

Querying and Manipulating Data Using LINQ

This section is about **Language Integrated Query (LINQ)**, a set of language extensions that add the ability to work with sequences of items and then filter, sort, and project them into different outputs.

This section will cover the following topics:

- Writing LINQ queries
- Working with sets using LINQ
- Using LINQ with EF Core
- Sweetening LINQ syntax with syntactic sugar
- Creating your own LINQ extension methods

Writing LINQ queries

Although we wrote a few LINQ queries in *Section 4, Working with Databases Using Entity Framework Core*, they weren't the focus, and so I didn't properly explain how LINQ works, but let's now take time to properly understand them.

LINQ has several parts; some are required, and some are optional:

- **Extension methods (required):** These include examples such as `Where`, `OrderBy`, and `Select`. These are what provide the functionality of LINQ.
- **LINQ providers (required):** These include LINQ to Objects, LINQ to Entities, LINQ to XML, LINQ to OData, and LINQ to Amazon. These are what convert standard LINQ operations into specific commands for different types of data.
- **Lambda expressions (optional):** These can be used instead of named methods to simplify LINQ extension method calls.
- **LINQ query comprehension syntax (optional):** These include `from`, `in`, `where`, `orderby`, `descending`, and `select`. These are C# keywords that are aliases for some of the LINQ extension methods, and their use can simplify the queries you write, especially if you already have experience with other query languages, such as **Structured Query Language (SQL)**.

When programmers are first introduced to LINQ, they often believe that LINQ query comprehension syntax is LINQ, but ironically, that is one of the parts of LINQ that is optional!

Extending sequences with the Enumerable class

The LINQ extension methods, such as `Where` and `Select`, are appended by the `Enumerable` static class to any type, known as a **sequence**, that implements `IEnumerable<T>`.

For example, an array of any type implements the `IEnumerable<T>` class, where `T` is the type of item in the array, so all arrays support LINQ to query and manipulate them.

All generic collections, such as `List<T>`, `Dictionary<TKey, TValue>`, `Stack<T>`, and `Queue<T>`, implement `IEnumerable<T>`, so they can be queried and manipulated with LINQ.

`Enumerable` defines more than 45 extension methods, as summarized in the following table:

Method(s)	Description
<code>First</code> , <code>FirstOrDefault</code> , <code>Last</code> , <code>LastOrDefault</code>	Gets the first or last item in the sequence or throws an exception, or returns the default value for the type, for example, 0 for an <code>int</code> and <code>null</code> for a reference type, if there is no first or last item.
<code>Where</code>	Returns a sequence of items that match a specified filter.
<code>Single</code> , <code>SingleOrDefault</code>	Returns an item that matches a specific filter or throws an exception, or returns the default value for the type, if there is not exactly one match.
<code>ElementAt</code> , <code>ElementAtOrDefault</code>	Returns an item at a specified index position or throws an exception, or returns the default value for the type, if there is not an item at that position.
<code>Select</code> , <code>SelectMany</code>	Projects items into a different shape, that is, type, and flattens a nested hierarchy of items.
<code>OrderBy</code> , <code>OrderByDescending</code> , <code>ThenBy</code> , <code>ThenByDescending</code>	Sorts items by a specified property.
<code>Reverse</code>	Reverses the order of items.
<code>GroupBy</code> , <code>GroupJoin</code> , <code>Join</code>	Group and join sequences.
<code>Skip</code> , <code>SkipWhile</code>	Skips a number of items or skips while an expression is true.
<code>Take</code> , <code>TakeWhile</code>	Take a number of items or take while an expression is true.
<code>Aggregate</code> , <code>Average</code> , <code>Count</code> , <code>LongCount</code> , <code>Max</code> , <code>Min</code> , <code>Sum</code>	Calculates aggregate values.
<code>All</code> , <code>Any</code> , <code>Contains</code>	Returns true if all or any of the items match the filter, or if the sequence contains a specified item.
<code>Cast</code>	Converts items into a specified type.
<code>OfType</code>	Removes items that do not match a specified type.
<code>Except</code> , <code>Intersect</code> , <code>Union</code>	Performs operations that return sets. Sets cannot have duplicate items. Although the inputs of these methods can be any sequence and so can have duplicates, the result is always a set.
<code>Append</code> , <code>Concat</code> , <code>Prepend</code>	Performs sequence-combining operations.
<code>Zip</code>	Performs a match operation on two sequences based on the position of items, for example, the item at position 1 in the first sequence matches the item at position 1 in the second sequence.
<code>Distinct</code>	Removes duplicate items from the sequence.
<code>ToArray</code> , <code>ToList</code> , <code>ToDictionary</code> , <code>ToLookup</code>	Converts the sequence into an array or collection.

Filtering entities with Where

The most common reason for using LINQ is to filter items in a sequence using the `Where` extension method. Let's explore filtering by defining a sequence of names and then applying LINQ operations to it:

1. In the Code folder, create a folder named `Section05`, with a subfolder named `LinqWithObjects`.
2. In Visual Studio Code, save a workspace as `Section05.code-workspace` in the `Section05` folder.
3. Add the folder named `LinqWithObjects` to the workspace.
4. Navigate to **Terminal | New Terminal**.
5. In **TERMINAL**, enter the following command:

```
dotnet new console
```

6. In `Program.cs`, add a `LinqWithArrayOfStrings` method, which defines an array of `string` values and then attempts to call the `Where` extension method on it, as shown in the following code:

```
static void LinqWithArrayOfStrings()
{
    var names = new string[] { "Michael", "Pam", "Jim", "Dwight",
        "Angela", "Kevin", "Toby", "Creed" };
    var query = names.
}
```

7. As you type the `Where` method, note that it is missing from the IntelliSense list of members of a `string` array, as shown in the following screenshot:

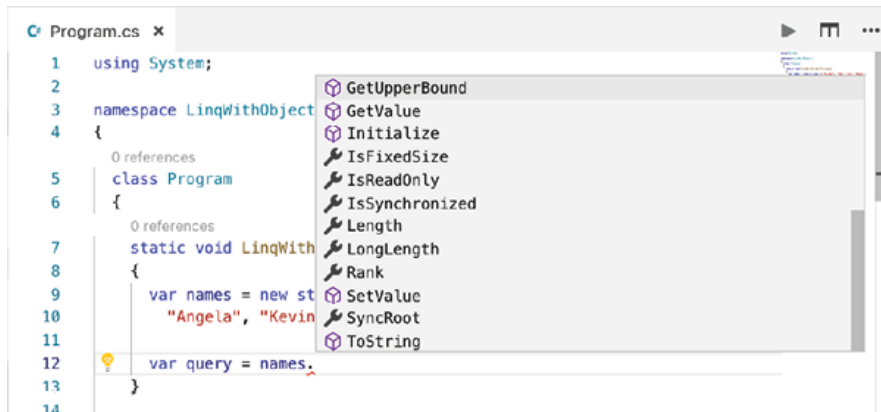


Figure 5.1: IntelliSense not showing the `Where` method

This is because `Where` is an **extension method**. It does not exist on the array type. To make the `Where` extension method available, we must import the `System.Linq` namespace.

8. Add the following statement to the top of the `Program.cs` file:

```
using System.Linq;
```

9. Retype the `Where` method and note that the IntelliSense list shows many more methods, including the extension methods added by the `Enumerable` class, as shown in the following screenshot:

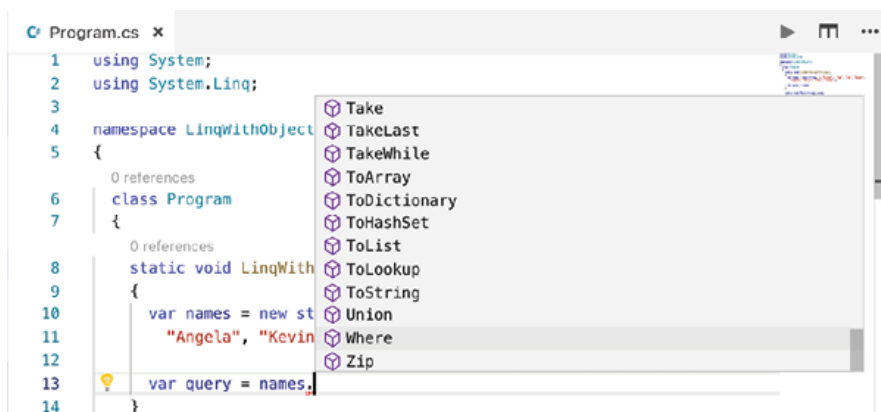


Figure 5.2: IntelliSense showing many more methods now

10. As you type the parentheses for the `Where` method, IntelliSense tells us that to call `Where`, we must pass in an instance of a `Func<string, bool>` delegate, as shown in the following screenshot:

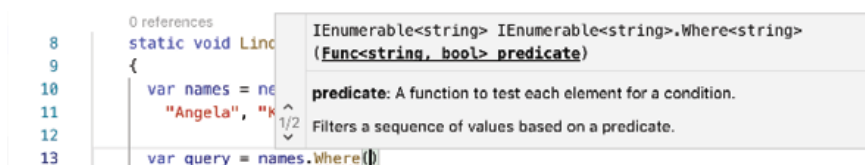


Figure 5.3: Method signature showing you must pass in a `Func<string, bool>` delegate

11. Enter an expression to create a new instance of a `Func<string, bool>` delegate, and for now note that we have not yet supplied a method name because we will define it in the next step, as shown in the following code:

```
var query = names.Where(new Func<string, bool>())
```

The `Func<string, bool>` delegate tells us that for each `string` variable passed to the method, the method must return a `bool` value. If the method returns `true`, it indicates that we should include the `string` in the results, and if the method returns `false`, it indicates that we should exclude it.

Targeting a named method

Let's define a method that only includes names that are longer than four characters:

1. Add a method to `Program`, as shown in the following code:

```
static bool NameLongerThanFour(string name)
{
    return name.Length > 4;
}
```

2. Back in the `LinqWithArrayOfStrings` method, pass the method's name into the `Func<string, bool>` delegate, and then loop through the query items, as shown in the following code:

```
var query = names.Where(
    new Func<string, bool>(NameLongerThanFour));
foreach (string item in query)
{
    WriteLine(item);
}
```

3. In `Main`, call the `LinqWithArrayOfStrings` method, run the console application, and view the results, noting that only names longer than four letters are listed, as shown in the following output:

```
Michael
Dwight
Angela
Kevin
Creed
```

Simplifying the code by removing the explicit delegate instantiation

We can simplify the code by deleting the explicit instantiation of the `Func<string, bool>` delegate because the C# compiler can instantiate the delegate for us.

1. To help you learn by seeing progressively improved code, copy and paste the query.
2. Comment out the first example, as shown in the following code:

```
// var query = names.Where(
//     new Func<string, bool>(NameLongerThanFour));
```

3. Modify the copy to remove the explicit instantiation of the delegate, as shown in the following code:

```
var query = names.Where(NameLongerThanFour);
```

4. Rerun the application and note that it has the same behavior.

Targeting a lambda expression

We can simplify our code even further using a **lambda expression** in place of a named method.

Although it can look complicated at first, a lambda expression is simply a **nameless function**. It uses the `=>` (read as "goes to") symbol to indicate the return value.

1. Copy and paste the query, comment the second example, and modify the query, as shown in the following code:

```
var query = names.Where(name => name.Length > 4);
```

Note that the syntax for a lambda expression includes all the important parts of the `NameLongerThanFour` method, but nothing more. A lambda expression only needs to define the following:

- The names of input parameters
- A return value expression

The type of the `name` input parameter is inferred from the fact that the sequence contains `string` values, and the return type must be a `bool` value for `Where` to work, so the expression after the `=>` symbol must return a `bool` value.

The compiler does most of the work for us, so our code can be as concise as possible.

2. Rerun the application and note that it has the same behavior.

Sorting entities

Other commonly used extension methods are `OrderBy` and `ThenBy`, used for sorting a sequence.

Extension methods can be chained if the previous method returns another sequence, that is, a type that implements the `IEnumerable<T>` interface.

Sorting by a single property using `OrderBy`

Let's continue working with the current project to explore sorting.

1. Append a call to `OrderBy` to the end of the existing query, as shown in the following code:

```
var query = names
    .Where(name => name.Length > 4)
    .OrderBy(name => name.Length);
```

Good Practice: Format the LINQ statement so that each extension method call happens on its own line to make them easier to read.

2. Rerun the application and note that the names are now sorted by shortest first, as shown in the following output:

```
Kevin
Creed
Dwight
Angela
Michael
```

To put the longest name first, you would use `OrderByDescending`.

Sorting by a subsequent property using `ThenBy`

We might want to sort by more than one property, for example, to sort names of the same length in alphabetical order.

1. Add a call to the `ThenBy` method at the end of the existing query, as shown highlighted in the following code:

```
var query = names
    .Where(name => name.Length > 4)
    .OrderBy(name => name.Length)
    .ThenBy(name => name);
```

2. Rerun the application and note the slight difference in the following sort order. Within a group of names of the same length, the names are sorted alphabetically by the full value of the `string`, so `Creed` comes before `Kevin`, and `Angela` comes before `Dwight`, as shown in the following output:

```
Creed
Kevin
Angela
Dwight
Michael
```

Filtering by type

`Where` is great for filtering by values, such as text and numbers. But what if the sequence contains multiple types, and you want to filter by a specific type and respect any inheritance hierarchy?

Imagine that you have a sequence of exceptions. Exceptions have a complex hierarchy, as shown in the following diagram:

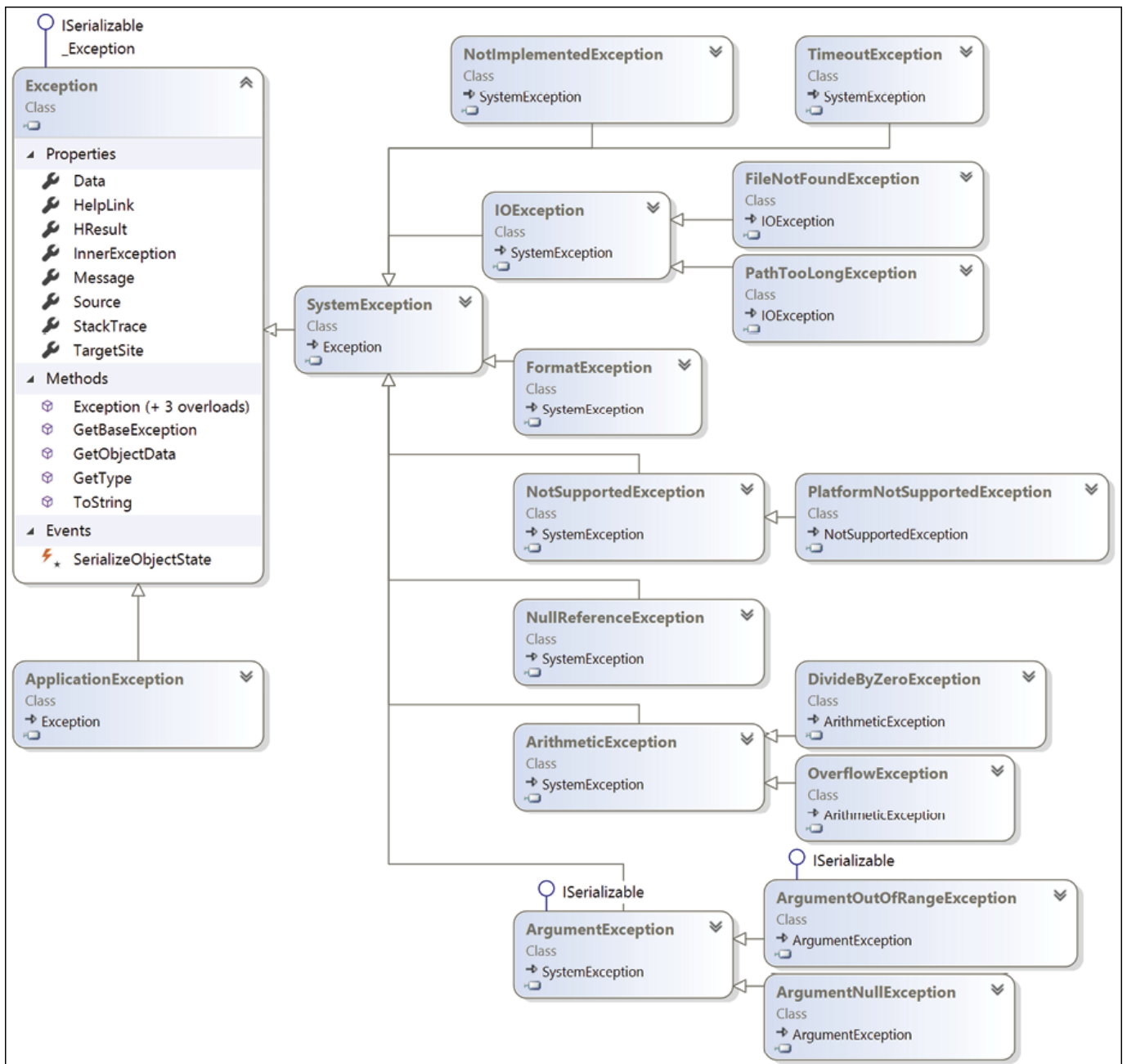


Figure 5.4: An exception's inheritance hierarchy

Let's explore filtering by type:

1. In Program, add a `LinqWithArrayOfExceptions` method, which defines an array of `Exception`-derived objects, as shown in the following code:

```
static void LinqWithArrayOfExceptions()
{
    var errors = new Exception[]
    {
        new ArgumentException(),
        new SystemException(),
        new IndexOutOfRangeException(),
        new InvalidOperationException(),
        new NullReferenceException(),
        new InvalidCastException(),
        new OverflowException(),
        new DivideByZeroException(),
        new ApplicationException()
    };
}
```

2. Write statements using the `OfType<T>` extension method to filter exceptions that are not arithmetic exceptions and write them to the console, as shown in the following code:

```
var numberErrors = errors.OfType<ArithmeticException>();
foreach (var error in numberErrors)
{
    WriteLine(error);
}
```

3. In the `Main` method, comment out the call to the `LinqWithArrayOfStrings` method and add a call to the `LinqWithArrayOfExceptions` method.
4. Run the console application and note that the results only include exceptions of the `ArithmeticException` type, or the `ArithmeticException`-derived types, as shown in the following output:

```
System.OverflowException: Arithmetic operation resulted in an overflow.  
System.DivideByZeroException: Attempted to divide by zero.
```

Working with sets and bags using LINQ

Sets are one of the most fundamental concepts in mathematics. A **set** is a collection of one or more unique objects. A **multiset** or **bag** is a collection of one or more objects that can have duplicates. You might remember being taught about Venn diagrams in school. Common set operations include the **intersect** or **union** between sets.

Let's create a console application that will define three arrays of `string` values for cohorts of apprentices and then perform some common set and multiset operations on them.

1. Create a new console application project named `LinqWithSets`, add it to the workspace for this section, and select the project as active for `OmniSharp`.
2. Import the following additional namespaces:

```
using System.Collections.Generic; // for IEnumerable<T>
using System.Linq; // for LINQ extension methods
```

3. In `Program`, before the `Main` method, add the following method that outputs any sequence of `string` variables as a comma-separated single `string` to the console output, along with an optional description:

```
static void Output(IEnumerable<string> cohort,
    string description = "")
{
    if (!string.IsNullOrEmpty(description))
    {
        WriteLine(description);
    }
    Write(" ");
    WriteLine(string.Join(", ", cohort.ToArray()));
}
```

4. In `Main`, add statements to define three arrays of names, output them, and then perform various set operations on them, as shown in the following code:

```
var cohort1 = new string[]
{ "Rachel", "Gareth", "Jonathan", "George" };
var cohort2 = new string[]
{ "Jack", "Stephen", "Daniel", "Jack", "Jared" };
var cohort3 = new string[]
{ "Declan", "Jack", "Jack", "Jasmine", "Conor" };
Output(cohort1, "Cohort 1");
Output(cohort2, "Cohort 2");
Output(cohort3, "Cohort 3");
WriteLine();
Output(cohort2.Distinct(), "cohort2.Distinct():");
WriteLine();
Output(cohort2.Union(cohort3), "cohort2.Union(cohort3):");
WriteLine();
Output(cohort2.Concat(cohort3), "cohort2.Concat(cohort3):");
WriteLine();
Output(cohort2.Intersect(cohort3), "cohort2.Intersect(cohort3):");
WriteLine();
Output(cohort2.Except(cohort3), "cohort2.Except(cohort3):");
WriteLine();
Output(cohort1.Zip(cohort2, (c1, c2) => $"{c1} matched with {c2}"),
    "cohort1.Zip(cohort2):");
```

5. Run the console application and view the results, as shown in the following output:

```
Cohort 1
Rachel, Gareth, Jonathan, George
Cohort 2
Jack, Stephen, Daniel, Jack, Jared
Cohort 3
Declan, Jack, Jack, Jasmine, Conor
cohort2.Distinct():
Jack, Stephen, Daniel, Jared
cohort2.Union(cohort3):
Jack, Stephen, Daniel, Jared, Declan, Jasmine, Conor
cohort2.Concat(cohort3):
Jack, Stephen, Daniel, Jack, Jared, Declan, Jack, Jack, Jasmine, Conor
cohort2.Intersect(cohort3):
Jack
```



```
cohort2.Except(cohort3):  
    Stephen, Daniel, Jared  
cohort1.Zip(cohort2):  
    Rachel matched with Jack, Gareth matched with Stephen, Jonathan matched with Daniel, George matched with Jack
```

With `zip`, if there are unequal numbers of items in the two sequences, then some items will not have a matching partner. Those without a partner will not be included in the result.

Using LINQ with EF Core

To learn about **projection**, it is best to have some more complex sequences to work with, so in the next project, we will use the `Northwind` sample database.

Building an EF Core model

Let's define an **Entity Framework (EF) Core** model to represent the database and tables that we will work with. We will define the entity classes manually to take complete control and to prevent a relationship from being automatically defined. Later, you will use LINQ to join the two entity sets.

1. Create a new console application project named `LinqWithEFCore`, add it to the workspace for this section, and select the project as active for OmniSharp.
2. Modify the `LinqWithEFCore.csproj` file, as shown highlighted in the following markup:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference
      Include="Microsoft.EntityFrameworkCore.Sqlite"
      Version="5.0.0" />
    </ItemGroup>
</Project>
```

3. In **TERMINAL**, download the referenced package and compile the current project, as shown in the following command:

```
dotnet build
```

4. Copy the `Northwind.sql` file into the `LinqWithEFCore` folder, and then use **TERMINAL** to create the `Northwind` database by executing the following command:

```
sqlite3 Northwind.db -init Northwind.sql
```

5. Be patient because this command might take a while to create the database structure, as shown in the following output:

```
-- Loading resources from Northwind.sql
SQLite version 3.28.0 2019-04-15 14:49:49
Enter ".help" for usage hints.
sqlite>
```

6. Press `Ctrl + D` on macOS or `Ctrl + C` on Windows to exit SQLite command mode.
7. Add three class files to the project, named `Northwind.cs`, `Category.cs`, and `Product.cs`.
8. Modify the class file named `Northwind.cs`, as shown in the following code:

```
using Microsoft.EntityFrameworkCore;
namespace Packt.Shared
{
    // this manages the connection to the database
    public class Northwind : DbContext
    {
        // these properties map to tables in the database
        public DbSet<Category> Categories { get; set; }
        public DbSet<Product> Products { get; set; }
        protected override void OnConfiguring(
            DbContextOptionsBuilder optionsBuilder)
        {
            string path = System.IO.Path.Combine(
                System.Environment.CurrentDirectory, "Northwind.db");
            optionsBuilder.UseSqlite($"Filename={path}");
        }
        protected override void OnModelCreating(
            ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Product>()
                .Property(product => product.UnitPrice)
                .HasConversion<double>();
        }
    }
}
```

9. Modify the class file named `Category.cs`, as shown in the following code:

```
using System.ComponentModel.DataAnnotations;
namespace Packt.Shared
{
    public class Category
    {
        public int CategoryID { get; set; }
        [Required]
        [StringLength(15)]
        public string CategoryName { get; set; }
        public string Description { get; set; }
    }
}
```

10. Modify the class file named `Product.cs`, as shown in the following code:

```
using System.ComponentModel.DataAnnotations;
namespace Packt.Shared
{
    public class Product
    {
        public int ProductID { get; set; }
        [Required]
        [StringLength(40)]
        public string ProductName { get; set; }
        public int? SupplierID { get; set; }
        public int? CategoryID { get; set; }
        [StringLength(20)]
        public string QuantityPerUnit { get; set; }
        public decimal? UnitPrice { get; set; }
        public short? UnitsInStock { get; set; }
        public short? UnitsOnOrder { get; set; }
        public short? ReorderLevel { get; set; }
        public bool Discontinued { get; set; }
    }
}
```

Filtering and sorting sequences

Now let's write statements to filter and sort sequences of rows from the tables.

1. Open the `Program.cs` file and import the following type and namespaces:

```
using static System.Console;
using Packt.Shared;
using Microsoft.EntityFrameworkCore;
using System.Linq;
```

2. Create a method to filter and sort products, as shown in the following code:

```
static void FilterAndSort()
{
    using (var db = new Northwind())
    {
        var query = db.Products
            // query is a DbSet<Product>
            .Where(product => product.UnitPrice < 10M)
            // query is now an IQueryable<Product>
            .OrderByDescending(product => product.UnitPrice);
        WriteLine("Products that cost less than $10:");
        foreach (var item in query)
        {
            WriteLine("{0}: {1} costs {2:$#,###0.00}",
                item.ProductID, item.ProductName, item.UnitPrice);
        }
        WriteLine();
    }
}
```

`DbSet<T>` implements `IQueryable<T>`, which implements `IEnumerable<T>`, so LINQ can be used to query and manipulate collections of entities in models built for EF Core.

You might have also noticed that the sequences implement `IQueryable<T>` (or `IOrderedQueryable<T>` after a call to an ordering LINQ method) instead of `IEnumerable<T>` or `IOrderedEnumerable<T>`.

This is an indication that we are using a LINQ provider that builds the query in memory using expression trees. They represent code in a tree-like data structure and enable the creation of dynamic queries, which is useful for building LINQ queries for external data providers like SQLite.

More Information: You can read more about expression trees at the following link:

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/expression-trees/>

The LINQ query will be converted into another query language, such as SQL. Enumerating the query with `foreach` or calling a method such as `ToArray` will force the execution of the query.

3. In `Main`, call the `FilterAndSort` method.
4. Run the console application and view the result, as shown in the following output:

```
Products that cost less than $10:
41: Jack's New England Clam Chowder costs $9.65
45: Rogede sild costs $9.50
47: Zaanse koeken costs $9.50
19: Teatime Chocolate Biscuits costs $9.20
23: Tunnbröd costs $9.00
75: Rhönbräu Klosterbier costs $7.75
54: Tourtière costs $7.45
52: Filo Mix costs $7.00
13: Konbu costs $6.00
24: Guaraná Fantástica costs $4.50
33: Geitost costs $2.50
```

Although this query outputs the information we want, it does so inefficiently because it gets all columns from the `Products` table instead of just the three columns we need, which is the equivalent of the following SQL statement:

```
SELECT * FROM Products;
```

In *Section 4, Working with Databases Using Entity Framework Core*, you learned how to log the SQL commands executed against SQLite to see this for yourself.

Projecting sequences into new types

Before we look at projection, we need to review object initialization syntax. If you have a class defined, then you can instantiate an object using `new`, the class name, and curly braces to set initial values for fields and properties, as shown in the following code:

```
var alice = new Person
{
    Name = "Alice Jones",
    DateOfBirth = new DateTime(1998, 3, 7)
};
```

C# 3.0 and later allow instances of **anonymous types** to be instantiated, as shown in the following code:

```
var anonymouslyTypedObject = new
{
    Name = "Alice Jones",
    DateOfBirth = new DateTime(1998, 3, 7)
};
```

Although we did not specify a type name, the compiler could infer an anonymous type from the setting of two properties named `Name` and `DateOfBirth`. This capability is especially useful when writing LINQ queries to project an existing type into a new type without having to explicitly define the new type. Since the type is anonymous, this can only work with `var`-declared local variables.

Let's add a call to the `Select` method to make the SQL command executed against the database table more efficient by projecting instances of the `Product` class into instances of a new anonymous type with only three properties.

1. In `Main`, modify the LINQ query to use the `Select` method to return only the three properties (that is, table columns) that we need, as shown highlighted in the following code:

```
var query = db.Products
    // query is a DbSet<Product>
    .Where(product => product.UnitPrice < 10M)
    // query is now an IQueryable<Product>
    .OrderByDescending(product => product.UnitPrice)
    // query is now an IOrderedQueryable<Product>
    .Select(product => new // anonymous type
    {
        product.ProductID,
        product.ProductName,
        product.UnitPrice
    });
```

2. Run the console application and confirm that the output is the same as before.

Joining and grouping sequences

There are two extension methods for joining and grouping:

- **Join**: This method has four parameters: the sequence that you want to join with, the property or properties on the *left* sequence to match on, the property or properties on the *right* sequence to match on, and a projection.
- **GroupJoin**: This method has the same parameters, but it combines the matches into a group object with a **Key** property for the matching value and an **IEnumerable<T>** type for the multiple matches.

Let's explore these methods when working with two tables: categories and products.

1. Create a method to select categories and products, join them, and output them, as shown in the following code:

```
static void JoinCategoriesAndProducts()
{
    using (var db = new Northwind())
    {
        // join every product to its category to return 77 matches
        var queryJoin = db.Categories.Join(
            inner: db.Products,
            outerKeySelector: category => category.CategoryID,
            innerKeySelector: product => product.CategoryID,
            resultSelector: (c, p) =>
                new { c.CategoryName, p.ProductName, p.ProductID });
        foreach (var item in queryJoin)
        {
            WriteLine("{0}: {1} is in {2}.",
                arg0: item.ProductID,
                arg1: item.ProductName,
                arg2: item.CategoryName);
        }
    }
}
```

In a join, there are two sequences, **outer** and **inner**. In the previous example, **categories** is the outer sequence and **products** is the inner sequence.

2. In **Main**, comment out the call to **FilterAndSort** and call **JoinCategoriesAndProducts**.
3. Run the console application and view the results. Note that there is a single line of output for each of the 77 products, as shown in the following output (edited to only include the first 10 items):

```
1: Chai is in Beverages.
2: Chang is in Beverages.
3: Aniseed Syrup is in Condiments.
4: Chef Anton's Cajun Seasoning is in Condiments.
5: Chef Anton's Gumbo Mix is in Condiments.
6: Grandma's Boysenberry Spread is in Condiments.
7: Uncle Bob's Organic Dried Pears is in Produce.
8: Northwoods Cranberry Sauce is in Condiments.
9: Mishi Kobe Niku is in Meat/Poultry.
10: Ikura is in Seafood.
```

4. At the end of the existing query, call the **OrderBy** method to sort by **CategoryName**, as shown in the following code:

```
.OrderBy(cp => cp.CategoryName);
```

5. Rerun the console application and view the results. Note that there is a single line of output for each of the 77 products, and the results show all products in the **Beverages** category first, then the **Condiments** category, and so on, as shown in the following partial output:

```
1: Chai is in Beverages.
2: Chang is in Beverages.
24: Guaraná Fantástica is in Beverages.
34: Sasquatch Ale is in Beverages.
35: Steeleye Stout is in Beverages.
38: Côte de Blaye is in Beverages.
39: Chartreuse verte is in Beverages.
43: Ipoh Coffee is in Beverages.
67: Laughing Lumberjack Lager is in Beverages.
70: Outback Lager is in Beverages.
75: Rhönbräu Klosterbier is in Beverages.
76: Lakkalikööri is in Beverages.
3: Aniseed Syrup is in Condiments.
4: Chef Anton's Cajun Seasoning is in Condiments.
```

6. Create a method to group and join, show the group name, and then show all the items within each group, as shown in the following code:

```
static void GroupJoinCategoriesAndProducts()
{
```

```

using (var db = new Northwind())
{
    // group all products by their category to return 8 matches
    var queryGroup = db.Categories.AsEnumerable().GroupJoin(
        inner: db.Products,
        outerKeySelector: category => category.CategoryID,
        innerKeySelector: product => product.CategoryID,
        resultSelector: (c, matchingProducts) => new {
            c.CategoryName,
            Products = matchingProducts.OrderBy(p => p.ProductName)
        });
    foreach (var item in queryGroup)
    {
        WriteLine("{0} has {1} products.",
            arg0: item.CategoryName,
            arg1: item.Products.Count());
        foreach (var product in item.Products)
        {
            WriteLine($" {product.ProductName}");
        }
    }
}

```

If we had not called the `AsEnumerable` method, then a runtime exception is thrown, as shown in the following output:

```

Unhandled exception. System.NotImplementedException: The method or operation is not implemented.
at Microsoft.EntityFrameworkCore.Relational.Query.Pipeline.RelationalQueryableMethodTranslatingExpressionVisitor
    .TranslateGroupJoin(ShapedQueryExpression outer, ShapedQueryExpression inner, LambdaExpression outerKeySelector,
    LambdaExpression innerKeySelector, LambdaExpression resultSelector)

```

This is because not all LINQ extension methods can be converted from expression trees into other query syntax like SQL. In these cases, we can convert from `IQueryable<T>` to `IEnumerable<T>` by calling the `AsEnumerable` method, which forces query processing to use LINQ to EF Core only to bring the data into the application and then use LINQ to Objects to execute more complex processing in-memory. But, often, this is less efficient.

7. In `Main`, comment the previous method call and call `GroupJoinCategoriesAndProducts`.
8. Rerun the console application, view the results, and note that the products inside each category have been sorted by their name, as defined in the query and as shown in the following partial output:

```

Beverages has 12 products.
Chai
Chang
Chartreuse verte
Côte de Blaye
Guaraná Fantástica
Ipoh Coffee
Lakkalikööri
Laughing Lumberjack Lager
Outback Lager
Rhönbräu Klosterbier
Sasquatch Ale
Steeleye Stout
Condiments has 12 products.
Aniseed Syrup
Chef Anton's Cajun Seasoning
Chef Anton's Gumbo Mix
...

```

Aggregating sequences

There are LINQ extension methods to perform aggregation functions, such as `Average` and `Sum`. Let's write some code to see some of these methods in action aggregating information from the `Products` table.

1. Create a method to show the use of the aggregation extension methods, as shown in the following code:

```

static void AggregateProducts()
{
    using (var db = new Northwind())
    {
        WriteLine("{0,-25} {1,10}",
            arg0: "Product count:",
            arg1: db.Products.Count());
        WriteLine("{0,-25} {1,10:$#,##0.00}",
            arg0: "Highest product price:",
            arg1: db.Products.Max(p => p.UnitPrice));
        WriteLine("{0,-25} {1,10:N0}",
            arg0: "Sum of units in stock:",

```

```

        arg1: db.Products.Sum(p => p.UnitsInStock));
        WriteLine("{0,-25} {1,10:N0}",
            arg0: "Sum of units on order:",
            arg1: db.Products.Sum(p => p.UnitsOnOrder));
        WriteLine("{0,-25} {1,10:$#,##0.00}",
            arg0: "Average unit price:",
            arg1: db.Products.Average(p => p.UnitPrice));
        WriteLine("{0,-25} {1,10:$#,##0.00}",
            arg0: "Value of units in stock:",
            arg1: db.Products.AsEnumerable()
                .Sum(p => p.UnitPrice * p.UnitsInStock));
    }
}

```

2. In Main, comment the previous method and call `AggregateProducts`.
3. Run the console application and view the result, as shown in the following output:

```

Product count:           77
Highest product price:   $263.50
Sum of units in stock:   3,119
Sum of units on order:   780
Average unit price:      $28.87
Value of units in stock: $74,050.85

```

In EF Core 3.0 and later, LINQ operations that cannot be translated to SQL are no longer automatically evaluated on the client side, so you must explicitly call `AsEnumerable` to force further processing of the query on the client.

More Information: You can learn more about this breaking change at the following link:

<https://docs.microsoft.com/en-us/ef/core/what-is-new/ef-core-3.x/breaking-changes#linq-queries-are-no-longer-evaluated-on-the-client>

Sweetening LINQ syntax with syntactic sugar

C# 3.0 introduced some new language keywords in 2008 in order to make it easier for programmers with experience with SQL to write LINQ queries. This **syntactic sugar** is sometimes called the **LINQ query comprehension syntax**.

More Information: The LINQ query comprehension syntax is limited in functionality. It only provides C# keywords for the most commonly used LINQ features. You must use extension methods to access all the features of LINQ. You can read more about why it is called comprehension syntax at the following link:

<https://stackoverflow.com/questions/6229187/linq-why-is-it-called-comprehension-syntax>

Consider the following array of `string` values:

```
var names = new string[] { "Michael", "Pam", "Jim", "Dwight",  
    "Angela", "Kevin", "Toby", "Creed" };
```

To filter and sort the names, you could use **extension methods** and **lambda expressions**, as shown in the following code:

```
var query = names  
    .Where(name => name.Length > 4)  
    .OrderBy(name => name.Length)  
    .ThenBy(name => name);
```

Or, you could achieve the same results by using **query comprehension syntax**, as shown in the following code:

```
var query = from name in names  
    where name.Length > 4  
    orderby name.Length, name  
    select name;
```

The compiler changes the query comprehension syntax to the equivalent extension methods and lambda expressions for you.

The `select` keyword is always required for LINQ query comprehension syntax. The `Select` extension method is optional when using extension methods and lambda expressions because the whole item is implicitly selected.

Not all extension methods have a C# keyword equivalent, for example, the `Skip` and `Take` extension methods, which are commonly used to implement paging for lots of data.

A query that skips and takes cannot be written using only the query comprehension syntax, so we could write the query using all extension methods, as shown in the following code:

```
var query = names  
    .Where(name => name.Length > 4)  
    .Skip(80)  
    .Take(10);
```

Or, you can wrap query comprehension syntax in parentheses and then switch to using extension methods, as shown in the following code:


```
var query = (from name in names
             where name.Length > 4
             select name)
            .Skip(80)
            .Take(10);
```

Good Practice: Learn both extension methods with lambda expressions and the query comprehension syntax ways of writing LINQ queries, because you are likely to have to maintain code that uses both.

Creating your own LINQ extension methods

To create LINQ extension methods, all you must do is extend the `IEnumerable<T>` type.

Good Practice: Put your own extension methods in a separate class library so that they can be easily deployed as their own assembly or NuGet package.

We will look at the `Average` extension method as an example. A well-educated school child will tell you that `average` can mean one of three things:

- **Mean:** Sum the numbers and divide by the count.
- **Mode:** The most common number.
- **Median:** The number in the middle of the numbers when ordered.

Microsoft's implementation of the `Average` extension method calculates the mean. We might want to define our own extension methods for `Mode` and `Median`.

1. In the `LinqWithEFCore` project, add a new class file named `MyLinqExtensions.cs`.
2. Modify the class, as shown in the following code:

```
using System.Collections.Generic;
namespace System.Linq // extend Microsoft's namespace
{
    public static class MyLinqExtensions
    {
        // this is a chainable LINQ extension method
        public static IEnumerable<T> ProcessSequence<T>(
            this IEnumerable<T> sequence)
        {
            // you could do some processing here
            return sequence;
        }
        // these are scalar LINQ extension methods
        public static int? Median(this IEnumerable<int?> sequence)
        {
            var ordered = sequence.OrderBy(item => item);
            int middlePosition = ordered.Count() / 2;
            return ordered.ElementAt(middlePosition);
        }
        public static int? Median<T>(
            this IEnumerable<T> sequence, Func<T, int?> selector)
        {
            return sequence.Select(selector).Median();
        }
        public static decimal? Median(
            this IEnumerable<decimal?> sequence)
        {
            var ordered = sequence.OrderBy(item => item);
            int middlePosition = ordered.Count() / 2;
            return ordered.ElementAt(middlePosition);
        }
        public static decimal? Median<T>(
            this IEnumerable<T> sequence, Func<T, decimal?> selector)
        {
            return sequence.Select(selector).Median();
        }
        public static int? Mode(this IEnumerable<int?> sequence)
        {
            var grouped = sequence.GroupBy(item => item);
            var orderedGroups = grouped.OrderByDescending(
```

```

        group => group.Count());
    return orderedGroups.FirstOrDefault().Key;
}
public static int? Mode<T>(
    this IEnumerable<T> sequence, Func<T, int?> selector)
{
    return sequence.Select(selector).Mode();
}
public static decimal? Mode(
    this IEnumerable<decimal?> sequence)
{
    var grouped = sequence.GroupBy(item => item);
    var orderedGroups = grouped.OrderByDescending(
        group => group.Count());
    return orderedGroups.FirstOrDefault().Key;
}
public static decimal? Mode<T>(
    this IEnumerable<T> sequence, Func<T, decimal?> selector)
{
    return sequence.Select(selector).Mode();
}
}
}

```

If this class was in a separate class library, to use your LINQ extension methods, you simply need to reference the class library assembly because the `System.Linq` namespace is often already imported.

3. In `Program.cs`, in the `FilterAndSort` method, modify the LINQ query for `Products` to call your custom chainable extension method, as shown highlighted in the following code:

```

var query = db.Products
    // query is a DbSet<Product>
    .ProcessSequence()
    .Where(product => product.UnitPrice < 10M)
    // query is now an IQueryable<Product>
    .OrderByDescending(product => product.UnitPrice)
    // query is now an IOrderedQueryable<Product>
    .Select(product => new // anonymous type
    {
        product.ProductID,
        product.ProductName,
        product.UnitPrice
    });

```

4. In the `Main` method, uncomment the `FilterAndSort` method and comment out any calls to other methods.
5. Run the console application and note that you see the same output as before because your method doesn't modify the sequence. But you now know how to extend LINQ with your own functionality.
6. Create a method to output the mean, median, and mode, for `UnitsInStock` and `UnitPrice` for products, using your custom extension methods and the built-in `Average` extension method, as shown in the following code:

```

static void CustomExtensionMethods()
{
    using (var db = new Northwind())
    {
        WriteLine("Mean units in stock: {0:N0}",
            db.Products.Average(p => p.UnitsInStock));
        WriteLine("Mean unit price: {0:$#,##0.00}",
            db.Products.Average(p => p.UnitPrice));
        WriteLine("Median units in stock: {0:N0}",
            db.Products.Median(p => p.UnitsInStock));
        WriteLine("Median unit price: {0:$#,##0.00}",

```

```

        db.Products.Median(p => p.UnitPrice));
        WriteLine("Mode units in stock: {0:N0}",
            db.Products.Mode(p => p.UnitsInStock));
        WriteLine("Mode unit price: {0:$#,##0.00}",
            db.Products.Mode(p => p.UnitPrice));
    }
}

```

7. In `Main`, comment any previous method calls and call `CustomExtensionMethods`.

8. Run the console application and view the result, as shown in the following output:

```

Mean units in stock: 41
Mean unit price: $28.87
Median units in stock: 26
Median unit price: $19.50
Mode units in stock: 0
Mode unit price: $18.00

```

There are four products with a unit price of \$18.00. There are five products with 0 units in stock.

Explore topics

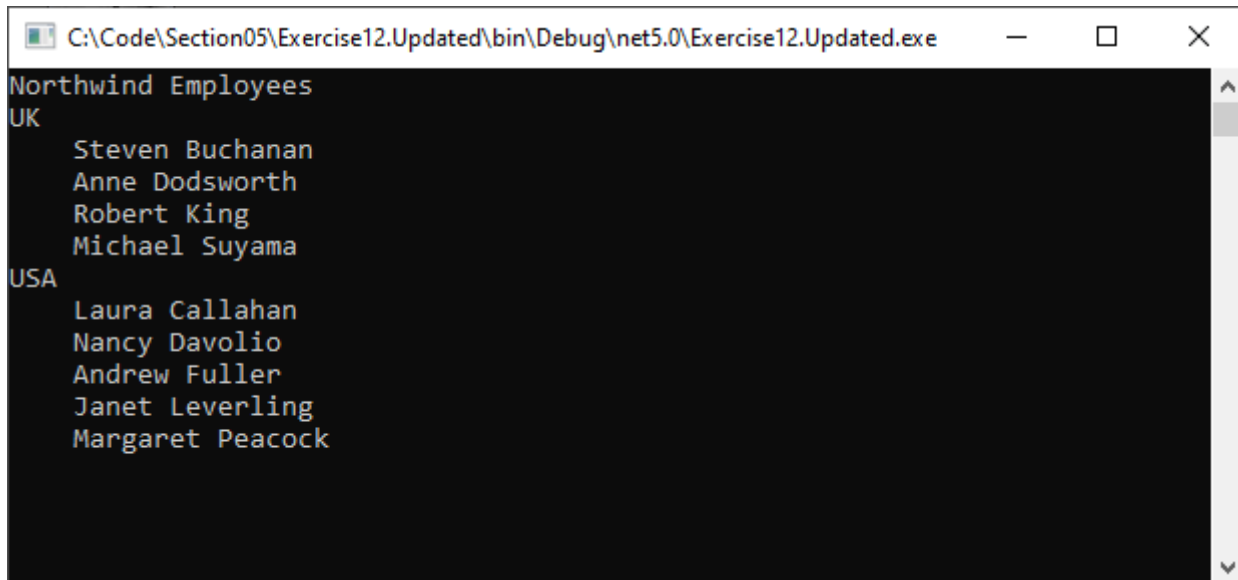
Use the following links to read more details about the topics covered in this section:

- **LINQ in C#:** <https://docs.microsoft.com/en-us/dotnet/csharp/linq/linq-in-csharp>
- **101 LINQ Samples:** <https://docs.microsoft.com/en-us/samples/dotnet/try-samples/101-linq-samples/>
- **Parallel LINQ (PLINQ):** <https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq>
- **LINQ to XML Overview (C#):** <https://docs.microsoft.com/en-gb/dotnet/csharp/programming-guide/concepts/linq/linq-to-xml-overview>
- **LINQPad:** <https://www.linqpad.net/>

Exercises

Exercise 5.1.

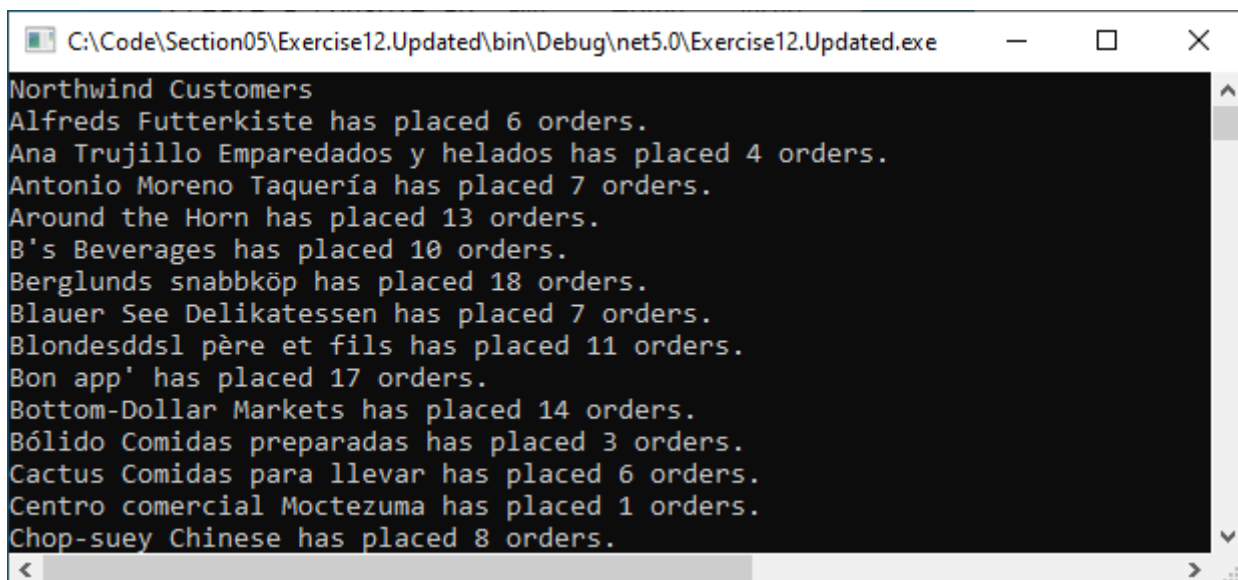
- create a console application that displays all the countries where Northwind Employees are based
- also display the full names of the employees who work in each country



```
C:\Code\Section05\Exercise12.Updated\bin\Debug\net5.0\Exercise12.Updated.exe
Northwind Employees
UK
  Steven Buchanan
  Anne Dodsworth
  Robert King
  Michael Suyama
USA
  Laura Callahan
  Nancy Davolio
  Andrew Fuller
  Janet Leverling
  Margaret Peacock
```

Exercise 5.2.

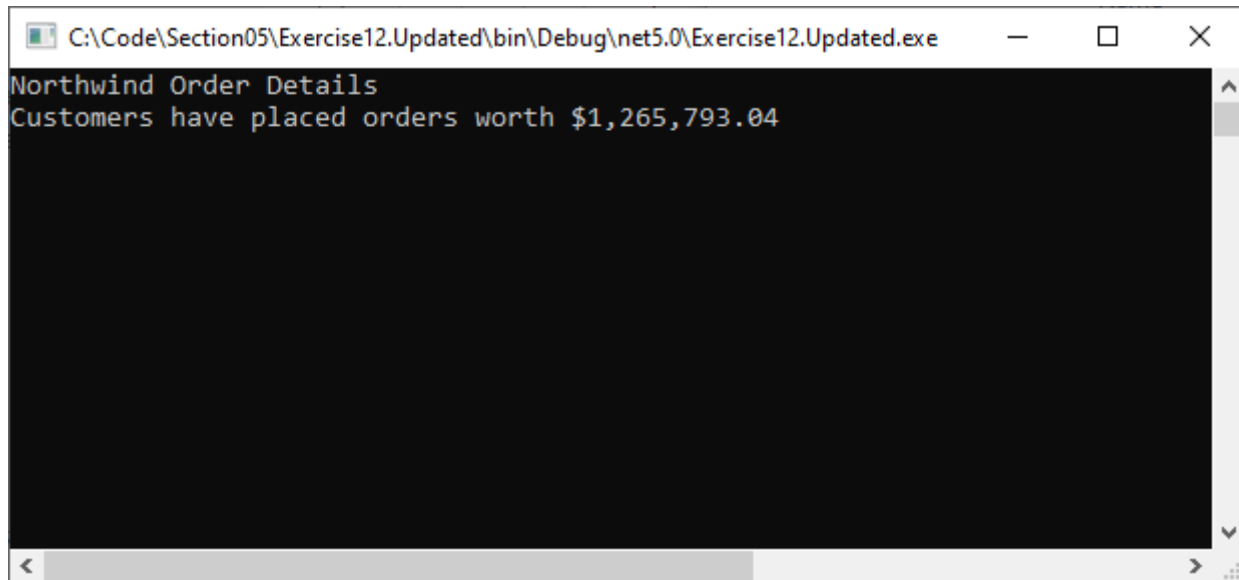
- create a console application that displays all the companies who have placed orders with Northwind Traders
- also display the number of orders each company has placed



```
C:\Code\Section05\Exercise12.Updated\bin\Debug\net5.0\Exercise12.Updated.exe
Northwind Customers
Alfreds Futterkiste has placed 6 orders.
Ana Trujillo Emparedados y helados has placed 4 orders.
Antonio Moreno Taquería has placed 7 orders.
Around the Horn has placed 13 orders.
B's Beverages has placed 10 orders.
Berglunds snabbköp has placed 18 orders.
Blauer See Delikatessen has placed 7 orders.
Blondesdds1 père et fils has placed 11 orders.
Bon app' has placed 17 orders.
Bottom-Dollar Markets has placed 14 orders.
Bólido Comidas preparadas has placed 3 orders.
Cactus Comidas para llevar has placed 6 orders.
Centro comercial Moctezuma has placed 1 orders.
Chop-suey Chinese has placed 8 orders.
```

Exercise 5.3.

- create a console application that displays the total value of orders placed with Northwind Traders



A screenshot of a Windows command prompt window. The title bar at the top shows the file path: C:\Code\Section05\Exercise12.Updated\bin\Debug\net5.0\Exercise12.Updated.exe. The window has standard minimize, maximize, and close buttons. The main area is black with white text. The text displayed is: Northwind Order Details
Customers have placed orders worth \$1,265,793.04. There is a vertical scrollbar on the right side of the text area and a horizontal scrollbar at the bottom.

```
C:\Code\Section05\Exercise12.Updated\bin\Debug\net5.0\Exercise12.Updated.exe
Northwind Order Details
Customers have placed orders worth $1,265,793.04
```