LINQ

Succinctly

by Jason Roberts

Syncfusion | Technology Resource Portal

LINQ Succinctly

By

Jason Roberts

Foreword by Daniel Jebaraj

Table of Contents

Important licensing information. Please read.

Technical Reviewer: Stephen Haunts
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Darren West, content producer, Syncfusion, Inc.

About the Author

Jason Roberts is a Microsoft C# MVP with over 13 years of commercial experience. He is a writer, blogger, open source contributor, and Pluralsight course author, and he holds a Bachelor of Science degree in computing.

You can find him on Twitter as @robertsjason and at his blog at http://dontcodetired.com/blog/.

Chapter 1 LINQ Fundamentals

Why LINQ?

At a high level, there are number of benefits that may be gained by using LINQ (Language Integrated Query), including:

- Reduce the amount of code that needs writing.
- A better understanding of the intent of what the code is doing.
- Once learned, can use a similar set of LINQ query knowledge against different data sources (in-memory objects, or remote sources such as SQL Server or even Twitter.)
- Queries can be composed together and built up in stages.
- Queries offer the safety of compile-time type checking.

Essentially, LINQ enables queries to be treated as first-class citizens in C# and Visual Basic.

The building blocks of LINQ

The two fundamental building blocks of LINQ are the concepts of **elements** and **sequences**.

A sequence can be thought of as a list of items, with each item in the list being an element. A sequence is an instance of a class that implements the **`IEnumerable<T>`** interface.

If an array of numbers were declared as: **`int[] fibonacci = {0, 1, 1, 2, 3, 5};`** the variable **`fibonacci`** represents a sequence with each **`int`** in the array being an individual element.

A sequence could be a **local sequence** of in-memory objects or a **remote sequence**, like a SQL Server database. In the case of remote data sources (for example SQL Server), these remote sequences also implement the **`IQueryable<T>`** interface.

Queries, or more specifically **query operators**, work on an **input sequence** and produce some output value. This output value could be a transformed version of the input sequence, i.e. an **output sequence** or single scalar value such as a count of the number of elements in the input sequence.

Queries that run on local sequences are known as **local queries** or **LINQ-to-objects** queries.

There are a whole host of query operators that are implemented as extension methods in the static **`System.Linq.Enumerable`** class. This set of query operators are known as the **standard query operators**. These query operators will be covered in depth in the LINQ Query Operators chapter later in this book.

One important thing to note when using query operators is that they do not alter the input sequence; rather, the query operator will return a new sequence (or a scalar value).

Scalar return values and output sequences

A query operator returns an output sequence or a scalar value. In the following code, the input sequence **`fibonacci`** is operated on first by the **`Count`** query operator that produces a single scalar value representing the number of elements in the input sequence. Next, the same input sequence is operated on by the **`Distinct`** query operator that produces a new output sequence containing the elements from the input sequence, but with duplicate elements removed.

```
int[] fibonacci = { 0, 1, 1, 2, 3, 5 };

// Scalar return value
int numberOfElements = fibonacci.Count();

Console.WriteLine("Count: {0}", numberOfElements);

// Output sequence return value
IEnumerable<int> distinctNumbers = fibonacci.Distinct();

Console.WriteLine("Elements in output sequence:");

foreach (var number in distinctNumbers)
{
    Console.WriteLine(number);

}
```

The output from this code produces the following:

```
Count: 6
Elements in output sequence:
0
1
2
3
5
```

Deferred execution

The majority of query operators do not execute immediately; their execution is deferred to a later time in the program execution. This means that the query does not execute when it is created, but when it is used or enumerated.

Deferred execution means that the input sequence can be modified after the query is constructed, but before the query is executed. It is only when the query is executed that the input sequence is processed.

In the following code, the input sequence is modified after the query is constructed, but before it is executed.

```
int[] fibonacci = { 0, 1, 1, 2, 3, 5 };

// Construct the query
IEnumerable<int> numbersGreaterThanTwoQuery = fibonacci.Where(x => x > 2);

// At this point the query has been created but not executed

// Change the first element of the input sequence
fibonacci[0] = 99;

// Cause the query to be executed (enumerated)
foreach (var number in numbersGreaterThanTwoQuery)
{
    Console.WriteLine(number);

}
```

The query is not actually executed until the **`foreach`** is executed. The results of running this code are as follows:

```
99
3
5
```

Notice that the value 99 has been included in the results, even though at the time the query was constructed the first element was still the original value of 0.

With the exception of those that return a scalar value (or a single element from the input sequence) such as **`Count`**, **`Min`**, and **`Last`**; the standard query operators work in this deferred execution way. So using operators such as **`Count`** will cause the query to be executed immediately, and not deferred.

There are a number of **conversion operators** that also cause immediate query execution, such as **`ToList`**, **`ToArray`**, **`ToLookup`**, and **`ToDictionary`**.

In the following code, the **fibonacci** array is being modified as in the preceding example code, but in this example, by the time the modification takes place (**fibonacci[0] = 99**), the query has already been executed because of the additional **ToArray()**.

```
int[] fibonacci = { 0, 1, 1, 2, 3, 5 };

// Construct the query
IEnumerable<int> numbersGreaterThanTwoQuery = fibonacci.Where(x => x > 2)
                                                       .ToArray();

// At this point the query has been executed because of the .ToArray()

// Change the first element of the input sequence
fibonacci[0] = 99;

// Enumerate the results
foreach (var number in numbersGreaterThanTwoQuery)
{
    Console.WriteLine(number);

}
```

The results of running this code are:

```
3
5
```

Notice here the value 99 is no longer present because the input sequence is being modified after the query has executed.

Lambda expressions in query operators

Some query operators allow custom logic to be supplied. This custom logic can be supplied to the query operator by way of a **lambda expression**.

The **fibonacci.Where(x => x > 2)** code in the preceding code sample is an example of a query operator being supplied some custom logic. Here the lambda expression **x => x > 2** will only return elements (**int**s in this case) that are greater than 2.

When a query operator takes a lambda expression, it will apply the logic in the lambda expression to each individual element in the input sequence.

The type of lambda expression that is supplied to a query operator depends on what task the query operator is performing. In Figure 1, we can see the signature of the **Where** query operator; here the input element **int** is provided and a **bool** needs to be returned that determines of the element will be included in the output sequence.

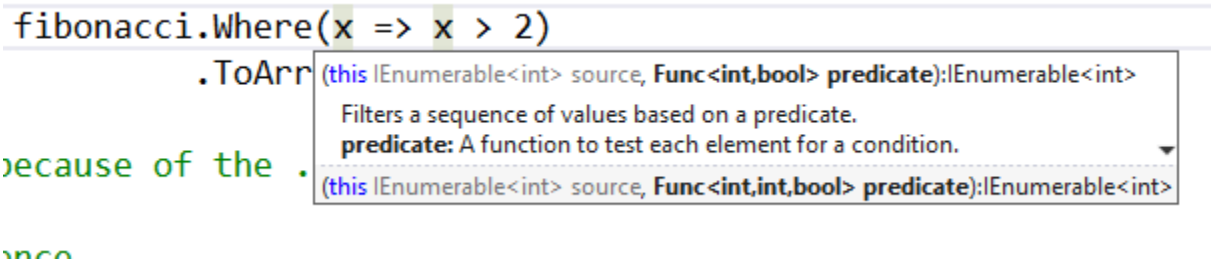


Figure 1: Visual Studio showing the signature of the Where operator

As an alternative to using a lambda expression, it is also possible to use a traditional delegate that points to a method.

Local and interpreted queries

LINQ provides for two distinct architectures: **local** and **interpreted**.

Local queries operate on **`IEnumerable<T>`** sequences and are compiled into the resulting assembly at compile time. Local queries, as the name suggests, can be thought of as operating on sequences local to the machine on which the query is executing (for example, querying an in-memory list of objects).

Interpreted queries are interpreted at runtime and work on sequences that can come from a remote source such as an SQL Server database. Interpreted queries operate on **`IQueryable<T>`** sequences.

Interpreted queries will be discussed later in Chapter 5, “Interpreted Queries.”

The following code demonstrates the use of the **`Attribute(XName)`** method to locate a specific **`XAttribute`** and get its value by reading its **`Value`** property. This example also shows the use of the **`FirstAttribute`** method to get an element’s first declared attribute, and also how to combine standard query operators such as **`Skip`** to query the **`IEnumerable<XAttribute>`** provided by the **`Attributes`** method.

```
var xml = @"
<ingredients>
    <ingredient name='milk' quantity='200' price='2.99'  />
    <ingredient name='sugar' quantity='100' price='4.99'  />
    <ingredient name='safron' quantity='1' price='46.77'  />
</ingredients>";

XElement xmlData = XElement.Parse(xml);

XElement milk =
    xmlData.Descendants("ingredient")
           .First(x => x.Attribute("name").Value == "milk");
```

```
XAttribute nameAttribute = milk.FirstAttribute; // name attribute

XAttribute priceAttribute = milk.Attribute("price");

string priceOfMilk = priceAttribute.Value; // 2.99

XAttribute quantity = milk.Attributes()
                          .Skip(1)
                          .First(); // quantity attribute
```

Chapter 2 Fluent and Query Expression Styles

There are two styles of writing LINQ queries: the **fluent style** (or **fluent syntax**) and the **query expression style** (or **query syntax**).

The fluent style uses query operator extension methods to create queries, while the query expression style uses a different syntax that the compiler will translate into fluent syntax.

Fluent style

The code samples up until this point have all used the fluent syntax.

The fluent syntax makes use of the query operator extension methods as defined in the static **`System.Linq.Enumerable`** class (or **`System.Linq.Queryable`** for interpreted or **`System.Linq.ParallelEnumerable`** for PLINQ queries). These extension methods add additional methods to instances of **`IEnumerable<TSource>`**. This means that any instance of a class that implements this interface can use these fluent LINQ extension methods.

Query operators can be used singularly, or chained together to create more complex queries.

Chained query operators

If the following class has been defined:

```
class Ingredient
{
    public string Name { get; set; }
    public int Calories { get; set; }
}
```

The following code will use three chained query operators: **`Where`**, **`OrderBy`**, and **`Select`**.

```
Ingredient[] ingredients =
{
    new Ingredient{Name = "Sugar", Calories=500},
    new Ingredient{Name = "Egg", Calories=100},
    new Ingredient{Name = "Milk", Calories=150},
    new Ingredient{Name = "Flour", Calories=50},
    new Ingredient{Name = "Butter", Calories=200}
};
```

```
IEnumerable<string> highCalorieIngredientNamesQuery =
                            ingredients.Where(x => x.Calories >= 150)
                                       .OrderBy(x => x.Name)
                                       .Select(x => x.Name);

foreach (var ingredientName in highCalorieIngredientNamesQuery)
{
    Console.WriteLine(ingredientName);
}
```

Executing this code produces the following output:

```
Butter
Milk
Sugar
```

Figure 2 shows a graphical representation of this chain of query operators. Each operator works on the sequence provided by the preceding query operator. Notice that the initial input sequence (represented by the variable **`ingredients`**) is an **`IEnumerable<Ingredient>`**, whereas the output sequence (represented by the variable **`highCalorieIngredientNamesQuery`**) is a different type; it is an **`IEnumerable<string>`**.

In this example, the chain of operators is working with a sequence of **`Ingredient`** elements until the **`Select`** operator transforms each element in the sequence; each **`Ingredient`** object is transformed to a simple **`string`**. This transformation is called **projection.** Input elements are **projected** into transformed output elements.

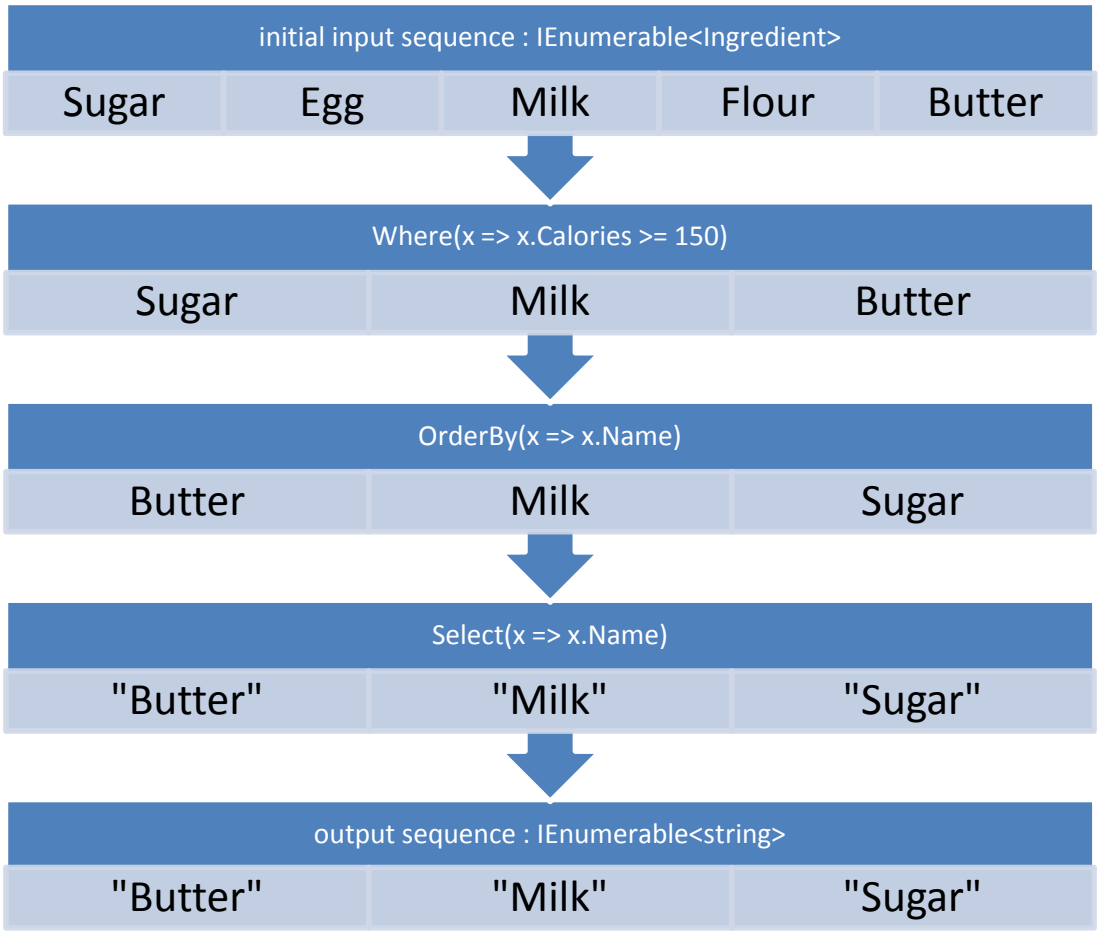


Figure 2: Multiple query operators acting in a chain

The lambda expression provided to the **`Select`** query operator decides what "shape" the elements in the output sequence will take. The lambda expression **`x => x.Name`** is telling the **`Select`** operator "for each **`Ingredient`** element, output a **`string`** element with the value of the **`Name`** property from the input **`Ingredient`**."

Note: Individual query operators will be discussed later in Chapter 3, LINQ Query Operators.

Query expression style

Query expressions offer a syntactical nicety on top of the fluent syntax.

The following code shows the equivalent version using query syntax of the preceding fluent style query.

```
Ingredient[] ingredients =
{
    new Ingredient{Name = "Sugar", Calories=500},
    new Ingredient{Name = "Egg", Calories=100},
    new Ingredient{Name = "Milk", Calories=150},
    new Ingredient{Name = "Flour", Calories=50},
    new Ingredient{Name = "Butter", Calories=200}
};

IEnumerable<string> highCalorieIngredientNamesQuery =
                                from i in ingredients
                                where i.Calories >= 150
                                orderby i.Name
                                select i.Name;


foreach (var ingredientName in highCalorieIngredientNamesQuery)
{
    Console.WriteLine(ingredientName);
}
```

Executing this code produces the same output as the fluent syntax version:

```
Butter
Milk
Sugar
```

The steps that are performed are the same as in the fluent syntax version, with each **query clause** (**`from`**, **`where`**, **`orderby`**, **`select`**) passing on a modified sequence to the next query clause.

The query expression in the preceding code begins with the **`from`** clause. The **`from`** clause has two purposes: the first is to describe what the input sequence is (in this case **`ingredients`**); the second is to introduce a **range variable**.

The final clause is the **`select`** clause, which describes what the output sequence will be from the entire query. Just as with the fluent syntax version, the **`select`** clause in the preceding code is projecting a sequence of **`Ingredient`** objects into a sequence of **`string`** objects.

Range variables

A range variable is an identifier that represents each element in the sequence in turn. It's similar to the variable used in a **`foreach`** statement; as the sequence is processed, this range variable represents the current element being processed. In the preceding code, the range variable **`i`** is being declared. Even though the same range variable identifier **`i`** is used in each clause, the sequence that the range variable "traverses" is different. Each clause works with the input sequence produced from the preceding clause (or the initial input **`IEnumerable<T>`**). This means that each clause is processing a different sequence; it is simply the name of the range variable identifier that is being reused.

In addition to the range variable introduced in the **`from`** clause, additional range variables can be added using other clauses or keywords. The following can introduce new range variables into a query expression:

- Additional **`from`** clauses
- The **`let`** clause
- The **`into`** keyword
- The **`join`** clause

The let clause

A **`let`** clause in a LINQ query expression allows the introduction of an additional range variable. This additional range variable can then be used in other clauses that follow it.

In the following code, the **`let`** clause is used to introduce a new range variable called **`isDairy`**, which will be of type **`Boolean`**.

```
Ingredient[] ingredients =
{
    new Ingredient{Name = "Sugar", Calories=500},
    new Ingredient{Name = "Egg", Calories=100},
    new Ingredient{Name = "Milk", Calories=150},
    new Ingredient{Name = "Flour", Calories=50},
    new Ingredient{Name = "Butter", Calories=200}
};

IEnumerable<Ingredient> highCalDairyQuery =
                        from i in ingredients
                        let isDairy = i.Name == "Milk" || i.Name == "Butter"
                        where i.Calories >= 150 && isDairy
                        select i;


foreach (var ingredient in highCalDairyQuery)
{
    Console.WriteLine(ingredient.Name);
}
```

The output of this code produces:

```
Milk
Butter
```

In the preceding code, the **isDairy** range variable is introduced and then used in the **where** clause. Notice that the original range variable **i** remains available to the **select** clause.

In this example, the new range variable is a simple scalar value, but **let** can also be used to introduce a subsequence. In the following code sample, the range variable **ingredients** is not a scalar value, but an array of strings.

```
string[] csvRecipes =
{
    "milk,sugar,eggs",
    "flour,BUTTER,eggs",
    "vanilla,ChEEsE,oats"
};


var dairyQuery =
    from csvRecipe in csvRecipes
    let ingredients = csvRecipe.Split(',')
    from ingredient in ingredients
    let uppercaseIngredient = ingredient.ToUpper()
    where uppercaseIngredient == "MILK"   ||
          uppercaseIngredient == "BUTTER" ||
          uppercaseIngredient == "CHEESE"
    select uppercaseIngredient;


foreach (var dairyIngredient in dairyQuery)
{
    Console.WriteLine("{0} is dairy", dairyIngredient);
}
```

Notice in the preceding code that we are using multiple **let** clauses as well as additional **from** clauses. Using additional **from** clauses is another way to introduce new range variables into a query expression.

The into keyword

The **into** keyword also allows a new identifier to be declared that can store the result of a **select** clause (as well as **group** and **join** clauses.)

The following code demonstrates using **into** to create a new anonymous type and then using this in the remainder of the query expression.

```
Ingredient[] ingredients =
{
    new Ingredient{Name = "Sugar", Calories=500},
    new Ingredient{Name = "Egg", Calories=100},
    new Ingredient{Name = "Milk", Calories=150},
    new Ingredient{Name = "Flour", Calories=50},
    new Ingredient{Name = "Butter", Calories=200}
};


IEnumerable<Ingredient> highCalDairyQuery =
    from i in ingredients
    select new // anonymous type
            {
                OriginalIngredient = i,
                IsDairy = i.Name == "Milk" || i.Name == "Butter",
                IsHighCalorie = i.Calories >= 150
            }
    into temp
    where temp.IsDairy && temp.IsHighCalorie
    // cannot write "select i;" as into hides the previous range variable i
    select temp.OriginalIngredient;



foreach (var ingredient in highCalDairyQuery)
{
    Console.WriteLine(ingredient.Name);
}
```

This code produces the following output:

```
Milk
Butter
```

Note that using **`into`** hides the previous range variable **`i`**. This means that **`i`** cannot be used in the final **`select`**.

The **`let`** clause, however, does not hide the previous range variable(s), meaning they can still be used later in query expressions.

The join clause

The **`join`** clause takes two input sequences in which elements in either sequence do not necessarily have any direct relationship in the class domain model.

To perform a join, some value of the elements in the first sequence is compared for equality with some value of the elements in the second sequence. It is important to note here that the **`join`** clause perform **equi-joins**; the values from both sequences are compared for equality. This means that the **`join`** clause does not support non-equijoins such as inequality, or comparisons such as greater-than or less-than. Because of this, rather than specifying joined elements using an operator such as **`==`** in C#, the **`equals`** keyword is used. The design thinking about introducing this keyword is to make it very clear that joins are equi-joins.

Common types of joins include:

- Inner joins.
- Group joins.
- Left outer joins.

The following **`join`** code examples assume that the following classes have been defined:

```
class Recipe
{
    public int Id { get; set; }
    public string Name { get; set; }
}

class Review
{
    public int RecipeId { get; set; }
    public string ReviewText { get; set; }
}
```

These two classes model the fact that recipes can have 0, 1, or many reviews. The **`Review`** class has a **`RecipeID`** property that holds the id of the recipe that the **`Review`** pertains to; notice here though that there is no direct relationship in the form of a property of type **`Recipe`**.

Inner join

An inner join returns an element in the output sequence for each item in the first sequence that has matching items in the second sequence. If an element in the first sequence does not have any matching elements in the second sequence, it will not appear in the output sequence.

Take the following code example:

```
Recipe[] recipes =
{
    new Recipe {Id = 1, Name = "Mashed Potato"},
    new Recipe {Id = 2, Name = "Crispy Duck"},
    new Recipe {Id = 3, Name = "Sachertorte"}
};

Review[] reviews =
{
    new Review {RecipeId = 1, ReviewText = "Tasty!"},
    new Review {RecipeId = 1, ReviewText = "Not nice :("},
    new Review {RecipeId = 1, ReviewText = "Pretty good"},
    new Review {RecipeId = 2, ReviewText = "Too hard"},
    new Review {RecipeId = 2, ReviewText = "Loved it"}
};


var query = from recipe in recipes
    join review in reviews on recipe.Id equals review.RecipeId
    select new // anonymous type
            {
                RecipeName = recipe.Name,
                RecipeReview = review.ReviewText
            };


foreach (var item in query)
{
    Console.WriteLine("{0} - '{1}'", item.RecipeName, item.RecipeReview);
}
```

In this preceding code, two input sequences are created: **recipes**, holding a number of recipes, and a second sequence, **reviews** of **Review** objects.

The query expression starts with the usual **from** clause pointing to the first sequence (**recipes**) and declaring the range variable **recipe**.

Next comes the use of the **join** clause. Here another range variable (**review**) is introduced that represents elements being processed in the **reviews** input sequence. The **on** keyword allows the specification of what value of the **recipe** range variable object is related to which value of the **review** range variable object. Again the **equals** keyword is used here to represent an equi-join. This **join** clause is essentially stating that reviews belong to recipes as identified by the common values of **Id** in **Recipe** and **RecipeId** in **Review**.

The result of executing this code is as follows:

```
Mashed Potato - 'Tasty!'
Mashed Potato - 'Not nice :('
Mashed Potato - 'Pretty good'
Crispy Duck - 'Too hard'
Crispy Duck - 'Loved it'
```

Notice here that the recipe “Sachertorte” does not exist in the output. This is because there are no reviews for it, i.e. there are no matching elements in the second input sequence (**reviews**).

Also notice that the result are “flat,” meaning there is no concept of groups of reviews belonging to a “parent” recipe.

Group join

A group join can produce a grouped hierarchical result where items in the second sequence are matched to items in the first sequence.

Unlike the previous inner join, the output of a group join can be organized hierarchically with reviews grouped into their related recipe.

```
Recipe[] recipes =
{
    new Recipe {Id = 1, Name = "Mashed Potato"},
    new Recipe {Id = 2, Name = "Crispy Duck"},
    new Recipe {Id = 3, Name = "Sachertorte"}
};

Review[] reviews =
{
    new Review {RecipeId = 1, ReviewText = "Tasty!"},
    new Review {RecipeId = 1, ReviewText = "Not nice :("},
    new Review {RecipeId = 1, ReviewText = "Pretty good"},
    new Review {RecipeId = 2, ReviewText = "Too hard"},
    new Review {RecipeId = 2, ReviewText = "Loved it"}
};

var query =
    from recipe in recipes
    join review in reviews on recipe.Id equals review.RecipeId
    into reviewGroup
    select new // anonymous type
            {
                RecipeName = recipe.Name,
                Reviews = reviewGroup // collection of related reviews
            };

foreach (var item in query)
{
    Console.WriteLine("Reviews for {0}", item.RecipeName);
```

```
    foreach (var review in item.Reviews)
    {
        Console.WriteLine(" - {0}", review.ReviewText);
    }
}
```

In this version, notice the addition of the **`into`** keyword. This allows the creation of hierarchical results. The **`reviewGroup`** range variable represents a sequence of reviews that match the join expression, in this case where the **`recipe.Id equals review.RecipeId`**.

To create the output sequence of groups (where each group contains the related reviews) the result of the query is projected into an anonymous type. Each instance of this anonymous type in the output sequence represents each group. The anonymous type has two properties: **`RecipeName`** coming from the element in the first sequence, and **`Reviews`** that come from the results of the join expression, i.e. the reviews that belong to the recipe.

The output of this code produces the following:

```
Reviews for Mashed Potato
 - Tasty!
 - Not nice :(
 - Pretty good
Reviews for Crispy Duck
 - Too hard
 - Loved it
Reviews for Sachertorte
```

Notice the hierarchical output here, and also that “Sachertorte” has been included in the output this time, even though it has no associated reviews.

Left outer join

To get flat, non-hierarchical output that also includes elements in the first sequence that have no matching elements in the second sequence (in our example “Sachertorte”), the **`DefaultIfEmpty()`** query operator can be used in conjunction with an additional **`from`** clause to introduce a new range variable, **`rg`**, that will be set to **`null`** if there are no matching elements in the second sequence. The select code to create the anonymous type projection is also modified to account for the fact that we may now have a null review.

Notice in the following code that **`RecipeReview = rg.ReviewText`** will produce a **`System.NullReferenceException`**; hence the need for the null checking code.

```
Recipe[] recipes =
{
    new Recipe {Id = 1, Name = "Mashed Potato"},
    new Recipe {Id = 2, Name = "Crispy Duck"},
    new Recipe {Id = 3, Name = "Sachertorte"}
};
```

```
Review[] reviews =
{
    new Review {RecipeId = 1, ReviewText = "Tasty!"},
    new Review {RecipeId = 1, ReviewText = "Not nice :("},
    new Review {RecipeId = 1, ReviewText = "Pretty good"},
    new Review {RecipeId = 2, ReviewText = "Too hard"},
    new Review {RecipeId = 2, ReviewText = "Loved it"}
};

var query =
    from recipe in recipes
    join review in reviews on recipe.Id equals review.RecipeId
    into reviewGroup
    from rg in reviewGroup.DefaultIfEmpty()
    select new // anonymous type
            {
                RecipeName = recipe.Name,
                // RecipeReview = rg.ReviewText System.NullReferenceException
                RecipeReview = ( rg == null ? "n/a" : rg.ReviewText )
            };

foreach (var item in query)
{
    Console.WriteLine("{0} - '{1}'", item.RecipeName, item.RecipeReview);
}
```

This produces the following output:

```
Mashed Potato - 'Tasty!'
Mashed Potato - 'Not nice :('
Mashed Potato - 'Pretty good'
Crispy Duck - 'Too hard'
Crispy Duck - 'Loved it'
Sachertorte - 'n/a'
```

Notice here that the results are flat, and because of the null checking code in the **`select`**, “Sachertorte” is included in the results with a review of “n/a.”

As an alternative to performing the null check in the **`select`**, the **`DefaultIfEmpty`** method can be instructed to create a new instance rather than creating a **`null`**. The following code shows this alternative query expression:

```
var query =
    from recipe in recipes
    join review in reviews on recipe.Id equals review.RecipeId
    into reviewGroup
    from rg in reviewGroup
                .DefaultIfEmpty(new Review{ReviewText = "n/a"})
    select new // anonymous type
    {
        RecipeName = recipe.Name,
        RecipeReview = rg.ReviewText
    };
```

Note that this code produces the same output as the previous version:

```
Mashed Potato - 'Tasty!'
Mashed Potato - 'Not nice :('
Mashed Potato - 'Pretty good'
Crispy Duck - 'Too hard'
Crispy Duck - 'Loved it'
Sachertorte - 'n/a'
```

Other query expression syntax

There are a number of other syntactical elements when using query expressions:

- The **group** clause
- The **orderby** clause
- The **ascending** and **descending** keywords
- The **by** keyword

The group clause and the by keyword

The **group** clause takes a flat input sequence and produces an output sequence of groups. Another way to think of this is that the output produced is like a list of lists; because of this, a nested **for** loop can be used to iterate over all the groups and group items.

The following code shows the use of the group clause and the **by** keyword to group all ingredients together that have the same amount of calories.

```
Ingredient[] ingredients =
{
    new Ingredient{Name = "Sugar", Calories=500},
    new Ingredient{Name = "Lard", Calories=500},
    new Ingredient{Name = "Butter", Calories=500},
    new Ingredient{Name = "Egg", Calories=100},
```

```
    new Ingredient{Name = "Milk", Calories=100},
    new Ingredient{Name = "Flour", Calories=50},
    new Ingredient{Name = "Oats", Calories=50}

};


IEnumerable<IGrouping<int, Ingredient>> query =
    from i in ingredients
    group i by i.Calories;


foreach (IGrouping<int, Ingredient> group in query)
{
    Console.WriteLine("Ingredients with {0} calories", group.Key);

    foreach (Ingredient ingredient in group)
    {
        Console.WriteLine(" - {0}", ingredient.Name);
    }
}
```

This code produces the following output:

```
Ingredients with 500 calories
 - Sugar
 - Lard
 - Butter
Ingredients with 100 calories
 - Egg
 - Milk
Ingredients with 50 calories
 - Flour
 - Oats
```

In the preceding code, the first generic type parameter (**int**) in the **IGrouping<int, Ingredient>** represents the key of the group, in this example the number of calories. The second generic type parameter (**Ingredient**) represents the type of the list of items that have the same key (calorie value).

Also note that explicit types have been used in the code to better illustrate what types are being operated upon. The code could be simplified by using the **var** keyword; for example: **var query = …** rather than **IEnumerable<IGrouping<int, Ingredient>> query = …**

The orderby clause and ascending & descending keywords

The **orderby** clause is used to produce a sequence sorted in either ascending (the default) or descending order.

The following code sorts a list of ingredients by name:

```
Ingredient[] ingredients =
{
    new Ingredient{Name = "Sugar", Calories=500},
    new Ingredient{Name = "Lard", Calories=500},
    new Ingredient{Name = "Butter", Calories=500},
    new Ingredient{Name = "Egg", Calories=100},
    new Ingredient{Name = "Milk", Calories=100},
    new Ingredient{Name = "Flour", Calories=50},
    new Ingredient{Name = "Oats", Calories=50}

};


IOrderedEnumerable<Ingredient> sortedByNameQuery =
    from i in ingredients
    orderby i.Name
    select i;


foreach (var ingredient in sortedByNameQuery)
{
    Console.WriteLine(ingredient.Name);
}
```

This produces the following sorted output:

```
Butter
Egg
Flour
Lard
Milk
Oats
Sugar
```

To sort in descending order (the non-default order), the query can be modified, and the **descending** keyword added, as shown in the following modified query:

```
IOrderedEnumerable<Ingredient> sortedByNameQuery =
    from i in ingredients
    orderby i.Name descending
    select i;
```

This produces the following output:

```
Sugar
Oats
Milk
Lard
Flour
Egg
Butter
```

The **orderby** clause can also be used when grouping is being applied. In the following code, the calorie-grouped ingredients are sorted by the number of calories (which is the **Key** of each group).

```
Ingredient[] ingredients =
{
    new Ingredient{Name = "Sugar", Calories=500},
    new Ingredient{Name = "Lard", Calories=500},
    new Ingredient{Name = "Butter", Calories=500},
    new Ingredient{Name = "Egg", Calories=100},
    new Ingredient{Name = "Milk", Calories=100},
    new Ingredient{Name = "Flour", Calories=50},
    new Ingredient{Name = "Oats", Calories=50}
};


IEnumerable<IGrouping<int, Ingredient>> query =
    from i in ingredients
    group i by i.Calories
    into calorieGroup
    orderby calorieGroup.Key
    select calorieGroup;


foreach (IGrouping<int, Ingredient> group in query)
{
    Console.WriteLine("Ingredients with {0} calories", group.Key);

    foreach (Ingredient ingredient in group)
    {
        Console.WriteLine(" - {0}", ingredient.Name);
    }
}
```

Notice in the preceding code, that even though the input sequence **ingredients** is in descending calorie order, the output below is sorted by the calorie value (the **Key**) of the group in **ascending** order:

```
Ingredients with 50 calories
 - Flour
 - Oats
Ingredients with 100 calories
 - Egg
 - Milk
Ingredients with 500 calories
 - Sugar
 - Lard
 - Butter
```

Using the different styles

The two LINQ styles can be used together; for example, fluent query operators can be mixed in with the query expression style.

Advantages of the different styles

Each syntax style has its own benefits.

Query expression style keyword availability

Not every query operator that is available using the fluent syntax has an equivalent query expression syntax keyword. The following query operators using the fluent style have associated keywords when using the query expression syntax:

- **`GroupBy`**
- **`GroupJoin`**
- **`Join`**
- **`OrderBy`**
- **`OrderByDescending`**
- **`Select`**
- **`SelectMany`**
- **`ThenBy`**
- **`ThenByDescending`**
- **`Where`**

Number of operators

If the query only requires the use of a single query operator to get the required results, then a single call to a fluent style operator is usually a smaller amount of code to write, and can also be comprehended more quickly by readers.

The following code shows a query returning all ingredients over 150 calories using both the fluent style (**`q1`**) and query expression style (**`q2`**). Notice the fluent style is much terser when needing to use only a single operator.

```
var q1 = ingredients.Where(x => x.Calories > 100);

var q2 = from i in ingredients
    where i.Calories > 100
    select i;
```

Simple queries using basic operators

If the query is relatively simple and uses a few basic query operators such as **`Where`** and **`OrderBy`**, either the fluent or the query expression styles can be used successfully. The choice between the two styles in these cases is usually down to the personal preference of the programmer or the coding standards imposed by the organization/client.

Complex queries with range additional range variables

Once queries become more complex and require the use of additional range variables (e.g. when using the **`let`** clause or performing joins), then query expressions are usually simpler than the equivalent fluent style version.

Mixing the styles in a single query

If a query is being written using the query expression style, it is still possible to make use of query operators that have no equivalent query syntax keyword. This is accomplished by mixing both fluent syntax and query syntax in the same query.

We have already seen an example of mixing the two styles with the use of the **`DefaultIfEmpty`** fluent style operator when we discussed left outer joins earlier in this chapter.

The following example shows mixing the **`Count`** query operator (using fluent syntax) with a query expression (inside the parentheses).

```
int mixedQuery = (from i in ingredients
                  where i.Calories > 100
                  select i).Count();
```

Chapter 3 LINQ Query Operators

This chapter discusses the standard query operators and is organized by the classification of the operators. Query operators can be grouped into categories such as join operators, generation operators, and projection operators.

At a higher level, query operators can be classified as falling into one of three categories based on their inputs/outputs:

- Input sequence(s) result in an output sequence.
- Input sequence results in scalar value/single element output.
- No input results in an output sequence (these operators **generate** their own elements).

The final category might seem strange at first glance, as these operators do not take an input sequence. These operators can be useful, for example, when sequence of integers needs creating, by reducing the amount of code we need to write to perform this generation.

Beyond this simple categorization based on input/output, the standard query operators can be grouped as shown in the following list. The operators in each category will be discussed in more detail later in this chapter.

Query operators:

- Restriction: **`Where`**
- Projection: **`Select`**, **`SelectMany`**
- Partitioning: **`Take`**, **`Skip`**, **`TakeWhile`**, **`SkipWhile`**
- Ordering: **`OrderBy`**, **`OrderByDescending`**, **`ThenBy`**, **`ThenByDescending`**, **`Reverse`**
- Grouping: **`GroupBy`**
- Set: **`Concat`**, **`Union`**, **`Intersect`**, **`Except`**
- Conversion: **`ToArray`**, **`ToList`**, **`ToDictionary`**, **`ToLookup`**, **`OfType`**, **`Cast`**
- Element: **`First`**, **`FirstOrDefault`**, **`Last`**, **`LastOrDefault`**, **`Single`**, **`SingleOrDefault`**, **`ElementAt`**, **`ElementAtOrDefault`**, **`DefaultIfEmpty`**
- Generation: **`Empty`**, **`Range`**, **`Repeat`**
- Quantifiers: **`Any`**, **`All`**, **`Contains`**, **`SequenceEqual`**
- Aggregate: **`Count`**, **`LongCount`**, **`Sum`**, **`Min`**, **`Max`**, **`Average`**, **`Aggregate`**
- Joining: **`Join`**, **`GroupJoin`**, **`Zip`**

Restriction Operators

Restriction query operators take an input sequence and output a sequence of elements that are restricted (“filtered”) in some way. The elements that make up the output sequence are those that match the specified restriction. Individual output elements themselves are not modified or transformed.

Where

The **Where** query operator returns output elements that match a given predicate.

Tip: A predicate is a function that returns a bool result of true if some condition is satisfied.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 150},
    new Ingredient {Name = "Flour", Calories = 50},
    new Ingredient {Name = "Butter", Calories = 200}
};

IEnumerable<Ingredient> query = ingredients
                                    .Where(x => x.Calories >= 200);

foreach (var ingredient in query)
{
    Console.WriteLine(ingredient.Name);
}
```

The preceding code shows the **Where** operator in use. Here the predicate **x => x.Calories >= 200** will return **true** for every ingredient that is greater than or equal to 200 calories.

This code produces the following output:

```
Sugar
Butter
```

An overload of **Where** provides the positional index of the input element for use in the predicate.

In the preceding code, “Sugar” has an index of 0 and “Butter” has an index of 4. The following query produces the same output as the previous version.

```
IEnumerable<Ingredient> queryUsingIndex = ingredients
                                    .Where( (ingredient, index) =>
                                        ingredient.Name == "Sugar" ||
                                        index == 4);
```

Notice the lambda expression is different. In this code, the predicate function will be of type **Func<Ingredient, int, bool>**; the **int** parameter provides the position of the element in the input sequence.

Query expression usage

The **where** keyword is used in query expression style code, and is followed by a Boolean expression, as shown in the following code.

```
IEnumerable<Ingredient> queryEx = from i in ingredients
                                  where i.Calories >= 200
                                  select i;
```

Projection Operators

Projection query operators take an input sequence, transform each element in that sequence, and produce an output sequence of these transformed elements. In this way, the input sequence is **projected** into a different output sequence.

Select

The **Select** query operator transforms each element in the input sequence to an element in the output sequence. There will be the same number of elements in the output sequence as there are in the input sequence.

The following query projects the input sequence of **Ingredient** elements into the output sequence of **string** elements. The lambda expression is describing the projection: taking each input **Ingredient** element and returning a **string** element.

```
IEnumerable<string> query = ingredients.Select(x => x.Name);
```

We could create an output sequence of **int**s with the following code:

```
IEnumerable<int> queryNameLength = ingredients.Select(x => x.Name.Length);
```

The projection can also result in new complex objects being produced in output. In the following code each input **Ingredient** is projected into a new instance of an **IngredientNameAndLength** object.

```
class IngredientNameAndLength
{
    public string Name { get; set; }
    public int Length { get; set; }

    public override string ToString()
```

```
    {
        return Name + " - " + Length;
    }
}

…

IEnumerable<IngredientNameAndLength> query = ingredients.Select(x =>
                                                new IngredientNameAndLength
                                                {
                                                   Name = x.Name,
                                                   Length = x.Name.Length
                                                });
```

This query could also be written using an anonymous type; notice in the following code the query return type has been change to **var**.

```
var query = ingredients.Select(x =>
                                    new
                                    {
                                        Name = x.Name,
                                        Length = x.Name.Length
                                    });
```

Query expression usage

The **select** keyword is used in query expression style code as shown in the following code that replicates the preceding fluent style version using an anonymous type.

```
var query = from i in ingredients
    select new
            {
                Name = i.Name,
                Length = i.Name.Length
            };
```

SelectMany

Whereas the **Select** query operator returns an output element for every input element, the **SelectMany** query operator produces a variable number of output elements for each input element. This means that the output sequence may contain more or fewer elements than were in the input sequence.

The lambda expression in the **Select** query operator returns a **single** item. The lambda expression in a **SelectMany** query operator produces a **child sequence**. This child sequence may contain a varying number of elements for each element in the input sequence.

In the following code, the lambda expression will produce a child sequence of **char**s, one char for each letter of each input element. For example. the input element “Sugar,” when processed by the lambda expression, will produce a child sequence containing five elements: ‘S’, ‘u’, ‘g’, ‘a’, and ‘r’. Each input ingredient string can be a different number of letters, so the lambda expression will produce child sequences of different lengths. Each of the child sequences produced are concatenated (“flattened”) into a single output sequence.

```
string[] ingredients = {"Sugar", "Egg", "Milk", "Flour", "Butter"};

IEnumerable<char> query = ingredients.SelectMany(x => x.ToCharArray());

foreach (char item in query)
{
    Console.WriteLine(item);
}
```

This code produces the following output (notice there are more output elements than there were in the input sequence that only contained five elements):

```
S
u
g
a
r
E
g
g
M
i
l
k
F
l
o
u
r
B
u
t
t
e
r
```

Query expression usage

In the query expression style, an additional **from** clause is added to produce the child sequence.

The following code produces the same output as the previous fluent version.

```
IEnumerable<char> query = from i in ingredients
                          from c in i.ToCharArray()
                          select c;
```

Partitioning Operators

The partitioning query operators take an input sequence and partition it, or “return a chunk” in the output sequence. One use of these operators is to break up a larger sequence into “pages,” for example, paging though ten results at a time in a user interface.

Take

The **Take** query operator takes in an input sequence and returns the specified number of elements as an output.

The following code shows how to use the **Take** query operator to return the first three ingredients in the sequence.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 150},
    new Ingredient {Name = "Flour", Calories = 50},
    new Ingredient {Name = "Butter", Calories = 200}
};


IEnumerable<Ingredient> firstThree = ingredients.Take(3);


foreach (var ingredient in firstThree)
{
    Console.WriteLine(ingredient.Name);
}
```

The preceding code produces the following output:

```
Sugar
Egg
Milk
```

As with other query operators, **Take** can be chained together. In the following code, **Where** is used to restrict the ingredients to only those containing more than 100 calories. The **Take** query operator then takes this restricted sequence and returns the first two ingredients.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 150},
    new Ingredient {Name = "Flour", Calories = 50},
    new Ingredient {Name = "Butter", Calories = 200}
};


IEnumerable<Ingredient> query = ingredients
                                        .Where(x => x.Calories > 100)
                                        .Take(2);


foreach (var ingredient in query)
{
    Console.WriteLine(ingredient.Name);
}
```

This code produces the following result:

```
Sugar
Milk
```

Query expression usage

There is no keyword for **Take** when using the query expression style; however, mixed style can be used as the following code demonstrates.

```
IEnumerable<Ingredient> query = (from i in ingredients
                                 where i.Calories > 100
                                 select i).Take(2);
```

TakeWhile

The **TakeWhile** query operator, rather than return elements based on a fixed specified number of elements, instead continues to return input elements until such time as a specified predicate becomes false.

The following code will continue to “take” elements from the input sequence while the number of calories is greater than or equal to 100.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 150},
    new Ingredient {Name = "Flour", Calories = 50},
    new Ingredient {Name = "Butter", Calories = 200}
};


IEnumerable<Ingredient> query = ingredients
                                    .TakeWhile(x => x.Calories >= 100);


foreach (var ingredient in query)
{
    Console.WriteLine(ingredient.Name);
}
```

This produces the following output:

```
Sugar
Egg
Milk
```

Notice here that as soon as an element in the input sequence is reached that does not match the predicate, the “taking” stops; no more elements from the remainder of the input sequence are returned, even if they match the predicate.

There is an overload of **`TakeWhile`** that allows the positional index of the input element to be used; in the preceding code, this predicate would take the form **`Func<Ingredient, int, bool>`**. The **`int`** generic type parameter here represents the positional index of the current **`Ingredient`** being examined.

Query expression usage

There is no keyword for **`TakeWhile`** when using the query expression style; however, mixed style can be used.

Skip

The **`Skip`** query operator will ignore the specified number of input elements from the start of the input sequence and return the remainder.

The following code skips over the first three elements and returns the rest:

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 150},
    new Ingredient {Name = "Flour", Calories = 50},
    new Ingredient {Name = "Butter", Calories = 200}
};


IEnumerable<Ingredient> query = ingredients.Skip(3);


foreach (var ingredient in query)
{
    Console.WriteLine(ingredient.Name);
}
```

This produces the following output:

```
Flour
Butter
```

Query expression usage

There is no keyword for **`Skip`** when using the query expression style; however, mixed style can be used.

Using Skip and Take for result set paging

When used together, the **`Skip`** and **`Take`** query operators can implement the paging of result sets for display to users, as demonstrated in the following code.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 150},
    new Ingredient {Name = "Flour", Calories = 50},
    new Ingredient {Name = "Butter", Calories = 200}
};
```

```
IEnumerable<Ingredient> firstPage = ingredients.Take(2);

IEnumerable<Ingredient> secondPage = ingredients
                                        .Skip(2)
                                        .Take(2);

IEnumerable<Ingredient> thirdPage = ingredients
                                        .Skip(4)
                                        .Take(2);

Console.WriteLine("Page One:");
foreach (var ingredient in firstPage)
{
    Console.WriteLine(" - " + ingredient.Name);
}

Console.WriteLine("Page Two:");
foreach (var ingredient in secondPage)
{
    Console.WriteLine(" - " + ingredient.Name);
}

Console.WriteLine("Page Three:");
foreach (var ingredient in thirdPage)
{
    Console.WriteLine(" - " + ingredient.Name);
}
```

This produces the following output:

```
Page One:
 - Sugar
 - Egg
Page Two:
 - Milk
 - Flour
Page Three:
 - Butter
```

SkipWhile

Like **`TakeWhile`**, the **`SkipWhile`** query operator uses a predicate to evaluate each element in the input sequence. **`SkipWhile`** will ignore items in the input sequence until the supplied predicate returns **`false`**.

The following code will skip input elements until “Milk” is encountered; at this point the predicate **`x => x.Name != "Milk"`** becomes false, and the remainder of the ingredients will be returned.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 150},
    new Ingredient {Name = "Flour", Calories = 50},
    new Ingredient {Name = "Butter", Calories = 200}
};


IEnumerable<Ingredient> query = ingredients.SkipWhile(x => x.Name != "Milk");


foreach (var ingredient in query)
{
    Console.WriteLine(ingredient.Name);
}
```

This produces the following output:

```
Milk
Flour
Butter
```

Just as with **`TakeWhile`**, there is an overload of **`SkipWhile`** that allows the positional index of the input element to be used.

Query expression usage

There is no keyword for **`SkipWhile`** when using the query expression style; however, mixed style can be used.

Ordering Operators

The ordering query operators return the same number of elements as the input sequence, but arranged in a (potentially) different order.

OrderBy

The **`OrderBy`** query operator sorts the input sequence by comparing the elements in the input sequence using a specified key.

The following code shows the sorting of a simple input sequence of strings. In this example, the key is the string itself as specified by the lambda expression **x => x**.

```
string[] ingredients =
{
    "Sugar",
    "Egg",
    "Milk",
    "Flour",
    "Butter"
};


var query = ingredients.OrderBy(x => x);


foreach (var item in query)
{
    Console.WriteLine(item);
}
```

This produces the following output:

```
Butter
Egg
Flour
Milk
Sugar
```

The lambda expression that is used to specify the sorting key can specify a property of the input element itself, as the following code example shows.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 150},
    new Ingredient {Name = "Flour", Calories = 50},
    new Ingredient {Name = "Butter", Calories = 200}
};


var query = ingredients.OrderBy(x => x.Calories);


foreach (var item in query)
{
    Console.WriteLine(item.Name + " " + item.Calories);
```

```
}
```

In the preceding code, the lambda expression **`x => x.Calories`** is selecting the **`Calories`** property to be the key that is sorted on.

This produces the following output:

```
Flour 50
Egg 100
Milk 150
Butter 200
Sugar 500
```

Query expression usage

The **`orderby`** keyword is used when using the query expression style, as shown in the following code.

```
var query = from i in ingredients
            orderby i.Calories
            select i;
```

ThenBy

The **`ThenBy`** query operator can be chained one or multiple times following an initial **`OrderyBy`**. The **`ThenBy`** operator adds additional levels of sorting. In the following code, the ingredients are first sorted with the initial **`OrderBy`** sort (sorting into calorie order); this sorted sequence is then further sorted by ingredient name while maintaining the initial calorie sort.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Milk", Calories = 100},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Flour", Calories = 500},
    new Ingredient {Name = "Butter", Calories = 200}
};


var query = ingredients.OrderBy(x => x.Calories)
                       .ThenBy(x => x.Name);


foreach (var item in query)
{
    Console.WriteLine(item.Name + " " + item.Calories);
}
```

This produces the following output:

```
Egg 100
Milk 100
Butter 200
Flour 500
Sugar 500
```

Notice that in the original input sequence, “Milk” came before “Egg,” but because of the **ThenBy** operator, in the output the ingredients are sorted by name within matching calorie values.

Query expression usage

The **orderby** keyword is used when using the query expression style, but rather than a single sort expression, subsequent sort expressions are added as a comma-separated list, as shown in the following code.

```
var query = from i in ingredients
            orderby i.Calories, i.Name
            select i;
```

OrderByDescending

The **OrderByDescending** query operator works in the same fashion as the **OrderBy** operator, except that the results are returned in the reverse order, as the following code demonstrates.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Milk", Calories = 100},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Flour", Calories = 500},
    new Ingredient {Name = "Butter", Calories = 200}
};


var query = ingredients.OrderByDescending(x => x.Calories);


foreach (var item in query)
{
    Console.WriteLine(item.Name + " " + item.Calories);
}
```

This produces the following output:

```
Sugar 500
Flour 500
Butter 200
Milk 100
Egg 100
```

Note that this time the calorie numbers are in descending order.

Query expression usage

The **`descending`** keyword is used when using the query expression style, and is applied after the sort expression itself, as shown in the following code.

```
var query = from i in ingredients
            orderby i.Calories descending
            select i;
```

ThenByDescending

The **`ThenByDescending`** query operator follows the same usage as the **`ThenBy`** operator, but returns results in the reverse sort order.

The following code shows the ingredients being sorted first by calorie (in ascending order), then by the ingredient names, but this time the names are sorted in descended order.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Flour", Calories = 500},
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 100},
    new Ingredient {Name = "Butter", Calories = 200}
};


var query = ingredients.OrderBy(x => x.Calories)
                       .ThenByDescending(x => x.Name);


foreach (var item in query)
{
    Console.WriteLine(item.Name + " " + item.Calories);
}
```

This produces the following output:

```
Milk 100
Egg 100
Butter 200
Sugar 500
Flour 500
```

Notice here that in the input sequence, “Flour” comes before “Sugar,” but because of the **`ThenByDescending`**, they are reversed in the output.

Query expression usage

The **`descending`** keyword is used when using the query expression style, as shown in the following code.

```
var query = from i in ingredients
            orderby i.Calories, i.Name descending
            select i;
```

Reverse

The **`Reverse`** query operator simply takes the input sequence and returns the elements in the reverse order in the output sequence; so, for example, the first element in the input sequence will become the last element in the output sequence.

```
char[] letters ={'A', 'B', 'C'};

var query = letters.Reverse();

foreach (var item in query)
{
    Console.WriteLine(item);
}
```

The preceding code produces the following output:

```
C
B
A
```

Query expression usage

There is no keyword for **`Reverse`** when using the query expression style; however, mixed style can be used.

Grouping Operators

GroupBy

The **GroupBy** query operator takes an input sequence and returns an output sequence of sequences (groups). The basic overload to the **GroupBy** method takes a lambda expression representing the key to create the groups for.

The following code shows how to group all ingredients by number of calories.

```
Ingredient[] ingredients =
{
    new Ingredient{Name = "Sugar", Calories=500},
    new Ingredient{Name = "Lard", Calories=500},
    new Ingredient{Name = "Butter", Calories=500},
    new Ingredient{Name = "Egg", Calories=100},
    new Ingredient{Name = "Milk", Calories=100},
    new Ingredient{Name = "Flour", Calories=50},
    new Ingredient{Name = "Oats", Calories=50}

};


IEnumerable<IGrouping<int, Ingredient>> query =
                                        ingredients.GroupBy(x => x.Calories);


foreach (IGrouping<int, Ingredient> group in query)
{
    Console.WriteLine("Ingredients with {0} calories", group.Key);

    foreach (Ingredient ingredient in group)
    {
        Console.WriteLine(" - {0}", ingredient.Name);
    }
}
```

Notice in the preceding code the output sequence of sequences is essentially a list (**IEnumerable**) of **IGrouping**. Each **IGrouping** element contains the key (in this case an **int** for the calories), and the list of ingredients that have the same number of calories.

This produces the following output:

```
Ingredients with 500 calories
 - Sugar
 - Lard
 - Butter
Ingredients with 100 calories
 - Egg
 - Milk
Ingredients with 50 calories
 - Flour
 - Oats
```

Query expression usage

The **group** keyword is used when using the query expression style; we saw the **group** keyword in use in Chapter 2, “Fluent and Query Expression Styles.”

Set Operators

The set query operators perform set-based operations. These operators take two input sequences representing the two “sets” and return a single output sequence.

The set query operators consist of the following:

- **Concat**
- **Union**
- **Distinct**
- **Intersect**
- **Except**

Concat

The **Concat** query operator takes a first sequence and returns this with all the elements of a second sequence added to it, as shown in the following code.

```
string[] applePie = {"Apple", "Sugar", "Pastry", "Cinnamon"};
string[] cherryPie = { "Cherry", "Sugar", "Pastry", "Kirsch" };

IEnumerable<string> query = applePie.Concat(cherryPie);

foreach (string item in query)
{
    Console.WriteLine(item);
}
```

This produces the following output:

```
Apple
Sugar
Pastry
Cinnamon
Cherry
Sugar
Pastry
Kirsch
```

Notice that any duplicate elements (such as “Sugar” and “Pastry”) are preserved.

Query expression usage

There is no keyword for **`Concat`** when using the query expression style; however, mixed style can be used.

Union

The **`Union`** query operator performs a similar “joining” operation as **`Concat`**; however, any duplicate elements that exist in both the first and second sequence are not duplicated in the output.

```
string[] applePie = { "Apple", "Sugar", "Pastry", "Cinnamon" };
string[] cherryPie = { "Cherry", "Sugar", "Pastry", "Kirsch" };

IEnumerable<string> query = applePie.Union(cherryPie);

foreach (string item in query)
{
    Console.WriteLine(item);
}
```

This produces the following output:

```
Apple
Sugar
Pastry
Cinnamon
Cherry
Kirsch
```

Notice here that any duplicates (e.g. “Sugar” and “Pastry”) have been removed from the output sequence.

Query expression usage

There is no keyword for **Union** when using the query expression style; however, mixed style can be used.

Distinct

While the **Distinct** query operator may not, strictly speaking, be a set operator in that it only operates on a single input sequence, it is included here to show how the **Union** operator can by simulated by using a **Concat** followed by a **Distinct**. Also note that the **Distinct** query operator may be chained together with other query operators and not just the set-based operators.

```
string[] applePie = { "Apple", "Sugar", "Pastry", "Cinnamon" };
string[] cherryPie = { "Cherry", "Sugar", "Pastry", "Kirsch" };

IEnumerable<string> query = applePie.Concat(cherryPie)
                                    .Distinct();

foreach (string item in query)
{
    Console.WriteLine(item);
}
```

This produces the following output:

```
Apple
Sugar
Pastry
Cinnamon
Cherry
Kirsch
```

Notice this is that same output as when using **Union**.

Query expression usage

There is no keyword for **Distinct** when using the query expression style; however, mixed style can be used.

Intersect

The **Intersect** query operator returns only those elements that exist in both the first and second input sequences. In the following code, the only two elements that are common to both sequences are “Sugar” and “Pastry.”

```
string[] applePie = { "Apple", "Sugar", "Pastry", "Cinnamon" };
string[] cherryPie = { "Cherry", "Sugar", "Pastry", "Kirsch" };

IEnumerable<string> query = applePie.Intersect(cherryPie);


foreach (string item in query)
{
    Console.WriteLine(item);
}
```

This produces the following output:

```
Sugar
Pastry
```

Also notice here that the common elements are not duplicated in the output sequence; we do not see two “Sugar” elements, for example.

Query expression usage

There is no keyword for **`Intersect`** when using the query expression style; however, mixed style can be used.

Except

The **`Except`** query operator will return only those elements in the first sequence where those same elements do not exist in the second sequence.

Take the following code:

```
string[] applePie = { "Apple", "Sugar", "Pastry", "Cinnamon" };
string[] cherryPie = { "Cherry", "Sugar", "Pastry", "Kirsch" };

IEnumerable<string> query = applePie.Except(cherryPie);

foreach (string item in query)
{
    Console.WriteLine(item);
}
```

This produces the following output:

```
Apple
Cinnamon
```

Note that we do **not** get any elements returned from the second sequence.

Query expression usage

There is no keyword for **`Except`** when using the query expression style; however, mixed style can be used.

Conversion Operators

The conversion query operators convert from **`IEnumerable<T>`** sequences to other types of collections.

The following query operators can be used to perform conversion of sequences:

- **`OfType`**
- **`Cast`**
- **`ToArray`**
- **`ToList`**
- **`ToDictionary`**
- **`ToLookup`**

OfType

The **`OfType`** query operator returns an output sequence containing only those elements of a specified type. The extension method that implements the **`OfType`** query operator has the following signature:

```
public static IEnumerable<TResult> OfType<TResult>(this IEnumerable source)
```

Notice here that the extension method is “extending” the non-generic type **`IEnumerable`**. This means that the input sequence may contain elements of different types.

The following code shows an **`IEnumerable`** (object array) containing objects of different types. The **`OfType`** query operator can be used here to return (for example) all the **`string`** objects. Also notice in the following code that the result of the query is the generic version of **`IEnumerable`**, namely **`IEnumerable<string>`**.

```
IEnumerable input = new object[]
                    {
                        "Apple", 33, "Sugar", 44, 'a', new DateTime()
                    };

IEnumerable<string> query = input.OfType<string>();

foreach (string item in query)
```

```
{
    Console.WriteLine(item);
}
```

This produces the following output:

```
Apple
Sugar
```

Because **`IEnumerable<T>`** implements **`IEnumerable`**, **`OfType`** can also be used on strongly-typed sequences. The following code show a simple object-oriented hierarchy.

```
class Ingredient
{
    public string Name { get; set; }
}

class DryIngredient : Ingredient
{
    public int Grams { get; set; }
}

class WetIngredient : Ingredient
{
    public int Millilitres { get; set; }
}
```

The following code shows how **`OfType`** can be used to get a specific subtype, in this example getting all the **`WetIngredient`** objects:

```
IEnumerable<Ingredient> input = new Ingredient[]
        {
            new DryIngredient{ Name = "Flour"},
            new WetIngredient{Name = "Milk"},
            new WetIngredient{Name = "Water"}
        };

IEnumerable<WetIngredient> query = input.OfType<WetIngredient>();

foreach (WetIngredient item in query)
{
    Console.WriteLine(item.Name);
}
```

This produces the following output:

```
Milk
Water
```

Query expression usage

There is no keyword for **`OfType`** when using the query expression style; however, mixed style can be used.

Cast

Like the **`OfType`** query operator, the **`Cast`** query operator can take an input sequence and return an output sequence containing elements of a specific type. Each element in the input sequence is cast to the specified type; however, unlike with **`OfType`**, which just ignores any incompatible types, if the cast is not successful, an exception will be thrown.

The following code shows an example of where using **`Cast`** will throw an exception.

```
IEnumerable input = new object[]
        {
            "Apple", 33, "Sugar", 44, 'a', new DateTime()
        };

IEnumerable<string> query = input.Cast<string>();

foreach (string item in query)
{
    Console.WriteLine(item);
}
```

Running this code will cause a “System.InvalidCastException : Unable to cast object of type 'System.Int32' to type 'System.String'” exception.

Query expression usage

While there is no keyword for **`Cast`** when using the query expression style, it is supported by giving the range variable a specific type, as in the following code.

```
IEnumerable input = new object[] {"Apple", "Sugar", "Flour"};

IEnumerable<string> query = from string i in input
                            select i;
```

Note in the preceding code the explicit type declaration **`string`** before the range variable name **`i`**.

ToArray

The **ToArray** query operator takes an input sequence and creates an output array containing the elements. **ToArray** will also cause immediate query execution (enumeration of the input sequence) and overrides the default deferred execution behavior of LINQ.

In the following code, the **IEnumerable<string>** is converted to an array of strings.

```
IEnumerable<string> input = new List<string> { "Apple", "Sugar", "Flour" };

string[] array = input.ToArray();
```

Query expression usage

There is no keyword for **ToArray** when using the query expression style; however, mixed style can be used.

ToList

The **ToList** query operator, like the **ToArray** operator, usually bypasses deferred execution. Whereas **ToArray** creates an array representing the input elements, **ToList**, as its name suggests, instead outputs a **List<T>**.

```
IEnumerable<string> input = new []{ "Apple", "Sugar", "Flour" };

List<string> list = input.ToList();
```

Query expression usage

There is no keyword for **ToList** when using the query expression style; however, mixed style can be used.

ToDictionary

The **ToDictionary** query operator converts an input sequence into a generic **Dictionary<TKey, TValue>**.

The simplest overload of the **ToDictionary** method takes a lambda expression representing a function, which selects what is used for the key for each item in the resulting Dictionary.

```
IEnumerable<Recipe> recipes = new[]
                {
                    new Recipe {Id = 1, Name = "Apple Pie", Rating = 5},
                    new Recipe {Id = 2, Name = "Cherry Pie", Rating = 2},
                    new Recipe {Id = 3, Name = "Beef Pie", Rating = 3}
                };
```

```
Dictionary<int, Recipe> dict = recipes.ToDictionary(x => x.Id);

foreach (KeyValuePair<int, Recipe> item in dict)
{
    Console.WriteLine("Key = {0}, Recipe = {1}", item.Key, item.Value.Name);
}
```

In the preceding code, the **`Key`** of the items in the resultant **`Dictionary`** is of type **`int`** and is set to the **`Id`** each **`Recipe`**. The **`Value`** of each item in the **`Dictionary`** represents the **`Recipe`** itself.

Running the preceding code produces the following output:

```
Key = 1, Recipe = Apple Pie
Key = 2, Recipe = Cherry Pie
Key = 3, Recipe = Beef Pie
```

Query expression usage

There is no keyword for **`ToDictionary`** when using the query expression style; however, mixed style can be used.

ToLookup

The **`ToLookup`** query operator works in a similar fashion to the **`ToDictionary`** operator, but rather than producing a dictionary, it creates an instance of an **`ILookUp`**.

The following code shows the creation of a lookup that groups recipes into similar ratings. In this example, the key comes from the **`byte`** rating.

```
IEnumerable<Recipe> recipes = new[]
    {
        new Recipe {Id = 1, Name = "Apple Pie", Rating = 5},
        new Recipe {Id = 1, Name = "Banana Pie", Rating = 5},
        new Recipe {Id = 2, Name = "Cherry Pie", Rating = 2},
        new Recipe {Id = 3, Name = "Beef Pie", Rating = 3}
    };

ILookup<byte, Recipe> look = recipes.ToLookup(x => x.Rating);

foreach (IGrouping<byte, Recipe> ratingGroup in look)
{
    byte rating = ratingGroup.Key;

    Console.WriteLine("Rating  {0}", rating);

    foreach (var recipe in ratingGroup)
    {
        Console.WriteLine(" - {0}", recipe.Name);
    }
}
```

This produces the following output:

```
Rating  5
 - Apple Pie
 - Banana Pie
Rating  2
 - Cherry Pie
Rating  3
 - Beef Pie
```

Query expression usage

There is no keyword for **ToLookup** when using the query expression style; however, mixed style can be used.

Element Operators

The element operators take an input sequence and return a single element from that input sequence, or some other default single value. In this way, the element operators do not themselves return sequences.

First

The **First** query operator simply returns the initial element in the input sequence, as the following code demonstrates.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 150},
    new Ingredient {Name = "Flour", Calories = 50},
    new Ingredient {Name = "Butter", Calories = 500}
};

Ingredient element = ingredients.First();

Console.WriteLine(element.Name);
```

This produces the following output:

```
Sugar
```

An overload of **First** allows a predicate to be specified. This will return the first item in the input sequence that satisfies this predicate. The following code shows how to find the first **Ingredient** element that has a calorie value of 150.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 150},
    new Ingredient {Name = "Flour", Calories = 50},
    new Ingredient {Name = "Butter", Calories = 500}
};

Ingredient element = ingredients.First(x => x.Calories == 150);

Console.WriteLine(element.Name);
```

This produces the following output:

```
Milk
```

If the input sequence contains zero elements, the use of **`First`** will result in an exception.

The following code will result in an exception: “System.InvalidOperationException : Sequence contains no elements”.

```
Ingredient[] ingredients = {};

Ingredient element = ingredients.First();
```

If **`First`** is used with a predicate and that predicate cannot be satisfied by any element in the input sequence, an exception will also be thrown. The following code produces the exception: “System.InvalidOperationException : Sequence contains no matching element.” This is because no **`Ingredient`** exists with 9,999 calories.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 150},
    new Ingredient {Name = "Flour", Calories = 50},
    new Ingredient {Name = "Butter", Calories = 500}
};

Ingredient element = ingredients.First(x => x.Calories == 9999);
```

Query expression usage

There is no keyword for **`First`** when using the query expression style; however, mixed style can be used.

FirstOrDefault

The **`FirstOrDefault`** query operator works in a similar way to **`First`**, but instead of an exception being thrown, the default value for the element type is returned. This default value will be null for reference types, zero for numeric types, and false for Boolean types.

The following code uses **`FirstOrDefault`** with an empty sequence:

```
Ingredient[] ingredients = { };

Ingredient element = ingredients.FirstOrDefault();

Console.WriteLine(element == null);
```

This produces the following output:

```
True
```

Notice that no exception is thrown, and the resulting element is set to null. The same behavior is true for the version of **`FirstOrDefault`** that takes a predicate, as the following example shows.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 150},
    new Ingredient {Name = "Flour", Calories = 50},
    new Ingredient {Name = "Butter", Calories = 500}
};

Ingredient element = ingredients.FirstOrDefault(x => x.Calories == 9999);

Console.WriteLine(element == null);
```

This produces the following output:

```
True
```

Query expression usage

There is no keyword for **`FirstOrDefault`** when using the query expression style; however, mixed style can be used.

Last

The **Last** query operator returns the final element in the input sequence. As with **First**, there is an overloaded version that also takes a predicate. This predicate version will return the last element in the sequence that satisfies the predicate. Also as with **First**, an exception will be thrown if the input sequence contains zero elements, or if the predicate cannot be satisfied by any element.

The following code shows the predicate version of **Last**.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 50},
    new Ingredient {Name = "Flour", Calories = 50},
    new Ingredient {Name = "Butter", Calories = 500}
};

Ingredient element = ingredients.Last(x => x.Calories == 50);

Console.WriteLine(element.Name);
```

This produces the following output:

```
Flour
```

Notice here that “Flour” is not the last element in the sequence as a whole, but rather the last element that has 50 calories.

Query expression usage

There is no keyword for **Last** when using the query expression style; however, mixed style can be used.

LastOrDefault

The **LastOrDefault** query operator works in a similar way to **Last**, and also has an overloaded version that takes a predicate.

Like **FirstOrDefault**, if the input sequence is empty or no element satisfies the predicate, rather than an exception being thrown, the default value for the element type is returned instead.

Query expression usage

There is no keyword for **LastOrDefault** when using the query expression style; however, mixed style can be used.

Single

The **`Single`** query operator returns the only element in the input sequence. If the input sequence contains more than one element, an exception is thrown: ”System.InvalidOperationException : Sequence contains more than one element.” If the input sequence contains zero elements, an exception will also be thrown: “System.InvalidOperationException : Sequence contains no elements.”

The following code uses **`Single`** to retrieve the only element in the input sequence.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500}
};

Ingredient element = ingredients.Single();

Console.WriteLine(element.Name);
```

This produces the following output:

```
Sugar
```

There is an overload of **`Single`** that allows a predicate to be specified. The predicate will locate and return the single element that matches it. If there is more than one element in the input sequence that matches the predicate, then an exception will be thrown: “System.InvalidOperationException : Sequence contains more than one matching element.”

The following code shows how to use the predicate version of **`Single`**. Even though there is more than one element in the input sequence, because there is **not** more than one element that matches the predicate, an exception will **not** be thrown.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Butter", Calories = 150},
    new Ingredient {Name = "Milk", Calories = 500}
};

Ingredient element = ingredients.Single(x => x.Calories == 150);

Console.WriteLine(element.Name);
```

This produces the following output:

```
Butter
```

If the predicate is changed (as in the following code) so that it matches more than one element, then an exception will be thrown.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Butter", Calories = 150},
    new Ingredient {Name = "Milk", Calories = 500}
};

Ingredient element = ingredients.Single(x => x.Calories == 500);
```

Notice the predicate in the preceding code will find two ingredients with calories of 500, there will be an exception.

Query expression usage

There is no keyword for **`Single`** when using the query expression style; however, mixed style can be used.

SingleOrDefault

The **`SingleOrDefault`** query operator works in a similar fashion to **`Single`**. However, rather than throwing an exception when there are zero input elements, the default for the type is returned. If the predicate version is being used and no matching elements are found, the default value is again returned, rather than an exception (as happens with **`Single`**). To summarize, for **`SingleOrDefault`** to **not** throw an exception, either zero or one element must be found. Zero found elements will result in a default value, one element found will be returned, more than one element found will result in an exception.

The following code shows the predicate version of **`SingleOrDefault`** being used where there will be zero matching elements found for 9,999 calories.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 50}
};

Ingredient element = ingredients.SingleOrDefault(x => x.Calories == 9999);

Console.WriteLine(element == null);
```

This produces the following output:

```
True
```

Note that no exception is thrown, rather the default value (null) is returned.

Query expression usage

There is no keyword for **`SingleOrDefault`** when using the query expression style; however, mixed style can be used.

ElementAt

The **`ElementAt`** query operator returns the element that exists at the specified position in the input sequence.

The following code selects the third element from the input sequence.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 50}
};

Ingredient element = ingredients.ElementAt(2);

Console.WriteLine(element.Name);
```

This produces the following output:

```
Milk
```

Notice that the value passed to **`ElementAt`** is a zero-based index; for example, to select the first element, the value would be **`0`**.

If the value passed to **`ElementAt`** tries to select an element that is greater than the number of elements in the input sequence, an exception will be thrown: “Index was out of range. Must be non-negative and less than the size of the collection.”

Query expression usage

There is no keyword for **`ElementAt`** when using the query expression style; however, mixed style can be used.

ElementAtOrDefault

The **`ElementAtOrDefault`** query operator works in a similar way to **`ElementAt`**, but rather than throw an exception if the value passed to it exceeds the size of the input sequence, it will return the default value for the input type. The following code demonstrates this.

```
Ingredient[] ingredients =
```

```
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 50}
};

Ingredient element = ingredients.ElementAtOrDefault(4);

Console.WriteLine(element == null);
```

This produces the following output:

```
True
```

Query expression usage

There is no keyword for **`ElementAtOrDefault`** when using the query expression style; however, mixed style can be used.

DefaultIfEmpty

The **`DefaultIfEmpty`** query operator takes an input sequence and does one of two things. If the input sequence contains at least one element, then the input sequence is returned without any changes. If, however, the input sequence contains zero elements, the output sequence returned will not be empty; it will contain a single element with a default value.

The following code shows what happens when the input sequence contains elements.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 50}
};

IEnumerable<Ingredient> query = ingredients.DefaultIfEmpty();

foreach (Ingredient item in query)
{
    Console.WriteLine(item.Name);
}
```

This produces the following output:

```
Sugar
Egg
Milk
```

Note this is exactly the same as the input sequence.

The following code shows the second mode of operation where the input sequence contains zero elements.

```
Ingredient[] ingredients = {};

IEnumerable<Ingredient> query = ingredients.DefaultIfEmpty();

foreach (Ingredient item in query)
{
    Console.WriteLine(item == null);
}
```

This produces the following output:

```
True
```

Notice here that the input sequence was empty, but the output sequence contains a single **`null`** element, a **`null`** being the default value for **`Ingredient`** (which is a reference type).

Query expression usage

There is no keyword for **`DefaultIfEmpty`** when using the query expression style; however, mixed style can be used.

Generation Operators

The generation query operators create sequences for us.

The generation operators differ from the majority of the other standard query operators in two main ways. The first is that they do not take an input sequence; the second is that they are not implemented as extension methods, but rather as plain static methods of the **`Enumerable`** class.

As an example, the following code shows the signature of the **`Empty`** query operator.

```
public static IEnumerable<TResult> Empty<TResult>()
```

Empty

The **`Empty`** query operator creates an empty sequence (zero elements) of a specified type.

The following code shows the creation of empty sequence of **`Ingredient`**. Notice that the type of the empty sequence we want is specified as a generic type parameter of the **`Empty`** method.

```
IEnumerable<Ingredient> ingredients = Enumerable.Empty<Ingredient>();

Console.WriteLine(ingredients.Count());
```

This produces the following output:

```
0
```

Query expression usage

There is no keyword for **Empty** when using the query expression style; however, mixed style can be used.

Range

The **Range** query operator creates a sequence of integer values. When using **Range**, the first parameter specified is the starting number of the range to be generated. The second parameter is the total number of integer elements to generate, starting at the first parameter value.

```
IEnumerable<int> fiveToTen = Enumerable.Range(5, 6);

foreach (int num in fiveToTen)
{
    Console.WriteLine(num);
}
```

This produces the following output:

```
5
6
7
8
9
10
```

Notice that a total of six elements have been generated, as specified by the second parameter in the call to **Range**.

Query expression usage

There is no keyword for **Range** when using the query expression style; however, mixed style can be used.

Repeat

The **Repeat** query operator repeats a specified integer a specified number of times, as the following code shows.

```
IEnumerable<int> nums = Enumerable.Repeat(42, 5);

foreach (int num in nums)
{
    Console.WriteLine(num);
}
```

This produces the following output:

```
42
42
42
42
42
```

Query expression usage

There is no keyword for **Repeat** when using the query expression style; however, mixed style can be used.

Quantifier Operators

The quantifier query operators take an input sequence (or in the case of **SequenceEqual**, two input sequences), evaluate it, and return a single Boolean result.

Contains

The **Contains** query operator evaluates the elements in the input sequence and returns **true** if a specified value exists.

```
int[] nums = {1, 2, 3};

bool isTwoThere = nums.Contains(2);
bool isFiveThere = nums.Contains(5);

Console.WriteLine(isTwoThere);
Console.WriteLine(isFiveThere);
```

This produces the following output:

```
True
False
```

Query expression usage

There is no keyword for **`Contains`** when using the query expression style; however, mixed style can be used.

Any

There are two overloads of the **`Any`** query operator. The first version simply returns **`true`** if the input sequence contains at least one element. The second version of **`Any`** takes a predicate as a parameter and returns **`true`** if at least one element in the input sequence satisfies the predicate.

The following code demonstrates the simple version of **`Any`**.

```
int[] nums = { 1, 2, 3 };

IEnumerable<int> noNums = Enumerable.Empty<int>();

Console.WriteLine( nums.Any() );
Console.WriteLine( noNums.Any() );
```

This produces the following output:

```
True
False
```

The following code shows the predicate overload of **`Any`** to determine if a sequence contains any even numbers.

```
int[] nums = { 1, 2, 3 };

bool areAnyEvenNumbers = nums.Any(x => x % 2 == 0);

Console.WriteLine(areAnyEvenNumbers);
```

This produces the following output:

```
True
```

Query expression usage

There is no keyword for **`Any`** when using the query expression style; however, mixed style can be used.

All

The **All** query operator takes a predicate and evaluates the elements in the input sequence to determine if every element satisfies this predicate.

The following code checks that a sequence of ingredients constitutes a low-fat recipe.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 150},
    new Ingredient {Name = "Flour", Calories = 50},
    new Ingredient {Name = "Butter", Calories = 400}
};

bool isLowFatRecipe = ingredients.All(x => x.Calories < 200);

Console.WriteLine(isLowFatRecipe);
```

This produces the following output:

```
False
```

Note: As soon as an element is found that does not satisfy the predicate, All() returns false and does not examine any subsequent elements.

Query expression usage

There is no keyword for **All** when using the query expression style; however, mixed style can be used.

SequenceEqual

The **SequenceEqual** query operator compares two sequences to see if they both have exactly the same elements in the same order.

The following code compares two identical sequences:

```
IEnumerable<int> sequence1 = new[] {1, 2, 3};
IEnumerable<int> sequence2 = new[] {1, 2, 3};

bool isSeqEqual = sequence1.SequenceEqual(sequence2);

Console.WriteLine(isSeqEqual);
```

This produces the following output:

```
True
```

If the two sequences being compared contain the same elements, but in different orders, then **`SequenceEqual`** will return false, as the following code demonstrates.

```
IEnumerable<int> sequence1 = new[] { 1, 2, 3 };
IEnumerable<int> sequence2 = new[] { 3, 2, 1 };

bool isSeqEqual = sequence1.SequenceEqual(sequence2);

Console.WriteLine(isSeqEqual);
```

This produces the following output:

```
False
```

Notice here that even though the sequences contain the same values (1, 2, 3) the order is different, so **`SequenceEqual`** returns **`false`**.

Query expression usage

There is no keyword for **`SequenceEqual`** when using the query expression style; however, mixed style can be used.

Aggregate Operators

The aggregate operators take an input sequence and return a single scalar value. The returned value is not one of the elements from the input sequence, but rather some derived value computed from the elements in the input sequence.

The aggregate query operators are:

- **`Count`**
- **`LongCount`**
- **`Sum`**
- **`Min`**
- **`Max`**
- **`Average`**
- **`Aggregate`**

Count

The **`Count`** query operator simply returns the number of elements contained in the input sequence. There are two overloads, one of which accepts a predicate.

The following example shows the non-predicate version.

```
int[] nums = {1, 2, 3};

int numberOfElements = nums.Count();

Console.WriteLine(numberOfElements);
```

This produces the following output:

```
3
```

When a predicate is supplied, the count is restricted to those elements satisfying the predicate. The following code shows a predicate being used to count all the even numbers in the input sequence.

```
int[] nums = { 1, 2, 3 };

int numberOfEvenElements = nums.Count(x => x % 2 == 0);

Console.WriteLine(numberOfEvenElements);
```

This produces the following output:

```
1
```

Tip: Using Any() instead of Count() != 0 usually conveys the intent of the code better, and in some circumstances, can perform better.

Query expression usage

There is no keyword for **`Count`** when using the query expression style; however, mixed style can be used.

LongCount

The **`LongCount`** query operator provides the same features as **`Count`** (including a predicate overload), but instead of returning an **`int`**, it returns a **`long`**, so it can be used with very long input sequences.

Query expression usage

There is no keyword for **`LongCount`** when using the query expression style; however, mixed style can be used.

Sum

The **Sum** query operator adds together all the elements of the input sequence and returns the total result.

The following code demonstrates adding together a sequence of **int**s.

```
int[] nums = { 1, 2, 3 };

int total = nums.Sum();

Console.WriteLine(total);
```

This produces the following output:

```
6
```

The **Sum** query operator is implemented as individual separate extension methods for **IEnumerable<T>** (for example, **IEnumerable<long>**) for each of the numeric types (and nullable equivalents). This means that in order to use **Sum** on non-numeric sequences, an overload must be used to select a numeric value from the non-numeric input elements. The following code shows how to **Sum** the calorie values of a sequence of **Ingredient**.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 150},
    new Ingredient {Name = "Flour", Calories = 50},
    new Ingredient {Name = "Butter", Calories = 400}
};

int totalCalories = ingredients.Sum(x => x.Calories);

Console.WriteLine(totalCalories);
```

This produces the following output:

```
1200
```

Query expression usage

There is no keyword for **Sum** when using the query expression style; however, mixed style can be used.

Average

The **Average** query operator works on a sequence of numeric values and calculates the average.

The following code shows **Average** in use.

```
int[] nums = { 1, 2, 3 };

var avg = nums.Average();

Console.WriteLine(avg);
```

This produces the following output:

```
2
```

Notice in the preceding code the type for total has not been explicitly set; instead, **var** is being used. Depending on the numeric input type, **Average** will return different output numeric types. For example, the following code will not compile because the implementation of **Average** for **IEnumerable<int>** returns a **double**.

```
int[] nums = { 1, 2, 3 };

int avg = nums.Average();  // this line causes compilation failure
```

Average will return either a **double**, **float**, or **decimal** return type (or the nullable versions of these.)

As with **Sum**, **Average** can be used on an input sequence of non-numeric values using an overload, which allows a numeric value to be selected, as shown in the following code.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 150},
    new Ingredient {Name = "Flour", Calories = 50},
    new Ingredient {Name = "Butter", Calories = 400}
};

var avgCalories = ingredients.Average(x => x.Calories);

Console.WriteLine(avgCalories);
```

This produces the following output:

```
240
```

Query expression usage

There is no keyword for **Average** when using the query expression style; however, mixed style can be used.

Min

The **Min** query operator returns the smallest element from the input sequence, as the following code demonstrates.

```
int[] nums = { 3, 2, 1 };

var smallest = nums.Min();

Console.WriteLine(smallest);
```

This produces the following output:

```
1
```

An overload of **Min** also allows the selector to be specified, as the following code shows.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Egg", Calories = 100},
    new Ingredient {Name = "Milk", Calories = 150},
    new Ingredient {Name = "Flour", Calories = 50},
    new Ingredient {Name = "Butter", Calories = 400}
};

var smallestCalories = ingredients.Min(x => x.Calories);

Console.WriteLine(smallestCalories);
```

This produces the following output:

```
50
```

Query expression usage

There is no keyword for **Min** when using the query expression style; however, mixed style can be used.

Max

The **Max** query operator complements the **Min** operator and returns the smallest value from the input sequence, as the following code demonstrates.

```
int[] nums = { 1, 3, 2 };

var largest = nums.Max();

Console.WriteLine(largest);
```

This produces the following output:

```
3
```

As with **Min**, there is an overload of **Max** that allows a selector to be specified.

Query expression usage

There is no keyword for **Max** when using the query expression style; however, mixed style can be used.

Aggregate

The previous aggregate operators all perform a specific aggregation; the **Aggregate** query operator is a more advanced operator that allows custom aggregations to be defined and executed against an input sequence.

There are two main versions of **Aggregate**: one that allows a starting "seed" value to be specified, and one that uses the first element in the sequence as the seed value.

All versions of **Aggregate** require an "accumulator function" to be specified. The accumulator, as its name suggests, accumulates a result as each element in the input sequence is processed. The accumulator function is passed the value of the current element being processed and the current accumulated value.

The following code demonstrates how to simulate the **Sum** operator using **Aggregate**.

```
int[] nums = { 1, 2, 3 };

var result = nums.Aggregate(0,
    (currentElement, runningTotal) => runningTotal + currentElement);

Console.WriteLine(result);
```

This produces the following output:

```
6
```

The preceding code uses an overload of **`Aggregate`** that specifies the initial seed value, in this example a seed of zero.

Query expression usage

There is no keyword for **`Aggregate`** when using the query expression style; however, mixed style can be used.

Joining Operators

The joining query operators take two input sequences, combine them in some way, and output a single sequence.

Join

The **`Join`** query operator takes an initial input sequence (the “outer sequence”), and to this outer sequence a second “inner sequence” is introduced. The output from the **`Join`** query operator is a flat (i.e. non-hierarchical) sequence.

The **`Join`** operator takes a number of parameters:

- **`IEnumerable<TInner> inner`** – the inner sequence.
- **`Func<TOuter, TKey> outerKeySelector`** – what key to join on in outer sequence elements.
- **`Func<TInner, TKey> innerKeySelector`** - what key to join on in inner sequence elements.
- **`Func<TOuter, TInner, TResult> resultSelector`** – What the output elements will look like

Note: There is also an overload that allows a specific IEqualityComparer to be used.

The following code shows the **`Join`** query operator in use. Note the explicit type definitions in the key selection lambdas; this is for demonstration code clarity. For example: **`(Recipe outerKey) => outerKey.Id`** could be simplified to **`outerKey=> outerKey.Id`**.

```
Recipe[] recipes = // outer sequence
{
    new Recipe {Id = 1, Name = "Mashed Potato"},
    new Recipe {Id = 2, Name = "Crispy Duck"},
```

```
    new Recipe {Id = 3, Name = "Sachertorte"}
};

Review[] reviews = // inner sequence
{
    new Review {RecipeId = 1, ReviewText = "Tasty!"},
    new Review {RecipeId = 1, ReviewText = "Not nice :("},
    new Review {RecipeId = 1, ReviewText = "Pretty good"},
    new Review {RecipeId = 2, ReviewText = "Too hard"},
    new Review {RecipeId = 2, ReviewText = "Loved it"}
};


var query = recipes // recipes is the outer sequence
        .Join(
                reviews, // reviews is the inner sequence
                (Recipe outerKey) => outerKey.Id, // the outer key
                (Review innerKey) => innerKey.RecipeId, // the inner key
                // choose the shape ("projection") of resulting elements
                (recipe, review) => recipe.Name + " - " + review.ReviewText);

foreach (string item in query)
{
    Console.WriteLine(item);
}
```

This produces the following output:

```
Mashed Potato - Tasty!
Mashed Potato - Not nice :(
Mashed Potato - Pretty good
Crispy Duck - Too hard
Crispy Duck - Loved it
```

Notice in this output that “Sachertorte” does not appear. This is because **`Join`** performs a left join, so elements in the outer sequence that have no matches in the inner sequence will not be included in the output sequence.

Query expression usage

The **join** keyword is used when using the query expression style. The following code shows how to perform an inner join using the query expression style. See Chapter 2, “Fluent and Query Expression Styles for usage of the **join** keyword.

```
var query = from recipe in recipes
    join review in reviews on recipe.Id equals review.RecipeId
    select new // anonymous type
            {
                RecipeName = recipe.Name,
                RecipeReview = review.ReviewText
            };
```

Tip: Consider using the query expression style when performing joins, as it is usually more readable.

GroupJoin

The **GroupJoin** query operator joins an inner and outer sequence, just as with the **Join** query operator; however, **GroupJoin** produces a hierarchical result sequence. The output sequence is essentially a sequence of groups, with each group able to contain the sequence of items from the inner sequence that belong to that group.

The **GroupJoin** operator takes a number of parameters:

- **IEnumerable<TInner> inner**: The inner sequence.
- **Func<TOuter, TKey> outerKeySelector**: What key to join on in outer sequence elements.
- **Func<TInner, TKey> innerKeySelector**: What key to join on in inner sequence elements.
- **Func<TOuter, IEnumerable<TInner>, TResult> resultSelector**: What the output groups will look like.

Notice the final parameter here is different from **Join**’s (**Func<TOuter, TInner, TResult> resultSelector**). Rather than a single **TInner**, **GroupJoin** has **IEnumerable<TInner>**. This **IEnumerable<TInner>** contains all the elements from the inner sequence that match the outer sequence key; it is essentially all the inner sequence elements that belong in the group.

The following code shows **GroupJoin** in use. Notice the result selector is creating an anonymous type containing the group “key” (i.e. the recipe name), and a list of the reviews that belong to that recipe.

```
Recipe[] recipes =
{
    new Recipe {Id = 1, Name = "Mashed Potato"},
```

```
    new Recipe {Id = 2, Name = "Crispy Duck"},
    new Recipe {Id = 3, Name = "Sachertorte"}
};

Review[] reviews =
{
    new Review {RecipeId = 1, ReviewText = "Tasty!"},
    new Review {RecipeId = 1, ReviewText = "Not nice :("},
    new Review {RecipeId = 1, ReviewText = "Pretty good"},
    new Review {RecipeId = 2, ReviewText = "Too hard"},
    new Review {RecipeId = 2, ReviewText = "Loved it"}
};


var query = recipes
        .GroupJoin(
                reviews,
                (Recipe outerKey) => outerKey.Id,
                (Review innerKey) => innerKey.RecipeId,
                (Recipe recipe, IEnumerable<Review> rev =>
                    new {
                            RecipeName = recipe.Name,
                            Reviews = revs
                        }
                );


foreach (var item in query)
{
    Console.WriteLine("Reviews for {0}", item.RecipeName);

    foreach (var review in item.Reviews)
    {
        Console.WriteLine(" - {0}", review.ReviewText);
    }
}
```

This produces the following output:

```
Reviews for Mashed Potato
 - Tasty!
 - Not nice :(
 - Pretty good
Reviews for Crispy Duck
 - Too hard
 - Loved it
Reviews for Sachertorte
```

Notice the “Sachertorte” group has been included even though it has no related reviews.

Query expression usage

The **join** keyword is used when using the query expression style; see Chapter 2, “Fluent and Query Expression Styles,” for usage of the **join** keyword. The following code shows a group join using the query expression style.

```
var query =
    from recipe in recipes
    join review in reviews on recipe.Id equals review.RecipeId
    into reviewGroup
    select new // anonymous type
            {
                RecipeName = recipe.Name,
                Reviews = reviewGroup // collection of related reviews
            };
```

Zip

The **Zip** query operator joins two sequences together, but unlike **Join** and **GroupJoin**, does not perform joining based on keys, and the concepts of inner and outer sequences do not apply. Instead, **Zip** simply interleaves each element from the two input sequences together, one after the other, much like joining two sides of a zipper on an item of clothing.

When using **Zip**, the resulting shape of the output elements is specified. In the following example, the result elements are of type **Ingredient**.

```
string[] names = {"Flour", "Butter", "Sugar"};
int[] calories = {100, 400, 500};

IEnumerable<Ingredient> ingredients =
    names.Zip(calories, (name, calorie) =>
        new Ingredient
        {
            Name = name,
            Calories = calorie
        });

foreach (var item in ingredients)
{
    Console.WriteLine("{0} has {1} calories", item.Name, item.Calories);
}
```

This produces the following output:

```
Flour has 100 calories
Butter has 400 calories
Sugar has 500 calories
```

If there is a greater number of elements in one of the two input sequences, these extra elements are ignored and do not appear in the output.

Query expression usage

There is no keyword for **`Zip`** when using the query expression style; however, mixed style can be used.

Chapter 4 LINQ to XML

The building blocks that comprise LINQ to XML allow the creation, modification, querying, and saving of XML data.

At a high level, LINQ to XML consists of two key architectural elements that are designed to work well together:

- XML document object model.
- Additional LINQ to XML query operators.

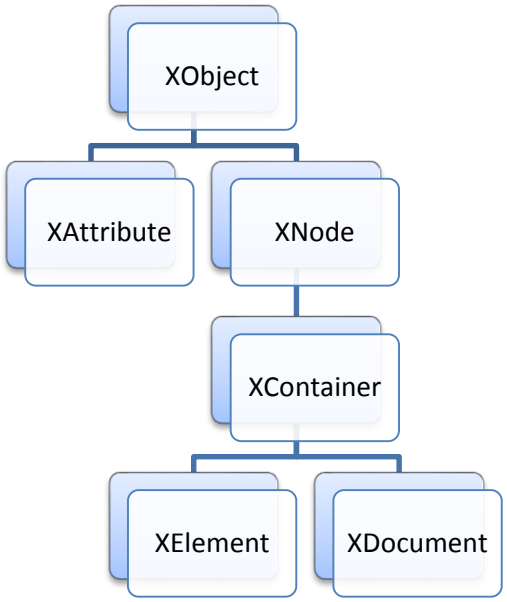
The XML document object model in LINQ to XML (the **X-DOM**) is not the W3C DOM; it is a set of .NET types that represent in-memory XML data. While the types that make up the X-DOM can be used independently from any LINQ queries, they are designed to be LINQ-friendly and facilitate easy interaction when using LINQ.

X-DOM Overview

The types that make up the X-DOM are used to represent the structure of an XML document or fragment. They represent an in-memory model of the structure and content of XML data.

Key X-DOM types

There are numerous types that make up the X-DOM object model. Some of the keys types are shown in the following class hierarchy diagram.



XContainer

An **XContainer** represents the idea of an item in the XML hierarchy that may have zero, one, or many children **XNode** objects. **XContainer** is an abstract class, so it cannot be instantiated directly; instead, either **XElement** or **XDocument** can be used.

XElement and XDocument

An **XElement** represents an element in an XML document such as an **<ingredient>** element. An **XElement** can contain a value such as “Butter,” and may contain zero, one, or many **XAttributes**. By virtue of it inheriting from **XContainer**, it may also have child **XNode** objects that belong to it.

An **XDocument** wraps a root **XElement** to provide the additional capability of XML declaration items such as XML version and encoding, as well as other “header” items, such a DTD for the XML document. Unless these kind of additional capabilities are required, using **XElement** is usually sufficient.

XAtrribute

An **XAttribute** represents an XML attribute on an XML element. For example, the **calories** attribute in the following XML would be represented as an **XAttribute** object: **<ingredient calories="500">**

XNode

While an **XNode** object has the concept of belonging to a parent (**XElement**) node that it can navigate to, unlike **XContainer** (and its subclasses **XElement** and **XDocument**), it has no concept of having any child nodes.

XName

XName is not part of the hierarchy, but it is used in many methods when wanting to specify node(s) to locate in the XML document hierarchy. It is used to represent the name of an XML element or attribute. In addition to representing elements as simple name strings, **XName** also contains logic for working with XML namespaced names.

Creating an X-DOM

There are a number of ways to arrive at an in-memory X-DOM, including loading XML files and instantiation through code.

Parsing Strings and Loading Files

To create an **XElement** from a **string**, the static **Parse** method can be used, as the following code shows.

```
string xmlString = @"<ingredients>
                        <ingredient>Sugar</ingredient>
                        <ingredient>Milk</ingredient>
                        <ingredient>Butter</ingredient>
                    </ingredients>";

XElement xdom = XElement.Parse(xmlString);

Console.WriteLine(xdom);
```

This produces the following output:

```
<ingredients>
  <ingredient>Sugar</ingredient>
  <ingredient>Milk</ingredient>
  <ingredient>Butter</ingredient>
</ingredients>
```

To create an **`XElement`** from a physical XML file, the **`XElement.Load`** method can be used with a path specifying the file to be loaded.

Manual Procedural Creation

The types that make up the X-DOM can be instantiated, just as with regular .NET types. To create an X-DOM that represents the previous **`<ingredients>`** XML, the following code can be written.

```
XElement ingredients = new XElement("ingredients");

XElement sugar = new XElement("ingredient", "Sugar");
XElement milk = new XElement("ingredient", "Milk");
XElement butter = new XElement("ingredient", "Butter");

ingredients.Add(sugar);
ingredients.Add(milk);
ingredients.Add(butter);

Console.WriteLine(ingredients);
```

This produces the following output:

```
<ingredients>
  <ingredient>Sugar</ingredient>
  <ingredient>Milk</ingredient>
  <ingredient>Butter</ingredient>
</ingredients>
```

Note in the preceding code that even though the XML structure is simple, this manual creation style does not make it particularly easy to understand what the XML structure that is being created is. An alternative approach is to use **functional construction**.

Functional Construction

Using the functional construction style can make the resulting XML structure easier to relate to when reading the code.

The following code shows the same **<ingredients>** XML created, this time using functional construction.

```
XElement ingredients =
    new XElement("ingredients",
        new XElement("ingredient", "Sugar"),
        new XElement("ingredient", "Milk"),
        new XElement("ingredient", "Butter")
    );

Console.WriteLine(ingredients);
```

Notice in the preceding code, it is much easier for the reader to form a mental picture of what the XML structure is.

Notice the overload of **XElement** constructor in the preceding code is one which allows any number of content objects to be specified after the **XElement** name ("ingredients"). The signature of this version of the constructor is: **public XElement(XName name, params object[] content)**.

Creation via Projection

One of the benefits of the functional construction style is that results from LINQ queries can be projected into an X-DOM.

The following code adapts the preceding code to populate the X-DOM with the result of a LINQ query.

```
Ingredient[] ingredients =
{
    new Ingredient {Name = "Sugar", Calories = 500},
    new Ingredient {Name = "Milk", Calories = 150},
    new Ingredient {Name = "Butter", Calories = 200}
};

XElement ingredientsXML =
    new XElement("ingredients",
        from i in ingredients
        select new XElement("ingredient", i.Name,
                            new XAttribute("calories", i.Calories))
    );

Console.WriteLine(ingredientsXML);
```

This produces the following output:

```
<ingredients>
  <ingredient calories="500">Sugar</ingredient>
  <ingredient calories="150">Milk</ingredient>
  <ingredient calories="200">Butter</ingredient>
</ingredients>
```

Notice in the preceding code the addition of an **`XAttribute`** to each **<ingredient>** representing the calories. Again, this is possible because the constructor takes any number of content objects, even when those objects are of different types. In the preceding code, this allows the text content of the **`XElement`** to be set (for example, “Sugar”) at the same time an attribute is added.

Querying X-DOM with LINQ

Querying and navigating an X-DOM is implemented in two main ways. The first of these are the methods that belong to the X-DOM types themselves. The second is a set of additional query operators (i.e. extension methods) that are defined in the **`System.Xml.Linq.Extensions`** class. The methods of the X-DOM types often return **`IEnumerable`** sequences. The additional LINQ-to-XML query operators can then do further work with these sequences.

Note: The standard query operators can also be combined with the XML query operators.

As a simple example, the following code demonstrates both implementations.

```
XElement xmlData = XElement.Load("recipes.xml");

// Here Descendants() is an instance method of XContainer
var allrecipes = xmlData.Descendants("recipe");

// Here Descendants() is a query operator extension method
var allIngredients = allrecipes.Descendants("ingredient");
```

In the preceding code, the first execution of the **Descendants** method belongs to **XElement**. More specifically, the method belongs to **XContainer**, from which **XElement** inherits. This method returns an **IEnumerable<XElement>** into the variable **allRecipes**.

The second **Descendants** method call is a call to the LINQ-to-XML query operator extension method that has the signature: **public static IEnumerable<XElement> Descendants<T>(this IEnumerable<T> source, XName name) where T : XContainer**.

Finding Child Nodes

There are a number of properties and methods (in some cases both instance and query operator extension methods) that allow the locating of child nodes.

The following tables outline the methods and properties available for finding children.

Finding Child Nodes

Method(parameter) / Property	Returns	Operates On
FirstNode	XNode	XContainer
LastNode	XNode	XContainer
Element(XName)	XElement	XContainer
Nodes()	IEnumerable<XNode>	XContainer, IEnumerable<XContainer>
DescendantNodes()	IEnumerable<XNode>	XContainer, IEnumerable<XContainer>

Method(parameter) / Property	Returns	Operates On
Elements()	IEnumerable<XElement>	XContainer, IEnumerable<XContainer>

Method(parameter) / Property	Returns	Operates On
Elements(XName)	IEnumerable<XElement>	XContainer, IEnumerable<XContainer>
Descendants()	IEnumerable<XElement>	XContainer, IEnumerable<XContainer>
Descendants(XName)	IEnumerable<XElement>	XContainer, IEnumerable<XContainer>
HasElements	Boolean	XElement
DescendantNodesAndSelf()	IEnumerable<XNode>	XElement, IEnumerable<XElement>
DescendantsAndSelf()	IEnumerable<XElement>	XElement, IEnumerable<XElement>
DescendantsAndSelf(XName)	IEnumerable<XElement>	XElement, IEnumerable<XElement>

The following code demonstrates how to retrieve the first **<ingredient>** element that has any child elements. Here, **Descendants** is used to retrieve all ingredients, in conjunction with the standard query operator **First**. The predicate supplied to **First** is restricting the returned element to the first one that has any child elements by using the XElement's **HasElements** property.

```
XElement xmlData = XElement.Load("recipes2.xml");

XElement firstIngredientWithSubElements = xmlData.Descendants("ingredient")
                                                 .First(x => x.HasElements);
```

Finding Parent Nodes

XDOM types that inherit from **`XNode`** contain a **`Parent`** property that returns the parent **`XElement`**.

Finding Parent Nodes

Method(parameter) / Property	Returns	Operates On
`Parent`	`XElement`	`XNode`
`Ancestors()`	`IEnumerable<XElement>`	`XNode`, `IEnumerable<XNode>`
`Ancestors( XName )`	`IEnumerable<XElement>`	`XNode`, `IEnumerable<XNode>`
`AncestorsAndSelf()`	`IEnumerable<XElement>`	`XElement`, `IEnumerable<XElement>`
`AncestorsAndSelf( XName )`	`IEnumerable<XElement>`	`XElement`, `IEnumerable<XElement>`

The following example shows the use of the **`Parent`** method to move up the XML hierarchy. This example finds the first ingredient that has child elements, then moves one level up to get the **`<ingredients>`** element. Then, another call to **`Parent`** moves another level up to get to the **`<recipe>`**.

```
XElement xmlData = XElement.Load("recipes2.xml");

XElement recipe = xmlData.Descendants("ingredient")
                         .First(x => x.HasElements)
                         .Parent // <ingredients>
                         .Parent; // <recipe name="Cherry Pie">
```

Finding Peer Nodes

The following table shows the methods defined in the **XNode** class for working with nodes at the same level.

Finding Peer Nodes

Method(parameter) / Property	Returns
IsBefore(XNode)	Boolean
IsAfter(XNode)	Boolean
PreviousNode	XNode
NextNode	XNode
NodesBeforeSelf()	IEnumerable<XNode>
NodesAfterSelf()	IEnumerable<XNode>
ElementsBeforeSelf()	IEnumerable<XElement>
ElementsBeforeSelf(XName)	IEnumerable<XElement>
ElementsAfterSelf()	IEnumerable<XElement>
ElementsAfterSelf(XName)	IEnumerable<XElement>

The following code shows the use of the **NextNode** and **IsBefore** methods. This code also demonstrates how the standard query operators (such as **Skip**) can be used in LINQ-to-XML queries.

```
XElement xmlData = XElement.Load("recipes2.xml");

var applePieIngredients =
   xmlData.Descendants("recipe")
          .First(x => x.Attribute("name").Value == "Apple Pie")
          .Descendants("ingredient");
```

```
var firstIngredient = applePieIngredients.First(); // Sugar
var secondIngredient = firstIngredient.NextNode; // Apples
var lastIngredient = applePieIngredients.Skip(2).First(); // Pastry

var isApplesBeforeSugar =
        secondIngredient.IsBefore(firstIngredient); // false
```

Finding Attributes

An **XElement** may have a number of **XAttributes**. The following table shows the methods defined in **XElement** for working with attributes that belong to it.

Finding XElement Attributes

Method(parameter) / Property	Returns
HasAttributes	Boolean
Attribute(XName)	XAttribute
FirstAttribute	XAttribute
LastAttribute	XAttribute
Attributes()	IEnumerable<XAttribute>
Attributes(XName)	IEnumerable<XAttribute>

The following code demonstrates the use of the **Attribute(XName)** method to locate a specific **XAttribute** and get its value by reading its **Value** property. This example also shows the use of the **FirstAttribute** method to get an element's first declared attribute, and also how to combine standard query operators such as **Skip** to query the **IEnumerable<XAttribute>** provided by the **Attributes** method.

```
var xml = @"
<ingredients>
    <ingredient name='milk' quantity='200' price='2.99'  />
```

```
    <ingredient name='sugar' quantity='100' price='4.99'  />
    <ingredient name='safron' quantity='1' price='46.77'  />
</ingredients>";

XElement xmlData = XElement.Parse(xml);

XElement milk =
    xmlData.Descendants("ingredient")
           .First(x => x.Attribute("name").Value == "milk");


XAttribute nameAttribute = milk.FirstAttribute; // name attribute

XAttribute priceAttribute = milk.Attribute("price");

string priceOfMilk = priceAttribute.Value; // 2.99

XAttribute quantity = milk.Attributes()
                          .Skip(1)
                          .First(); // quantity attribute
```

Chapter 5 Interpreted Queries

Overview

Up until this point in the book, we have focused mainly on local queries. Local queries are those that operate on **`IEnumerable<T>`** sequences. Local queries result in the query operators that are defined in the **`System.Linq.Enumerable`** class being executed. In this way, executing local queries results in specific code running (from the query operators in the **`Enumerable`** class) as defined at compile time.

Interpreted queries, on the other hand, describe the "shape" of the query that instead is interpreted at runtime—hence the name "interpreted." Interpreted queries operate on **`IQuerable<T>`** sequences, and when writing LINQ queries, the query operators resolve to methods defined in the **`System.Linq.Queryable`** class rather than the **`System.Linq.Enumerable`** class as with local queries.

Another way to think about the difference between local (**`IEnumerable<T>`**) queries and interpreted (**`IQueryable<T>`**) queries is that the local query operators contain the actual query implementation code that gets executed, whereas the interpreted query operators do not. With interpreted queries, the actual query code that gets executed and returns some result is defined in a query provider. The query provider is passed the description of the query that needs to be executed; it executes the query (for example, executing some SQL against a database) and returns the result.

Expression trees

The query operators that work on **`IQueryable<T>`** differ from their equivalent versions for local queries.

Local (**`IEnumerable<T>`**) versions of query operators take **delegates** as parameters (often described using a lambda expression). In contrast, interpreted queries take **expression trees** as parameters.

Because C# will implicitly convert a lambda expression when used as a query operator parameter to an expression, the same lambda expression can be used regardless of whether the query operator is a local or remote one. The following code shows both the local (**`IEnumerable<T>`**) version of the **`Where`** query operator and the interpreted (**`IQueryable<T>`**) version being used. Notice that the lambda expression is the same in both cases.

```
IQueryable<int> interpretedQuery = remoteData.Where(x => x > 100);
IEnumerable<int> localQuery = localData.Where(x => x > 100);
```

The following code shows the method signatures for the **Where** query operator for both the local and interpreted versions.

```
// Local version of Where from System.Linq.Enumerable
public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource>
source, Func<TSource, bool> predicate)

// Interpreted version of Where from System.Linq.Queryable
public static IQueryable<TSource> Where<TSource>(this IQueryable<TSource>
source, Expression<Func<TSource, bool>> predicate)
```

Notice in the preceding code that the local version takes a predicate parameter of type **Func<TSource, bool>**. Contrast this with the interpreted version where the predicate parameter type is **Expression<Func<TSource, bool>>**. This is the expression tree that the query provider can turn into some meaningful code execution, such as generating an equivalent SQL statement and executing against a database.

Because **IQueryable<T>** is a subtype of **IEnumerable<T>**, the compiler has to choose whether to use the local or remote version of a give query operator. The compiler rules dictate that the more specific version will be chosen. Because **IQueryable<T>** is more specific than **IEnumerable<T>**, the compiler will choose the interpreted versions of the query operators (as defined in **System.Linq.Queryable**) when dealing with interpreted queries, and local versions of query operators (as defined in **System.Linq.Enumerable**) when dealing with local queries. This fact (and C#'s implicit conversion of lambda expressions to **Expression<T>**) means that LINQ queries can provide a unified API regardless of whether the query is a local one or a remote interpreted query.

Note: Not all of the standard query operators will be supported by all interpreted query providers.

Query providers

Microsoft™ has provided pre-built query providers that operate with databases, such as the older LINQ to SQL and the newer Entity Framework.

There are also numerous third party or open source query providers including:

- LINQ to Twitter
- LINQ to flickr
- LINQ to JSON

These examples serve to demonstrate the powerful and flexible nature of LINQ interpreted queries. Remote data stores of different types can be queried using a common set of LINQ query operators.

In addition to LINQ providers built by others, it is also possible to create them ourselves, though the work required may be quite involved.

Entity Framework

As an example of using LINQ to query a database using Entity Framework, imagine a SQL Server database that contains the following entities (tables).

```
public class Recipe
{
    public int ID { get; set; }
    public string Name { get; set; }
    public virtual ICollection<Review> Reviews { get; set; }
}

public class Review
{
    public int ID { get; set; }
    public int RecipeID { get; set; }
    public string ReviewText { get; set; }
    public virtual Recipe Recipe { get; set; }
}
```

In the preceding code, the model represents that a recipe can have multiple reviews.

Tip: To learn more about using Entity Framework, be sure to check out the Entity Framework Code First Succinctly eBook from Syncfusion.

Now that the entities are defined, a data context class can be defined, as the following code demonstrates.

```
public class RecipeDbContext : DbContext
{
    public RecipeDbContext() : base ("RecipeDbConnectionString")
    {
    }

    public DbSet<Recipe> Recipes { get; set; }
    public DbSet<Review> Reviews { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
    }
}
```

A database initializer is created to always recreate the database and populate it with some initial demo data, as the following code shows.

```
public class RecipeDbInitializer :
               System.Data.Entity.DropCreateDatabaseAlways<RecipeDbContext>
{
    protected override void Seed(RecipeDbContext context)
    {
        context.Recipes.Add(new Recipe { Name = "Cherry Pie" });
        context.Recipes.Add(new Recipe {Name = "Apple Pie"});
        context.SaveChanges();


        context.Reviews.Add(new Review {ReviewText = "Quite nice",
                                        RecipeID = 1});
        context.Reviews.Add(new Review {ReviewText = "I hate cherries!",
                                        RecipeID = 1});
        context.Reviews.Add(new Review {ReviewText = "Ok",
                                        RecipeID = 1});
        context.Reviews.Add(new Review {ReviewText = "Not too bad",
                                        RecipeID = 2});
        context.SaveChanges();
    }
}
```

Now that there is an EF model and a database containing some demo data, it can be queried using LINQ via the **`RecipeDbContext`**. The following code shows how to use fluent style syntax to get all the recipes sorted by the recipe name.

```
RecipeDbContext ctx = new RecipeDbContext();

IQueryable<Recipe> allRecipies = ctx.Recipes
                                    .OrderBy(x => x.Name);

foreach (var recipe in allRecipies)
{
    Console.WriteLine(recipe.Name);
}
```

This produces the following output:

```
Apple Pie
Cherry Pie
```

Just as with local queries, query expression syntax can also be used with interpreted queries.

The following code shows the combination of using query expression syntax against an Entity Framework model and projecting into an X-DOM using functional construction.

```
RecipeDbContext ctx = new RecipeDbContext();

XElement xml = new XElement("recipes",
    from rec in ctx.Recipes.Include(x => x.Reviews).ToList()
    select
        new XElement("recipe",
        new XAttribute("name", rec.Name),
        new XAttribute("id", rec.ID),
            new XElement("reviews",
                from rev in rec.Reviews
                select
                    new XElement("review", rev.ReviewText))
        ));


Console.WriteLine(xml);
```

This produces the following output:

```
<recipes>
  <recipe name="Cherry Pie" id="1">
    <reviews>
      <review>Quite nice</review>
      <review>I hate cherries!</review>
      <review>Ok</review>
    </reviews>
  </recipe>
  <recipe name="Apple Pie" id="2">
    <reviews>
      <review>Not too bad</review>
    </reviews>
  </recipe>
</recipes>
```

In the preceding code, there are a couple of noteworthy items. The first is the Entity Framework **`Include`** extension method. This instructs Entity Framework to retrieve all the related reviews for each of the recipes. Without this, Entity Framework will end up submitting multiple select queries to the database, rather than just a single **`SELECT`**.

The second item of note is the explicit call to **`ToList()`**. This is required to retrieve the results into local entity objects before the X-DOM projection takes place. Without this, an exception will be thrown: “System.NotSupportedException : Only parameterless constructors and initializers are supported in LINQ to Entities.” This is because **`XElement`** does not have a default parameterless constructor defined.

Other than these Entity Framework specific items, the LINQ query follows the same pattern and syntax it would if we were working with a local data.

Chapter 6 Parallel LINQ

Overview

The area of parallel programming is vast and potentially complicated. This chapter introduces parallel LINQ (**PLINQ**) as a method of reducing processing time when executing LINQ queries.

PLINQ is a higher level abstraction that sits above various lower level .NET multithreading related components and aims to abstract away the lower-level details of multithreading, while at the same time offering the familiar general LINQ query semantics.

Note: Because PLINQ is at a higher level of abstraction, the programmer still needs to have basic working knowledge of multithreaded programming.

When converting a LINQ query to a PLINQ query, PLINQ takes care of the following lower-level aspects of parallelization:

- Split the query work (input sequence) into a number of smaller sub-segments.
- Execute query code against each sub-segment on different threads.
- Once all sub-segments have been processed, reassemble all the results from the sub-segments back into a single output sequence.

In this way, PLINQ **may** help reduce overall query processing time by utilizing multiple cores of the machine on which it is executing. It should also be noted that PLINQ works on local queries, as opposed to remote interpreted queries.

Not all queries will benefit from PLINQ, and indeed, depending on the number of input elements and the amount of processing required, PLINQ queries may actually take longer to run due to the overhead of splitting/threading/reassembly. As with all performance-related tasks, measurements/profiling and methodical performance-tuning adjustments should be made rather than randomly turning all LINQ queries in the code base into PLINQ queries.

It should also be noted that simply using PLINQ is no guarantee of the query actually **executing** in parallel. This is because not all query operators are able to be parallelized. Even with those query operators that are parallelizable, during execution, PLINQ may decide to still run them sequentially if it determines that it may be perform better.

Applying PLINQ

To turn a normal LINQ query to a PLINQ query, the **`AsParallel`** method is added to the query. This extension method is defined in the **`System.Linq.ParallelEnumerable`** class. When the **`AsParallel`** method is added to a query, it essentially “switches on” PLINQ for that query.

The following code shows the method signature for **AsParallel** method.

```
public static ParallelQuery<TSource> AsParallel<TSource>(
                                              this IEnumerable<TSource> source)
```

Notice that the **AsParallel** extension method extends **IEnumerable<TSource>**. Because local LINQ queries operate on **IEnumberable<T>** sequence, the method is available to be added to regular local LINQ queries. The other important thing to note in the preceding code is that the return type is **ParallelQuery<TSource>**. It is this **different return type** that "switches on" PLINQ. Because the query "stream" has now been converted from an **IEnumerable<T>** to a **ParallelQuery<TSource>** the other PLINQ versions of the query operators can now be applied to the query.

To illustrate this, the following code shows the differences in the signature of the **Where** operator between regular local LINQ and PLINQ

```
// Where query operator from System.Linq.Enumerable
public static IEnumerable<TSource> Where<TSource>(
             this IEnumerable<TSource> source, Func<TSource, bool> predicate)

// Where query operator from System.Linq.ParallelEnumerable
public static ParallelQuery<TSource> Where<TSource>(
           this ParallelQuery<TSource> source, Func<TSource, bool> predicate)
```

Notice in the preceding code the second PLINQ version of the **Where** query operator is extending **ParallelQuery<T>** and not **IEnumerable<T>**.

So once we have switched our regular LINQ query to PLINQ using **AsParallel**, when we use subsequent query operators, they will use the PLINQ versions and be executed in parallel where possible, and when PLINQ does not decide sequential (non-parallel) execution is better.

The following code shows two queries that both filter the input sequence of numbers where the string version of that number contains the string "3". This example code is to create enough of a CPU load to see some meaningful output timings. The second version of the query uses PLINQ to parallelize the query. In both queries the output sequence result is being ignored and **ToArray** is being used to force the query to run. The execution time for both queries is being captured using a **StopWatch**.

```
var someNumbers = Enumerable.Range(1, 10000000).ToArray();


var sw = new Stopwatch();

sw.Start();
someNumbers.Where(x => x.ToString().Contains("3")).ToArray();
sw.Stop();

Console.WriteLine("Non PLINQ query took {0} ms", sw.ElapsedMilliseconds);
```

```
sw.Restart();
someNumbers.AsParallel().Where(x => x.ToString().Contains("3")).ToArray();
sw.Stop();

Console.WriteLine("PLINQ query took {0} ms", sw.ElapsedMilliseconds);
```

This produces the following output:

```
Non PLINQ query took 3218 ms
PLINQ query took 1494 ms
```

We can see from these results that the PLINQ version of the query executed about twice as fast as the non-PLINQ version.

Output element ordering

When a PLINQ query executes in parallel, the input sequence is broken up into sub-segments, so once each sub-segment has been processed, it needs to be added into the overall resulting output sequence.

While regular LINQ queries respect the ordering of input elements when they appear in the output sequence, PLINQ queries may not return elements in the same order as they were put in.

The following code demonstrates this by performing a simple **`Select`** query that just returns the input number unchanged on ten input numbers. The query uses PLINQ. The numbers are in the input and output sequences and output for comparison.

```
var inputNumbers = Enumerable.Range(1, 10).ToArray();

Console.WriteLine("Input numbers");
foreach (var num in inputNumbers)
{
    Console.Write(num + " ");
}


var outputNumbers = inputNumbers.AsParallel().Select(x => x);

Console.WriteLine();
Console.WriteLine("Output numbers");
foreach (var num in outputNumbers)
{
    Console.Write(num + " ");
}
```

This produces the following output:

```
Input numbers
1 2 3 4 5 6 7 8 9 10

Output numbers
1 4 7 9 2 5 8 10 3 6
```

Notice the output sequence element ordering does not match the input ordering.

To force the output sequence to be in the same order as the input sequence, the **AsOrdered** PLINQ extension method can be used, as the following code demonstrates.

```
var inputNumbers = Enumerable.Range(1, 10).ToArray();

var outputNumbers = inputNumbers.AsParallel().AsOrdered().Select(x => x);

Console.WriteLine("Output numbers");
foreach (var num in outputNumbers)
{
    Console.Write(num + " ");
}
```

This produces the following output:

```
Output numbers
1 2 3 4 5 6 7 8 9 10
```

Notice now the output elements are in the same order as the input elements. It should be noted that PLINQ has to do extra work to track the position of input elements when **AsOrdered** is used, so there may be some negative performance implications, depending on the size of the input sequence. If only part of the query requires ordering preservation, then **AsOrdered** can be used for those parts. If other parts of the query do not require order preservation, then the default unordered behavior can be restored by adding a call to the **AsUnordered** extension method. Subsequent query operators after the **AsUnordered** will not track the input ordering of elements.

Potential PLINQ Problems

There are a number of things to be aware of when using PLINQ:

- PLINQ may not always be faster than LINQ
- Avoid writing to shared memory (such as static variables)
- Do not call non-thread safe methods from PLINQ
- Calling thread-safe methods may incur locking/synchronization overheads

For further information and a more comprehensive list, see the MSDN documentation at https://msdn.microsoft.com/en-us/library/dd997403%28v=vs.110%29.aspx.

Mixing LINQ and PLINQ

A single query can execute partly in standard sequential mode and have some parts of the query execute in parallel. This provides flexibility to the query author and enables non-thread-safe parts of the query (or parts where parallelization may run more slowly) to run sequentially, while other parts run in parallel.

While the **`AsParallel`** method switches the query into PLINQ mode, the companion **`AsSequential`** extension method switches back to regular LINQ.

The following code shows the method signature of the **`AsSequential`** method. Notice that the extension method works on a **`ParallelQuery<TSource>`** input sequence and returns a standard **`IEnumerable<T>`**. Because the return type is a normal **`IEnumerable<T>`**, subsequent query operators will bind to the standard local LINQ operators, not the PLINQ ones.

```
public static IEnumerable<TSource> AsSequential<TSource>(
                                        this ParallelQuery<TSource> source)
```

The following code demonstrates how to switch between LINQ and PLINQ in a single query.

```
IEnumerable<int> inputNumbers = Enumerable.Range(1, 10).ToArray();

IEnumerable<int> outputNumbers = inputNumbers
                                    .AsParallel()
                                    .Select(x => x) // PLINQ version of Select
                                    .AsSequential()
                                    .Select(x => x); // LINQ version of Select
```

Chapter 7 LINQ Tools and Resources

MSDN LINQ Home

The starting page for the MSDN coverage of LINQ can be found at https://msdn.microsoft.com/en-AU/library/bb397926.aspx.

101 LINQ Samples

This MSDN resource provides a multitude of sample code demonstrating the various LINQ query operators. You can download all the samples or browse the code at https://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b.

C# 5.0 In a Nutshell

This book (authored by Joseph Albahari & Ben Albahari, published by O'Reilly) covers just about everything related to C#, including comprehensive coverage of LINQ.

LINQPad

LINQPad is the ".NET Programmer's Playground," and is a desktop tool that helps you quickly prototype and execute LINQ queries, in addition to many other features. LINQPad can be downloaded (including free and paid-for versions) from http://www.linqpad.net/.