**Introducing LINQ**

Language Integrated Query (LINQ) defines a range of standard query operators that enable you to retrieve exactly the data that you require in a declarative way.  LINQ also supports compile-time syntax-checking and type-checking and also uses Microsoft IntelliSense® in Visual Studio.

You can use LINQ to query data from a wide range of data sources, including .NET Framework collections, Microsoft SQL Server® databases, ADO.NET data sets, and XML documents. In fact, you can use it to query any data source that implements the IEnumerable interface.

The syntax of all LINQ queries has the same basis, as follows:

**from** <variable names> **in** <data source>  
**select** <variable names>

However, you can customize this syntax in many ways to retrieve exactly the data that you require in the format that you want.  In this module, you will learn how to query data and use anonymous methods and how to force query execution to override the default deferred query execution behavior.

**Selecting Data**

The following code example shows how to use a simple select clause to return all of the data in a single entity.

// Using a select Clause  
IQueryable<Employee> emps = from e in Employees select e;

The return data type from the query is an IQueryable<Employee>, enabling you to iterate through the data that is returned.

**Filtering Data by Row**  
The following code example shows how to use the where keyword to filter the returned data by row to contain only employees with a last name of Prescott.

// Using a where Clause  
string \_LastName = "Prescott";  
IQueryable<Employee> emps = from e in Employees  
                     where e.LastName == \_LastName  
                     select e;

**Filtering Data by Column**

The following code example shows how to declare a new type in which to store a subset of columns that the query returns; in this case, just the FirstName and LastName properties of the Employee entity.

// Using a New Class to Return a Subset of Columns  
private class FullName  
{  
    public string Forename { get; set; }  
    public string Surname { get; set; }  
}  
private void FilteringDataByColumn()  
{  
    IQueryable<FullName> names = from e in Employees  
                         select new FullName { Forename = e.FirstName, Surname = e.LastName };  
}

**Working with the Results**

To then work with the data that is returned from any of these queries, you use dot notation to access the properties of the members of the IQueryable<> type, as the following code example shows.

// Accessing the Returned Data   
foreach (var emp in emps)  
{  
    Console.WriteLine("{0} {1}", emp.FirstName, emp.LastName);  
}

**Grouping Data**

The following code example shows how to use a group clause to group the returned employees by their job title ID.

// Using a group Clause  
var emps = from e in Employees  
                   group e by e.JobTitle into eGroup  
                   select new { Job = eGroup.Key, Names = eGroup };

**Aggregating Data**

The following code example shows how to use a group clause with an aggregate function to count the number of employees with each job title.

// Using a group Clause with an Aggregate Function  
var emps = from e in Employees  
                   group e by e.JobTitle into eGroup  
                   select new { Job = eGroup.Key, CountOfEmployees = eGroup.Count() };

**Navigating Data**

The following code example shows how to use navigation properties to retrieve data from the Employees entity and the related JobTitles entity.

// Using Dot Notation to Navigate Related Entities  
var emps = from e in Employees  
select new  
{  
    FirstName = e.FirstName, LastName = e.LastName, Job = e.JobTitle1.Job  
};

**LINQ Joins**

You can use LINQ to join data from different sources through the one of three join types, inner, group, and left outer.  The disparate data must have a common element value that canbe compared.  When thinking about a relational database join operation on an orders and order details set of tables, you would typically use an order id as the common value.

**Inner Join**

The followng code is an example of an inner join in LINQ:

var innerJoinQuery =  
from order in orders  
join prod in products on category.ID equals prod.CategoryID  
select new { ProductName = prod.Name, Category = category.Name }; //produces flat sequence

**Group Join**

A join clause with an into expression is called a group join as shown in this code sample:  
  
var innerGroupJoinQuery =  
from category in categories  
join prod in products on category.ID equals prod.CategoryID into prodGroup  
select new { CategoryName = category.Name, Products = prodGroup };  
  
A group join produces a hierarchical result associating the elements returned from the left table (source) with one or more matching elements from right side data source. If you are familiar with relational database joins, you may find that group joins have no equivalent in relational database join terminology. What gets returned with a group join is a sequence of object arrays.  If no matching elements are found, the join clause will produce an empty array for that item.   
  
**Left Outer Join**  
  
In a left outer join, all elements in the left source are returned, even if there are no matching elements are in the right sequence. A left outer join is performed using the DefaultIfEmpty method in combination with a group join. This is done to specify a default right-side element to produce in the even that a left-side element is not matched. The following code sample demonstrates this:  
  
var leftOuterJoinQuery =  
from category in categories  
join prod in products on category.ID equals prod.CategoryID into prodGroup  
from item in prodGroup.DefaultIfEmpty(new Product { Name = String.Empty, CategoryID = 0 })  
select new { CatName = category.Name, ProdName = item.Name };