LINQ

LINQ , or Language-Integrated Query, is a set of language and framework features for writing structured type-safe queries over local object collections and remote data sources

Fluent syntax

Fluent syntax is the most flexible and fundamental.

We describe how to chain query operators to form more complex queries.

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

Query Syntax

Query expression is not a means of embedding SQL into C#.

In fact, the design of query expressions was inspired primarily by list comprehensions from functional programming languages.

```
{
        IEnumerable<string> query =       from   n  in  names  where
                                                  n.Contains ("a")
                                                  n.Length
                                                  n.ToUpper();
}
```

Chaining Decorator

When query operators are chained as in this example, the output sequence of one operator is the input sequence of the next. The complete query resembles a production line of conveyor belts

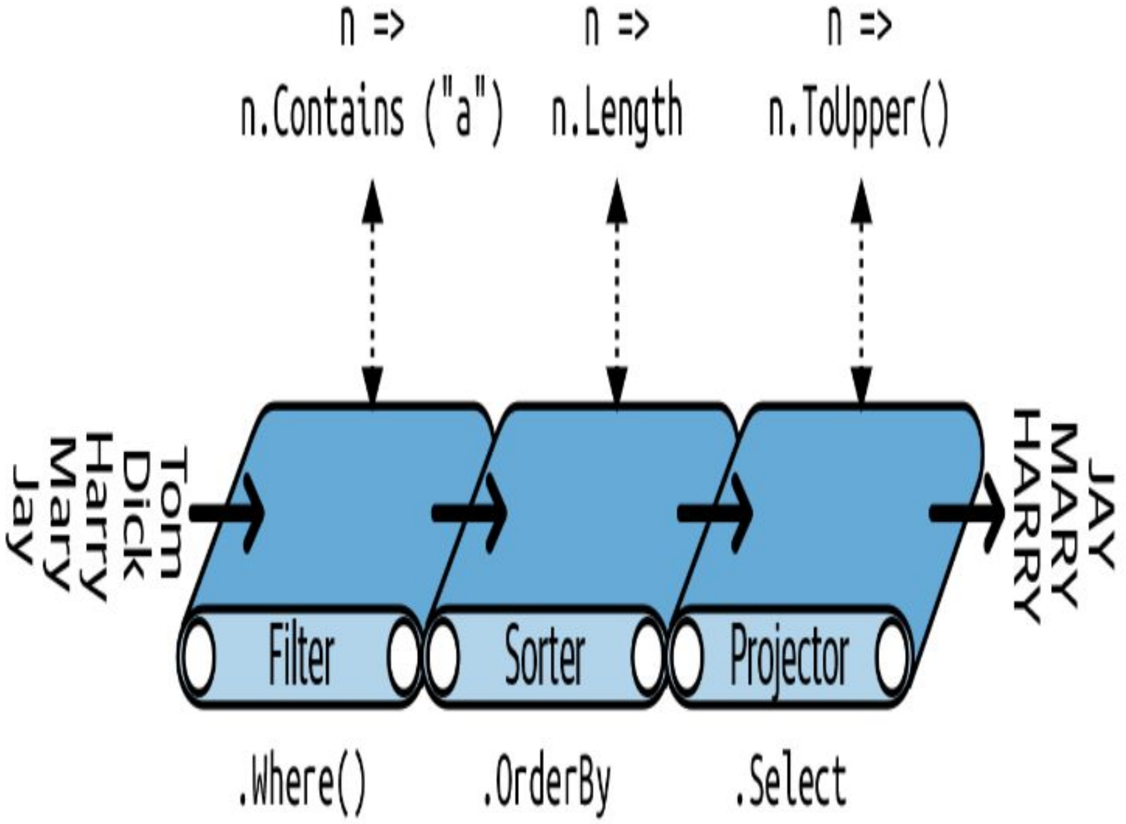

Figure 8-1. Chaining query operators

Deferred Execution

An important feature of most query operators is that they execute not when constructed,

but when enumerated

.

```
var numbers = new List<int> { 1 };

var query = numbers.Select (n => n * 10);

numbers.Add (2);

foreach (int n in query)

            Console.Write (n + "|");   // 10|20|
```

Interpreted Queries

LINQ provides two parallel architectures:

1- local queries for local object collections .
2- interpreted queries for remote data sources .

To write interpreted queries, you need to start with an API that exposes sequences of type `IQueryable<T>`.

An example is Microsoft’s Entity Framework Core (EF Core),

LINQ OPERATIONS

standard query operators fall into three categories .

1- Sequence in, sequence out (sequence→sequence)

2- Sequence in, single element or scalar value out

3- Nothing in, sequence out (generation methods)

Filtering

`Where` : Returns a subset of elements that satisfy a given condition.

`Take` : Returns the first `count` elements and discards the rest.

`Skip` : Ignores the first `count` elements and returns the rest.

`TakeWhile` : Emits elements from the input sequence until the predicate is false

`SkipWhile` : Ignores elements from the input sequence until the predicate is false, and then emits the rest

`Distinct` : Returns a sequence that excludes duplicates

Projecting

Select : Transforms each input element with the given lambda expression

SelectMany : Transforms each input element, and then flattens and concatenates the resultant subsequences

```
{
IEnumerable<string> query = from f in FontFamily.Families
                            select f.Name;
}
```

Joining

`Join` : Applies a lookup strategy to match elements from twocollections, emitting a flat result set

`GroupJoin` : Similar to `Join`, but emits a hierarchical result set

`Zip` : Enumerates two sequences in step (like a zipper), applying afunction over each element pair

Grouping

GroupBy : Groups a sequence into subsequences

```
{
string[ ] files = Directory.GetFiles(Path.GetTempPath());

IEnumerable<IGrouping<string,string>> query =
                        files.GroupBy (file => Path.GetExtension (file));
}
```