Dag 4 Øvelser

Øvelse 4.1

Lav en extension metode Lang() til String, der returnerer true, hvis længden af pågældende String er over 5, ellers false;

Test i en main metode.

Tilret, så metoden Lang tager en parameter n (int), som erstatter 5 (altså returner true, hvis længden er over n)

Øvelse 4.2

Lav (i et nyt projekt) en delegate type void Warning().

I en Main-klasse (med en Main-metode) laves en metode void warningToConsole(). Denne skal bare skrive noget (f.eks. ”Advarsel”) til Console. (Metoden skal nok være static, ellers skal man oprette en instans af klassen)

Lav en klasse PowerPlant.

PowerPlant skal have en delegate instans af type Warning og en metode til at sætte denne (f.eks. `public void SetWarning(Warning wa)`.)

PowerPlant skal have en metode HeatUp. HeatUp skal generere et tilfældigt tal mellem 0 og 100. Hvis tallet er over 50 skal Warning-instansen invokes. (Den anvendte Random-instans skal erklæres på klasse-niveau)

Test i Main.
Der skal oprettes en PowerPlant, der skal sættes en Warning-delegate på PowerPlant. Denne skal pege på warningToConsole().
Der skal komme en advarsel ud i konsollen, hvis det genererede tal er over 50.

Tilret, så metoden, der sætter delegate instansen, i stedet tilføjer denne. Dette gøres med +=. (altså _Warning += wa; i stedet for _Warning = wa;)

Lav en metode mere, som passer til Warning-delegate-signaturen. Tilføj denne til PowerPlants Warning-delegate. Se, at begge metoder køres, når instansen invokes.

Øvelse 4.3

Lav et nyt consol project til øvelse 2 og 3, hvor de begge kan være i det samme projekt.

a)

Skriv en rekursiv metode public static int Factorial(int n) der beregner n!, n>=0.

(Ja, eller hvis du hader rekursion så lav en løkke)

Den rekursive definition er givet ved.

Termineringsregel: 0! = 1

Rekurrensregel: n! = n * (n -1)!, n > 0

b)

Brug din factorial metode til at lave en extension metode, så man kan skrive 4.Factorial() og få resultatet 24. (så din extension method skal virke på ints).

Husk at en extension method skal defineres i en ny static class – se slides.

Øvelse 4.4

a)

Skriv en metode public static int Power(int n, int p) der beregner n opløftet i p hvor p >=0. Den kan f.eks. implementeres med en for loop.

b)

Lav en extension metode, så man for eksempel kan skrive 2.Power(4) og få resultatet 16

Øvelse 4.5

Lav et nyt program beregnet til at håndtere spillekort.

Repræsentér et spillekort som en instans af klassen Card, så hver instans indeholder 2 properties Suit (Clubs, Diamonds, Hearts eller Spades) samt Number (Ace, Two, Three, Four, Five…. Ten, Jack, Queen eller King) (Du kan også lave dem på dansk!)

Du kan passende lave to nye enums i din Card klasse til at kunne modellere dette – f.eks. en

https://www.tutorialsteacher.com/csharp/csharp-enum

Lav også en ny ToString metode – f.eks. sådan her:

```
public override string ToString()
        {
            return Suit + " " + Number;
```

```
        }
```

Lav også en ny collection klasse af spillekort, kald den f.eks. CardGame eller noget i den stil. Inde i klassen kan du f.eks. gemme kortene i en List<Card> og så have en AddCard metode, som har signaturen: `public void AddCard(Suit suit, Number number)`

Så du skal f.eks. kunne tilføje elementer sådan her :

```
game.AddCard(Suit.Clubs, Number.Ace); //tilføjer så klør es
```

Udvid CardGame klassen med en filterCardame metode, som kan tage en delegate som parameter. Denne delegate skal kunne afgøre om et vilkårligt kort skal inkluderes i filtreret eller ikke.

Så du kan definere en delegate i Card Klassen sådan her:

```
    delegate bool FilterCardDelegate(Card card);
```

Dette er en generelle opskrift på hvordan en filter funktion så skal være. Dvs. en filter funktion skal tage et kort som input og så returnere true eller false afhængig af om kortet skal inkluderes eller ikke.

I CardGame klassen har du så en filter funktion med følgende signatur – dvs. den har en delegate af typen FilterCardDelegate som parameter:

```
  public List<Card> filterCardGame(FilterCardDelegate filter)
```

I din main klasse kan du så lave et par filtre for at teste om din implementation virker.

Her er f.eks. en delegate, som finder alle kløerne:

```
static public bool FilterByKlør(Card card)
        {
            if (card.Suit == Suit.Clubs)
                return true;
            else
                return false;

        }  //Eller lav den lidt pænere, det skal jo være en one-liner
```

Lav en tilsvarende metode, som har samme type som delegaten – (altså Card->bool), som kun medtager alle billedkort (dvs. knægt, dame og konge). Lav derefter endnu en delegate, som finder alle ”ikke-billedkort”)

Du skal nu implementere filterCardGame funktionen fra før – ved at bruge den delegate du får med som parameter til at lave en ny liste, hvor kun de kort som opfylder delegate funktionen er med – altså hvor de returnere true.

Du skal så kunne lave et kald på følgende måde:

```
     var billedKort = game.filterCardGame(FilterByBilledKort);
```

Hint: Du skal nok have gang i en foreach loop og så anvende den delegate, du får ind som parameter på hvert kort….

Du kan derefter løbe den nye liste igennem, som returneres fra filterCardGame, med en foreach loop og skrive kortene ud i consolen, for at se om din delegate er lavet korrekt.

```
foreach (Card card in billedKort)
            {
                Console.WriteLine(card.ToString());


            }
```

Prøv derefter at lave nogle flere interessante filtre – som selvfølgelig skal opfylde signaturen på din delegate, altså tage et Card som input og returnere true eller false.

Prøv at lave dem som lambda expressions.

Øvelse 4.6 (stof fra dag 3)

Der er forskellige måder at sortere en liste på. En af dem er at implementere interfacet IComparer<T> eller IComparer Interface

https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.icomparer-1?view=netframework-4.7.2 (Links til en ekstern webside.)

eller

https://docs.microsoft.com/en-us/dotnet/api/system.collections.icomparer.compare?view=netframework-4.7.2 (Links til en ekstern webside.)

Lav en Person klasse med Age, Weight og Name og lav et par objekter af denne type og sæt ind i en liste af typen List<Person>

Implementer 2-3 sorteringsklasser, som alle implementerer IComparer interfacet:

```
public class ByAgeSorter : IComparer<Person>
public class ByWeightSorter : IComparer<Person>
public class ByNameSorter : IComparer<Person>
```

Nu kan du så sortere dine personer og se om dine sorteringeren virker. Så du kan f.eks. skrive kode sådan her (hvis din liste af personer hedder people).

```
people.Sort(new SortByWeight());
```

and people are sorted by weight.

Test de 3 sorteringsfunktioner og skriv personerne ud i consolen – du bør nok også override ToString() metoden i Person.

Lav om, så du ikke bruger de 3 klasser men delegates/lambda expressions i stedet. Her bruger du en anden overload af Sort, se dem her:

https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1.sort?view=net-6.0

Øvelse 4.7

Start fra projektet DelegateExercise. Da opgaven kan være lidt svær at forstå er givet en alternativ forklaring i koden i filen Person.cs. Sig til hvis det giver problemer. Opgaven går basalt set ud på at generalisere de forskellige formateringsmetoder ved hjælp af en delegate.

Person.cs indeholder en klasse *"Person"* med to properties *"FirstName"* og *"LastName"*, samt et antal metoder til at udskrive personens navn på forskellige måder.

Hvis du kikker på de fire metoder kan du se, at der er noget kode, der går igen i alle fire formaterings metoder, indrømmet det er ikke meget.

Din opgave er at erstatte de fire metoder med én enkelt metode, så vi undgår duplikeret kode.

1. Start med at lave en delegate i Person klassen (definer den i samme fil som Person klassen, men udenfor selve person klassen – men indenfor Namespace) Formålet med denne delegate er at den skal bruges til at give en metode med som parameter til vores nye Print metode. Således at man kan kalde vores nye metode med forskellige metoder, der formaterer navnet på forskellige måder.

```
public delegate string FormatPersonName(string firstname, string lastname);
```

   Så delegaten tager to inputs, firstname og lastname og returnerer en formateret string – det er jo den generelle måde alle fire formateringsfunktioner virker på.

2. Lav på Person klassen nu en ny metode, der tager den nyoprettede delegate som parameter. Metoden udskriver til konsollen resultatet af et kald til parameteren. Dvs. signaturen på denne metode bliver sådan her:

   ```
   public void PrintPersonName(FormatPersonName formatter) og den skal så bruge
   formatteren på firstname,lastname og udskriver resultatet til consol.
   ```

3. Gå nu tilbage PersonUserClass og tilføj fire metoder, der hver især formaterer en persons navn til et af de kendte formater. De skal så overholde delegate signaturen og altså alle have firstname, lastname som input og give den formaterede string retur.
4. Endelig skal du nu erstatte kaldene til PrintFullNameLastNameFirst(), PrintFullNameAllCaps(), PrintFullNameLowerCase() og PrintShortName() med kald til din nye metode – PrintPersonName med de relevante metoder med som parameter. Altså f.eks:

```
 people.ForEach(p => p.PrintPersonName((F, L) => PrintFullNameLastNameFirst(F,
L)));
/*Bemærk: Dette er et lambda udtryk, hvor vi bruger en delegate, som jo har et
input (F står for førstname og L for Lastname – man kan bruge et hvilket som
helst navn her)*/
```

5. Ekstra opgave: Prøv nu også at slippe af med de gentagne kald til ForEach for hver formatering af personnavnene. Dette kan du f.eks. gøre ved at lave en PrintPeople metode som tager din delegate som parameter og bruger den under gennemløb af alle personer. Dvs. signaturen på denne metode bliver :

   ```
   public void PrintPeople(FormatPersonName formatter)
   ```

   (der er stadigvæk en Foreach inde i PrintPeople, men nu har vi kun 1 foreach og ikke en foreach hvergang vi bruger delegaten).