Query Operators

Putting LINQ to Work

Overview

- Filtering
- Projecting
- Joining
- Ordering
- Grouping
- Conversions
- Sets
- Aggregation
- Quantifiers
- Generation
- Elements

What Is A Standard Operator?

- **Operators are extension methods defined in the System.Linq namespace**
  - Attached to the static Enumerable and Queryable classes
- **Operate on IEnumerable<T> and IQueryable<T>**
- **Two categories of operators**
  - Most operators defer execution
  - Some operators require immediate execution
- **Operators using deferred execution fall into two categories**
  - Streaming
  - Non-streaming
- **Some operators have dedicated keyword (`Where`)**

Filtering

Method	Description
Where	Filter values by a predicate function (where)
OfType	Filter values based on their ability to be coerced to a type (can use on IEnumerable)

```
ArrayList list = new ArrayList();
list.Add("Dash");
list.Add(new object());
list.Add("Skitty");
list.Add(new object());

// selects the two strings
var query =
     from name in list.OfType<string>()
     select name;
```

Sorting

Method	Description
OrderBy OrderByDescending	Sort values in ascending or descending order (orderby)
ThenBy / ThenByDescending	A secondary sort
Reverse	Reverse the order of elements

```
string[] names = { "Bob", "Alice", "Alex", "Carol" };

var query =
    names.OrderBy(s => s)
         .ThenBy(s => s.Length);
```

```
query =
    from name in names
    orderby name, name.Length
    select name;
```

Ordered Sequences

- **Most standard operators that return a sequence return IEnumerable<T> or IQueryable<T>**
  - OrderBy and ThenBy return IOrderedEnumerable<T> and IOrderedQueryable<T>
  - ThenBy is an extension method for an ordered enumerable

```
var query =
    names.OrderBy(s => s)
         .ThenBy(s => s.Length);

// error (query is IOrderedEnumerable<string>)
//       (where returns IEnumerable<T>)
query = names.Where(s => s.Length > 3);
```

Set Operations

Method	Description
Distinct	Remove duplicate values
Except	Returns the differences of two sequences
Intersect	Returns the intersection of two sequences
Union	Returns unique elements from both sequences

```
int[] twos = { 2, 4, 6, 8, 10 };
int[] threes = { 3, 6, 9, 12, 15 };

// 6
var intersection = twos.Intersect(threes);

// 2, 4, 8, 10
var except = twos.Except(threes);

// 2, 4, 6, 8, 10, 3, 9, 12, 15
var union = twos.Union(threes);
```

Equality In LINQ to Objects

- **Operators that test equality use default IEqualityComparer**
  - Will accept a custom comparer
- **Anonymous types generated by C# compiler are special**
  - Override Equals and GetHashCode
  - Uses all public properties on type to test for equality

```
var employees = new List<Employee> {
  new Employee() { ID=1, Name="Scott" },
  new Employee() { ID=2, Name="Poonam" },
  new Employee() { ID=1, Name="Scott"}
};
```

```
// yields a sequence of 3 employees
var employees =
      (from employee in employees
       select employee).Distinct();
```

```
// yields a sequence of 2 employees
var query = (from employee in employees
                  select new { employee.ID,
                               employee.Name })
                  .Distinct();
```

Quantifiers

Method	Description
All	Tests if all elements satisfy a condition
Any	Tests if any elements satisfy a condition
Contains	Tests if the sequence contains a specific element

```
int[] twos = { 2, 4, 6, 8, 10 };

// true
bool areAllevenNumbers = twos.All(i => i % 2 == 0);

// true
bool containsMultipleOfThree = twos.Any(i => i % 3 == 0);

// false
bool hasSeven = twos.Contains(7);
```

Projection Operators

Method	Description
Select	Projects values in a sequence based on a transformation function
SelectMany	Flattens and projects across multiple sequences

```
string[] famousQuotes =
{
    "Advertising is legalized lying",
    "Advertising is the greatest art form of the twentieth century"
};

var query =
        (from sentence in famousQuotes
         from word in sentence.Split(' ')
         select word).Distinct();
```

```
Advertising
is
legalized
lying
the
greatest
Art
form
of
twentieth
century
```

pluralsight
see what you can learn

SelectMany

- **Select returns one element for each input element**
- **SelectMany can return multiple elements for each input**
  - Think of SelectMany as a sub-iterator
  - Triggered with additional from clauses in a query

```
var query =
    famousQuotes.SelectMany(s => s.Split(' '))
                .Distinct();
```

Partitioning

Method	Description
Skip / SkipWhile	Skip elements until a condition or predicate is met
Take / TakeWhile	Take elements until a condition or predicate is met

```
// yields 5, 7
var query = numbers.Skip(2).Take(2);
```

```
// yields 5, 7, 9
var query = numbers.SkipWhile(n => n < 5)
                   .TakeWhile(n => n < 10);
```

Joining

Method	Description
Join	Join two sequences on a key and yields a sequence (flat result)
GroupJoin	Join two sequences on a key and yields groups of sequences (hierarchical result)

```
var query = employees.Join(
                           departments,             // inner sequence
                           e => e.DepartmentID,     // outer key selector
                           d => d.ID,               // inner key selector
                       (e, d) => new {              // result projector
                           EmployeeName = e.Name,
                           DeparmentName = d.Name
                       });
```

Comparisons With SQL

- **LINQ Join operator is an inner join**
  - Only outputs an element when a match is present
  - Only allows equijoins
- **GroupJoin can offer outer join capabilities**
  - Can return an outer element with no matching inner elements
  - Trigger by an into clause in query syntax
  - Use a SelectMany to flatten (additional from clause)

Grouping

Method	Description
GroupBy	Group elements from a sequence
ToLookup	Insert elements into a one to many dictionary

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

var query = numbers.GroupBy(i => i % 2);

foreach (var group in query)
{
    Console.WriteLine("Key: {0}", group.Key);
    foreach (var number in group)
    {
        Console.WriteLine(number);
    }
}
```

IGrouping Interface

- **GroupBy and ToLookup return a sequence of objects**
  - Object’s implement IGrouping<K, V> interface
- **Similar to a Dictionary<K, V>**
  - Contains a sequence instead of individual items
  - Each grouping contains a Key property

```
foreach (var group in query)
{
    Console.WriteLine("Key: {0}", group.Key);
    foreach (var number in gro
    {
        Console.WriteLine(numb
    }
}
```

```
foreach (IGrouping<int, int> group in query)
{
    Console.WriteLine("Key: {0}", group.Key);
    foreach (int number in group)
    {
        Console.WriteLine(number);
    }
}
```

Lookups

- **Lookup<K,V> is the data structure behind groupings**
  - An immutable dictionary of sequences
- **GroupBy execution is deferred**
- **ToLookup execution is immediate**

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

var query = numbers.GroupBy(i => i % 2);
```

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

var query = numbers.ToLookup(i => i % 2);
```

Generation Operations

Method	Description
Empty	Returns an empty collection
Range	Generates a sequence of numbers
Repeat	Generates a collection of repeated values
DefaultIfEmpty	Replaces empty collection with collection of 1 default value

```
var query =
     from department in departments
     join employee in employees
      on department.ID equals employee.DepartmentID
      into employeeGroup
     from eg in employeeGroup.DefaultIfEmpty()
     select new { department.Name,
                  Employee = eg == null ? "" : eg.Name };
```

Equality

Method	Description
SequenceEqual	Compares elements in two sequences

```
Employee e1 = new Employee() { ID = 1 };
Employee e2 = new Employee() { ID = 2 };
Employee e3 = new Employee() { ID = 3 };

var employees1 = new List<Employee>() { e1, e2, e3 };
var employees2 = new List<Employee>() { e3, e2, e1 };

bool result = employees1.SequenceEqual(employees2);
```

Element Operations

Method	Description
ElementAt / ElementAtOrDefault	Returns the element at a specified index
First / FirstOrDefault	Returns the first element of a collection
Last / LastOrDefault	Returns the last element of a collection
Single / SingleOrDefault	Returns a single element

```
string[] empty = { };
string[] notEmpty = { "Hello", "World" };

var result = empty.FirstOrDefault(); // null
result = notEmpty.Last();            // World
result = notEmpty.ElementAt(1);      // World
result = empty.First();              // InvalidOperationException
result = notEmpty.Single();          // InvalidOperationException
result = notEmpty.First(s => s.StartsWith("W"));
```

pluralsight
see what you can learn

Conversions

Method	Description
AsEnumerable	Returns input as IEnumerable<T>
AsQueryable	Converts IEnumerable<T> to IQueryable<T>
Cast	Coerce all elements to a type
OfType	Filters values that can be coerced to a type
ToArray	Converts sequence to an array (immediate)
ToDictionary	Convert sequence to Dictionary<K, V>
ToList	Converts sequence to List<T>
ToLookup	Group elements into an IGrouping<K, V>

Conversion Tips

- Use the To operators (ToArray, ToList) to force execution
- Use OfType and Cast to convert non-generic collections to IEnumerable<T>
- Use AsQueryable to simulate a remote LINQ provider
- Use AsEnumerable to move query processing local

```
var employees = new List<Employee> {
    new Employee { ID=1, Name="Scott", DepartmentID=1 },
    new Employee { ID=2, Name="Poonam", DepartmentID=1 },
    new Employee { ID=3, Name="Andy", DepartmentID=2}
};

Dictionary<int, Employee> employeeDictionary =
    employees.ToDictionary(e => e.ID, // key selector
                           e => e);   // value selector
```

Concatenation

Method	Description
Concat	Concatenates two sequences into a single sequence

```
string[] firstNames = { "Scott", "James", "Allen", "Greg" };
string[] lastNames = { "James", "Allen", "Scott", "Smith" };

var concatNames = firstNames.Concat(lastNames).OrderBy(s => s);
var unionNames = firstNames.Union(lastNames).OrderBy(s => s);
```

```
Allen
Allen
Greg
James
James
Scott
Scott
Smith
```

```
Allen
Greg
James
Scott
Smith
```

Aggregation

Method	Description
Aggregate	Computes a custom aggregation on a sequence
Average	Calculates the average value in a sequence
Count / LongCount	Counts the elements in a sequence, overload accepts a predicate
Max	Returns the maximum value in a sequence
Min	Returns the minimum value in a sequence
Sum	Calculates the sum of values in a sequence

Using Aggregation

```
Process[] runningProcesses = Process.GetProcesses();

 var summary = new
 {
     ProcessCount = runningProcesses.Count(),
     WorkerProcessCount = runningProcesses.Count(
                                   p => p.ProcessName == "w3wp"),
     TotalThreads = runningProcesses.Sum(p => p.Threads.Count),
     MinThreads = runningProcesses.Min(p => p.Threads.Count),
     MaxThreads = runningProcesses.Max(p => p.Threads.Count),
     AvgThreads = runningProcesses.Average(p => p.Threads.Count)
 };
```

Summary

- **Standard operators are the methods that define LINQ’s abiltites**
- **Two types of operators**
  - Immediate execution
  - Deferred execution
    - Streaming
    - Non-streaming
- **Operators defined on IEnumerable<T> and IQueryable<T>**