Klasser i C#

I denne gennemgang af klasser i C# tager vi hul på begrebet klasser og hvad formålet med dem er. Klasser er et stort område, så man ved ikke alt om klasser når man har været gennem opgaverne i dette dokument, men man vil have en god forståelse af formålet med klasser og hvordan man bruger dem.

Mere avanceret brug af klasser til objekt orienteret programmering behandles i senere forløb.

Faglige læringsmål

* At forstå sammensatte datatyper
* At få en begyndende forståelse for klassebegrebet
* At få en begyndende forståelse for indkapsling

Konkrete elementer

* Nøgleordet class.
* Sammensatte datatyper med class.
* Indkapsling med metoder og properties.
* Constructor
* Metoden ToString().

Forudsætninger

Forståelse og en vis rutine i at arbejde med variabler, datatyper, arrays, betingelser og løkker, f.eks. ved at gennemarbejde dokumenterne

[1 - I gang med Visual Studio](https://drive.google.com/open?id=1ROCPhWN0eAHunoLNpySEzBKU0eYBuTEV)

[2 - Basal programmering](https://drive.google.com/open?id=1vvAy8nJ_K5xBczEGBcp_HB7jafIcScgj)

Indhold

[1. Introduktion 3](#_Toc530746594)

[2. Hvad er en klasse? 3](#_Toc530746595)

[2.1. Opgave 3](#_Toc530746596)

[2.2. Klassens bestanddele 3](#_Toc530746597)

[3. Klassen Player 4](#_Toc530746598)

[3.1. Eksempel: Array af spillere 4](#_Toc530746599)

[3.2. Sådan gør du i Visual Studio 4](#_Toc530746600)

[3.3. Opgave 5](#_Toc530746601)

[3.4. Oprettelse af objekter 5](#_Toc530746602)

[3.5. Opgave 5](#_Toc530746603)

[4. Klassemetoder 6](#_Toc530746604)

[4.1. Eksempel: Metoden Play() 6](#_Toc530746605)

[4.2. Opgave (slå ingen enere) 7](#_Toc530746606)

[5. Initialisering af objekter - constructor 7](#_Toc530746607)

[5.1. Hvad er en constructor 7](#_Toc530746608)

[5.2. Syntax for constructor 7](#_Toc530746609)

[5.3. Formålet constructoren 7](#_Toc530746610)

[5.4. Opgave 8](#_Toc530746611)

[5.5. Opgave (slå ingen enere) 8](#_Toc530746612)

[5.6. Kald af constructor 9](#_Toc530746613)

[5.7. Opgave 9](#_Toc530746614)

[6. Properties 9](#_Toc530746615)

[6.1. Syntax for properties 9](#_Toc530746616)

[6.1.1. Eksempel 10](#_Toc530746617)

[6.1.2. Hvordan fungerer det? 10](#_Toc530746618)

[6.1.3. Navnekonvention 10](#_Toc530746619)

[6.1.4. Manuel property (er det vi har implementeret ovenfor) 10](#_Toc530746620)

[6.1.5. Automatisk property 10](#_Toc530746621)

[6.2. Opgave 11](#_Toc530746622)

[6.3. Hvorfor benytte properties? 11](#_Toc530746623)

[6.4. Opgave 11](#_Toc530746624)

# Introduktion

I [2. Basal programmering](https://drive.google.com/open?id=1vvAy8nJ_K5xBczEGBcp_HB7jafIcScgj) sluttede vi af med at løse en opgave hvor vi skulle indtaste navn og pointtal for deltagerne i en konkurrence. Uden kendskab til klasser og sammensatte datatyper ville vi være nødt til at løse den opgave ved hjælp af to arrays, et med deltagernes navne og et med deres points.

Den måde at løse opgaven på kan dog hurtigt gøre programmer uoverskuelige, for det bånd der er mellem de to arrays er implicit, man skal vide det. Hvis programmeringssproget kunne hjælpe os med at fastholde båndet mellem brugerens navn og hans pointtal, så ville det gøre programmet langt mere overskueligt.

Der findes heldigvis en god måde at knytte disse oplysninger sammen, nemlig begrebet class.

Jeg har lavet en oversigt over forskellige begreber i C# og samlet dem i et dokument. Hvis du støder på begreber der forvirrer dig, kan du måske finde en kort i dette dokument: [C-sharp begreber](https://drive.google.com/open?id=1GPP4ZeVMk37AJ7SVnBci7azMbmwghJtz). Dokumentet redigeres løbende.

En oversigt over de mest almindelige C#-instruktioner kan findes i dette dokument: [Erklæringer i C-sharp](https://drive.google.com/open?id=1iXW8fN6KYML_8YVPosyAzmtforfq86W8). Dokumentet redigeres løbende.

# Hvad er en klasse?

En klasse er i bund og grund et værktøj til at programmøren kan definere sine egne datatyper. De datatyper der er indbygget i C# dækker de mest almindelige behov, og er tilstrækkeligt til at vi kan lave alle programmer. Men det er ikke altid lige nemt at lave og heller ikke at læse sådanne programmer. Husk de fleste programmer (og dele af programmer) bliver skrevet én gang, rettet en del gange og *læst* rigtig mange gange, så at koden er læsbar og nem at rette i er meget vigtige faktorer i forbindelse med programmering.

Du har allerede lavet flere klasser i C# selvom du måske ikke har været bevidst om det. Det skyldes at man altid har mindst én klasse i et program.

## Opgave

Lav et nyt C# console projekt i Visual Studio. Kald det *Klasser i C-Sharp* eller noget i den retning så det vil være nemt at finde igen senere.

## Klassens bestanddele

Betragt nu indholdet af filen Program.cs:

class Program

{

static void Main(string[] args)

{

}

}

Her kan du se at du automatisk har fået oprettet klassen Program. Klassen indeholder en static metode der hedder Main(). Denne klasse er ret speciel fordi den har som hovedformål at afvikle programmet. Når man starter projektet kaldes metoden Main().

# Klassen Player

## Eksempel: Array af spillere

Hvis vi tager udgangspunkt i eksemplet fra [2. Basal programmering](https://drive.google.com/open?id=1vvAy8nJ_K5xBczEGBcp_HB7jafIcScgj) hvor vi skulle registrere navn og point på deltagere i et spil, så ville det være rart hvis vi kunne knytte navn og point sammen på en mere direkte måde end ved at de har samme indeks i to forskellige arrays. Det kan man gøre ved at lave en ny datatype (class) for spillere:

namespace Klasser\_i\_C\_sharp

{

class Player

{

string navn;

int score;

}

}

­­­­­­Som det fremgår består Player-klassen af to fields - navn­­­­ og score.

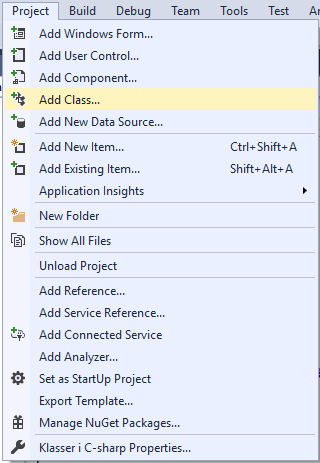
Nu kan vi i vores program gemme alle spillere i ét array i stedet for to:

Player[] players = new Player[10];

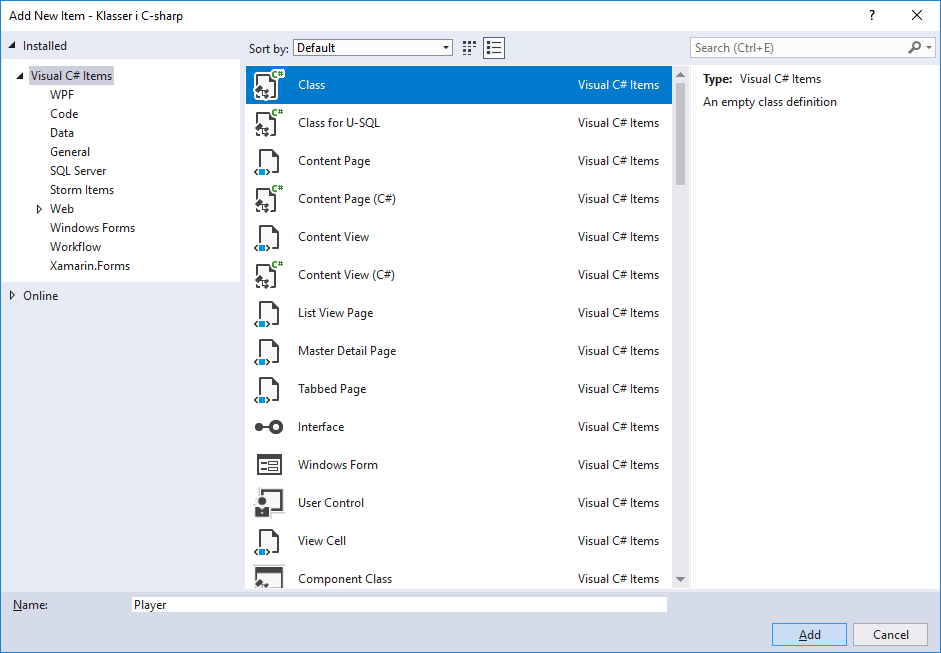
## Sådan gør du i Visual Studio

Hvis du ikke har prøvet at oprette nye klasser i Visual Studio, er der her en kort anvisning. Der er flere veje, her er angivet én af dem.

Først vælg Add Class… i Project-menuen.



Indtast derefter navnet på klassen i den dialog der åbner og tryk Add.



## Opgave

Opret klassen Player med to fields (kaldes også nogle gange member variabler) som vist ovenfor i afsnit 3.1.

## Oprettelse af objekter

Man kan oprette et Player-objekt på denne måde:

Player player; // Erklæring

player = new Player(); // Nyt objekt

Bemærk at der optræder Player med stor ’P’ og player med lille ’p’. C# er case sensitiv og det udnyttes i rigtig høj grad når man programmerer.

Det er en navnekonvention af variabler (også fields) begynder med et lille bogstav, mens klassenavne begynder med et stort bogstav.

* I den første linje erklæres *variablen* player til at være en variabel af *type* (klasse) Player.
* I den anden linje oprettes et nyt objekt af *typen* Player og *variablen* player sættes til at referere dette objekt.

## Opgave

Lav et program der opretter 5 objekter af typen (klassen) Player og lægger dem ind i players-array’et.

## Metoden ToString()

Alle klasser og typer har en indbygget virtual metode der hedder ToString(). Mertoden returnerer en default-visning af objektet og Metoden kaldes automatisk mange steder, f.eks. i forbindelse med Write af en variabel eller et objekt eller i en liste i en grafisk brugergrænseflade.

Det smarte ved metoden ToString() er at man kan overskrive den i sine egne klasser sådan at man selv kan bestemme hvordan objekter af klassen skal vises eller udskrives. Eksempel:

public override string ToString()

{

Return $”Navn: {name}. Score: {score}”;

}

Her returnerer ToString() for eksempel teksten ”Navn: Claus. Score 67”, hvis Player-objektet har navnet Claus og en score på 67.

## Opgave

Tilret programmet sådan at det udskriver alle spillere i players-arrayet.

# Klassemetoder

Vi har set ovenfor hvordan klassebegrebet kan bruges til at samle forskellige data i ét objekt sådan at man kan se på den information man har om en spiller som én samlet enhed. Men man har også ofte behov for at objekter af samme klasse skal dele *opførsel*. Dette kan man implementere ved hjælp af metoder på klassen.

## Eksempel: Klasse-Metoden Play()

Indtil nu består Player-klassen af en samling af fields, hvilket giver os mulighed for at samle flere stykker information om en spiller i ét objekt (et objekt at typen (klassen) Player).

Man kunne forestille sig at alle spillere gerne ville have mulighed for at ”spille”[[1]](#footnote-1). Til dette formål findes begrebet methods som også kendes også som procedure eller funktion i andre programmeringssprog. Da vi ønsker at programmet understøtte at spilleren selv spiller, skal metoden implementeres i klassen Spiller.

Vi kunne altså implementere en metode Play med følgende metode-hoved:

class Player

{

…

public void Play()

{

…

}

…

}

Så når man ønsker at spilleren skal spille kalder man metoden. I vores lille program, kunne man kalde metoden fra Main() eller en anden metode der er aktiveret fra Main().

Hvis man nu vil have spilleren til at spille, kalder man metoden på et objekt af typen Player som vist nedenfor:

Player player; // Erklæring

player = new Player(); // Nyt objekt

player.Play() // Metodekald

Perspektivering: Hvis man forestiller sig at man har flere forskellige spillere, så kunne Play være forskellig for forskellige spillere, nogle kunne være mennesker, andre computerspillere hvor nogle f.eks. kunne være optimister, andre mere forsigtige. Det er lidt mere avanceret brug af klasser end vi kommer til at arbejde med i dette dokument, men det er med til at illustrere fordelen ved at lægge metoder på klassen.

## Opgave (slå ingen enere)

Implementér metoden Play() sådan at den lader spilleren slå med en terning så mange gange som han vil – dog slutter han automatisk hvis han slår 1.

Hvis han vælger at stoppe inden han har slået 1, lægges det samlede antal opnåede resultater til spillerens (samlede) score. Hvis han slår 1 smides alle opnåede slag væk og (den samlede) score forbliver uændret - det vil sige, (den samlede) score kan aldrig blive mindre.

Lav et kald af metoden Play() sådan at metoden (kan) aktiveres når du kører programmet.

Kør programmet, test at Play() opfører sig som forventet.

# Initialisering af objekter - constructor

Ovenfor har vi set hvordan vi kan oprette nye spiller-objekter, men vi har snydt lidt, for vi har ikke lagt information ind i objekterne. Grunden til at vi har sprunget over det er at for at gøre det på en måde der overholder best practice, skal vi have lidt mere konkret viden på bordet først.

## Hvad er en constructor

Alle klasser[[2]](#footnote-2) har en constructor som benyttes når man vil oprette objekter af denne klasse (type). Betragt instruktionen vi har brugt til at oprette nye objekter:

player = new Player(); // Nyt objekt

Denne linje indeholder faktisk et kald af en metode. Det kan man se ved at der efter klassenavnet er en parentes – (). Både i metodekald og erklæringer af en metode er der altid en parentes til parametre, også selv om parentesen er tom.

Men … så har vi lavet et kald af en metode i vores kode, men vi har ikke erklæret metoden!? Hvorfor går det ikke galt?

Grunden til at vi kan oprette nye objekter af typen (klassen) Player uden at vi har lavet en constructor er at hvis man ikke selv laver en constructor, så laver C# selv en default constructor uden parametre. Det kan være en hjælp hvis man skal lave en simpel lille klasse. Men er det så ikke bare godt nok, hvorfor skulle man så lave sin egen constructor?

## Syntax for constructor

En constructor er en speciel method der har samme navn som klassen og som ikke har nogen returværdi. Metoden kaldes implicit når man opretter nye objekter af klassetypen og kan ikke kaldes eksplicit.

## Formålet constructoren

En constructor er en metode der kaldes som det allerførste efter der er reserveret hukommelse i computeren til objektet. I constructoren kan man derfor sætte de initielle værdier (startværdier) for det objekt man vil oprette. I vores eksempel, kan vi sætte navn og score i forbindelse med oprettelsen. Vi erklærer følgende metode i Player-klassen

class Player

{

string navn;

int score;

public Player(string navn, int score)

{

this.navn = navn;

this.score = score;

}

…

}

Objektets field Lokal variabel

Her har vi erklæret en constructor til klassen Player med parametrene name (som skal være en string) og score (som skal være en int). Constructoren er erklæret public, sådan at den kan kaldes udefra.

I kroppen af construktoren er der nogle ”mærkelige” instruktioner, this.name = name og this.score = score. Det er noget man ofte støder på i bl.a. constructors, så lad os se på det med det samme.

Klassen Player har et field der hedder name og constructor-metoden har en parameter der hedder name. Hvordan skal compileren vide hvilken name vi mener? Compileren bruger scope-regler, det vil sige den først leder i det umiddelbare scope.[[3]](#footnote-3) Det umiddelbare scope for koden i kroppen af constructoren er de variabler der er erklæret i kroppen eller hovedet af constructoren.

I vores constructor er variablerne name og score erklæret i hovedet af constructoren, så hvis vi benytter disse variabelnavne i kroppen vil de referere til de lokale variabler. Hvis vi gerne vil have fat i de variabler der er defineret på klassen kan man benytte this. this henviser det det nuværende objekt, så this.name henviser til det field name der er defineret på objektet, altså i objektets klassedefinition.

Instruktionen this.name = name; gemmer altså det name der overføres via parameteren i objektets field name.

## Opgave

Implementér constructoren for Player som vist ovenfor. Får du fejl? Kan du gennemskue hvorfor? Det forklares i afsnit 5.6 nedenfor , så du kan jo lige snyde-læse lidt.

## Opgave (slå ingen enere)

Implementér spillet ”slå ingen enere” sådan at man kan oprette et antal spillere som skiftes til at spille. Brugeren skal kunne indtaste navnene på deltagerne. Programmet skal holde styr på hvis tur det er og benytte metoden Play() som du har implementeret i opgave 4.2 når en spiller skal spille.

## Kald af constructor

Ovenfor har vi set at vi før kunne oprette Player-objekter med instruktionen

player = new Player(); // Nyt objekt

De to parenteser til sidst i instruktionen indikerer at der er tale om et metodekald. Hvis du prøver at oversætte (bygge) det program du har skrevet indtil nu, får du en fejl. Den skælder ud over at der ikke findes en constructor der ikke tager nogen parametre. Den constructor vi lavede i 5.4 tager jo to parametre.

Men hvorfor kunne den så godt oversætte før vi lavede constructoren?

Svaret er at hvis man ikke selv laver en constructor (default constructor), så laver C# selv én uden parametre, men så snart man laver sin egen constructor, findes default constructoren ikke længere. Derfor kan programmet ikke længere oversætte.

Vi skal altså have rettet kaldet til sådan at det nu sender den nødvendige information:

Player player = new Player("Claus", 0);

Ovenstående linje opretter en ny spiller ved navn Claus med initielt 0 (nul) point.

Hvis man ønsker at en spiller skal have et forspring tilbyder denne løsning at man kan initialisere objektet med det antal point han skal have fra starten.

## Opgave

Tilret kaldet af constructoren sådan at programmet kan oversætte og køre. Test programmet.

# Properties

Vi har nu fået kendskab til fields og methods. Fields er objektets variabler, methods er objektets funktioner. I C# findes der også noget der hedder properties. Properties opfører sig udadtil på samme måde som fields men tilbyder samtidig mulighed for avanceret programmering.

Properties bliver hurtigt meget centrale byggeklodser i dine C#-programmer.

## Syntax for properties

En property-erklæring ser sådan ud

<access> <type/class> <Navn> {<get- og set metoder>}

### Eksempel

public int Score

{

get

{

return score;

}

set

{

score = value;

}

}

Hvis ovenstående property er defineret på klassen Player, så kan man nu hvis man har et objekt af typen Player benytte propertien som om den var et public field:

Player player = new Player(”Claus”, 0);

player.Score = 10;

int gammelScore = player.Score;

### Hvordan fungerer det?

Når man har propertien på venstre side af et assignment, kaldes property-metoden set, variablen value bliver sat til værdien fra højre side i assignment instruktionen.

Det der sker i assignment instruktionen her er altså at propertiens set-metode bliver kaldt med variablen value sat til 10. Inde i set-metoden gemmes værdien af value (altså 10) i feltet score (med lille ’s’).

Når propertien optræder som et udtryk, f.eks. på højre side af en assignment-instruktion, kaldes property-metoden get. Det som get returnerer bliver propertiens værdi.

### Navnekonvention

Navne på properties skal altid begynde med stort.

Man benytter ofte et field med samme navn som propertien, blot med lille begyndelsesbogstav til at gemme propertiens værdi.

### Manuel property (er det vi har implementeret ovenfor)

I den løsning vi har implementeret ovenfor har vi selv som programmører styr over hvad der sker i get- og set-metoderne. I dette tilfælde gemmer vi værdien i et private field i set og henter værdien fra samme field i get.

### Automatisk property

Hvis man bare skal have nogle meget simple properties, kan man få C# til selv at implementere get og set og sørge for at gemme værdierne på en fornuftig måde. Det gør man sådan:

<access> <type/class> <Name> {get; set;}

Ved at lave en automatisk property, kan man hurtigt få sine properties op at køre. Efterhånden som man udvikler sig program, finder man sikkert ud af at man har brug for at get og/eller set gør noget lidt mere avanceret end bare at hente og gemme værdier. Så implementerer man propertien manuelt som vist i 6.1.1 til 6.1.3.

## Opgave

Lav en automatisk property sådan at man kan ændre spillerens navn også efter at Player-objektet er blevet oprettet.

## Hvorfor benytte properties?

Umiddelbart kan det se ud som om der ikke er vundet meget ved at lave properties. Specielt hvis man benytter automatiske properties hvor man i realiteten har direkte adgang til den bagvedliggende variabel. Men når man designer sin klasse med properties er det nemt at udvide indkapslingen ved at implementere get og set.

Man vil selvfølgelig kunne implementere samme opførsel som properties ved hjælp af metoder, men der er meget kode der bliver pæn hvis man benytter properties. Derudover benyttes properties til at binde komponenter i den grafiske brugergrænseflade op på.

## Opgave

Tilret propertien fra opgave 6.2 sådan at man kun kan ændre spillerens navn hvis han endnu ikke har opnået point.

Lav en property Score som kan læses men ikke ændres (se evt. eksempel 6.1.1 for inspiration).

1. At ”spille” kunne betyde at spillerobjektet fik lov til at kaste med terninger. Resultatet af terningkastet ville så kunne blive registreret på spilleren. [↑](#footnote-ref-1)
2. En abstrakt klasse har *ikke* en constructor, men begrebet abstract bliver ikke behandlet i dette dokument. [↑](#footnote-ref-2)
3. Scope og scope-regler er regler for hvilke identifiers der er kendt et bestemt sted i programmet. [↑](#footnote-ref-3)