

# 87410

## "Avanceret C#"

---

### Lab Manual

Wincubate ApS  
18-09-2018





## Indledning

### Øvelsestyper

Opgaverne i disse labs er af varierende svær- og frihedsgrad. Sædvanligvis vil de indledende øvelser være relativt bundet omkring en slags "lab walkthrough", hvorimod de senere øvelser vil være friere eller være af mere eksperimenterede karakter. Derfor er de forskellige opgaver klassificeret med en indikation af type.

De indledende og grundlæggende øvelser af walkthrough-typen er ikke yderligere kategoriseret. De resterende øvelser, som man ikke nødvendigvis forventes at nå i de afsatte tid, er klassificeret som følger



Øvelser markeret med en enkelt stjerne er lidt løsere specificeret baseret på hints eller få punkter, men omhandler centrale dele af stoffet.



Øvelser markeret med to stjerner indeholder enten kun et enkelt hint, er lidt sværere eller kræver vidensindsamling.



Øvelser markeret med tre stjerner er svære og sjove udfordringer, der kan grubles over for de interesserede.

### Forudsætninger

Disse opgaver forudsætter, at materialet hørende til kurset er udpakket i

C:\Wincubate\87410

samt at Visual Studio 2015 (og helst 2017, dvs. 15.7.\* eller senere) er installeret på maskinen.



## Module 1: "Advanced Types and Methods"

### Øvelse 1.1: "Extension-metoder"

Denne øvelse går ud på at definere to extension-metoder på `int` og `DateTime`, hhv., for på den måde at bygge genbrugbare metoder, som I kan udvide jeres værktøjskasse med.

#### Implementation af extension-metoder

1. Lav et nyt projekt "ExtensionsMethods" i  
C:\Wincubate\87410\Module 1\Lab 1.1\Starter.
2. Lav en ny fil "Extensions.cs"
  - Lav namespace i filen om til `<Jeres_firmanavn>.Utility`
    - Eksempelvis `Wincubate.Utility`.
  - Implementér i denne fil en extension-metode `IsEven()` på `int`, som returnerer en `bool`, der angiver om tallet er lige.
  - Implementér ligeledes i denne fil en extension-metode `To<Jeres_firmanavn>TimeStamp()` på `DateTime`, som returnerer en streng med jeres yndlingsmåde at lave timestamps på
    - Eksempelvis som vist i slidesene.

#### Test af extension-metoder

3. Åbn den eksisterende "Program.cs" fil og lav kode til at teste jeres ny-implementerede metoder.
4. Hvad sker der, når I prøver at kalde metoderne?
5. For at I kan kalde extension-metoderne, skal de "bringes ind i scope" – Enten vha. at benytte hele stien til metoden eller at benytte  
`using <Jeres_firmanavn>.Utility;`
6. Prøv nu at kalde metoderne på udvalgte heltal og `DateTime`.
  - Virker det nu?
  - Hvordan ser Intellisense ud for metoderne?

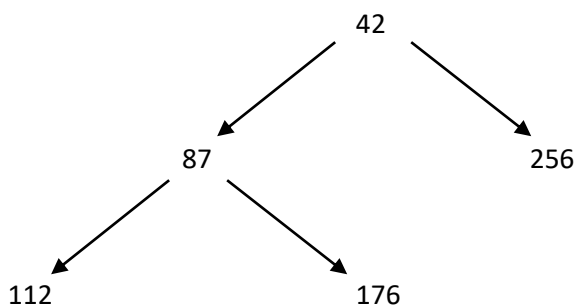
#### Hvis du vil se flere extension-metoder...

7. Find en spændende extension-metode fra <http://www.extensionmethod.net> og afprøv den 😊

## Øvelse 1.2: "Rekursive datatyper" (☆☆)

I denne opgave vil vi beskæftige os med rekursive datatyper. Vi vil lave en klasse Node, der selv kan referere instanser af Node. **Bemærk** at structs i C# ikke er tilladt at være rekursive!

Node skal udgøre knuderne i et binært træ, som for eksempel



Hver af knuderne i træet har en heltalsværdi, der kan repræsenteres som en `int`.

### Lav `Node()` klassen

- Lav et nyt projekt i Visual Studio 2015 til ny C# konsol-applikation
  - Kald projektet "BinaryTrees" og
  - Placér det i  
C:\Wincubate\87410\Module 1\Lab 1.2\Starter
- Definér klassen Node.
- Lav en constructor, der accepterer
  - et heltal, der er knudens værdi, samt
  - to instanser af Node (defaultet til null)

### Lav binært træ

- Lav en variabel node af type Node.
- Skriv et udtryk, der sætter node til at indeholde det binære træ afbildet ovenfor.

### Udskriv træet

- Lav en metode `Print()` på Node-klassen, der gennemløber træet og udskriver værdierne i knuderne.
- Test din metode på node defineret ovenfor
  - Den udskrevne sekvens af værdier for `node.Print()` skal være 42, 87, 112, 176, 256.

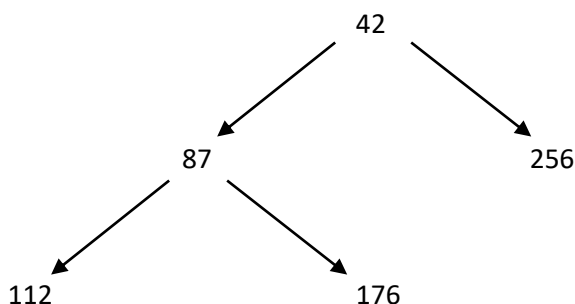
### Alternativ udskrift

Gennemløbet af træet, som du lavede ovenfor, kaldes et *prefix-gennemløb*.

- Overvej, hvordan du kan lave en alternativt `Print()` metode, så den udskrevne sekvens bliver 112, 176, 87, 256, 42.

### Øvelse 1.3: "IEnumerable på rekursive datatyper" (☆☆☆)

I denne opgave vil vi implementere `IEnumerable` på den rekursive datatyper, som vi definerede i Øvelse 1.2, hvor vi skrev en klasse `Node`, der selv kan referere instanser af `Node`. `Node` udgjorde knuderne i et binært træ, som for eksempel



Hver af knuderne i træet har en heltalsværdi repræsenteret ved en `int`. (Se evt. Øvelse 1.2 for flere detaljer)

- Tag udgangspunkt i løsningen fra Øvelse 1.2, der er at finde i `C:\Wincubate\87410\Module 1\Lab 1.3\Starter`.
- Fjern metoden `Print()` og implementér `IEnumerable` på en snedig måde, så ovenstående træ enumereres som sekvensen 42, 87, 112, 176, 256.
- Test din implementation ved at ændre `Main()` til at udskrive node vha. en `foreach`-sætning.
- Fjern tilsvarende metoden `PrintPostfix()` og implementér `IEnumerable` på en snedig måde, så ovenstående træ enumereres som sekvensen 112, 176, 87, 256, 42
  - **Bemærk:** Da der nu er to forskellige implementationer af iteratorer til `IEnumerable` på `Node`, er det nødvendigt nu at benytte "**named iterators**".
- Test din implementation ved at ændre `Main()` til at også at udskrive node vha. en `foreach`-sætning.

## Øvelse 1.4: "Sorterede mængder" (☆☆)

De nye mængde-collections introduceret i .NET 4.0 er ofte særdeles brugbare. I denne opgave vil vi benytte dem til at finde og sortere mængder af personnavne.

Datagrundlaget for opgaven er givet i start-projektet i form af en `List<Person>`, der fra start indeholder en masse instanser af typen `Person`.

### Sorteret mængde forekommende fornavne

- Tag udgangspunkt i navne-datagrundlaget, der er at finde i `C:\Wincubate\87410\Module 1\Lab 1.4\Starter`.
- Lokalisér `TODO 1` i `Program.cs`.
- Lav et stykke kode til at beregne mængden af fornavne, der forekommer i data-grundlaget, og udskriv dem sorteret alfabetisk.

### Forekommende efternavne sorteret efter længde

- Lokalisér `TODO 2` i `Program.cs`.
- Udvid din løsning ovenfor til at beregne mængden af efternavne, der forekommer i data-grundlaget, og udskriv dem sorteret efter længde og derefter alfabetisk.
  - Hint: Lav en klasse `LastNameComparer`, der implementerer `IComparer<T>` og lav et `SortedSet<T>` vha. denne.



## Øvelse 1.5: "Metoder med optional og named parametre" (★)

Formålet med denne øvelse er på et simpelt eksempel at vise, hvorledes optional og named parametre kan benyttes i C#. Vi vil lave et simpelt veksleprogram, hvor et beløb i en givet valuta kan indtastes sammen med en vekslekurs for denne (eks. EUR eller USD), hvorefter programmet beregner det tilsvarende beløb i DKK.

### Lav ComputeCurrency() metode

1. Lav et projekt i Visual Studio 2015 til ny C# konsol-applikation
  - a. Kald projektet "Currency" og
  - b. Placer det i  
C:\Wincubate\87410\Module 1\Lab 1.5\Starter
2. I Program.cs på linie med Main() metoden defineres en metode ved navn ComputeCurrency
  - a. Metoden skal acceptere et første argument amount af typen double
  - b. Metoden skal acceptere et andet argument rate af typen double
  - c. Metoden skal returnere en værdi af typen double
  - d. Metoden skal være static som Main().
3. Kroppen af ComputeCurrency() skal nu udfyldes således, at der returneres værdien af beløbet i amount konverteret med en valutakurs svarende til værdien af rate.
4. Når du er færdig, bør resultatet ligne

```
static double ComputeCurrency( double amount, double rate )
{
    return amount * rate;
}
```

5. Lav kode i Main() til at teste, hvad USD 50 svarer til i DKK ved en vekslekurs på 5,7781.
6. Når du er færdig, bør resultatet ligne

```
static void Main( string[] args )
{
    Console.WriteLine( ComputeCurrency( 50, 5.7781 ) );
}
```

7. Kør programmet og test at din løsning er korrekt.

### Lav parametrene i ComputeCurrency() optional

8. Modificér ComputeCurrency(), således at
  - a. amount er optional med en default-værdien værende 100,00, som sædvanligvis er standardbeløbet man sammenligner valutaer i.
  - b. rate er optional med en default-værdien værende 7,4520, der er raten for EUR, som vi vil lade være default-valuaten.
9. Når du er færdig, bør resultatet ligne

```
static double ComputeCurrency( double amount = 100.00,
                               double rate = 7.4520 )
{
    return amount * rate;
}
```

10. Lav den letteste og pæneste kode i Main() til at teste,

- c. hvad 200 GBP til kurs 8,9011 svarer til i DKK.
- d. hvad 50 enheder i default-valutaen svarer til i DKK.
- e. hvad et default antal enheder i default-valutaen svarer til i DKK.

11. Når du er færdig, bør resultatet ligne

```
static void Main( string[] args )
{
    Console.WriteLine( ComputeCurrency( 200, 8.9011 ) );
    Console.WriteLine( ComputeCurrency( 50 ) );
    Console.WriteLine( ComputeCurrency() );
}
```

12. Kør programmet og test at din løsning er korrekt.

### Brug named parametre i ComputeCurrency()

13. Lav nu den letteste og pæneste kode i Main() til at teste,

- a. Hvad et default antal enheder i GBP til kurs 8,9011 svarer til i DKK.

14. Når du er færdig, bør resultatet ligne

```
static void Main( string[] args )
{
    Console.WriteLine( ComputeCurrency( rate: 8.9011 ) );
}
```

15. Kør programmet og test at din løsning er korrekt.

### Hvis du har lyst...

... så kan du lave ComputeCurrency() metoden om til at returnere en streng indeholdende en korrekt formatteret valuta-streng i stedet for at returnere en double med beløbet.

- Brug `string.Format` med "c" til at lave en korrekt udskrivning af det korrekte beløb med valuta-indikation og to decimaler.

## Øvelse 1.6: "Generiske metoder" (☆☆)

Ligesom typer kan være generiske, kan man lave generiske metoder, der på tilsvarende vis har type-parametre og constraints. Se evt. [Chapter 9, s. 350 – 352] for en forklaring af constraints, hvis du ikke allerede kender disse.

- Implementér en generisk metode  
`int GreaterCount<T>(IEnumerable<T> data, T threshold)`  
der returnerer antallet af elementer i data, som er større end threshold
  - Hint: Hvad skal der kræves om T for at dette er muligt?
- Test din implementation på data som eksempelvis
  - `List<int>`
  - `Stack<string>`
  - `Queue<Car>...?`

Læg i øvrigt mærke til, at T ikke nødvendigvis behøver specificeres ved metode-kaldene...

## Øvelse 1.7: "Generisk, enumerérbar LinkedList" (☆☆)

Den meget simple LinkedList-klasse, som vi så i gennemgangen, er at finde i  
C:\Wincubate\87410\Module 1\Lab 1.7\Starter

Tag udgangspunkt i denne klasse og

- Gør LinkedList-klassen generisk, så den kan iterere gennem elementer af type T i Node.
- Implementér IEnumerable<T> på LinkedList<T>.
- Lav en liste af elementerne 42, 87, 112, 176, 255 og brug foreach til at udskrive elementerne.

Bemærk at det at implementere det generiske IEnumerable<T> interface kræver, at man holder tungen lidt lige i munden...

## Øvelse 1.8: "SequencePacker<T>" (☆☆☆)

Ud fra eksisterende generiske typer kan der laves mere og mere kraftfulde generiske typer, som kan genbruges mange gange i forskellige sammenhænge.

1. Lav en generisk klasse `SequencePacker<T>`, der lagrer sekvenser af `T` på en komprimeret form, så sekvensen  
42 87 87 87 11 22 22 87 99  
med et passende valg af type logisk set repræsenteres som sekvensen  
(42,1) (87,4) (11,1) (22,2) (87,1) (99,1)
2. Din klasse skal
  - virke for alle passende `T` (og ikke kun `int` som i eksemplet ovenfor).
  - implementere `IEnumerable<T>`
  - understøtte operationerne
    - `void Clear()`
    - `void Add( T t )`.
3. Test din implementation.

Bemærk igen, at det at implementere det generiske `IEnumerable<T>` interface kræver, at man holder tungen lidt lige i munden...

(Lav evt.

4. `T Remove( int index )`

hvis du keder dig... ☺)



## Module 2: "Delegates, Events, and Lambdas"

### Øvelse 2.1: "Delegates"

Denne øvelse laver et par eksempler for at illustrere, hvorledes delegates er referencer til metoder, som kan kaldes præcis som metoder.

#### Delegates til heltalsfunktioner

1. Lav et nyt projekt "DelegateFun" i  
C:\Wincubate\87410\Module 2\Lab 2.1\Starter.
2. Lav en ny fil "IntAction.cs"
  - Definér i denne fil en delegate-type ved navn `IntAction`, som kan referere metoder, der tager et heltal som parameter og returnerer `void`.
3. Åbn den eksisterende "Program.cs" fil
  - Lav en statisk metode `PrintInt()` med en signatur, som matcher `IntAction`
    - Metoden skal skrive det medfølgende argument ud på Console
  - Lav ydermere en statisk metode `DoForAll()`, der tager to parametre af typerne `IntAction` og `int[]`, henholdsvis, og returnerer `void`
    - Metoden skal kalde den medfølgende delegate på hvert heltal i arrayet.
4. Test dine implementationer ved at lave et array bestående af tallene 42, 87, 112, 176, 256 og print tallene i dette array ud vha. `DoForAll()`.

## Øvelse 2.2: "Aktier og events"

Denne opgave går ud på at benytte publishers og subscribers af aktie-kurser, der kommunikerer vha. events. Vi vil først konstruere en StockChanged event for en publisher med tilhørende event-argumenter, for derefter at implementere en subscriber til disse events.

### Lav StockChangedEventArgs

1. Åbn det eksisterende projekt "Stocks" i  
C:\Wincubate\87410\Module 2\Lab 2.2\Starter.
  - a. Verificér, at projektet i "StockPublisher.cs" indeholder en skabelon til en klasse kaldet StockPublisher, som senere skal udfyldes nedenfor.
    - i. Læg mærke til member-variablen `_ticker`, der indeholder navnet på aktien.
    - ii. Inspicér kort resten af klassen, der sørger for, at metoden `OnStockChanged()` kaldes med jævne mellemrum med en ny aktiekurs.
  - b. Verificér, at projektet i "Program.cs" indeholder den sædvanlige `Main()`-metode.
2. Tilføj en ny klasse i en ny fil "StockChangedEventArgs.cs"
  - a. Lav en public klasse `StockChangedEventArgs`, der arver fra `EventArgs`
  - b. Lav på klassen en public property af type `string`, der hedder `Ticker`
    - i. Lav en tilhørende `get`-metode, der returnerer en underliggende `private` og `readonly` member variable `_ticker`
  - c. Lav på klassen en public property af type `double`, der hedder `StockValue`
    - i. Lav en tilhørende `get`-metode, der returnerer en underliggende `private` og `readonly` member variable `_stockValue`
  - d. Lav på klassen en public property af type `DateTime`, der hedder `TimeStamp`
    - i. Lav en tilhørende `get`-metode, der returnerer en underliggende `private` og `readonly` member variable `_timeStamp`
  - e. Lav en public constructor for `StockChangedEventArgs`, der tager to parametre `ticker` og `stockValue` og sætter de tilsvarende property members
    - i. Initialisér `_timeStamp` til at være det aktuelle tidspunkt.
3. Når du er færdig, bør resultatet ligne



```

public class StockChangedEventArgs : EventArgs
{
    public string Ticker
    {
        get
        {
            return _ticker;
        }
    }
    private readonly string _ticker;

    public double StockValue
    {
        get
        {
            return _stockValue;
        }
    }
    private readonly double _stockValue;

    public DateTime TimeStamp
    {
        get
        {
            return _timeStamp;
        }
    }
    private readonly DateTime _timeStamp;

    public StockChangedEventArgs( string ticker,
                                  double stockValue )
    {
        _ticker = ticker;
        _stockValue = stockValue;
        _timeStamp = DateTime.Now;
    }
}

```

4. Byg programmet og korriger eventuelle fejl.
5. **Ekstra:** Hvis du kender til de nye features introduceret til C# 6.0, så kan ovenstående skrives meget lettere og kortere... Ellers vent til disse introduceres i Module 08. 😊

### Implementér StockPublisher

6. Åbn nu filen "StockPublisher.cs".
7. Lokalisér "TODO: Define StockChanged event"
  - a. Definér en public event kaldet StockChanged, som man kan subscribe til med delegates af typen EventHandler<StockChangedEventArgs>
8. Lokalisér "TODO: Raise StockChanged event"
  - b. Lav nu kode til at fyre StockChanged eventen

- c. For at gøre det trådsikkert, skal der laves en lokal delegate af typen `EventHandler<StockChangedEventArgs>`, der skal indeholde en kopi af `StockChanged`
  - d. Hvis denne lokale delegate ikke er `null`, skal den kaldes med event-argumenter indeholdende
    - i. Navnet på den underliggende aktie
    - ii. Den medfølgende `stockValue`
9. Når du er færdig, bør resultatet ligne

```
// TODO: Define StockChanged event
public event EventHandler<StockChangedEventArgs> StockChanged;

protected void OnStockChanged(double stockValue)
{
    // TODO: Raise StockChanged event
    EventHandler<StockChangedEventArgs> del = StockChanged;
    if (del != null)
    {
        del(this, new StockChangedEventArgs(
            _ticker, stockValue ));
    }
}
```

10. Byg programmet og korriger eventuelle fejl.
11. **Ekstra:** Hvis du kender til de nye features introduceret til C# 6.0, så kan ovenstående skrives meget lettere og kortere... Ellers vent til disse introduceres i Module 08. ☺

### Implementér StockSubscriber

12. Tilføj en ny klasse i en ny fil "StockSubscriber.cs"
- a. Lav en `public` klasse `StockSubscriber`
  - b. Lav på klassen en `public` metode, der hedder `SubscribeTo()`, som returnerer `void` og som parameter tager en instans af `StockPublisher`
    - i. Sørg for, at denne metode subscriber på `StockChanged`-eventen på den medfølgende `StockPublisher` ved at kalde en ny metode, som du laver nedenfor
  - c. Lav på klassen en `private` metode `OnStockChanged()`, der matcher signaturen på `StockChanged`-eventen,
    - i. Udskriv på Console inde i metoden nedenstående streng  
`Stock X was on Y priced at Z`
    - ii. Den underliggende aktiekurs ønskes udskrevet med to decimaler.
13. Når du er færdig, bør resultatet ligne

```

public class StockSubscriber
{
    public void SubscribeTo(StockPublisher p)
    {
        p.StockChanged += OnStockChanged;
    }

    private void OnStockChanged(object sender,
                                StockChangedEventArgs e)
    {
        Console.WriteLine(
            "Stock {0} was on {1} priced at {2:f2}",
            e.Ticker,
            e.TimeStamp,
            e.StockValue );
    }
}

```

14. Byg programmet og korriger eventuelle fejl.

### Sæt alle elementerne sammen

15. Åbn nu filen "Program.cs".

16. Lokalisér "TODO: Create publishers and subscriber"

- a. Lav en StockPublisher med aktienavn "MSFT"
- b. Lav en StockPublisher med aktienavn "WCB"
- c. Lav en StockSubscriber, der abonnerer på de to StockPublisher-instanser
- d. Tilføj til sidst

```
Console.ReadLine();
```

så programmet ikke stopper med den samme

17. Når du er færdig, bør resultatet ligne

```

// TODO: Create publishers and subscriber
StockPublisher p1 = new StockPublisher("MSFT");
StockPublisher p2 = new StockPublisher("WCB");

StockSubscriber subscriber = new StockSubscriber();
subscriber.SubscribeTo(p1);
subscriber.SubscribeTo(p2);

Console.ReadLine();

```

18. Byg programmet og korriger eventuelle fejl.

19. Kør programmet

- e. Hvad ser du?

### Variér konceptet

20. Prøv at udvide programmet med flere subscribers og flere publishers

- a. Hvad sker der nu, når du kører?

21. Hvad sker der mon, hvis du kalder `SubscribeTo()` flere gange på samme `StockSubscriber` med samme `StockPublisher`...? Prøv! 😊

### Øvelse 2.3: "Anonyme metoder og lambda-udtryk" (★)

Et prædikat er en afbildning, som tager et argument af en given type og returnerer en boolean, der angiver om argumentet opfylder prædikatet. I .NET findes indbygget en generisk delegate-type `Predicate<T>`, der er et prædikat af type `T`.

Tilsvarende findes der på `Array`-klassen en statisk, generisk metode `FindAll()`, der som parametre tager et array af type `T` samt et `Predicate<T>` og returnerer et array af bestående af netop de elementer, som opfylder prædikatet.

Tag udgangspunkt i et array defineret ved

```
string[] names = { "Kim", "Mads", "Rasmus", "Bo", "Jesper" };
```

Opgaven er nu – på tre forskellige måder – ved hjælp af prædikater og `Array.FindAll()` at udskrive alle navne i listen med mindst 4 bogstaver:

1. Først vha. en metode, der matcher `Predicate<T>`.
2. Dernæst ved brug af en anonym metode.
3. Slutteligt – og mest elegant – vha. et passende lambda-udtryk.

Til sidst:

4. Kan man lave et lambda-udtryk, som skriver navnene ud mens den filtrerer, så vi ikke behøver skrive det resulterende array ud bagefter? Hvordan...?

(Husk at få en tåre i øjenkrogen af beundring, når du har fundet løsningen. 😊)

## Øvelse 2.4: "Event-problemer i praksis" (☆☆☆)

Events er en fleksibel og elegant implementarion af det klassiske Observer-pattern. Men der er stadig en hel del ting, som man selv i praksis skal håndtere i samtlige event-implementationer. Denne opgave skitserer nogle problematikker, som man med fordel kunne løse én gang for alle i termer af en passende hjælpeklasse, som vi vil kalde EventsHelper.

### Lav event-setup

1. Tag udgangspunkt i det eksisterende projekt "EventTest" i  
C:\Wincubate\87410\Module 2\Lab 2.4\Starter.
  - a. Se NumbersEventArgs-klassen og indse, hvad den indeholder.
  - b. Se og forstå Publisher-klassen
    - i. Implementér Start() således, Numbers korrekt fyres med alle tal-par i data.
  - c. Se og forstå Subscriber-klassen
    - i. Implementér constructoren således, at når den kaldes med "+", en sum-funktion og en given Publisher, så subscribes til sidstnævntes Numbers-event, så der eksempelvis for et data-par (5,7) vha. et passende lambda-udtryk udskrives nedenstående streng ud på konsollen:  
"5+7=12"
  - d. Se og forstå Program-klassen
    - i. Lav en Subscriber add for +
    - ii. Lav en Subscriber sub for -
    - iii. Lav en Subscriber mul for \*
    - iv. Lav en Subscriber div for /
2. Kør programmet!
  - a. Hvad sker der?
  - b. Hvad konkluderer du om sammenhængen mellem event-kald tilbage til Subscriber-instanserne?

### Lav EventsHelper-klasse

3. Lav en klasse EventsHelper med en passende Fire()-metode, som kalder hver af de tilknyttede event-abonnenter korrekt og med indbyrdes "fejl-isolering".
  - a. Lav evt. metoden som en extension-metode.
4. Modificér Publisher-klassen, så den nu benytter den nye EventsHelper.
5. Kør programmet!
  - a. Hvad sker der nu?
6. Hvis du har lyst: Overvej om der kunne være andre features, som man kunne udstyre EventsHelper med...

## Module 3: "LINQ"

### Øvelse 3.1: "Kunder og ordrer i LINQ"

Vi vil i denne opgave beskæftige os med grundlæggende queries i LINQ.

1. Betragt typerne

```
enum CustomerCity { Aarhus, Horsens }

class Customer
{
    public string Name { get; set; }
    public CustomerCity City { get; set; }
    public Order[] Orders { get; set; }
}

class Order
{
    public int Quantity { get; set; }
    public Product Product { get; set; }
}

enum ProductName { Mælk, Smør, Brød, Øl, PepsiMax }

class Product
{
    public ProductName Name { get; set; }
    public double Price { get; set; }
}
```

der alle er at finde i projektet i

C:\Wincubate\87410\Module 3\Lab 3.1\Starter .

2. I samme projekt findes et udtryk af type `List<Customer>` lavet vha. object initializers, der består af nedenstående data

- Kunde: "Kim" fra Aarhus
  - 3 stk. ordrer: Mælk, Smør, Brød
- Kunde: "Mads" fra Horsens
  - 4. stk. ordrer: Mælk, Smør, Brød, Øl
- Kunde: "Jesper" fra Aarhus
  - 1 stk. ordre: Pepsi Max

(med en relevant pris for hver...)

#### Udvælg alle kunder med navn og by

3. Lokalisér "TODO 1: All customers name and city".
4. Skriv en LINQ query, der udvælger alle kunder i `customers`.
5. Udskriv vha. ovenstående query for hver kunde deres navn og by.
6. Byg og kørs programmet
  - Test af din query er korrekt.

#### Udskriv alle kunder fra Aarhus med navn

7. Lokalisér "TODO 2: All customers from Aarhus by name".
8. Udskriv vha. en passende LINQ query alle de kunder i customers, som kommer fra Aarhus

#### Udskriv antallet af ordrer for Kim

9. Lokalisér "TODO 3: Number of orders placed by Kim".
10. Udskriv vha. en passende LINQ query antal order for Kim.



### Øvelse 3.2: "LINQ Queries og extension-metoder" (★)

Ofte kan det være fordelagtigt at kombinere de gængse LINQ-operatorer med udvalgte LINQ extension-metoder til at konstruere sammensatte resultater. Andre gange kan det være lettere at benytte extension-metoderne hørende til de kendte LINQ-keywords. Som regel er valget baseret på smag og behag.

Som data-grundlag i denne opgave vil vi benytte listen af spilnavne til Wii og Xbox 360.

1. Åbn skabelonen med det beskrevne data-grundlag i projektet i  
C:\Wincubate\87410\Module 3\Lab 3.2\Starter .

#### Udskriv alle Wii-spil sorteret efter titel

2. Lokalisér "TODO 1: All Wii games sorted by title".
3. Udskriv vha. en passende LINQ query alle Wii-spil sorteret alfabetisk efter titel.

#### Udskriv alle Wii-spil sorteret efter titel-længde med længste først

4. Lokalisér "TODO 2: All Wii games sorted by title length (longest title first)".
5. Udskriv vha. en passende LINQ query alle Wii-spil aftagende efter titel-længde.

#### Udskriv med store bogstaver alle spil på enten Wii eller Xbox

6. Lokalisér "TODO 3: All games for either machine sorted by title".
7. Udskriv vha. en passende LINQ query med store bogstaver alle spil til enten Wii eller Xbox 360 (uden dubletter!)

#### Udskriv alle spil til Wii, men ikke til Xbox, dog på nær dem med "Wii"

8. Lokalisér "TODO 4: All games for Wii but not for Xbox 360 except those with 'Wii' in the title".
9. Udskriv vha. en passende LINQ query de spil som findes til Wii men ikke Xbox – dog fraregnet dem, som indeholder "Wii" i titlen.

### Øvelse 3.3: "Flere kunder og ordrer i LINQ" (☆☆)

Denne øvelse er en fortsættelse af Øvelse 3.1, hvor strukturen på de indgående queries dog er en smule mere komplicerede.

- Åbn skabelonen med de velkendte kunde-data i projektet i  
C:\Wincubate\87410\Module 3\Lab 3.3\Starter .

#### Udskriv alle kunder der har købt mælk

- Udskriv vha. en passende LINQ query navnene på alle de kunder, som har købt mælk

#### Udskriv total forbrugt ordresum

- Udskriv vha. en passende LINQ query den totale sum brugt på ordrene for all kunder tilsammen

#### Udskriv forbrugt ordresum for hver kunde individuelt

- Udskriv vha. en passende LINQ query hver individuelle kundes sum brugt på vedkommendes ordrer

#### Udskriv kunder grupperet pr. by

- Udskriv vha. en passende LINQ query alle kunder grupperet efter, hvilken by de kommer fra.

### Øvelse 3.4: "Find første 25 Fibonacci-tal, som 4 går op i" (☆☆☆)

Fibonacci-tallene er en talrække, hvor hvert element er defineret ud fra de to foregående elementer i rækken. Mere præcist er det  $i$ 'te Fibonacci-tal for  $i \geq 0$  defineret ved

- $\text{Fib}(1) = 1$
- $\text{Fib}(2) = 1$
- $\text{Fib}(i) = \text{Fib}(i-2) + \text{Fib}(i-1)$ , for  $i \geq 3$ .

Lav en klasse Fibonacci, der kan query'es vha. LINQ, og lav et program, der på skærmen udskriver de første 25 Fibonacci-tal, som 4 går op i.

### Øvelse 3.5: "Aggregering med LINQ to XML" (★)

Denne øvelse beskæftiger sig med brug af LINQ to XML queries til aggregering af XML data.

- Åbn det eksisterende konsolprojekt i  
C:\Wincubate\87410\Module 3\Lab 3.5\Starter .
- Programmet indeholder en skabelon med en XML-fil i CustomersOrders.xml.

Skriv nu kode i programmet, således at der for hver kunde i XML-filen beregnes summen af Freight for alle denne kundes ordrer tilsammen.

Udskriv også resultatet af queryen så det kan ses, at dit resultat er korrekt. 😊

### Øvelse 3.6: "En tur igennem Entity Framework" (☆☆)

**NB:** Denne øvelse kan kun laves i tilfælde af, at du har SQL Server eller SQL Server Express på din kursusmaskine.

Denne øvelse beskæftiger sig med brug af LINQ to Entities gennem Entity Framework.

- Restore eksempel-databasen i  
C:\Wincubate\87410  
på en database-server, som du har adgang til.
- Eksperimentér med Entity Framework ved at genskabe de skridt, som benyttedes ved den fælles gennemgang.



## Module 4: “Overloading Operators”

### Øvelse 4.1: “Indexere på kortspil”

Formålet med denne øvelse er at illustrere, hvorledes man kan introducere indexere på en type af kortspil.

- Tag udgangspunkt i programmet, der er at finde i  
C:\Wincubate\87410\Module 4\Lab 4.1\Starter
  - Indse, hvordan typerne Card og Deck fungerer.

#### Implementér en heltals-indexer på Deck

- Lav i filen “Deck.cs” en heltals-indexer på Deck, der
  - har både en `get;` og `set;`
  - tager et indeks af typen `int` kaldet `index`
  - returnerer en værdi af type `Card`
    - Den værdi, som findes på indeks `index` i kortspillet
  - kaster en `IndexOutOfRangeException` med en passende informativ tekst, hvis det anvendte indeks er udenfor det underliggende array af kort.

#### Implementér en streng-indexer på Deck

- Lav ligeledes i filen “Deck.cs” en streng-indexer på Deck, der
  - kun har en `get;`
  - tager et indeks af typen `string` kaldet `s`
  - returnerer en værdi af type `int`
    - Det indeks i kortspillet (hvis det findes), hvor `Card.ToString()` er netop `s`
  - kaster en `IndexOutOfRangeException` med en passende informativ tekst, hvis det anvendte indeks ikke findes i det underliggende array af kort.

#### Test dine implementationer

- Test din implementation ved at tilføje kode til `Main()`, der tester begge indexers ovenfor.

## Øvelse 4.2: "Operatorer på Money"

I denne øvelse vil vi undersøge, hvordan man selv kan overskrive en lang række operatorer og metoder på egne typer. Vi vil definere en ny type Money, der består af euro og cents og gøre den maksimalt fleksibel og brugbar vha. et antal operator-overskrivninger.

- Tag udgangspunkt i en skabelon til Money, der er at finde i  
C:\Wincubate\87410\Module 4\Lab 4.2\Starter
  - Verificér, at struct Money findes i "Money.cs"
  - Verificér, at der findes test-kode til alle operatorer i "Program.cs"
    - Denne kode kan indkommenteres undervejs eller til sidst for at prøve din implementation.

### Implementér constructors og hjælpemetoder for Money

- Find først "TODO 1: Normalize to regular euro and cents" og implementér først koden i constructoren til at "normalisere" Euro og Cents
  - Denne kode skal "normalisere" til lovlige beløb. Eksempelvis bliver 9 Euro og 176 Cents til 10 Euro og 76 Cents osv.
- Nu kan vi så lave en constructor, der kun tager et antal cents (eksempelvis enten 65, 176 eller 2010). Så find derfor "TODO 2: Implement cents-constructor by chaining" i koden og lav den her ved at kalde den allerede eksisterende constructor, der tager to parametre.
- Find nu "TODO 3: Implement total cents property" og lav en hjælpe-property  
`public int TotalCents { get; }`  
der f.eks. returnerer 2010 for en Money-værdi med 20 Euro og 10 Cents.
- Implementér nu `IComparable<Money>` via en særdeles nyttig `CompareTo()`-metode, som vi kan benytte senere
  - Du kan benytte `TotalCents()` ovenfor til at sammenligne.

### Overskriv aritmetiske operatorer på Money

- Overskriv nu nedenstående aritmetiske operatorer på Money
  - `+(Money, Money)` lægger to Money sammen
  - `+(Money, int)` lægger et antal cents til Money
  - `-(Money, Money)` trækker to Money fra hinanden
  - `-(Money, int)` trækker et antal cents fra Money
  - `++` tæller Money én cent op
  - `--` tæller Money én cent ned

### Overskriv virtuelle metoder på Money

- Overskriv nu nedenstående virtuelle metoder på Money
  - `Money.ToString()` giver en streng-repræsentation af Money
  - `Money.Equals(object obj)` undersøger om de Money og obj er de samme (Bemærk dog, at da Money er en struct er dette strengt taget ikke nødvendigt her)
  - `Money.GetHashCode()` returnerer en unik hash-værdi



### Overskriv relationelle operatorer på Money

- Overskriv nu nedenstående relationelle operatorer på Money
  - >
  - <
  - >=
  - <=
  - ==
  - !=

### Overskriv konverteringsoperatorer på Money

- Konstruér nu nedenstående konverteringsoperatorer på Money
  - (Explicit) Money fra int cents
  - (Explicit) Money fra float cents
  - (Implicit) int fra Money
  - (Explicit) float fra Money
  - (Implicit) string fra Money

### Test din implementation

- Test nu endeligt din implementation vha. koden i Main().



## Module 5: "Object Lifetime"

### Øvelse 5.1: "Hold øje med garbage collection"

I denne øvelse vil vi prøve at få en fornemmelse af, hvornår garbage collectoren sætter ind for at rydde objekter op. Vi vil definere en klasse A, der kan tælle antallet af sine instanser. Dette antal vil vi så løbende skrive ud, så vi kan følge lidt med i, hvornår der ryddes op.

#### Lav ny klasse

1. Lav et projekt i Visual Studio 2015 til ny C# konsol-applikation
  - a. Kald projektet "ObjectCounter" og
  - b. Placer det i  
C:\Wincubate\87410\Module 5\Lab 5.1\Starter
2. Tilføj en ny C#-fil med navn "A.cs" og definér en klasse A i denne fil
  - a. Lav en `public static int` property på A og kald denne `InstanceCount`
    - i. Lav `set` `private`
  - b. Lav en statisk constructor, der sætter `InstanceCount` til 0
  - c. Definér en default constructor, som tæller `InstanceCount` én op
  - d. Definér en destructor, som tilsvarende tæller `InstanceCount` én ned
    - i. På denne måde vil `InstanceCount` hele tiden indeholde antallet af A-instanser i applikationen
3. Når du er færdig, bør resultatet ligne

```
class A
{
    public static int InstanceCount { get; private set; }

    static A()
    {
        InstanceCount = 0;
    }

    public A()
    {
        InstanceCount++;
    }

    ~A()
    {
        InstanceCount--;
    }
}
```

4. Byg programmet og korriger eventuelle fejl.

#### Generér objekter

5. I `Program.cs` skal `Main()` skrives, så
  - a. Der laves en uendelig `while`-løkke indeholdende

- i. Udskrift af teksten  
Press ENTER to generate objects
- ii. En efterfølgende Console.ReadLine(), der venter på tryk på Enter
- iii. En efterfølgende for-løkke, der genererer 10000 instanser af A, som med det samme er i spil til at blive garbage collected
  1. Lav en lokal instans i hver iteration af for-løkken
- iv. Udskrift af teksten  
There are currently  $n$  instances of A  
hvor  $n$  fås fra InstanceCount.

6. Når du er færdig, bør resultatet ligne

```
static void Main(string[] args)
{
    while (true)
    {
        Console.WriteLine("Press ENTER to generate objects");
        Console.ReadLine();

        for (int i = 0; i < 10000; i++)
        {
            A a = new A();
        }

        Console.WriteLine("There are currently {0} instances of A",
            A.InstanceCount);
    }
}
```

7. Byg programmet og korriger eventuelle fejl.
8. Kør programmet og tryk Enter nogle gange, mens du observerer resultaterne.
9. Find forklaringen på, hvad det er, der sker
  - b. Hvad sker der med antallet af A-instanser?

Kan I mon observere noget mærkeligt...? ☺ (Se evt. Øvelse 7.3)

## Øvelse 5.2: "Implementering af IDisposable" (★)

En gang imellem vil de klasser, som man får skrevet, skulle eksplicit ryddes op, fordi de benytter "eksterne" ressourcer som helst skal lukkes og håndteres med det samme, da det kan tage timer og endda dage før garbage collectoren næste gang kører. Dette er tilfældet med klassen `FileWriter`, der er udgangspunktet for denne opgave. Vi vil her håndtere oprydningen ved at bringe klassen til at implementere `IDisposable`.

### Inspicér `FileWriter`

1. Åbn det eksisterende OOP projekt i Visual Studio 2015 i  
`C:\Wincubate\87410\Module 5\Lab 5.2\Starter`  
hvilket indeholder et simpelt program baseret på `FileWriter`.
2. Byg programmet
  - a. Indse, at programmet laver en instans af `FileWriter`, kalder dens `Log()`-metode et antal gange, hvilket hver gang skriver et timestamp ned i filen `"FileWriter.txt"`
  - b. Herefter er der ikke længere brug for `FileWriter`-instansen, og programmet venter på et tryk på Enter, hvorefter det lukker ned.
3. Kør programmet
  - a. Lokalisér file `"FileWriter.txt"` mens programmet stadig venter på tryk på Enter.
  - b. Prøv at åbne filen.
  - c. Hvad sker der? Hvorfor?
4. Indse, at `FileWriter` bør implementere `IDisposable`, så filen lukkes, når der ikke længere er brug for den.

### Implementér `IDisposable`

5. Lad nu `FileWriter` implementere `IDisposable`-interfacet.
6. Følg nu den i materialet beskrevne vejledning til, hvordan man bør implementere `IDisposable`
  - a) Tilføj en private boolean `_disposed`, der initielt sættes til `false`
  - b) Tilføj en private metode, der hedder `CleanUp()`, der som argument tager en boolean `disposing` og returnerer `void`
    - I `CleanUp()` skal der tilføjes en if-sætning, der spørger, om `_disposed` er `false`
      - Inde i denne if-sætning skal der – hvis `disposing` er `true` – ryddes `_fs` op
    - Efter if-sætningen sættes `_disposed` til `true`
  - c) Tilføj en public `Dispose()`-metode, der
    - Kalder `CleanUp()` med `true` som parameter
    - Kalder `GC.SuppressFinalize()` med `this` som parameter
  - d) Tilføj en class destructor for `FileWriter`, der
    - Kalder `CleanUp()` med `false` som parameter
  - e) Den eksisterende `Log()`-metode skal bringes til at kaste en `ObjectDisposedException` med teksten `Object is already disposed!` hvis `_disposed` er `true`, når metoden kaldes.
7. Når du er færdig, bør resultatet ligne

```

class FileWriter : IDisposable
{
    private bool _disposed = false;
    protected FileStream _fs;

    public FileWriter()
    {
        _fs = File.Create(@"FileWriter.txt");
    }

    ~FileWriter()
    {
        CleanUp( false );
    }

    public void Dispose()
    {
        CleanUp(true);
        GC.SuppressFinalize(this);
    }

    private void CleanUp(bool disposing)
    {
        if (_disposed == false)
        {
            if (disposing)
            {
                // Dispose managed here
                _fs.Dispose();
            }

            // Clean up unmanaged here.
        }
        _disposed = true;
    }

    public void Log()
    {
        if (_disposed == true)
        {
            throw new ObjectDisposedException(
                "Object is already disposed!" );
        }

        string s = DateTime.Now.ToLongTimeString() +
            Environment.NewLine;
        _fs.Write(Encoding.ASCII.GetBytes( s ),
            0,
            s.Length );
    }
}

```

8. Byg programmet og korriger eventuelle fejl.
9. I Program.cs skal Main() opdateres, så FileWriter-klassen nu benyttes korrekt og disposes efter brug
  - f) Benyt using-konstruktionen til at sikre, at `fileWriter.Dispose()` kaldes efter `Log()`-metoderne er kaldt.
10. Når du er færdig, bør resultatet ligne

```
using( FileWriter fileWriter = new FileWriter() )
{
    fileWriter.Log();
    fileWriter.Log();
    fileWriter.Log();

    // FileWriter is no longer needed
}
```

11. Byg programmet og korriger eventuelle fejl.
12. Kør igen programmet, som nu er modificeret
  - g) Lokalisér igen file "FileWriter.txt" mens programmet stadig venter på tryk på Enter.
  - h) Prøv igen at åbne filen.
  - i) Hvad sker nu der? Hvorfor?





## Module 6: "Dynamic Types"

### Øvelse 6.1: "ExpandoObject" (☆☆)

Formålet med denne øvelse er at undersøge `ExpandoObject` i `System.Dynamic` namespaceset for at se, hvorledes `dynamic`-keywordet og dynamiske typer virker.

- Lav et nyt projekt `MondoExpando` i  
C:\Wincubate\87410\Module 6\Lab 6.1\Starter .

#### Opbyg `ExpandoObject` med kontakt-informationer

- Lav et passende `ExpandoObject` kaldet `contact` med
  - Fornavn
  - Efternavn
  - Telefonnummer
  - Adresse
    - Adresse skal være et nyt `ExpandoObject` med
      - Vej
      - Nummer
      - Postnummer
      - By
- Udskriv oplysningerne pænt formatteret.

#### Implicitte typer vs. Dynamiske typer

- Eksperimentér med at bytte `dynamic` ud med `ExpandoObject` eller `var`.
  - Hvad sker der?
- Kan man bruge `ToString()` til udskrift?

#### Manipulation af members på runtime... ☺

- Hvordan sletter man `members` på runtime?
  - Advarsel: Dette er ikke let at lure!
- Kan man `members` endda iterere over `members` på runtime?
  - Advarsel: Dette er heller ikke let at lure!

## Øvelse 6.2: "Elegant brug af dynamic-keyword" (☆☆)

Denne øvelse illustrerer et groft undervurderet aspekt af dynamisk typning.

- Åbn det eksisterende konsolprojekt i  
C:\Wincubate\87410\Module 6\Lab 6.2\Starter.

Programmet indeholder en skabelon, der laver en svagt typet liste indeholdende et mix af forskellige objekter af forskellige typer

```
List<object> mixOfObjects = new List<object>()  
{  
    true, 87, "Hello World", 176.0  
};
```

### Dynamic Power of One ;-)

- Modificér programmet ét eneste sted, således at programmet ved kørsel udskriver følgende

```
Handling bool...      True  
Handling int...       87  
Handling string...    Hello World  
Handling double...    176
```

### Variation af konceptet...

- Hvad sker der mon, hvis du udkommenterer metoden `void Handle( int o )`?
  - Prøve at gætte inden du kører programmet og tester ☺
- Hvad sker der mon, hvis du tilføjer en metode med `void Handle( object o )`?
  - Prøve igen at gætte inden tester ☺

## Module 7: "Asynchronous Programming"

### Øvelse 7.1: "Tasks" (☆☆)

Denne øvelse handler om at skrive tasks, asynkrone metoder samt benytte await. Vi vil lave et program, der starter tre tasks, som hver især henter indholdet af en avis-webseite, og regner den samlede længde ud.

- Åbn programskeletonen til Windows-applikationen, der er at finde i  
C:\Wincubate\87410\Module 7\Lab 7.1\Starter .
- Implementér først metoden  
`Task<string> CreateFetchTaskAsync( string url )`  
der laver en instans af WebClient og downloader strengen på den givne url vha.  
`WebClient.DownloadStringTaskAsync()`.
- Implementér så  
`async Task<int> ComputeLengthSumAsync()`  
ved at udfylde TODO-delen (og tilføj asynkron modifier!), så metoden er udgør en task, der – når alle del-tasks er færdige – udskriver den samlede længde af strenge downloadet af task t1, t2 og t3.
- Implementér til sidst  
`void OnComputeClick( object sender, RoutedEventArgs e )`  
så den kalder `ComputeLengthSum()` og afventer, at resultatet er beregnet færdigt, og derefter opdaterer brugerflade-kontrollen.

## Øvelse 7.2: "Håndtering af flere Tasks" (☆☆☆)

Denne øvelse undersøger samspillet mellem tasks.

- Åbn det eksisterende Windows-projekt i  
C:\Wincubate\87410\Module 7\Lab 7.2\Starter .

Projektet indeholder et projekt med en brugerflade, der indeholder en knap samt et status-tekstfelt. Ydermere indeholder `MainWindow.xaml.cs` en metode `HeavyWork()`, der simulerer et stykke længerevarende beregning, der tager mellem 0 og 10 sekunder om at udføre. Trykkes knappen, udføres metoden `OnClick()`.

- Åbn `MainWindow.xaml.cs` og udfyld den nødvendige kode i de forskellige TODOs
  - TODO 1: Start tre udgaver af `HeavyWork()`, så de kører sideløbende.
  - TODO 2: Såfremt én eller flere af disse tasks fra TODO 1 stadig kører efter 7 sekunder, vis da en passende streng som indikation heraf i tekst-feltet på brugerfladen.
  - TODO 3: Når alle tasks fra TODO 1 er færdige, skriv da dette i tekstfeltet.
- Bemærk: Du må kun bruge klasserne `Task`, `Task<T>` samt metoder på disse klasser. ☺

### Øvelse 7.3: "Låsning vha. Lock" (☆☆)

Denne øvelse beskæftiger sig med tråd-sikkerhed af programmer.

- Åbn det eksisterende konsol-projekt i  
C:\Wincubate\87410\Module 7\Lab 7.3\Starter .

Programmet benytter constructors og destructors til løbende at tælle antal levende instanser af en givet klasse.

- Indse, hvordan programmet virker. Er det trådsikkert? Hvorfor (ikke)?

Til at trådsikre C#-programmer, kan man med fordel benytte det indbyggede keyword `lock` i C#. Læs eventuelt om dette keyword i [Troelsen]. Som en lille hjælp kan det dog siges, at adgangen til de statiske members i en klasse, kan beskyttes med programstumpen nedenfor

```
lock( typeof( A ) )  
{  
    // Tilgå A's statisk members her  
}
```

- Lav nu en version af ObjectCounter-programmet ovenfor, som du er sikker på, er trådsikkert.



## Module 8: "New Features in C# 6.0"

### Øvelse 8.1: "Auto-properties, Elvis-operator og String Interpolation"

Denne øvelse går ud på at vise, hvorledes eksisterende C# 5.0-programmer kan skrives pænere, kortere og mere overskueligt vha. de nyeste C# 6.0-features.

- Åbn det eksisterende konsol-projekt i  
C:\Wincubate\87410\Module 8\Lab 8.1\Starter ,

hvilket er løsningen fra Øvelse 2.2 omhandlende events.

- Skriv programmet så "lækkert" som muligt vha. getter-only auto-properties med tilhørende initializers
  - Benyt også både ?. og string interpolation, hvor det giver mening

## Module 9: “An Introduction to C# 7”

### Lab 9.1: “Basic Tuples”

This exercise implements a simple function using tuples for computing compound values.

- Open the starter project in  
*PathToCourseFiles\Labs\Module 9\Lab 9.1\Starter* ,  
which contains a project called BasicTuples.

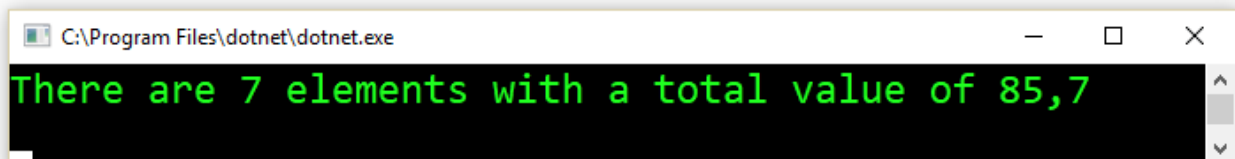
The Main() method contains the following code:

```
IEnumerable<decimal> numbers = new List<decimal>
{
    4.2m, 8.7m, 17.6m, 11.2m, 25.5m, 7.5m, 11.0m
};

// TODO

Console.WriteLine( $"There are {count} elements with a total value
    of {total}");
```

You should use tuples to complete the TODO above such that the program will print the following when run:



You should complete your task by

- Defining a method Tally()
  - accepting an argument of `IEnumerable<decimal>` and
  - returning an appropriate tuple type
- Replace the TODO with just a single line invoking the Tally() method and handling the return values appropriately with changing any other line in Main().



## Lab 9.2: "Object Deconstruction" (★)

The purpose of this exercise is to implement object destruction to tuples of a preexisting class.

- Open the starter project in  
*PathToCourseFiles\Labs\Module 9\Lab 9.2\Starter* ,  
which contains a project called *ObjectDestruction*.

The starter solution consists of two projects – a client project and a class library called *DiscographyLab*.

The class library contains an existing class class *Album*. That class is part of an externally supplied API and cannot be modified or derived from:

```
public sealed class Album
{
    public Guid Id { get; }
    public string Artist { get; }
    public string AlbumName { get; }
    public DateTime ReleaseDate { get; }

    public Album( string artist, string albumName,
                 DateTime releaseDate )
    {
        Id = Guid.NewGuid();
        Artist = artist;
        AlbumName = albumName;
        ReleaseDate = releaseDate;
    }
}
```

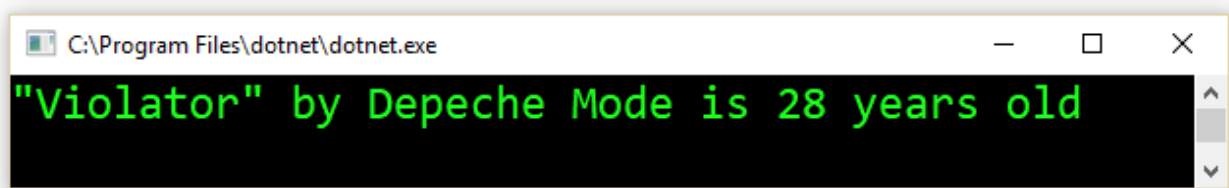
The client project contains a *Main()* method with the following code:

```
Album album = new Album(
    "Depeche Mode",
    "Violator",
    new DateTime( 1990, 3, 19 )
);

( _, string summary, int age ) = album;
Console.WriteLine( $"{summary} is {age} years old" );
```

Currently this code does not compile. You need to fix that.

- Your task is to add an appropriate class and method to make the code compile.
  - Note: You should not change anything in the *Main()* method or the *Album* class.
- When completed, your *Main()* method should produce the following output:



```
C:\Program Files\dotnet\dotnet.exe  
"Violator" by Depeche Mode is 28 years old
```

### Lab 9.3: "Pattern Matching Shapes" (★)

The purpose of this exercise is to use pattern matching for distinguishing shapes and computing their respective areas.

- Open the starter project in  
*PathToCourseFiles\Labs\Module 9\Lab 9.3\Starter* ,  
which contains a project called MatchingShapes.

The project contains two predefined shape structs. Circle.cs contains

```
struct Circle
{
    public double Radius { get; }

    public Circle( int radius ) => Radius = radius;
}
```

and Rectangle.cs has this definition:

```
struct Rectangle
{
    public double Width { get; }
    public double Height { get; }

    public Rectangle( int width, int height )
    {
        Width = width;
        Height = height;
    }
}
```

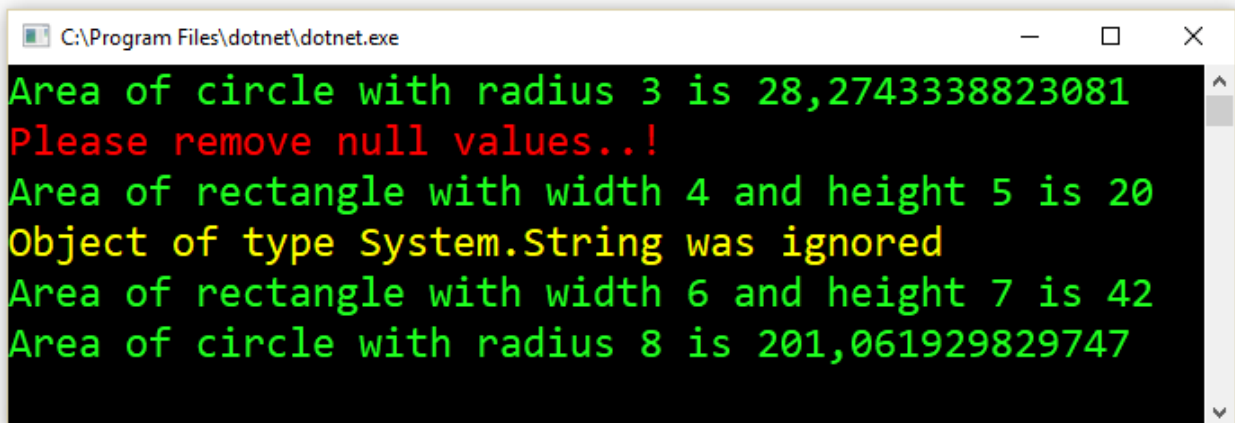
The Main() method contains the following data and method call:

```
List<object> objects = new List<object>
{
    new Circle( 3 ),
    null,
    new Rectangle( 4, 5 ),
    "Not really a shape",
    new Rectangle( 6, 7 ),
    new Circle( 8 )
};

objects.ForEach(ComputeArea);
```

Your task is to use pattern matching to figure out which area calculation is to be employed for each particular object.

- Complete the ComputeArea() method in Program.cs appropriately such that the program will produce the following output:



A screenshot of a Windows command prompt window. The title bar at the top reads "C:\Program Files\dotnet\dotnet.exe". The window contains the following text output:

```
Area of circle with radius 3 is 28,2743338823081
Please remove null values..!
Area of rectangle with width 4 and height 5 is 20
Object of type System.String was ignored
Area of rectangle with width 6 and height 7 is 42
Area of circle with radius 8 is 201,061929829747
```

The text is color-coded: the first line is green, the second line is red, the third line is green, the fourth line is yellow, the fifth line is green, and the sixth line is green. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

## Lab 9.4: "More Tuples, Pattern Matching, and a Local Function" (★)

This exercise extends the solution of Lab 01.1 to implement processing of recursive sequences of numbers using a pattern matching technique and a local function to assist it.

- Open the starter project in  
`PathToCourseFiles\Labs\Module 09\Lab 9.4\Starter` ,  
which is essentially the solution to Lab 01.1. containing a project called `MoreTuples`.

However, now the `Main()` method contains the following code:

```
IEnumerable<object> numbers = new List<object>
{
    4.2m, 8.7m, new object[] { 17.6m, 11.2m, 25.5m },
    7.5m, new List<object> { 11.0m }
};

var (count, total) = Tally(numbers);
Console.WriteLine( $"There are {count} elements with a total value
                    of {total}");
```

The data has now been generalized from `IEnumerable<decimal>` to `IEnumerable<object>`, and the data is now “recursive” in the sense that some elements of the object sequence are – in turn – an object sequence. Apart from that, `Main()` has not been modified.

The signature of the `Tally()` method you completed in Lab 01.1 has been generalized accordingly as follows:

```
static (int count, decimal total) Tally( IEnumerable<object> data )
{
    (int count, decimal total) tuple = (0, 0);

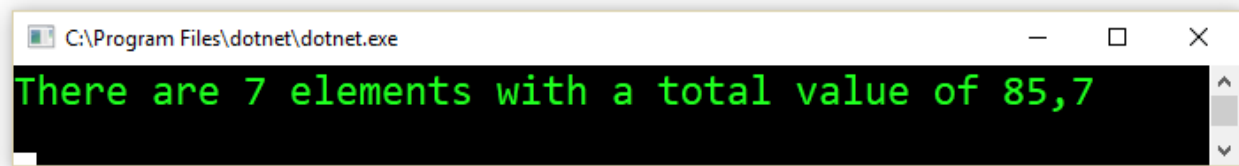
    // TODO

    return tuple;
}
```

Your task is now to use pattern matching and a local helper function to complete the updated `Tally()` method to correctly recursively compute the tally of the sequence.

- Within `Tally()` introduce a local helper function called `Update()` which updates the local tuple variable by adding a subcount and a subtotal to the constituent tuple values.
- Use a switch statement with pattern matching to process the sequence (and subsequence ) elements
  - Use `Update()` in each of the cases to update tuple.

As before, your completed program should produce the following output:



A screenshot of a Windows command prompt window. The title bar at the top reads "C:\Program Files\dotnet\dotnet.exe". The window has standard Windows window controls (minimize, maximize, close) on the right. The main area of the window has a black background with green text that says "There are 7 elements with a total value of 85,7". A vertical scrollbar is visible on the right side of the text area.

```
C:\Program Files\dotnet\dotnet.exe  
There are 7 elements with a total value of 85,7
```

## Lab 9.5: "Dictionaries and Tuples" (★)

This exercise illustrates a simple, but neat trick for composite keys in dictionaries.

- Open the starter project in  
*PathToCourseFiles\Labs\Module 9\Lab 9.5\Starter* ,  
which contains a project called KeyToAwesomeness.

The project defines two enumeration types

```
enum CoffeeKind
{
    Latte,
    Cappuccino,
    Espresso
}
```

and

```
enum CoffeeSize
{
    Small,
    Regular,
    Large
}
```

A coffee consists of a `CoffeeKind`, a `CoffeeSize`, and a strength between 1 and 5. The `Main()` method contains some very simple code serving 100 random coffees to random customers:

```
void Serve( string customerName, CoffeeKind kind, CoffeeSize size,
            int strength )
{
    Console.WriteLine($"Serving a {size} {kind} of strength {strength}
                       to {customerName}");
}

RandomHelper helper = new RandomHelper();

for (int i = 0; i < 100; i++)
{
    CoffeeKind kind = helper.GetRandomCoffeeKind();
    CoffeeSize size = helper.GetRandomCoffeeSize();
    int strength = helper.GetRandomCoffeeStrength();

    Serve(helper.GetRandomName(), kind, size, strength);
}

Console.WriteLine();
```

When run the program produces a number of output lines like the following:

```
C:\WINDOWS\system32\cmd.exe

Serving a Regular Cappuccino of strength 1 to Ane Olsen
Serving a Small Cappuccino of strength 2 to Maria Sana
Serving a Large Latte of strength 5 to Nils Christensen
Serving a Large Espresso of strength 1 to Nils Gulmann
Serving a Small Latte of strength 5 to Ane Riel
Serving a Regular Latte of strength 5 to Bo Mortensen
Serving a Small Cappuccino of strength 4 to Noah Leth
Serving a Small Cappuccino of strength 2 to Nina Kirk
Serving a Regular Latte of strength 3 to Jesper Thomassen
Serving a Small Espresso of strength 2 to Jørgen Olsen
Serving a Regular Latte of strength 1 to Nina Thomassen
Serving a Large Cappuccino of strength 2 to Heidi Kirk
Serving a Small Espresso of strength 1 to Bo Henriksen
Serving a Small Latte of strength 3 to Maria Gulmann
```

However, the coffee shop would like to print a summary of all the coffee served, i.e. how many coffees were served of each combination of a `CoffeeKind`, a `CoffeeSize`, and a strength.

They would like to augment the program with a `PrintSummary()` method which provides a summary like:

```
C:\WINDOWS\system32\cmd.exe

Served 6 Regular Latte of strength 4
Served 5 Large Latte of strength 2
Served 4 Small Latte of strength 3
Served 4 Regular Cappuccino of strength 5
Served 4 Regular Cappuccino of strength 2
Served 4 Regular Espresso of strength 3
Served 3 Large Latte of strength 5
Served 3 Small Latte of strength 2
Served 3 Large Cappuccino of strength 3
Served 3 Large Cappuccino of strength 1
Served 3 Regular Cappuccino of strength 4
Served 3 Regular Cappuccino of strength 1
Served 3 Small Cappuccino of strength 5
```

Your task will be to produce this result in a simple manner.

- Augment the `Serve()` method with a means for counting how many coffees were served for each combination of kind, size, and strength



- Define a `PrintSummary()` method outputting a number of strings to the console as illustrated
  - Sort first by the count of coffees served (from high to low)
  - Sort secondly by kind (from first to last)
  - Sort thirdly by size within that kind (from largest to smallest)
  - Use the strength as the final sort criterion (from strongest to weakest).

## Lab 9.6: "Maximum Subsum Problem" (☆☆☆)

In this brain teaser you will employ tuples inside a LINQ statement to solve the maximum subsum problem, which is a thoroughly studied algorithmic problem within the theory of computer science.

Any finite sequence,  $s$ , of integers of length  $n$ , has a number of distinct subsequences of length at most  $n$ .

As an example, consider the sequence

2, -3, 7, 1, 4, -6, 9, -8

Here, the subsequences include (among many, many others) e.g.

2, -3, 7, 1, 4

-3, 7

-6

...

Note that the empty sequence as well as the entire original sequence are both legal subsequences.

Each such subsequence can be viewed as defining a subsum of  $s$  obtaining by adding all the integers of the subsequence. The subsums for the example subsequences above are, resp.:

2 + (-3) + 7 + 1 + 4 = 11

-3 + 7 = 4

-6 = -6

Note: The sum of the empty sequence is 0.

The maximum subsum for the example sequence above is 15 – illustrated by the sequence highlighted in green.

And with that, let's finally get on to the exercise itself..!

- Open the starter project in  
*PathToCourseFiles\Labs\Module 9\Lab 9.6\Starter* ,  
which contains a project called `MaximumSubsumProblem`.

In the starter project you will find the following code inside of `Main()`:

```
IEnumerable<int> sequence = new List<int> { 2, -3, 7, 1, 4, -6, 9, -8 };

// TODO
int result = ...;

Console.WriteLine( $"Maximum subsum is {result}");
```

Your task is to replace the "... " with a single (but relatively complex) LINQ statement containing tuples to compute the maximum subsum of the sequence.

In more detail,

- Create a single LINQ query computing the maximum subsum of a specific sequence supplied as a `IEnumerable<int>`

- Use tuples inside of the single LINQ query as computational state
- Make sure you compute the maximum subsum in linear time (in the length of the sequence).

Uncle Google is probably your friend here... 😊

## Module 10: “What’s New in C# 7.x?”

### Lab 10.1: “Upgrading Tuples”

The topic of this exercise is to bring tuples equality to work in an existing C# 7 project.

- Open the starter project in  
*PathToCourseFiles\Labs\Module 10\Lab 10.1\Starter* ,  
which contains a project called UpgradingTuples.

The Main() method contains the following code:

```
(int x, int y) tuple = (8, 4);  
var other = (a: 8, b: 4);  
  
WriteLine( tuple == other );
```

Unfortunately, this does not compile in the current project.

- Make this program compile...! 😊

## Lab 10.2: "Ref Locals and Ref Returns" (★)

This exercise provides some additional infrastructure to an existing project by using refs.

- Open the starter project in  
*PathToCourseFiles\Labs\Module 10\Lab 10.2\Starter* ,  
which contains a project called *TrackingCurrentAverage*.

The `Main()` method contains the following code:

```
MeasureUnit unit = new MeasureUnit();  
// TODO 3: initialize currentAverage variable to be read repeatedly  
// in loop  
  
unit.Start();  
  
while (true)  
{  
    _ = Console.ReadLine();  
  
    // TODO 4: Comment in the line below to read currentAverage  
    Console.WriteLine($"Average read: {currentAverage:f2}"  
        + Environment.NewLine);  
}
```

Essentially, this code instantiates a `MeasureUnit` instance which at random intervals produces a reading between 0 and 100 when started. The main loop – from now to eternity – awaits the user pressing ENTER to force a reading of the `MeasureUnit`'s current average.

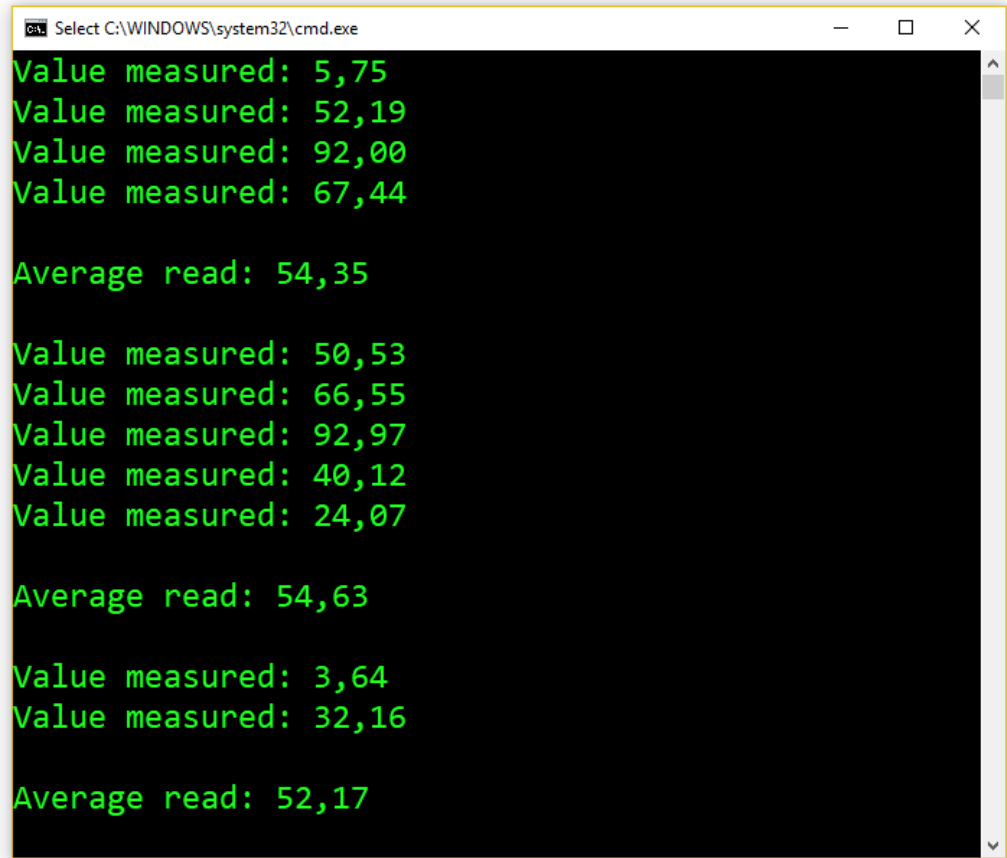
The current average should consist of the latest 10 measured values of the `MeasureUnit`. It should be exposed as an appropriate property of the class to be invoked only once – and then subsequently consulted for the current average whenever the user presses ENTER.

Your task is now to use ref locals and ref returns appropriately to fill in all the pieces missing from the `MeasureUnit` class.

- Inspect the `MeasureUnit` class to familiarize yourself with the existing code in it.
- Locate **TODO 1**:
  - Define an appropriate `CurrentAverage` property to be continuously kept up-to-date
  - Define the data storage needed for the storage of the latest 10 measurement values read.
- Locate **TODO 2**:
  - Implement the `Add()` method appropriately
  - Calculate the average of the last 10 values and keep `CurrentAverage` up-to-date
    - Note: Do not care about thread-safety when updating.
- Locate **TODO 3**:
  - Declare and initialize an appropriate `currentAverage` local variable to be read repeatedly within the loop.
- Locate **TODO 4**:

- Comment in the line reading the currentAverage local variable and printing the result
- Don't change anything else within Main()..!

When your task is completed, a successful execution could result in a screen dump similar to the following:



```
Select C:\WINDOWS\system32\cmd.exe

Value measured: 5,75
Value measured: 52,19
Value measured: 92,00
Value measured: 67,44

Average read: 54,35

Value measured: 50,53
Value measured: 66,55
Value measured: 92,97
Value measured: 40,12
Value measured: 24,07

Average read: 54,63

Value measured: 3,64
Value measured: 32,16

Average read: 52,17
```

## Lab 10.3: "Enum Constraints" (☆☆)

This exercise is concerned with the new generic Enum constraint in C# 7.3.

- Open the starter project in  
*PathToCourseFiles\Labs\Module 10\Lab 10.1\Starter* ,  
which contains a project called *RelaxingConstraints*.

The Starter project already contains two enum types:

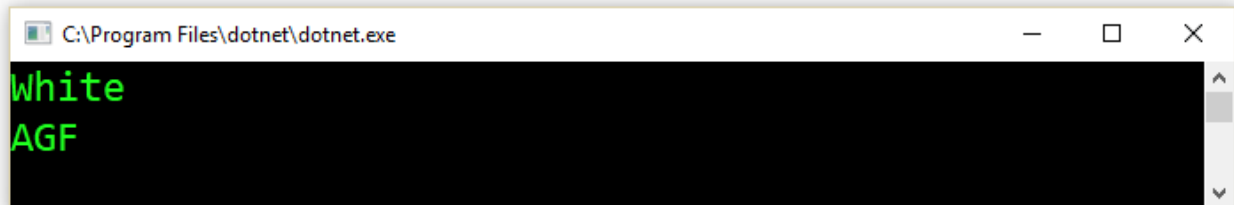
```
enum Color
{
    White,
    Red,
    Green,
    Blue,
    Yellow,
    Gray
}

enum Team
{
    AGF,
    Brøndby,
    FCK,
    OB,
    AaB,
    Horsens,
    Lyngby,
    Randers,
    Helsingør,
    Silkeborg,
    Hobro,
    Sønderjyske,
    FCM,
    FCN
}
```

Your task is to implement a method supplying a random member of a specified concrete enumeration type, such as *Color* or *Team*.

- Write a method *GetRandomMember()* such that
  - the following lines of code (and any other concrete *enum* types) compile:  
`Console.WriteLine(GetRandomMember<Color>());`  
`Console.WriteLine(GetRandomMember<Team>());`
  - these lines however should not compile:  
`Console.WriteLine(GetRandomMember<Enum>());`  
`Console.WriteLine(GetRandomMember<string>());`  
`Console.WriteLine(GetRandomMember<int>());`  
`Console.WriteLine(GetRandomMember<object>());`

A typical output of the program lines within Main() could be:

A screenshot of a Windows command prompt window. The title bar at the top reads 'C:\Program Files\dotnet\dotnet.exe'. The window has standard minimize, maximize, and close buttons. The command prompt area has a black background with green text. The first line of output is 'White' and the second line is 'AGF'. A vertical scrollbar is visible on the right side of the text area.

```
C:\Program Files\dotnet\dotnet.exe  
White  
AGF
```



## Lab 10.4: "Avoiding Copying of Value Types" (☆☆☆)

This exercise is concerned with using the new features for achieving performance for value types C# 7.x.

- Open the starter project in  
*PathToCourseFiles\Labs\Module 10\Lab 10.4\Starter* ,  
which contains a project called *EffectiveValueTypes*.

In the supplied solution, somebody has implemented an Abstract Factory pattern for *Coffee* instances as follows:

```
interface ICoffeeFactory
{
    Coffee CreateCoffee( CoffeeType coffeeType );
}
```

However, these *Coffee* elements are actually value types, so while the factory creates and caches these values, it produces a large number of copying of value types.

- Please take a second to ponder: Why is that?

Your task is to

- modify the factory (and if necessary also in *Program.cs*) to eliminate as much unnecessary copying of values as you can
  - You can use any of the new C# 7.x features that you see fit.

Feel free to eliminate any number of allocations as well. 😊