Working with Sequences in a Functional Way

Zoran Horvat
CEO AT CODING HELMET

@zoranh75 http://csharpmentor.com

Using Collections

Consume a collection of objects

This module

Modify a collection of objects

Next module

Understanding Collections of Objects

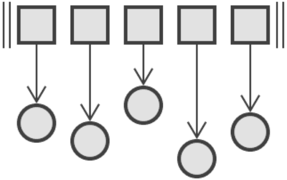
List

Understanding Collections of Objects

List

Array

Understanding Collections of Objects

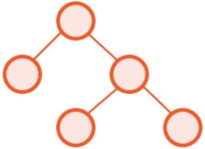


List

Array

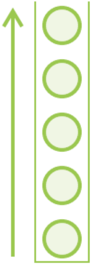
Dictionary

Understanding Collections of Objects



List
Array
Dictionary
Tree

Understanding Collections of Objects



List

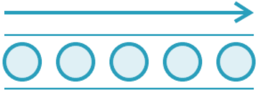
Array

Dictionary

Tree

Stack

Understanding Collections of Objects



List

Array

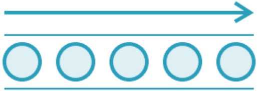
Dictionary

Tree

Stack

Queue

Understanding Collections of Objects



List

Array

Dictionary

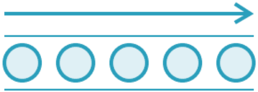
Tree

Stack

Queue

used to build a compiler

Understanding Collections of Objects



List

Array

Dictionary

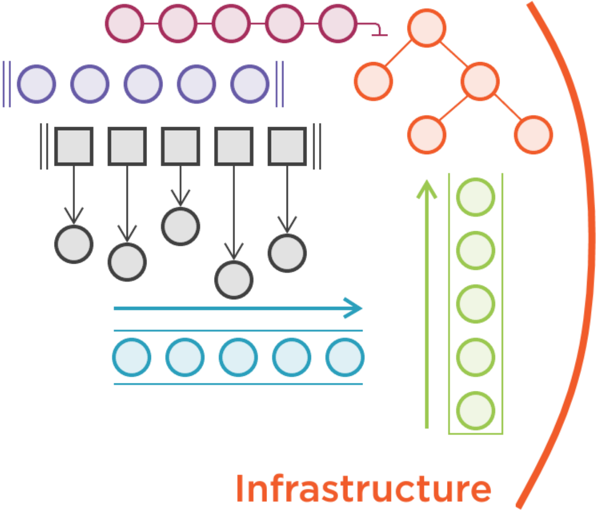
Tree

Stack

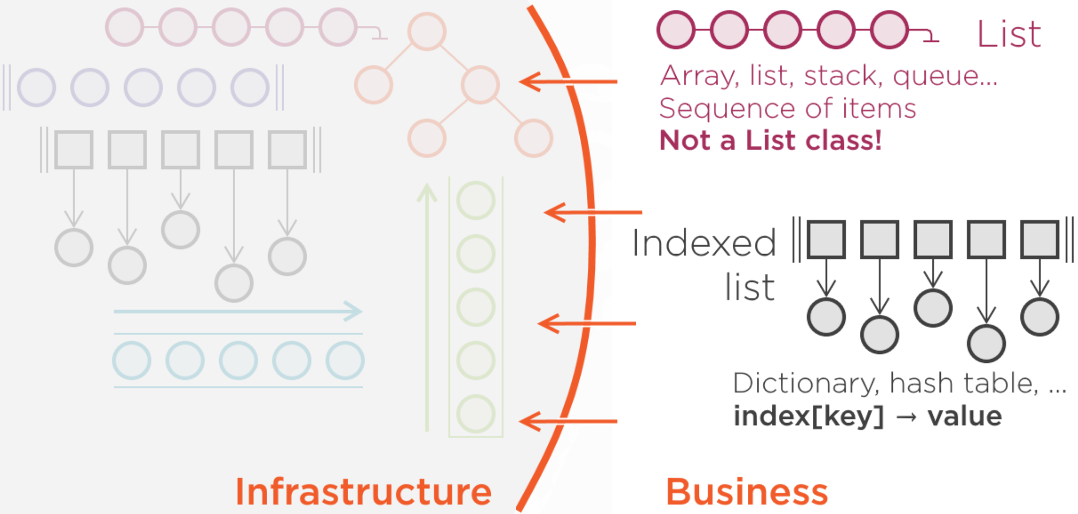
Queue

used to pass an exam

Understanding Collections of Objects



Understanding Collections of Objects



List

Array, list, stack, queue...

Sequence of items

Not a List class!

Indexed list

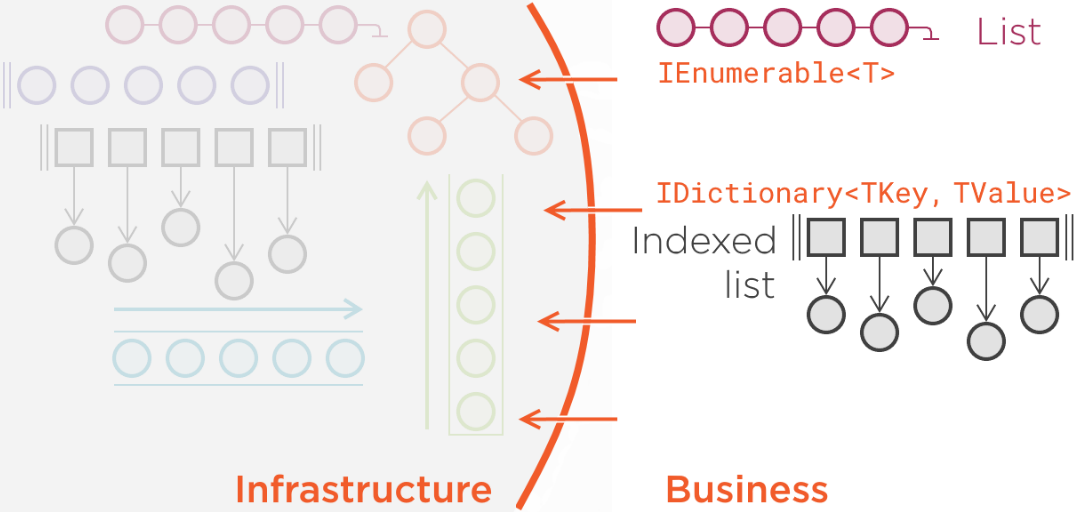
Dictionary, hash table, ...

index[key] → value

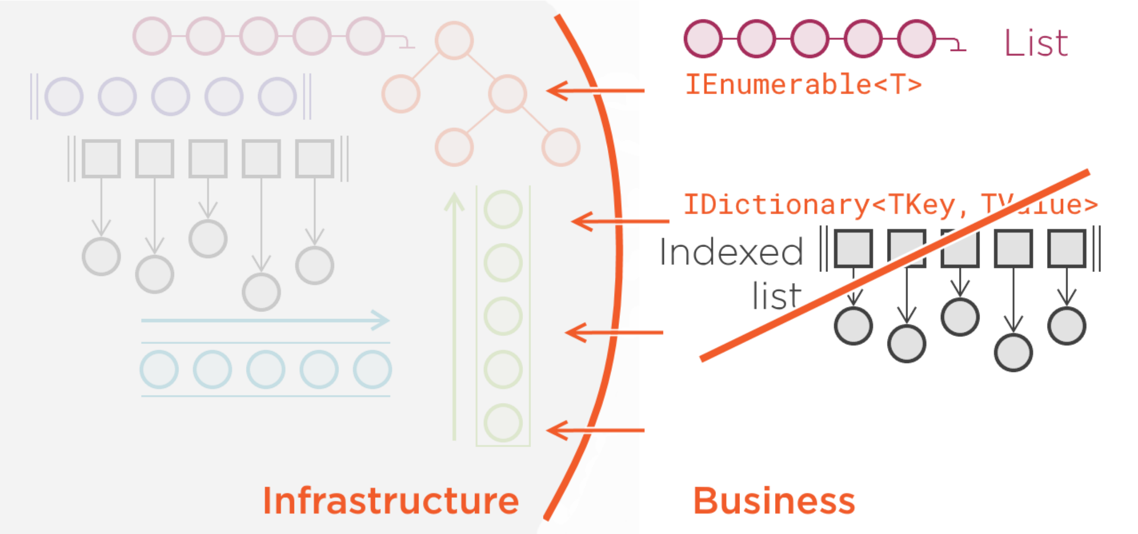
Infrastructure

Business

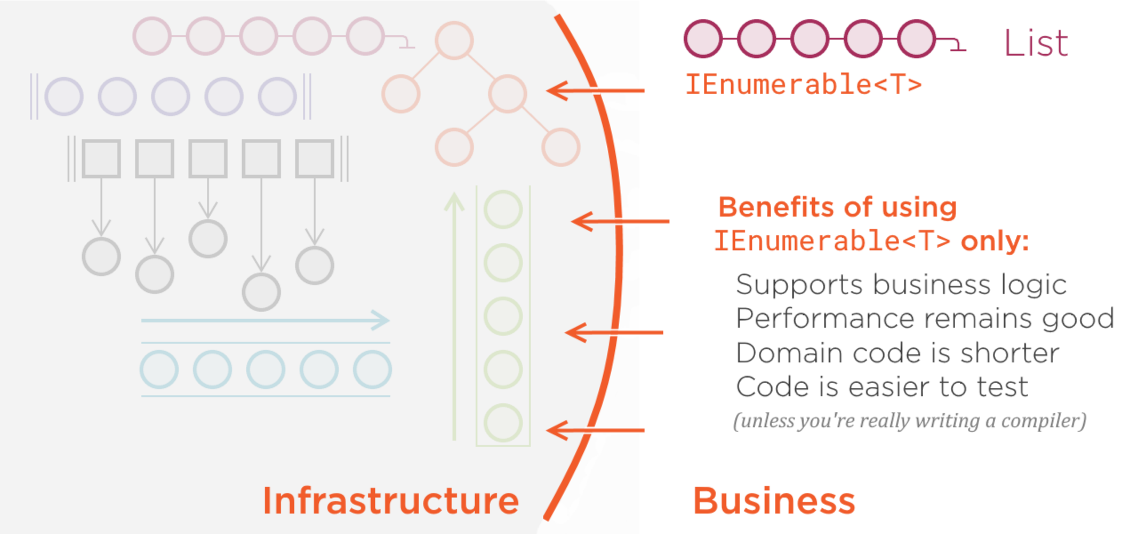
Understanding Collections of Objects



Understanding Collections of Objects



Understanding Collections of Objects

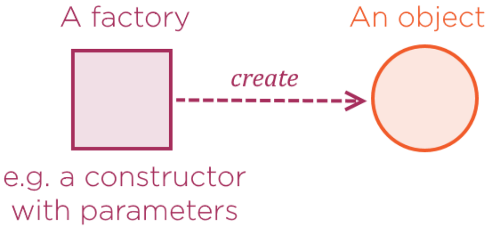


Benefits of using `IEnumerable<T>` only:

Supports business logic
Performance remains good
Domain code is shorter
Code is easier to test

(unless you're really writing a compiler)

Understanding .NET List Comprehension



Can we construct a sequence like a common object?

Understanding .NET List Comprehension

Set comprehension (mathematics)

$$P = \{2k | k \in N\}$$

$$\{2, 4, 6, 8, \dots\}$$

Set

An unordered collection of values.

Only defines "belongs to" relation.

Sequence comprehension (mathematics)

$$S = (a_k)_{k=1}^{\infty},\ a_i = 2i$$

yields $(2, 4, 6, 8, \dots)$

Infinite sequence

Possible on the set of natural numbers.

Not possible on `Int32`!

Understanding .NET List Comprehension

Set comprehension (mathematics)

$$P = \{2k | k \in N\}$$

Sequence comprehension (mathematics)

$$S = (a_k)_{k=1}^{\lfloor MaxInt/2 \rfloor},\ a_i = 2i$$

List comprehension with `yield return`

```
IEnumerable<int> Evens()
{
  int n = 0;
  while (n <= Int32.MaxValue - 2)
  {
    n += 2;
    yield return n;
  }
}
```

```
Evens().Take(12).Count();
```

Exactly 12 iterations will execute!

Understanding .NET List Comprehension

Set comprehension (mathematics)

$$P = \{2k | k \in N\}$$

List comprehension with **yield return**

```
IEnumerable<int> Evens()
{
  int n = 0;
  while (n <= Int32.MaxValue - 2)
  {
    n += 2;
    yield return n;
  }
}
```

Sequence comprehension (mathematics)

$$S = (a_k)_{k=1}^{\lfloor MaxInt/2 \rfloor}, a_i = 2i$$

List comprehension with **Select**

```
IEnumerable<int> Evens() =>
  Enumerable.Range(1, Int32.MaxValue)
    .Where(i => i % 2 == 0);
```

Understanding .NET List Comprehension

Set comprehension (mathematics)

$$P = \{2k | k \in N\}$$

List comprehension with **yield return**

```
IEnumerable<int> Evens()
{
  int n = 0;
  while (n <= Int32.MaxValue - 2)
  {
    n += 2;
    yield return n;
  }
}
```

Sequence comprehension (mathematics)

$$S = (a_k)_{k=1}^{\lfloor MaxInt/2 \rfloor}, a_i = 2i$$

List comprehension with **Select**

```
IEnumerable<int> Evens() =>
  Enumerable.Range(1, Int32.MaxValue)
    .Where(i => i % 2 == 0);
```

Filtering is not an efficient strategy

Dividing by 100 would throw away 99% of objects!

Understanding .NET List Comprehension

Set comprehension (mathematics)

$$P = \{2k | k \in N\}$$

List comprehension with **yield return**

```
IEnumerable<int> Evens()
{
  int n = 0;
  while (n <= Int32.MaxValue - 2)
  {
    n += 2;
    yield return n;
  }
}
```

Sequence comprehension (mathematics)

$$S = (a_k)_{k=1}^{\lfloor MaxInt/2 \rfloor}, a_i = 2i$$

List comprehension with **Select**

```
IEnumerable<int> Evens() =>
  Enumerable.Range(1, Int32.MaxValue / 2)
    .Select(i => 2 * i);
```

Understanding .NET List Comprehension

Set comprehension (mathematics)

$$P = \{2k | k \in N\}$$

List comprehension with **`yield return`**

```
IEnumerable<int> Evens()
{
  int n = 0;
  while (n <= Int32.MaxValue - 2)
  {
    n += 2;
    yield return n;
  }
}
```

Sequence comprehension (mathematics)

$$S = (a_k)_{k=1}^{\lfloor MaxInt/2 \rfloor}, a_i = 2i$$

List comprehension with **`Select`**

```
IEnumerable<int> Evens() =>
  Enumerable.Range(1, Int32.MaxValue / 2)
    .Select(i => 2 * i);
```

Lazy evaluated

Understanding .NET List Comprehension

Set comprehension (mathematics)

$$P = \{2k | k \in N\}$$

List comprehension with **yield return**

```
IEnumerable<int> Evens()
{
  int n = 0;
  while (n <= Int32.MaxValue - 2)
  {
    n += 2;
    yield return n;
  }
}
```

Sequence comprehension (mathematics)

$$S = (a_k)_{k=1}^{\lfloor MaxInt/2 \rfloor}, a_i = 2i$$

List comprehension with **Select**

```
IEnumerable<int> Evens() =>
  Enumerable.Range(1, Int32.MaxValue / 2)
    .Select(i => 2 * i);
```

Every index is used

Understanding .NET List Comprehension

Set comprehension (mathematics)

$$P = \{2k | k \in N\}$$

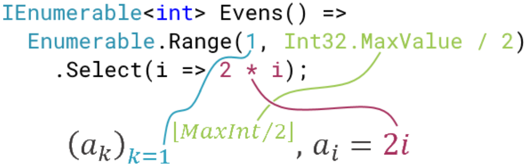
Sequence comprehension (mathematics)

$$S = (a_k)_{k=1}^{\lfloor MaxInt/2 \rfloor}, a_i = 2i$$

List comprehension with `yield return`

```
IEnumerable<int> Evens()
{
  int n = 0;
  while (n <= Int32.MaxValue - 2)
  {
    n += 2;
    yield return n;
  }
}
```

List comprehension with `Select`



```
IEnumerable<int> Evens() =>
  Enumerable.Range(1, Int32.MaxValue / 2)
    .Select(i => 2 * i);
```

$$(a_k)_{k=1}^{\lfloor MaxInt/2 \rfloor}, a_i = 2i$$

Understanding .NET List Comprehension

Set comprehension (mathematics)

$$P = \{2k | k \in N\}$$

List comprehension with **yield return**

```
IEnumerable<int> Evens()
{
  int n = 0;
  while (n <= Int32.MaxValue - 2)
  {
    n += 2;
    yield return n;
  }
}
```

Sequence comprehension (mathematics)

$$S = (a_k)_{k=1}^{\lfloor MaxInt/2 \rfloor}, a_i = 2i$$

List comprehension with **Select**

```
IEnumerable<int> Evens() =>
  Enumerable.Range(1, Int32.MaxValue / 2)
    .Select(i => 2 * i);
```

And what about **IEnumerable<IMoney>?**

LINQ Operators vs. Common Functions

List comprehension
with LINQ

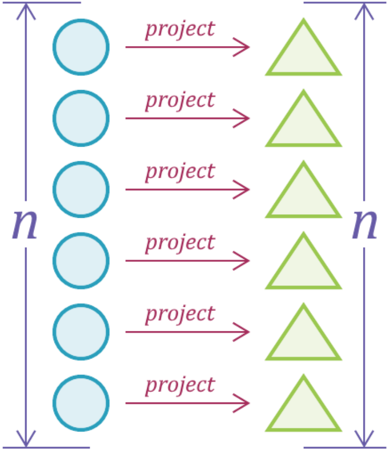
sequence $\xrightarrow{\textit{transform}}$ sequence

Object construction
with functions

object $\xrightarrow{\textit{function}}$ object

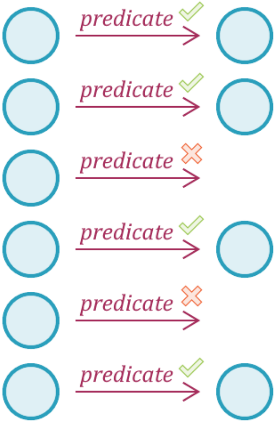
Understanding LINQ Operators

Mapping	Select(map)



Understanding LINQ Operators

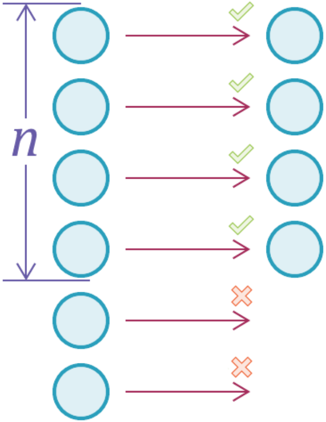
Mapping	Select(map)
Filtering	Where(predicate)



Understanding LINQ Operators

Mapping	Select(map)

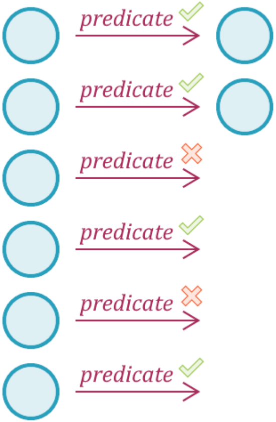
Filtering	Where(predicate)
	Take(n)



Understanding LINQ Operators

Mapping	Select(map)

Filtering	Where(predicate)
	Take(n), TakeWhile(predicate)



Understanding LINQ Operators

Mapping	Select(map)

Filtering	Where(predicate)
	Take(n), TakeWhile(predicate)
	Skip(n)

Understanding LINQ Operators

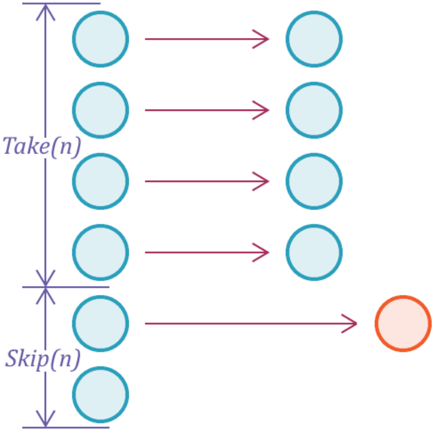
Mapping	Select(map)

Filtering	Where(predicate)
	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

Understanding LINQ Operators

Mapping	Select(map)
Filtering	Where(predicate)

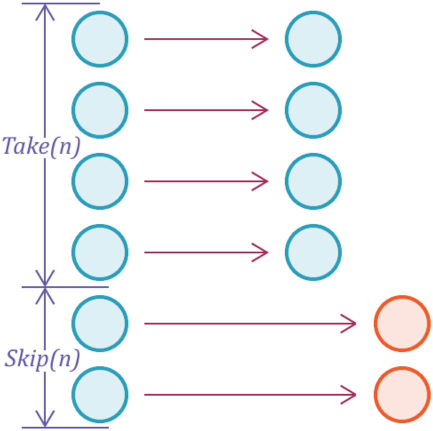
Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)



Understanding LINQ Operators

Mapping	Select(map)
Filtering	Where(predicate)

Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

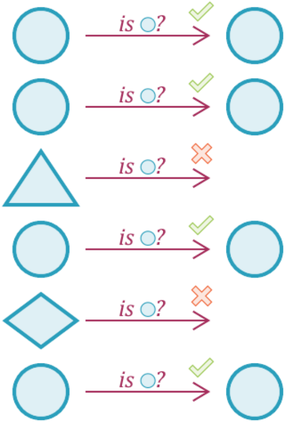


Understanding LINQ Operators

Mapping	Select(map)
Filtering	Where(predicate)

Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>

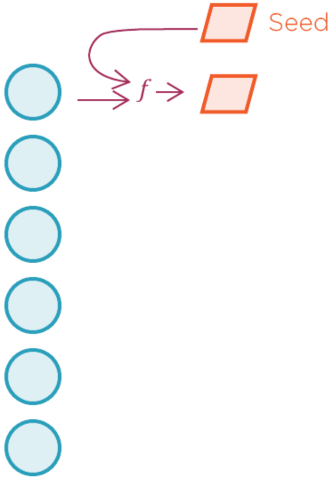


Understanding LINQ Operators

Mapping	Select(map)
Filtering	Where(predicate)

Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>
Aggregating	Aggregate(seed, f)

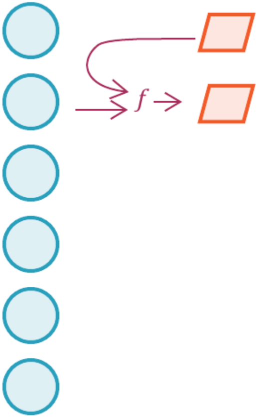


Understanding LINQ Operators

Mapping	Select(map)
Filtering	Where(predicate)

Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>
Aggregating	Aggregate(seed, f)

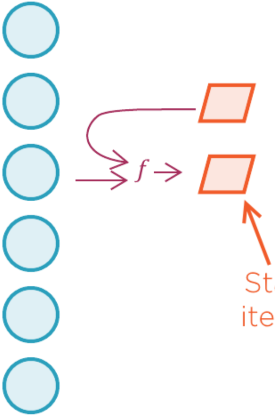


Understanding LINQ Operators

Mapping	Select(map)
Filtering	Where(predicate)

Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

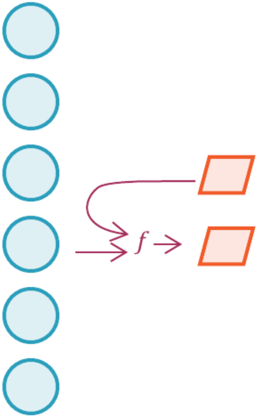
	OfType<T>
Aggregating	Aggregate(seed, f)



Stateful iteration

Understanding LINQ Operators

Mapping	Select(map)
Filtering	Where(predicate)
Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)
	OfType<T>
Aggregating	Aggregate(seed, f)

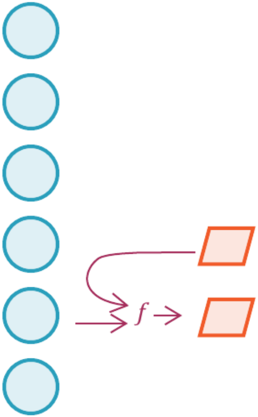


Understanding LINQ Operators

Mapping	Select(map)
Filtering	Where(predicate)

Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>
Aggregating	Aggregate(seed, f)

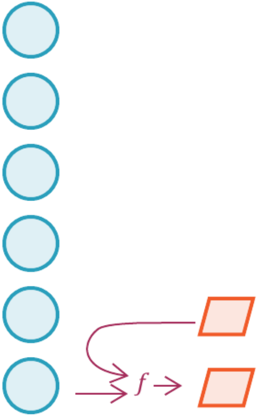


Understanding LINQ Operators

Mapping	Select(map)
Filtering	Where(predicate)

Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>
Aggregating	Aggregate(seed, f)

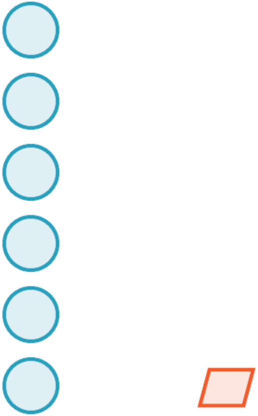


Understanding LINQ Operators

Mapping	Select(map)
Filtering	Where(predicate)

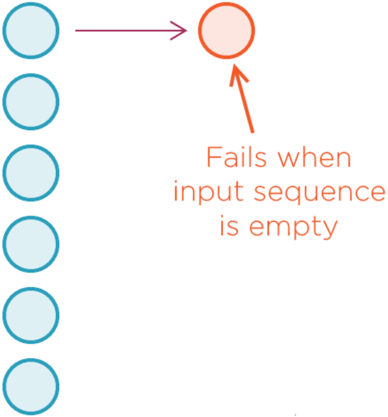
Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>
Aggregating	Aggregate(seed, f)



Understanding LINQ Operators

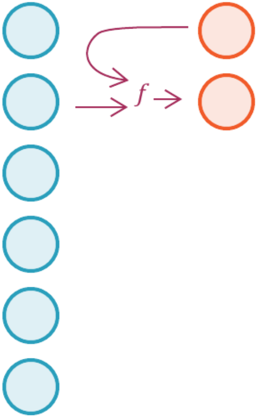
Mapping	Select(map)
Filtering	Where(predicate)
Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)
	OfType<T>
Aggregating	Aggregate(seed, f)
	Aggregate(f)



Fails when input sequence is empty

Understanding LINQ Operators

Mapping	Select(map)
Filtering	Where(predicate)
Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)
	OfType<T>
Aggregating	Aggregate(seed, f)
	Aggregate(f)



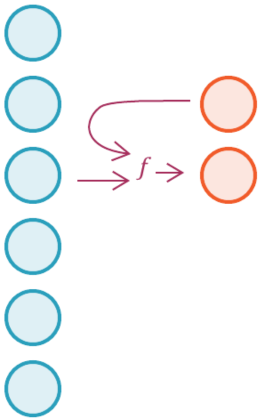
Understanding LINQ Operators

Mapping	Select(map)
Filtering	Where(predicate)

Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>

Aggregating	Aggregate(seed, f)
	Aggregate(f)



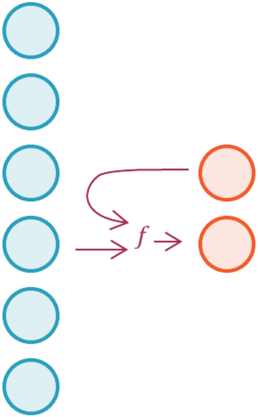
Understanding LINQ Operators

Mapping	Select(map)
Filtering	Where(predicate)

Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>

Aggregating	Aggregate(seed, f)
	Aggregate(f)



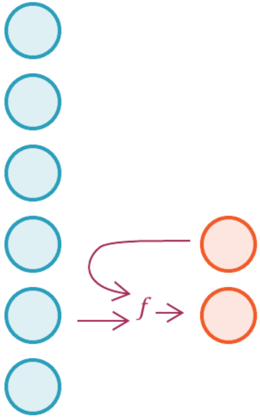
Understanding LINQ Operators

Mapping	Select(map)
Filtering	Where(predicate)

Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>

Aggregating	Aggregate(seed, f)
	Aggregate(f)



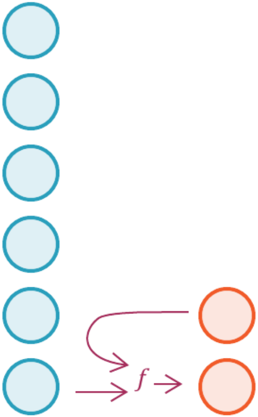
Understanding LINQ Operators

Mapping	Select(map)
Filtering	Where(predicate)

Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>

Aggregating	Aggregate(seed, f)
	Aggregate(f)



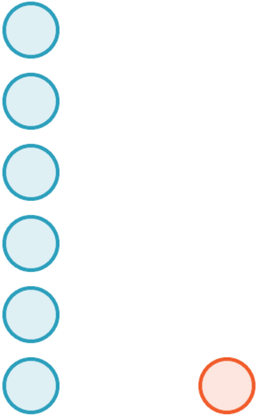
Understanding LINQ Operators

Mapping	Select(map)
Filtering	Where(predicate)

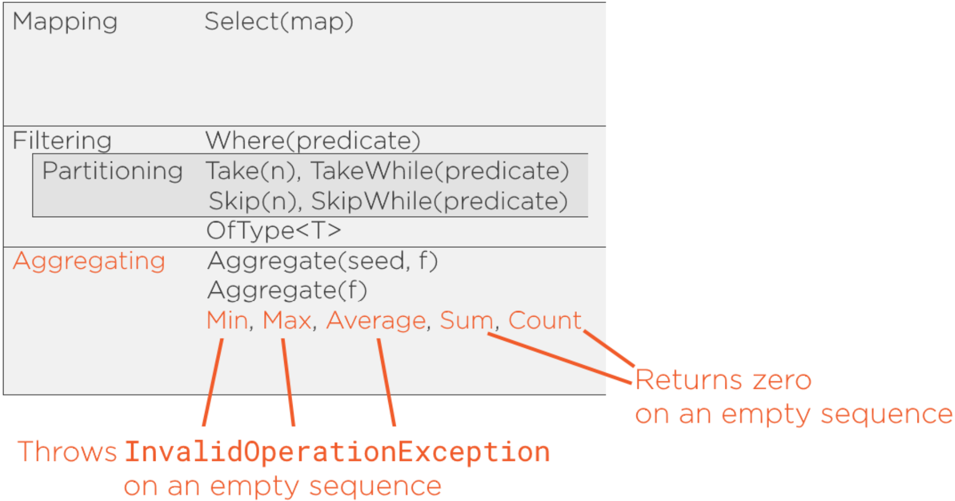
Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>

Aggregating	Aggregate(seed, f)
	Aggregate(f)



Understanding LINQ Operators



Mapping		Select(map)
Filtering		Where(predicate)

	Partitioning	Take(n), TakeWhile(predicate)
		Skip(n), SkipWhile(predicate)

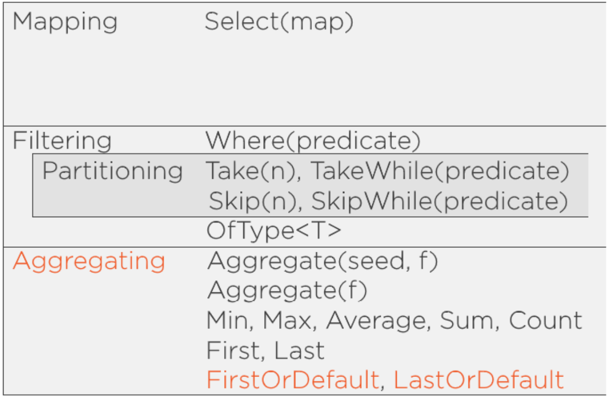
		OfType<T>

Aggregating		Aggregate(seed, f)
		Aggregate(f)
		Min, Max, Average, Sum, Count

Returns zero
on an empty sequence

Throws **`InvalidOperationException`**
on an empty sequence

Understanding LINQ Operators



Mapping	Select(map)
Filtering	Where(predicate)

Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>
Aggregating	Aggregate(seed, f) Aggregate(f) Min, Max, Average, Sum, Count First, Last FirstOrDefault, LastOrDefault

Understanding LINQ Operators

Mapping	Select(map)
Filtering	Where(predicate)

Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>
Aggregating	Aggregate(seed, f) Aggregate(f) Min, Max, Average, Sum, Count First, Last FirstOrDefault, LastOrDefault
Concatenation	SelectMany

Understanding LINQ Operators

Mapping	Select(map)
Filtering	Where(predicate)

Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>
Aggregating	Aggregate(seed, f) Aggregate(f) Min, Max, Average, Sum, Count First, Last FirstOrDefault, LastOrDefault
Concatenation	SelectMany Concat

Understanding LINQ Operators

Mapping	Select(map)
	GroupBy(keySelector)

Filtering	Where(predicate)

Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>
Aggregating	Aggregate(seed, f) Aggregate(f) Min, Max, Average, Sum, Count First, Last FirstOrDefault, LastOrDefault
Concatenation	SelectMany Concat

⚠ Eager execution!

Less efficient:

```
seq.GroupBy(x => x.key).Take(10)
```

More efficient:

```
seq.Select(x => x.key).Take(10)
```

Understanding LINQ Operators

Mapping	Select(map)
	GroupBy(keySelector)
	OrderBy(keySelector)

Filtering	Where(predicate)

Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>
Aggregating	Aggregate(seed, f) Aggregate(f) Min, Max, Average, Sum, Count First, Last FirstOrDefault, LastOrDefault
Concatenation	SelectMany Concat

⚠ Often questionable in domain logic

✓ Useful in presentation logic

Understanding LINQ Operators

Mapping	Select(map)
	GroupBy(keySelector)
	OrderBy(keySelector)
	Join

Filtering	Where(predicate)

Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>
Aggregating	Aggregate(seed, f) Aggregate(f) Min, Max, Average, Sum, Count First, Last FirstOrDefault, LastOrDefault
Concatenation	SelectMany Concat

Subset of a Cartesian product

```
leftSeq.Join(
  rightSeq,
  x => x.Id,
  y => y.Id,
  (x, y) => (x.a, y.b));
```

Looks like we need
this as a new type

Understanding LINQ Operators

Mapping	Select(map)
	GroupBy(keySelector)
	OrderBy(keySelector)
	Join

Filtering	Where(predicate)

Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>
Aggregating	Aggregate(seed, f) Aggregate(f) Min, Max, Average, Sum, Count First, Last FirstOrDefault, LastOrDefault
Concatenation	SelectMany Concat
Set operators	Distinct

Understanding LINQ Operators

Mapping	Select(map)
	GroupBy(keySelector)
	OrderBy(keySelector)
	Join

Filtering	Where(predicate)

Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>
Aggregating	Aggregate(seed, f) Aggregate(f) Min, Max, Average, Sum, Count First, Last FirstOrDefault, LastOrDefault
Concatenation	SelectMany Concat
Set operators	Distinct, Except

Understanding LINQ Operators

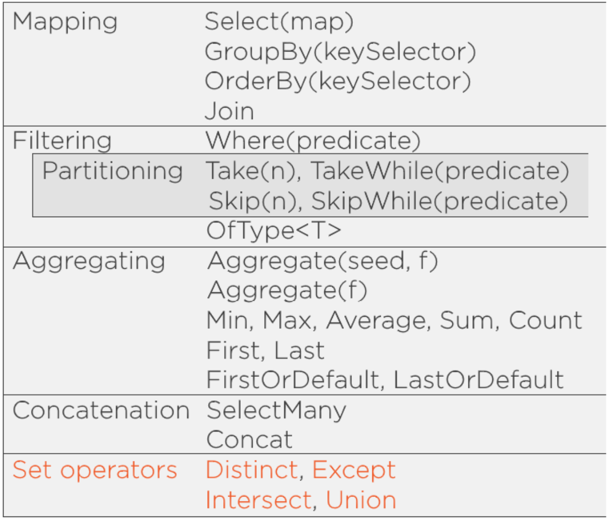
Mapping	Select(map)
	GroupBy(keySelector)
	OrderBy(keySelector)
	Join

Filtering	Where(predicate)

Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>
Aggregating	Aggregate(seed, f) Aggregate(f) Min, Max, Average, Sum, Count First, Last FirstOrDefault, LastOrDefault
Concatenation	SelectMany Concat
Set operators	Distinct, Except Intersect

Understanding LINQ Operators



Mapping	Select(map)
	GroupBy(keySelector)
	OrderBy(keySelector)
	Join

Filtering	Where(predicate)

Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>
Aggregating	Aggregate(seed, f) Aggregate(f) Min, Max, Average, Sum, Count First, Last FirstOrDefault, LastOrDefault
Concatenation	SelectMany Concat
Set operators	Distinct, Except Intersect, Union

Understanding LINQ Operators

Mapping	Select(map)
	GroupBy(keySelector)
	OrderBy(keySelector)
	Join

Filtering	Where(predicate)

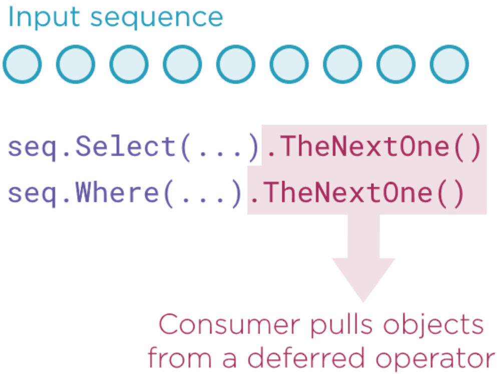
Partitioning	Take(n), TakeWhile(predicate)
	Skip(n), SkipWhile(predicate)

	OfType<T>
Aggregating	Aggregate(seed, f) Aggregate(f) Min, Max, Average, Sum, Count First, Last FirstOrDefault, LastOrDefault
Concatenation	SelectMany Concat
Set operators	Distinct, Except Intersect, Union

General-purpose operators

(very efficient)

Understanding Deferred/Lazy Evaluation



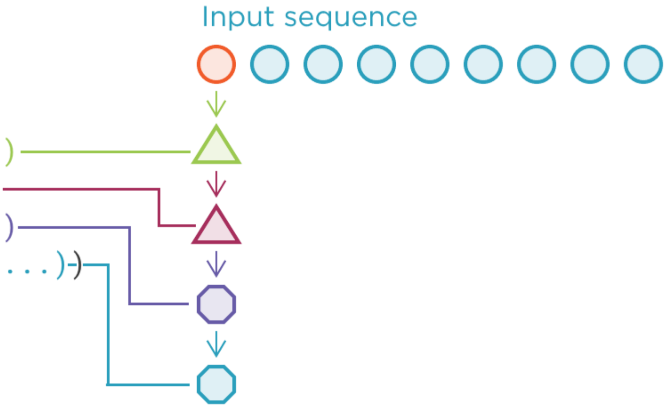
Input sequence

```
seq.Select(...).TheNextOne()
seq.Where(...).TheNextOne()
```

Consumer pulls objects from a deferred operator

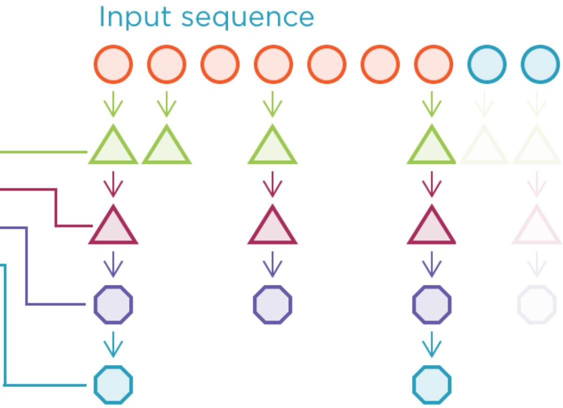
Understanding Deferred/Lazy Evaluation

```
foreach (var x in
         seq.OfType(...)
            .Where(...)
            .Select(...)
            .TakeWhile(...))
{
    Do stuff with x...
}
```

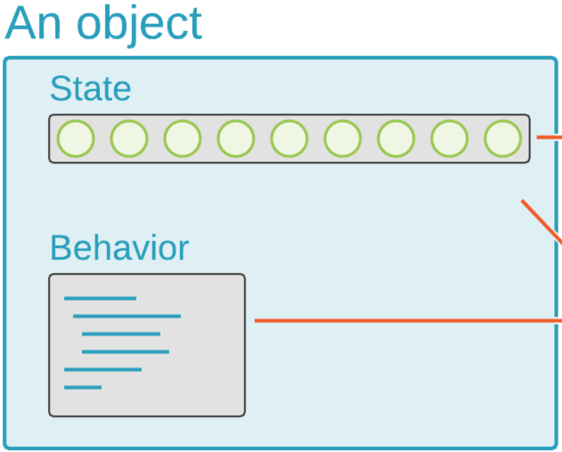


Understanding Deferred/Lazy Evaluation

```
foreach (var x in
        seq.OfType(...)
           .Where(...)
           .Select(...)
           .TakeWhile(...))
{
    Do stuff with x...
}
```



Lazy vs. Eager Evaluation

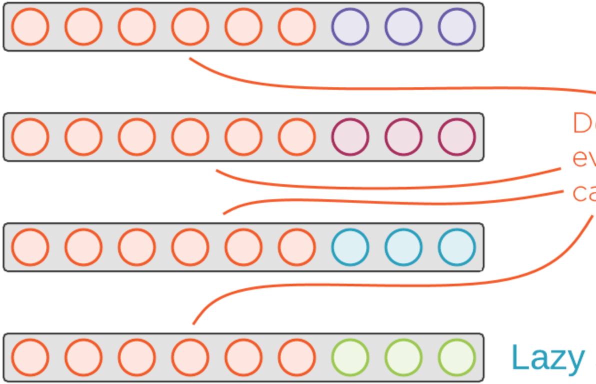


*To be **deferred***
or not to be:
that is the question!

Iterating many times over?

That will affect performance!

Lazy vs. Eager Evaluation



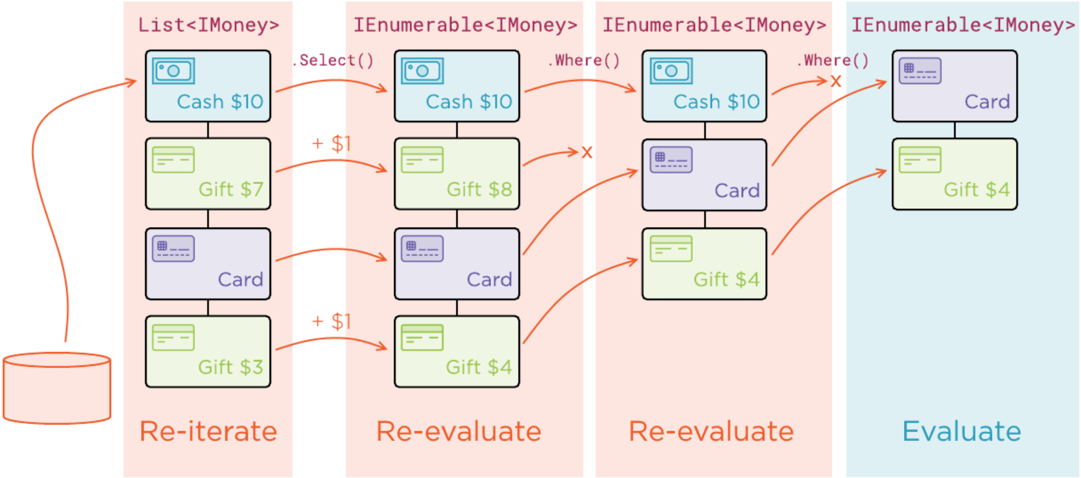
Lazy sequence

Deferred list evaluation can cost a lot — Call **seq.ToList()** on a sequence stored as state

Otherwise, object will work slow **sometimes**

Not visible to automated tests!

Inadvertently Consuming Lazy Sequences



Summary

Functional list processing

- Lean on `IEnumerable<T>` for lists
- Abstraction of a list of items
- Real collection might not even exist!

Dealing with sequences

- Focus on list mapping
- Do not think of the data structure

Summary

LINQ library

- Process lists almost as common objects
- Construct – map – map – take result

Operations on sequences

- Map to a new sequence
- Filter to a new sequence
- Aggregate to a single object
- Plus many other operators
- Mapping, filtering and aggregation operators are very efficient

Summary

Lazy evaluation on sequences

- Many LINQ operators defer evaluation
- Repeated evaluation is wasteful
- No distinction between "eager" and "lazy" `IEnumerable<T>`
- Represent eager sequence with a custom type
- Or make sure to not forget `ToList()`

Next module:

Treating Sequences as Immutable Objects