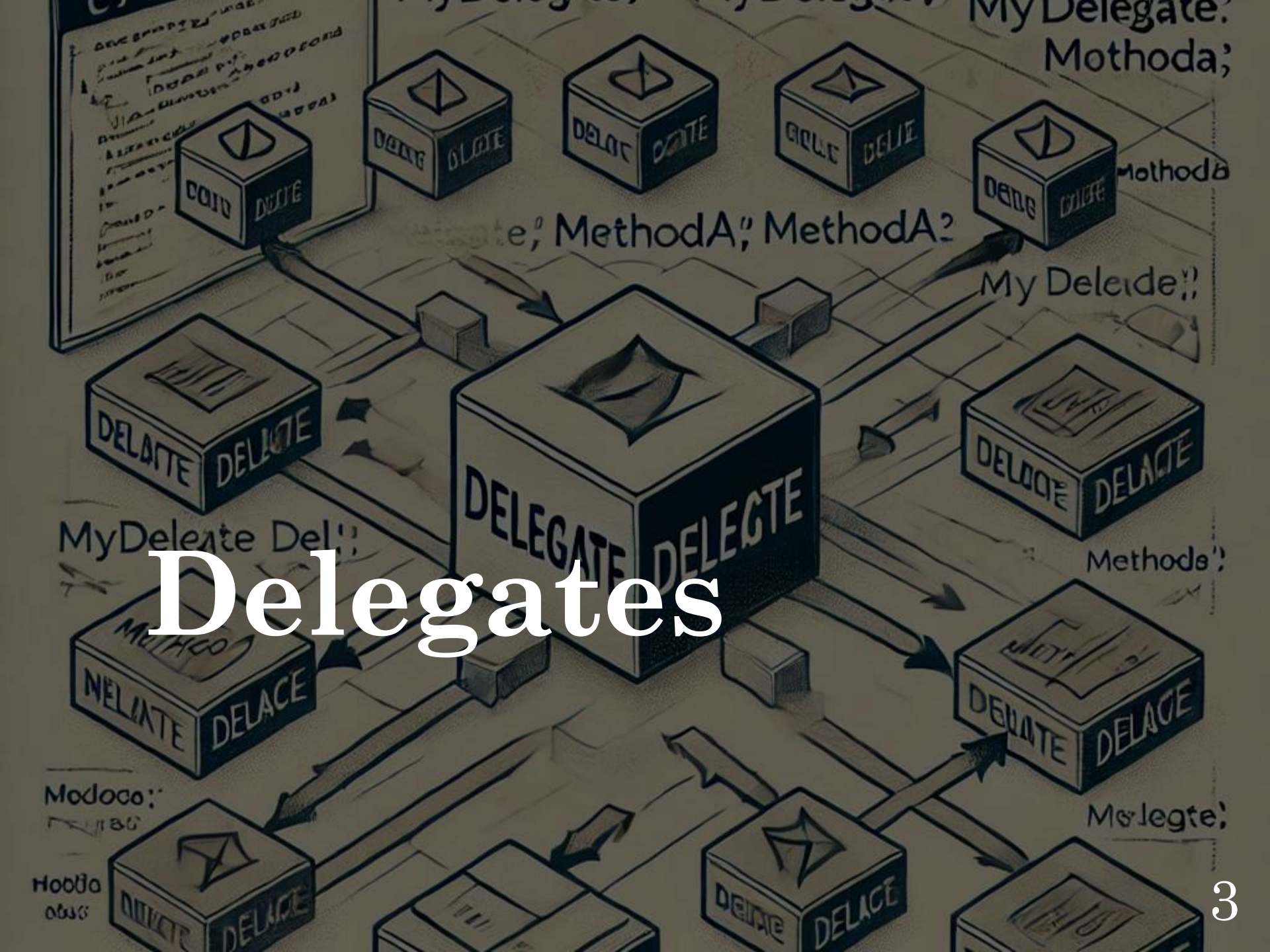
Complex IT-Systems Section 2

Henrik Bulskov

LINQ Language-Integrated Queries

- Language Features
  - Lambda expressions
  - Extensions Methods
  - Anonymous Types
  - Query Expression Syntax
  - Generics
  - yield
  - var


Delegates

Delegates

- A delegate is an object that knows how to call a method
- A *delegate type* defines the kind of method that *delegate instances* can call, i.e. the signature

```
delegate int Transformer (int x);
```

- Transformer is compatible with any method with an int return type and a single int parameter, e.g.

```
static int Square (int x) { return x * x; }
```

Delegates

- A delegate instance literally acts as a delegate for the caller

```
Transformer t = Square;
```

- Inworking the delegate, calls the target method

```
t.Invoke(3)
```
or
```
t(3)
```

- Delegate instances have multicast capability
- Using the += and -= to add or remove methods

```
SomeDelegate d = SomeMethod1;
d += SomeMethod2;
```

- Invoking d will now call both SomeMethod1 and SomeMethod2, in the order they are added.

Generic Delegates

- A delegate type may contain generic type parameters

```
public delegate T Transformer<T> (T arg);
```

- Func and Action Delegates are defined in the System namespace
- Func

```
delegate TResult Func <out TResult>                ();
delegate TResult Func <in T, out TResult>          (T arg);
delegate TResult Func <in T1, in T2, out TResult>  (T1 arg1, T2 arg2);
... and so on, up to T16
```

- Action

```
delegate void Action                ();
delegate void Action <in T>         (T arg);
delegate void Action <in T1, in T2> (T1 arg1, T2 arg2);
... and so on, up to T16
```

Lambda Expressions

- A lambda expression is an unnamed method written in place of a delegate instance

```
Transformer sqr = x => x * x;
Console.WriteLine (sqr(3));    // 9
```

- A lambda expression has the following forms

```
(parameters) => expression-or-statement-block
x => x * x;
x => { return x * x; };
```

- Lambda expressions are used most commonly with the Func and Action delegates

```
Func<int,int> sqr = x => x * x;
```

Lambda Expressions Parameter Types

- The compiler can usually infer the type of lambda parameters contextually

```
Func<int,int> sqr = x => x * x;
```

- Otherwise explicitly specify the types

```
Func<int,int> sqr = (int x) => x * x;
```

Outer Variables - Closure

- Lambda expression can reference the local variables and parameters of the method in which it's defined

```
int factor = 2;
Func<int, int> multiplier = n => n * factor;
Console.WriteLine (multiplier (3));           // 6
```

- Outer variables referenced by a lambda expression are called *captured variables*.
- A lambda expression that captures variables is called a *closure*

Outer Variables - Closure

- When you capture the iteration variable of a for loop, C# treats that variable as though it was declared *outside* the loop!!!

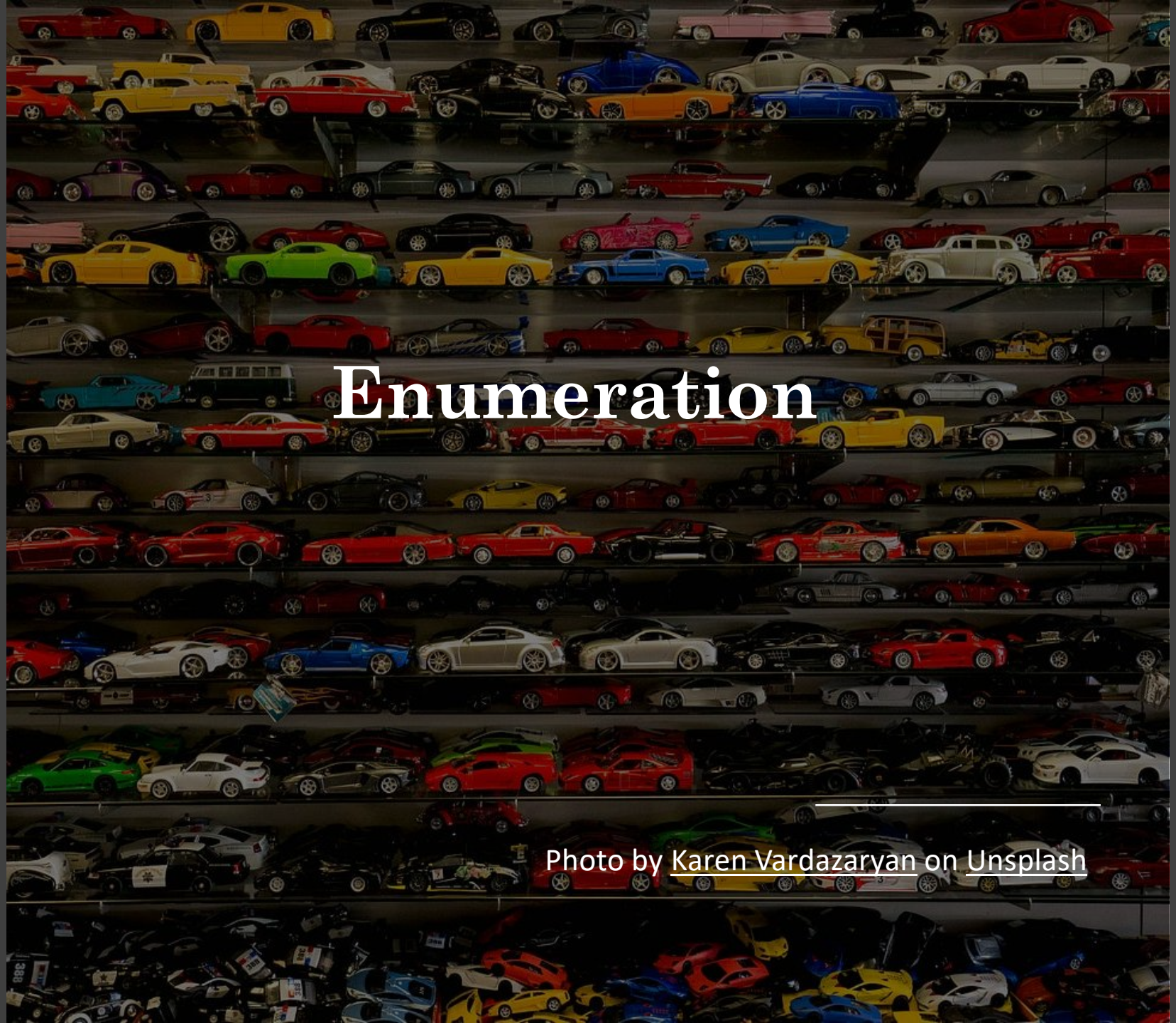
```
Action[] actions = new Action[3];

for (int i = 0; i < 3; i++)
  actions [i] = () => Console.Write (i);

foreach (Action a in actions) a();     // 333
```

Outer Variables - Closure

- When you capture the iteration variable of a for loop, C# treats that variable as though it was declared *outside* the loop!!!

```
Action[] actions = new Action[3];
for (int i = 0; i < 3; i++)
{
  int loopScopedi = i;
  actions [i] = () => Console.Write (loopScopedi);
}
foreach (Action a in actions) a();     // 012
```

- The solution, if we want to write 012, is to assign the iteration variable to a local variable that's scoped within the loop:



Enumeration

Photo by Karen Vardazaryan on Unsplash

Enumeration and Iterators

- An enumerator is a read-only, forward-only cursor over a sequence of values.
- An enumerator is an object that implements either of the following interfaces:

```
System.Collections.IEnumerator
System.Collections.Generic.IEnumerator<T>
```

- Technically, any object that has a method named MoveNext and a property called Current is treated as an enumerator
- The foreach statement iterates over *enumerable* objects

Enumeration and Iterators

- An enumerator is a read-only, forward-only cursor over a sequence of values.
- An enumerator is an object that implements either of the following interfaces:

```
System.Collections.IEnumerator
System.Collections.Generic.IEnumerator<T>
```

- Technically, any object property called Curren
- The foreach statement

```
public interface IEnumerator
{
  bool MoveNext();
  object Current { get; }
  void Reset();
}
```

Iterator

- An iterator is a method, property, or indexer that contains one or more yield statements, and return one of the four interfaces

```
System.Collections.IEnumerable
System.Collections.Generic.IEnumerable<T>
```

```
System.Collections.IEnumerator
System.Collections.Generic.IEnumerator<T>
```

- yield

```
static IEnumerable<string> Foo()
{
  yield return "One";
  yield return "Two";
  yield return "Three";
}
```

Extensions Methods

Extensions Methods

Extend any type with additional methods

LINQ provides extension methods on IEnumerable<T>

Connect these extension methods together into “pipelines”

```
public static string ReverseCase(this string s)
{
    var res = "";
    foreach (var c in s)
    {
        if (char.IsUpper(c)) res += char.ToLower(c);
        else res += char.ToUpper(c);
    }
    return res;
}

class Program
{
    0 references | 0 changes | 0 authors, 0 changes
    static void Main(string[] args)
    {

        Console.WriteLine("We Love Programming".ReverseCase());
    }
}
```

Extensions Methods - Example

```
public static string ReverseCase(this string s)
{
    var res = "";
    foreach (var c in s)
    {
        if (char.IsUpper(c)) res += char.ToLower(c);
        else res += char.ToUpper(c);
    }
    return res;
}

class Program
{
    0 references | 0 changes | 0 authors, 0 changes
    static void Main(string[] args)
    {

        Console.WriteLine("We Love Programming".ReverseCase());
    }
}
```

Extensions Methods - Example

Anonymous Types

https://unsplash.com/photos/person-hiding-on-white-curtain-gV7l2YslRS4

Anonymous Types

```
var a = new {Name = "Peter", Age = 23};

var b = new {First = 7, Last = 54, Time = DateTime.Now};

var title = "Some title";
var volume = 7;


var c = new {title, volume};

Console.WriteLine("c = " + c);
```

```
c = { title = Some title, volume = 7 }
```


Tuples

Tuples

- You can create a tuple by assigning a value to each member:

```
var letters = ("a", "b");
```

```
(string Alpha, string Beta) namedLetters = ("a", "b");
```

```
var alphabetStart = (Alpha: "a", Beta: "b");
```

- Creating a tuple is more efficient and more productive

```
private static (int Max, int Min) Range(IEnumerable<int> numbers)
```

- You can extract the individual fields by assigning to a tuple:

```
var p = new Point(3.14, 2.71);
(double X, double Y) = p;
```

https://docs.microsoft.com/en-us/dotnet/csharp/tuples

Pattern Matching

```
public static double ComputeArea(object shape)
{
    if (shape is Square)
    {
        var s = (Square)shape;
        return s.Side * s.Side;
    }
    else if (shape is Circle)
    {
        var c = (Circle)shape;
        return c.Radius * c.Radius * Math.PI;
    }
    // elided
    throw new ArgumentException(
        message: "shape is not a recognized shape",
        paramName: nameof(shape));
}
```

Pattern Matching

https://docs.microsoft.com/en-us/dotnet/csharp/pattern-matching

```
public static double ComputeAreaModernSwitch(object shape)
{
    switch (shape)
    {
        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        case Rectangle r:
            return r.Height * r.Length;
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}
```

Pattern Matching

https://docs.microsoft.com/en-us/dotnet/csharp/pattern-matching

```
public static double ComputeAreaModernIs(object shape)
{
    if (shape is Square s)
        return s.Side * s.Side;
    else if (shape is Circle c)
        return c.Radius * c.Radius * Math.PI;
    else if (shape is Rectangle r)
        return r.Height * r.Length;
    // elided
    throw new ArgumentException(
        message: "shape is not a recognized shape",
        paramName: nameof(shape));
}
```

Pattern Matching

https://docs.microsoft.com/en-us/dotnet/csharp/pattern-matching

```
public static double ComputeArea_Version3(object shape)
{
    switch (shape)
    {
        case Square s when s.Side == 0:
        case Circle c when c.Radius == 0:
            return 0;

        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}
```

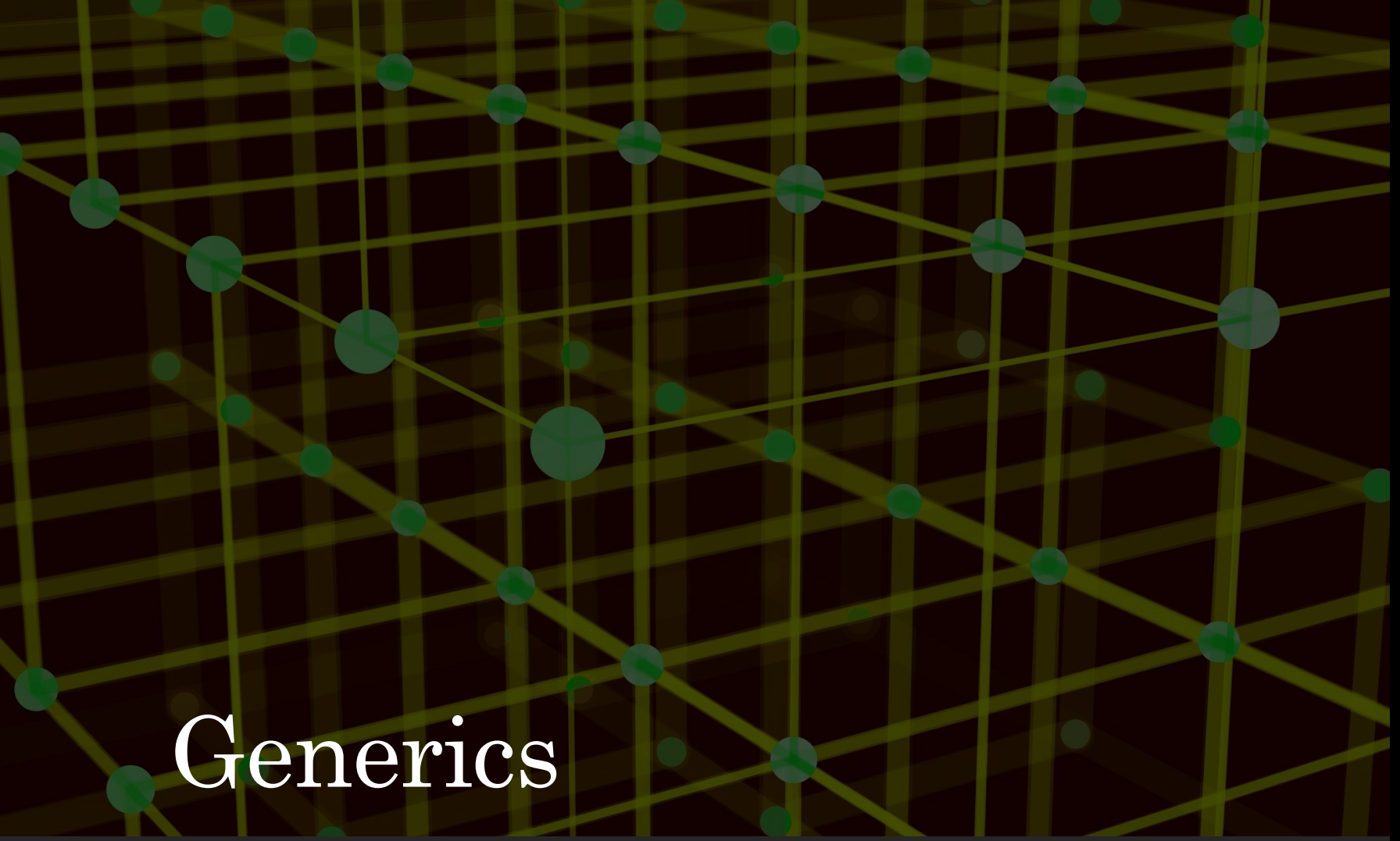
Pattern Matching

https://docs.microsoft.com/en-us/dotnet/csharp/pattern-matching

```
static object CreateShape(string shapeDescription)
{
    switch (shapeDescription)
    {
        case "circle":
            return new Circle(2);

        case "square":
            return new Square(4);

        case "large-circle":
            return new Circle(12);

        case var o when (o?.Trim().Length ?? 0) == 0:
            // white space
            return null;
        default:
            return "invalid shape description";
    }
}
```

Pattern Matching

https://docs.microsoft.com/en-us/dotnet/csharp/pattern-matching



Generics

Generics

Generics exist to write code that is reusable across different types.

We can make one structure that works with many different types, e.g.

Collections

Methods

We get well defined type definitions, instead of e.g. using the object class

Generic types

- declares type parameters—placeholder types to be filled in by the consumer of the generic type, which supplies the type arguments

```
public class Stack<T>
{
  int position;
  T[] data = new T[100];
  public void Push (T obj)  => data[position++] = obj;
  public T Pop()            => data[--position];
}
```

Generic types

- declares type parameters—placeholder types to be filled in by the consumer of the generic type, which supplies the type arguments

```
public class Stack<T>
{
  int position;
  T[] data = new T[100];
  public void Push (T obj)  => data[position++] = obj;
  public T Pop()            => data[--position];
}
```

```
var stack = new Stack<int>();
stack.Push (5);
stack.Push (10);
int x = stack.Pop();          // x is 10
int y = stack.Pop();          // y is 5
```

Use of Type Parameters

- Use it as type of fields, variables, properties, method parameters and return types
- Use it to create arrays e.g. `new T[10]`
- Call `default(T)` to get the appropriate default value
- Create a new instance with `new T()` if the new() constraint is specified
- Use methods of the interfaces or base classes in the constraint specification.
- CANNOT call static methods

```
static void Swap<T> (ref T a, ref T b)
{
   T temp = a;
   a = b;
   b = temp;
}
```

Generic Methods

```
static void Swap<T> (ref T a, ref T b)
{
   T temp = a;
   a = b;
   b = temp;
}
```

```
int x = 5;
int y = 10;
Swap (ref x, ref y);
```

Generic Methods

```
where T : base-class    // Base-class constraint
where T : interface     // Interface constraint
where T : class          // Reference-type constraint
where T : class?         // (See "Nullable reference types")
where T : struct         // Value-type constraint (excludes Nullable types)
where T : unmanaged      // Unmanaged constraint
where T : new()          // Parameterless constructor constraint
where U : T              // Naked type constraint
where T : notnull        // Non-nullable value type, or from C# 8
                         // a non-nullable reference type.
```

Generic Constraints

By default, you can substitute a type parameter with any type whatsoever.

Generic Constraints

```
class     SomeClass {}
interface Interface1 {}

class GenericClass<T,U> where T : SomeClass, Interface1
                        where U : new()
{...}
```

```
static T Max <T> (T a, T b) where T : IComparable<T>
{
  return a.CompareTo (b) > 0 ? a : b;
}
```

Subclassing Generic Types

- A generic class can be subclassed just like a nongeneric class

```
class Stack<T>                          {...}
class SpecialStack<T> : Stack<T> {...}
```

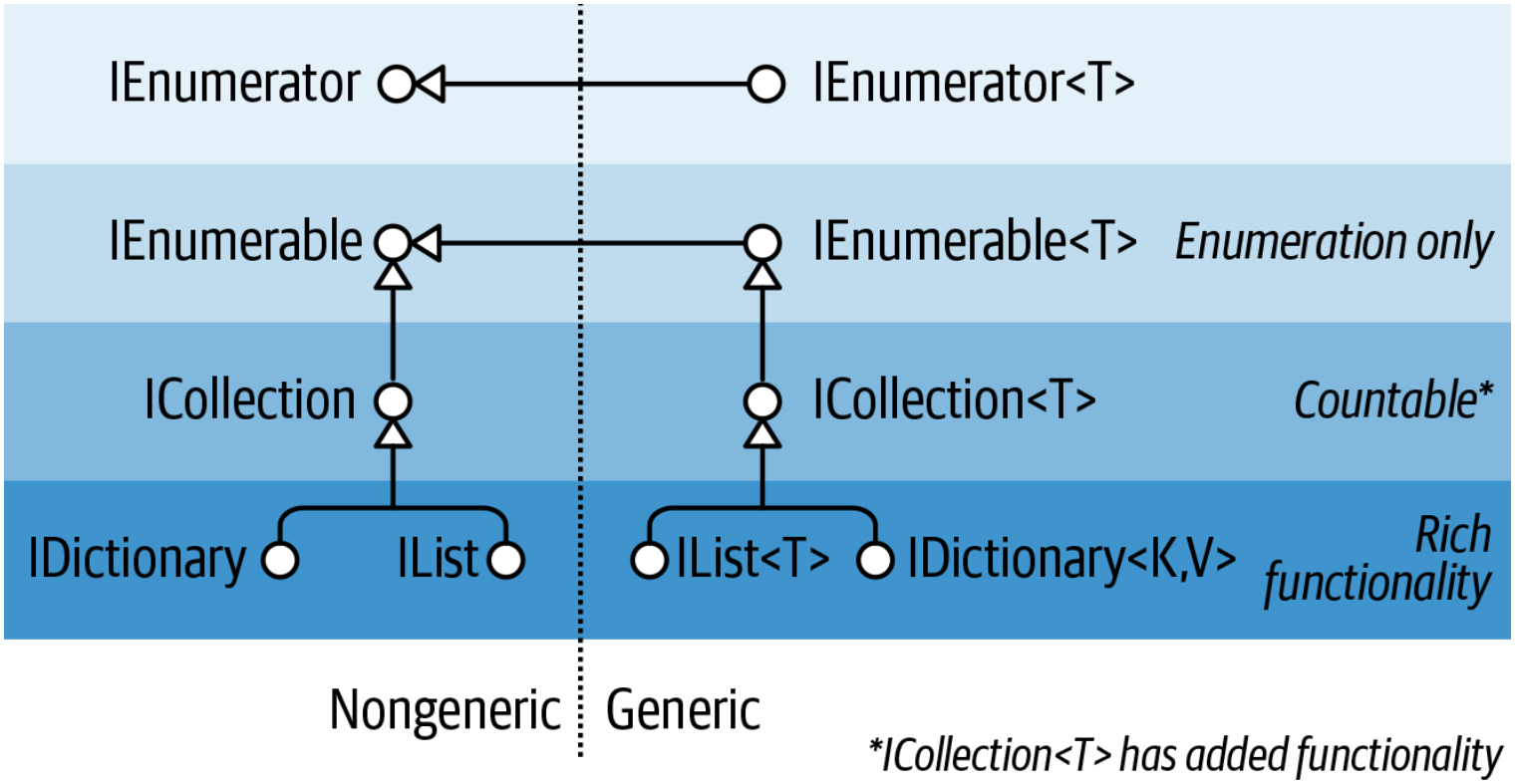
- can close the generic type parameters with a concrete type

```
class IntStack : Stack<int>  {...}
```

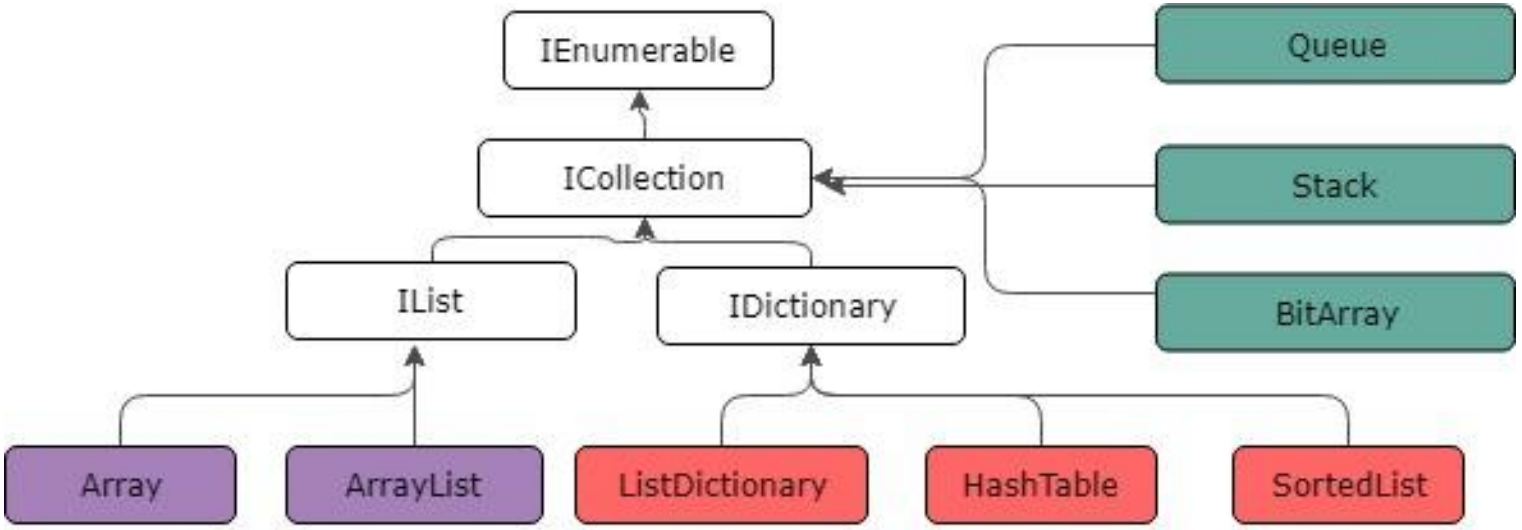
- can also introduce fresh type arguments:

```
class List<T>                              {...}
class KeyedList<T,TKey> : List<T> {...}
```

Collections



Collections



Collections

IEnumerable and IEnumerator

```
public interface IEnumerator
{
  bool MoveNext();
  object Current { get; }
  void Reset();
}
```

```
public interface IEnumerable
{
  IEnumerator GetEnumerator();
}
```

IEnumerable<T> and IEnumerator<T>

```
public interface IEnumerator<T> : IEnumerator, IDisposable
{
  T Current { get; }
}

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

```
List<int> list = new List<int>();
list.Add(3);
list.Add(5);
list.Add(6);
Console.Out.WriteLine(list[2]);// writes: 6
```

```
List<int> list = new List<int>{3,4,6};
Console.Out.WriteLine(list[2]);// writes: 6
```

List<T>

```
struct Contact
{
  public int Number;
  public string Name;
  public Contact(int number, string name){...}
  public override string ToString(){...}
}
```

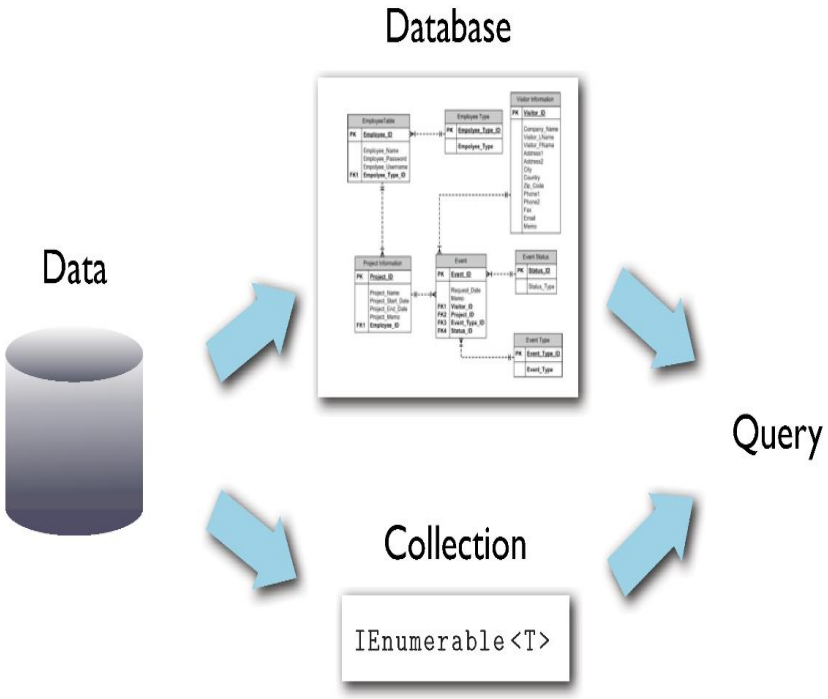
```
List<Contact> contacts = new List<Contact>{
  new Contact(123,"Tom"),
  new Contact(345,"Fred")
};
foreach(Contact c in contacts)
{
  Console.Out.WriteLine(c);
}
```

List<T>

```
Dictionary<string, int> Variable = new Dictionary<string,int>();

Variable["x_1"] = 30;
Variable["x_2"] = 60;

Console.Out.WriteLine(Variable["x_1"]+Variable["x_2"]);
```

DICTONARY<K,V>
SORTEDDICTIONARY<K,V>

LINQ



LINQ

```
IEnumerable<string> query =
  from    n in names
  where   n.Contains ("a")     // Filter elements
  orderby n.Length             // Sort elements
  select  n.ToUpper();         // Translate each element (project)
```

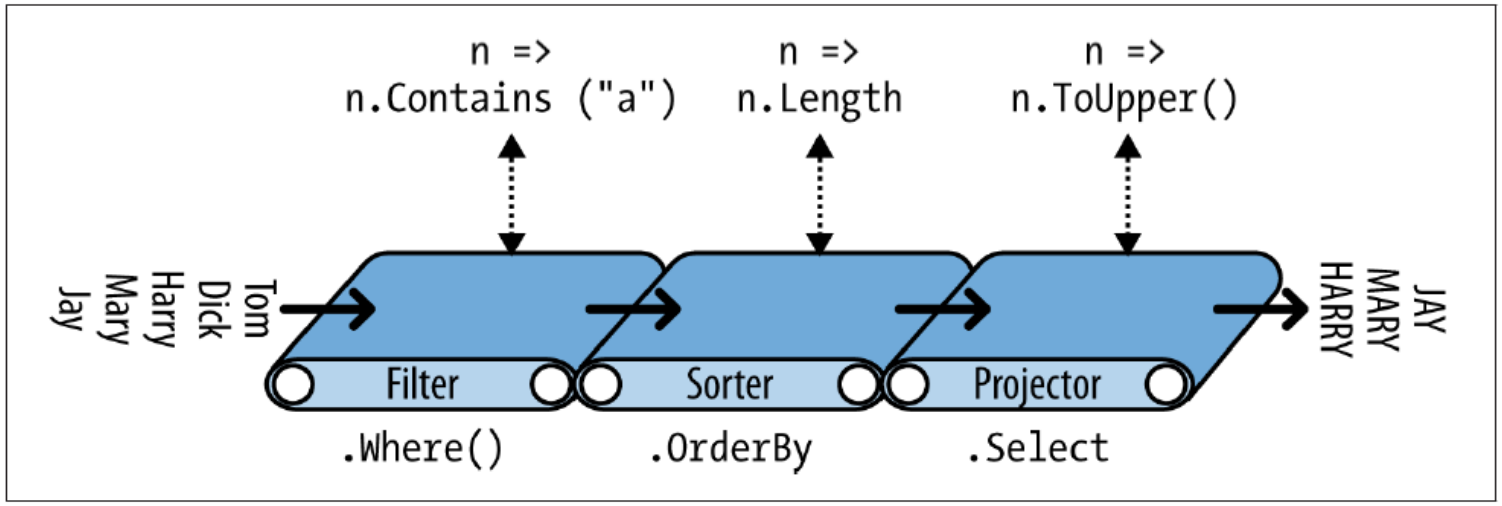
Query Expressions

C# provides a syntactic shortcut for writing LINQ queries, called query expressions

```
IEnumerable<string> query = names
  .Where   (n => n.Contains ("a"))
  .OrderBy (n => n.Length)
  .Select  (n => n.ToUpper());
```

Fluent Syntax

Chaining Query Operators



Chaining query operators

Deferred Execution

- An important feature of most query operators is that they are not executed when constructed, but when enumerated

```
var numbers = new List<int>();
numbers.Add (1);

IEnumerable<int> query = numbers.Select (n => n * 10);    // Build query

numbers.Add (2);                          // Sneak in an extra element

foreach (int n in query)
  Console.Write (n + "|");                // 10|20|
```

```
string[] musos =
  { "David Gilmour", "Roger Waters", "Rick Wright", "Nick Mason" };

IEnumerable<string> query = musos.OrderBy (m => m.Split().Last());
```

Subqueries

A subquery is a query contained within another query's lambda expression

Strategies

Composition Strategies

- Progressive Query Building

Projection Strategies

- Anonymous Types

Most Concise
Solves the problem in the fewest lines of code

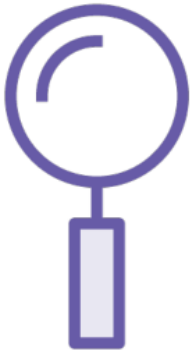
Most Readable
More code, but easier to understand what's going on

Fastest
More complicated but produces results quickly

Clean Code*

What Are We Aiming For?

* Book of Robert C. Martin – Read It!



Most Concise
Solves the problem in the fewest lines of code

Most Readable
More code, but easier to understand what's going on

Fastest
More complicated but produces results quickly

Clean Code*

What Are We Aiming For?

* Book of Robert C. Martin – Read It!

Method	Description	SQL equivalents
`Where`	Returns a subset of elements that satisfy a given condition	`WHERE`
`Take`	Returns the first `count` elements and discards the rest	`WHERE ROW_NUMBER()...` *or* `TOP` *n* subquery
`Skip`	Ignores the first `count` elements and returns the rest	`WHERE ROW_NUMBER()...` *or* `NOT IN (SELECT TOP n...)`
`TakeWhile`	Emits elements from the input sequence until the predicate is false	Exception thrown
`SkipWhile`	Ignores elements from the input sequence until the predicate is false, and then emits the rest	Exception thrown
`Distinct`	Returns a sequence that excludes duplicates	`SELECT DISTINCT...`

Filtering

Method	Description	SQL equivalents
`Select`	Transforms each input element with the given lambda expression	`SELECT`
`SelectMany`	Transforms each input element, and then flattens and concatenates the resultant subsequences	`INNER JOIN`, `LEFT OUTER JOIN`, `CROSS JOIN`

Projecting

Method	Description	SQL equivalents
`Join`	Applies a lookup strategy to match elements from two collections, emitting a flat result set	`INNER JOIN`
`GroupJoin`	As above, but emits a *hierarchical* result set	`INNER JOIN`, `LEFT OUTER JOIN`
`Zip`	Enumerates two sequences in step (like a zipper), applying a function over each element pair	

Joining

Method	Description	SQL equivalents
`OrderBy, ThenBy`	Sorts a sequence in ascending order	`ORDER BY ...`
`OrderByDescending, ThenByDescending`	Sorts a sequence in descending order	`ORDER BY ... DESC`
`Reverse`	Returns a sequence in reverse order	Exception thrown

Ordering

Method	Description	SQL equivalents
GroupBy	Groups a sequence into subsequences	GROUP BY

Grouping

Method	Description	SQL equivalents
`Concat`	Returns a concatenation of elements in each of the two sequences	`UNION ALL`
`Union`	Returns a concatenation of elements in each of the two sequences, excluding duplicates	`UNION`
`Intersect`	Returns elements present in both sequences	`WHERE ... IN (...)`

`Except`	Returns elements present in the first, but not the second sequence	`EXCEPT`
		or
		`WHERE ... NOT IN (...)`

Set Operators

Method	Description
OfType	Converts IEnumerable to IEnumerable<T>, discarding wrongly typed elements
Cast	Converts IEnumerable to IEnumerable<T>, throwing an exception if there are any wrongly typed elements
ToArray	Converts IEnumerable<T> to T[]
ToList	Converts IEnumerable<T> to List<T>
ToDictionary	Converts IEnumerable<T> to Dictionary<TKey,TValue>
ToLookup	Converts IEnumerable<T> to ILookup<TKey,TElement>
AsEnumerable	Downcasts to IEnumerable<T>
AsQueryable	Casts or converts to IQueryable<T>

Conversion Methods

Method	Description	SQL equivalents
`First, FirstOrDefault`	Returns the first element in the sequence, optionally satisfying a predicate	`SELECT TOP 1 ... ORDER BY ...`
`Last,` `LastOrDefault`	Returns the last element in the sequence, optionally satisfying a predicate	`SELECT TOP 1 ... ORDER BY ... DESC`
`Single, SingleOrDefault`	Equivalent to `First/FirstOrDefault`, but throws an exception if there is more than one match	
`ElementAt, ElementAtOrDefault`	Returns the element at the specified position	Exception thrown
`DefaultIfEmpty`	Returns a single-element sequence whose value is `default(TSource)` if the sequence has no elements	`OUTER JOIN`

Element Operators

Method	Description	SQL equivalents
`Count, LongCount`	Returns the number of elements in the input sequence, optionally satisfying a predicate	`COUNT (...)`
`Min, Max`	Returns the smallest or largest element in the sequence	`MIN (...), MAX (...)`
`Sum, Average`	Calculates a numeric sum or average over elements in the sequence	`SUM (...), AVG (...)`
`Aggregate`	Performs a custom aggregation	Exception thrown

Aggregation Methods

Method	Description	SQL equivalents
`Contains`	Returns `true` if the input sequence contains the given element	`WHERE ... IN (...)`
`Any`	Returns `true` if any elements satisfy the given predicate	`WHERE ... IN (...)`
`All`	Returns `true` if all elements satisfy the given predicate	`WHERE (...)`
`SequenceEqual`	Returns `true` if the second sequence has identical elements to the input sequence	

Quantifiers

Method	Description
`Empty`	Creates an empty sequence
`Repeat`	Creates a sequence of repeating elements
`Range`	Creates a sequence of integers

Generation Methods