Classes and LINQ

oxygen

10. oktober 2017

Yaaaaay, we got a TA

Christian will start from next week and your TA in the hours 16.00-18.00

I’ve given up on getting away from you and will therefore teach the remainder of the semester.

Agenda

Abstract classes

Interfaces

Patterns (cloning in C#)

LINQ

Abstract classes

Abstract

At the conceptual level, we start by a study of class specialization and class extension. Following that the programming language concept of inheritance is introduced.

In the last lecture we study classes in C#

Method combination

We sometimes talk about method combination when two or more methods of the same name ***Op*** cooperate in solving a given problem

$$\begin{tikzcd}
\text{Class A} \arrow[r] & \text{Op(S x)} \\
\text{Class B} \arrow[u] \arrow[r] & \text{Op(S x)}
\end{tikzcd}$$

Imperative control

Programmatic (imperative) control of the combination of ***Op*** methods

Superclass controlled:

- The ***Op*** method in class ***A*** controls the activation of Op in class B

Subclass controlled:

- The ***Op*** method in class ***B*** controls the activation of Op in class A

This is called **Imperative method combination**

Declarative control

An overall (declarative) pattern controls the mutual cooperation among ***Op*** methods

- ***A***.*Op*(…) does not call ***B***.*Op*(…)   -   ***B***.*Op*(…) does not call ***A***.*Op*(…)
- A separate abstraction controls how ***Op*** methods in different classes are combined

Declarative method combination

C# is imperative

C# supports subclass controlled, imperative method combination

Use the notation ***base***.*Op*(...)

Parameter variance

How do the parameters of *Op* in class ***A*** and ***B*** vary in relation the variation of ***A*** and ***B***

$$\begin{tikzcd}
A \arrow[r, dash, dotted] & Op(S\ x) & S \\
B \arrow[u] \arrow[r, dash, dotted] & Op(T\ x) & T \arrow[u]
\end{tikzcd}$$

Not a relevant subject in C#

Covariance and Contravariance

$$\begin{tikzcd}
A \arrow[r, dotted, no head] & \mathrm{Op}(S\ x) & S \\
B \arrow[u] \arrow[r, dotted, no head] & \mathrm{Op}(T\ x) & T \arrow[u]
\end{tikzcd}$$

Covariance: The parameters *S* and *T* vary the same way as ***A*** and ***B***

$$\begin{tikzcd}
A \arrow[r, dotted, no head] & \mathrm{Op}(S\ x) & T \\
B \arrow[u] \arrow[r, dotted, no head] & \mathrm{Op}(T\ x) & S \arrow[u]
\end{tikzcd}$$

Contravariance: The parameters *S* and *T* vary the opposite way as ***A*** and ***B***

Abstract classes

Abstract classes are used for concepts that we cannot or will not implement in full details

Abstract classes (cont.)

The concept abstract class:

- An abstract class is a class with one or more abstract operations

The concept abstract operation:

- An abstract operation is specially marked operation with a name and with formal parameters, but without a body

An abstract class

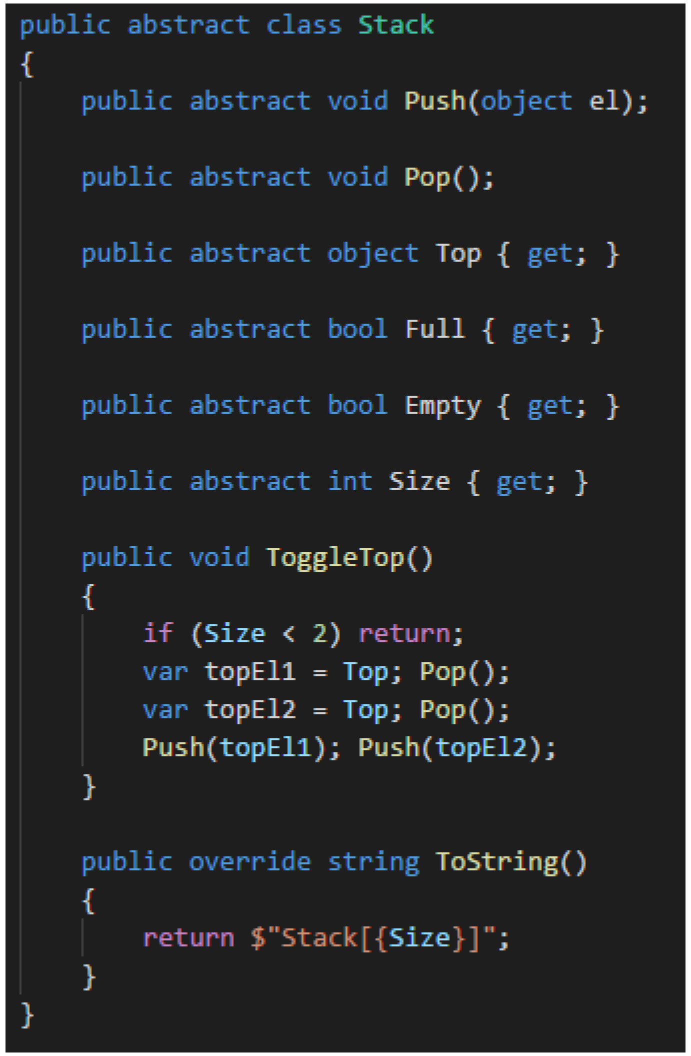
- may announce a number of abstract operations, which must be supplied in subclasses
- cannot be instantiated
- is intended to be completed/finished in a subclass

Example – Abstract stack

An abstract Stack without data representation, but with some implemented operations

Detailed rules - some are surprising

- Abstract classes
  - can be derived from a non-abstract class
  - do not need not to have abstract members
  - can have constructors
- Abstract methods
  - are implicitly virtual



```csharp
public abstract class Stack
{
    public abstract void Push(object el);

    public abstract void Pop();

    public abstract object Top { get; }

    public abstract bool Full { get; }

    public abstract bool Empty { get; }

    public abstract int Size { get; }

    public void ToggleTop()
    {
        if (Size < 2) return;
        var topEl1 = Top; Pop();
        var topEl2 = Top; Pop();
        Push(topEl1); Push(topEl2);
    }

    public override string ToString()
    {
        return $"Stack[{Size}]";
    }
}
```

Exercise – Implement the stack

Make a non-abstract specialization of Stack, and decide on a reasonable data representation of the stack.

In this exercise it is OK to ignore exception/error handling.

- You can, for instance, assume that the capacity of the stack is unlimited;
- That popping an empty stack does nothing;
- And that the top of an empty stack returns the string "Not Possible".

In a later lecture we will revisit this exercise in order to introduce exception handling. Exception handling is relevant when we work on full or empty stacks.

Exercise – Revisit Course

In the exercises for last time, you were asked to create ***BooleanCourse*** and ***GradedCourse***

Revise and reorganize your solution (or the model solution) such that ***BooleanCourse*** and ***GradedCourse*** have a common abstract superclass called ***Course***

Be sure to implement the method Passed as an abstract method in class ***Course***

Demonstrate that both boolean courses and graded courses can be referred to by variables of static type ***Course***

Abstract properties

Properties and indexers may be abstract in the same way as methods

oxygen

Sealed classes

Sealed classes

A sealed class ***C*** prevents the use of ***C*** as base class of other classes

Sealed class

- Cannot be inherited by other classes
- Seals all virtual methods in the class

Sealed method

- Cannot be redefined and overridden in a subclass
- The modifier sealed must be used together with override
  - A sealed, overridden method prevents additional overriding

oxygen

Interfaces

Interfaces

An interface corresponds to a fully abstract class.

No matters of substance is found in the interface, just declarations of intent

The concept interface:

- An interface describes signatures of operations, but it does not implement any of them

Interfaces (cont.)

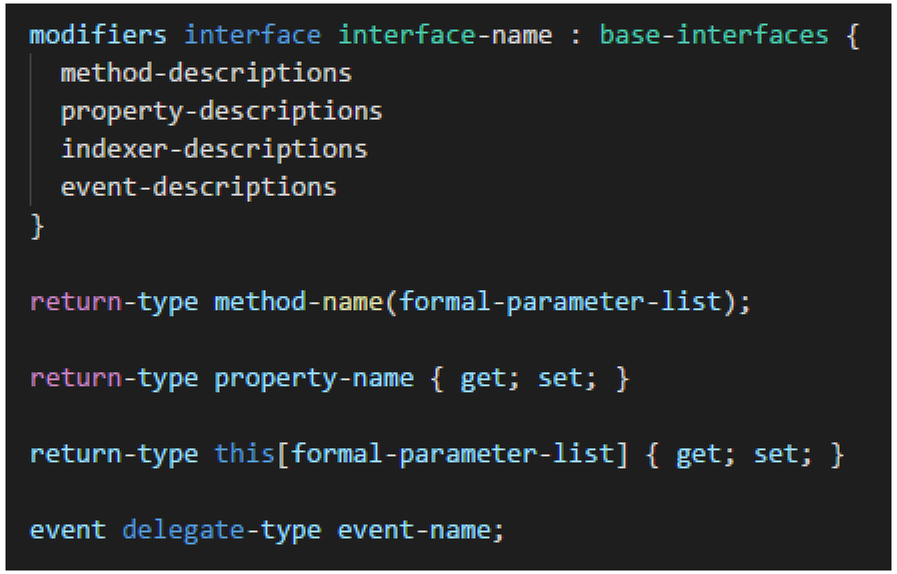
Classes and structs can implement one or more interfaces

An interface can be used as a type, just like classes

- Variables and parameters can be declared of interface types

Interfaces can be organized in multiple inheritance hierarchies

Interfaces in C# - Syntax



```
modifiers interface interface-name : base-interfaces {
  method-descriptions
  property-descriptions
  indexer-descriptions
  event-descriptions
}

return-type method-name(formal-parameter-list);

return-type property-name { get; set; }

return-type this[formal-parameter-list] { get; set; }

event delegate-type event-name;
```

Exercise - ITaxable

For the purpose of this exercise you are given a couple of very simple classes called Bus and House (See Taxable.cs). Class Bus specializes the class Vehicle. Class House specializes the class FixedProperty.

First in this exercise, program an interface ITaxable with a parameterless operation TaxValue. The operation should return a decimal number.

Next, program variations of class House and class Bus which implement the interface ITaxable. Feel free to invent the concrete taxation of houses and busses. Notice that both class House and Bus have a superclass, namely FixedProperty and Vehicle, respectively. Therefore it is essential that taxation is introduced via an interface.

Demonstrate that taxable house objects and taxable bus objects can be used together as objects of type ITaxable.

Interfaces in C#

Both classes, structs and interfaces can implement one or more interfaces

Interfaces can contain signatures of methods, properties, indexers, and events

Two or more unrelated classes can be used together if they implement the same interface

Interfaces from C# library

IComparable

- An interface that prescribes a CompareTo method
- Used to support general sorting and searching methods

IEnumerable

- An interface that prescribes a method for accessing an enumerator

IEnumerator

- An interface that prescribes methods for traversal of data collections
- Supports the underlying machinery of the foreach control structure

IDisposable

- An interface that prescribes a Dispose method
- Used for deletion of resources that cannot be deleted by the garbage collector
- Supports the C# using control structure

ICloneable

- An interface that prescribes a Clone method

IFormattable

- An interface that prescribes an extended ToString method

Exercise – Comparable Die

In this exercise we will arrange that two dice can be compared to each other.

The result of die1.CompareTo(die2) is an integer.

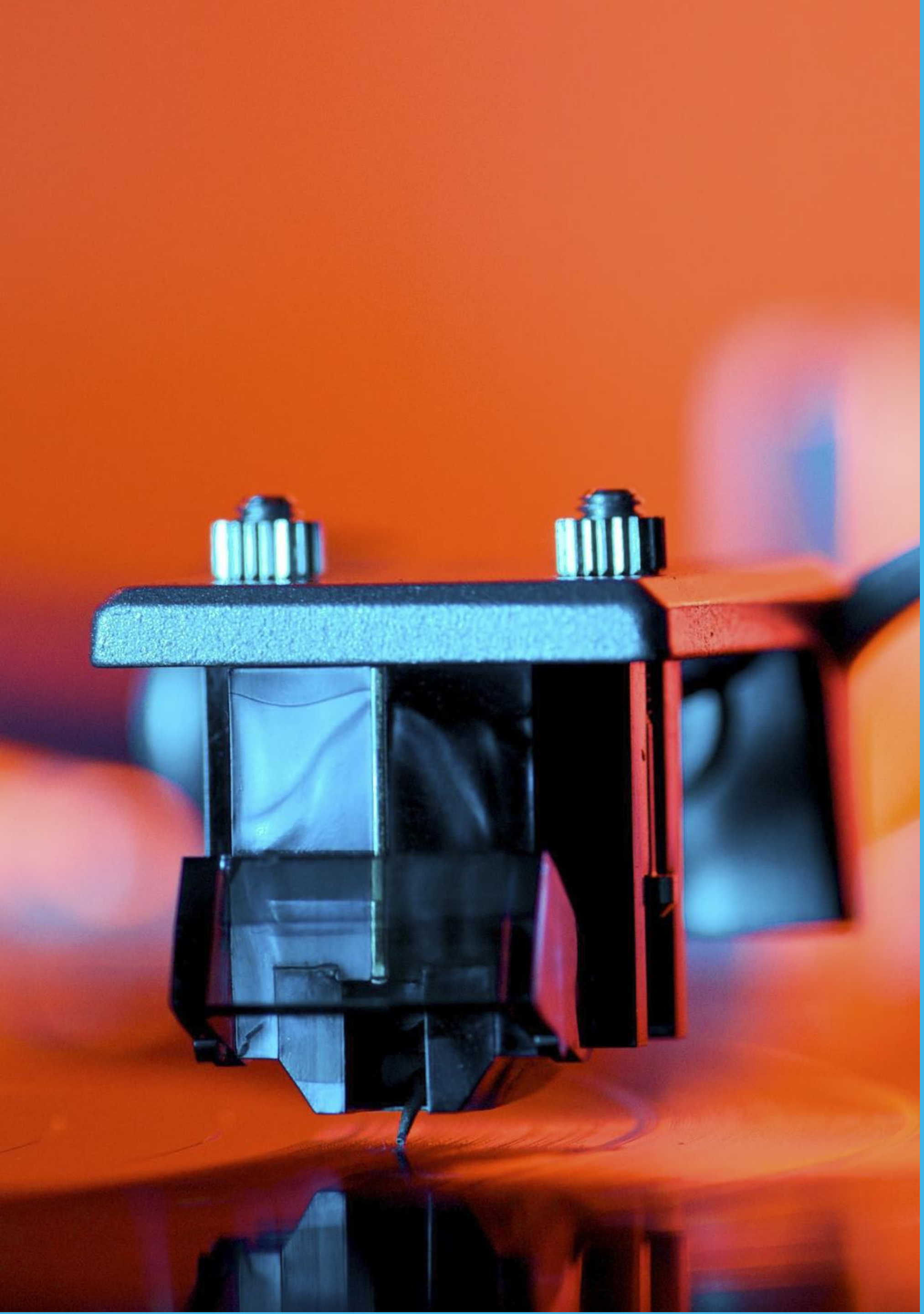
If the integer is negative, die1 is considered less than die2;
If zero, die1 is considered equal to die2;
And if positive, die1 is considered greater than die2.

When two dice can be compared to each other, it is possible sort an array of dice with the standard Sort method in C#.

Program a version of class Die which implements the interface System.IComparable.

Consult the documentation of the (overloaded) static method System.Array.Sort and locate the Sort method which relies on IComparable elements.

Make an array of dice and sort them by use of the Sort method.



oxygen

Patterns in C#

Cloning in C#

Object cloning

- **Shallow cloning**:
  - Instance variables of value type: Copied bit-by-bit
  - Instance variables of reference types:
  - The reference is copied
  - The object pointed at by the reference is not copied
- **Deep cloning**:
  - Like shallow cloning
  - But objects referred by references are copied recursively

Cloning in C# (cont.)

Internally, C# supports shallow cloning of every object.

Externally, it must be decided on a class-by-class basis if cloning is supported.

Shallow cloning is facilitated by the protected method *MemberwiseClone* in ***System.Object***

A class ***C*** that allows clients to clone instances of ***C,*** should implement the interface ICloneable

LINQ

Origin and Rationale

LINQ → Language Integrated Query

The main ideas behind LINQ come from functional programming languages and database query languages (SQL)

LINQ and functional programming

- Builds on old ideas from Lisp: Map, filter, reduce.
- Programming without (side)effects
- Lazy evaluation

LINQ and SQL

- LINQ is a query language
- LINQ fragments are translated to SQL (via so-called Expression Trees)

LINQ - as used from C# - can both be used on all kinds of .NET collections, and instead of SQL

Concepts

Sequence

- A collection - of type IEnumerable<T> - that can be traversed by use of an iterator
- IQueryable<T> is a sub-interface of IEnumerable<T>

Data source

- The sequence considered the original input to a query

Query

- An expression which (lazily) extracts information from a sequence

Query Operator

- A pure function, defined on Sequence types, and used in a query

LINQ Provider

- An implementation of a number of query operators on a given kind of data source
- Examples: LINQ to Objects and LINQ to SQL

Map, filter and reduce

Classical wisdom from functional programming on lists:

A large set of problems can be solved by mapping, filtering, and reduction.

Map, filter and reduce (cont.)

Mapping

- Apply a function to each element of a collection and return the resulting collection
- LINQ: Select

Filtering

- Apply a predicate on each element of a collection, and return those elements that satisfy the predicate
- LINQ: Where

Reduction

- Apply a binary function on neighbour pairs of collection elements (either from left to right, or right to left), and "boil" down the collection to a single value.
- LINQ: Aggregate
  - Specialized aggregations: Count, Min, Max, Sum, Average.

Exercise: Try LINQ

Create console app

Copy contents of Person.cs into a console app

Create a collection of test data, i.e.

new List<Person>{

    new Person(“Nicolai", “Oksen”, Gender.Male, 27), …

}

1. Find all females (using where)
2. Select the age of all females (using select)
3. Find the average age (using average)

An overview of query operators

Operators that return sequences *s*

- s.Select(Func)
- s.Where(Func)
- s.Distinct()
- s.Take(int)
- s.Intersect(Sequence)
- s.Union(Sequence)
- s.Concat(Sequence)
- s.OrderBy(Func)

Operators that return an element

- s.Aggregate(Func)
- s.Aggregate(seed, Func)
- S.First()
- S.Last()
- S.Max()
- S.Min()
- S.ElementAt(int)
- s.Single()

An overview of query operators (cont.)

Predicates (return boolean)

- s.Any()
  - Returns true if there is at least one element
- s.Contains(element)
  - Returns true if a specific element meets the constraint

Operators that return an other type of object

- s.Count()

LINQ vs List<T> methods

Many methods in List<T> mutates the list

LINQ query operations are pure functions - they return a new modified copy of the list

LINQ vs List<T> methods

Traversal of collections

- List<T>: list.ForEach(Action);
- LINQ: newList = list.Select(Func);

Removal form collections

- List<T>: list.RemoveAll(Predicate);
- LINQ: newList =
  list.Where(negated Predicate);

Appending collections

- List<T>: list.AddRange(otherList);
- LINQ: newList = list.Concat(otherList);

Reversing collections

- List<T>: list.Reverse();
- LINQ: newList = list.Reverse();

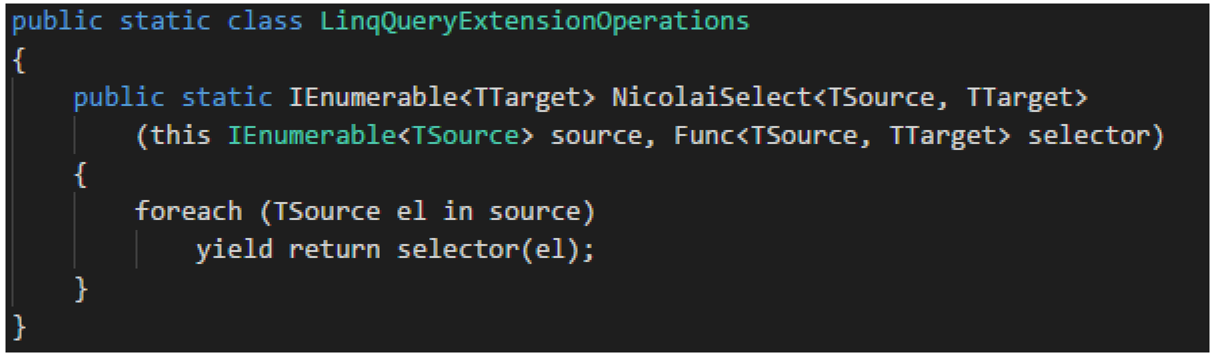
Sorting collections

- List<T>: list.Sort(comparerMethod);
- LINQ: sortedList =
  list.OrderBy(KeySelector);

How LINQ Query Operation works

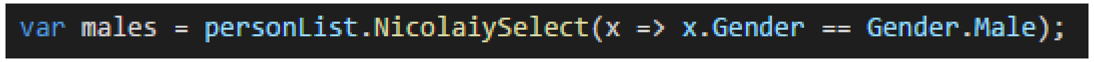
LINQ Query Operations are extension methods in the generic interface IEnumerable<T>

Reproduce Select → NicolaiSelect



```
public static class LinqQueryExtensionOperations
{
    public static IEnumerable<TTarget> NicolaiSelect<TSource, TTarget>
        (this IEnumerable<TSource> source, Func<TSource, TTarget> selector)
    {
        foreach (TSource el in source)
            yield return selector(el);
    }
}
```

It is possible to program the fundamental query operations in a simple and straightforward way



```
var males = personList.NicolaiySelect(x => x.Gender == Gender.Male);
```

How LINQ Query Operation works (cont.)

An object of type IEnumerable<T> is not evaluated and expanded before it is traversed, typically in a foreach statement.

LINQ query operators return a linear list of iterators that decorates the data source

It uses the decorator design pattern → go study it yourselves

The technical basis of LINQ

Most elements of C# 3.0 are invented as the technical basis of LINQ

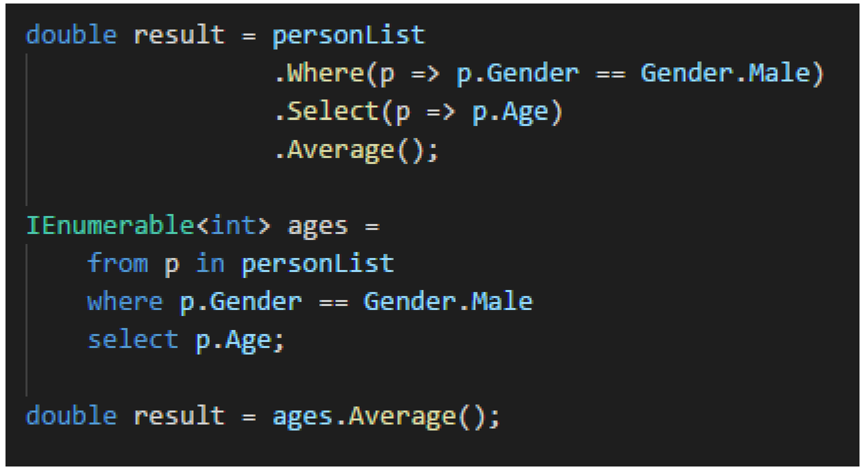
Crucial prerequisites for LINQ:

- Extension methods
- Lambda expressions
- Anonymous types
- Implicitly typed local variables
- Object initializers

Query syntax vs. method syntax

Query Syntax is syntactic sugar on top of the Method syntax,

which we have introduced above

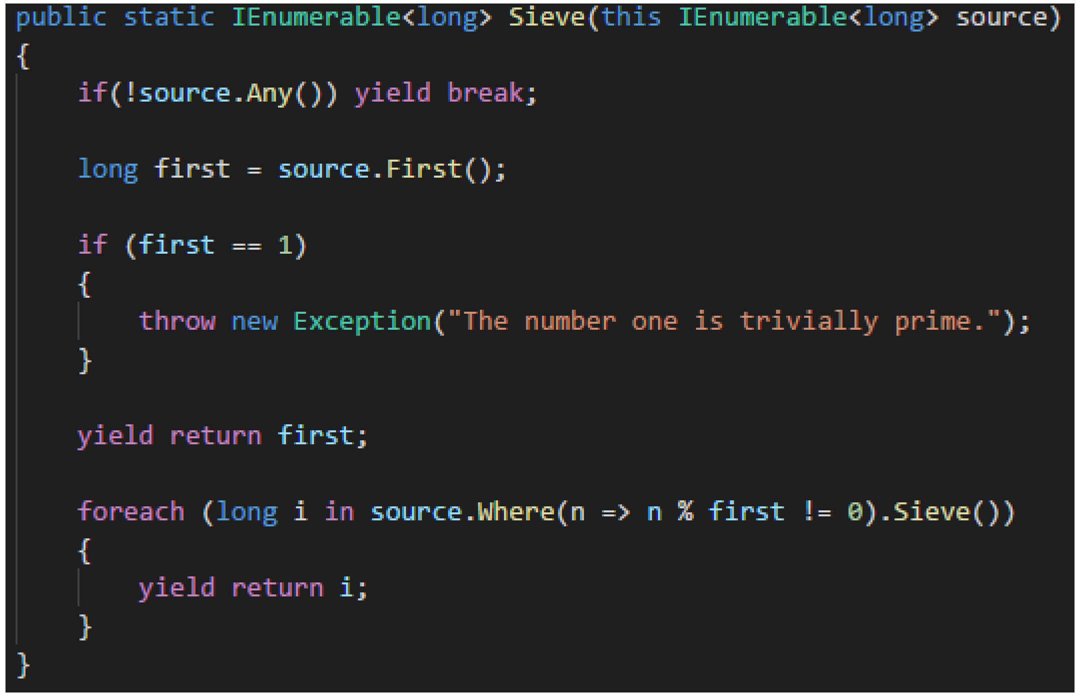

```
double result = personList
                .Where(p => p.Gender == Gender.Male)
                .Select(p => p.Age)
                .Average();

IEnumerable<int> ages =
    from p in personList
    where p.Gender == Gender.Male
    select p.Age;

double result = ages.Average();
```

Exercise: Sieve of Eratosthenes

An simple and elegant way of finding prime numbers

1. Implement an extension method to find prime numbers
2. Test your extension method on a collection of longs

My solution: Sieve of Eratosthenes



```
public static IEnumerable<long> Sieve(this IEnumerable<long> source)
{
    if(!source.Any()) yield break;

    long first = source.First();

    if (first == 1)
    {
        throw new Exception("The number one is trivially prime.");
    }

    yield return first;

    foreach (long i in source.Where(n => n % first != 0).Sieve())
    {
        yield return i;
    }
}
```

Questions?