LINQ & Operator Overload

Peter Levinsky IT Roskilde

11.04.2023

LINQ – A query language – like SQL

- Preconditions
  - For every Data structure that support the interface:

```
public interface IEnumerable<out T>
{
   IEnumerator<T> GetEnumerator();
}
```

  - The **IEnumerator** interface consists of three methods:

```
bool MoveNext();
void Reset();
T Current { get; }
```

LINQ – A query language – like SQL

foreach (var t in anyDataStructure){ **//** initial **Reset**

the variable t // point at **Current**

} // **MoveNext** -> true if a next element exists otherwise false
conditional for continuing looping

LINQ – A query language – like SQL

- What can you do with LINQ
- Make queries into the datastructure
- You have two forms
  - Query Language
  - Fluent syntax (Method calls)

LINQ – A query language – like SQL

- Example Query Language

```
var titlesAndYears =

from m in movies
where (m.Year < 1996 && m.Year > 1980)
orderby m.Year
select new {m.Title, m.Year};
```

LINQ – A query language – like SQL

- Example Fluent syntax

```
var titlesAndYears =

movies.Where(m => m.Year < 1996 && m.Year > 1980).
Orderby(m => m.Year).Select new {m.Title, m.Year};
```

LINQ – A query language – like SQL

- **Strength of LINQ**
  - Can be used on any datastructure
    if only the **IEnumerable** interface is implemented
  - I.e on a datastructure of your own,
    if the **IEnumerable** interface is implemented

    You can use Select, Where, Orderby, ….

LINQ – A more advanced use

- LINQ as **Data Transformation**
  - Direct transformation
  - Aggregate functionality
- **Set-oriented** operation with LINQ
- **Parallel** LINQ (PLINQ)

LINQ – Data Transformation

- I have
- Movie: Title, Year, DurationInMins, StudioName

- I wish
- MovieInfo: Title, YearSince1900, TimeHours

```
List<MovieInfo> miList =
   movies.Select(m => new MovieInfo(
                              m.Title,
                              m.Year - 1900,
                              m.DurationInMins / 60.0)
                         )
                         .ToList();
```

LINQ – Data Transformation

```
public static List<V> TransformItems<T, V>(
                          List<T> items, Func<T,V> transformer)
{
    List<V> transformedItems = new List<V>();
    foreach (T item in items)
    {
        V transformedItem = transformer(item);
        transformedItems.Add(transformedItem);
    }
    return transformedItems;
}
```

Convert from T to V

Call:

```
List<MovieInfo> miList = Transformer.TransformItems<Movie, MovieInfo>(
                         movies, m => new MovieInfo(
                         m.Title, m.Year - 1900, m.DurationInMins / 60.0)
                         );
```

LINQ – Actions at the object level

I wish:

```
movies.Select(m => Console.WriteLine(m));  // not working !
```

On List:

```
movies.ForEach(m => Console.WriteLine(m));
movies.ForEach(Console.WriteLine);
```

LINQ convert to List:

```
movies
   .Select(m => $"{m.Title}, made in {m.Year}")
   .ToList()  // this do the trick ☺
   .ForEach(Console.WriteLine);
```

LINQ – Aggregation (different functions)

Examples:

```
List<int> numbers = new List<int>{ 21, 8, 14, 45, 30, 9, 22 };

int sum = numbers.Sum();
int max = numbers.Max();
double avg = numbers.Avarage();
```

LINQ – Aggregation (different functions)

General description:

```
public static T Aggregate(
    IEnumerable<V> collection,     // the collection
    Func<T> initialValueFunc,      // the starting value
    Func<T,V,T> updateValueFunc)   // what to do in each iteration
    {
        T value = initialValueFunc();
        foreach (V item in collection)
        {
            value = updateValueFunc(value, item);
        }
        return value;
    }
```

LINQ – Aggregation (different functions)

```
List<int> numbers = new List<int> { 21, 8, 14, 45 };
int product = AggregateCalculator<int,int>.Aggregate(
    numbers,                    // collection
    () => 1,                    // initial value
    (val, item) => val * item); // each iteration
Console.WriteLine($"Product is {product}");
```

```
List<string> words = new List<string>{ "This ", "is ", "Sparta!"};
string concatStr = AggregateCalculator<string, string>.Aggregate(
    words,                  // collection
    () => "" ,              // initial value
    (val, item) => val + item); // each iteration
Console.WriteLine(concatStr);
```

LINQ – Set-oriented operation with LINQ

Method	Called on	Argument	Description
Except	A	B	Returns items which are in A but not in B.
Intersect	A	B	Returns items which are in both A and B.
Union	A	B	Returns items which are in either A or B.
Distinct	A	(none)	Removes duplicate items from A
Concat	A	B	Concatenates B to A.

If you like to support logical equality => implement the Equal-methods in the class!

LINQ – Parallel LINQ (PLINQ)

Seen Threads / Tasks for handling CPU bounds

In LINQ we have a build in functionality to parallels calculations called PLINQ

LINQ – Parallel LINQ (PLINQ)

Example – finding prime numbers – WITHOUT parallel

Method to investigate if number is a prime

```
Stopwatch watch = new Stopwatch();
watch.Restart();

IEnumerable<int> primes = Enumerable.Range(2, 1000000).Where(IsPrime);

int primesCount = primes.Count();
watch.Stop();
Console.WriteLine($"Primes up to 1,000,000: {primesCount}");
Console.WriteLine($"Time spent: {watch.ElapsedMilliseconds} ms");
```

LINQ – Parallel LINQ (PLINQ)

Same example – finding prime numbers – WITH parallel

```
Stopwatch watch = new Stopwatch();
watch.Restart();

IEnumerable<int> primes = Enumerable.Range(2, 1000000)
    .AsParallel()       // THIS do the trick
    .AsOrdered()        // Keep the same order
    .Where(IsPrime);

int primesCount = primes.Count();
watch.Stop();
Console.WriteLine($"Primes up to 1,000,000: {primesCount}");
Console.WriteLine($"Time spent: {watch.ElapsedMilliseconds} ms");
```

That’s it

- **Training: PRO.3.4, PRO.3.7**

- **And the Mandatory Assignment**

Operator Overload

Peter Levinsky IT Roskilde

11.04.2023

Overload of operators

- Four categories:
  - +,–, *, / … Operator
  - Indexer (i.e. [])
  - <= and >= Operator
  - == and != Operator

Overload: +,-,*,/,++ and -- Operator

- Example:

```
public static Time operator +(Time tA, Time tB)
{
…     // this add two time objects and return a new
}
```

Call:

```
Time timeC = timeA + timeB;
```

The general definition

```
public static TReturn operator operatorGoesHere(TA opA, TB opB)
{
      …
}
```

Overload: +,-,*,/,++ and -- Operator

- Example variation:

```
public static Time operator *(int val, Time t)
{
   int totalMin = (t.Hours * 60 + t.Minutes) * val;
   return new Time(totalMin / 60, totalMin % 60);
}
```

Call:

```
Time timeMult3A = 3 * timeA; // OK
```

but

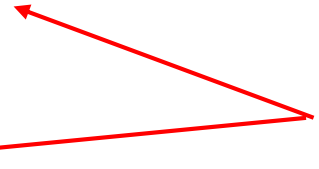
```
Time timeMultA3 = timeA * 3; // Error
```

```
public static Time operator *(Time t, int val) // reverse parameter order
{
      return val * t;
}
```

Overload - Indexer (i.e. [])

```
public class NPC
{
   private Dictionary<NPCStateTypes, int> State;

   public NPC()
   {
      State = new Dictionary<NPCStateTypes, int>();
      SetDefaultValues();
   }

   public void SetStateValue(NPCStateTypes stateType, int
   value)
   {
      State[stateType] = value;
   }

   public int GetStateValue(NPCStateTypes stateType)
   {
      return State[stateType];
   }

   private void SetDefaultValues() { … }
}
```

```
public enum NPCStateTypes
{
   hungry, rested, aggressive, fear, gullible
}
```



Wish to use [] Instead of GetStateValue, SetStateValue

Overload - Indexer (i.e. [])

- The implementation in the class:

meaning this class

define the brackets + what's inside

```
public int this[NPCStateTypes stateType]
{
    get { return State[stateType]; }
    set { State[stateType] = value; }
}
```

Overload == and != Operator

If overload == also overload != ☺

```
public static bool operator ==(Time tA, Time tB)
{
   return (tA.Length == tB.Length); // tA.Equals(tB)
}

public static bool operator !=(Time tA, Time tB)
{
   return !(tA == tB);
}
```

BUT this implementations do not work – risk for NullReferenceException! DO THIS!

1. First override Equals appropriately (i.e. also checking for null)
2. As a consequence of overriding Equals, we must also override the method GetHashCode from Object.
3. Now override == and != by using Equals

Overload: <= and >= Operator (< and >)

If overload >= also overload <= ☺

```
public static bool operator >=(Time tA, Time tB)
{  // ToDo check for null
   return (tA.Length >= tB.Length);
}

public static bool operator <=(Time tA, Time tB)
{
   return (tA.Length <= tB.Length);
}
```

That’s it

- **Training: OOP.4.1, OOP.4.2**

- **And the Mandatory Assignment**