en `Func<T, TKey>` som argument, hvor `TKey` er den værdi, der skal grupperes ud fra, og den returnerer grupperede samlinger med en nøgle (Key), der repræsenterer den grupperede værdi:

```
List<int> tal = new List<int> { 2, 5, 5, 7, 3, 2 };
```

```
var res = tal.GroupBy(i => i);
```

```
/*
```

```
    res er nu en samling af gruppede samlinger af heltal
```

```
    og hver samling har en key
```

```
    [2, 2] key = 2
```

```
    [5, 5] key = 5
```

```
    [7]    key = 7
```

```
    [3]    key = 3
```

```
*/
```

Du kan løbe samlingen af samlinger igennem på flere måder – men her er den nemmeste:

```
List<int> tal = new List<int> { 2, 5, 5, 7, 3, 2 };
```

```
var res = tal.GroupBy(i => i);
```

```
foreach (var gruppe in res)
```

```
{
```

```
    Console.WriteLine(gruppe.Key);
```

```
    foreach (var t in gruppe)
```

```
    {
```

```
        Console.WriteLine("\t" + t);
```

```
    }
```

```
}
```

```
/*
```

```
2
```

```
2
```

```

        2
    5
        5
        5
    7
        7
    3
        3
*/
```

Du kan naturligvis gruppere samlinger af komplekse objekter også – prøv følgende på din egen maskine:

```
FileInfo[] filer = new DirectoryInfo(@"c:\tmp")
    .GetFiles("*.*", SearchOption.AllDirectories);

var res = filer.OrderBy(i => i.FullName).GroupBy(i =>
i.Extension).ToArray();
foreach (var gruppe in res)
{ Console.WriteLine(gruppe.Key);
  foreach (var fil in filer)
  {
      Console.WriteLine($"{fil.Name} ({fil.Length})");
  }
}
```

I koden vil filer blive grupperet og udskrevet efter Extension.

GroupBy-metoden kan bruges til mange ting. Prøv eksempelvis at tænke på, hvordan den kan bruges til at gruppere en samling terninger efter værdi, og på den måde finde ud af, om der er yatzy, fuldt hus, to ens, tre ens og så videre.

Projektion

Der kan tit være behov for at ændre resultatet fra et LINQ udtryk. Det kunne eksempelvis være noget så simpelt som en generel ændring af værdierne eller typen. Her kan metoden Select hjælpe dig. Se eksempelvis følgende:

```
List<int> tal = new List<int> { 2, 5, 5, 7, 3, 2 };
```



```
List<int> plusEn = tal.Select(i => i + 1).ToList();  
Console.WriteLine(string.Join(' ', plusEn)); // 3 6 6 8 4 3
```

Metoden `Select` tager som argument en `Func<T, TResult>` (i dette tilfælde er `T` og `TResult` en `int`), og baseret på resultatet af denne metode skabes en ny liste. Det er underordnet hvad metoden gør, eller hvilken type der returneres:

```
List<int> tal = new List<int> { 2, 5, 5, 7, 3, 2 };  
List<string> res = tal.Select(i => i.ToString().PadRight(3)).ToList();  
Console.WriteLine(string.Join("", res)); // 2 5 5 7 3 2
```

Her returneres en liste af formaterede strenge.

Metoden `Select` kan også benyttes til at ændre typen på komplekse typer. I følgende eksempel findes alle filer i sorteret form fra en mappe, men der returneres ikke en liste af `FileInfo`-objekter, men i stedet en liste af en defineret type som kunne se således ud:

```
class MinFil {  
    public string Navn { get; set; }  
    public long Længde { get; set; }  
}
```

Nu kan typen benyttes i `Select`-metoden som følger:

```
FileInfo[] filer = new DirectoryInfo(@"c:\tmp")  
    .GetFiles("*.*", SearchOption.AllDirectories);  
List<MinFil> res = filer.OrderBy(i => i.Name)  
    .Select(i => new MinFil { Navn = i.Name, Længde = i.Length })  
    .ToList();
```

Hvis du vælger at benytte en post i stedet:

```
record MinFil(string Navn, long Længde);
```

kan du skære lidt kode væk:

```
FileInfo[] filer = new DirectoryInfo(@"c:\tmp")  
    .GetFiles("*.*", SearchOption.AllDirectories);  
List<MinFil> res = filer.OrderBy(i => i.Name)
```

```
.Select(i => new MinFil(i.Name, i.Length))  
.ToList();
```

Du kan også vælge at returnere en tuple (se side 306) og helt droppe type definition:

```
FileInfo[] filer = new DirectoryInfo(@"c:\tmp")  
.GetFiles("*.*", SearchOption.AllDirectories);  
var res = filer.OrderBy(i => i.Name)  
.Select(i => (Navn: i.Name, Længde: i.Length))  
.ToList();
```

Eller du kan vælge at returnere et såkaldt anonymt objekt. Det ligger uden for rammerne i denne bog at beskrive anonyme typer dybere, men du kan se det lidt som en defineret tuple.

```
FileInfo[] filer = new DirectoryInfo(@"c:\tmp")  
.GetFiles("*.*", SearchOption.AllDirectories);  
var res = filer.OrderBy(i => i.Name)  
.Select(i => new { Navn = i.Name, Længde = i.Length })  
.ToList();
```

Nu består res af en liste af en anonym type med egenskaberne Navn (string) og Længde (long). Der er masser af information omkring anonyme typer i C# på nettet.

Begrænsning af resultat

Der findes en del metoder, som kan bruges til at begrænse resultatet af et udtryk.

Metoden First returnerer det første element, metoden Last det sidste, og metoden ElementAt returnerer et element ud fra et indeks. Der findes ligeledes metoder, som returnerer et element eller default-værdien – herunder FirstOrDefault og LastOrDefault:

```
FileInfo[] filer = new DirectoryInfo(@"c:\tmp")  
.GetFiles("*.*", SearchOption.AllDirectories);  
var fil = filer.FirstOrDefault();  
if (fil != null)
```

```
Console.WriteLine(fil.FullName);
```

Her findes den første fil i arrayet, og hvis arrayet ikke har nogen elementer, returneres null (default for klasser og dermed FileInfo).

Yderligere kan metoden Take returnere et antal, og metoden Skip vil springe elementer over.

Især de sidste metoder er meget brugbare:

```
FileInfo[] filer = new DirectoryInfo(@"c:\tmp")
    .GetFiles("*.*", SearchOption.AllDirectories);
```

```
// Hopper over de første 10 og tager de næste 10
var res = filer.Skip(10).Take(10).ToArray();
```

Som du kan se, kan kombinationen Skip og Take være meget brugbar.

Metoder relateret til tal

Hvis du har med tal at gøre (enten en samling af tal eller egenskaber som tal) findes der nogle specielle metoder, der kan være ret smarte.

Metoden Max finder den største værdi, Min den mindste, Sum en sum og Average finder gennemsnittet. Her eksempler på tal:

```
List<int> tal = new List<int> { 2, 5, 5, 7, 3, 2 };
Console.WriteLine(tal.Min());           // 2
Console.WriteLine(tal.Max());           // 7
Console.WriteLine(tal.Sum());            // 24
Console.WriteLine(tal.Average());       // 4
```

og her objekter:

```
FileInfo[] filer = new DirectoryInfo(@"c:\tmp")
    .GetFiles("*.*", SearchOption.AllDirectories);

Console.WriteLine(filer.Min(i=>i.Length));
// Mindste fil størrelse
Console.WriteLine(filer.Max(i => i.Length));
// Største fil størrelse
Console.WriteLine(filer.Average(i => i.Length));
```

```
// Gns af fil størrelse
```

Hvis du vil vide mere

Dette var kun et begrænset udsnit af de LINQ-metoder, der findes. Mange andre metoder som Select, Aggregate, Cast, GroupJoin, SelectMany og mange flere giver en masse muligheder for at arbejde med samlinger uden særlig meget kode.

Du kan finde en masse information om LINQ hos Microsoft på <https://docs.microsoft.com/en-us/dotnet/api/system.linq>, og der findes et hav af ressourcer på nettet om brugen af LINQ.

Det er vigtigt, at du lærer at kode med LINQ. Det vil gøre dig til en bedre og mere produktiv C# udvikler.

Asynkron programmering

Nu nærmer du dig afslutningen på bogen. Det sidste kapitel hører måske i virkeligheden ikke med i en bog om grundlæggende programmering, men jeg syntes, du bør kende til asynkron programmering – om ikke andet, så på et basalt niveau.

I traditionel programmering, herunder C#, er det normalt at afvikle en samling af instruktioner i et afviklingsrum, hvor ressourcer som proces-sorkraft og hukommelse fordeles ligeligt. Det kaldes også afvikling på en *tråd* (*thread* på engelsk), hvilket er et ret godt billede af, hvad der reelt sker – en samling af instruktioner placeret på en tråd, der bliver afviklet én for én.

I moderne programmering vil du dog falde over kode, som afvikles over flere tråde, hvilket er muligt, fordi computere i dag typisk har flere kerner, eller fordi operativsystemet fordeler alle instruktioner til afvikling. Det kaldes *multitrådet* kode (multithreading).

Optimering af tidskrævende metoder

Der kan være flere årsager til, at du ønsker kode afviklet over flere tråde. Den oplagte er, at du kan afvikle flere instruktioner på samme tid, og hvis du har blokke af instruktioner i din kode, som ikke er afhængig af hinanden, kan det være en fordel at afvikle dem alle på samme tid i stedet for at afvikle en ad gangen.

Se følgende eksempel:

```
using System;
using System.Diagnostics;
using System.Threading;

namespace MinTest
{
    class Program
    {
        static void Main(string[] args)
```

```
{

    Stopwatch s = new Stopwatch();
    s.Start();
    Console.WriteLine("Start");

    Console.WriteLine("Sleep 1");
    Thread.Sleep(500);
    Console.WriteLine("Sleep 2");
    Thread.Sleep(500);
    Console.WriteLine("Sleep 3");
    Thread.Sleep(500);

    Console.WriteLine("Slut");
    s.Stop();
    Console.WriteLine($"Tid: {s.ElapsedMilliseconds}");

}
}
```

Koden benytter et stopur til at måle, hvor lang tid det tager at afvikle tre simulerede operationer, der tager 500 ms i rækkefølge (Thread.Sleep kan bruges til at holde en pause). Hvis du afvikler koden, vil du konstatere, at det tager omkring 1,5 sekund at afvikle applikationen – hvilket jo ikke er super overraskende.

Men hvis operationerne ikke er afhængige af hinanden, kan de jo godt sættes i gang samtidig, og koden vil dermed have 4 tråde på et tidspunkt – nemlig hovedtråden og tre enkelte tråde til de simulerede operationer. Hvis du ser bort fra nye kodeord, du endnu ikke kender til, kunne det se således ud:

```
using System;
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks;

namespace MinTest
```

```
{
    class Program
    {
        static async Task Main(string[] args)
        {

            Stopwatch s = new Stopwatch();
            s.Start();
            Console.WriteLine("Start");

            Task t1 = Task.Run(() =>
            {
                Console.WriteLine("Sleep 1");
                Thread.Sleep(500);
            });

            Task t2 = Task.Run(() =>
            {
                Console.WriteLine("Sleep 2");
                Thread.Sleep(500);
            });

            Task t3 = Task.Run(() =>
            {
                Console.WriteLine("Sleep 3");
                Thread.Sleep(500);
            });

            await Task.WhenAll(t1, t2, t3);

            Console.WriteLine("Slut");
            s.Stop();
            Console.WriteLine($"Tid: {s.ElapsedMilliseconds}");

        }
    }
}
```

Som det fremgår, bruges der nye kodeord som `async` og `await` samt en ny klasse kaldet `Task`. Det kommer vi tilbage til senere i kapitlet, men

hvis du afvikler koden, vil du konstatere, at den afvikles på cirka 0,5 sekund – altså 1 sekund hurtigere end i eksemplet som kun benyttede hovedtråden. Og det giver jo fint mening. Hvis operativsystemet kan afvikle de tre operationer på samme tid (cirka), burde det tage 0,5 sekunder i alt at afvikle alle tre.

Men det kræver, at operationerne ikke er afhængige af hinanden på nogen måde, fordi vi i en rigtig applikation måske ikke har nogen anelse om, hvornår de er færdige. Runtime skal nok sørge for at vende tilbage til hovedtråden, men der er ingen garanti for hvornår.

Hvis du afvikler ovennævnte kode nogle gange, vil du se tydelige tegn på, at du har mistet noget af kontrollen. Hver sleep-operation udskriver, som det fremgår, noget på konsollen, og når du afvikler, bliver der udskrevet i vilkårlig rækkefølge.

Denne ene gang, du afvikler applikationen, kan det se således ud på konsollen:

```
Start
Sleep 1
Sleep 2
Sleep 3
Slut
Tid: 518
```

men næste gang:

```
Start
Sleep 3
Sleep 1
Sleep 2
Slut
Tid: 520
```

og så eventuelt:

```
Start
Sleep 1
Sleep 3
```


Sleep 2

Slut

Tid: 521

Faktum er, at du logisk nok ikke har kontrol over, hvornår en asynkron operation er færdig, og måske er det også ligegyldigt. Men hvis det har betydning, skal du i hvert fald være bevidst om det.

I programmeringsteori vil du falde over begreberne parallel og seriel afvikling. I det første eksempel afvikles kode serielt (en ad gangen), og i det andet eksempel afvikles koden parallelt (samtidigt).

Optimering af brugerflade applikationer

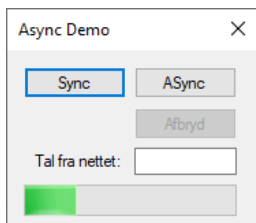
I applikationstyper, som benytter brugerflader i en eller anden form, er asynkron programmering typisk også brugt. Ikke nødvendigvis for at spare afviklingstid, men måske for at frigøre hovedtråden, som styrer brugerfladen.

I en applikationstype, som Windows Forms (WinForm) eller Windows Presentation Foundation (WPF), vil en længerevarende operation låse brugerfladen, så brugeren ikke kan flytte vinduer eller trykke på knapper. Dermed kan en operation heller ikke afbrydes. Hele applikationen låser simpelthen, indtil operationen er afsluttet.

Hvis du har lyst, kan du se et eksempel ved at hente min demo fra:

<https://github.com/devcronberg/async-winform-task-await>

På siden kan du finde en download-knap, hente hele projektet som en zip-fil og åbne og afvikle det gennem VS/VSC. Applikationen ser således ud:



Figur 70 Eksempel på en sync/async WinForm-applikation

Hvis du klikker på knapperne Sync eller ASync, bliver der hentet et tilfældigt tal fra nettet, og det tager et par sekunder. Applikationen har også en grøn linje nederst i vinduet, som løbende bliver fyldt ud.

Forskellen på de to knapper er naturligvis, at Sync-knappen henter tallet synkront og dermed låser brugerfladen. Når du klikker på knappen, kan du ikke flytte vinduet eller trykke på andre knapper. Den grønne linje bliver heller ikke tegnet mere. Først når tallet er hentet, er der liv i applikationen igen.

Hvis du klikker på ASync-knappen, kan du derimod flytte rundt på vinduet, mens tallet bliver hentet, og sågar klikke på Afbryd-knappen. Den grønne linje kører også uden problemer.

Hvis du kigger nærmere i koden, vil du også tydeligt kunne se, at Sync-knappen henter tallet via hovedtråden, mens ASync-knappen benytter en asynkron metode og dermed arbejder på en tråd for sig selv. Du kan eventuelt også se koden direkte på GitHub på:

<https://git.io/Jf3ue> (sync)

og:

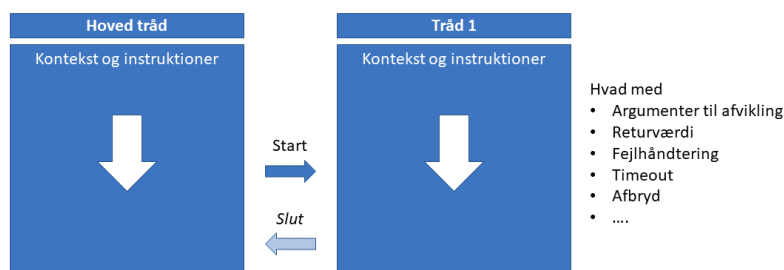
<https://git.io/Jf3uf> (async)

Så asynkron kode er meget benyttet i denne slags Windows-applikationer, men også i mobile applikationer, med en brugerflade, og i kode til afvikling på en webserver (ASP.NET Core). Sidstnævnte handler ikke om brugerfladen, fordi den dannes på serveren under alle omstændig-

heder, men om at give webserveren så meget luft som muligt til at håndtere forespørgsler fra andre klienter.

Udfordringer med asynkron kode

Nu ved du lidt om asynkron kode generelt, og det lyder jo oplagt, at du bare skal skrive så meget asynkron kode som muligt. Men i virkeligheden er det meget komplekst at jonglere mange tråde, fordi der er så mange ting, du skal tage hensyn til. Du skal jo regne med, at asynkron kode bliver afviklet på sin helt egen tråd, så du ved som udgangspunkt ikke, hvornår instruktionerne er afviklet, og har dermed svært ved at få fat på en eventuel returværdi. Det er også komplekst at sende og modtage data fra en eller flere tråde. Og fejlhåndtering er et kapitel for sig! Hvad hvis der sker en fejl i en af de tråde, du har sat i gang. Hvordan får du besked om det?



Figur 71 Udfordringerne ved flere tråde

I de seneste versioner af C# er der tilføjet forskellige muligheder for at gøre det nemmere at arbejde med asynkron kode. Det er især kodeordene `async` og `await` samt klasserne `Task` og `Task<T>`, vi skal se nærmere på.

En asynkron konsol-applikation

Denne bog benytter udelukkende konsol-applikationer for at kunne holde fokus på kode og ikke brugerflade, og som du lige har lært, så er asynkron kode brugt en del i applikationer, hvor man gerne vil holde en brugerflade levende. Det er jo ikke tilfældet i en konsolapplikation,

men derfor kan der godt være situationer, hvor du ønsker at afvikle asynkron kode i en konsolapplikation (se bare eksemplet i starten af kapitlet).

Men en konsolapplikation er som udgangspunkt ikke asynkron. Skabelonen ser således ud, som du ved:

```
using System;

namespace Demo
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Men hvis du ønsker at afvikle moderne asynkron C# kode skal Main-metoden ændres således, at den supporterer de nyeste features.

Derfor kan du fra C# 8 ændre skabelonen til dette i stedet:

```
using System;
using System.Threading.Tasks;

namespace Demo
{
    class Program
    {
        static async Task Main(string[] args)
        {
        }
    }
}
```

Den eneste ændring er, at Main nu ikke længere returnerer void, men en reference til et Task objekt, og at metoden er markeret med async-

kodeordet. Sidstnævnte er en forudsætning for, at du efterfølgende kan benytte await-kodeordet, som i den grad simplificerer asynkron kode. Faktisk vil Visual Studio give en advarsel om, at du benytter async-kodeordet uden at bruge await-kodeordet, fordi der bliver autogenerated og afviklet noget kode uden grund.

Task og Task<T>

Task og Task<T> er klasser, som repræsenterer en enkelt asynkron operation med en samling instruktioner, der afvikles på en tråd for sig selv. Klasserne har metoder til at starte og stoppe afvikling, samt en mulighed for at aflæse en status, som eksempelvis kan være:

- startet
- færdig
- fejlet
- afbrudt

Task betyder jo opgave på engelsk, så objektet repræsenterer en opgave, der på et tidspunkt enten er færdig, fejlet eller timet ud. I andre sprog kendes et Task objekt som et Promise-objekt, hvilket også er et meget godt navn – et løfte om en opgave der afvikles på et tidspunkt i fremtiden.

Task findes i to versioner:

- Task repræsenterer en operation uden returnværdi. Du kan se Task som en Action-delegate.
- Task<T> repræsenterer en operation med returnværdi (svarende til typen T). Du kan se Task<T> som en Func-delegate.

I grundlæggende C# modtager du Task og Task<T> objekter som returnværdier fra mange forskellige metodekald i frameworket. Det kunne eksempelvis være, når du henter eller skriver til en fil, henter data fra nettet via HTTP eller kommunikerer med en database. I den

mere avancerede C# kan du selv skabe metoder, der skaber og returnerer Task-objekter.

I dette kapitel vil vi dog fokusere på brugen af eksisterende metoder.

Om async og await

Du kan vælge at arbejde direkte med Task-objekterne og selv løbende spørge efter objektets status (er objektet startet, færdigt, fejlet med videre) og benytte objektets metoder til at få afviklet kode på givne tidspunkter, men du bør benytte kodeordene *async* og *await* for at simplificere koden så meget som muligt.

Kodeordet *async* skal placeres i metodedefinitionen og er et krav for at kunne benytte *await*-kodeordet. Når runtime ser *async*-kodeordet, vil der blive autogenereret kode, der repræsenterer en tilstandsmaskine, som vil holde styr på både status og kontekst. Koden, der genereres, er meget kompleks, men du behøver ikke forstå den. Det eneste du skal være bevidst om er, at *async* er et krav for *await*, og at *await* kodeordet vil afvente, at status i et Task-objekt ændrer sig til eksempelvis afsluttet eller fejlet. I mellemtiden fortsætter hovedtråden sin afvikling, og når Task objektet er færdigt, sørger den autogenerede tilstandsmaskine for, at kontekst reetableres og afvikling fortsætter fra instruktionen efter *await*-kodeordet. At hovedtråden fortsætter sin afvikling, giver ikke meget mening i en konsolapplikation som tidligere nævnt, men tænk eksempelvis på Windows-applikationen, hvor hovedtråden har til formål at holde applikationen levende.

Hvis der er tale om void-metoder, vil *await* blot afvente at afviklingen er færdig. Hvis der er tale om en metode, der returnerer en værdi, vil *await* kodeordet betyde, at du kan tildele returværdien direkte til en variabel og på den måde konvertere Task<T> objektet (hvor T er typen af returværdien) til en konkret værdi af typen T.

Så *await* kodeordet vil få din kode til at ligne synkron kode, selv om det i virkeligheden er asynkront. Samtidig vil kodeordene *async* og *await*

gøre fejlhåndtering meget simpel, for du skal blot benytte en try/catch struktur, som du hele tiden har gjort.

Forestil dig, at du i en Windows-applikation skal skrive koden, der henter et id fra en database, og benytter dette id til at hente en stor fil fra disken, og indholdet af denne fil skal sendes til en webserver. I synkron kode vil disse tre operationer skulle afvikles i rækkefølge, og i mellemtiden låser applikationen. I en asynkron applikation kan det skrives således:

```
try
{
    int id = await HentIdFraDatabase();
    string tekst = await HentTekstFraFil(id);
    await SendTekstTilServer(tekst);
}
catch (Exception ex)
{
    // log/besked ... der er sket en fejl
}
```

Metoderne er opfundet til lejligheden, men pointen er, at de tre metoder afvikles på en tråd for sig selv, og at afvikling automatisk afventer resultatet af den forrige metode. Samtidig fortsætter hovedtråden sin afvikling.

Uden brugen af await kodeordet ville det kræve meget kompleks kode at opnå samme funktionalitet, for slet ikke at tale om den oplevelse det ville være at fejlfinde med debuggeren i Visual Studio uden brug af await.

Hvis du skal lege lidt med asynkron kode og Task eller Task<T> objektet, kan du kigge på metoder på File-klassen under System.IO. Der finder du metoderne ReadAllTextAsync (Task) og WriteAllTextAsync (Task<string>).

Her er et eksempel på kode, der henter data fra en fil og gemmer i en anden fil:

```
using System;
using System.IO;
using System.Threading.Tasks;

namespace MinTest
{
    class Program
    {
        static async Task Main(string[] args)
        {
            try
            {
                string tekst = await File.
                    ReadAllTextAsync(@"y:\temp\data.txt");
                await File.AppendAllTextAsync
                    (@x:\temp\data.txt", tekst);
            }
            catch (Exception ex)
            {
                // log ... der er sket en fejl
            }
        }
    }
}
```

Prøv at se på dokumentationen til de to metoder. Der vil du opdage, at de returnerer en Task<T> og en Task:



```
static async Task Main(string[] args)
{
    try
    {
        string tekst = await File.ReadAllTextAsync(@"y:\temp\data.txt");
        await File.AppendAllTextAsync
            (@x:\temp\data.txt", tekst);
    }
    catch (Exception ex)
    {
        // log ... der er sket en fejl
    }
}
```

Tooltip text: (awaitable) Task<string> File.ReadAllTextAsync(string path, System.Threading.CancellationToken cancellationToken) Asynchronously opens a text file, reads all the text in the file, and then closes the file.

Figur 72 ReadAllTextAsync returnerer en Task<string>

Læg også mærke til at metoden er *awaitable*.

Da metoderne returnerer en Task, kunne du vælge at holde fast på en reference til denne og så afvente resultatet senere:

```
using System;
using System.IO;
using System.Threading.Tasks;

namespace MinTest
{
    class Program
    {
        static async Task Main(string[] args)
        {
            try
            {
                Task<string> t1 = File.
                    ReadAllTextAsync(@"y:\temp\data.txt");
                string tekst = await t1;
                Task t2 = File.AppendAllTextAsync
                    (@"x:\temp\data.txt", tekst);
                await t2;
            }
            catch (Exception ex)
            {
                // log ... der er sket en fejl
            }
        }
    }
}
```

Det koster jo lidt mere kode, men giver omvendt lidt flere muligheder.

Afvent flere Task-objekter

I stedet for at afvente et enkelt Task eller Task<T> objekt, kan du vælge at afvente flere på en gang. Det betyder jo, at et metodekald ikke kan være afhængigt af returværdien fra et andet, men der er masser af eksempler på kode, hvor du blot ønsker at afvikle kode på en tråd for

sig selv og enten er ligeglad med resultatet eller blot ønsker at afvente alle eller en enkelt.

Til det brug kan du bruge metoderne `WhenAll` eller `WhenAny`, som begge er statiske metoder på `Task`-klassen. De vil afvente, at alle eller et `Task`-objekt er færdigt, og returnerer i sig selv en `Task`.

Forestil dig, at du skal skrive kode, der gemmer tre store filer. Du vil sikkert kunne opnå en væsentlig performanceforbedring, hvis du kan sætte hver operation i gang på en tråd for sig selv, og så afvente at alle er færdige:

```
using System;
using System.IO;
using System.Threading.Tasks;

namespace MinTest
{
    class Program
    {
        static async Task Main(string[] args)
        {
            try
            {
                Task t1 = File.AppendAllTextAsync
                    (@":\temp\data1.txt", "tekst");
                Task t2 = File.AppendAllTextAsync
                    (@":\temp\data2.txt", "tekst");
                Task t3 = File.AppendAllTextAsync
                    (@":\temp\data3.txt", "tekst");
                await Task.WhenAll(t1, t2, t3);
            }
            catch (Exception ex)
            {
                // log ... der er sket en fejl
            }
        }
    }
}
```

Et andet eksempel kunne være en opgave, hvor du skal hente tekst fra tre filer, og afvente at alle tekster er hentet:

```
using System;
using System.IO;
using System.Threading.Tasks;

namespace MinTest
{
    class Program
    {
        static async Task Main(string[] args)
        {
            try
            {
                Task<string> t1 = File.ReadAllTextAsync
                    (@\"x:\temp\data1.txt\");
                Task<string> t2 = File.ReadAllTextAsync
                    (@\"x:\temp\data2.txt\");
                Task<string> t3 = File.ReadAllTextAsync
                    (@\"x:\temp\data3.txt\");
                string[] tekster = await Task.WhenAll(t1, t2, t3);
                // nu er tekster i et string array
            }
            catch (Exception ex)
            {
                // log ... der er sket en fejl
            }
        }
    }
}
```

Bemærk, at `WhenAll`-metoden returnerer et array af `Task<string>`, og når objektet afventer, kan indholdet i de tre filer aflæses i et array af strenge.

Hvis du vil vide mere

Dette var blot en introduktion til asynkron kode i C#. Der er masser af emner, du kan kaste dig over, når du nu har fået den grundlæggende

forståelse for brugen af async og await. Her tænker jeg især på emner som:

- Udvikling af dine egne asynkrone metoder der returnerer et Task-objekt
- Afbryd afvikling ved hjælp af et cancellation token
- Anden form for asynkron kode i C# – herunder brug af lock-kodeordet
- Asynkrone strømme af data.

Men nu kan du starte med emnerne i dette kapitel, og så begynde at skrive noget kode i en asynkron konsolapplikation.

Efterskrift

Nu er du igennem alle de emner, jeg har valgt at tage med i bogen her om grundlæggende C#. Og du burde have viden nok til at kunne deltage som udvikler i et udviklingsteam. Der er naturligvis mange flere avancerede emner end bogen rummer, og jeg har nævnt en del af dem i slutningen af hvert kapitel, så du kan finde mere information, hvis du har tid, lyst og overskud.

Det du mangler nu, er øvelse.

Skriv så meget kode som overhovedet muligt og helst sammen med andre med lidt mere erfaring. Hvis du ikke kan koble dig på et udviklingsprojekt, må du opfinde dine egne applikationer. Vi har alle skrevet utallige applikationer, som i virkeligheden var ret ubrugelige, men som blev skrevet for at få øvelse. Jeg nævner i flæng fra egen skuffe: kartotek over bøger, beregninger relateret til geometriske figurer, finansielle applikationer, yatzy-spil, vendespil, styring af vagtplan, styring af kilometerpenge og meget andet.

Held og lykke :)

Michell / Februar 2021

Figurer

Figur 1 Fra C++ til binær eksekverbar kode.....	6
Figur 2 Fra C# til mellemkode til binær eksekverbar kode.....	7
Figur 3 Eksempel på IL kode som er langt mere kompleks end C#	7
Figur 4 Konsol applikation.....	9
Figur 5 Windows Forms applikation.....	10
Figur 6 WPF applikation	10
Figur 7 Installation af Visual Studio 2019	15
Figur 8 Installation af Visual Studio 2019 (.NET 5)	16
Figur 9 Profil ved start af Visual Studio	16
Figur 10 Valg af settings og tema	17
Figur 11 Automatisk linjenummerering i Visual Studio	22
Figur 12 Typer i ét namespace	27
Figur 13 Typer i hvert sit namespace	28
Figur 14 Namespace i namespace	29
Figur 15 C# er et typestærkt sprog.....	33
Figur 16 En konsol-applikation.....	35
Figur 17 En konsol-applikation i Visual Studio	36
Figur 18 Klik på projektfilen for at tilrette runtime	37
Figur 19 En konsol-applikation i Visual Studio Code (på Windows)	38
Figur 20 En C# konsol-applikation i Visual Studio Code	38
Figur 21 Udskrift på konsol med Console.WriteLine	45
Figur 22 De mest benyttede vinduer i VS (version 16.8.3).....	47
Figur 23 De mest benyttede vinduer i VSC (version 1.52.1).....	48
Figur 24 Intellisense i VS (også tilgængeligt i VCS)	49
Figur 25 Machine Learning i VS.....	50
Figur 26 Brug af en snippet i Visual Studio (1)	51
Figur 27 Brug af en snippet i Visual Studio (2)	51
Figur 28 Hjælp i Visual Studio (pære-ikon).....	54
Figur 29 Hjælp i Visual Studio (skrue-nøgle-ikon)	55
Figur 30 Breakpoint i Visual Studio	56
Figur 31 Visual Studio i breakmode.....	57
Figur 32 Visual Studio Code i breakmode	58
Figur 33 Stop breakmode (her VS)	58
Figur 34 Hukommelse	60
Figur 35 Implicit datakonvertering.....	72
Figur 36 EksPLICIT datakonvertering.....	73
Figur 37 AddDays-metoden returnerer en ny DateTime	80
Figur 38 En int (System.Int32) er en struct	93
Figur 39 En string er en klasse.....	94

Figur 40 string er en reference-variabel.....	94
Figur 41 Konstant i Visual Studio.....	100
Figur 42 Brug af en enum i Visual Studio	103
Figur 43 If-strukturen	105
Figur 44 En løkke-struktur.....	112
Figur 45 Kald af metoder med samme navn.	137
Figur 46 Billede af den første bug	138
Figur 47 Exception i Visual Studio	141
Figur 48 Informationer om en Exception	141
Figur 49 Lille udsnit af forskellige Exception-klasser	145
Figur 50 Array over nedbør	153
Figur 51 Array af heltal.....	155
Figur 52 Array af bool.....	155
Figur 53 Et endimensionelt array i et regneark.....	159
Figur 54 Et todimensionelt array i et regneark	160
Figur 55 En OOP-terning	172
Figur 56 Metoder fra System.Object.....	187
Figur 57 Hukommelse tildelt til instruktioner og data	203
Figur 58 Et simpelt arvehierarki	229
Figur 59 Et arvehierarki med dyr.....	230
Figur 60 Et klassehierarki relateret til kurser	231
Figur 61 System.Object.	242
Figur 62 Alt arver fra System.Object	243
Figur 63 Advarsel om mulig null-fejl	301
Figur 64 Ny mulighed for fejl ved Null værdi	301
Figur 65 Det er ikke nemt at snyde kompileren ved null check	302
Figur 66 Visual Studio kan hjælpe med at overskrive operatorer	312
Figur 67 Autogenerering af kode til overskrivning.....	313
Figur 68 Husk "using System.Linq"	322
Figur 69 Sen afvikling i LINQ.....	326
Figur 70 Eksempel på en sync/async WinForm-applikation	343
Figur 71 Udfordringerne ved flere tråde	344
Figur 72 ReadAllTextAsync returnerer en Task<string>	349

Tabeller

Tabel 1 C# versioner.....	3
Tabel 2 De mest benyttede snippets i VS/VSC	52
Tabel 3 Nogle af de mange genveje i VS	53
Tabel 4 Nogle af de mange genveje i Visual Studio Code	54
Tabel 5 Styring af afvikling under breakmode.....	58
Tabel 6 Heltal i C#	65
Tabel 7 Kommatal i C#	68
Tabel 8 Operatorer relateret til tal.....	69
Tabel 9 Tegn til formatering af tal.....	71
Tabel 10 Kode for datatype til brug i konstanter	74
Tabel 11 Type til at indeholde en sand eller falsk værdi	75
Tabel 12 Logiske operatorer	77
Tabel 13 System.DateTime	78
Tabel 14 Tegn til formatering af dato og tid	83
Tabel 15 System.Char.....	87
Tabel 16 System.String.....	88
Tabel 17 Operatorer relateret til System.String	89
Tabel 18 Specialtegn	91
Tabel 19 Operatorer relateret til null.....	95
Tabel 20 Medlemmer på Exception-klassen.	142
Tabel 21 Medlemstyper	174
Tabel 22 Synlighed	175
Tabel 23 Synlighed i et arvehierarki	236
Tabel 24 Indbyggede delegates	275

Specificeret indholdsfortegnelse

Indledning	2
Introduktion til .NET	3
.NET	4
Generelt typebibliotek	5
Kompilering af kode i .NET	6
Afvikling af .NET applikationer	8
Type af applikationer	8
NuGet	13
Udviklingsmiljøer	14
Valg af udviklingsmiljø	14
Installation af Visual Studio	15
Installation af Visual Studio Code	17
Introduktion til C#	19
Tuborgklammer	19
Semikolon afslutter en instruktion	20
Kommentarer	20
Store og små bogstaver	21
Linjenumre	21
Programmeringsparadigmer	22
Alt er typer i C#	24
Hierarki af typer	26
Helt grundlæggende om tekster	30
Helt grundlæggende om metoder	31
C# er typestærkt	32
C# er typesikkert	33
En konsol-applikation i C#	35
En konsol-applikation i Visual Studio	35
En konsol-applikation i Visual Studio Code	38
Skabelon til en konsol-applikation	39
En C# løsning	41
Using	42
Brug af Console-klassen	44
Hvis du vil vide mere	46
Praktisk brug af VS og VSC	47

En tur rundt i Visual Studio	47
En tur rundt i Visual Studio Code	48
IntelliSense	49
Snippets	51
Genvejstaster	52
Hjælp fra VS og VSC	54
Debugging	55
Kompilering af kode	59
Hvis du vil vide mere	59
Simple variabler	60
Hukommelse	60
Variabelnavne	62
Virkefelter	63
Heltal	65
Kommatal	68
Operatorer relateret til tal	69
Formatering af tal	71
Typekonvertering af tal	72
Sand eller falsk	75
Boolske operatorer	76
Dato	78
Tid	81
Formatering af dato og tid	83
Typekonvertering af dato og tid	85
Hvis du vil vide mere	86
Tekster	87
Tegn	87
Streng	88
Operatorer relateret til streng	89
Metoder relateret til streng	90
Specialtegn	91
Interpolerede streng	92
Streng er en reference-type	93
Streng er immutable	96
Hvis du vil vide mere	98
Konstanter	99
Simple konstanter	99
Indbyggede simple konstanter	99
Relaterede konstanter	100
Brug af enum	102
Typekonvertering	103
Indbyggede relaterede konstanter	103
Programflow	105

If-betingelsen	105
Switch-betingelsen	108
For-løkken	112
Do- og while-løkken	113
Brug af continue	115
Brug af break	115
Løkker i løkker	115
ForEach-løkken	116
Goto	117
Metoder	119
Hvorfor benytte metoder	119
Programpointeren og metoder	121
Definition af metoder	124
Placering af metoder	125
Metoder der ikke returnerer en værdi	126
Metoder der returnerer en værdi	128
Argumenter og variabler	130
Argumenter som ikke behøves angivet	132
Navngivne argumenter	133
Metoder med samme navn (overload)	134
Hvis du vil vide mere	137
Fejlhåndtering	138
Eksempler på fejl	139
Brug af try/catch	142
Flere catch-blokke	144
Brug af finally	146
Skab dine egne fejl	147
Log	150
Hvis du vil vide mere	152
Arrays	153
Oprettelse af et array	154
Tilgang til elementer	156
Gennemløb af arrays	156
Manipulering af arrays	157
Flere dimensioner	159
Hvis du vil vide mere	161
Samlinger	162
Generiske samlinger	162
Liste	163
Stak	166
Kø	167
Nøgle og værdi	168
Hvis du vil vide mere	169

Dine egne typer.....	170
Hvad er objektorienteret programmering.....	170
Terningen	172
Medlemstyper	173
Synlighed	174
Felter	175
Egenskaber	176
Metoder.....	177
Hændelser	177
Konstruktører	178
Destruktør	179
Eksempler på brug af medlemstyper	179
Konklusion	181
Klasser	183
Oprettelse af objekter.....	185
Medlemmer fra System.Objekt.....	186
Felter	187
Offentlige data.....	189
Brug af this.....	190
Metoder.....	191
Standard konstruktør.....	193
Brugerdefineret konstruktør.....	196
Genbrug af konstruktører	198
Hvis du vil vide mere.....	200
Grundlæggende hukommelsesteori	201
Diagram over hukommelsen	203
Stack	203
Værdibaserede og referencebaserede typer	207
Argumenter til metoder	211
Hvis du vil vide mere.....	213
Indkapsling	214
Indkapsling af felter	215
Egenskaber	218
Komplette egenskaber.....	219
Initialisering	222
Init-egenskab	223
Automatiske egenskaber	225
Hvis du vil vide mere.....	227
Arv.....	228
Hierarki af klasser	228
Brug af arv	232
Tilgang til medlemmer	234
Konstruktører	236

Brug af base-kodeordet	238
En ludoterning	239
Hvis du vil vide mere.....	240
Polymorfi.....	241
System.Object.....	241
Virtuelle metoder	244
Base	248
Abstrakte metoder.....	248
Mor kan altid pege på sine børn	250
Typecheck og typekonvertering.....	252
Virtuelle metoder (igen)	253
Polymorfi og typestærke samlinger	254
Hvis du vil vide mere.....	256
Interface.....	257
Definition af et interface.....	258
Implementering af et interface	258
IComparable <T>.....	260
Afkobling.....	262
Hvis du vil vide mere.....	265
Delegates	266
En terning med log.....	266
Den endelige terning	268
Hvad er en delegate.....	271
Indbyggede delegates.....	275
Flere referencer i en delegate.....	279
Lambda-udtryk	280
Hvis du vil vide mere.....	285
Hændelser.....	286
Brug af delegates	286
Brug af event-kodeordet.....	288
EventHandler	291
Avancerede typer.....	295
Værdibaserede variabler med null-værdier	295
Referencebaserede variabler med null-værdier.....	298
Tuples	303
System.Tuple	304
System.ValueTuple	306
Nedbrydning til variabler	309
Overskrivning af operatorer	311
Poster (records)	316
LINQ	320
Hvorfor LINQ.....	320
Om LINQ	322

Syntaks	323
Sen afvikling.....	325
Filtrering	327
Sortering.....	331
Gruppering	332
Projektion	333
Begrænsning af resultat.....	335
Metoder relateret til tal.....	336
Hvis du vil vide mere.....	337
Asynkron programmering	338
Optimering af tidskrævende metoder	338
Optimering af brugerflade applikationer	342
Udfordringer med asynkron kode	344
En asynkron konsol-applikation	344
Task og Task<T>	346
Om async og await.....	347
Afvent flere Task-objekter	350
Hvis du vil vide mere.....	352
Efterskrift	354
Figurer	355
Tabeller	357
Specificeret indholdsfortegnelse	358
Indeks.....	364

Indeks

- .csproj-fil; 41
- .sln-fil; 41
- =>, Lambda; 281
- abstrakte metoder; 248
- Action; 275
- Adams, Douglas; 88
- afkobling, data; 262
- afkobling, funktionalitet; 271
- afrundingsfejl; 68
- agil udvikling; 172
- allokeret, hukommelse; 60
- Android; 11
- anonyme typer; 335
- argumenter; 31; 125
- argumenter, default værdier; 132
- argumenter, metoder; 211
- argumenter, navngivne; 133
- array; 153; 162
- arv; 228
- as; 253
- ASP.NET Core; 343
- ASP.NET MVC; 11
- ASP.NET Web Pages.; 11
- async; 347
- Average, LINQ; 336
- await; 347
- barn, arv; 232
- base; 248
- base, arv; 238
- Beep, Console; 45
- betingelse; 105
- Blazor; 12
- bool; 75
- Boole, George; 76
- Borland; 3
- break; 109; 115
- breakmode; 57
- breakpoint; 56
- bugs; 138
- Build project; 59
- Build solution; 59
- byte; 65
- C++; 3
- Call Stack-vinduet; 57
- camelCasing*; 63
- catch; 144
- char; 87
- checked; 70
- Church, Alonzo; 23; 280
- class; 170; 183
- code; 38
- Console App; 35
- Console-klassen; 44
- const; 99
- constructor; 178; 193; 196
- constructor based dependency injection; 265
- continue; 115
- datastruktur; 162
- dato; 78
- debugging; 55
- decimal; 68
- deconstruction; 309
- Deep Thought; 88
- defaultværdi; 66
- deferred execution; 327
- delegate; 266; 271; 286
- delegate, indbyggede; 275
- Delphi; 3
- dependency injection; 262
- desktop applikation; 9
- destructor; 179
- destruktør; 179
- Diagnostics tools-vinduet; 57
- Dictionary<TKey, T>; 168
- dimension, array; 159
- direktiver; 300
- dotnet new; 38
- double; 68
- DTO (Data Transfer Object); 304

EF; 12
egenskaber; 176; 218
egenskaber, automatiske; 225
egenskaber, komplette; 219
entitet; 171
Entity Framework; 12; 322
enum; 100
Equals; 314
Error List-vinduet; 48
errors (null); 303
event; 288
EventArgs; 291
EventHandler; 291
events; 177
exception; 139
extensionsmetoder; 322
falsk; 75
farveskema; 16
fejl; 138
felter; 175; 187
fields; 176
FIFO; 167
filtrering, LINQ; 327
finally; 146
firkantede parenteser; 154
First, LINQ; 335
FirstOrDefault, LINQ; 330
flerformet; 241
float; 68
floating-point; 68
ForEach; 116
forgrening; 107
formatering af kode; 65
formatering, dato og tid; 83
formatering, tal; 71
Func; 275
funktioner; 119
Funktionsorienteret programmering;
23
funktionspointer, delegate; 266
garbage collector; 207
generics; 162
gennemløb, arrays; 156
genvejstaster; 52
GetHashCode; 314
get-metode; 215
goto; 117; 120
GroupBy, LINQ; 332
gRPC; 12
gruppering, LINQ; 332
Hamilton, Margaret; 22
handled exception; 139
heap; 203; 213
Hejlsberg, Anders; 3
heltal; 65
hierarki; 26
hierarki, arv; 241
hierarki, klasse; 228
hukommelse; 60
hukommelsesdiagram; 203
hukommelsesteori; 201
hændelser; 177; 286
IComparable; 261
if, betingelse; 105
immutable data; 81; 96
Imperative programming; 22
implementering, interface; 258
indeks, array; 153
indkapsling; 189; 214
int; 65
IntelliCode; 18; 50
IntelliSense; 49
interface; 257
Intermediate Language; 6
internal; 183
interpolerede strenge; 92
iOS; 11
is; 253
Java; 3
JavaScript; 3
kildekode; 14; 19
klasse; 170; 183; 201
kommandoprompt; 38
kommatal; 68
Kommentar; 20
kompiler; 6
Kompilering; 59
konsol applikation; 35
konsolapplikation, asynkron; 344
konstant; 99

- konstant, relaterede; 100
- konstruktør; 178; 193; 196
- konstruktør, arv; 236
- kultur; 72
- kø; 167
- Lambdakalkyle; 23
- lambdaudtryk; 271; 280; 324
- Language INtegrated Query; 320
- LIFO; 166
- linjenummerering; 21
- LINQ; 320
- Linux; 14; 17
- List<T>; 163
- Locals-vinduet; 57
- Log; 150
- Log4Net; 150
- long; 65
- ludoterning; 239
- løkke, arrays; 156
- løkke, do; 113
- løkke, for; 112
- løkke, while; 114
- løkker i løkker; 115
- løsning*; 41
- Mac; 14; 17
- Main-metode; 40
- MAUI; 11
- Max, LINQ; 336
- medlemstyper; 173
- memory leaks; 279
- methods; 177
- metode, anonym; 281
- metode, argumenter; 125
- metode, definition; 124
- metode, overload; 137
- metode, virkefelt; 131
- metoder; 119; 177
- metoder, signatur; 134
- Min, LINQ; 336
- mor, arv; 232
- multitrådet; 338
- mutable data; 81
- MVC; 11
- namespace; 26
- navngivningsstandard; 63
- nedarvning; 228
- nedbrydning; 309
- NLog; 150
- NuGet; 5; 13
- null; 295; 299
- null, delegate; 274
- Nullable; 296
- NullReferenceException; 298
- Nøgle- og værdi, Dictionary; 168
- Object-Relational Mapping; 12
- objekter, oprettelse; 185
- objektorienteret programmering; 170
- Objektorienteret programmering; 23
- operatorer, boolske; 76
- operatorer, null; 95
- operatorer, overskrivning; 311
- operatorer, strenge; 89
- operatorer, tal; 69
- OrderBy, LINQ; 325; 331
- ORM; 12
- overflow; 70
- overload, metode; 137
- override; 245
- override, abstrakte metoder; 249
- overskrivning, arv; 244
- overskrivning, operatorer; 311
- parallel afvikling; 342
- PI, konstant; 99
- Polly; 152
- polymorfi; 241; 257
- post; 316; 334
- Predicate; 275
- Predicate<T>, LINQ; 327
- Procedural programmering; 23
- progamflow; 105
- Program-klassen; 40
- Programmeringsparadigmer; 22
- programpointeren; 121
- projektion; 333
- Promise; 346
- Properties-vinduet; 48
- property; 176; 218
- protected; 234
- public; 183
- punktumnotation; 26

- Python; 3
- Queue<T>; 167
- ReadAllTextAsync, File; 348
- record; 316
- reference, polymorfi; 250
- referencebaserede typer; 201; 207
- referencebaseret; 298
- referencebaseret variabel, delegate;
266
- referencebaseret variabel, interface;
259
- rekursivitet; 124
- returtypen fra en metode; 124
- RPC; 12
- sand; 75
- sandkasse; 34
- scope; 63
- Select, LINQ; 333
- Semikolon; 20
- seriel afvikling; 342
- Serilog; 152
- set-metode; 215
- short; 65
- signatur, metode; 134
- signatur, metoder; 228
- skabelon; 24; 171
- Skip, LINQ; 336
- Snippets; 51
- softwarefejl; 138
- solution; 41
- Solution Explorer-vinduet; 48
- Sortering, LINQ; 331
- source code; 19
- SPA applikation; 12
- specialtegn; 91
- SQL; 323
- stack; 203; 213
- Stack<T>; 166
- stak; 166
- strenge; 87; 88
- strenge, immutabel*; 96
- strenge, interpolerede; 92
- strenge, StringBuilder; 98
- string; 88
- struct; 171
- struktur; 171; 202
- Sum, LINQ; 336
- switch; 108
- synlighed; 214
- synlighed; 174
- synlighed, arv; 234
- synlighed, klasser; 183
- System.Boolean; 75
- System.Byte; 65
- System.Char; 87
- System.Collections; 162
- System.Collections.Generic; 162
- System.Console-klassen; 44
- System.Convert; 73
- System.DateTime; 78
- System.Decimal; 68
- System.Double; 68
- System.Exception; 141
- System.Int16; 65
- System.Int32; 65
- System.Int64; 65
- System.Nullable; 296
- System.Object; 241
- System.Objekt; 186
- System.Single; 68
- System.String; 88
- System.TimeSpan; 81
- System.Tuple; 303; 304
- System.ValueTuple; 303; 306
- Take, LINQ; 336
- Task; 346
- Task<T>; 346
- tegn; 87
- tekster; 87
- terning; 172
- terning, ludo (arv); 240
- ThenBy, LINQ; 331
- this; 190
- this, konstruktør; 199
- thread; 338
- Thread.Sleep; 339
- throw; 147
- tid; 81
- tidszoner; 81
- tilstandsmaskine; 347

- ToArray, LINQ; 326
- ToList, LINQ; 326
- ToString; 187; 241; 245
- ToString-metoden; 71; 84
- try/catch; 144
- tråd; 338
- tuborgklammer; 63
- Tuborgklammer; 19
- tuple; 303; 335
- typekonvertering, arv; 253
- typekonvertering, dato og tid; 85
- typekonvertering, enum; 103
- typekonvertering, tal; 72
- Typer; 24
- typesikkert; 33
- typestærk; 61
- typestærke samlinger; 254
- typestærkt; 32
- tællevariabel; 112
- udviklingsmiljø; 14
- UML; 172
- unhandled exception; 139
- Unicode; 87
- unsafe; 62
- using; 42
- variabel; 60
- variabelnavn; 62
- variabler, referencebaserede; 132;
156; 185
- variabler, værdibaserede; 132
- virkefelt; 34; 63
- virtual, arv; 242; 244
- virtuelle metoder; 253
- Visual Basic; 3
- Visual Studio; 14; 15; 47
- Visual Studio Code; 14; 17; 47
- Visual Studio Community; 15
- Visual Studio Enterprise; 15
- Visual Studio Professional; 15
- void; 31
- værdibaserede typer; 201; 207
- værdibaseret; 295
- warnings (null); 303
- WebAssembly; 12
- when; 110
- WhenAll, Task; 351
- WhenAny, Task; 351
- Where, LINQ; 325; 327
- whiteboard; 172
- Windows Forms; 9; 342
- Windows Presentation Foundation;
10; 342
- WinUI; 11
- WPF; 10
- Write, Console; 44
- WriteAllTextAsync, File; 348
- WriteLine, Console; 44
- Xamarin; 11
- Yatzy; 170

Hvis du gerne vil lære grundlæggende programmering med Microsofts populære programmeringssprog C#, eller har brug for et opslagsværk, har du nu den helt rigtige bog i hænderne.

Bogen om C# 9.0 er opdelt i følgende kapitler:

- Introduktion til .NET
- Udviklingsmiljøer
- En konsol-applikation i C#
- Praktisk brug af VS og VSC
- Introduktion til C#
- Simple variabler
- Tekster
- Konstanter
- Programflow
- Metoder
- Fejlhåndtering
- Arrays
- Samlinger
- Dine egne typer
- Klasser
- Hukommelsesteori
- Indkapsling
- Arv
- Polymorfi
- Interface
- Delegates
- Hændelser
- Avancerede typer
- LINQ
- Asynkron programmering

På www.bogenomcsharp.dk kan du desuden finde masser af videoer, som understøtter mange af kapitlerne.

Bogen kan benyttes som lærebog i C# programmering for private og på uddannelsesinstitutionerne og tager udgangspunkt i både Visual Studio og Visual Studio Code.

Bogen er skrevet af Michell Cronberg, som har mange års erfaring med programmering – både som underviser, foredragsholder, konsulent og forfatter. Michell Cronberg har udgivet flere bøger og hæfter om forskellige former for programmering.

ISBN 978-87-993382-3-8

