C# begreber

Dette dokument indeholder en kort gennemgang af forskellige C#-begreber sorteret alfabetisk.

`abstract` En klasse eller en metode kan være abstract. En abstract klasse er en klasse der ikke direkte kan oprettes objekter af. En abstrakt klasse skal viderespecificeres i en underklasse som man så kan bruge til at oprette objekter.

En abstrakt metode er en metode der ikke er implementeret. Metoden erklæres i klassen for at sikre at alle viderespecificeringer af klassen bliver nødt til at implementere metoden. Kun abstrakte klasser kan have abstrakte metoder.

`aggregation` Aggregation (og composition) benyttes når et objekt *indeholder* et andet objekt. Dette skal ses i modsætning til nedarvning, hvor objektet *er* et andet objekt (og mere).

Ved composition er der en tæt sammenhæng sådan at man kan sige at objektet består af blandt andet det andet objekt. Det giver ikke mening at slette det ene uden det andet.

Ved aggregation er der en løsere sammenhæng mellem de to objekter og man kan meningsfyldt slette det ene uden det andet.

`bibliotek` se library nedenfor.

`binært træ` En binær træstruktur er et **træ** hvor hver knude højest kan have to underknuder. Disse to knuder betragtes som børn af overknuden og i deres indbyrdes forhold beskrives de som "søskende" (siblings), dvs. i eksemplet nedenfor er B og C søskende, D og E er søskende osv.

Binære træer er meget anvendelige til løsning af forskellige datalogiske problemstillinger.

`catch` Instruktionen catch bruges til at fange exceptions som smides (is thrown). Hvis exceptionen bliver smidt inde fra en **`try`**-blok, aktiveres den `catch`-blok der er defineret efter `try`-blokken.

`class` Klassebegrebet er en vigtig byggeklods til at lave objektorienteret programmering. Klasser indeholder samlinger af data og metoder. Når man laver sit første projekt får man automatisk oprettet en klasse med navnet Program.

Klasser og typer er to begreber for det samme, men de bruges ikke helt i flæng. Normalt benyttes ordet type om de indbyggede C#-typer, mens `class` benyttes om de typer (klasser) man selv har lavet eller som er en del af .Net-bibliotekerne. Klasser kaldes dog ofte typer hvis det er den sammenhæng klassen benyttes i, hvis man opretter en variabel af klassen Kunde, siger man ofte at man opretter en variabel af *typen* Kunde.

Det er sjældent at indbyggede typer som `int`, `double`, `string`, etc. omtales som klasser, selv om de i C# faktisk behandles præcis som om de var egentlige klasser og de alle har en fælles parent klasse `object` hvis egenskaber de arver.

`composition` Composition (og aggregation) benyttes når et objekt *indeholder* et andet objekt. Dette skal ses i modsætning til nedarvning, hvor objektet *er* et andet objekt (og mere).

Ved composition er der en tæt sammenhæng sådan at man kan sige at objektet består af blandt andet det andet objekt. Det giver ikke mening at slette det ene uden det andet.

Ved aggregation er der en løsere sammenhæng mellem de to objekter og man kan meningsfyldt slette det ene uden det andet.

`constructor` En constructor er en speciel metode der kaldes når man laver et nyt objekt med kommandoen `new`. En constructor kan kendes på at den hedder det samme som den klasse den er constructor for, og så har den ingen returtype – *heller ikke* `void`.

Hvis man ikke laver en constructor til sin klasse, laver C#/Visual Studio automatisk en constructor uden parametre. Denne constructor kan man ikke få at se.

Hvis man laver sin egen constructor, forsvinder den constructor C#/Visual Studio ellers ville have lavet.

`di` Dependency injection.

DI (dependency injection). Polymorfi - vi overfører controlleren som parameter til det specielle program.

Bruges til ioc - inversion of control.

`delegate` Delegate er en datatype for *methods* som definerer en metodes signatur. Man kan oprette variabler af delegates på samme måde som til andre typer. Se mere her.

`exception` Ordet "exception" betyder undtagelse. Systemet aktiverer en exception hvis programmet gør noget der ikke er tilladt, f.eks. hvis man dividerer med nul eller kommer til at implementere uendelig rekurssion sådan at stakken løber fuld. Eller hvis man f.eks. forsøger at konvertere noget tekst "abekat" til et heltal.

Hvis ikke en exception fanges (med `catch`), så bliver den sendt til systemet der stopper programmet.

Ved hjælp af `try`, `catch`, `finally` kan man håndtere exceptions I sit program sådan at man kan fange problemet og håndtere det uden at programmet går ned. Det kan være en god måde at håndtere fejl f.eks. i brugerinput fordi det kan være svært at tage højde for alle de mange forskellige fejlsituationer der kan opstå i et program. Hvis man fanger en exception kan man vise fejlen til brugeren og sætte programmet i en meningsfuld tilstand.

`extension Methods`

Extension methods er en måde at tilføje methods til eksisterende typer (klasser) uden at nedarve denne klasse. Extension methods er statiske methods, men de kaldes som om de objekt-metoder.

`finally` Instruktionen `finally` bruges sammen med `try` og `catch`. `finally` blokken udføres efter `try`-blokken er færdigudført eller hvis der blev smidt en exception, efter catch-blokken er færdig.

Finally bruges til at udføre kode man vil have kørt både når try er gået godt og når den ikke er gået godt, f.eks. kunne man bruge en finally til at lukke en fil efter læsning eller skrivning, filen skal lukkes både når det er gået godt og når det er gået skidt.

`graf` En graf er i datalogien en mængde af knuder (nodes) der er forbundet ved et antal kanter (edges). Et træ kan ses som et specialtilfælde af en graf, hvor kanterne er child- og parentreferencerne. En graf kan være directed eller undirected. I en directed graf går kanter fra én knude *til* en anden, i en undirected graf går alle kanter begge veje.

`heap` På dansk: bunke eller hob. En bunke er en *binær træstruktur* der er organiseret på en speciel måde sådan at begge knuder under en given knude har værdier der er større end den givne knudes værdi. På den måde vil den mindste værdi altid være i roden. Det samme vil gælde for alle undertræer.

Bunken kan selvfølgelig også være konstrueret omvendt med den største værdi i roden.

`identifier` En identifier er navnet på et field, en property eller en method. Hvis det er et field, identificerer identifieren feltet, så i udtrykket

```
count = count +1;
```

er count en identifier.

`implementere` (på engelsk implement).

Implementere betyder noget i retning af *at udføre i virkeligheden.* I softwareverdenen betyder det at *programmere* en løsning.

`ioc` inversion of control.

dI (dependency injection). Polymorfi - vi overfører controlleren som parameter til det specielle program.

`lambda notation`

Lambda notation er en speciel syntax der er velegnet til at definere simple anonyme metoder.

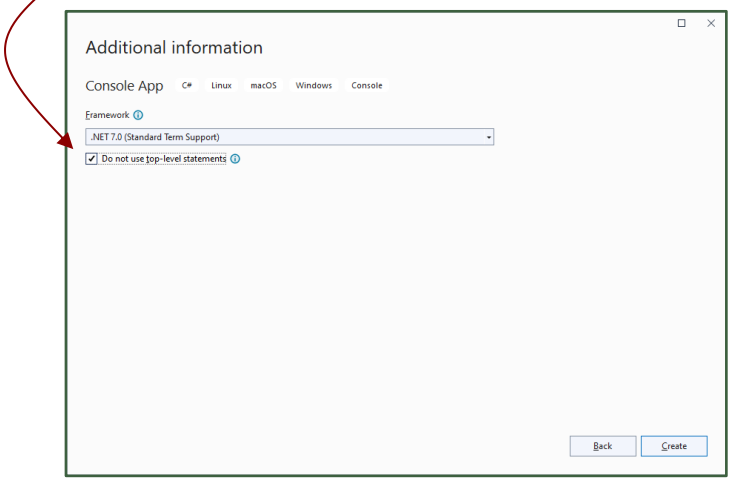
`library` Et library (på dansk = bibliotek) er en klasse eller en samling af klasser der er konstrueret selvstændigt sådan at man kan benytte dem i andre klasser. Et eksempel på et library kan være `System.Xml` som er et .Net bibliotek der understøtter Xml, sådan at man kan operere på et Xml-dokument med C#-objekter.

`LINQ` `LINQ` er et query sprog i C# der gør mange søgninger i lister mv. nemmere. Sproget er inspireret af SQL, og anvendes det mod et `DbSet`, oversættes det også til sql. Få mere information her.

`Main` Main-metoden er noget C# har arvet fra det oprindelige C-sprog. Når man starter et C-program op, kaldes metoden (som i C kaldes en function) Main.

I C# benyttes Main mest i forbindelse med console-programmer. I den nyeste version af Visual Studio (2022 og senere) er det muligt at få programmet til at køre uden at lave en Main-metode. Det er implementeret ved at Visual Studio bag om ryggen på programmøren sætter hans kode ind i en Main-metode som holdes skjult.

Hvis man gerne vil benytte Main metode eksplicit i sit projekt, , skal man vinge Do not use top-level statements af når man opretter projektet.



Hvis man gør det, opretter Visual Studio en stump kode der indeholder method, det er den der hedder `Main`. Metoden `Main` kaldes automatisk når programmet startes:

```
namespace ConsoleApp2
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

Metoden `Main` erklæres generelt sådan:

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

, hvor man erstatter de tre prikker med den kode man gerne vil have udført.

Efter metodens navn `Main` står der (`string[] args`). Det der står inden i parentesen er det information der sendes til metoden fra den der kalder metoden. Det ser lidt kryptisk ud og bruges til at implementere kommandoer man kan skrive i en kommandoprompt.

`method` Metoder i C# kendes som procedurer eller funktioner (undertiden underrutiner) i andre programmeringssprog. En metode er et navngivet stykke kode som kan *kaldes* hver gang man vil have udført en bestemt samling af instruktioner.

`mvc` Designmønster - ModelViewController.
Controllerens ansvar er er hente data og sende det til viewet.
Viewet ansvar er er at vise data.
Modellens ansvar er at gemme data.

`namespace` Som navnet antyder, så betyder namespace navneområde, det vil sige en samling navne (klasser) som hører sammen. Det betyder at hvis man har en using-instruktion i toppen af en fil, så kan man i filen referere alle klasser der er defineret i det namespace

instruktionen angiver. For at programmet skal kunne oversætte, skal man også have en reference til det bibliotek hvor namespacet er defineret.

`objekter` Begrebet objekter dækker over variabler af `class` type. Hvis vi har en `class` med navnet `Company`, så kan vi erklære *en variabel* af denne type ved at skrive:

```
Company mitFirma;
```

Modsat de indbyggede variabler[1] kan man ikke bare bruge variablen MitFirma nu. Vi skal have en instans af typen, også kaldt *et objekt*. Objektet skal oprettes – på samme måde som vi skulle oprette arrays. Det gør man sådan:

```
mitFirma = new Company();
```

`oversætte` Visual Studio oversætter C#-programmer til noget kode din computere kan udføre. Det kaldes at oversætte, compile eller bygge (build).

`parameter` I alle erklæringer og kald af metoder angiver man parametre i parentes lige efter metodenavnet. I nedenstående metode erklæres en metode der *tager*[2] en parameter af typen (klassen) `Company` og navngiver en lokal variabel `company` til at referere objektet.

```
void WriteName(Company company)
{
   Console.WriteLine(company.Name);
}
```

`private` Klasser og metoder kan erklæres `private`, `public` og `protected`[3]. Metoder der erklæres som private kan kun ses og kaldes fra andre metoder i samme klasse.

Metoder der er `public` kan kaldes af alle, `proctected` metoder kan kaldes af denne klasse og underklasser.

Hvis man erklærer en klasse for `private` (eller undlader at erklære den public), kan man kun bruge den inden for samme namespace.

`polymorfi` I objektorientering betyder polymorfi, at to klasser kan have samme grænseflade defineret via nedarvning, men udføre dem forskelligt. En Cykel kan arve funktionaliteten brems fra Køretøj, dvs. udføre opgaven, sådan som den er defineret i superklassen, men kan også have defineret sin egen måde at udføre opgaven på. Herved siges Bil at "overskrive" Køretøj mht. denne funktionalitet.

`referencer` I C# gemmes alle objekter ved hjælp af referencer. Det vil sige at en variabel der holder et objekt refererer et område i hukommelsen hvor objektet ligger. Antag vi har følgende linjer kode:

```
Company nytFirma = mitFirma;
```

Så bliver firmaet *ikke* kopieret, i stedet har vi to variabler der refererer det samme objekt i hukommelsen. Hvis man nu ændrer `mitFirma`'s adresse, så ændrer man derfor også

[1] Array er en undtagelse - et array kan man heller ikke benytte før man har oprettet det med new-kommandoen.
[2] Metoder med parametre sige *at tage* det man angiver i parameterlisten. Meningen er at det er noget information som metoden skal bruge for at den kan udføre sin handling.
[3] Ud over `private`, `public` og `protected` findes der også `internal`, `protected internal` og `private protected`.

automatisk `nytFirma`'s adresse. Dette er i modsætning til de indbyggede variabler hvor kodelinjen `int a = b;` har den effekt at *værdien* af `b` *kopieres* og tildeles variablen `a` som automatisk har sit eget hukommelsesområde.

`references` Når man i et program vil benytte klasser defineret i et library (bibliotek), skal programmet referere biblioteket. En sådan reference får programmerne til at blive bundet sammen, sådan at man som programmør kan tænke på biblioteket som en del af ens eget program.

`repository` Software repository: En samling af metoder.
Ellers kan et repository være en samling af objekter evt i et program.

`scope` Begrebet scope dækker over det område hvor en identifier (f.eks. en variabel) er kendt. I C# er scopes tydeligt markeret med symbolerne '`{`' og '`}`'. Området mellem de to krøllede parentes-symboler er et scope. Det vil sige at hvis man erklærer f.eks. en variabel, så vil den være kendt i hele scopet og scopes inden i dette scope (husk dog at variabler skal erklæres *før* de kan bruges, det vil sige erklæringen skal stå ovenover det første sted variablen bruges - evt. kan initialisering af variablen stå i samme linje som erklæringen).

`signatur` En metodes signatur består af retur-type, navn og parameterdefinition. Det vil sige alt det man skal vide for at kunne kalde metoden syntaktisk korrekt. Eksempel:

```
public static string ReadLine();
```

Bemærk at også de reserverede ord `public` og `static` er en del af signaturen, da de fortæller i hvilken kontekst metoden kan kaldes.

Signaturer kan bruges selvstændigt til at definere abstrakte metoder i klasser eller interfaces.

`soc` Separation of concerns.
Hvilke dele af programmet skal tage sig af hvilke opgaver

`static` `static` er kort nævnt i starten af dokumentet. En `static` metode kan kaldes direkte på klassen. Er metoden ikke `static`, kan metoden kun kaldes på et objekt af denne klasse (type).

Når man starter med at programmere console-programmer, vil Main-metoden være `static`. Derfor skal alle metoder Main kalder også være `static`. Det betyder reelt at langt de fleste metoder man laver i simple console-programmer er `static`.

`struct` `struct` minder meget om `class`, der er ikke noget man kan i `struct` som man ikke kan i `class`, omvendt har `struct` nogle begrænsninger i forhold til `class`. `struct` er med i C# af historiske årsager (det fandtes i `C` inden begrebet `class` blev indført med C++), og umiddelbart er der ikke nogen grund til at benytte `struct` i C# programmering.

`switch` En switch er en betingelses-instruktion der minder lidt om if-sætningen. Der er to grundlæggende forskelle. Den umiddelbare forskel er at en switch har indbygget flere muligheder i konstruktionen, det svarer til at benytte en masse `else if` i sin kode. Men man kan kun benytte switch til at undersøge om et udtryk *er lig* med forskellige *konstante værdier*.

`try` Instruktionen `try` benyttes når man vil udføre noget kode man ikke er sikker på kan gennemføres uden at der bliver smidt en exception.

En `try`-instruktion kræver at man også har en `catch` eller `finally` instruktion defineret efter `try`-blokken.

`træ` Et træ er en datastruktur bygget op på samme måde som en linket liste, bortset fra at hver telement (knude) har links til flere underknuder. Oftest vil underknuderne (child) også kende deres overknude (parent) på samme måde som elementerne i en dobbeltlinket liste kender begge naboelementer.

En træstruktur består af et antal knuder og nogle forbindelser mellem knuderne der er ordnet sådan at man kan tegne strukturen som et "træ", f.eks. et stamtræ. En folderstruktur hvor en folder kan have underholdere der igen kan have underfoldere etc. er også et eksempel på en træstruktur.

Samtidig gælder der at to knuder ikke kan have samme knude som underknude. Det betyder der ikke kan være nogen cyklisk struktur, man kan bevæge sig op og ned i træet, men ikke på tværs.

Der vil altid være en såkaldt *rod* i et træ, det vil sige en knude som ikke er underknude til nogen anden knude (og dermed ikke selv har nogen overknude).

Et træ har også altid et antal *blade*, det er knuder som ikke har underknuder.

Et **binært træ** er et træ hvor hver knude har to underknuder (eller færre).

`udkommentere` Nogle gange har man brug for at have noget kode til at stå i sit program, men som man af en eller anden grund ikke vil have til at blive udført, men man har alligevel ikke lyst til at slette det (endnu). Så kan man lave koden om til en kommentar ved f.eks. at sætte tegnene **//** foran linjen. Det kalder man at udkommentere kode.

`using` I toppen af alle C#-filer står nogle linjer med using-instruktioner. De angiver hvilke namespaces man vil have adgang til at referere klasser fra.

For at programmet kan oversætte skal man også lave en reference til de biblioteker som de anvendte namespaces ligger i.

`virtual` Man kan erklære både metoder og properties som virtuelle. I beskrivelsen nedenfor, kan metode alle steder erstattes af property.
En virtuel metode er en metode man erklærer i en basisklasse sådan at underklasser kan overskrive den.

```
virtual void Play() {<kode>}
```

Et eksempel på en virtuel metode der er indbygget i C# er metoden `ToString()` som er defineret på `object` (som bag om ryggen på os er basisklasser for alle klasser og typer i C#). Den kan man overskrive i sin egen klasse ved at skrive:

```
override void ToString() { <kode> }
```

	Et specialeksempel på en virtuel metode er en abstrakt metode. En abstrakt metode har ingen krop og *skal* overrides i den nedarvede klasse. Abstrakte metoder kan kun defineres i abstrakte klasser. En metode defineret i et interface svarer til en abstrakt metode.
`void`	`Det engelske` ord *void* betyder *ingenting*, i C# betyder det "ingen type". Hvis man skriver `void` i en metodeerklæring betyder at metoden *ikke sender information tilbage* til den metode der har kaldt den. I C# kan `void` kun stå foran metoder, man kan altså *ikke* have `void` variabler.
`WPF`	Windows Presentation Foundation (WPF) er et .NET bibliotek med Gui-komponenter til at opbygge en grafisk brugergrænseflade. Brugergrænseflader med WPF opbygges ved hjælp af Xml-formatet XAML.