The Addison-Wesley Microsoft Technology Series

LINQ to Objects Using C# 4.0

Using and Extending LINQ to Objects and Parallel LINQ (PLINQ)

Troy Magennis

Foreword by Barry Vandevier, *Chief Information Officer, Sabre Holdings*

LINQ to Objects Using C# 4.0

The null-coalescing operator is used to define a default value if the variable it follows has a null value, that is, `X = (variable) ?? (default if null)`. Listing 4-2 demonstrates this usage.

Listing 4-2 Example of using the null-coalescing operator and the ternary operator to protect **`keySelector`** expressions from null values

```
// Guard against null data using
// ternary (? as shown)
var q1 = from c in Contact.SampleData()
         group c by
             c.State == null ? "(null)" : c.State;

// Guard against null data using
// the null coalescing operator (??).
var q2 = from c in Contact.SampleData()
         group c by
             c.State ?? "(null)";
```

Grouping by Composite Keys (More Than One Value)

To group using more than one value as the key (often referred to as a composite key), you specify the grouping selector clause as an anonymous type (anonymous types are introduced in Chapter 2, "Introducing LINQ to Objects"). Any number of key values can be specified in this anonymous type, and any element that contains identical values will dutifully be co-located in a group.

Listing 4-3 demonstrates the simplicity of specifying multiple key values in the group by expression. In this case, the `LastName` and the `State` fields are used for grouping, placing all contacts with the same last name from the same state in a group. The Console output from this example is shown in Output 4-2.

Listing 4-3 Anonymous types can be used to group by more than one value (composite key)—see Output 4-2

```
/* this sample uses the same data as we saw in Table 2-1,
   but i've added 2 Gottshall's (one from the same state
   and another out of that state), and 2 Gauwain's -
```

```
        Firstname   Lastname    State
        -----------------------------
        Barney      Gottshall   CA
        Mandy       Gottshall   CA   *added
        Bernadette  Gottshall   WA   *added
        Armando     Valdes      WA
        Adam        Gauwain     AK
        Chris       Gauwain     AK   *added
        Anthony     Gauwain     CA   *added
        Jeffery     Deane       CA
        Collin      Zeeman      FL
        Stewart     Kagel       WA
        Chance      Lard        WA
        Blaine      Reifsteck   TX
        Mack        Kamph       TX
        Ariel       Hazelgrove  OR
*/

var q = from c in Contact.SampleData()
        group c by new { c.LastName, c.State };

foreach (var grp in q)
{
    Console.WriteLine("Group - {0}, {1} - count = {2}",
        grp.Key.LastName,
        grp.Key.State,
        grp.Count());
}
```

Output 4-2

```
Group - Gottshall, CA - count = 2
Group - Gottshall, WA - count = 1
Group - Valdes, WA - count = 1
Group - Gauwain, AK - count = 2
Group - Gauwain, CA - count = 1
Group - Deane, CA - count = 1
Group - Zeeman, FL - count = 1
Group - Kagel, WA - count = 1
Group - Lard, WA - count = 1
Group - Reifsteck, TX - count = 1
Group - Kamph, TX - count = 1
Group - Hazelgrove, OR - count = 1
```

It is not essential to use an anonymous type as the grouping key selector to achieve multiple-property groups. A named type containing all of the properties participating in the key can be used for the same purpose, although the method is more complex because the type being constructed in the projection needs a custom override of the methods `GetHashCode` and `Equals` to force comparison by property values rather than reference equality. Listing 4-4 demonstrates the mechanics of creating a class that supports composite keys using the fields `LastName` and `State`. The results are identical to that shown in Output 4-2.

NOTE Writing good `GetHashCode` implementations is beyond the scope of this book. I've used a simple implementation in this example—for more details see the MSDN article for recommendations of how to override the `GetHashCode` implementation available at http://msdn.microsoft.com/en-us/library/system.object.gethashcode.aspx.

Listing 4-4 Creating a composite join key using a normal class type

```
public class LastNameState
{
    public string LastName { get; set; }
    public string State { get; set; }

    // follow the MSDN guidelines -
    // http://msdn.microsoft.com/en-us/library/
    //                          ms173147(VS.80).aspx
    public override bool Equals(object obj)
    {
        if (this != obj)
        {
            LastNameState item = obj as LastNameState;
            if (item == null) return false;
            if (State != item.State) return false;
            if (LastName != item.LastName) return false;
        }

        return true;
    }
```

```
    // follow the MSDN guidelines -
    // http://msdn.microsoft.com/en-us/library/
    //             system.object.gethashcode.aspx
    public override int GetHashCode()
    {
        int result = State != null ? State.GetHashCode() : 1;
        result = result ^
              (LastName != null ? LastName.GetHashCode() : 2);

        return result;
    }
}

var q = from c in Contact.SampleData()
        group c by new LastNameState {
            LastName = c.LastName, State = c.State };
```

Specifying Your Own Key Comparison Function

The default behavior of grouping is to equate key equality using the normal equals comparison for the type being tested. This may not always suit your needs, and it is possible to override this behavior and specify your own grouping function. To implement a custom comparer, you build a class that implements the interface `IEqualityComparer<T>` and pass this class as an argument into the `GroupBy` extension method.

The `IEqualityComparer<T>` interface definition has the following definition:

```
public interface IEqualityComparer<T>
{
    public bool Equals(T x, T y)
    public int GetHashCode(T obj)
}
```

The `Equals` method is used to indicate that one instance of an object has the same equality to another instance of an object. Overriding this method allows specific logic to determine object equality based on the data within those objects, rather than instances being the same object instance. (That is, even though two objects were constructed at different times and are two different objects as far as the compiler is concerned, we want them to be deemed equal based on their specific combination of internal values or algorithm.)

The `GetHashCode` method is intended for use in hashing algorithms and data structures such as a hash table. The implementation of this method returns an integer value based on at least one value contained in the instance of an object. The resulting hash code can be used by other data structures as a unique value (but not guaranteed unique). It is intended as a quick way of segmenting instances in a collection or as a short-cut check for value equality (although a further check needs to be carried out for you to be certain of object value equality). The algorithm that computes the hash code should return the same integer value when two instances of an object have the same values (in the data fields being checked), and it should be fast, given that this method will be called many times during the grouping evaluation process.

The following example demonstrates one possible use of a custom equality comparer function in implementing a simple Soundex comparison routine (see http://en.wikipedia.org/wiki/Soundex for a full definition of the Soundex algorithm) that will group phonetically similar names. The code for the `SoundexEqualityComparer` is shown in Listing 4-5. Soundex is an age-old algorithm that computes a reliable four character string value based on the phonetics of a given word; an example would be "Katie" is phonetically identical to "Katy."

The approach for building the Soundex equality operator is to

1. Code the Soundex algorithm to return a four-character string result representing phonetic sounding given a string input. The form of the Soundex code is a single character, followed by three numerical digits, for example A123 or V456.
2. Implement the `GetHashCode` for a given string. This will call the Soundex method and then convert the Soundex code to an integer value. It builds the integer by using the ASCII value of the character, multiplying it by 1000 and then adding the three digit suffix of the Soundex code to this number, for example A123 would become, (65 x 1000) + 123 = 65123.
3. Implement the `Equals` method by calling `GetHashCode` on both input arguments `x` and `y` and then comparing the return integer results. Return true if the hash codes match (`GetHashCode` can be used in this implementation of overloading the `Equals` operator because it is known that the Soundex algorithm implementation returns a unique value—this is not the case with other `GetHashCode` implementations).

Care must be taken for null values and empty strings in deciding what behavior you want. I decided that I wanted null or empty string entries to be in one group, but this null handling logic should be considered for each specific implementation. (Maybe an empty string should be in a different group than null entries; it really depends on the specific situation.)

Listing 4-5 The custom **`SoundexEqualityComparer`** allows phonetically similar sounding strings to be easily grouped

```
public class SoundexEqualityComparer
    : IEqualityComparer<string>
{
    public bool Equals(string x, string y)
    {
        return GetHashCode(x) == GetHashCode(y);
    }

    public int GetHashCode(string obj)
    {
        // E.g. convert soundex code A123,
        // to an integer: 65123
        int result = 0;

        string s = soundex(obj);
        if (string.IsNullOrEmpty(s) == false)
            result = Convert.ToInt32(s[0]) * 1000 +
                     Convert.ToInt32(s.Substring(1, 3));

        return result;
    }

    private string soundex(string s)
    {
        // Algorithm as listed on
        //     http://en.wikipedia.org/wiki/Soundex.
        // builds a string code in the format:
        //     [A-Z][0-6][0-6][0-6]
        // based on the phonetic sound of the input.

        if (String.IsNullOrEmpty(s))
            return null;
```