LINQ

Typesafe Queries

This document covers

Introduction

## Prerequisites

Before we can understand LINQ queries we need to introduce some prerequisites.

* Enumerators
* Enumerables
* Foreach statements
* Iterators

#### Enumerators

An enumerator is a read-only forward cursor over a collection of elements. We define enumerators by implementing the IEnumerator<T> interface.

Listing 1 Simple Enumerator

public class SimpleEnumerator : IEnumerator<int>

{

public int Current => i;

object IEnumerator.Current => i;

public void Dispose() {}

public bool MoveNext() => i++ < 4;

public void Reset() => i = -1;

private int i = -1;

}

We can use an our enumerator as follows

var e = new SimpleEnumerator();

while (e.MoveNext())

WriteLine(e.Current);

#### Enumerables

Enumerables produce enumerators.

Listing 2Enumerable

public class SimpleEnumerable : IEnumerable<int>

{

public IEnumerator<int> GetEnumerator()

{

return new SimpleEnumerator();

}

IEnumerator IEnumerable.GetEnumerator()

{

return GetEnumerator();

}

}

Enumerators are consumed by foreach statements

var e = new SimpleEnumerable();

foreach (var element in e)

WriteLine(e);

#### Foreach statements

Foreach statements consume enumerables. If the compiler sees a foreach statement like this

Listing 3Foreach Statements

var sequence = new List<int>( new [] {1,2,3});

foreach (var element in sequence)

{

WriteLine(element);

}

It generates something along the lines of this

using (IEnumerator<int> en = sequence.GetEnumerator())

{

while (en.MoveNext())

WriteLine(en.Current);

}

#### Iterators

Iterators provide an elegant means of creating enumerators and enumerables. The following code uses iterators to produce an enumerable whose enuerators walk over the first n items in the fibonacci sequence from 0 to n-1

Listing 4 Iterators

IEnumerator<int> GetFibonacci(int numEntries)

{

for (int i = 0, current = 0, next = 1, nextnext = 1;

i < numEntries; i++)

{

yield return current;

nextnext = current + next;

current = next;

next = nextnext;

}

}

## Basics

LINQ enables one to write type-safe queries on enumerable collections. The basic concepts are

* Sequence Any collection that implements IEnumerable<T>
* Element A single constituent of the collection
* Query operator A method that transforms one sequence into another
* Query A combination of query operators that performs a transform

The following piece of code shows all four together

Listing 5 Linq Basics

// 1. A sequence is any collection implementing IEnumerable<T>

IEnumerable<int> sequence = new int[] {0,1,2,3,4};

// 2. An element is a single constituent of the sequence

foreach (var element in sequence)

WriteLine(element);

// 4. Queries combine query operators

IEnumerable<int> output = sequence

.Where(s => s%2 ==0)

.Select(s => s\*s);

…

// 3. Query operators transform sequences

public static class QueryOperators

{

public static IEnumerable<T> Where<T>(this IEnumerable<T> input,

Func<T,bool> predicate)

{

foreach (var element in input)

if ( predicate(element))

yield return element;

}

public static IEnumerable<TOut> Select<TOut,TIn>(this IEnumerable<TIn> input, Func<TIn,TOut> trans)

{

foreach (var element in input)

yield return trans(element);

}

}

Query operators are implemented as extension methods that take an enumerable argument representing an input sequence and a delegate that applies some transformation to create an output sequence. As such query operators are easily composed into queries. Most query operators are not executed when they are constructed. Instead they are executed when they are enumerated. Delayed or lazy execution provides the following benefits.

* Decouples construction from execution
* Allows one to construct a query in multiple steps
* One can re-evaluate a query by enumerating it again

Exceptions to lazy execution

The following operators are exceptions which cause immediate execution

Single element or scalar values such as *First* or *Count*, *ToArray, ToList, ToDictionary, ToLookup*

## How Linq works

We consider three aspects of the followng query

1. Static structure of the decorator after query instantiation
2. Data Flow during query execution
3. Control flow during execution (Sequence diagram)

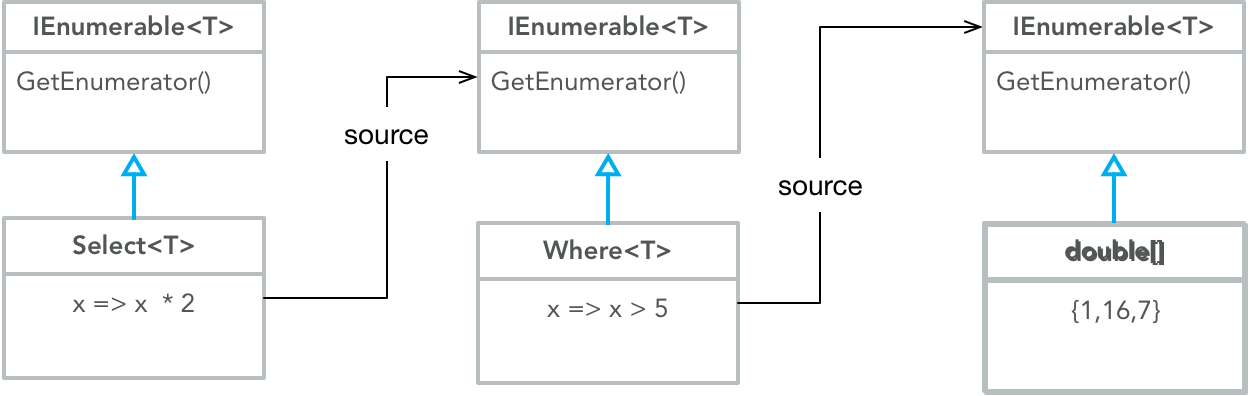
Consider the following simple linq query

var input = new [] {1,16,7};

var output = input.Where(i => i > 5)

.Select(i => i \* 2);

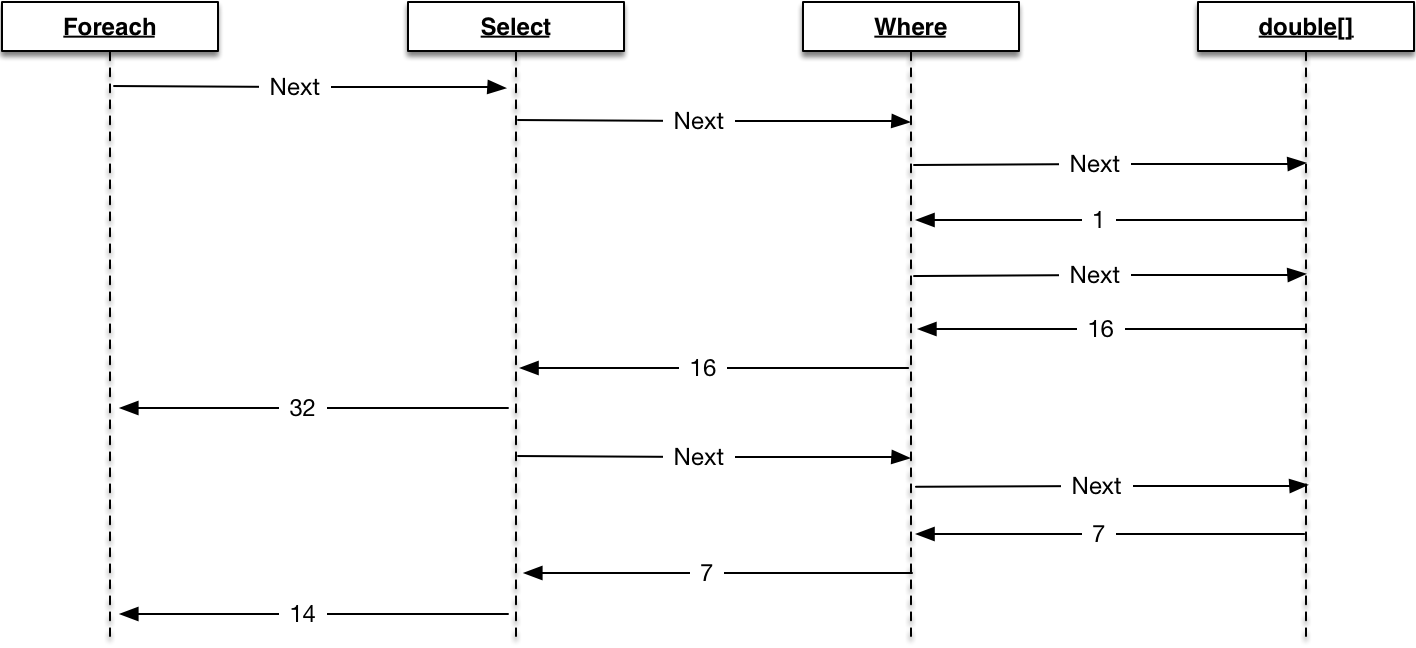
Figure 1 Static Structure On Creation



Listing 6 Data Flow On Execution



Figure 2 Control Flow On Execution



## Operator List

### Where

Listing 7 Where Implementation

public static IEnumerable<T> Where<T>(this IEnumerable<T> source,

Func<T,bool> predicate)

{

foreach (var element in source)

if (predicate(element))

yield return element;

}

public static IEnumerable<T> Where<T>(this IEnumerable<T> source,

Func<T,int, bool> predicate)

{

int i = 0;

foreach (var element in source)

if (predicate(element,i++))

yield return element;

}

Listing 8 Where examples

IEnumerable<string> sanat = new[] { "Äiti", "Suomi", "Ranta", "Isi", "Sisu", "Edessä" };

// Example one basic filtration {"Äiti", "Suomi","Isi","Sisu""}

IEnumerable<string> os1 = sanat.Where(e => e.Contains("i"));

WriteLine(os1); //

// Example two Index Filtration {"Äiti", "Ranta","Sisu""}

IEnumerable<string> os2 = sanat.Where((e, i) => i % 2 == 0);

### ToDictionary

### ToArray

### ToLookup

### Take, Skip, TakeWhile, SkipWhile, Distinct

Listing 9Take,TakeWhile,SkipWhile,Distinct examples

IEnumerable<string> sanat = new[] {"Äiti", "Suomi","Ranta","Isi","Sisu","Edessä"};

WriteLine(sanat.Take(2)); // {"Äiti", "Suomi"}

WriteLine(sanat.Skip(2)); // {"Ranta","Isi","Sisu","Edessä"}

WriteLine(sanat.Take(2)); // {"Äiti", "Suomi"}

WriteLine(sanat.Skip(2).Take(2)); // {"Ranta","Isi"}

WriteLine(sanat.Skip(4).Take(2)); // {"Sisu","Edessä"}

WriteLine(sanat.TakeWhile(e => e.Contains("i"))); // {"Äiti", "Suomi"}

WriteLine(sanat.SkipWhile(e => e.Contains("i"))); // {"Ranta","Isi","Sisu","Edessä"}

WriteLine(new[] { "Moi", "MOI", "MoI"}.Distinct(StringComparer.InvariantCultureIgnoreCase)); // {Moi}

### Select

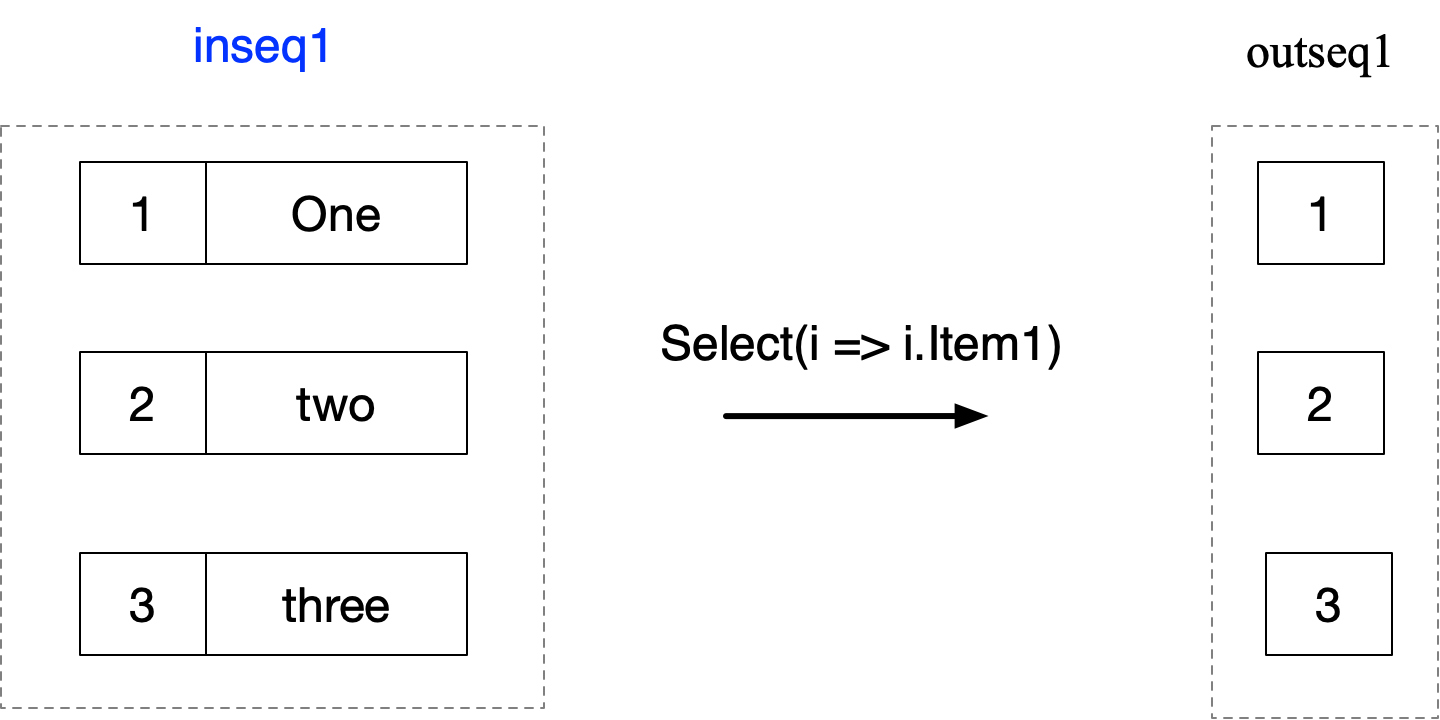
#### The basic case

Select is a projection operator. It maps on projects input sequences onto output sequences. The output sequences have the same number of elements as the input sequences.

Listing 10 Select – Basic Projection

var inseq1 = new [] {(1, "one"),(2, "two"),(3, "three")};

var outseq11 = inseq1.Select(i => i.Item1);



#### Correlated Subqueries

We can insert subqueries inside select clauses. If the subquery references the elements of the outer query we call the query a correlated sub query.

Listing 11 Correlated Subquery

var f1 = inseq1

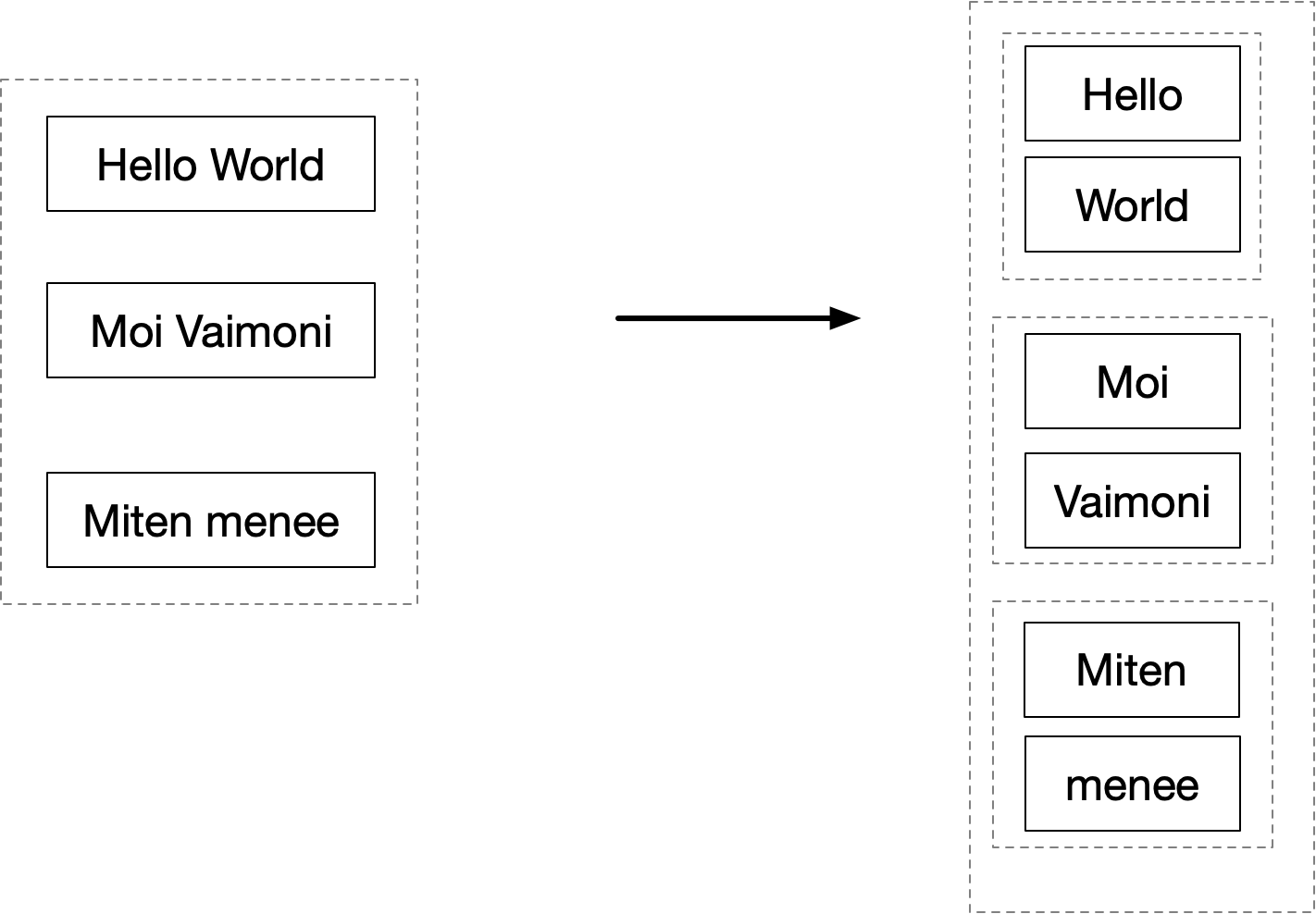
.Select(i => new

{

Words = i.Split()

.Select(j => j)

});



#### Cross Product (UNFLATtENED)

Uncorrelated queries mean when the sub query does not reference the elements of the outer query. This generates a kind of hierarchical cross product type result. It is not quite a cross product as per a relational database as there is no flattening

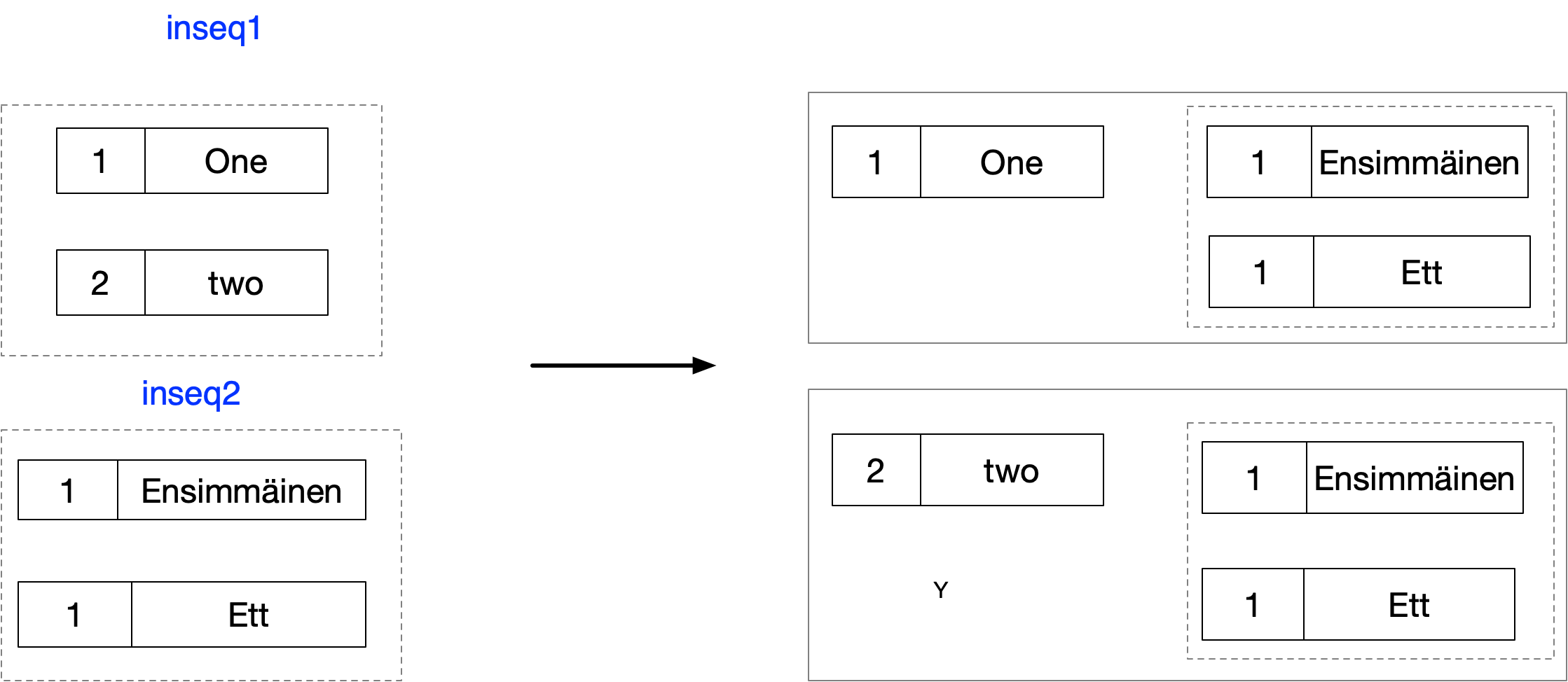
Listing 12Uncorrelated Subqueris - Cross Product

var inseq1 = new[] { (1, "one"), (2, "two"), };

var inseq2 = new[] { (1, "Ensimmäinen"), (1, "Ett") };

var f1 = inseq1.Select(i => (i,inseq2.Select(j=>j) ));

f1.Dump();



#### LEFT OUTER Join (Unflattened)

The following code shows how to perform a left outer join style query using select. Unlike a relational database join the data in not flattened. We will show how to flatten the data in the section on SelectMany. Also this is an inefficient way to perform a left equi join. In the section on GroupJoin we will show how to achieve this result in a more efficient means as GroupJoin uses a lookup internallly to prevent multiple linear traversals of the inner sequence.

Listing 13 Uncorrelated Subquery and Join

var inseq1 = new[] { (1, "one"), (2, "two"), };

var inseq2 = new[] { (1, "ensimmainen"),(1, "ett") };

var f1 = inseq1.Select(i =>

{

return new {

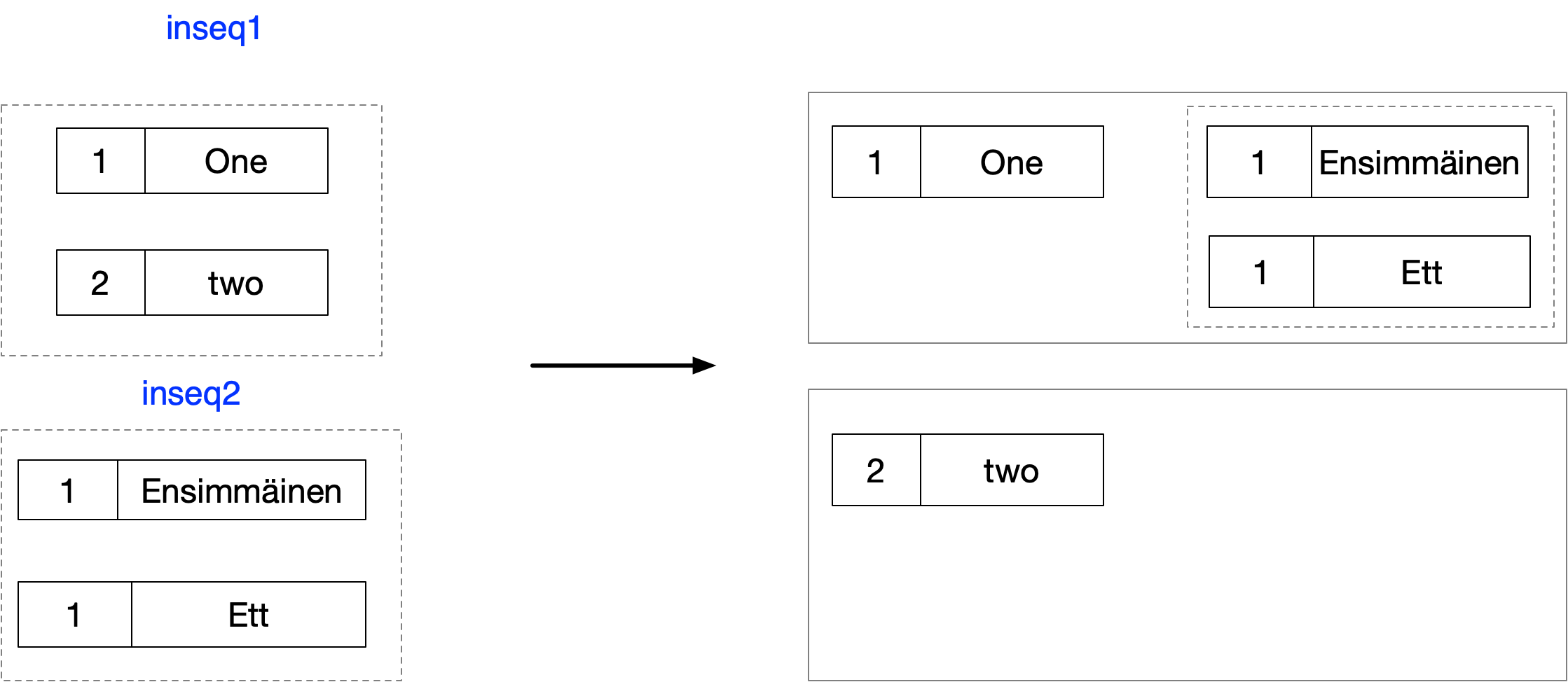
Left = i,

Rigth = inseq2

.Where(j => i.Item1 == j.Item1)

};

});



#### Implementing SELECT

Listing 14 Select Implementation

public static IEnumerable<TResult> Select<TIn,TResult>(this IEnumerable<TIn> source, Func<TIn,TResult> project)

{

foreach (var element in source)

yield return project(element);

}

public static IEnumerable<TResult> Select<TIn,TResult>(this IEnumerable<TIn> source, Func<TIn,int, TResult> project)

{

int i = 0;

foreach (var element in source)

yield return project(element,i++);

}

### SelectMany

SelectMany can be used to carry out the following

* Subsequence concatenation / flattening
* Cross Joins
* Non-equi joins
* Inner joins
* Left outer joins

We now consider each in turn

#### Flattening Subsequences

Listing 15 SelectMany - Flattening Suqsequences

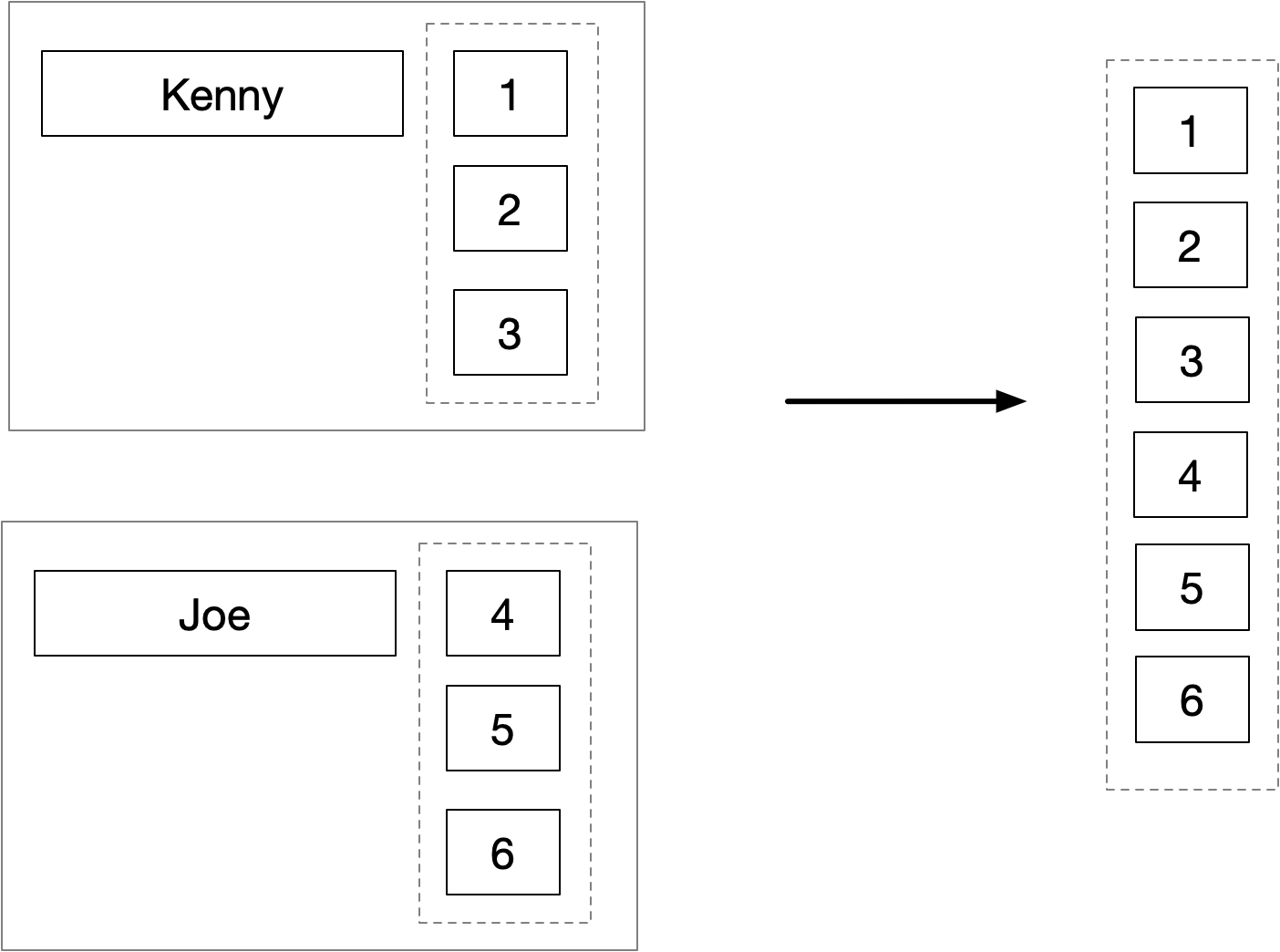
var parit = new[] { ("Kenny", new[] { 1, 2, 3 }),

("Joe", new[] { 4, 5, 6 }) };

var seq2 =

parit

.SelectMany(s => s.Item2);

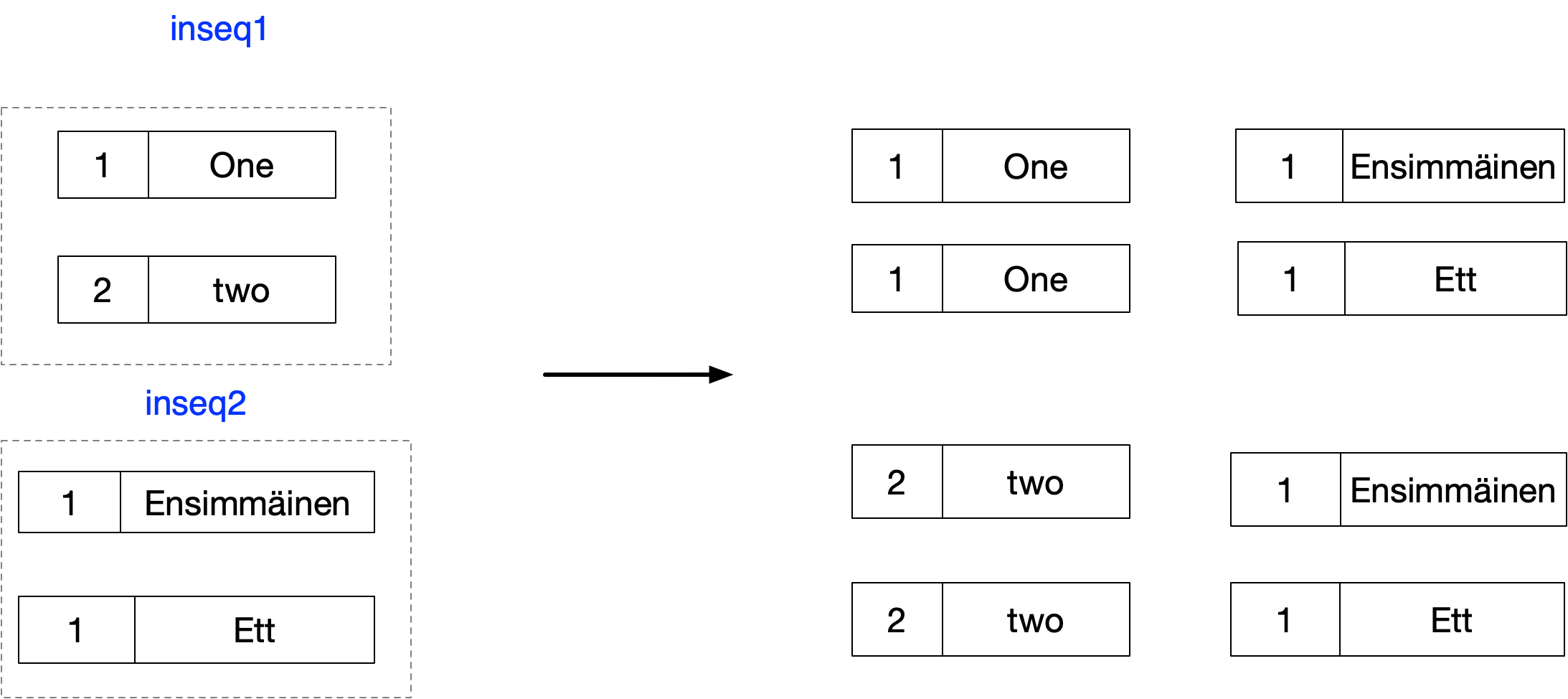


#### Cross Joins

var inseq1 = new[] { (1, "one"), (2, "two"), };

var inseq2 = new[] { (1, "Ensimmäinen"), (1, "Ett") };

var f1 = inseq1.SelectMany( i => inseq2.Select(j => (i,j)));



#### Inner Join (Flattened)

The following performs an inner join using SelectMany. This code is inneficient as it requires a complete traversal of the inner sequence for each element of the outer sequence

var inseq1 = new[] { (1, "one"), (2, "two"), };

var inseq2 = new[] { (1, "Ensimmäinen"), (1, "Ett") };

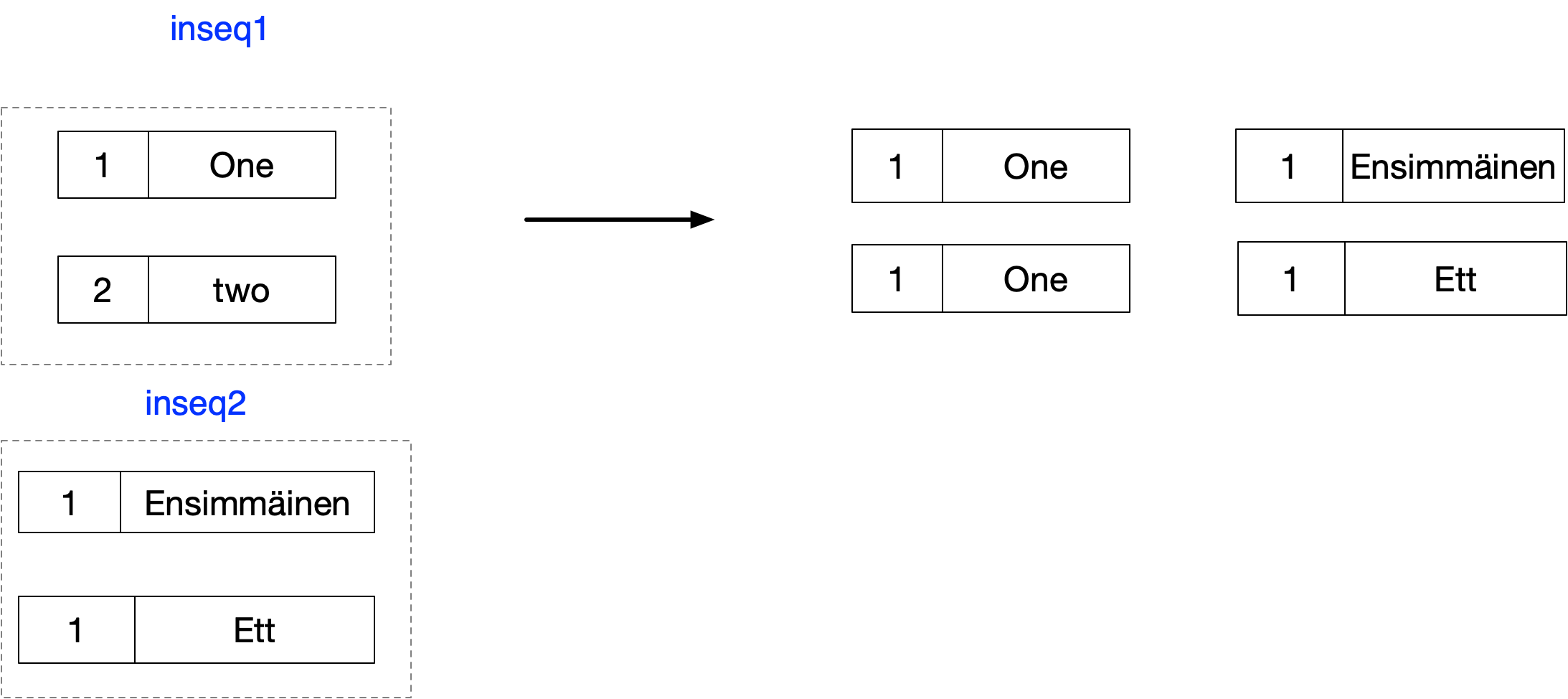
var f1 = inseq1.SelectMany( outer =>

inseq2

.Where(inner => outer.Item1 == inner.Item1)

.Select(inner => (outer,inner)));

s



#### Implementing SELECTMANY

public static IEnumerable<TResult> SelectMany<TSource,TResult>(

this IEnumerable<TSource> source,

Func<TSource,IEnumerable<TResult>> project)

{

foreach (var element in source)

foreach (var subelement in project(element))

yield return subelement;

}

public static IEnumerable<TResult> SelectMany<TSource,TCollection, TResult>(

this IEnumerable<TSource> source,

Func<TSource, IEnumerable<TCollection>> collectionSelector,

Func<TSource,TCollection,TResult> resultSelector)

{

foreach (var sourceElement in source)

{

foreach (var subelement in collectionSelector(sourceElement))

{

yield return resultSelector(sourceElement,subelement);

}

}

}

public static IEnumerable<TResult> SelectMany<TIn, TResult>(

this IEnumerable<TIn> source,

Func<TIn,int, IEnumerable<TResult>> project)

{

int i = 0;

foreach (var element in source)

foreach (var subelement in project(element,i++))

yield return subelement;

}

#### Subsequence concatenation / flattening

### OrderBy/ThenBy/OrderByDescencing/ThenByDescending

string[] seq = {"Four","Two", "One","Three"};

// {Two,One,Four,Three}

seq.OrderBy(s => s.Length);

// {Three,Four,Two,One}

seq.OrderByDescending(s => s.Length);

// {One,Two,Four,Three}

seq.OrderBy(s => s.Length).ThenBy(s => s);

// {Three,Four,Two,One}

seq

.OrderByDescending(s => s.Length)

.ThenByDescending(s => s)

### Aggregation

Aggregation can lead to some surpising results if we do not use a seed. Especially where we want to parallelize we need a function that can combine sub results which is both commutative and assocaitive

int[] s = new[] {2,3,4};

// 27 rather than 29

s.Aggregate ((x, y) => x+y\*y).Dump();

// Fix with seed

s.Aggregate (0,(x, y) => x+y\*y).Dump();

// For parallelisation we often specify a separate function

// for combining intermediate results. This function must be

// assocaite and commutative

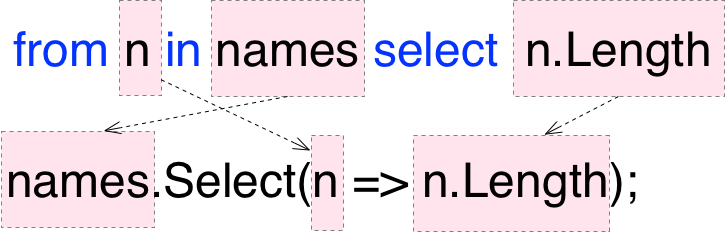
s.AsParallel().Aggregate (()=>0,(a,e) => a+e\*e,(a1,a2) => a1+a2,a=>a).Dump();

## Query Syntax

Linq also supports an alternative syntax called query syntax. Query syntax supports only a subset of the LINQ operators.

### Range variables

The following shows how a very simple query syntax query is mapped by the compiler to a fluent query. Notice how the range variable n, defined in the query, maps to the left hand side of the lambda expression in the generated fluent query. The expression to the right of select maps to the right hand side of the lambda in the fluent query



### Transparent identifiers

Let allows a query syntax query to introduce a second range variable which keeping the original range variable in scope. The following query uses the let keyword to introduce a new range variable.

var names = new[] { "Wren", "Bill", "Bob", "Will" };

IEnumerable<(char,string)> s1 =

from n in names

**let c = n[0]**

orderby c

select (c,n);

Notice how the select clause can now reference two range varaibles. If we consider how we might translate this to fluent syntax we can see the compiler is doing some extra work for us.

IEnumerable<(char,string)> s2 =

names

.Select(n => **new {c=n[0], n=n}**)

.OrderBy(x => **x.c**)

.Select(x => (**x.c,x.n**));

Notice how we had to use an anonymous type to support the extra range variable **c**. This is what the compiler does and it is known as transparent identifiers.

### Ordering

The following shows how to implement ordering using query syntax

IEnumerable<string> s1 =

from n in names

**orderby n.Length,n**

select n.ToUpper();

we can map this to fluent syntax as follows

IEnumerable<string> s2 =

names

**.OrderBy(n => n.Length)**

**.ThenBy(n => n)**

.Select(n => n.ToUpper());

If we want to order by in descending order we can use the following

IEnumerable<string> s3 =

from n in names

orderby n.Length **descending**, n **descending**

select n.ToUpper();

### SelectMany

We can generate a SelectMany in query syntax by using two from clauses.

IEnumerable<char> q1 =

**from n in names**

**from c in n**

select c;

The corresponding fluent syntax is as follows

IEnumerable<char> f1 =

**names.SelectMany(n => n)**;

If, however we have anything after the select clause in the query expression the compiler uses a transparent identifier to make both range range variables available for subsequent queries

IEnumerable<char> q2 =

from n in names

from c in n

where n == "Kenny" && c == 'n'

select Char.ToUpper(c);

The following shows how we can do this with anonymous types as the compiler might with transparent identifiers

IEnumerable<char> f2 =

names

.SelectMany(n => n, (n, c) => **new {n,c}**)

.Where(**x** => **x**.n=="Kenny" && **x**.c=='n')

.Select(**x** => Char.ToUpper(**x**.c));

### Join

We can join two sequences using the keyword join as follows

var s1 =

from ol in outerSeq

**join il in innerSeq on ol.Item1 equals il.Item1**

select (ol.Item2,il.Item2);

This can be translated into a fluent query as follows. Notice the final select (projection) is mapped directly to the projection function argument of Join

var f1 = outerSeq.Join(

innerSeq,

ol=>ol.Item1,

il=>il.Item1,

(ol,il) => (ol.Item2,il.Item2));

If, however we have anything after the select clause in the query expression the compiler uses a transparent identifier to make both range range variables available for subsequent queries

var s2 =

from ol in outerSeq

join il in innerSeq on ol.Item1 equals il.Item1

where ol.Item2 == "one" && il.Item2 == "Ett"

select (ol.Item2, il.Item2);

The following shows how we can express this using anonymous types in a similar fashion to what the compiler would do with transparent identifiers.

var f2 = outerSeq

.Join(

innerSeq,

ol => ol.Item1,

il => il.Item1,

(ol, il) => **new {ol,il}**)

.Where(x=>**x**.ol.Item2 == "one" && **x**.il.Item2 == "Ett")

.Select(x => (**x**.ol.Item2, **x**.il.Item2));

### GroupJoin

We can specifiy a group join using query syntax as follows. We specify an into clause directly after the join clause

Into and GroupJoin or Query Continuation

If a query systax query contains an into clause directly after a join clause it is translated as a GroupJoin. After select or group it causes query continutation which is quite different

var s1 =

from ol in outerSeq

**join il in innerSeq on ol.Item1 equals il.Item1**

**into matches**

select (ol.Item2,matches);

We can translate this into fluent syntax as follows.

var f1 = outerSeq.GroupJoin(

innerSeq,

ol=>ol.Item1,

il=>il.Item1,

(ol,il) => (ol.Item2,il));

As with join if there is only a simple select after the group join the select is implemented simply as the projection expression passed to the GroupJoin operator. If there is anything else then the compiler has to use a transparent identifier.

var q2 =

from ol in outerSeq

join il in innerSeq on ol.Item1 equals il.Item1

into matches

where ol.Item2 == "one" && matches.Count() == 2

select (ol.Item2, matches);

The following shows how we can express this using anonymous types in a similar fashion to what the compiler would do with transparent identifiers.

var f2 = outerSeq

.GroupJoin(

innerSeq,

ol => ol.Item1,

il => il.Item1,

(ol, matches) => **new {ol,matches})**

.Where(**x**=>**x**.ol.Item2 == "one" && **x**.matches.Count() == 2)

.Select(**x** => (**x**.ol.Item2, **x**.matches));

### Left Outer Join

A left outer join is a little bit tricky. A join performs flattening but gives us an inner join. A group join gives us outer join like functionality without flattening. The solution is to use groupjoin together with a select many to flatten. The query syntax is as follows

IEnumerable<(string, string)> q1 =

from outerEl in outerSeq

join innerEl in innerSeq on outerEl.Item1 equals innerEl.Item1

into matches

from e in matches.DefaultIfEmpty()

select default((int, string)).Equals(e)

? (outerEl.Item2, "")

: (outerEl.Item2, e.Item2);

And the fluent query is as follows. Note we add a function to simply the result generation and default check.

(string, string) ResGenFunc((int, string) outerEl, (int, string) innerEl) =>

default((int, string)).Equals(innerEl)

? (outerEl.Item2, "")

: (outerEl.Item2, innerEl.Item2);

IEnumerable<(string, string)> f1 =

outerSeq

.GroupJoin( innerSeq,

oel => oel.Item1,

inel => inel.Item1,

(oel, inEls) => inEls

.DefaultIfEmpty()

.Select(innerEl => ResGenFunc(oel, innerEl)))

.SelectMany(el => el);

### Joining

#### Implementing Join

public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(

this IEnumerable<TOuter> outer,

IEnumerable<TInner> inner,

Func<TOuter, TKey> outerKeySelector,

Func<TInner, TKey> innerKeySelector, Func<TOuter,

TInner, TResult> resultSelector)

{

var lookup = inner.ToLookup(innerKeySelector);

foreach (var outerEl in outer)

foreach (var innerEl in lookup[outerKeySelector(outerEl)])

yield return resultSelector(outerEl, innerEl);

}

#### Inner Joins

(int, string)[] outerSeq = { (1, "one"), (2, "two"), };

(int, string)[] innerSeq = { (1, "Ensimmäinen"), (1, "Ett") };

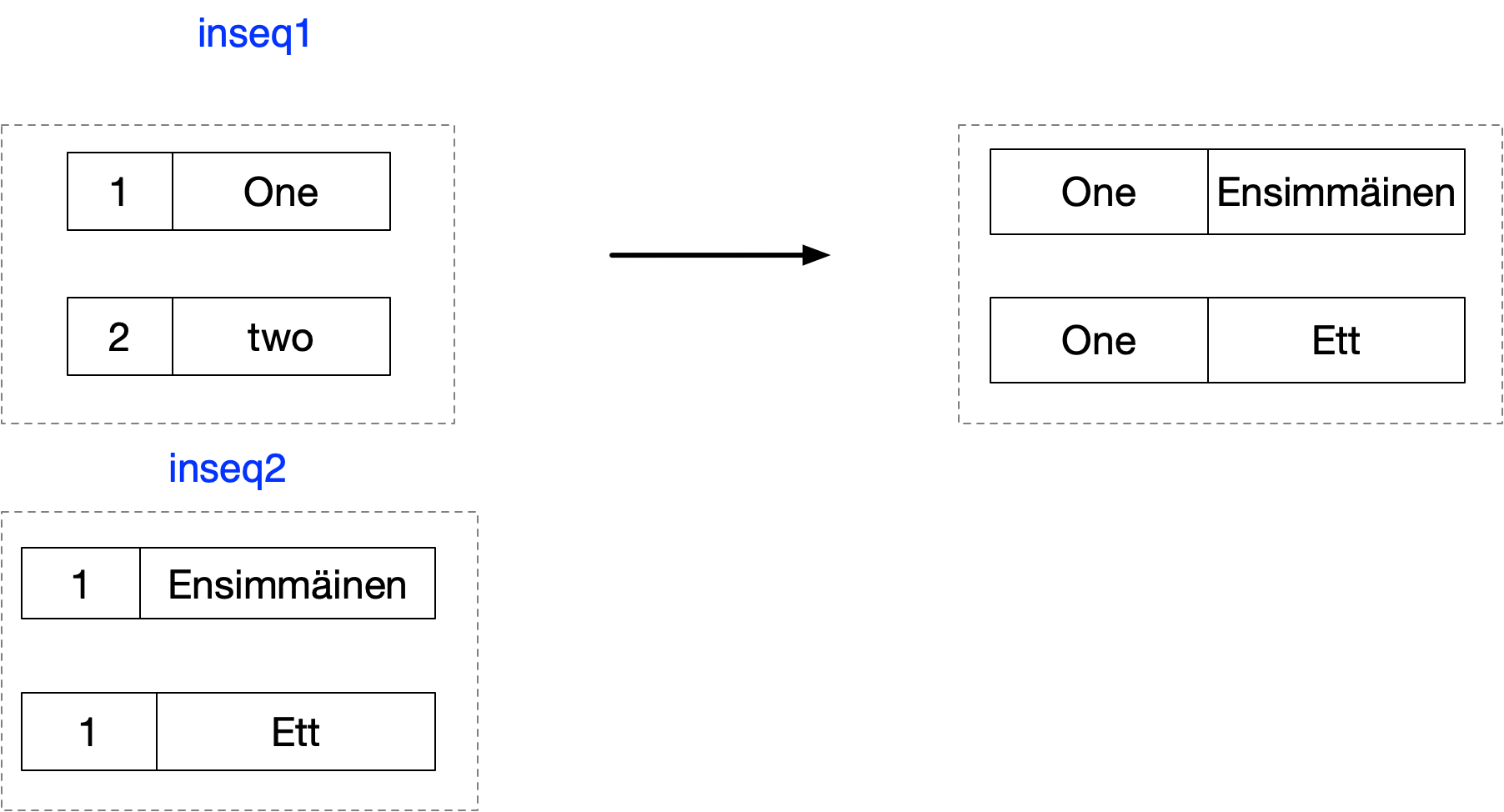
var res1 =

from outer in outerSeq

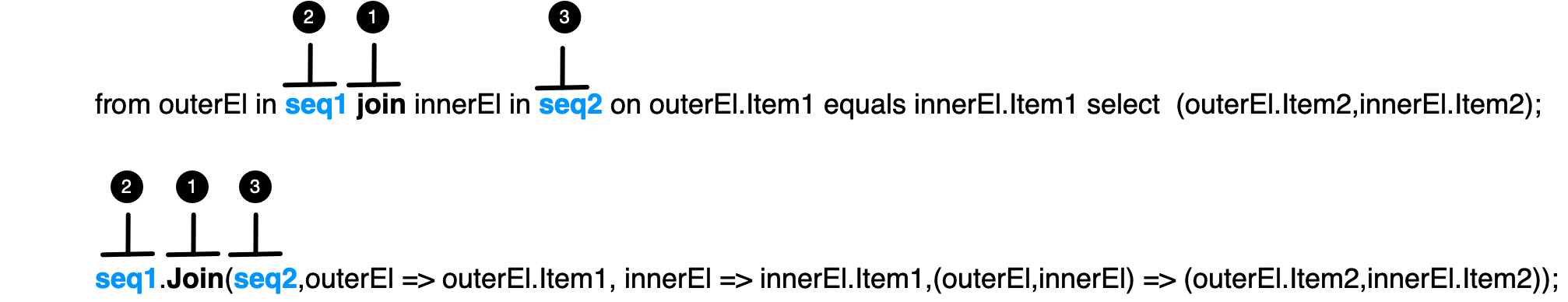
join inner in innerSeq

on outer.Item1 equals inner.Item1

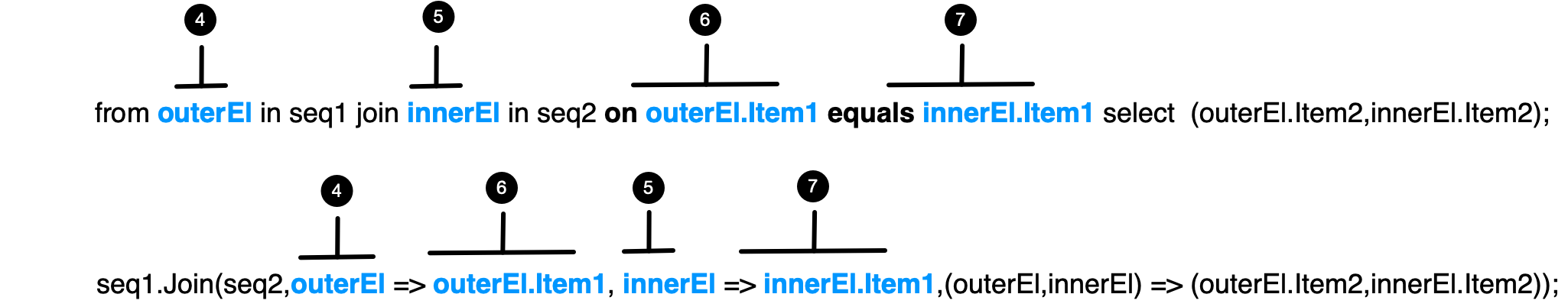
select (outer.Item2,inner.Item2);



Let us consider how the query with a join is mapped to the operator Enumerable.Join. First consider the sequences and the contextual keyword join. The keyword join instructs the compiler to invoke the Join operator on the sequence defined in the expression before the join keyword to the expresion defined after the in keyword.

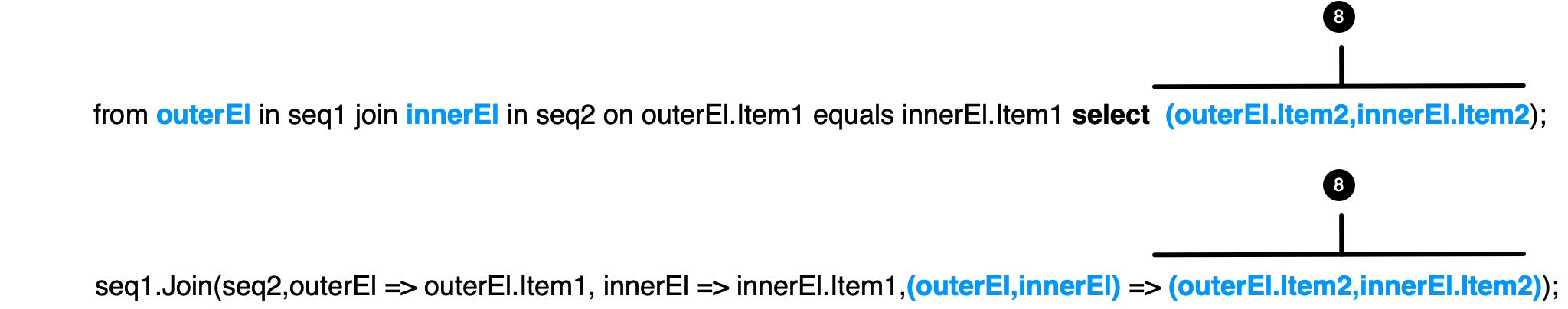


Then the keywords on and equals are used to define the keys of the outer an inner sequence respectively



The order of the key selectors is important. For the first key selector after the on keyword only the range varaible of the outer sequence is in scope. For the second key selector only the range variable of the second sequence is in scope.

Finally we consider how the query expression maps to the final projection function parameter of the join operator. In our simple query expression there is nothing after the join other than a simple select. In this case the selet is directly mapped into the selector function



In the case where we have something other than a simple select after the join the compiler has to work some **transparent idenfier** magic to make sure the elements from both the inner and outer sequences are available to operators after the join.

### Example One

Listing 16Query Syntax

var names = new List<String> (new[] { "Wren", "Bill", "Bob", "Will" });

IEnumerable<string> s1 =

from n in names

where n.StartsWith("W")

orderby n

select n.ToUpper();

Listing 17 Fluent Syntax

IEnumerable<string> s2 = names

.Where(n => n.StartsWith("W"))

.OrderBy(n => n)

.Select(n => n.ToUpper());

### Example Two

This example shows a situation where query syntax is more elegant than the corresponding fluent syntax query. We have two range variabes in the query syntax query which remain in scope for subsequent clauses. Note how the orderby and select access both range variables. In this example we are flattening

Listing 18Query Syntax

IEnumerable<string> parit = new[] { "Minun äiti", "Suomi on Mun", "Iso ranta", "Sanan isi", };

// A SelectManyQuery which wants to access both the outer elements and the

// flattened inner elements can be easier to write in query syntax.

from p in parit

from s in p.Split()

where s.Contains("u")

orderby p,s

select $"{p} -> {s}"

Listing 19 Fluent Syntax

parit

.SelectMany(pari => pari.Split().Select(sana => ( pari, sana)))

.Where( x=> x.sana.Contains("u"))

.OrderBy(x=>x.pari)

.OrderBy(x=>x.sana)

.Select(x => $"{x.pari} -> {x.sana}")

## Subqueries

## Projecting

## Subqueries

We can write quite ineffient queries using subqueries in LINQ

string[] names = new[] {"Kenny", "John", "Bob", "Jimmy","Rob"};

var enumerable1 = from n in names

where n.Length == names.Min(s => s.Length)

select n;

We can make this code much more efficient by restructuring as follows

var min = names.Min(s => s.Length);

IEnumerable<string> enumerable2 = from n in names

where n.Length == min

select n;

Questions

**What is LINQ?**

A language feature that enables us to write type safe queries over any collection that implements IEnumerable<T>

**What inspired LINQ?**

The functional programming paradigm

**What are the basic elements**

* Sequences
* Elements
* Query operators
* Queries

**What do lambda expressions in query operators always operate on?**

Individual elements

**Do query operators alter the input sequence?**

No they always generate a new sequence

**What does LINQ query comprise?**

A pipeline of operators that accept and return ordered sequences

**What does an SQL query comprise?**

A netwoek of clauses working on unordered sets

**How is deferred execution implemented?**

Query operators provide deferred execution by returning decorator sequences.

**What are the advantages of deferred execution?**

* Decouples construction from execution
* Allows one to construct a query in multipe steps
* You can re-evaluate a query by enumerating it again.

**What are the exceptions that return immediately?**

ToList, ToArray, ToDictionary, ToLookup

Single element or scalar operators such as First or Count

**How do decorator sequences differ from traditional collection classes?**

In general a decorator sequence has no storage of its own to store elements

**What does it have instead?**

A reference to another sequence supplied at runtime

**What happends when you request data from a decorator?**

It must in turn ask for data from its wrapped input sequence

**What happens when you chain query operators?**

A chain of decorators are created

**What happens when you enumerate a query?**

You query the original input sequence transformed thorugh a layering chain of decorators

**What happens if you call ToList() on query?**

The whole chain is collapsed into a single list