

C# erklæringer

1. Erklæring og brug af variabler og metoder

1.1. Variabler, felter, properties

```
<access> <type> <name>;
```

Eksempler:

```
private decimal beløb;  
protected DateTime dato;  
public int kundeNummer;
```

1.1.1. Properties – automatiske og manuelle

```
<access> <type> <name> {<get and/or set>}
```

Eksempler:

```
public int kundeNummer {get; set}  
public int kundeNummer {get; protected set;}
```

```
// Manuel property  
private int knr;  
public string KundeNummer  
{  
    get  
    {  
        return Convert.ToString(knr);  
    }  
    set  
    {  
        knr = Convert.ToInt(Value);  
    }  
}
```

1.2. Metoder (procedurer og functions – flere navne for det samme)

```
<access> <type> <name>(<declaration parameters>) {<body>}
```

, hvor <declaration parameters> er

```
<type> <name>, <type> <name>, ... (et vilkårligt antal parametre)
```

Og <body> er metodens kode. Typen void benyttes hvis metoden ikke returnerer en værdi.

Declaration parameters kaldes også de formelle parametre, det er her man angiver typen og det navn der bruges *inde* i metoden.

Eksempler:

```
void NyUdløbsdato(DateTime dato)
{
    udløbsdato = dato;
}
DateTime HvorLangeTilUdløb()
{
    return udløbsdato - DateTime.Today();
}
```

2. Brug af variabler og kald af metoder

2.1. Brug af variabler

Assignment (tildeling af værdi):

```
<variable name> = <expression>
```

Hvor <expression> er et udtryk der ved beregning giver en værdi af samme type som variabelen.

```
kundeNummer = 1107;
prisMedMoms = prisUdenMoms * 1,25;
```

Access (brug af variabelens værdi): En variabel kan bruges i alle <expressions>. Når man tilgår variabelen på den måde, er det variabelens værdi der

```
if (kundeNummer == 1107)
{
    throw new Exception("Ulovligt kundenummer");
}
```

2.2. Kald af metoder

Void metoder - metoder uden returværdi

```
<method name>(<actual parameters>);
```

Hvor <actual parameters> er:

```
<parameter value>, <parameter value>, ...
```

Altså et antal værdier der skal overføres til den kaldte metodes declaration parameters (se erklæring ovenfor).

Function methods (metoder med returværdi kaldes også funktioner).

Når en metode med returværdi indgår i et udtryk, benyttes den værdi der er resultatet af funktionen (det den returnerer) i forbindelse med beregningen af udtrykket. Det kan foregå på mange forskellige måder:

```
<variable name> = <method name>(<actual parameters>);
if (<method name>(<actual parameters>)) {<body>}
```

Eksempler:

```
NyUdløbsdato(Convert.ToDateTime("2019-05-02"));  
DateTime tidTil = kunde.HvorLangeTilUdløb();
```

Betingelser og løkker

Betingelser og løkker er hjørnestenene i alle programmer.

3. Betingelser

Betingelser bruges til at få noget kode udført når en bestemt betingelse er opfyldt, som f.eks. at udskrive en fejl hvis man forsøger at hæve penge fra en konto hvor saldoen er 0.

3.1. If-sætninger

If-sætninger består af nøgleordet `if` efterfulgt af en betingelse i parentes og en body-del. Sætningen kan udvides med et antal `else if` betingelser og en `else`-del. `else`-delen udføres hvis ingen af betingelserne ovenfor er opfyldt.

3.1.1. Simpel `if`-sætning:

```
if (<condition>) {<body>}
```

3.1.2. `if`-sætning med `else`-del

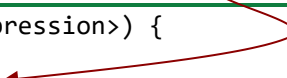
```
if (<condition>) {<body>} else {<body>}
```

3.1.3. `if`-sætning med `else if` dele og `else`-del

```
if (<condition>) {<body>}  
else if (<condition>) {<body>}  
else if (<condition>) {<body>}  
else {<body>}
```

3.2. `switch/case` konstruktioner

Hvis man skal checke om et udtryk har en af mange værdier, kan det være nemmere og mere effektivt at anvende en `switch`. Man skal bare være opmærksom på at betingelserne i en `switch` kun kan være værdier, altså konstante tal eller strenge.

```
switch (<expression>) {  
{  
    case 1:   
        Console.WriteLine("case 1");  
        break;  
    case 2:  
        Console.WriteLine("case 2");  
        break;  
    case 3:  
        Console.WriteLine("case 3");  
        break;  
}
```

, hvor `<expression>` er en beregnet værdi. `break`-instruktionen sørger for at programmet springer ud af `switch`en når den kode der hører til betingelsen er udført. Uden `break`-instruktionerne i koden ovenfor ville alle tre udskrifter blive udført hvis beregningen af `<expression>` gav 1.

4. Løkker

Ofte har man brug for at udføre noget kode flere gange, f.eks. gennemløbe en liste og udskrive hvert element. Der findes forskellige løkkekonstruktioner i C#, hver med deres fordele.

4.1. foreach

foreach-løkker er nemme at bruge hvis man skal lave et hurtigt gennemløb af en liste eller et array. En foreach-løkke ser sådan ud:

```
foreach (<type/class> <variable> in <list/array>) {<body>}
```

Variablen bruges til at tilgå elementerne i listen, så hver gang <body> udføres, så ligger det nuværende listeelement i variabelen.

4.2. for-løkke

Nogle gange vil man gerne udføre noget kode et bestemt antal gange. Det kan også være at man har brug for at vide hvilket nummer i listen man har fat i ved gennemløb af en liste. I disse tilfælde kan en for-løkke være interessant.

```
for (<initialize>; <condition>; <iterator>) {<body>}
```

Man kan gøre meget avancerede ting med for-løkker, men normalt bruges strukturen til simple ting da while-løkker er nemmere at læse (og konstruere) når løkkerne bliver mere komplekse. Vi vil ikke her gå i detaljer med de avancerede muligheder med for-løkker, men bare give et eksempel på en simpel brug af for (bortset fra udskriften af indekset, kunne vi med fordel have anvendt foreach):

```
for (int i=0; i<kundeListe.Length; i++)  
{  
    Console.WriteLine("index: "+i+", kunde: "+kundeListe[i].Navn);  
}
```

Det mest almindelige er en konstruktion som ovenstående, hvor <body>-delen udføres lige så mange gange som der er elementer i en liste og variabelen *i* tælles op for hver iteration af løkken. *i* kan benyttes til at slå op i kundelisten, men hvis det er det eneste man har behov for, vil en foreach-løkke være at foretrække.

4.3. while-løkke

En while-løkke er mere generel end for og foreach. Her kan koden udføres lige så længe en betingelse er opfyldt uafhængigt af f.eks. listers længder. Til gengæld skal man selv holde styr på elementer og variabler i koden, løkke-konstruktionen giver ikke hjælp til dette.

```
while (<condition>) {<body>}
```

Eksempel hvor første element hvor type er "Hest" findes og navnet gemmes:

```
int i = 0;
string navn;
while (not found && i < list.length)
{
    if (list[i].type == "Hest")
    {
        Navn = list[i].Navn;
        found = true;
    }
    n++;
}
if (found)
{
    Console.WriteLine("Hesten hedder: "+navn);
}
else
{
    Console.WriteLine("Der er ingen hest på gården");
}
```

4.4. do-løkker

Som man kan se kontrolleres betingelsen for alle ovennævnte løkker i starten af hver iteration. Man kunne godt forestille sig situationer hvor man gerne vil have udført noget kode (koble på Tcp, f.eks.) og gentaget koden indtil man rent faktisk kommer ind. Det kan selvfølgelig godt ordnes med en `while`, men så skal man som udgangspunkt skrive det samme koble-op kode før løkken og inde i løkken. Tanken med `do`-løkker er at undgå denne gentagne kode.

Erfaring viser dog at man oftest alligevel bliver nødt til at skrive `do`-løkker om til `while`-løkker, så de er sjældent brugt.

```
do {<body>} while (<condition>)
```

Eksempel:

```
do
{
    bool connected = ConnectToServer(serveraddress);
} while (!connected)
```

C# Constructors

5. Constructor

(Konstruktør, den metode der kaldes når et nyt objekt laves)

Constructoren er den første metode der bliver kaldt på alle objekter.

5.1. Erklæring af constructor

- En constructor er speciel på den måde at den ikke har nogen type – heller ikke en void type.
- En constructor har samme navn som klassen.
- En constructor kaldes automatisk ved brug af `new` kommandoen.

Eksempler

```
Class Customer
{
    // Member variable
    private string customerName;

    // Constructor - no return value, same name as class.
    public Customer(string name)
    {
        customerName = name;
    }
    ...
}
```

5.2. Kald af constructor

Constructoren kaldes automatisk når man opretter et nyt objekt af denne klasse. Som ved almindelige methods, skal man sende de korrekte aktuelle parametre med til metoden, dvs. parametrene skal passe til de formelle parametre der er defineret i erklæringen af metoden.

```
<variable name> = new <class name>;
```

Eksempler

```
Customer kunde;
Kunde = new Customer("Hans");
```

Eller

```
Customer kunde = new Customer("Hans");
```

Dot-notation

Dot-notation bruges til at referere ind i objekter, klasser og namespaces.

Eksempel:

```
Data.selectedkunde.primærekonto
```

Hvis man skal navigere i sin egen objektstruktur, benyttes altid objekterne (altså objektnavnene, ikke klassenavnene). Når man vil kalde funktioner defineret i et .NET namespace kan man skrive en "dot-sti" der referer ind i namespaces og under-namespaces";

6. Dot-notation i objekter

Når man benytter dot-notation, kan man få fat i både methods og fields (member attributter/variabler). Ved methods skal man benytte syntaxen for metodekald som beskrevet under overskriften "Kald af metoder".

Notationen kan benyttes meget fleksibelt, hver gang man står med et objekt, kan man anvende " ." for a "se ind i objektet". Man kan se alt det man har access-ret til. Men her er nogle eksempler

Adgang til en attribut:

```
<Object Name>.<Field Name>
```

Adgang til en metode:

```
<Object Name>.<Method Name>(<Actual parameters>);
```

Hvert led i dot-stien kan være et field eller en method, det er meget fleksibelt.

7. Dot-notation i namespaces

I .NET-bibliotekerne findes en stor mængde funktioner man kan kalde direkte uden objekter. Når namespaces er defineret inden i namespaces, kan man "dotte" sig ind til den ønskede metode.

Eksempel:

```
Console.WriteLine("Saldo på kontoen er: " + saldo);
```

Her kaldes metoden WriteLine i namespace Console.