LINQ

Troelsen Kapitel 13

Grundläggande applikationsutveckling med C#

LINQ to Objects

- Data can be found in numerous locations, including XML files, relational databases, in-memory collections, and primitive arrays. Historically speaking, based on the location of said data, programmers needed to make use of different and unrelated APIs. The Language Integrated Query (LINQ) technology set, introduced initially in .NET 3.5, provides a concise, symmetrical, and strongly typed manner to access a wide variety of data stores.
- C# programming constructs that enable LINQ; implicitly typed local variables, object initialization syntax, lambda expressions, extension methods, anonymous types.
- The role of query operators and query expressions, which allow you to define statements that will interrogate a data source to yield the requested result set.

LINQ-Specific Programming Constructs

- From a high level, LINQ can be understood as a strongly typed query language, embedded directly into the grammar of C#. Using LINQ, you can build any number of expressions that have a look and feel similar to that of a database **SQL** query. However, a LINQ query can be applied to any number of data stores, including stores that have nothing to do with a literal relational database.
- Although LINQ queries look similar to SQL queries, the syntax is *not* identical. In fact, many LINQ queries seem to be the exact opposite format of a similar database query! If you attempt to map LINQ directly to SQL, you will surely become frustrated. To keep your sanity, I recommend you try your best to regard LINQ queries as unique statements, which just “happen to look” *similar* to SQL.
- When LINQ was first introduced to the .NET platform in version 3.5, the C# and VB languages were each expanded with a large number of new programming constructs used to support the LINQ technology set. Specifically, the C# language uses the following core LINQ-centric features:
  - Implicitly typed local variables
  - Object/collection initialization syntax
  - Lambda expressions
  - Extension methods
  - Anonymous types

Implicit Typing of Local Variables

- Using the **var** keyword, you can define a local variable without explicitly specifying the underlying data type. The variable, however, is strongly typed, as the compiler will determine the correct data type based on the initial assignment.

```
static void DeclareImplicitVars()
{
   // Implicitly typed local variables.
   var myInt = 0;
   var myBool = true;
   var myString = "Time, marches on...";

   // Print out the underlying type.
   Console.WriteLine("myInt is a: {0}", myInt.GetType().Name);
   Console.WriteLine("myBool is a: {0}", myBool.GetType().Name);
   Console.WriteLine("myString is a: {0}", myString.GetType().Name);
}
```

- This language feature is helpful, and often mandatory, when using LINQ. As you will see during this chapter, many LINQ queries will return a sequence of data types, which are not known until compile time. Given that the underlying data type is not known until the application is compiled, you obviously can't declare a variable *explicitly*!

Object and Collection Initialization Syntax

- Object initialization syntax allows you to create a class or structure variable and to set any number of its public properties in one fell swoop. The end result is a compact (yet still easy on the eyes) syntax that can be used to get your objects ready for use.
- Also recall from Chapter 9, the C# language allows you to use a similar syntax to initialize collections of objects.

```
List<Rectangle> myListOfRects = new List<Rectangle>
{
   new Rectangle {TopLeft = new Point { X = 10, Y = 10 },
                  BottomRight = new Point { X = 200, Y = 200}},
   new Rectangle {TopLeft = new Point { X = 2, Y = 2 },
                  BottomRight = new Point { X = 100, Y = 100}},
   new Rectangle {TopLeft = new Point { X = 5, Y = 5 },
                  BottomRight = new Point { X = 90, Y = 75}}
};
```

- While you are never required to use collection/object initialization syntax, doing so results in a more compact codebase. Furthermore, this syntax, when combined with implicit typing of local variables, allows you to declare an anonymous type, which is useful when creating a “LINQ projection”.

Lambda Expressions

- The C# lambda operator (**=>**) allows you to build a *lambda expression*, which can be used any time you invoke a method that requires a strongly typed delegate as an argument. Lambdas greatly simplify how you work with .NET delegates, in that they reduce the amount of code you have to author by hand.

```
( ArgumentsToProcess ) => { StatementsToProcessThem }
```

- Lambdas will be useful when working with the underlying object model of LINQ. As you will soon find out, the C# LINQ query operators are simply a shorthand notation for calling true-blue methods on a class named System.Linq.Enumerable. These methods typically always require delegates (the **Func<>** delegate in particular) as parameters, which are used to process your data to yield the correct result set. Using lambdas, you can streamline your code and allow the compiler to infer the underlying delegate.

Extension Methods

- C# *extension methods* allow you to tack on new functionality to existing classes without the need to subclass. As well, extension methods allow you to add new functionality to sealed classes and structures, which could never be subclassed in the first place. When you author an extension method, the first parameter is qualified with the **this** keyword and marks the type being extended. Extension methods must always be defined within a static class and must, therefore, also be declared using the static keyword.

```
namespace MyExtensions
{
   static class ObjectExtensions
   {
      // Define an extension method to System.Object.
      public static void DisplayDefiningAssembly(this object obj)
      {
         Console.WriteLine("{0} lives here:\n\t->{1}\n", obj.GetType().Name, Assembly.GetAssembly(obj.GetType()));
      }
   }
}
```

- To use this extension, an application must import the namespace defining the extension (and possibly add a reference to the external assembly). At this point, simply import the defining namespace and code away.

```
static void Main(string[] args)
{
   // Since everything extends System.Object, all classes and structures can use this extension.
   int myInt = 12345678;
   myInt.DisplayDefiningAssembly();
   System.Data.DataSet d = new System.Data.DataSet();
   d.DisplayDefiningAssembly();
}
```

Extension Methods

- When you are working with LINQ, you will seldom, if ever, be required to manually build your own extension methods. However, as you create LINQ query expressions, you will actually be making use of numerous extension methods already defined by Microsoft. In fact, each C# LINQ query operator is a shorthand notation for making a manual call on an underlying extension method, typically defined by the System.Linq.Enumerable utility class.

Anonymous Types

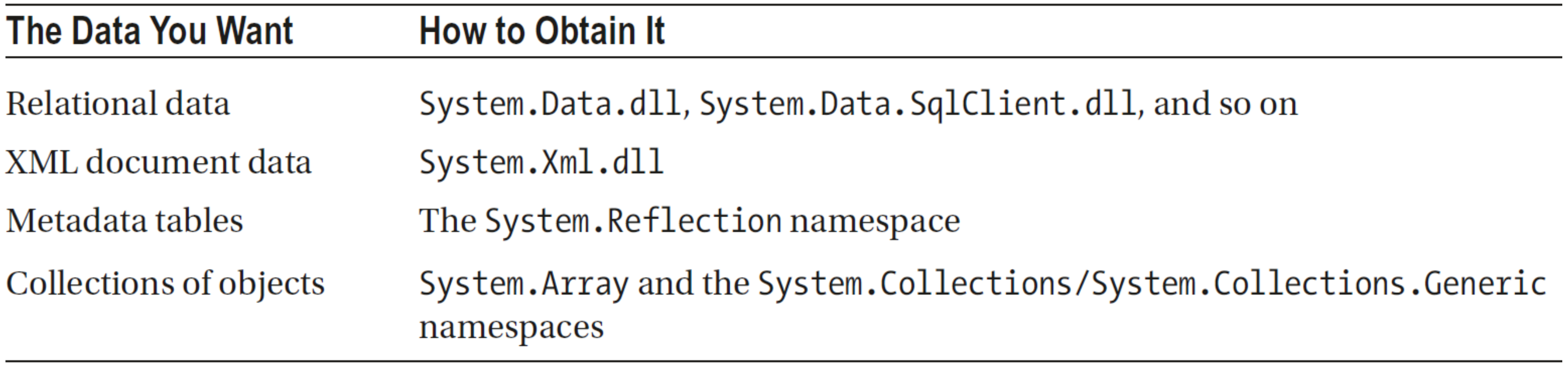
- Anonymous types can be used to quickly model the "shape" of data by allowing the compiler to generate a new class definition at compile time, based on a supplied set of name-value pairs. This type will be composed using value-based semantics, and each virtual method of **System.Object** will be overridden accordingly. To define an anonymous type, declare an implicitly typed variable and specify the data's shape using object initialization syntax.

```
// Make an anonymous type that is composed of another.
var purchaseItem = new
{
   TimeBought = DateTime.Now,
   ItemBought = new {Color = "Red", Make = "Saab", CurrentSpeed = 55},
   Price = 34.000
};
```

- LINQ makes frequent use of anonymous types when you want to project new forms of data on the fly. For example, assume you have a collection of **Person** objects and want to use LINQ to obtain information on the age and Social Security number of each. Using a LINQ projection, you can allow the compiler to generate a new *anonymous type* that contains your information.

Understanding the Role of LINQ

- Why LINQ? As software developers, the vast majority of our programming time is spent obtaining and manipulating data. When speaking of “data,” it is easy to immediately envision information contained within relational databases. However, another popular location for data is within XML documents or simple text files.
- Data can be found in numerous places beyond these two common homes for information. For instance, say you have an array or generic **List<T>** type containing 300 integers and you want to obtain a subset that meets a given criterion (e.g., only the odd or even members in the container, only prime numbers, only nonrepeating numbers greater than 50).
- Prior to .NET 3.5, interacting with a particular flavor of data required programmers to use very diverse APIs.



The Data You Want	How to Obtain It
Relational data	`System.Data.dll`, `System.Data.SqlClient.dll`, and so on
XML document data	`System.Xml.dll`
Metadata tables	The `System.Reflection` namespace
Collections of objects	`System.Array` and the `System.Collections`/`System.Collections.Generic` namespaces

Understanding the Role of LINQ

- Of course, nothing is wrong with these approaches to data manipulation. In fact, you can (and will) certainly make direct use of ADO.NET, the XML namespaces, and the various collection types. However, the basic problem is that each of these APIs is an island unto itself, which offers little in the way of integration.
- The LINQ API is an attempt to provide a consistent, symmetrical manner in which programmers can obtain and manipulate "data" in the broad sense of the term. Using LINQ, you are able to create directly within the C# programming language constructs called *query expressions*. These query expressions are based on numerous query operators that have been intentionally designed to look and feel similar (but not quite identical) to a SQL expression.
- A query expression can be used to interact with numerous types of data— even data that has nothing to do with a relational database. Strictly speaking, "LINQ" is the term used to describe this overall approach to data access. However, based on where you are applying your LINQ queries, you will encounter various terms, such as the following:
    - *LINQ to Objects*: This term refers to the act of applying LINQ queries to arrays and collections.
    - *LINQ to XML*: This term refers to the act of using LINQ to manipulate and query XML documents.
    - *LINQ to DataSet*: This term refers to the act of applying LINQ queries to ADO.NET DataSet objects.
    - *LINQ to Entities*: This aspect of LINQ allows you to make use of LINQ queries within the Entity Framework (EF) API.
    - *Parallel LINQ (aka PLINQ)*: This allows for parallel processing of data returned from a LINQ query.

LINQ Expressions are Strongly Typed

- It is also important to point out that a LINQ query expression (unlike a traditional SQL statement) is *strongly typed*. Therefore, the C# compiler will keep you honest and make sure that these expressions are syntactically well-formed.

The Core LINQ Assemblies

- The New Project dialog of Visual Studio has the option of selecting which version of the .NET platform you want to compile against. When you opt to compile against .NET 3.5 or higher, each of the project templates will automatically reference the key LINQ assemblies.

Assembly	Meaning in Life
`System.Core.dll`	Defines the types that represent the core LINQ API. This is the one assembly you must have access to if you want to use any LINQ API, including LINQ to Objects.
`System.Data.DataSetExtensions.dll`	Defines a handful of types to integrate ADO.NET types into the LINQ programming paradigm (LINQ to DataSet).
`System.Xml.Linq.dll`	Provides functionality for using LINQ with XML document data (LINQ to XML).

- To work with LINQ to Objects, you must make sure that every C# code file that contains LINQ queries imports the **System.Linq** namespace (primarily defined within **System.Core.dll**). If you see a compiler error looking similar the below, the chances are extremely good that your C# file does not have the following using directive: `using System.Linq;`

```
Error 1 Could not find an implementation of the query pattern for source type 'int[]'.
'Where' not found. Are you missing a reference to 'System.Core.dll' or a using directive
for 'System.Linq'?
```

Applying LINQ Queries to Primitive Arrays

- When you have any array of data, it is common to extract a subset of items based on a given requirement (e.g. only subitems that contain a number, have more or less than some number of characters, or don't contain embedded spaces. LINQ query expressions can greatly simplify the process.
- Assume you want to obtain, from the array, only items that contain an embedded blank space and you want these items listed in alphabetical order:

```
static void QueryOverStrings()
{
   // Assume we have an array of strings.
   string[] currentVideoGames = {"Morrowind", "Uncharted 2", "Fallout 3", "Daxter", "System Shock 2"};

   // Build a query expression to find the items in the array that have an embedded space.
   IEnumerable<string> subset = from g in currentVideoGames
                                where g.Contains(" ")
                                orderby g
                                select g;
   // Print out the results.
   foreach (string s in subset)
      Console.WriteLine("Item: {0}", s);
}
```

```
Item: Fallout 3
Item: System Shock 2
Item: Uncharted 2
```

- Each item that matches the search criteria has been given the name “g”, but any valid variable name will do.
- Notice that the returned sequence is held in a variable named subset, typed as a type that implements the generic version of **IEnumerable<T>**, where T is of type **System.String**. After you obtain the result set, you then simply print out each item using a standard **foreach** construct.

Once Again Using Extension Methods

- The LINQ syntax used in the previous example is referred to as a LINQ ***query expression***, which is a format that is similar to SQL but (somewhat annoying) different. There is another syntax that uses ***extension methods***. Most LINQ statements can be written using either format; however, some of the more complex queries will require using ***query expressions***.

```
static void QueryOverStringsWithExtensionMethods()
{
   // Assume we have an array of strings.
   string[] currentVideoGames = {"Morrowind", "Uncharted 2", "Fallout 3", "Daxter", "System Shock 2"};

   // Build a query expression to find the items in the array that have an embedded space.
   IEnumerable<string> subset = currentVideoGames.Where(g => g.Contains(" ")).OrderBy(g => g).Select(g => g);

   // Print out the results.
   foreach (string s in subset)
      Console.WriteLine("Item: {0}", s);
}
```

- Everything is the same as the previous method, except for the line in **red**. This is using the ***extension method*** syntax. This syntax uses lambda expressions within each method to define the operation. For example, the lambda in the **Where()** method defines the condition (where a value contains a space). Just as in the query expression syntax, the letter "g" used to indicate the value being evaluated in the lambda is random (*any* variable name works).
- While the results are the same, you will see soon that the *type* of the result set is slightly different. For most (if not practically all) scenarios, this difference doesn't cause any issues, and the formats can be used interchangeably.

Once Again Without LINQ

- To be sure, LINQ is *never* mandatory. If you so choose, you could have found the same result set by forgoing LINQ altogether and making use of programming primitives such as if statements and for loops. Here is a method that yields the same result as the **QueryOverStrings()** method but in a much more verbose manner.

```
static void QueryOverStringsLongHand()
{
   // Assume we have an array of strings.
   string[] currentVideoGames = {"Morrowind", "Uncharted 2", "Fallout 3", "Daxter", "System Shock 2"};
   string[] gamesWithSpaces = new string[5];
   for (int i = 0; i < currentVideoGames.Length; i++)
   {
      if (currentVideoGames[i].Contains(" "))
         gamesWithSpaces[i] = currentVideoGames[i];
   }
   // Now sort them.
   Array.Sort(gamesWithSpaces);
   // Print out the results.
   foreach (string s in gamesWithSpaces)
   {
      if( s != null)
         Console.WriteLine("Item: {0}", s);
   }
   Console.WriteLine();
}
```

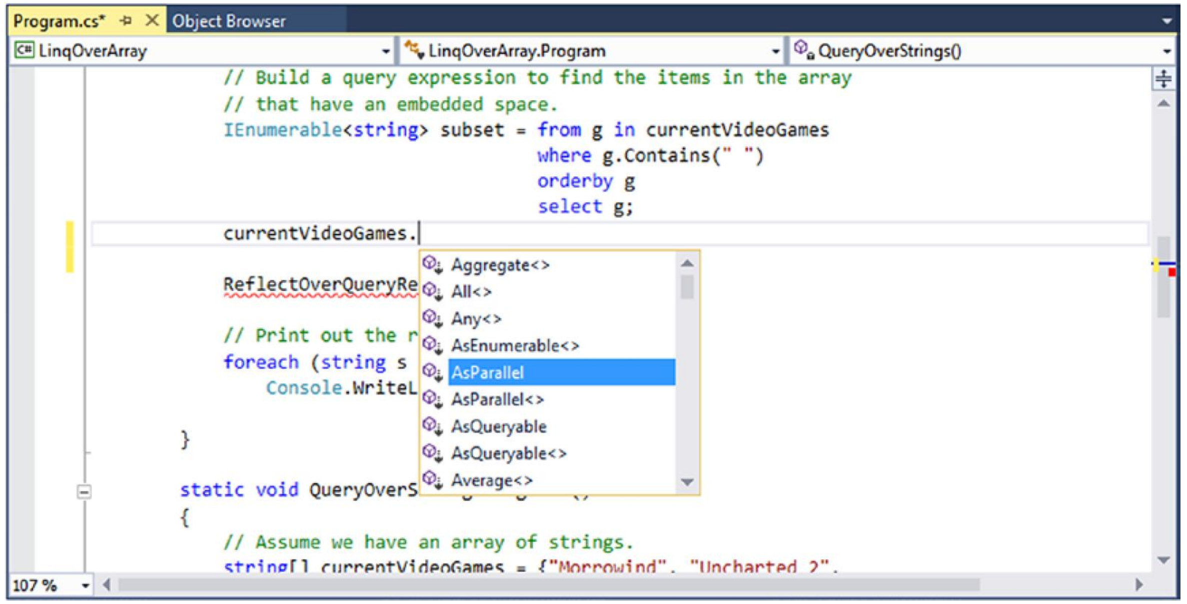
LINQ and Implicitly Typed Local Variables

- Depending on the LINQ query, the returned result set can be of different types (e.g. **OrderedEnumerable<TElement, TKey>**, **WhereArrayIterator<T>**). Although all the types implement the **IEnumerable<T>** interface (that derives from **IEnumerable**).
- Given that LINQ result sets can be represented using a good number of types in various LINQ-centric namespaces, it would be tedious to define the proper type to hold a result set, because in many cases the underlying type may not be obvious or even directly accessible from your codebase (in some cases the type is generated at compile time).
- Thankfully, implicit typing cleans things up considerably when working with LINQ queries.
- As a rule of thumb, you will always want to make use of implicit typing when capturing the results of a LINQ query. Just remember, however, that (in a vast majority of cases) the *real* return value is a type implementing the generic **IEnumerable<T>** interface.
- Exactly what this type is under the covers (**OrderedEnumerable<TElement, TKey>**, **WhereArrayIterator<T>**, etc.) is irrelevant and not necessary to discover. You can simply use the **var** keyword within a **foreach** construct to iterate over the fetched data.

```
static void QueryOverInts()
{
   int[] numbers = {10, 20, 30, 40, 1, 2, 3, 8};

   // Use implicit typing here...
   var subset = from i in numbers where i < 10 select i;

   // ...and here.
   foreach (var i in subset)
      Console.WriteLine("Item: {0} ", i);
}
```

LINQ and Extension Methods

- LINQ uses extension methods seamlessly in the background. LINQ query expressions can be used to iterate over data containers that implement the generic **IEnumerable<T>** interface. However, the .NET **System.Array** class type (used to represent the array of strings and array of integers) does *not* implement this contract.
- While **System.Array** does not directly implement the **IEnumerable<T>** interface, it indirectly gains the required functionality of this type (as well as many other LINQ-centric members) via the static **System.Linq.Enumerable** class type.
- This utility class defines a good number of generic extension methods (such as **Aggregate<T>()**, **First<T>()**, **Max<T>()**, etc.), which **System.Array** (and other types) acquires in the background. Thus, if you apply the dot operator on the currentVideoGames local variable, you will find a good number of members *not* found within the formal definition of **System.Array**.

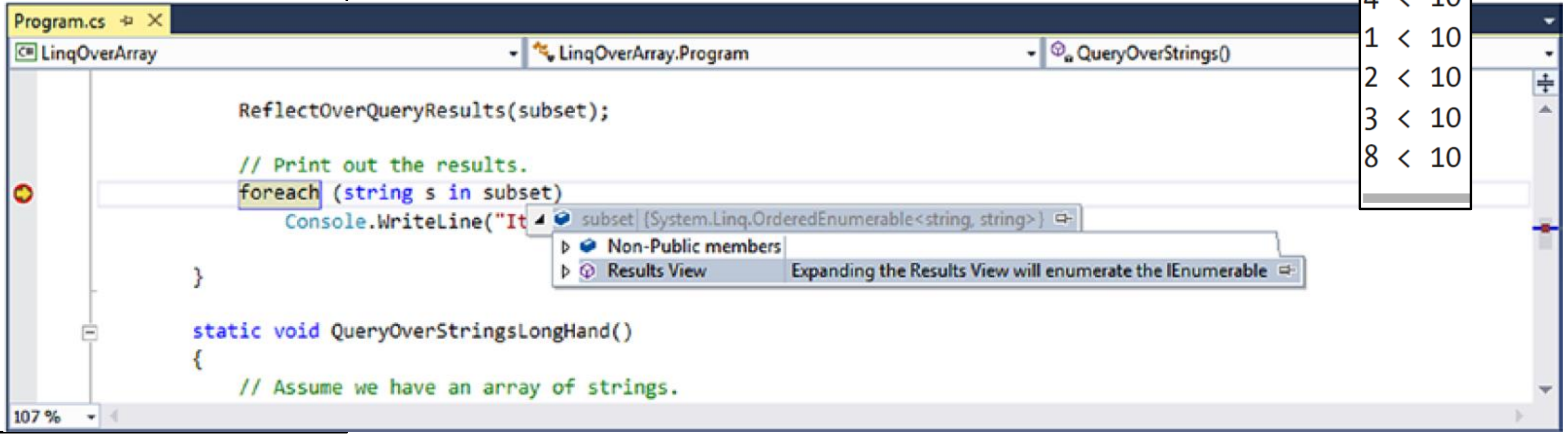


```
            // Build a query expression to find the items in the array
            // that have an embedded space.
            IEnumerable<string> subset = from g in currentVideoGames
                                         where g.Contains(" ")
                                         orderby g
                                         select g;
            currentVideoGames.

            ReflectOverQueryRe

            // Print out the r
            foreach (string s
                Console.WriteL

        }

        static void QueryOverS
        {
            // Assume we have an array of strings.
            string[] currentVideoGames = {"Morrowind", "Uncharted 2",
```

The Role of Deferred Execution

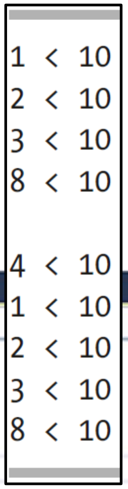
- Another important point regarding LINQ query expressions is that they are not actually evaluated until you iterate over the sequence. Formally speaking, this is termed ***deferred execution***. The benefit of this approach is that you are able to apply the same LINQ query multiple times to the same container and rest assured you are obtaining the latest and greatest results.
- One useful aspect of Visual Studio is that if you set a breakpoint before the evaluation of a LINQ query, you are able to view the contents during a debugging session. Simply locate your mouse cursor over the LINQ result set variable. When you do, you will be given the option of evaluating the query at that time by expanding the Results View option.

```
static void QueryOverInts()
{
   int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };
   // Get numbers less than ten.
   var subset = from i in numbers where i < 10 select i;
   // LINQ statement evaluated here!
   foreach (var i in subset)
      Console.WriteLine("{0} < 10", i);
   Console.WriteLine();
   // Change some data in the array.
   numbers[0] = 4;
   // Evaluated again!
   foreach (var j in subset)
      Console.WriteLine("{0} < 10", j);
   Console.WriteLine();
   ReflectOverQueryResults(subset);
}
```



```
        ReflectOverQueryResults(subset);

        // Print out the results.
        foreach (string s in subset)
          Console.WriteLine("It
      }

      static void QueryOverStringsLongHand()
      {
        // Assume we have an array of strings.
```

```
subset {System.Linq.OrderedEnumerable<string, string>}
  Non-Public members
  Results View   Expanding the Results View will enumerate the IEnumerable
```



```
1 < 10
2 < 10
3 < 10
8 < 10

4 < 10
1 < 10
2 < 10
3 < 10
8 < 10
```

The Role of Immediate Execution

- When you need to evaluate a LINQ expression from outside the confines of foreach logic, you are able to call any number of extension methods defined by the **Enumerable** type such as **ToArray<T>()**, **ToDictionary<TSource,TKey>()** and **ToList<T>()**. These methods will cause a LINQ query to execute at the exact moment you call them to obtain a snapshot of the data. After you have done so, the snapshot of data may be independently manipulated.

```
static void ImmediateExecution()
{
   int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };
   // Get data RIGHT NOW as int[].
   int[] subsetAsIntArray = (from i in numbers where i < 10 select i).ToArray<int>();
   // Get data RIGHT NOW as List<int>.
   List<int> subsetAsListOfInts = (from i in numbers where i < 10 select i).ToList<int>();
}
```

- Notice that the entire LINQ expression is wrapped within parentheses to cast it into the correct underlying type (whatever that might be) in order to call the extension methods of **Enumerable**.
- Also recall from Chapter 9 that when the C# compiler can unambiguously determine the type parameter of a generic, you are not required to specify the type parameter. Thus, you could also call **ToArray<T>()** (or **ToList<T>()** for that matter) as follows:

```
int[] subsetAsIntArray = (from i in numbers where i < 10 select i).ToArray();
```

- The usefulness of immediate execution is obvious when you need to return the results of a LINQ query to an external caller.

Returning the Result of a LINQ Query

- It is possible to define a field within a class (or structure) whose value is the result of a LINQ query. To do so, however, you cannot make use of implicit typing (as the **var** keyword cannot be used for fields), and the target of the LINQ query cannot be instance-level data; therefore, it must be **static**. Given these limitations, you will seldom need to author code like the following:

```
class LINQBasedFieldsAreClunky
{
   private static string[] currentVideoGames = {"Morrowind", "Uncharted 2", "Fallout 3", "Daxter", "System Shock 2"};

   // Can't use implicit typing here! Must know type of subset!
   private IEnumerable<string> subset = from g in currentVideoGames where g.Contains(" ") orderby g select g;

   public void PrintGames()
   {
      foreach (var item in subset)
      {
         Console.WriteLine(item);
      }
   }
}
```

Returning the Result of a LINQ Query

- More often than not, LINQ queries are defined within the scope of a method or property. Moreover, to simplify your programming, the variable used to hold the result set will be stored in an implicitly typed local variable using the **var** keyword. Implicitly typed variables cannot be used to define parameters, return values, or fields of a class or structure.
- Given this point, you might wonder exactly how you could return a query result to an external caller. The answer is, it depends. If you have a result set consisting of strongly typed data, such as an array of strings or a **List<T>** of Cars, you could abandon the use of the **var** keyword and use a proper **IEnumerable<T>** or **IEnumerable** type (again, as **IEnumerable<T>** extends **IEnumerable**).

```
class Program
{
   static void Main(string[] args)
   {
      IEnumerable<string> subset = GetStringSubset();
      foreach (string item in subset)
         Console.WriteLine(item);
      Console.ReadLine();
   }
   static IEnumerable<string> GetStringSubset()
   {
      string[] colors = {"Light Red", "Green", "Yellow", "Dark Red", "Red", "Purple"};
      // Note subset is an IEnumerable<string>-compatible object.
      IEnumerable<string> theRedColors = from c in colors where c.Contains("Red") select c;
      return theRedColors;
   }
}
```

```
Light Red
Dark Red
Red
```

Returning LINQ Query Results via Immediate Execution

- This example works as expected, only because the return value of GetStringSubset() and the LINQ query within this method has been strongly typed. If you used the **var** keyword to define the subset variable, it would be permissible to return the value *only* if the method is still prototyped to return **IEnumerable<string>** (and if the implicitly typed local variable is in fact compatible with the specified return type).
- Because it is a bit inconvenient to operate on **IEnumerable<T>**, you could make use of immediate execution. For example, rather than returning **IEnumerable<string>**, you could simply return a **string[]**, provided that you transform the sequence to a strongly typed array using the **ToArray()** method.

```
static string[] GetStringSubsetAsArray()
{
   string[] colors = {"Light Red", "Green", "Yellow", "Dark Red", "Red", "Purple"};
   var theRedColors = from c in colors where c.Contains("Red") select c;
   // Map results into an array.
   return theRedColors.ToArray();
}
```

- With this, the caller can be blissfully unaware that their result came from a LINQ query and simply work with the array of strings as expected.

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

- Immediate execution is also critical when attempting to return to the caller the results of a LINQ projection.

Applying LINQ Queries to Collection Objects

- Beyond pulling results from a simple array of data, LINQ query expressions can also manipulate data within members of the **System.Collections.Generic** namespace, such as the **List<T>** type.

```
static void Main(string[] args)
{
   // Make a List<> of Car objects.
   List<Car> myCars = new List<Car>() {
      new Car{ PetName = "Henry", Color = "Silver", Speed = 100, Make = "BMW"},
      new Car{ PetName = "Daisy", Color = "Tan", Speed = 90, Make = "BMW"},
      new Car{ PetName = "Mary", Color = "Black", Speed = 55, Make = "VW"},
      new Car{ PetName = "Clunker", Color = "Rust", Speed = 5, Make = "Yugo"},
      new Car{ PetName = "Melvin", Color = "White", Speed = 43, Make = "Ford"}
   };
   Console.ReadLine();
}
```

```
class Car
{
   public string PetName {get; set;}
   public string Color {get; set;}
   public int Speed {get; set;}
   public string Make {get; set;}
}
```

- Applying a LINQ query to a generic container is no different from doing so with a simple array, as LINQ to Objects can be used on any type implementing **IEnumerable<T>**. This time, your goal is to build a query expression to select only the **Car** objects within the **myCars** list, where the speed is greater than 55.

Accessing Contained Subobjects

- After you get the subset, you will print out the name of each **Car** object by calling the **PetName** property. Assume you have the following helper method (taking a **List<Car>** parameter), which is called from within **Main()**:
- Notice that your query expression is grabbing only those items from the **List<T>** where the Speed property > 55. If you run the application, you will find that Henry and Daisy are the only two items that match the search criteria.

```
static void GetFastCars(List<Car> myCars)
{
   // Find all Car objects in the List<>, where the Speed is greater than 55.
   var fastCars = from c in myCars where c.Speed > 55 select c;
   foreach (var car in fastCars)
   {
      Console.WriteLine("{0} is going too fast!", car.PetName);
   }
}
```

- If you want to build a more complex query, you might want to find only the BMWs that have a Speed value greater than 90. To do so, simply build a compound Boolean statement using the C# **&&** operator. In this case, the only pet name printed out is Henry.

```
static void GetFastBMWs(List<Car> myCars)
{
   // Find the fast BMWs!
   var fastCars = from c in myCars where c.Speed > 90 && c.Make == "BMW" select c;
   foreach (var car in fastCars)
   {
      Console.WriteLine("{0} is going too fast!", car.PetName);
   }
}
```

Applying LINQ Queries to Nongeneric Collections

- Recall that the query operators of LINQ are designed to work with any type implementing **IEnumerable<T>** (either directly or via extension methods). Given that **System.Array** has been provided with such necessary infrastructure, it might surprise you that the legacy (nongeneric) containers within **System.Collections** have not. Thankfully, it is still possible to iterate over data contained within nongeneric collections using the generic **Enumerable.OfType<T>()** extension method.
- When calling **OfType<T>()** from a nongeneric collection object (such as the **ArrayList**), simply specify the type of item within the container to extract a compatible **IEnumerable<T>** object. In code, you can store this data point using an implicitly typed variable.
- Consider the following new method, which fills an **ArrayList** with a set of **Car** objects. Similar to the previous examples, this method, when called from **Main()**, will display only the names Henry and Daisy, based on the format of the LINQ query.

```
static void LINQOverArrayList()
{
   // Here is a nongeneric collection of cars.
   ArrayList myCars = new ArrayList() {
      new Car{ PetName = "Henry", Color = "Silver", Speed = 100, Make = "BMW"},
      new Car{ PetName = "Daisy", Color = "Tan", Speed = 90, Make = "BMW"},
      new Car{ PetName = "Mary", Color = "Black", Speed = 55, Make = "VW"},
      new Car{ PetName = "Clunker", Color = "Rust", Speed = 5, Make = "Yugo"},
      new Car{ PetName = "Melvin", Color = "White", Speed = 43, Make = "Ford"}
   };
   // Transform ArrayList into an IEnumerable<T>-compatible type.
   var myCarsEnum = myCars.OfType<Car>();
   // Create a query expression targeting the compatible type.
   var fastCars = from c in myCarsEnum where c.Speed > 55 select c;
   foreach (var car in fastCars)
      Console.WriteLine("{0} is going too fast!", car.PetName);
}
```

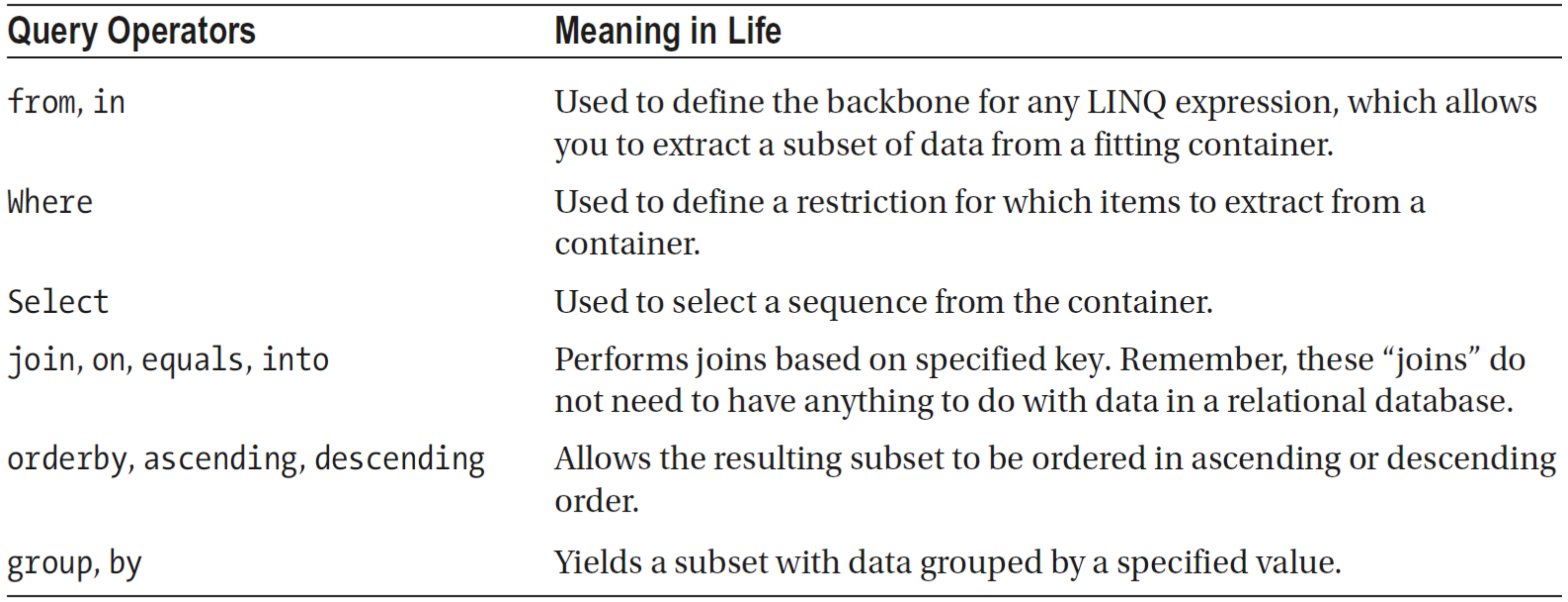
Filtering Data Using OfType<T>()

- As you know, nongeneric types are capable of containing any combination of items, as the members of these containers (again, such as the **ArrayList**) are prototyped to receive **System.Objects**. For example, assume an **ArrayList** contains a variety of items, only a subset of which are numerical. If you want to obtain a subset that contains only numerical data, you can do so using **OfType<T>()** since it filters out each element whose type is different from the given type during the iterations.

```
static void OfTypeAsFilter()
{
   // Extract the ints from the ArrayList.
   ArrayList myStuff = new ArrayList();
   myStuff.AddRange(new object[] { 10, 400, 8, false, new Car(), "string data" });
   var myInts = myStuff.OfType<int>();

   // Prints out 10, 400, and 8.
   foreach (int i in myInts)
   {
      Console.WriteLine("Int value: {0}", i);
   }
}
```

Investigating the C# LINQ Query Operators

- C# defines a good number of query operators out of the box.
- In addition to the partial list of operators shown in the table below, the **System.Linq.Enumerable** class provides a set of methods that do not have a direct C# query operator shorthand notation but are instead exposed as extension methods. These generic methods can be called to transform a result set in various manners (**Reverse<>()**, **ToArray<>()**, **ToList<>()**, etc.).
- Some are used to extract singletons from a result set, others perform various set operations (**Distinct<>()**, **Union<>()**, **Intersect<>()**, etc.).
- Others aggregate results (**Count<>()**, **Sum<>()**, **Min<>()**, **Max<>()**, etc.).



Query Operators	Meaning in Life
`from`, `in`	Used to define the backbone for any LINQ expression, which allows you to extract a subset of data from a fitting container.
`Where`	Used to define a restriction for which items to extract from a container.
`Select`	Used to select a sequence from the container.
`join`, `on`, `equals`, `into`	Performs joins based on specified key. Remember, these "joins" do not need to have anything to do with data in a relational database.
`orderby`, `ascending`, `descending`	Allows the resulting subset to be ordered in ascending or descending order.
`group`, `by`	Yields a subset with data grouped by a specified value.

Investigating the C# LINQ Query Operators

- Assume an array of **ProductInfo** objects is defined.

```
class ProductInfo
{
   public string Name {get; set;} = "";
   public string Description {get; set;} = "";
   public int NumberInStock {get; set;} = 0;
   public override string ToString() => $"Name={Name}, Description={Description}, Number in Stock={NumberInStock}";
}
```

- The array is populated **ProductInfo** objects in the **Main()** method.

```
static void Main(string[] args)
{
   // This array will be the basis of our testing...
   ProductInfo[] itemsInStock = new[] {
      new ProductInfo{ Name = "Mac's Coffee", Description = "Coffee with TEETH", NumberInStock = 24},
      new ProductInfo{ Name = "Milk Maid Milk", Description = "Milk cow's love", NumberInStock = 100},
      new ProductInfo{ Name = "Pure Silk Tofu", Description = "Bland as Possible", NumberInStock = 120},
      new ProductInfo{ Name = "Crunchy Pops", Description = "Cheezy, peppery goodness", NumberInStock = 2},
      new ProductInfo{ Name = "RipOff Water", Description = "From the tap to your wallet", NumberInStock = 100},
      new ProductInfo{ Name = "Classic Valpo Pizza", Description = "Everyone loves pizza!", NumberInStock = 73}
   };
   // We will call various methods here!
   Console.ReadLine();
}
```

Basic Selection Syntax

- Because the syntactical correctness of a LINQ query expression is validated at compile time, you need to remember that the ordering of these operators is critical. In the simplest terms, every LINQ query expression is built using the **from**, **in** and **select** operators.

```
var result = from matchingItem in container select matchingItem;
```

- The item after the **from** operator represents an item that matches the LINQ query criteria, which can be named anything you choose. The item after the **in** operator represents the data container to search (an array, collection, XML document, etc.).

- Here is a simple query, selecting every item in the container (like a SQL SELECT statement).

```
static void SelectEverything(ProductInfo[] products)
{
   // Get everything!
   Console.WriteLine("All product details:");
   var allProducts = from p in products select p;
   foreach (var prod in allProducts)
   {
      Console.WriteLine(prod.ToString());
   }
}
```

- To be honest, this query expression is not entirely useful, given that your subset is identical to that of the data in the incoming parameter. If you want, you could extract only the Name values of each car:

```
static void ListProductNames(ProductInfo[] products)
{
   // Now get only the names of the products.
   Console.WriteLine("Only product names:");
   var names = from p in products select p.Name;
   foreach (var n in names)
   {
      Console.WriteLine("Name: {0}", n);
   }
}
```

Obtaining Subsets of Data

- To obtain a specific subset from a container, you can use the **where** operator.

```
var result = from item in container where BooleanExpression select item;
```

- Notice that the **where** operator expects an expression that resolves to a **Boolean**. For example, to extract from the **ProductInfo[]** argument only the items that have more than 25 items on hand, you could author the following code:

```
static void GetOverstock(ProductInfo[] products)
{
   Console.WriteLine("The overstock items!");
   // Get only the items where we have more than 25 in stock.
   var overstock = from p in products where p.NumberInStock > 25 select p;
   foreach (ProductInfo c in overstock)
      Console.WriteLine(c.ToString());
}
```

- As shown earlier in this chapter, when you are building a **where** clause, it is permissible to make use of any valid **C# operators** to build **complex expressions**. For example, recall this query that extracts out only the BMWs going at least 100 mph:

```
// Get BMWs going at least 100 mph.
var onlyFastBMWs = from c in myCars where c.Make == "BMW" && c.Speed >= 100 select c;
foreach (Car c in onlyFastBMWs)
   Console.WriteLine("{0} is going {1} MPH", c.PetName, c.Speed);
```

Projecting New Data Types

- It is also possible to **project** new forms of data from an existing data source. Let's assume you want to take the incoming **ProductInfo[]** parameter and obtain a result set that accounts only for the name and description of each item. To do so, you can define a select statement that dynamically yields a **new anonymous type**.

```
static void GetNamesAndDescriptions(ProductInfo[] products)
{
   Console.WriteLine("Names and Descriptions:");
   var nameDesc = from p in products select new { p.Name, p.Description };
   foreach (var item in nameDesc)
   {
      // Could also use Name and Description properties directly.
      Console.WriteLine(item.ToString());
   }
}
```

- Always remember that when you have a LINQ query that makes use of a **projection**, you have no way of knowing the underlying data type, as this is determined at compile time. In these cases, the **var** keyword is mandatory. As well, recall that you cannot create methods with implicitly typed return values. Therefore, the following method would not compile:

```
static var GetProjectedSubset(ProductInfo[] products)
{
   var nameDesc = from p in products select new { p.Name, p.Description };
   return nameDesc; // Nope!
}
```

Projecting New Data Types

- When you need to return **projected data** to a caller, one approach is to transform the query result into a .NET **System.Array** object using the **ToArray()** extension method.

```
// Return value is now an Array.
static Array GetProjectedSubset(ProductInfo[] products)
{
   var nameDesc = from p in products select new { p.Name, p.Description };
   // Map set of anonymous objects to an Array object.
   return nameDesc.ToArray();
}
```

- You could invoke and process the data from Main() as follows:

```
Array objs = GetProjectedSubset(itemsInStock);
foreach (object o in objs)
   Console.WriteLine(o); // Calls ToString() on each anonymous object.
```

- Note that you must use a literal **System.Array** object and cannot make use of the C# array declaration syntax, given that you don’t know the underlying type of type because you are operating on a compiler-generated anonymous class! Also note that you are not specifying the **type parameter** to the generic **ToArray<T>()** method, as you once again don’t know the underlying data type until compile time, which is too late for your purposes.
- The obvious problem is that you lose any strong typing, as each item in the **Array** object is assumed to be of type **Object**.

Obtaining Counts and Reversing Result Sets

- When **projecting data**, you may need to discover exactly how many items have been returned into the sequence. Any time you need to determine the number of items returned from a LINQ query expression, simply use the **Count()** extension method of the **Enumerable** class.

```
static void GetCountFromQuery()
{
   string[] currentVideoGames = {"Morrowind", "Uncharted 2", "Fallout 3", "Daxter", "System Shock 2"};

   // Get count from the query.
   int numb = (from g in currentVideoGames where g.Length > 6 select g).Count();

   // Print out the number of items.
   Console.WriteLine("{0} items honor the LINQ query.", numb);
}
```

- You can reverse the items within a result set quite simply using the **Reverse<>()** extension method of the **Enumerable** class. For example, the following method selects all items from the incoming **ProductInfo[]** parameter, in reverse.

```
static void ReverseEverything(ProductInfo[] products)
{
   Console.WriteLine("Product in reverse:");
   var allProducts = from p in products select p;
   foreach (var prod in allProducts.Reverse())
   {
      Console.WriteLine(prod.ToString());
   }
}
```

Sorting Expressions

- As you have seen over this chapter's initial examples, a query expression can take an **orderby** operator to sort items in the subset by a specific value. By default, the order will be **ascending**; thus, ordering by a string would be alphabetical, ordering by numerical data would be lowest to highest, and so forth. If you need to view the results in a descending order, simply include the **descending** operator.

```
static void AlphabetizeProductNames(ProductInfo[] products)
{
   // Get names of products, alphabetized.
   var subset = from p in products orderby p.Name select p;
   Console.WriteLine("Ordered by Name:");
   foreach (var p in subset)
   {
      Console.WriteLine(p.ToString());
   }
}
```

- You are able to make your intentions clear by using the **ascending** operator explicitly.

```
var subset = from p in products orderby p.Name ascending select p;
```

- If you want to get the items in descending order, you can do so via the **descending** operator.

```
var subset = from p in products orderby p.Name descending select p;
```

LINQ as a Better Venn Diagramming Tool

- The **Enumerable** class supports a set of extension methods that allows you to use two (or more) LINQ queries as the basis to find **unions**, **differences**, **concatenations** and **intersections** of data. First, consider the **Except()** extension method, which will return a LINQ result set that contains the differences between two containers, which, in this case, is the value Yugo:

```
static void DisplayDiff()
{
   List<string> myCars = new List<String> {"Yugo", "Aztec", "BMW"};
   List<string> yourCars = new List<String>{"BMW", "Saab", "Aztec" };
   var carDiff = (from c in myCars select c).Except(from c2 in yourCars select c2);
   Console.WriteLine("Here is what you don't have, but I do:");
   foreach (string s in carDiff)
      Console.WriteLine(s); // Prints Yugo.
}
```

- The **Intersect()** method will return a result set that contains the common data items in a set of containers. For example, the following method returns the sequence Aztec and BMW.

```
static void DisplayIntersection()
{
   List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
   List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };
   // Get the common members.
   var carIntersect = (from c in myCars select c).Intersect(from c2 in yourCars select c2);
   Console.WriteLine("Here is what we have in common:");
   foreach (string s in carIntersect)
      Console.WriteLine(s); // Prints Aztec and BMW.
}
```

LINQ as a Better Venn Diagramming Tool

- The **Union()** method, as you would guess, returns a result set that includes all members of a batch of LINQ queries. Like any proper **union**, you will not find repeating values if a common member appears more than once. Therefore, the following method will print out the values Yugo, Aztec, BMW, and Saab:

```
static void DisplayUnion()
{
   List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
   List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };
   // Get the union of these containers.
   var carUnion = (from c in myCars select c).Union(from c2 in yourCars select c2);
   Console.WriteLine("Here is everything:");
   foreach (string s in carUnion)
      Console.WriteLine(s); // Prints all common members.
}
```

- Finally, the **Concat()** extension method returns a result set that is a direct concatenation of LINQ result sets. For example, the following method prints out the results Yugo, Aztec, BMW, BMW, Saab, and Aztec:

```
static void DisplayConcat()
{
   List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
   List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };
   var carConcat = (from c in myCars select c).Concat(from c2 in yourCars select c2);
   // Prints:
   // Yugo Aztec BMW BMW Saab Aztec.
   foreach (string s in carConcat)
      Console.WriteLine(s);
}
```

Removing Duplicates

- When you call the **Concat()** extension method, you could very well end up with redundant entries in the fetched result, which could be exactly what you want in some cases. However, in other cases, you might want to remove duplicate entries in your data. To do so, simply call the **Distinct()** extension method, as shown here.

```
static void DisplayConcatNoDups()
{
   List<string> myCars = new List<String> { "Yugo", "Aztec", "BMW" };
   List<string> yourCars = new List<String> { "BMW", "Saab", "Aztec" };
   var carConcat = (from c in myCars select c).Concat(from c2 in yourCars select c2);

   // Prints:
   // Yugo Aztec BMW Saab.
   foreach (string s in carConcat.Distinct())
      Console.WriteLine(s);
}
```

LINQ Aggregation Operations

- LINQ queries can also be designed to perform various aggregation operations on the result set. The **Count()** extension method is one such aggregation example. Other possibilities include obtaining an **average**, **maximum**, **minimum** or **sum** of values using the **Max()**, **Min()**, **Average()** or **Sum()** members of the **Enumerable** class.

```
static void AggregateOps()
{
   double[] winterTemps = { 2.0, -21.3, 8, -4, 0, 8.2 };
   // Various aggregation examples.
   Console.WriteLine("Max temp: {0}", (from t in winterTemps select t).Max());
   Console.WriteLine("Min temp: {0}", (from t in winterTemps select t).Min());
   Console.WriteLine("Average temp: {0}", (from t in winterTemps select t).Average());
   Console.WriteLine("Sum of all temps: {0}", (from t in winterTemps select t).Sum());
}
```

The Internal Representation of LINQ Query Statements

- The C# compiler translates all C# LINQ operators into calls on methods of the **Enumerable** class.
- A great many of the methods of **Enumerable** have been prototyped to take **delegates** as arguments. In particular, many methods require a generic delegate **Func<>**. Consider the **Where()** method of **Enumerable**, which is called on your behalf when you use the C# **where** LINQ query operator.

```
// Overloaded versions of the Enumerable.Where<T>() method.
// Note the second parameter is of type System.Func<>.

public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source, System.Func<TSource,int,bool> predicate)

public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source, System.Func<TSource,bool> predicate)
```

The Internal Representation of LINQ Query Statements

- The **Func<>** delegate (as the name implies) represents a pattern for a given function with a set of up to 16 arguments and a return value. If you were to examine this type using the Visual Studio object browser, you would notice various forms of the **Func<>** delegate.

```
// The various formats of the Func<> delegate.
public delegate TResult Func<T1,T2,T3,T4,TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)
public delegate TResult Func<T1,T2,T3,TResult>(T1 arg1, T2 arg2, T3 arg3)
public delegate TResult Func<T1,T2,TResult>(T1 arg1, T2 arg2)
public delegate TResult Func<T1,TResult>(T1 arg1)
public delegate TResult Func<TResult>()
```

- Given that many members of **System.Linq.Enumerable** demand a delegate as input, when invoking them, you can either manually create a new **delegate type** and author the necessary target methods, make use of a **C# anonymous method** or define a proper **lambda expression**. Regardless of which approach you take, the end result is identical.

Building Query Expressions

- Using **query operators**, **lambda expressions** and **anonymous methods**.

```
static void QueryStringWithOperators()
{
   string[] currentVideoGames = {"Morrowind", "Uncharted 2", "Fallout 3", "Daxter", "System Shock 2"};
   var subset = from game in currentVideoGames where game.Contains(" ") orderby game select game;
   foreach (string s in subset)
      Console.WriteLine("Item: {0}", s);
}
```

```
static void QueryStringsWithEnumerableAndLambdas()
{
   string[] currentVideoGames = {"Morrowind", "Uncharted 2", "Fallout 3", "Daxter", "System Shock 2"};
   var subset = currentVideoGames.Where(game => game.Contains(" ")).OrderBy(game => game).Select(game => game);
   foreach (var game in subset)
      Console.WriteLine("Item: {0}", game);
}
```

```
static void QueryStringsWithAnonymousMethods()
{
   string[] currentVideoGames = {"Morrowind", "Uncharted 2", "Fallout 3", "Daxter", "System Shock 2"};
   Func<string, bool> searchFilter = delegate(string game) { return game.Contains(" "); };
   Func<string, string> itemToProcess = delegate(string s) { return s; };
   var subset = currentVideoGames.Where(searchFilter).OrderBy(itemToProcess).Select(itemToProcess);
   foreach (var game in subset)
      Console.WriteLine("Item: {0}", game);
}
```

Building Query Expressions

- Keep the following points in mind regarding how LINQ query expressions are represented under the covers:
  - Query expressions are created using various C# **query operators**.
  - Query operators are simply shorthand notations for invoking **extension methods** defined by the **System.Linq.Enumerable** type.
  - Many methods of Enumerable require **delegates** (**Func<>** in particular) as parameters.
  - Any method requiring a delegate parameter can instead be passed a **lambda expression**.
  - Lambda expressions are simply **anonymous methods** in disguise.
  - Anonymous methods are shorthand notations for allocating a **raw delegate** and manually building a **delegate target method**.