

# Contents

## Language-Integrated Query (LINQ)

### Overview

### Getting Started with LINQ in C#

#### Introduction to LINQ Queries

#### LINQ and Generic Types

#### Basic LINQ Query Operations

#### Data Transformations with LINQ

#### Type Relationships in LINQ Query Operations

#### Query Syntax and Method Syntax in LINQ

#### C# Features That Support LINQ

#### Walkthrough: Writing Queries in C# (LINQ)

## Standard Query Operators Overview

### Overview

### Query Expression Syntax for Standard Query Operators

### Classification of Standard Query Operators by Manner of Execution

### Sorting Data

### Set Operations

### Filtering Data

### Quantifier Operations

### Projection Operations

### Partitioning Data

### Join Operations

### Grouping Data

### Generation Operations

### Equality Operations

### Element Operations

### Converting Data Types

### Concatenation Operations

### Aggregation Operations

## [LINQ to Objects](#)

### [Overview](#)

### [LINQ and Strings](#)

### [How to articles](#)

[How to: Count Occurrences of a Word in a String \(LINQ\)](#)

[How to: Query for Sentences that Contain a Specified Set of Words \(LINQ\)](#)

[How to: Query for Characters in a String \(LINQ\)](#)

[How to: Combine LINQ Queries with Regular Expressions](#)

[How to: Find the Set Difference Between Two Lists \(LINQ\)](#)

[How to: Sort or Filter Text Data by Any Word or Field \(LINQ\)](#)

[How to: Reorder the Fields of a Delimited File \(LINQ\)](#)

[How to: Combine and Compare String Collections \(LINQ\)](#)

[How to: Populate Object Collections from Multiple Sources \(LINQ\)](#)

[How to: Split a File Into Many Files by Using Groups \(LINQ\)](#)

[How to: Join Content from Dissimilar Files \(LINQ\)](#)

[How to: Compute Column Values in a CSV Text File \(LINQ\)](#)

### [LINQ and Reflection](#)

[How to: Query An Assembly's Metadata with Reflection \(LINQ\)](#)

### [LINQ and File Directories](#)

#### [Overview](#)

[How to: Query for Files with a Specified Attribute or Name](#)

[How to: Group Files by Extension \(LINQ\)](#)

[How to: Query for the Total Number of Bytes in a Set of Folders \(LINQ\)](#)

[How to: Compare the Contents of Two Folders \(LINQ\)](#)

[How to: Query for the Largest File or Files in a Directory Tree \(LINQ\)](#)

[How to: Query for Duplicate Files in a Directory Tree \(LINQ\)](#)

[How to: Query the Contents of Files in a Folder \(LINQ\)](#)

[How to: Query an ArrayList with LINQ](#)

[How to: Add Custom Methods for LINQ Queries](#)

## [LINQ to XML](#)

[Getting Started \(LINQ to XML\)](#)

[LINQ to XML Overview](#)

[LINQ to XML vs. DOM](#)

[LINQ to XML vs. Other XML Technologies](#)

[Programming Guide \(LINQ to XML\)](#)

[LINQ to XML Programming Overview](#)

[Functional vs. Procedural Programming \(LINQ to XML\)](#)

[LINQ to XML Classes Overview](#)

[XElement Class Overview](#)

[XAttribute Class Overview](#)

[XDocument Class Overview](#)

[How to: Build LINQ to XML Examples](#)

[XML Trees](#)

[Functional Construction \(LINQ to XML\)](#)

[Creating XML Trees in C# \(LINQ to XML\)](#)

[Parsing XML](#)

[How to: Parse a String](#)

[How to: Load XML from a File](#)

[Preserving White Space while Loading or Parsing XML](#)

[How to: Catch Parsing Errors](#)

[How to: Create a Tree from an XmlReader](#)

[How to: Stream XML Fragments from an XmlReader](#)

[How to: Populate an XML Tree with an XmlWriter \(LINQ to XML\)](#)

[How to: Validate Using XSD \(LINQ to XML\)](#)

[Valid Content of XElement and XDocument Objects](#)

[Working with XML Namespaces](#)

[Namespaces Overview \(LINQ to XML\)](#)

[How to: Create a Document with Namespaces \(LINQ to XML\)](#)

[How to: Control Namespace Prefixes \(LINQ to XML\)](#)

[Scope of Default Namespaces in C#](#)

[How to: Write Queries on XML in Namespaces](#)

[Serializing XML Trees](#)

[Preserving White Space While Serializing](#)

[Serializing with an XML Declaration](#)

[Serializing to Files, TextWriters, and XmlWriters](#)

[Serializing to an XmlReader \(Invoking XSLT\)](#)

## [LINQ to XML Axes](#)

[LINQ to XML Axes Overview](#)

[How to: Retrieve a Collection of Elements \(LINQ to XML\)](#)

[How to: Retrieve the Value of an Element \(LINQ to XML\)](#)

[How to: Filter on Element Names \(LINQ to XML\)](#)

[How to: Chain Axis Method Calls \(LINQ to XML\)](#)

[How to: Retrieve a Single Child Element \(LINQ to XML\)](#)

[How to: Retrieve a Collection of Attributes \(LINQ to XML\)](#)

[How to: Retrieve a Single Attribute \(LINQ to XML\)](#)

[How to: Retrieve the Value of an Attribute \(LINQ to XML\)](#)

[How to: Retrieve the Shallow Value of an Element](#)

## [Querying XML Trees](#)

[Basic Queries \(LINQ to XML\)](#)

[How to: Find an Element with a Specific Attribute](#)

[How to: Find an Element with a Specific Child Element](#)

[Querying an XDocument vs. Querying an XElement](#)

[How to: Find Descendants with a Specific Element Name](#)

[How to: Find a Single Descendant Using the Descendants Method](#)

[How to: Write Queries with Complex Filtering](#)

[How to: Filter on an Optional Element](#)

[How to: Find All Nodes in a Namespace](#)

[How to: Sort Elements](#)

[How to: Sort Elements on Multiple Keys](#)

[How to: Calculate Intermediate Values](#)

[How to: Write a Query that Finds Elements Based on Context](#)

[How to: Debug Empty Query Results Sets](#)

## [Projections and Transformations \(LINQ to XML\)](#)

[How to: Work with Dictionaries Using LINQ to XML](#)

[How to: Transform the Shape of an XML Tree](#)

[How to: Control the Type of a Projection](#)

[How to: Project a New Type \(LINQ to XML\)](#)

[How to: Project an Object Graph](#)

[How to: Project an Anonymous Type](#)

[How to: Generate Text Files from XML](#)

[How to: Generate XML from CSV Files](#)

## [Advanced Query Techniques \(LINQ to XML\)](#)

[How to: Join Two Collections \(LINQ to XML\)](#)

[How to: Create Hierarchy Using Grouping](#)

[How to: Query LINQ to XML Using XPath](#)

[How to: Write a LINQ to XML Axis Method](#)

[How to: Perform Streaming Transformations of Text to XML](#)

[How to: List All Nodes in a Tree](#)

[How to: Retrieve Paragraphs from an Office Open XML Document](#)

[How to: Modify an Office Open XML Document](#)

[How to: Populate an XML Tree from the File System](#)

## [LINQ to XML for XPath Users](#)

[Comparison of XPath and LINQ to XML](#)

[How to: Find a Child Element \(XPath-LINQ to XML\)](#)

[How to: Find a List of Child Elements \(XPath-LINQ to XML\)](#)

[How to: Find the Root Element \(XPath-LINQ to XML\)](#)

[How to: Find Descendant Elements \(XPath-LINQ to XML\)](#)

[How to: Filter on an Attribute \(XPath-LINQ to XML\)](#)

[How to: Find Related Elements \(XPath-LINQ to XML\)](#)

[How to: Find Elements in a Namespace \(XPath-LINQ to XML\)](#)

[How to: Find Preceding Siblings \(XPath-LINQ to XML\)](#)

[How to: Find Descendants of a Child Element \(XPath-LINQ to XML\)](#)

[How to: Find a Union of Two Location Paths \(XPath-LINQ to XML\)](#)

[How to: Find Sibling Nodes \(XPath-LINQ to XML\)](#)

[How to: Find an Attribute of the Parent \(XPath-LINQ to XML\)](#)

[How to: Find Attributes of Siblings with a Specific Name \(XPath-LINQ to XML\)](#)

[How to: Find Elements with a Specific Attribute \(XPath-LINQ to XML\)](#)

[How to: Find Child Elements Based on Position \(XPath-LINQ to XML\)](#)

[How to: Find the Immediate Preceding Sibling \(XPath-LINQ to XML\)](#)

## [Pure Functional Transformations of XML](#)

[Introduction to Pure Functional Transformations](#)

[Concepts and Terminology \(Functional Transformation\)](#)

[Functional Programming vs. Imperative Programming](#)

[Refactoring Into Pure Functions](#)

[Applicability of Functional Transformation](#)

[Functional Transformation of XML](#)

[Tutorial: Chaining Queries Together](#)

[Tutorial: Chaining Queries Together](#)

[Tutorial: Manipulating Content in a WordprocessingML Document](#)

## [Modifying XML Trees \(LINQ to XML\)](#)

[In-Memory XML Tree Modification vs. Functional Construction \(LINQ to XML\)](#)

[Adding Elements, Attributes, and Nodes to an XML Tree](#)

[Modifying Elements, Attributes, and Nodes in an XML Tree](#)

[Removing Elements, Attributes, and Nodes from an XML Tree](#)

[Maintaining Name-Value Pairs](#)

[How to: Change the Namespace for an Entire XML Tree](#)

## [Performance \(LINQ to XML\)](#)

[Performance of Chained Queries \(LINQ to XML\)](#)

[Atomized XName and XNamespace Objects \(LINQ to XML\)](#)

[Pre-Atomization of XName Objects \(LINQ to XML\)](#)

[Statically Compiled Queries \(LINQ to XML\)](#)

## [Advanced LINQ to XML Programming](#)

[LINQ to XML Annotations](#)

[LINQ to XML Events](#)

[Programming with Nodes](#)

[Mixed Declarative Code-Imperative Code Bugs \(LINQ to XML\)](#)

[How to: Stream XML Fragments with Access to Header Information](#)

[How to: Perform Streaming Transform of Large XML Documents](#)

[How to: Read and Write an Encoded Document](#)

[Using XSLT to Transform an XML Tree](#)

[How to: Use Annotations to Transform LINQ to XML Trees in an XSLT Style](#)

[Serializing Object Graphs that Contain XElement Objects](#)

[How to: Serialize Using XmlSerializer](#)

[How to: Serialize Using DataContractSerializer](#)

[LINQ to XML Security](#)

[Sample XML Documents \(LINQ to XML\)](#)

[Sample XML File: Typical Purchase Order \(LINQ to XML\)](#)

[Sample XML File: Typical Purchase Order in a Namespace](#)

[Sample XML File: Multiple Purchase Orders \(LINQ to XML\)](#)

[Sample XML File: Multiple Purchase Orders in a Namespace](#)

[Sample XML File: Test Configuration \(LINQ to XML\)](#)

[Sample XML File: Test Configuration in a Namespace](#)

[Sample XML File: Customers and Orders \(LINQ to XML\)](#)

[Sample XSD File: Customers and Orders](#)

[Sample XML File: Customers and Orders in a Namespace](#)

[Sample XML File: Numerical Data \(LINQ to XML\)](#)

[Sample XML File: Numerical Data in a Namespace](#)

[Sample XML File: Books \(LINQ to XML\)](#)

[Sample XML File: Consolidated Purchase Orders](#)

[Reference \(LINQ to XML\)](#)

[LINQ to ADO.NET \(Portal Page\)](#)

[Enabling a Data Source for LINQ Querying](#)

[Visual Studio IDE and Tools Support for LINQ](#)

# Language Integrated Query (LINQ)

6/4/2019 • 3 minutes to read • [Edit Online](#)

Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language. Traditionally, queries against data are expressed as simple strings without type checking at compile time or IntelliSense support. Furthermore, you have to learn a different query language for each type of data source: SQL databases, XML documents, various Web services, and so on. With LINQ, a query is a first-class language construct, just like classes, methods, events. You write queries against strongly typed collections of objects by using language keywords and familiar operators. The LINQ family of technologies provides a consistent query experience for objects (LINQ to Objects), relational databases (LINQ to SQL), and XML (LINQ to XML).

For a developer who writes queries, the most visible "language-integrated" part of LINQ is the query expression. Query expressions are written in a declarative *query syntax*. By using query syntax, you can perform filtering, ordering, and grouping operations on data sources with a minimum of code. You use the same basic query expression patterns to query and transform data in SQL databases, ADO .NET Datasets, XML documents and streams, and .NET collections.

You can write LINQ queries in C# for SQL Server databases, XML documents, ADO.NET Datasets, and any collection of objects that supports [IEnumerable](#) or the generic [IEnumerable<T>](#) interface. LINQ support is also provided by third parties for many Web services and other database implementations.

The following example shows the complete query operation. The complete operation includes creating a data source, defining the query expression, and executing the query in a `foreach` statement.

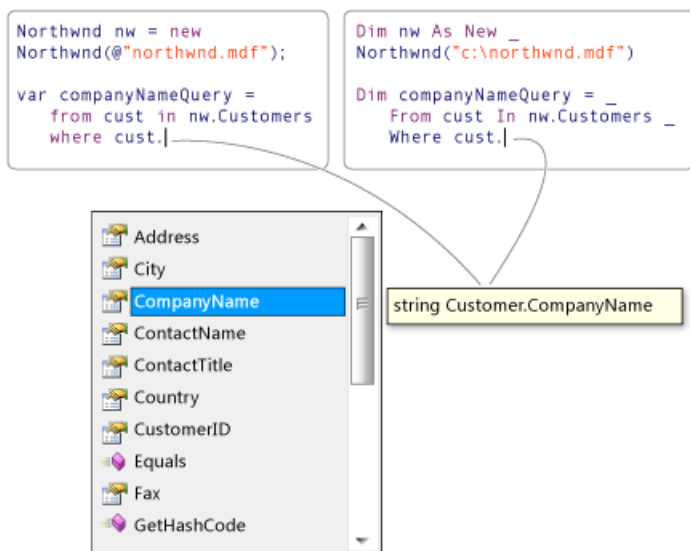
```
class LINQQueryExpressions
{
    static void Main()
    {
        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 97 92 81
```

The following illustration from Visual Studio shows a partially-completed LINQ query against a SQL Server database in both C# and Visual Basic with full type checking and IntelliSense support:





## Query expression overview

- Query expressions can be used to query and to transform data from any LINQ-enabled data source. For example, a single query can retrieve data from a SQL database, and produce an XML stream as output.
- Query expressions are easy to master because they use many familiar C# language constructs.
- The variables in a query expression are all strongly typed, although in many cases you do not have to provide the type explicitly because the compiler can infer it. For more information, see [Type relationships in LINQ query operations](#).
- A query is not executed until you iterate over the query variable, for example, in a `foreach` statement. For more information, see [Introduction to LINQ queries](#).
- At compile time, query expressions are converted to Standard Query Operator method calls according to the rules set forth in the C# specification. Any query that can be expressed by using query syntax can also be expressed by using method syntax. However, in most cases query syntax is more readable and concise. For more information, see [C# language specification](#) and [Standard query operators overview](#).
- As a rule when you write LINQ queries, we recommend that you use query syntax whenever possible and method syntax whenever necessary. There is no semantic or performance difference between the two different forms. Query expressions are often more readable than equivalent expressions written in method syntax.
- Some query operations, such as [Count](#) or [Max](#), have no equivalent query expression clause and must therefore be expressed as a method call. Method syntax can be combined with query syntax in various ways. For more information, see [Query syntax and method syntax in LINQ](#).
- Query expressions can be compiled to expression trees or to delegates, depending on the type that the query is applied to. [IEnumerable<T>](#) queries are compiled to delegates. [IQueryable](#) and [IQueryable<T>](#) queries are compiled to expression trees. For more information, see [Expression trees](#).

## Next steps

To learn more details about LINQ, start by becoming familiar with some basic concepts in [Query expression basics](#), and then read the documentation for the LINQ technology in which you are interested:

- XML documents: [LINQ to XML](#)
- ADO.NET Entity Framework: [LINQ to entities](#)
- .NET collections, files, strings and so on: [LINQ to objects](#)

To gain a deeper understanding of LINQ in general, see [LINQ in C#](#).

To start working with LINQ in C#, see the tutorial [Working with LINQ](#).

## See also

- [Language-Integrated Query \(LINQ\) \(C#\)](#)

# Introduction to LINQ Queries (C#)

5/18/2019 • 5 minutes to read • [Edit Online](#)

A *query* is an expression that retrieves data from a data source. Queries are usually expressed in a specialized query language. Different languages have been developed over time for the various types of data sources, for example SQL for relational databases and XQuery for XML. Therefore, developers have had to learn a new query language for each type of data source or data format that they must support. LINQ simplifies this situation by offering a consistent model for working with data across various kinds of data sources and formats. In a LINQ query, you are always working with objects. You use the same basic coding patterns to query and transform data in XML documents, SQL databases, ADO.NET Datasets, .NET collections, and any other format for which a LINQ provider is available.

## Three Parts of a Query Operation

All LINQ query operations consist of three distinct actions:

1. Obtain the data source.
2. Create the query.
3. Execute the query.

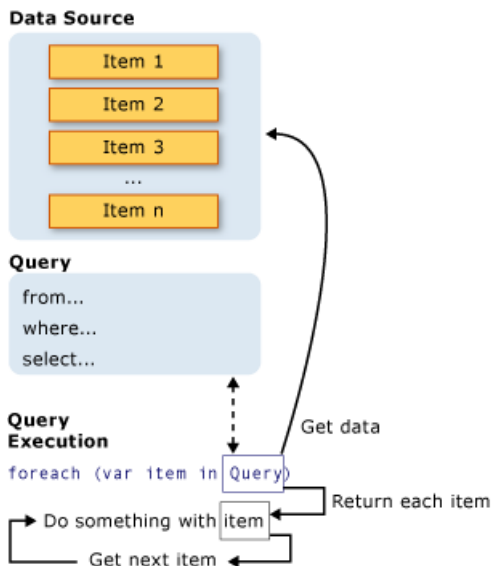
The following example shows how the three parts of a query operation are expressed in source code. The example uses an integer array as a data source for convenience; however, the same concepts apply to other data sources also. This example is referred to throughout the rest of this topic.

```
class IntroToLINQ
{
    static void Main()
    {
        // The Three Parts of a LINQ Query:
        // 1. Data source.
        int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

        // 2. Query creation.
        // numQuery is an IEnumerable<int>
        var numQuery =
            from num in numbers
            where (num % 2) == 0
            select num;

        // 3. Query execution.
        foreach (int num in numQuery)
        {
            Console.WriteLine("{0,1} ", num);
        }
    }
}
```

The following illustration shows the complete query operation. In LINQ the execution of the query is distinct from the query itself; in other words you have not retrieved any data just by creating a query variable.



## The Data Source

In the previous example, because the data source is an array, it implicitly supports the generic `IEnumerable<T>` interface. This fact means it can be queried with LINQ. A query is executed in a `foreach` statement, and `foreach` requires `IEnumerable` or `IEnumerable<T>`. Types that support `IEnumerable<T>` or a derived interface such as the generic `IQueryable<T>` are called *queryable types*.

A queryable type requires no modification or special treatment to serve as a LINQ data source. If the source data is not already in memory as a queryable type, the LINQ provider must represent it as such. For example, LINQ to XML loads an XML document into a queryable `XElement` type:

```
// Create a data source from an XML document.
// using System.Xml.Linq;
XElement contacts = XElement.Load(@"c:\myContactList.xml");
```

With LINQ to SQL, you first create an object-relational mapping at design time either manually or by using the [LINQ to SQL Tools in Visual Studio](#) in Visual Studio. You write your queries against the objects, and at run-time LINQ to SQL handles the communication with the database. In the following example, `Customers` represents a specific table in the database, and the type of the query result, `IQueryable<T>`, derives from `IEnumerable<T>`.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

// Query for customers in London.
IQueryable<Customer> custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;
```

For more information about how to create specific types of data sources, see the documentation for the various LINQ providers. However, the basic rule is very simple: a LINQ data source is any object that supports the generic `IEnumerable<T>` interface, or an interface that inherits from it.

### NOTE

Types such as `ArrayList` that support the non-generic `IEnumerable` interface can also be used as a LINQ data source. For more information, see [How to: Query an ArrayList with LINQ \(C#\)](#).

# The Query

The query specifies what information to retrieve from the data source or sources. Optionally, a query also specifies how that information should be sorted, grouped, and shaped before it is returned. A query is stored in a query variable and initialized with a query expression. To make it easier to write queries, C# has introduced new query syntax.

The query in the previous example returns all the even numbers from the integer array. The query expression contains three clauses: `from`, `where` and `select`. (If you are familiar with SQL, you will have noticed that the ordering of the clauses is reversed from the order in SQL.) The `from` clause specifies the data source, the `where` clause applies the filter, and the `select` clause specifies the type of the returned elements. These and the other query clauses are discussed in detail in the [LINQ Query Expressions](#) section. For now, the important point is that in LINQ, the query variable itself takes no action and returns no data. It just stores the information that is required to produce the results when the query is executed at some later point. For more information about how queries are constructed behind the scenes, see [Standard Query Operators Overview \(C#\)](#).

## NOTE

Queries can also be expressed by using method syntax. For more information, see [Query Syntax and Method Syntax in LINQ](#).

## Query Execution

### Deferred Execution

As stated previously, the query variable itself only stores the query commands. The actual execution of the query is deferred until you iterate over the query variable in a `foreach` statement. This concept is referred to as *deferred execution* and is demonstrated in the following example:

```
// Query execution.
foreach (int num in numQuery)
{
    Console.WriteLine("{0,1} ", num);
}
```

The `foreach` statement is also where the query results are retrieved. For example, in the previous query, the iteration variable `num` holds each value (one at a time) in the returned sequence.

Because the query variable itself never holds the query results, you can execute it as often as you like. For example, you may have a database that is being updated continually by a separate application. In your application, you could create one query that retrieves the latest data, and you could execute it repeatedly at some interval to retrieve different results every time.

### Forcing Immediate Execution

Queries that perform aggregation functions over a range of source elements must first iterate over those elements. Examples of such queries are `Count`, `Max`, `Average`, and `First`. These execute without an explicit `foreach` statement because the query itself must use `foreach` in order to return a result. Note also that these types of queries return a single value, not an `IEnumerable` collection. The following query returns a count of the even numbers in the source array:

```
var evenNumQuery =  
    from num in numbers  
    where (num % 2) == 0  
    select num;  
  
int evenNumCount = evenNumQuery.Count();
```

To force immediate execution of any query and cache its results, you can call the [ToList](#) or [ToArray](#) methods.

```
List<int> numQuery2 =  
    (from num in numbers  
     where (num % 2) == 0  
     select num).ToList();  
  
// or like this:  
// numQuery3 is still an int[]  
  
var numQuery3 =  
    (from num in numbers  
     where (num % 2) == 0  
     select num).ToArray();
```

You can also force execution by putting the `foreach` loop immediately after the query expression. However, by calling `ToList` or `ToArray` you also cache all the data in a single collection object.

## See also

- [Getting Started with LINQ in C#](#)
- [Walkthrough: Writing Queries in C#](#)
- [LINQ Query Expressions](#)
- [foreach, in](#)
- [Query Keywords \(LINQ\)](#)

# LINQ and Generic Types (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

LINQ queries are based on generic types, which were introduced in version 2.0 of the .NET Framework. You do not need an in-depth knowledge of generics before you can start writing queries. However, you may want to understand two basic concepts:

1. When you create an instance of a generic collection class such as `List<T>`, you replace the "T" with the type of objects that the list will hold. For example, a list of strings is expressed as `List<string>`, and a list of `Customer` objects is expressed as `List<Customer>`. A generic list is strongly typed and provides many benefits over collections that store their elements as `Object`. If you try to add a `Customer` to a `List<string>`, you will get an error at compile time. It is easy to use generic collections because you do not have to perform run-time type-casting.
2. `IEnumerable<T>` is the interface that enables generic collection classes to be enumerated by using the `foreach` statement. Generic collection classes support `IEnumerable<T>` just as non-generic collection classes such as `ArrayList` support `IEnumerable`.

For more information about generics, see [Generics](#).

## `IEnumerable<T>` variables in LINQ Queries

LINQ query variables are typed as `IEnumerable<T>` or a derived type such as `IQueryable<T>`. When you see a query variable that is typed as `IEnumerable<Customer>`, it just means that the query, when it is executed, will produce a sequence of zero or more `Customer` objects.

```
IEnumerable<Customer> customerQuery =  
    from cust in customers  
    where cust.City == "London"  
    select cust;  
  
foreach (Customer customer in customerQuery)  
{  
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);  
}
```

For more information, see [Type Relationships in LINQ Query Operations](#).

## Letting the Compiler Handle Generic Type Declarations

If you prefer, you can avoid generic syntax by using the `var` keyword. The `var` keyword instructs the compiler to infer the type of a query variable by looking at the data source specified in the `from` clause. The following example produces the same compiled code as the previous example:

```
var customerQuery2 =  
    from cust in customers  
    where cust.City == "London"  
    select cust;  
  
foreach(var customer in customerQuery2)  
{  
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);  
}
```

The `var` keyword is useful when the type of the variable is obvious or when it is not that important to explicitly specify nested generic types such as those that are produced by group queries. In general, we recommend that if you use `var`, realize that it can make your code more difficult for others to read. For more information, see [Implicitly Typed Local Variables](#).

## See also

- [Generics](#)



# Basic LINQ Query Operations (C#)

6/4/2019 • 5 minutes to read • [Edit Online](#)

This topic gives a brief introduction to LINQ query expressions and some of the typical kinds of operations that you perform in a query. More detailed information is in the following topics:

[LINQ Query Expressions](#)

[Standard Query Operators Overview \(C#\)](#)

[Walkthrough: Writing Queries in C#](#)

## NOTE

If you already are familiar with a query language such as SQL or XQuery, you can skip most of this topic. Read about the "from" clause in the next section to learn about the order of clauses in LINQ query expressions.

## Obtaining a Data Source

In a LINQ query, the first step is to specify the data source. In C# as in most programming languages a variable must be declared before it can be used. In a LINQ query, the `from` clause comes first in order to introduce the data source (`customers`) and the *range variable* (`cust`).

```
//queryAllCustomers is an IEnumerable<Customer>
var queryAllCustomers = from cust in customers
                        select cust;
```

The range variable is like the iteration variable in a `foreach` loop except that no actual iteration occurs in a query expression. When the query is executed, the range variable will serve as a reference to each successive element in `customers`. Because the compiler can infer the type of `cust`, you do not have to specify it explicitly. Additional range variables can be introduced by a `let` clause. For more information, see [let clause](#).

## NOTE

For non-generic data sources such as [ArrayList](#), the range variable must be explicitly typed. For more information, see [How to: Query an ArrayList with LINQ \(C#\)](#) and [from clause](#).

## Filtering

Probably the most common query operation is to apply a filter in the form of a Boolean expression. The filter causes the query to return only those elements for which the expression is true. The result is produced by using the `where` clause. The filter in effect specifies which elements to exclude from the source sequence. In the following example, only those `customers` who have an address in London are returned.

```
var queryLondonCustomers = from cust in customers
                           where cust.City == "London"
                           select cust;
```

You can use the familiar C# logical `AND` and `OR` operators to apply as many filter expressions as necessary in the

`where` clause. For example, to return only customers from "London" `AND` whose name is "Devon" you would write the following code:

```
where cust.City=="London" && cust.Name == "Devon"
```

To return customers from London or Paris, you would write the following code:

```
where cust.City == "London" || cust.City == "Paris"
```

For more information, see [where clause](#).

## Ordering

Often it is convenient to sort the returned data. The `orderby` clause will cause the elements in the returned sequence to be sorted according to the default comparer for the type being sorted. For example, the following query can be extended to sort the results based on the `Name` property. Because `Name` is a string, the default comparer performs an alphabetical sort from A to Z.

```
var queryLondonCustomers3 =  
    from cust in customers  
    where cust.City == "London"  
    orderby cust.Name ascending  
    select cust;
```

To order the results in reverse order, from Z to A, use the `orderby...descending` clause.

For more information, see [orderby clause](#).

## Grouping

The `group` clause enables you to group your results based on a key that you specify. For example you could specify that the results should be grouped by the `city` so that all customers from London or Paris are in individual groups. In this case, `cust.City` is the key.

```
// queryCustomersByCity is an IEnumerable<IGrouping<string, Customer>>  
var queryCustomersByCity =  
    from cust in customers  
    group cust by cust.City;  
  
// customerGroup is an IGrouping<string, Customer>  
foreach (var customerGroup in queryCustomersByCity)  
{  
    Console.WriteLine(customerGroup.Key);  
    foreach (Customer customer in customerGroup)  
    {  
        Console.WriteLine("    {0}", customer.Name);  
    }  
}
```

When you end a query with a `group` clause, your results take the form of a list of lists. Each element in the list is an object that has a `key` member and a list of elements that are grouped under that key. When you iterate over a query that produces a sequence of groups, you must use a nested `foreach` loop. The outer loop iterates over each group, and the inner loop iterates over each group's members.

If you must refer to the results of a group operation, you can use the `into` keyword to create an identifier that can

be queried further. The following query returns only those groups that contain more than two customers:

```
// custQuery is an IEnumerable<IGrouping<string, Customer>>
var custQuery =
    from cust in customers
    group cust by cust.City into custGroup
    where custGroup.Count() > 2
    orderby custGroup.Key
    select custGroup;
```

For more information, see [group clause](#).

## Joining

Join operations create associations between sequences that are not explicitly modeled in the data sources. For example you can perform a join to find all the customers and distributors who have the same location. In LINQ the `join` clause always works against object collections instead of database tables directly.

```
var innerJoinQuery =
    from cust in customers
    join dist in distributors on cust.City equals dist.City
    select new { CustomerName = cust.Name, DistributorName = dist.Name };
```

In LINQ you do not have to use `join` as often as you do in SQL because foreign keys in LINQ are represented in the object model as properties that hold a collection of items. For example, a `Customer` object contains a collection of `Order` objects. Rather than performing a join, you access the orders by using dot notation:

```
from order in Customer.Orders...
```

For more information, see [join clause](#).

## Selecting (Projections)

The `select` clause produces the results of the query and specifies the "shape" or type of each returned element. For example, you can specify whether your results will consist of complete `Customer` objects, just one member, a subset of members, or some completely different result type based on a computation or new object creation. When the `select` clause produces something other than a copy of the source element, the operation is called a *projection*. The use of projections to transform data is a powerful capability of LINQ query expressions. For more information, see [Data Transformations with LINQ \(C#\)](#) and [select clause](#).

## See also

- [LINQ Query Expressions](#)
- [Walkthrough: Writing Queries in C#](#)
- [Query Keywords \(LINQ\)](#)
- [Anonymous Types](#)

# Data Transformations with LINQ (C#)

6/4/2019 • 5 minutes to read • [Edit Online](#)

Language-Integrated Query (LINQ) is not only about retrieving data. It is also a powerful tool for transforming data. By using a LINQ query, you can use a source sequence as input and modify it in many ways to create a new output sequence. You can modify the sequence itself without modifying the elements themselves by sorting and grouping. But perhaps the most powerful feature of LINQ queries is the ability to create new types. This is accomplished in the [select](#) clause. For example, you can perform the following tasks:

- Merge multiple input sequences into a single output sequence that has a new type.
- Create output sequences whose elements consist of only one or several properties of each element in the source sequence.
- Create output sequences whose elements consist of the results of operations performed on the source data.
- Create output sequences in a different format. For example, you can transform data from SQL rows or text files into XML.

These are just several examples. Of course, these transformations can be combined in various ways in the same query. Furthermore, the output sequence of one query can be used as the input sequence for a new query.

## Joining Multiple Inputs into One Output Sequence

You can use a LINQ query to create an output sequence that contains elements from more than one input sequence. The following example shows how to combine two in-memory data structures, but the same principles can be applied to combine data from XML or SQL or DataSet sources. Assume the following two class types:

```
class Student
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public List<int> Scores;
}

class Teacher
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public string City { get; set; }
}
```

The following example shows the query:

```

class DataTransformations
{
    static void Main()
    {
        // Create the first data source.
        List<Student> students = new List<Student>()
        {
            new Student { First="Svetlana",
                          Last="Omelchenko",
                          ID=111,
                          Street="123 Main Street",
                          City="Seattle",
                          Scores= new List<int> { 97, 92, 81, 60 } },
            new Student { First="Claire",
                          Last="O'Donnell",
                          ID=112,
                          Street="124 Main Street",
                          City="Redmond",
                          Scores= new List<int> { 75, 84, 91, 39 } },
            new Student { First="Sven",
                          Last="Mortensen",
                          ID=113,
                          Street="125 Main Street",
                          City="Lake City",
                          Scores= new List<int> { 88, 94, 65, 91 } },
        };

        // Create the second data source.
        List<Teacher> teachers = new List<Teacher>()
        {
            new Teacher { First="Ann", Last="Beebe", ID=945, City="Seattle" },
            new Teacher { First="Alex", Last="Robinson", ID=956, City="Redmond" },
            new Teacher { First="Michiyo", Last="Sato", ID=972, City="Tacoma" }
        };

        // Create the query.
        var peopleInSeattle = (from student in students
                               where student.City == "Seattle"
                               select student.Last)
                               .Concat(from teacher in teachers
                                       where teacher.City == "Seattle"
                                       select teacher.Last);

        Console.WriteLine("The following students and teachers live in Seattle:");
        // Execute the query.
        foreach (var person in peopleInSeattle)
        {
            Console.WriteLine(person);
        }

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   The following students and teachers live in Seattle:
   Omelchenko
   Beebe
*/

```

For more information, see [join clause](#) and [select clause](#).

## Selecting a Subset of each Source Element

There are two primary ways to select a subset of each element in the source sequence:

1. To select just one member of the source element, use the dot operation. In the following example, assume that a `Customer` object contains several public properties including a string named `City`. When executed, this query will produce an output sequence of strings.

```
var query = from cust in Customers
            select cust.City;
```

2. To create elements that contain more than one property from the source element, you can use an object initializer with either a named object or an anonymous type. The following example shows the use of an anonymous type to encapsulate two properties from each `Customer` element:

```
var query = from cust in Customer
            select new {Name = cust.Name, City = cust.City};
```

For more information, see [Object and Collection Initializers](#) and [Anonymous Types](#).

## Transforming in-Memory Objects into XML

LINQ queries make it easy to transform data between in-memory data structures, SQL databases, ADO.NET Datasets and XML streams or documents. The following example transforms objects in an in-memory data structure into XML elements.

```
class XMLTransform
{
    static void Main()
    {
        // Create the data source by using a collection initializer.
        // The Student class was defined previously in this topic.
        List<Student> students = new List<Student>()
        {
            new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores = new List<int>{97, 92, 81, 60}},
            new Student {First="Claire", Last="O'Donnell", ID=112, Scores = new List<int>{75, 84, 91, 39}},
            new Student {First="Sven", Last="Mortensen", ID=113, Scores = new List<int>{88, 94, 65, 91}},
        };

        // Create the query.
        var studentsToXML = new XElement("Root",
            from student in students
            let scores = string.Join(",", student.Scores)
            select new XElement("student",
                new XElement("First", student.First),
                new XElement("Last", student.Last),
                new XElement("Scores", scores)
            ) // end "student"
        ); // end "Root"

        // Execute the query.
        Console.WriteLine(studentsToXML);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

The code produces the following XML output:

```

<Root>
  <student>
    <First>Svetlana</First>
    <Last>Omelchenko</Last>
    <Scores>97,92,81,60</Scores>
  </student>
  <student>
    <First>Claire</First>
    <Last>O'Donnell</Last>
    <Scores>75,84,91,39</Scores>
  </student>
  <student>
    <First>Sven</First>
    <Last>Mortensen</Last>
    <Scores>88,94,65,91</Scores>
  </student>
</Root>

```

For more information, see [Creating XML Trees in C# \(LINQ to XML\)](#).

## Performing Operations on Source Elements

An output sequence might not contain any elements or element properties from the source sequence. The output might instead be a sequence of values that is computed by using the source elements as input arguments. The following simple query, when it is executed, outputs a sequence of strings whose values represent a calculation based on the source sequence of elements of type `double`.

### NOTE

Calling methods in query expressions is not supported if the query will be translated into some other domain. For example, you cannot call an ordinary C# method in LINQ to SQL because SQL Server has no context for it. However, you can map stored procedures to methods and call those. For more information, see [Stored Procedures](#).

```

class FormatQuery
{
    static void Main()
    {
        // Data source.
        double[] radii = { 1, 2, 3 };

        // Query.
        IEnumerable<string> query =
            from rad in radii
            select $"Area = {rad * rad * Math.PI:F2}";

        // Query execution.
        foreach (string s in query)
            Console.WriteLine(s);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
Area = 3.14
Area = 12.57
Area = 28.27
*/

```

## See also

- [Language-Integrated Query \(LINQ\) \(C#\)](#)
- [LINQ to SQL](#)
- [LINQ to DataSet](#)
- [LINQ to XML \(C#\)](#)
- [LINQ Query Expressions](#)
- [select clause](#)



# Type Relationships in LINQ Query Operations (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

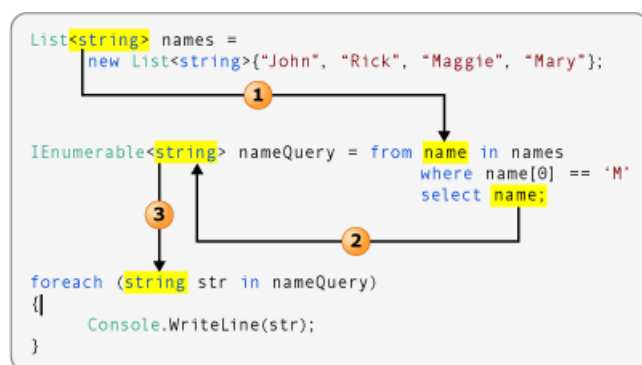
To write queries effectively, you should understand how types of the variables in a complete query operation all relate to each other. If you understand these relationships you will more easily comprehend the LINQ samples and code examples in the documentation. Furthermore, you will understand what occurs behind the scenes when variables are implicitly typed by using `var`.

LINQ query operations are strongly typed in the data source, in the query itself, and in the query execution. The type of the variables in the query must be compatible with the type of the elements in the data source and with the type of the iteration variable in the `foreach` statement. This strong typing guarantees that type errors are caught at compile time when they can be corrected before users encounter them.

In order to demonstrate these type relationships, most of the examples that follow use explicit typing for all variables. The last example shows how the same principles apply even when you use implicit typing by using `var`.

## Queries that do not Transform the Source Data

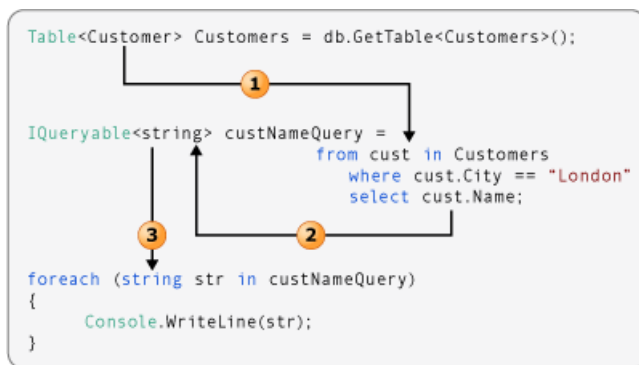
The following illustration shows a LINQ to Objects query operation that performs no transformations on the data. The source contains a sequence of strings and the query output is also a sequence of strings.



1. The type argument of the data source determines the type of the range variable.
2. The type of the object that is selected determines the type of the query variable. Here `name` is a string. Therefore, the query variable is an `IEnumerable<string>`.
3. The query variable is iterated over in the `foreach` statement. Because the query variable is a sequence of strings, the iteration variable is also a string.

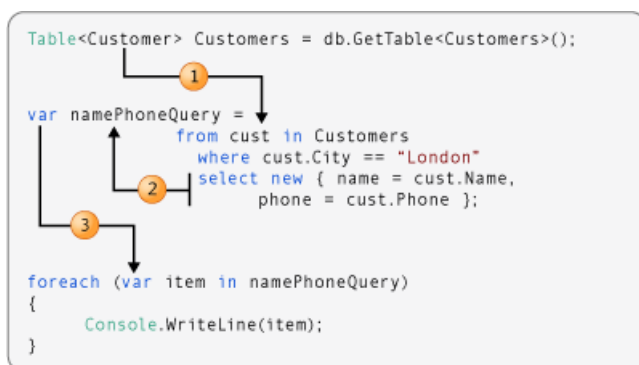
## Queries that Transform the Source Data

The following illustration shows a LINQ to SQL query operation that performs a simple transformation on the data. The query takes a sequence of `Customer` objects as input, and selects only the `Name` property in the result. Because `Name` is a string, the query produces a sequence of strings as output.



1. The type argument of the data source determines the type of the range variable.
2. The `select` statement returns the `Name` property instead of the complete `Customer` object. Because `Name` is a string, the type argument of `custNameQuery` is `string`, not `Customer`.
3. Because `custNameQuery` is a sequence of strings, the `foreach` loop's iteration variable must also be a `string`.

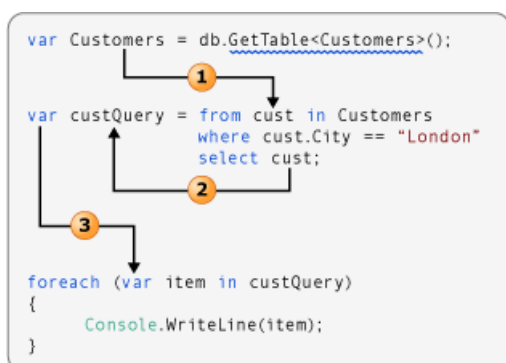
The following illustration shows a slightly more complex transformation. The `select` statement returns an anonymous type that captures just two members of the original `Customer` object.



1. The type argument of the data source is always the type of the range variable in the query.
2. Because the `select` statement produces an anonymous type, the query variable must be implicitly typed by using `var`.
3. Because the type of the query variable is implicit, the iteration variable in the `foreach` loop must also be implicit.

## Letting the compiler infer type information

Although you should understand the type relationships in a query operation, you have the option to let the compiler do all the work for you. The keyword `var` can be used for any local variable in a query operation. The following illustration is similar to example number 2 that was discussed earlier. However, the compiler supplies the strong type for each variable in the query operation.



For more information about `var`, see [Implicitly Typed Local Variables](#).

# Query Syntax and Method Syntax in LINQ (C#)

6/4/2019 • 4 minutes to read • [Edit Online](#)

Most queries in the introductory Language Integrated Query (LINQ) documentation are written by using the LINQ declarative query syntax. However, the query syntax must be translated into method calls for the .NET common language runtime (CLR) when the code is compiled. These method calls invoke the standard query operators, which have names such as `Where`, `Select`, `GroupBy`, `Join`, `Max`, and `Average`. You can call them directly by using method syntax instead of query syntax.

Query syntax and method syntax are semantically identical, but many people find query syntax simpler and easier to read. Some queries must be expressed as method calls. For example, you must use a method call to express a query that retrieves the number of elements that match a specified condition. You also must use a method call for a query that retrieves the element that has the maximum value in a source sequence. The reference documentation for the standard query operators in the [System.Linq](#) namespace generally uses method syntax. Therefore, even when getting started writing LINQ queries, it is useful to be familiar with how to use method syntax in queries and in query expressions themselves.

## Standard Query Operator Extension Methods

The following example shows a simple *query expression* and the semantically equivalent query written as a *method-based query*.

```

class QueryVMMethodSyntax
{
    static void Main()
    {
        int[] numbers = { 5, 10, 8, 3, 6, 12};

        //Query syntax:
        IEnumerable<int> numQuery1 =
            from num in numbers
            where num % 2 == 0
            orderby num
            select num;

        //Method syntax:
        IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 == 0).OrderBy(n => n);

        foreach (int i in numQuery1)
        {
            Console.Write(i + " ");
        }
        Console.WriteLine(System.Environment.NewLine);
        foreach (int i in numQuery2)
        {
            Console.Write(i + " ");
        }

        // Keep the console open in debug mode.
        Console.WriteLine(System.Environment.NewLine);
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/*
Output:
6 8 10 12
6 8 10 12
*/

```

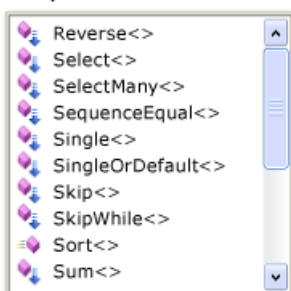
The output from the two examples is identical. You can see that the type of the query variable is the same in both forms: `IEnumerable<T>`.

To understand the method-based query, let's examine it more closely. On the right side of the expression, notice that the `where` clause is now expressed as an instance method on the `numbers` object, which as you will recall has a type of `IEnumerable<int>`. If you are familiar with the generic `IEnumerable<T>` interface, you know that it does not have a `Where` method. However, if you invoke the IntelliSense completion list in the Visual Studio IDE, you will see not only a `Where` method, but many other methods such as `Select`, `SelectMany`, `Join`, and `OrderBy`. These are all the standard query operators.

```

List<string> list = new List<string>();
list.

```



Although it looks as if `IEnumerable<T>` has been redefined to include these additional methods, in fact this is not the case. The standard query operators are implemented as a new kind of method called *extension methods*. Extensions methods "extend" an existing type; they can be called as if they were instance methods on the type. The

standard query operators extend `IEnumerable<T>` and that is why you can write `numbers.Where(...)`.

To get started using LINQ, all that you really have to know about extension methods is how to bring them into scope in your application by using the correct `using` directives. From your application's point of view, an extension method and a regular instance method are the same.

For more information about extension methods, see [Extension Methods](#). For more information about standard query operators, see [Standard Query Operators Overview \(C#\)](#). Some LINQ providers, such as LINQ to SQL and LINQ to XML, implement their own standard query operators and additional extension methods for other types besides `IEnumerable<T>`.

## Lambda Expressions

In the previous example, notice that the conditional expression (`num % 2 == 0`) is passed as an in-line argument to the `Where` method: `Where(num => num % 2 == 0)`. This inline expression is called a lambda expression. It is a convenient way to write code that would otherwise have to be written in more cumbersome form as an anonymous method or a generic delegate or an expression tree. In C# `=>` is the lambda operator, which is read as "goes to". The `num` on the left of the operator is the input variable which corresponds to `num` in the query expression. The compiler can infer the type of `num` because it knows that `numbers` is a generic `IEnumerable<T>` type. The body of the lambda is just the same as the expression in query syntax or in any other C# expression or statement; it can include method calls and other complex logic. The "return value" is just the expression result.

To get started using LINQ, you do not have to use lambdas extensively. However, certain queries can only be expressed in method syntax and some of those require lambda expressions. After you become more familiar with lambdas, you will find that they are a powerful and flexible tool in your LINQ toolbox. For more information, see [Lambda Expressions](#).

## Composability of Queries

In the previous code example, note that the `OrderBy` method is invoked by using the dot operator on the call to `Where`. `Where` produces a filtered sequence, and then `OrderBy` operates on that sequence by sorting it. Because queries return an `IEnumerable`, you compose them in method syntax by chaining the method calls together. This is what the compiler does behind the scenes when you write queries by using query syntax. And because a query variable does not store the results of the query, you can modify it or use it as the basis for a new query at any time, even after it has been executed.

# C# Features That Support LINQ

6/6/2019 • 3 minutes to read • [Edit Online](#)

The following section introduces new language constructs introduced in C# 3.0. Although these new features are all used to a degree with LINQ queries, they are not limited to LINQ and can be used in any context where you find them useful.

## Query Expressions

Query expressions use a declarative syntax similar to SQL or XQuery to query over `IEnumerable` collections. At compile time query syntax is converted to method calls to a LINQ provider's implementation of the standard query operator extension methods. Applications control the standard query operators that are in scope by specifying the appropriate namespace with a `using` directive. The following query expression takes an array of strings, groups them according to the first character in the string, and orders the groups.

```
var query = from str in stringArray
            group str by str[0] into stringGroup
            orderby stringGroup.Key
            select stringGroup;
```

For more information, see [LINQ Query Expressions](#).

## Implicitly Typed Variables (var)

Instead of explicitly specifying a type when you declare and initialize a variable, you can use the `var` modifier to instruct the compiler to infer and assign the type, as shown here:

```
var number = 5;
var name = "Virginia";
var query = from str in stringArray
            where str[0] == 'm'
            select str;
```

Variables declared as `var` are just as strongly-typed as variables whose type you specify explicitly. The use of `var` makes it possible to create anonymous types, but it can be used only for local variables. Arrays can also be declared with implicit typing.

For more information, see [Implicitly Typed Local Variables](#).

## Object and Collection Initializers

Object and collection initializers make it possible to initialize objects without explicitly calling a constructor for the object. Initializers are typically used in query expressions when they project the source data into a new data type. Assuming a class named `Customer` with public `Name` and `Phone` properties, the object initializer can be used as in the following code:

```
var cust = new Customer { Name = "Mike", Phone = "555-1212" };
```

Continuing with our `Customer` class, assume that there is a data source called `IncomingOrders`, and that for each order with a large `OrderSize`, we would like to create a new `Customer` based off of that order. A LINQ query can

be executed on this data source and use object initialization to fill a collection:

```
var newLargeOrderCustomers = from o in IncomingOrders
    where o.OrderSize > 5
    select new Customer { Name = o.Name, Phone = o.Phone };
```

The data source may have more properties lying under the hood than the `Customer` class such as `OrderSize`, but with object initialization, the data returned from the query is molded into the desired data type; we choose the data that is relevant to our class. As a result, we now have an `IEnumerable` filled with the new `Customer`s we wanted. The above can also be written in LINQ's method syntax:

```
var newLargeOrderCustomers = IncomingOrders.Where(x => x.OrderSize > 5).Select(y => new Customer { Name = y.Name, Phone = y.Phone });
```

For more information, see:

- [Object and Collection Initializers](#)
- [Query Expression Syntax for Standard Query Operators](#)

## Anonymous Types

An anonymous type is constructed by the compiler and the type name is only available to the compiler. Anonymous types provide a convenient way to group a set of properties temporarily in a query result without having to define a separate named type. Anonymous types are initialized with a new expression and an object initializer, as shown here:

```
select new {name = cust.Name, phone = cust.Phone};
```

For more information, see [Anonymous Types](#).

## Extension Methods

An extension method is a static method that can be associated with a type, so that it can be called as if it were an instance method on the type. This feature enables you to, in effect, "add" new methods to existing types without actually modifying them. The standard query operators are a set of extension methods that provide LINQ query functionality for any type that implements `IEnumerable<T>`.

For more information, see [Extension Methods](#).

## Lambda Expressions

A lambda expression is an inline function that uses the `=>` operator to separate input parameters from the function body and can be converted at compile time to a delegate or an expression tree. In LINQ programming, you will encounter lambda expressions when you make direct method calls to the standard query operators.

For more information, see:

- [Anonymous Functions](#)
- [Lambda Expressions](#)
- [Expression Trees \(C#\)](#)

## See also



- [Language-Integrated Query \(LINQ\) \(C#\)](#)

# Walkthrough: Writing Queries in C# (LINQ)

6/4/2019 • 10 minutes to read • [Edit Online](#)

This walkthrough demonstrates the C# language features that are used to write LINQ query expressions.

## Create a C# Project

### NOTE

The following instructions are for Visual Studio. If you are using a different development environment, create a console project with a reference to System.Core.dll and a `using` directive for the [System.Linq](#) namespace.

#### To create a project in Visual Studio

1. Start Visual Studio.
2. On the menu bar, choose **File, New, Project**.  
  
The **New Project** dialog box opens.
3. Expand **Installed**, expand **Templates**, expand **Visual C#**, and then choose **Console Application**.
4. In the **Name** text box, enter a different name or accept the default name, and then choose the **OK** button.  
  
The new project appears in **Solution Explorer**.
5. Notice that your project has a reference to System.Core.dll and a `using` directive for the [System.Linq](#) namespace.

## Create an in-Memory Data Source

The data source for the queries is a simple list of `Student` objects. Each `Student` record has a first name, last name, and an array of integers that represents their test scores in the class. Copy this code into your project. Note the following characteristics:

- The `Student` class consists of auto-implemented properties.
- Each student in the list is initialized with an object initializer.
- The list itself is initialized with a collection initializer.

This whole data structure will be initialized and instantiated without explicit calls to any constructor or explicit member access. For more information about these new features, see [Auto-Implemented Properties](#) and [Object and Collection Initializers](#).

#### To add the data source

- Add the `Student` class and the initialized list of students to the `Program` class in your project.

```

public class Student
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public List<int> Scores;
}

// Create a data source by using a collection initializer.
static List<Student> students = new List<Student>
{
    new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 92, 81, 60}},
    new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},
    new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {88, 94, 65, 91}},
    new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {97, 89, 85, 82}},
    new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {35, 72, 91, 70}},
    new Student {First="Fadi", Last="Fakhouri", ID=116, Scores= new List<int> {99, 86, 90, 94}},
    new Student {First="Hanying", Last="Feng", ID=117, Scores= new List<int> {93, 92, 80, 87}},
    new Student {First="Hugo", Last="Garcia", ID=118, Scores= new List<int> {92, 90, 83, 78}},
    new Student {First="Lance", Last="Tucker", ID=119, Scores= new List<int> {68, 79, 88, 92}},
    new Student {First="Terry", Last="Adams", ID=120, Scores= new List<int> {99, 82, 81, 79}},
    new Student {First="Eugene", Last="Zabokritski", ID=121, Scores= new List<int> {96, 85, 91, 60}},
    new Student {First="Michael", Last="Tucker", ID=122, Scores= new List<int> {94, 92, 91, 91}}
};

```

#### To add a new Student to the Students list

1. Add a new `Student` to the `Students` list and use a name and test scores of your choice. Try typing all the new student information in order to better learn the syntax for the object initializer.

## Create the Query

#### To create a simple query

- In the application's `Main` method, create a simple query that, when it is executed, will produce a list of all students whose score on the first test was greater than 90. Note that because the whole `Student` object is selected, the type of the query is `IEnumerable<Student>`. Although the code could also use implicit typing by using the `var` keyword, explicit typing is used to clearly illustrate results. (For more information about `var`, see [Implicitly Typed Local Variables](#).)

Note also that the query's range variable, `student`, serves as a reference to each `Student` in the source, providing member access for each object.

```

// Create the query.
// The first line could also be written as "var studentQuery ="
IEnumerable<Student> studentQuery =
    from student in students
    where student.Scores[0] > 90
    select student;

```

## Execute the Query

#### To execute the query

1. Now write the `foreach` loop that will cause the query to execute. Note the following about the code:
  - Each element in the returned sequence is accessed through the iteration variable in the `foreach` loop.
  - The type of this variable is `Student`, and the type of the query variable is compatible, `IEnumerable<Student>`.

2. After you have added this code, build and run the application to see the results in the **Console** window.

```
// Execute the query.  
// var could be used here also.  
foreach (Student student in studentQuery)  
{  
    Console.WriteLine("{0}, {1}", student.Last, student.First);  
}  
  
// Output:  
// Omelchenko, Svetlana  
// Garcia, Cesar  
// Fakhouri, Fadi  
// Feng, Hanying  
// Garcia, Hugo  
// Adams, Terry  
// Zabokritski, Eugene  
// Tucker, Michael
```

#### To add another filter condition

1. You can combine multiple Boolean conditions in the `where` clause in order to further refine a query. The following code adds a condition so that the query returns those students whose first score was over 90 and whose last score was less than 80. The `where` clause should resemble the following code.

```
where student.Scores[0] > 90 && student.Scores[3] < 80
```

For more information, see [where clause](#).

## Modify the Query

#### To order the results

1. It will be easier to scan the results if they are in some kind of order. You can order the returned sequence by any accessible field in the source elements. For example, the following `orderby` clause orders the results in alphabetical order from A to Z according to the last name of each student. Add the following `orderby` clause to your query, right after the `where` statement and before the `select` statement:

```
orderby student.Last ascending
```

2. Now change the `orderby` clause so that it orders the results in reverse order according to the score on the first test, from the highest score to the lowest score.

```
orderby student.Scores[0] descending
```

3. Change the `WriteLine` format string so that you can see the scores:

```
Console.WriteLine("{0}, {1} {2}", student.Last, student.First, student.Scores[0]);
```

For more information, see [orderby clause](#).

#### To group the results

1. Grouping is a powerful capability in query expressions. A query with a group clause produces a sequence of groups, and each group itself contains a `Key` and a sequence that consists of all the members of that group. The following new query groups the students by using the first letter of their last name as the key.

```
// studentQuery2 is an IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0];
```

- Note that the type of the query has now changed. It now produces a sequence of groups that have a `char` type as a key, and a sequence of `Student` objects. Because the type of the query has changed, the following code changes the `foreach` execution loop also:

```
// studentGroup is a IGrouping<char, Student>
foreach (var studentGroup in studentQuery2)
{
    Console.WriteLine(studentGroup.Key);
    foreach (Student student in studentGroup)
    {
        Console.WriteLine("  {0}, {1}",
            student.Last, student.First);
    }
}

// Output:
// O
//  Omelchenko, Svetlana
//  O'Donnell, Claire
// M
//  Mortensen, Sven
// G
//  Garcia, Cesar
//  Garcia, Debra
//  Garcia, Hugo
// F
//  Fakhouri, Fadi
//  Feng, Hanying
// T
//  Tucker, Lance
//  Tucker, Michael
// A
//  Adams, Terry
// Z
//  Zabokritski, Eugene
```

- Run the application and view the results in the **Console** window.

For more information, see [group clause](#).

#### To make the variables implicitly typed

- Explicitly coding `IEnumerables` of `IGroupings` can quickly become tedious. You can write the same query and `foreach` loop much more conveniently by using `var`. The `var` keyword does not change the types of your objects; it just instructs the compiler to infer the types. Change the type of `studentQuery` and the iteration variable `group` to `var` and rerun the query. Note that in the inner `foreach` loop, the iteration variable is still typed as `Student`, and the query works just as before. Change the `s` iteration variable to `var` and run the query again. You see that you get exactly the same results.

```

var studentQuery3 =
    from student in students
    group student by student.Last[0];

foreach (var groupOfStudents in studentQuery3)
{
    Console.WriteLine(groupOfStudents.Key);
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine("    {0}, {1}",
            student.Last, student.First);
    }
}

// Output:
// O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
// M
//   Mortensen, Sven
// G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
// F
//   Fakhouri, Fadi
//   Feng, Hanying
// T
//   Tucker, Lance
//   Tucker, Michael
// A
//   Adams, Terry
// Z
//   Zabokritski, Eugene

```

For more information about [var](#), see [Implicitly Typed Local Variables](#).

#### To order the groups by their key value

1. When you run the previous query, you notice that the groups are not in alphabetical order. To change this, you must provide an `orderby` clause after the `group` clause. But to use an `orderby` clause, you first need an identifier that serves as a reference to the groups created by the `group` clause. You provide the identifier by using the `into` keyword, as follows:

```

var studentQuery4 =
    from student in students
    group student by student.Last[0] into studentGroup
    orderby studentGroup.Key
    select studentGroup;

foreach (var groupOfStudents in studentQuery4)
{
    Console.WriteLine(groupOfStudents.Key);
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine("    {0}, {1}",
            student.Last, student.First);
    }
}

// Output:
//A
//  Adams, Terry
//F
//  Fakhouri, Fadi
//  Feng, Hanying
//G
//  Garcia, Cesar
//  Garcia, Debra
//  Garcia, Hugo
//M
//  Mortensen, Sven
//O
//  Omelchenko, Svetlana
//  O'Donnell, Claire
//T
//  Tucker, Lance
//  Tucker, Michael
//Z
//  Zabokritski, Eugene

```

When you run this query, you will see the groups are now sorted in alphabetical order.

#### To introduce an identifier by using let

1. You can use the `let` keyword to introduce an identifier for any expression result in the query expression. This identifier can be a convenience, as in the following example, or it can enhance performance by storing the results of an expression so that it does not have to be calculated multiple times.

```
// studentQuery5 is an IEnumerable<string>
// This query returns those students whose
// first test score was higher than their
// average score.
var studentQuery5 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where totalScore / 4 < student.Scores[0]
    select student.Last + " " + student.First;

foreach (string s in studentQuery5)
{
    Console.WriteLine(s);
}

// Output:
// Omelchenko Svetlana
// O'Donnell Claire
// Mortensen Sven
// Garcia Cesar
// Fakhouri Fadi
// Feng Hanying
// Garcia Hugo
// Adams Terry
// Zabokritski Eugene
// Tucker Michael
```

For more information, see [let clause](#).

#### To use method syntax in a query expression

1. As described in [Query Syntax and Method Syntax in LINQ](#), some query operations can only be expressed by using method syntax. The following code calculates the total score for each `Student` in the source sequence, and then calls the `Average()` method on the results of that query to calculate the average score of the class.

```
var studentQuery6 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    select totalScore;

double averageScore = studentQuery6.Average();
Console.WriteLine("Class average score = {0}", averageScore);

// Output:
// Class average score = 334.166666666667
```

#### To transform or project in the select clause

1. It is very common for a query to produce a sequence whose elements differ from the elements in the source sequences. Delete or comment out your previous query and execution loop, and replace it with the following code. Note that the query returns a sequence of strings (not `Students`), and this fact is reflected in the `foreach` loop.



```

IEnumerable<string> studentQuery7 =
    from student in students
    where student.Last == "Garcia"
    select student.First;

Console.WriteLine("The Garcias in the class are:");
foreach (string s in studentQuery7)
{
    Console.WriteLine(s);
}

// Output:
// The Garcias in the class are:
// Cesar
// Debra
// Hugo

```

2. Code earlier in this walkthrough indicated that the average class score is approximately 334. To produce a sequence of `Students` whose total score is greater than the class average, together with their `Student ID`, you can use an anonymous type in the `select` statement:

```

var studentQuery8 =
    from student in students
    let x = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where x > averageScore
    select new { id = student.ID, score = x };

foreach (var item in studentQuery8)
{
    Console.WriteLine("Student ID: {0}, Score: {1}", item.id, item.score);
}

// Output:
// Student ID: 113, Score: 338
// Student ID: 114, Score: 353
// Student ID: 116, Score: 369
// Student ID: 117, Score: 352
// Student ID: 118, Score: 343
// Student ID: 120, Score: 341
// Student ID: 122, Score: 368

```

## Next Steps

After you are familiar with the basic aspects of working with queries in C#, you are ready to read the documentation and samples for the specific type of LINQ provider you are interested in:

[LINQ to SQL](#)

[LINQ to DataSet](#)

[LINQ to XML \(C#\)](#)

[LINQ to Objects \(C#\)](#)

## See also

- [Language-Integrated Query \(LINQ\) \(C#\)](#)
- [LINQ Query Expressions](#)

# Standard Query Operators Overview (C#)

1/23/2019 • 3 minutes to read • [Edit Online](#)

The *standard query operators* are the methods that form the LINQ pattern. Most of these methods operate on sequences, where a sequence is an object whose type implements the [IEnumerable<T>](#) interface or the [IQueryable<T>](#) interface. The standard query operators provide query capabilities including filtering, projection, aggregation, sorting and more.

There are two sets of LINQ standard query operators, one that operates on objects of type [IEnumerable<T>](#) and the other that operates on objects of type [IQueryable<T>](#). The methods that make up each set are static members of the [Enumerable](#) and [Queryable](#) classes, respectively. They are defined as *extension methods* of the type that they operate on. This means that they can be called by using either static method syntax or instance method syntax.

In addition, several standard query operator methods operate on types other than those based on [IEnumerable<T>](#) or [IQueryable<T>](#). The [Enumerable](#) type defines two such methods that both operate on objects of type [IEnumerable](#). These methods, [Cast<TResult>\(IEnumerable\)](#) and [OfType<TResult>\(IEnumerable\)](#), let you enable a non-parameterized, or non-generic, collection to be queried in the LINQ pattern. They do this by creating a strongly-typed collection of objects. The [Queryable](#) class defines two similar methods, [Cast<TResult>\(IQueryable\)](#) and [OfType<TResult>\(IQueryable\)](#), that operate on objects of type [Queryable](#).

The standard query operators differ in the timing of their execution, depending on whether they return a singleton value or a sequence of values. Those methods that return a singleton value (for example, [Average](#) and [Sum](#)) execute immediately. Methods that return a sequence defer the query execution and return an enumerable object.

In the case of the methods that operate on in-memory collections, that is, those methods that extend [IEnumerable<T>](#), the returned enumerable object captures the arguments that were passed to the method. When that object is enumerated, the logic of the query operator is employed and the query results are returned.

In contrast, methods that extend [IQueryable<T>](#) do not implement any querying behavior, but build an expression tree that represents the query to be performed. The query processing is handled by the source [IQueryable<T>](#) object.

Calls to query methods can be chained together in one query, which enables queries to become arbitrarily complex.

The following code example demonstrates how the standard query operators can be used to obtain information about a sequence.

```

string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words to create a collection.
string[] words = sentence.Split(' ');

// Using query expression syntax.
var query = from word in words
             group word.ToUpper() by word.Length into gr
             orderby gr.Key
             select new { Length = gr.Key, Words = gr };

// Using method-based query syntax.
var query2 = words.
    GroupBy(w => w.Length, w => w.ToUpper()).
    Select(g => new { Length = g.Key, Words = g }).
    OrderBy(o => o.Length);

foreach (var obj in query)
{
    Console.WriteLine("Words of length {0}:", obj.Length);
    foreach (string word in obj.Words)
        Console.WriteLine(word);
}

// This code example produces the following output:
//
// Words of length 3:
// THE
// FOX
// THE
// DOG
// Words of length 4:
// OVER
// LAZY
// Words of length 5:
// QUICK
// BROWN
// JUMPS

```

## Query Expression Syntax

Some of the more frequently used standard query operators have dedicated C# and Visual Basic language keyword syntax that enables them to be called as part of a *query expression*. For more information about standard query operators that have dedicated keywords and their corresponding syntaxes, see [Query Expression Syntax for Standard Query Operators \(C#\)](#).

## Extending the Standard Query Operators

You can augment the set of standard query operators by creating domain-specific methods that are appropriate for your target domain or technology. You can also replace the standard query operators with your own implementations that provide additional services such as remote evaluation, query translation, and optimization. See [AsEnumerable](#) for an example.

## Related Sections

The following links take you to topics that provide additional information about the various standard query operators based on functionality.

[Sorting Data \(C#\)](#)

[Set Operations \(C#\)](#)

[Filtering Data \(C#\)](#)

[Quantifier Operations \(C#\)](#)

[Projection Operations \(C#\)](#)

[Partitioning Data \(C#\)](#)

[Join Operations \(C#\)](#)

[Grouping Data \(C#\)](#)

[Generation Operations \(C#\)](#)

[Equality Operations \(C#\)](#)

[Element Operations \(C#\)](#)

[Converting Data Types \(C#\)](#)

[Concatenation Operations \(C#\)](#)

[Aggregation Operations \(C#\)](#)

## See also

- [Enumerable](#)
- [Queryable](#)
- [Introduction to LINQ Queries \(C#\)](#)
- [Query Expression Syntax for Standard Query Operators \(C#\)](#)
- [Classification of Standard Query Operators by Manner of Execution \(C#\)](#)
- [Extension Methods](#)

# Query Expression Syntax for Standard Query Operators (C#)

1/23/2019 • 2 minutes to read • [Edit Online](#)

Some of the more frequently used standard query operators have dedicated C# language keyword syntax that enables them to be called as part of a *query expression*. A query expression is a different, more readable form of expressing a query than its *method-based* equivalent. Query expression clauses are translated into calls to the query methods at compile time.

## Query Expression Syntax Table

The following table lists the standard query operators that have equivalent query expression clauses.

METHOD	C# QUERY EXPRESSION SYNTAX
<a href="#">Cast</a>	Use an explicitly typed range variable, for example:  <code>from int i in numbers</code>  (For more information, see <a href="#">from clause</a> .)
<a href="#">GroupBy</a>	<code>group ... by</code>  -or-  <code>group ... by ... into ...</code>  (For more information, see <a href="#">group clause</a> .)
<a href="#">GroupJoin&lt;TOuter,TInner,TKey,TResult&gt;(IEnumerable&lt;TOuter&gt;, IEnumerable&lt;TInner&gt;, Func&lt;TOuter,TKey&gt;, Func&lt;TInner,TKey&gt;, Func&lt;TOuter,IEnumerable&lt;TInner&gt;,TResult&gt;)</a>	<code>join ... in ... on ... equals ... into ...</code>  (For more information, see <a href="#">join clause</a> .)
<a href="#">Join&lt;TOuter,TInner,TKey,TResult&gt;(IEnumerable&lt;TOuter&gt;, IEnumerable&lt;TInner&gt;, Func&lt;TOuter,TKey&gt;, Func&lt;TInner,TKey&gt;, Func&lt;TOuter,TInner,TResult&gt;)</a>	<code>join ... in ... on ... equals ...</code>  (For more information, see <a href="#">join clause</a> .)
<a href="#">OrderBy&lt;TSource,TKey&gt;(IEnumerable&lt;TSource&gt;, Func&lt;TSource,TKey&gt;)</a>	<code>orderby</code>  (For more information, see <a href="#">orderby clause</a> .)
<a href="#">OrderByDescending&lt;TSource,TKey&gt;(IEnumerable&lt;TSource&gt;, Func&lt;TSource,TKey&gt;)</a>	<code>orderby ... descending</code>  (For more information, see <a href="#">orderby clause</a> .)
<a href="#">Select</a>	<code>select</code>  (For more information, see <a href="#">select clause</a> .)

METHOD	C# QUERY EXPRESSION SYNTAX
<a href="#">SelectMany</a>	Multiple <code>from</code> clauses.  (For more information, see <a href="#">from clause</a> .)
<a href="#">ThenBy&lt;TSource,TKey&gt;(IEnumerable&lt;TSource&gt;, Func&lt;TSource,TKey&gt;)</a>	<code>orderby ..., ...</code>  (For more information, see <a href="#">orderby clause</a> .)
<a href="#">ThenByDescending&lt;TSource,TKey&gt;(IEnumerable&lt;TSource&gt;, Func&lt;TSource,TKey&gt;)</a>	<code>orderby ..., ... descending</code>  (For more information, see <a href="#">orderby clause</a> .)
<a href="#">Where</a>	<code>where</code>  (For more information, see <a href="#">where clause</a> .)

## See also

- [Enumerable](#)
- [Queryable](#)
- [Standard Query Operators Overview \(C#\)](#)
- [Classification of Standard Query Operators by Manner of Execution \(C#\)](#)

# Classification of Standard Query Operators by Manner of Execution (C#)

1/23/2019 • 2 minutes to read • [Edit Online](#)

The LINQ to Objects implementations of the standard query operator methods execute in one of two main ways: immediate or deferred. The query operators that use deferred execution can be additionally divided into two categories: streaming and non-streaming. If you know how the different query operators execute, it may help you understand the results that you get from a given query. This is especially true if the data source is changing or if you are building a query on top of another query. This topic classifies the standard query operators according to their manner of execution.

## Manners of Execution

### Immediate

Immediate execution means that the data source is read and the operation is performed at the point in the code where the query is declared. All the standard query operators that return a single, non-enumerable result execute immediately.

### Deferred

Deferred execution means that the operation is not performed at the point in the code where the query is declared. The operation is performed only when the query variable is enumerated, for example by using a `foreach` statement. This means that the results of executing the query depend on the contents of the data source when the query is executed rather than when the query is defined. If the query variable is enumerated multiple times, the results might differ every time. Almost all the standard query operators whose return type is `IEnumerable<T>` or `IOrderedEnumerable<TElement>` execute in a deferred manner.

Query operators that use deferred execution can be additionally classified as streaming or non-streaming.

### Streaming

Streaming operators do not have to read all the source data before they yield elements. At the time of execution, a streaming operator performs its operation on each source element as it is read and yields the element if appropriate. A streaming operator continues to read source elements until a result element can be produced. This means that more than one source element might be read to produce one result element.

### Non-Streaming

Non-streaming operators must read all the source data before they can yield a result element. Operations such as sorting or grouping fall into this category. At the time of execution, non-streaming query operators read all the source data, put it into a data structure, perform the operation, and yield the resulting elements.

## Classification Table

The following table classifies each standard query operator method according to its method of execution.

### NOTE

If an operator is marked in two columns, two input sequences are involved in the operation, and each sequence is evaluated differently. In these cases, it is always the first sequence in the parameter list that is evaluated in a deferred, streaming manner.

STANDARD QUERY OPERATOR	RETURN TYPE	IMMEDIATE EXECUTION	DEFERRED STREAMING EXECUTION	DEFERRED NON-STREAMING EXECUTION
Aggregate	TSource	X		
All	Boolean	X		
Any	Boolean	X		
AsEnumerable	IEnumerable<T>		X	
Average	Single numeric value	X		
Cast	IEnumerable<T>		X	
Concat	IEnumerable<T>		X	
Contains	Boolean	X		
Count	Int32	X		
DefaultIfEmpty	IEnumerable<T>		X	
Distinct	IEnumerable<T>		X	
ElementAt	TSource	X		
ElementAtOrDefault	TSource	X		
Empty	IEnumerable<T>	X		
Except	IEnumerable<T>		X	X
First	TSource	X		
FirstOrDefault	TSource	X		
GroupBy	IEnumerable<T>			X
GroupJoin	IEnumerable<T>		X	X
Intersect	IEnumerable<T>		X	X
Join	IEnumerable<T>		X	X
Last	TSource	X		
LastOrDefault	TSource	X		
LongCount	Int64	X		



STANDARD QUERY OPERATOR	RETURN TYPE	IMMEDIATE EXECUTION	DEFERRED STREAMING EXECUTION	DEFERRED NON-STREAMING EXECUTION
Max	Single numeric value, TSource, or TResult	X		
Min	Single numeric value, TSource, or TResult	X		
OfType	IEnumerable<T>		X	
OrderBy	IOrderedEnumerable<TElement>			X
OrderByDescending	IOrderedEnumerable<TElement>			X
Range	IEnumerable<T>		X	
Repeat	IEnumerable<T>		X	
Reverse	IEnumerable<T>			X
Select	IEnumerable<T>		X	
SelectMany	IEnumerable<T>		X	
SequenceEqual	Boolean	X		
Single	TSource	X		
SingleOrDefault	TSource	X		
Skip	IEnumerable<T>		X	
SkipWhile	IEnumerable<T>		X	
Sum	Single numeric value	X		
Take	IEnumerable<T>		X	
TakeWhile	IEnumerable<T>		X	
ThenBy	IOrderedEnumerable<TElement>			X
ThenByDescending	IOrderedEnumerable<TElement>			X
ToArray	TSource array	X		
ToDictionary	Dictionary<TKey,TValue>	X		

STANDARD QUERY OPERATOR	RETURN TYPE	IMMEDIATE EXECUTION	DEFERRED STREAMING EXECUTION	DEFERRED NON-STREAMING EXECUTION
<a href="#">ToList</a>	<a href="#">IList&lt;T&gt;</a>	X		
<a href="#">ToLookup</a>	<a href="#">ILookup&lt;TKey,TElement&gt;</a>	X		
<a href="#">Union</a>	<a href="#">IEnumerable&lt;T&gt;</a>		X	
<a href="#">Where</a>	<a href="#">IEnumerable&lt;T&gt;</a>		X	

## See also

- [Enumerable](#)
- [Standard Query Operators Overview \(C#\)](#)
- [Query Expression Syntax for Standard Query Operators \(C#\)](#)
- [LINQ to Objects \(C#\)](#)

# Sorting Data (C#)

3/16/2019 • 2 minutes to read • [Edit Online](#)

A sorting operation orders the elements of a sequence based on one or more attributes. The first sort criterion performs a primary sort on the elements. By specifying a second sort criterion, you can sort the elements within each primary sort group.

The following illustration shows the results of an alphabetical sort operation on a sequence of characters:

Source

G C F E B A D

Results

A B C D E F G

The standard query operator methods that sort data are listed in the following section.

## Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
OrderBy	Sorts values in ascending order.	<code>orderby</code>	<a href="#">Enumerable.OrderBy</a> <a href="#">Queryable.OrderBy</a>
OrderByDescending	Sorts values in descending order.	<code>orderby ... descending</code>	<a href="#">Enumerable.OrderByDescending</a> <a href="#">Queryable.OrderByDescending</a>
ThenBy	Performs a secondary sort in ascending order.	<code>orderby ..., ...</code>	<a href="#">Enumerable.ThenBy</a> <a href="#">Queryable.ThenBy</a>
ThenByDescending	Performs a secondary sort in descending order.	<code>orderby ..., ... descending</code>	<a href="#">Enumerable.ThenByDescending</a> <a href="#">Queryable.ThenByDescending</a>
Reverse	Reverses the order of the elements in a collection.	Not applicable.	<a href="#">Enumerable.Reverse</a> <a href="#">Queryable.Reverse</a>

## Query Expression Syntax Examples

### Primary Sort Examples

#### Primary Ascending Sort

The following example demonstrates how to use the `orderby` clause in a LINQ query to sort the strings in an array by string length, in ascending order.

```

string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                           orderby word.Length
                           select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
    quick
    brown
    jumps
*/

```

### Primary Descending Sort

The next example demonstrates how to use the `orderby descending` clause in a LINQ query to sort the strings by their first letter, in descending order.

```

string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                           orderby word.Substring(0, 1) descending
                           select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    quick
    jumps
    fox
    brown
*/

```

## Secondary Sort Examples

### Secondary Ascending Sort

The following example demonstrates how to use the `orderby` clause in a LINQ query to perform a primary and secondary sort of the strings in an array. The strings are sorted primarily by length and secondarily by the first letter of the string, both in ascending order.

```

string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                           orderby word.Length, word.Substring(0, 1)
                           select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    fox
    the
    brown
    jumps
    quick
*/

```

### Secondary Descending Sort

The next example demonstrates how to use the `orderby descending` clause in a LINQ query to perform a primary sort, in ascending order, and a secondary sort, in descending order. The strings are sorted primarily by length and secondarily by the first letter of the string.

```

string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                           orderby word.Length, word.Substring(0, 1) descending
                           select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
    quick
    jumps
    brown
*/

```

## See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [orderby clause](#)
- [How to: Order the Results of a Join Clause](#)
- [How to: Sort or Filter Text Data by Any Word or Field \(LINQ\) \(C#\)](#)

# Set Operations (C#)

3/25/2019 • 2 minutes to read • [Edit Online](#)

Set operations in LINQ refer to query operations that produce a result set that is based on the presence or absence of equivalent elements within the same or separate collections (or sets).

The standard query operator methods that perform set operations are listed in the following section.

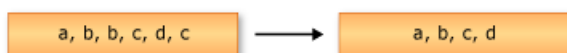
## Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
Distinct	Removes duplicate values from a collection.	Not applicable.	<a href="#">Enumerable.Distinct</a> <a href="#">Queryable.Distinct</a>
Except	Returns the set difference, which means the elements of one collection that do not appear in a second collection.	Not applicable.	<a href="#">Enumerable.Except</a> <a href="#">Queryable.Except</a>
Intersect	Returns the set intersection, which means elements that appear in each of two collections.	Not applicable.	<a href="#">Enumerable.Intersect</a> <a href="#">Queryable.Intersect</a>
Union	Returns the set union, which means unique elements that appear in either of two collections.	Not applicable.	<a href="#">Enumerable.Union</a> <a href="#">Queryable.Union</a>

## Comparison of Set Operations

### Distinct

The following illustration depicts the behavior of the [Enumerable.Distinct](#) method on a sequence of characters. The returned sequence contains the unique elements from the input sequence.



### Except

The following illustration depicts the behavior of [Enumerable.Except](#). The returned sequence contains only the elements from the first input sequence that are not in the second input sequence.



### Intersect

The following illustration depicts the behavior of [Enumerable.Intersect](#). The returned sequence contains the

elements that are common to both of the input sequences.



## Union

The following illustration depicts a union operation on two sequences of characters. The returned sequence contains the unique elements from both input sequences.



## See also

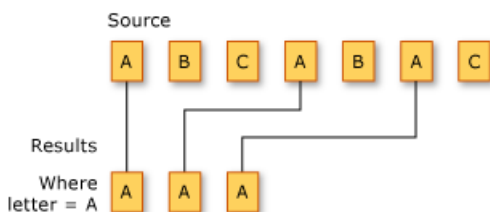
- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [How to: Combine and Compare String Collections \(LINQ\) \(C#\)](#)
- [How to: Find the Set Difference Between Two Lists \(LINQ\) \(C#\)](#)

# Filtering Data (C#)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Filtering refers to the operation of restricting the result set to contain only those elements that satisfy a specified condition. It is also known as selection.

The following illustration shows the results of filtering a sequence of characters. The predicate for the filtering operation specifies that the character must be 'A'.



The standard query operator methods that perform selection are listed in the following section.

## Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
OfType	Selects values, depending on their ability to be cast to a specified type.	Not applicable.	<a href="#">Enumerable.OfType</a> <a href="#">Queryable.OfType</a>
Where	Selects values that are based on a predicate function.	<code>where</code>	<a href="#">Enumerable.Where</a> <a href="#">Queryable.Where</a>

## Query Expression Syntax Example

The following example uses the `where` clause to filter from an array those strings that have a specific length.

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                           where word.Length == 3
                           select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
*/
```

## See also

- [System.Linq](#)



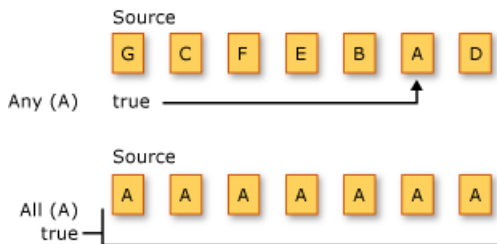
- [Standard Query Operators Overview \(C#\)](#)
- [where clause](#)
- [How to: Dynamically Specify Predicate Filters at Runtime](#)
- [How to: Query An Assembly's Metadata with Reflection \(LINQ\) \(C#\)](#)
- [How to: Query for Files with a Specified Attribute or Name \(C#\)](#)
- [How to: Sort or Filter Text Data by Any Word or Field \(LINQ\) \(C#\)](#)

# Quantifier Operations (C#)

3/25/2019 • 2 minutes to read • [Edit Online](#)

Quantifier operations return a [Boolean](#) value that indicates whether some or all of the elements in a sequence satisfy a condition.

The following illustration depicts two different quantifier operations on two different source sequences. The first operation asks if one or more of the elements are the character 'A', and the result is `true`. The second operation asks if all the elements are the character 'A', and the result is `true`.



The standard query operator methods that perform quantifier operations are listed in the following section.

## Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
All	Determines whether all the elements in a sequence satisfy a condition.	Not applicable.	<a href="#">Enumerable.All</a> <a href="#">Queryable.All</a>
Any	Determines whether any elements in a sequence satisfy a condition.	Not applicable.	<a href="#">Enumerable.Any</a> <a href="#">Queryable.Any</a>
Contains	Determines whether a sequence contains a specified element.	Not applicable.	<a href="#">Enumerable.Contains</a> <a href="#">Queryable.Contains</a>

## See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [How to: Dynamically Specify Predicate Filters at Runtime](#)
- [How to: Query for Sentences that Contain a Specified Set of Words \(LINQ\) \(C#\)](#)

# Projection Operations (C#)

3/25/2019 • 3 minutes to read • [Edit Online](#)

Projection refers to the operation of transforming an object into a new form that often consists only of those properties that will be subsequently used. By using projection, you can construct a new type that is built from each object. You can project a property and perform a mathematical function on it. You can also project the original object without changing it.

The standard query operator methods that perform projection are listed in the following section.

## Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
Select	Projects values that are based on a transform function.	<code>select</code>	<a href="#">Enumerable.Select</a> <a href="#">Queryable.Select</a>
SelectMany	Projects sequences of values that are based on a transform function and then flattens them into one sequence.	Use multiple <code>from</code> clauses	<a href="#">Enumerable.SelectMany</a> <a href="#">Queryable.SelectMany</a>

## Query Expression Syntax Examples

### Select

The following example uses the `select` clause to project the first letter from each string in a list of strings.

```
List<string> words = new List<string>() { "an", "apple", "a", "day" };

var query = from word in words
            select word.Substring(0, 1);

foreach (string s in query)
    Console.WriteLine(s);

/* This code produces the following output:

    a
    a
    a
    d
*/
```

### SelectMany

The following example uses multiple `from` clauses to project each word from each string in a list of strings.

```

List<string> phrases = new List<string>() { "an apple a day", "the quick brown fox" };

var query = from phrase in phrases
            from word in phrase.Split(' ')
            select word;

foreach (string s in query)
    Console.WriteLine(s);

/* This code produces the following output:

    an
    apple
    a
    day
    the
    quick
    brown
    fox
*/

```

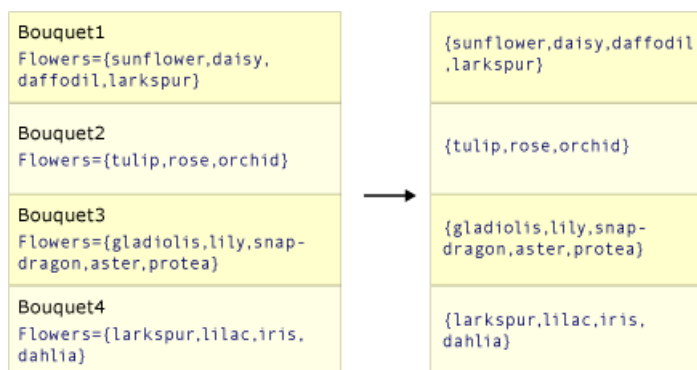
## Select versus SelectMany

The work of both `Select()` and `SelectMany()` is to produce a result value (or values) from source values.

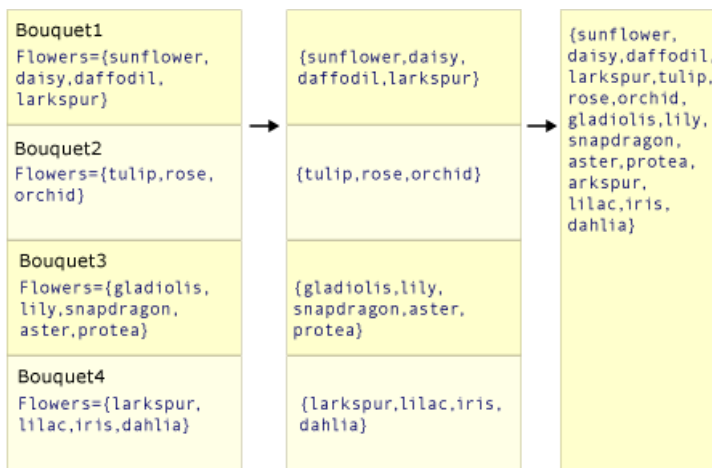
`Select()` produces one result value for every source value. The overall result is therefore a collection that has the same number of elements as the source collection. In contrast, `SelectMany()` produces a single overall result that contains concatenated sub-collections from each source value. The transform function that is passed as an argument to `SelectMany()` must return an enumerable sequence of values for each source value. These enumerable sequences are then concatenated by `SelectMany()` to create one large sequence.

The following two illustrations show the conceptual difference between the actions of these two methods. In each case, assume that the selector (transform) function selects the array of flowers from each source value.

This illustration depicts how `Select()` returns a collection that has the same number of elements as the source collection.



This illustration depicts how `SelectMany()` concatenates the intermediate sequence of arrays into one final result value that contains each value from each intermediate array.



## Code Example

The following example compares the behavior of `Select()` and `SelectMany()`. The code creates a "bouquet" of flowers by taking the first two items from each list of flower names in the source collection. In this example, the "single value" that the transform function `Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)` uses is itself a collection of values. This requires the extra `foreach` loop in order to enumerate each string in each sub-sequence.

```
class Bouquet
{
    public List<string> Flowers { get; set; }
}

static void SelectVsSelectMany()
{
    List<Bouquet> bouquets = new List<Bouquet>() {
        new Bouquet { Flowers = new List<string> { "sunflower", "daisy", "daffodil", "larkspur" } },
        new Bouquet { Flowers = new List<string> { "tulip", "rose", "orchid" } },
        new Bouquet { Flowers = new List<string> { "gladiolis", "lily", "snapdragon", "aster", "protea" } },
        new Bouquet { Flowers = new List<string> { "larkspur", "lilac", "iris", "dahlia" } }
    };

    // ***** Select *****
    IEnumerable<List<string>> query1 = bouquets.Select(bq => bq.Flowers);

    // ***** SelectMany *****
    IEnumerable<string> query2 = bouquets.SelectMany(bq => bq.Flowers);

    Console.WriteLine("Results by using Select():");
    // Note the extra foreach loop here.
    foreach (IEnumerable<String> collection in query1)
        foreach (string item in collection)
            Console.WriteLine(item);

    Console.WriteLine("\nResults by using SelectMany():");
    foreach (string item in query2)
        Console.WriteLine(item);

    /* This code produces the following output:

    Results by using Select():
    sunflower
    daisy
    daffodil
    larkspur
    tulip
    rose
    orchid
    gladiolis
    lily
    snapdragon
```

```
snapdragon  
aster  
protea  
larkspur  
lilac  
iris  
dahlia
```

```
Results by using SelectMany():
```

```
sunflower  
daisy  
daffodil  
larkspur  
tulip  
rose  
orchid  
gladiolis  
lily  
snapdragon  
aster  
protea  
larkspur  
lilac  
iris  
dahlia
```

```
*/
```

```
}
```

## See also

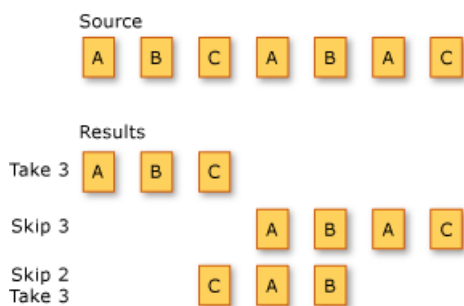
- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [select clause](#)
- [How to: Populate Object Collections from Multiple Sources \(LINQ\) \(C#\)](#)
- [How to: Split a File Into Many Files by Using Groups \(LINQ\) \(C#\)](#)

# Partitioning Data (C#)

4/2/2019 • 2 minutes to read • [Edit Online](#)

Partitioning in LINQ refers to the operation of dividing an input sequence into two sections, without rearranging the elements, and then returning one of the sections.

The following illustration shows the results of three different partitioning operations on a sequence of characters. The first operation returns the first three elements in the sequence. The second operation skips the first three elements and returns the remaining elements. The third operation skips the first two elements in the sequence and returns the next three elements.



The standard query operator methods that partition sequences are listed in the following section.

## Operators

OPERATOR NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
Skip	Skips elements up to a specified position in a sequence.	Not applicable.	<a href="#">Enumerable.Skip</a> <a href="#">Queryable.Skip</a>
SkipWhile	Skips elements based on a predicate function until an element does not satisfy the condition.	Not applicable.	<a href="#">Enumerable.SkipWhile</a> <a href="#">Queryable.SkipWhile</a>
Take	Takes elements up to a specified position in a sequence.	Not applicable.	<a href="#">Enumerable.Take</a> <a href="#">Queryable.Take</a>
TakeWhile	Takes elements based on a predicate function until an element does not satisfy the condition.	Not applicable.	<a href="#">Enumerable.TakeWhile</a> <a href="#">Queryable.TakeWhile</a>

## See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)

# Join Operations (C#)

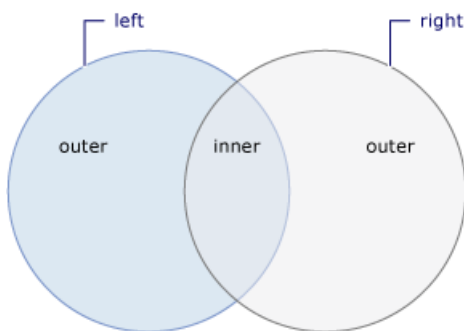
3/25/2019 • 2 minutes to read • [Edit Online](#)

A *join* of two data sources is the association of objects in one data source with objects that share a common attribute in another data source.

Joining is an important operation in queries that target data sources whose relationships to each other cannot be followed directly. In object-oriented programming, this could mean a correlation between objects that is not modeled, such as the backwards direction of a one-way relationship. An example of a one-way relationship is a Customer class that has a property of type City, but the City class does not have a property that is a collection of Customer objects. If you have a list of City objects and you want to find all the customers in each city, you could use a join operation to find them.

The join methods provided in the LINQ framework are [Join](#) and [GroupJoin](#). These methods perform equijoins, or joins that match two data sources based on equality of their keys. (For comparison, Transact-SQL supports join operators other than 'equals', for example the 'less than' operator.) In relational database terms, [Join](#) implements an inner join, a type of join in which only those objects that have a match in the other data set are returned. The [GroupJoin](#) method has no direct equivalent in relational database terms, but it implements a superset of inner joins and left outer joins. A left outer join is a join that returns each element of the first (left) data source, even if it has no correlated elements in the other data source.

The following illustration shows a conceptual view of two sets and the elements within those sets that are included in either an inner join or a left outer join.



## Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
Join	Joins two sequences based on key selector functions and extracts pairs of values.	<pre>join ... in ... on ... equals ...</pre>	<a href="#">Enumerable.Join</a> <a href="#">Queryable.Join</a>
GroupJoin	Joins two sequences based on key selector functions and groups the resulting matches for each element.	<pre>join ... in ... on ... equals ... into ...</pre>	<a href="#">Enumerable.GroupJoin</a> <a href="#">Queryable.GroupJoin</a>

## See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)



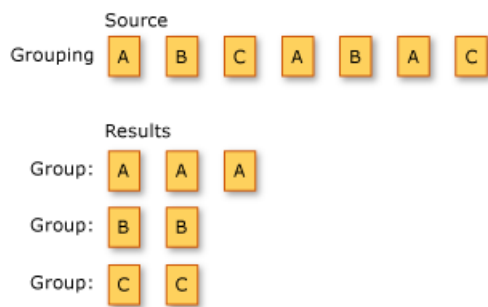
- [Anonymous Types](#)
- [Formulate Joins and Cross-Product Queries](#)
- [join clause](#)
- [How to: Join by Using Composite Keys](#)
- [How to: Join Content from Dissimilar Files \(LINQ\) \(C#\)](#)
- [How to: Order the Results of a Join Clause](#)
- [How to: Perform Custom Join Operations](#)
- [How to: Perform Grouped Joins](#)
- [How to: Perform Inner Joins](#)
- [How to: Perform Left Outer Joins](#)
- [How to: Populate Object Collections from Multiple Sources \(LINQ\) \(C#\)](#)

# Grouping Data (C#)

4/24/2019 • 2 minutes to read • [Edit Online](#)

Grouping refers to the operation of putting data into groups so that the elements in each group share a common attribute.

The following illustration shows the results of grouping a sequence of characters. The key for each group is the character.



The standard query operator methods that group data elements are listed in the following section.

## Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
GroupBy	Groups elements that share a common attribute. Each group is represented by an <a href="#">IGrouping&lt;TKey,TElement&gt;</a> object.	<code>group ... by</code>  -or-  <code>group ... by ... into ...</code>	<a href="#">Enumerable.GroupBy</a>  <a href="#">Queryable.GroupBy</a>
ToLookup	Inserts elements into a <a href="#">Lookup&lt;TKey,TElement&gt;</a> (a one-to-many dictionary) based on a key selector function.	Not applicable.	<a href="#">Enumerable.ToLookup</a>

## Query Expression Syntax Example

The following code example uses the `group by` clause to group integers in a list according to whether they are even or odd.

```

List<int> numbers = new List<int>() { 35, 44, 200, 84, 3987, 4, 199, 329, 446, 208 };

IEnumerable<IGrouping<int, int>> query = from number in numbers
                                         group number by number % 2;

foreach (var group in query)
{
    Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd numbers:");
    foreach (int i in group)
        Console.WriteLine(i);
}

/* This code produces the following output:

    Odd numbers:
    35
    3987
    199
    329

    Even numbers:
    44
    200
    84
    4
    446
    208

*/

```

## See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [group clause](#)
- [How to: Create a Nested Group](#)
- [How to: Group Files by Extension \(LINQ\) \(C#\)](#)
- [How to: Group Query Results](#)
- [How to: Perform a Subquery on a Grouping Operation](#)
- [How to: Split a File Into Many Files by Using Groups \(LINQ\) \(C#\)](#)

# Generation Operations (C#)

1/23/2019 • 2 minutes to read • [Edit Online](#)

Generation refers to creating a new sequence of values.

The standard query operator methods that perform generation are listed in the following section.

## Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
DefaultIfEmpty	Replaces an empty collection with a default valued singleton collection.	Not applicable.	<a href="#">Enumerable.DefaultIfEmpty</a> <a href="#">Queryable.DefaultIfEmpty</a>
Empty	Returns an empty collection.	Not applicable.	<a href="#">Enumerable.Empty</a>
Range	Generates a collection that contains a sequence of numbers.	Not applicable.	<a href="#">Enumerable.Range</a>
Repeat	Generates a collection that contains one repeated value.	Not applicable.	<a href="#">Enumerable.Repeat</a>

## See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)

# Equality Operations (C#)

1/23/2019 • 2 minutes to read • [Edit Online](#)

Two sequences whose corresponding elements are equal and which have the same number of elements are considered equal.

## Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
SequenceEqual	Determines whether two sequences are equal by comparing elements in a pair-wise manner.	Not applicable.	<a href="#">Enumerable.SequenceEqual</a> <a href="#">Queryable.SequenceEqual</a>

## See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [How to: Compare the Contents of Two Folders \(LINQ\) \(C#\)](#)

# Element Operations (C#)

1/23/2019 • 2 minutes to read • [Edit Online](#)

Element operations return a single, specific element from a sequence.

The standard query operator methods that perform element operations are listed in the following section.

## Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
ElementAt	Returns the element at a specified index in a collection.	Not applicable.	<a href="#">Enumerable.ElementAt</a> <a href="#">Queryable.ElementAt</a>
ElementAtOrDefault	Returns the element at a specified index in a collection or a default value if the index is out of range.	Not applicable.	<a href="#">Enumerable.ElementAtOrDefault</a> <a href="#">Queryable.ElementAtOrDefault</a>
First	Returns the first element of a collection, or the first element that satisfies a condition.	Not applicable.	<a href="#">Enumerable.First</a> <a href="#">Queryable.First</a>
FirstOrDefault	Returns the first element of a collection, or the first element that satisfies a condition. Returns a default value if no such element exists.	Not applicable.	<a href="#">Enumerable.FirstOrDefault</a> <a href="#">Queryable.FirstOrDefault</a> <a href="#">Queryable.FirstOrDefault&lt;TSource&gt;(IQueryable&lt;TSource&gt;)</a>
Last	Returns the last element of a collection, or the last element that satisfies a condition.	Not applicable.	<a href="#">Enumerable.Last</a> <a href="#">Queryable.Last</a>
LastOrDefault	Returns the last element of a collection, or the last element that satisfies a condition. Returns a default value if no such element exists.	Not applicable.	<a href="#">Enumerable.LastOrDefault</a> <a href="#">Queryable.LastOrDefault</a>
Single	Returns the only element of a collection or the only element that satisfies a condition. Throws an <a href="#">InvalidOperationException</a> if there is no element or more than one element to return.	Not applicable.	<a href="#">Enumerable.Single</a> <a href="#">Queryable.Single</a>

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
SingleOrDefault	Returns the only element of a collection or the only element that satisfies a condition. Returns a default value if there is no element to return. Throws an <a href="#">InvalidOperationException</a> if there is more than one element to return.	Not applicable.	<a href="#">Enumerable.SingleOrDefault</a> <a href="#">Queryable.SingleOrDefault</a>

## See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [How to: Query for the Largest File or Files in a Directory Tree \(LINQ\) \(C#\)](#)

# Converting Data Types (C#)

4/28/2019 • 2 minutes to read • [Edit Online](#)

Conversion methods change the type of input objects.

Conversion operations in LINQ queries are useful in a variety of applications. Following are some examples:

- The [Enumerable.AsEnumerable](#) method can be used to hide a type's custom implementation of a standard query operator.
- The [Enumerable.OfType](#) method can be used to enable non-parameterized collections for LINQ querying.
- The [Enumerable.ToArray](#), [Enumerable.ToDictionary](#), [Enumerable.ToList](#), and [Enumerable.ToLookup](#) methods can be used to force immediate query execution instead of deferring it until the query is enumerated.

## Methods

The following table lists the standard query operator methods that perform data-type conversions.

The conversion methods in this table whose names start with "As" change the static type of the source collection but do not enumerate it. The methods whose names start with "To" enumerate the source collection and put the items into the corresponding collection type.

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
AsEnumerable	Returns the input typed as <a href="#">IEnumerable&lt;T&gt;</a> .	Not applicable.	<a href="#">Enumerable.AsEnumerable</a>
AsQueryable	Converts a (generic) <a href="#">IEnumerable</a> to a (generic) <a href="#">IQueryable</a> .	Not applicable.	<a href="#">Queryable.AsQueryable</a>
Cast	Casts the elements of a collection to a specified type.	Use an explicitly typed range variable. For example: <div>from string str in words</div>	<a href="#">Enumerable.Cast</a> <a href="#">Queryable.Cast</a>
OfType	Filters values, depending on their ability to be cast to a specified type.	Not applicable.	<a href="#">Enumerable.OfType</a> <a href="#">Queryable.OfType</a>
ToArray	Converts a collection to an array. This method forces query execution.	Not applicable.	<a href="#">Enumerable.ToArray</a>
ToDictionary	Puts elements into a <a href="#">Dictionary&lt;TKey,TValue&gt;</a> based on a key selector function. This method forces query execution.	Not applicable.	<a href="#">Enumerable.ToDictionary</a>



METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
ToList	Converts a collection to a <a href="#">List&lt;T&gt;</a> . This method forces query execution.	Not applicable.	<a href="#">Enumerable.ToList</a>
ToLookup	Puts elements into a <a href="#">Lookup&lt;TKey,TElement&gt;</a> (a one-to-many dictionary) based on a key selector function. This method forces query execution.	Not applicable.	<a href="#">Enumerable.ToLookup</a>

## Query Expression Syntax Example

The following code example uses an explicitly-typed range variable to cast a type to a subtype before accessing a member that is available only on the subtype.

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

class CarnivorousPlant : Plant
{
    public string TrapType { get; set; }
}

static void Cast()
{
    Plant[] plants = new Plant[] {
        new CarnivorousPlant { Name = "Venus Fly Trap", TrapType = "Snap Trap" },
        new CarnivorousPlant { Name = "Pitcher Plant", TrapType = "Pitfall Trap" },
        new CarnivorousPlant { Name = "Sundew", TrapType = "Flypaper Trap" },
        new CarnivorousPlant { Name = "Waterwheel Plant", TrapType = "Snap Trap" }
    };

    var query = from CarnivorousPlant cPlant in plants
                where cPlant.TrapType == "Snap Trap"
                select cPlant;

    foreach (Plant plant in query)
        Console.WriteLine(plant.Name);

    /* This code produces the following output:

        Venus Fly Trap
        Waterwheel Plant
    */
}
```

## See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [from clause](#)
- [LINQ Query Expressions](#)
- [How to: Query an ArrayList with LINQ \(C#\)](#)

# Concatenation Operations (C#)

5/6/2019 • 2 minutes to read • [Edit Online](#)

Concatenation refers to the operation of appending one sequence to another.

The following illustration depicts a concatenation operation on two sequences of characters.



The standard query operator methods that perform concatenation are listed in the following section.

## Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
Concat	Concatenates two sequences to form one sequence.	Not applicable.	<a href="#">Enumerable.Concat</a> <a href="#">Queryable.Concat</a>

## See also

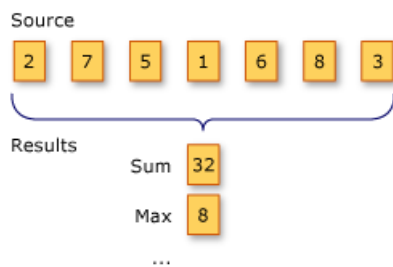
- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [How to: Combine and Compare String Collections \(LINQ\) \(C#\)](#)

# Aggregation Operations (C#)

3/30/2019 • 2 minutes to read • [Edit Online](#)

An aggregation operation computes a single value from a collection of values. An example of an aggregation operation is calculating the average daily temperature from a month's worth of daily temperature values.

The following illustration shows the results of two different aggregation operations on a sequence of numbers. The first operation sums the numbers. The second operation returns the maximum value in the sequence.



The standard query operator methods that perform aggregation operations are listed in the following section.

## Methods

METHOD NAME	DESCRIPTION	C# QUERY EXPRESSION SYNTAX	MORE INFORMATION
Aggregate	Performs a custom aggregation operation on the values of a collection.	Not applicable.	<a href="#">Enumerable.Aggregate</a> <a href="#">Queryable.Aggregate</a>
Average	Calculates the average value of a collection of values.	Not applicable.	<a href="#">Enumerable.Average</a> <a href="#">Queryable.Average</a>
Count	Counts the elements in a collection, optionally only those elements that satisfy a predicate function.	Not applicable.	<a href="#">Enumerable.Count</a> <a href="#">Queryable.Count</a>
LongCount	Counts the elements in a large collection, optionally only those elements that satisfy a predicate function.	Not applicable.	<a href="#">Enumerable.LongCount</a> <a href="#">Queryable.LongCount</a>
Max	Determines the maximum value in a collection.	Not applicable.	<a href="#">Enumerable.Max</a> <a href="#">Queryable.Max</a>
Min	Determines the minimum value in a collection.	Not applicable.	<a href="#">Enumerable.Min</a> <a href="#">Queryable.Min</a>
Sum	Calculates the sum of the values in a collection.	Not applicable.	<a href="#">Enumerable.Sum</a> <a href="#">Queryable.Sum</a>

## See also

- [System.Linq](#)
- [Standard Query Operators Overview \(C#\)](#)
- [How to: Compute Column Values in a CSV Text File \(LINQ\) \(C#\)](#)
- [How to: Query for the Largest File or Files in a Directory Tree \(LINQ\) \(C#\)](#)
- [How to: Query for the Total Number of Bytes in a Set of Folders \(LINQ\) \(C#\)](#)

# LINQ to Objects (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

The term "LINQ to Objects" refers to the use of LINQ queries with any [IEnumerable](#) or [IEnumerable<T>](#) collection directly, without the use of an intermediate LINQ provider or API such as [LINQ to SQL](#) or [LINQ to XML](#). You can use LINQ to query any enumerable collections such as [List<T>](#), [Array](#), or [Dictionary<TKey,TValue>](#). The collection may be user-defined or may be returned by a .NET Framework API.

In a basic sense, LINQ to Objects represents a new approach to collections. In the old way, you had to write complex `foreach` loops that specified how to retrieve data from a collection. In the LINQ approach, you write declarative code that describes what you want to retrieve.

In addition, LINQ queries offer three main advantages over traditional `foreach` loops:

1. They are more concise and readable, especially when filtering multiple conditions.
2. They provide powerful filtering, ordering, and grouping capabilities with a minimum of application code.
3. They can be ported to other data sources with little or no modification.

In general, the more complex the operation you want to perform on the data, the more benefit you will realize by using LINQ instead of traditional iteration techniques.

The purpose of this section is to demonstrate the LINQ approach with some select examples. It is not intended to be exhaustive.

## In This Section

### [LINQ and Strings \(C#\)](#)

Explains how LINQ can be used to query and transform strings and collections of strings. Also includes links to topics that demonstrate these principles.

### [LINQ and Reflection \(C#\)](#)

Links to a sample that demonstrates how LINQ uses reflection.

### [LINQ and File Directories \(C#\)](#)

Explains how LINQ can be used to interact with file systems. Also includes links to topics that demonstrate these concepts.

### [How to: Query an ArrayList with LINQ \(C#\)](#)

Demonstrates how to query an ArrayList in C#.

### [How to: Add Custom Methods for LINQ Queries \(C#\)](#)

Explains how to extend the set of methods that you can use for LINQ queries by adding extension methods to the [IEnumerable<T>](#) interface.

### [Language-Integrated Query \(LINQ\) \(C#\)](#)

Provides links to topics that explain LINQ and provide examples of code that perform queries.

# LINQ and strings (C#)

1/23/2019 • 2 minutes to read • [Edit Online](#)

LINQ can be used to query and transform strings and collections of strings. It can be especially useful with semi-structured data in text files. LINQ queries can be combined with traditional string functions and regular expressions. For example, you can use the [String.Split](#) or [Regex.Split](#) method to create an array of strings that you can then query or modify by using LINQ. You can use the [Regex.IsMatch](#) method in the `where` clause of a LINQ query. And you can use LINQ to query or modify the [MatchCollection](#) results returned by a regular expression.

You can also use the techniques described in this section to transform semi-structured text data to XML. For more information, see [How to: Generate XML from CSV Files](#).

The examples in this section fall into two categories:

## Querying a block of text

You can query, analyze, and modify text blocks by splitting them into a queryable array of smaller strings by using the [String.Split](#) method or the [Regex.Split](#) method. You can split the source text into words, sentences, paragraphs, pages, or any other criteria, and then perform additional splits if they are required in your query.

- [How to: Count Occurrences of a Word in a String \(LINQ\) \(C#\)](#)  
Shows how to use LINQ for simple querying over text.
- [How to: Query for Sentences that Contain a Specified Set of Words \(LINQ\) \(C#\)](#)  
Shows how to split text files on arbitrary boundaries and how to perform queries against each part.
- [How to: Query for Characters in a String \(LINQ\) \(C#\)](#)  
Demonstrates that a string is a queryable type.
- [How to: Combine LINQ Queries with Regular Expressions \(C#\)](#)  
Shows how to use regular expressions in LINQ queries for complex pattern matching on filtered query results.

## Querying semi-structured data in text format

Many different types of text files consist of a series of lines, often with similar formatting, such as tab- or comma-delimited files or fixed-length lines. After you read such a text file into memory, you can use LINQ to query and/or modify the lines. LINQ queries also simplify the task of combining data from multiple sources.

- [How to: Find the Set Difference Between Two Lists \(LINQ\) \(C#\)](#)  
Shows how to find all the strings that are present in one list but not the other.
- [How to: Sort or Filter Text Data by Any Word or Field \(LINQ\) \(C#\)](#)  
Shows how to sort text lines based on any word or field.
- [How to: Reorder the Fields of a Delimited File \(LINQ\) \(C#\)](#)  
Shows how to reorder fields in a line in a .csv file.
- [How to: Combine and Compare String Collections \(LINQ\) \(C#\)](#)

Shows how to combine string lists in various ways.

- [How to: Populate Object Collections from Multiple Sources \(LINQ\) \(C#\)](#)

Shows how to create object collections by using multiple text files as data sources.

- [How to: Join Content from Dissimilar Files \(LINQ\) \(C#\)](#)

Shows how to combine strings in two lists into a single string by using a matching key.

- [How to: Split a File Into Many Files by Using Groups \(LINQ\) \(C#\)](#)

Shows how to create new files by using a single file as a data source.

- [How to: Compute Column Values in a CSV Text File \(LINQ\) \(C#\)](#)

Shows how to perform mathematical computations on text data in .csv files.

## See also

- [Language-Integrated Query \(LINQ\) \(C#\)](#)
- [How to: Generate XML from CSV Files](#)

# How to: Count Occurrences of a Word in a String (LINQ) (C#)

5/14/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to use a LINQ query to count the occurrences of a specified word in a string. Note that to perform the count, first the [Split](#) method is called to create an array of words. There is a performance cost to the [Split](#) method. If the only operation on the string is to count the words, you should consider using the [Matches](#) or [IndexOf](#) methods instead. However, if performance is not a critical issue, or you have already split the sentence in order to perform other types of queries over it, then it makes sense to use LINQ to count the words or phrases as well.

## Example

```
class CountWords
{
    static void Main()
    {
        string text = @"Historically, the world of data and the world of objects" +
            @" have not been well integrated. Programmers work in C# or Visual Basic" +
            @" and also in SQL or XQuery. On the one side are concepts such as classes," +
            @" objects, fields, inheritance, and .NET Framework APIs. On the other side" +
            @" are tables, columns, rows, nodes, and separate languages for dealing with" +
            @" them. Data types often require translation between the two worlds; there are" +
            @" different standard functions. Because the object world has no notion of query, a" +
            @" query can only be represented as a string without compile-time type checking or" +
            @" IntelliSense support in the IDE. Transferring data from SQL tables or XML trees to" +
            @" objects in memory is often tedious and error-prone.";

        string searchTerm = "data";

        //Convert the string into an array of words
        string[] source = text.Split(new char[] { '.', '?', '!', ' ', ';', ':', ',' },
            StringSplitOptions.RemoveEmptyEntries);

        // Create the query. Use ToLowerInvariant to match "data" and "Data"
        var matchQuery = from word in source
            where word.ToLowerInvariant() == searchTerm.ToLowerInvariant()
            select word;

        // Count the matches, which executes the query.
        int wordCount = matchQuery.Count();
        Console.WriteLine("{0} occurrences(s) of the search term \"{1}\" were found.", wordCount, searchTerm);

        // Keep console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
3 occurrences(s) of the search term "data" were found.
*/
```

## Compiling the Code

Create a C# console application project, with `using` directives for the System.Linq and System.IO namespaces.



## See also

- [LINQ and Strings \(C#\)](#)

# How to: Query for Sentences that Contain a Specified Set of Words (LINQ) (C#)

5/14/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to find sentences in a text file that contain matches for each of a specified set of words. Although the array of search terms is hard-coded in this example, it could also be populated dynamically at runtime. In this example, the query returns the sentences that contain the words "Historically," "data," and "integrated."

## Example

```
class FindSentences
{
    static void Main()
    {
        string text = @"Historically, the world of data and the world of objects " +
            @"have not been well integrated. Programmers work in C# or Visual Basic " +
            @"and also in SQL or XQuery. On the one side are concepts such as classes, " +
            @"objects, fields, inheritance, and .NET Framework APIs. On the other side " +
            @"are tables, columns, rows, nodes, and separate languages for dealing with " +
            @"them. Data types often require translation between the two worlds; there are " +
            @"different standard functions. Because the object world has no notion of query, a " +
            @"query can only be represented as a string without compile-time type checking or " +
            @"IntelliSense support in the IDE. Transferring data from SQL tables or XML trees to " +
            @"objects in memory is often tedious and error-prone.";

        // Split the text block into an array of sentences.
        string[] sentences = text.Split(new char[] { '.', '?', '!' });

        // Define the search terms. This list could also be dynamically populated at runtime.
        string[] wordsToMatch = { "Historically", "data", "integrated" };

        // Find sentences that contain all the terms in the wordsToMatch array.
        // Note that the number of terms to match is not specified at compile time.
        var sentenceQuery = from sentence in sentences
            let w = sentence.Split(new char[] { '.', '?', '!', ' ', ';', ':', ',' },
                StringSplitOptions.RemoveEmptyEntries)
            where w.Distinct().Intersect(wordsToMatch).Count() == wordsToMatch.Count()
            select sentence;

        // Execute the query. Note that you can explicitly type
        // the iteration variable here even though sentenceQuery
        // was implicitly typed.
        foreach (string str in sentenceQuery)
        {
            Console.WriteLine(str);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}

/* Output:
Historically, the world of data and the world of objects have not been well integrated
*/
```

The query works by first splitting the text into sentences, and then splitting the sentences into an array of strings

that hold each word. For each of these arrays, the [Distinct](#) method removes all duplicate words, and then the query performs an [Intersect](#) operation on the word array and the `wordsToMatch` array. If the count of the intersection is the same as the count of the `wordsToMatch` array, all words were found in the words and the original sentence is returned.

In the call to [Split](#), the punctuation marks are used as separators in order to remove them from the string. If you did not do this, for example you could have a string "Historically," that would not match "Historically" in the `wordsToMatch` array. You may have to use additional separators, depending on the types of punctuation found in the source text.

## Compiling the Code

Create a C# console application project, with `using` directives for the System.Linq and System.IO namespaces.

## See also

- [LINQ and Strings \(C#\)](#)

# How to: Query for Characters in a String (LINQ) (C#)

5/14/2019 • 2 minutes to read • [Edit Online](#)

Because the [String](#) class implements the generic [IEnumerable<T>](#) interface, any string can be queried as a sequence of characters. However, this is not a common use of LINQ. For complex pattern matching operations, use the [Regex](#) class.

## Example

The following example queries a string to determine the number of numeric digits it contains. Note that the query is "reused" after it is executed the first time. This is possible because the query itself does not store any actual results.

```
class QueryAString
{
    static void Main()
    {
        string aString = "ABCDE99F-J74-12-89A";

        // Select only those characters that are numbers
        IEnumerable<char> stringQuery =
            from ch in aString
            where Char.IsDigit(ch)
            select ch;

        // Execute the query
        foreach (char c in stringQuery)
            Console.Write(c + " ");

        // Call the Count method on the existing query.
        int count = stringQuery.Count();
        Console.WriteLine("Count = {0}", count);

        // Select all characters before the first '-'
        IEnumerable<char> stringQuery2 = aString.TakeWhile(c => c != '-');

        // Execute the second query
        foreach (char c in stringQuery2)
            Console.Write(c);

        Console.WriteLine(System.Environment.NewLine + "Press any key to exit");
        Console.ReadKey();
    }
}

/* Output:
Output: 9 9 7 4 1 2 8 9
Count = 8
ABCDE99F
*/
```

## Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

## See also

- [LINQ and Strings \(C#\)](#)
- [How to: Combine LINQ Queries with Regular Expressions \(C#\)](#)

# How to: Combine LINQ Queries with Regular Expressions (C#)

5/14/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to use the [Regex](#) class to create a regular expression for more complex matching in text strings. The LINQ query makes it easy to filter on exactly the files that you want to search with the regular expression, and to shape the results.

## Example

```
class QueryWithRegex
{
    public static void Main()
    {
        // Modify this path as necessary so that it accesses your version of Visual Studio.
        string startFolder = @"C:\Program Files (x86)\Microsoft Visual Studio 14.0\";
        // One of the following paths may be more appropriate on your computer.
        //string startFolder = @"C:\Program Files (x86)\Microsoft Visual Studio\2017\";

        // Take a snapshot of the file system.
        IEnumerable<System.IO.FileInfo> fileList = GetFiles(startFolder);

        // Create the regular expression to find all things "Visual".
        System.Text.RegularExpressions.Regex searchTerm =
            new System.Text.RegularExpressions.Regex(@"Visual (Basic|C#|C\+|Studio)");

        // Search the contents of each .htm file.
        // Remove the where clause to find even more matchedValues!
        // This query produces a list of files where a match
        // was found, and a list of the matchedValues in that file.
        // Note: Explicit typing of "Match" in select clause.
        // This is required because MatchCollection is not a
        // generic IEnumerable collection.
        var queryMatchingFiles =
            from file in fileList
            where file.Extension == ".htm"
            let fileText = System.IO.File.ReadAllText(file.FullName)
            let matches = searchTerm.Matches(fileText)
            where matches.Count > 0
            select new
            {
                name = file.FullName,
                matchedValues = from System.Text.RegularExpressions.Match match in matches
                               select match.Value
            };

        // Execute the query.
        Console.WriteLine("The term \"{0}\" was found in:", searchTerm.ToString());

        foreach (var v in queryMatchingFiles)
        {
            // Trim the path a bit, then write
            // the file name in which a match was found.
            string s = v.name.Substring(startFolder.Length - 1);
            Console.WriteLine(s);

            // For this file, write out all the matching strings
            foreach (var v2 in v.matchedValues)
            {

```

```

        Console.WriteLine("  " + v2);
    }
}

// Keep the console window open in debug mode
Console.WriteLine("Press any key to exit");
Console.ReadKey();
}

// This method assumes that the application has discovery
// permissions for all folders under the specified path.
static IEnumerable<System.IO.FileInfo> GetFiles(string path)
{
    if (!System.IO.Directory.Exists(path))
        throw new System.IO.DirectoryNotFoundException();

    string[] fileNames = null;
    List<System.IO.FileInfo> files = new List<System.IO.FileInfo>();

    fileNames = System.IO.Directory.GetFiles(path, "*", System.IO.SearchOption.AllDirectories);
    foreach (string name in fileNames)
    {
        files.Add(new System.IO.FileInfo(name));
    }
    return files;
}
}

```

Note that you can also query the [MatchCollection](#) object that is returned by a `Regex` search. In this example only the value of each match is produced in the results. However, it is also possible to use LINQ to perform all kinds of filtering, sorting, and grouping on that collection. Because [MatchCollection](#) is a non-generic [IEnumerable](#) collection, you have to explicitly state the type of the range variable in the query.

## Compiling the Code

Create a C# console application project with `using` directives for the `System.Linq` and `System.IO` namespaces.

## See also

- [LINQ and Strings \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

# How to: Find the Set Difference Between Two Lists (LINQ) (C#)

5/14/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to use LINQ to compare two lists of strings and output those lines that are in names1.txt but not in names2.txt.

## To create the data files

1. Copy names1.txt and names2.txt to your solution folder as shown in [How to: Combine and Compare String Collections \(LINQ\) \(C#\)](#).

## Example

```
class CompareLists
{
    static void Main()
    {
        // Create the IEnumerable data sources.
        string[] names1 = System.IO.File.ReadAllLines(@"../.././names1.txt");
        string[] names2 = System.IO.File.ReadAllLines(@"../.././names2.txt");

        // Create the query. Note that method syntax must be used here.
        IEnumerable<string> differenceQuery =
            names1.Except(names2);

        // Execute the query.
        Console.WriteLine("The following lines are in names1.txt but not names2.txt");
        foreach (string s in differenceQuery)
            Console.WriteLine(s);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
    The following lines are in names1.txt but not names2.txt
    Potra, Cristina
    Noriega, Fabricio
    Aw, Kam Foo
    Toyoshima, Tim
    Guy, Wey Yuan
    Garcia, Debra
    */
```

Some types of query operations in C#, such as [Except](#), [Distinct](#), [Union](#), and [Concat](#), can only be expressed in method-based syntax.

## Compiling the Code

Create a C# console application project, with `using` directives for the System.Linq and System.IO namespaces.

## See also

- [LINQ and Strings \(C#\)](#)





# How to: Sort or Filter Text Data by Any Word or Field (LINQ) (C#)

5/14/2019 • 2 minutes to read • [Edit Online](#)

The following example shows how to sort lines of structured text, such as comma-separated values, by any field in the line. The field may be dynamically specified at runtime. Assume that the fields in scores.csv represent a student's ID number, followed by a series of four test scores.

## To create a file that contains data

1. Copy the scores.csv data from the topic [How to: Join Content from Dissimilar Files \(LINQ\) \(C#\)](#) and save it to your solution folder.

## Example

```

public class SortLines
{
    static void Main()
    {
        // Create an IEnumerable data source
        string[] scores = System.IO.File.ReadAllLines(@"../../../../../scores.csv");

        // Change this to any value from 0 to 4.
        int sortField = 1;

        Console.WriteLine("Sorted highest to lowest by field [{0}]:", sortField);

        // Demonstrates how to return query from a method.
        // The query is executed here.
        foreach (string str in RunQuery(scores, sortField))
        {
            Console.WriteLine(str);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Returns the query variable, not query results!
    static IEnumerable<string> RunQuery(IEnumerable<string> source, int num)
    {
        // Split the string and sort on field[num]
        var scoreQuery = from line in source
                        let fields = line.Split(',')
                        orderby fields[num] descending
                        select line;

        return scoreQuery;
    }
}

/* Output (if sortField == 1):
Sorted highest to lowest by field [1]:
116, 99, 86, 90, 94
120, 99, 82, 81, 79
111, 97, 92, 81, 60
114, 97, 89, 85, 82
121, 96, 85, 91, 60
122, 94, 92, 91, 91
117, 93, 92, 80, 87
118, 92, 90, 83, 78
113, 88, 94, 65, 91
112, 75, 84, 91, 39
119, 68, 79, 88, 92
115, 35, 72, 91, 70
*/

```

This example also demonstrates how to return a query variable from a method.

## Compiling the Code

Create a C# console application project, with `using` directives for the System.Linq and System.IO namespaces.

## See also

- [LINQ and Strings \(C#\)](#)

# How to: Reorder the Fields of a Delimited File (LINQ) (C#)

5/14/2019 • 2 minutes to read • [Edit Online](#)

A comma-separated value (CSV) file is a text file that is often used to store spreadsheet data or other tabular data that is represented by rows and columns. By using the [Split](#) method to separate the fields, it is very easy to query and manipulate CSV files by using LINQ. In fact, the same technique can be used to reorder the parts of any structured line of text; it is not limited to CSV files.

In the following example, assume that the three columns represent students' "last name," "first name", and "ID." The fields are in alphabetical order based on the students' last names. The query produces a new sequence in which the ID column appears first, followed by a second column that combines the student's first name and last name. The lines are reordered according to the ID field. The results are saved into a new file and the original data is not modified.

## To create the data file

1. Copy the following lines into a plain text file that is named spreadsheet1.csv. Save the file in your project folder.

```
Adams, Terry, 120
Fakhouri, Fadi, 116
Feng, Hanying, 117
Garcia, Cesar, 114
Garcia, Debra, 115
Garcia, Hugo, 118
Mortensen, Sven, 113
O'Donnell, Claire, 112
Omelchenko, Svetlana, 111
Tucker, Lance, 119
Tucker, Michael, 122
Zabokritski, Eugene, 121
```

## Example

```

class CSVFiles
{
    static void Main(string[] args)
    {
        // Create the IEnumerable data source
        string[] lines = System.IO.File.ReadAllLines(@"../../../spreadsheet1.csv");

        // Create the query. Put field 2 first, then
        // reverse and combine fields 0 and 1 from the old field
        IEnumerable<string> query =
            from line in lines
            let x = line.Split(',')
            orderby x[2]
            select x[2] + ", " + (x[1] + " " + x[0]);

        // Execute the query and write out the new file. Note that WriteAllLines
        // takes a string[], so ToArray is called on the query.
        System.IO.File.WriteAllLines(@"../../../spreadsheet2.csv", query.ToArray());

        Console.WriteLine("Spreadsheet2.csv written to disk. Press any key to exit");
        Console.ReadKey();
    }
}

/* Output to spreadsheet2.csv:
111, Svetlana Omelchenko
112, Claire O'Donnell
113, Sven Mortensen
114, Cesar Garcia
115, Debra Garcia
116, Fadi Fakhouri
117, Hanying Feng
118, Hugo Garcia
119, Lance Tucker
120, Terry Adams
121, Eugene Zabokritski
122, Michael Tucker
*/

```

## Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

## See also

- [LINQ and Strings \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)
- [How to: Generate XML from CSV Files \(C#\)](#)

# How to: Combine and Compare String Collections (LINQ) (C#)

5/14/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to merge files that contain lines of text and then sort the results. Specifically, it shows how to perform a simple concatenation, a union, and an intersection on the two sets of text lines.

## To set up the project and the text files

1. Copy these names into a text file that is named names1.txt and save it in your project folder:

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

2. Copy these names into a text file that is named names2.txt and save it in your project folder. Note that the two files have some names in common.

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

## Example

```
class MergeStrings
{
    static void Main(string[] args)
    {
        //Put text files in your solution folder
        string[] fileA = System.IO.File.ReadAllLines(@"../../names1.txt");
        string[] fileB = System.IO.File.ReadAllLines(@"../../names2.txt");

        //Simple concatenation and sort. Duplicates are preserved.
        IEnumerable<string> concatQuery =
            fileA.Concat(fileB).OrderBy(s => s);

        // Pass the query variable to another function for execution.
        OutputQueryResults(concatQuery, "Simple concatenate and sort. Duplicates are preserved:");

        // Concatenate and remove duplicate names based on
        // default string comparer.
        IEnumerable<string> uniqueNamesQuery =
```

```

        fileA.Union(fileB).OrderBy(s => s);
        OutputQueryResults(uniqueNamesQuery, "Union removes duplicate names:");

        // Find the names that occur in both files (based on
        // default string comparer).
        IEnumerable<string> commonNamesQuery =
            fileA.Intersect(fileB);
        OutputQueryResults(commonNamesQuery, "Merge based on intersect:");

        // Find the matching fields in each list. Merge the two
        // results by using Concat, and then
        // sort using the default string comparer.
        string nameMatch = "Garcia";

        IEnumerable<String> tempQuery1 =
            from name in fileA
            let n = name.Split(',')
            where n[0] == nameMatch
            select name;

        IEnumerable<string> tempQuery2 =
            from name2 in fileB
            let n2 = name2.Split(',')
            where n2[0] == nameMatch
            select name2;

        IEnumerable<string> nameMatchQuery =
            tempQuery1.Concat(tempQuery2).OrderBy(s => s);
        OutputQueryResults(nameMatchQuery, $"Concat based on partial name match \"{nameMatch}\":"");

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    static void OutputQueryResults(IEnumerable<string> query, string message)
    {
        Console.WriteLine(System.Environment.NewLine + message);
        foreach (string item in query)
        {
            Console.WriteLine(item);
        }
        Console.WriteLine("{0} total names in list", query.Count());
    }
}

/* Output:
Simple concatenate and sort. Duplicates are preserved:
Aw, Kam Foo
Bankov, Peter
Bankov, Peter
Beebe, Ann
Beebe, Ann
El Yassir, Mehdi
Garcia, Debra
Garcia, Hugo
Garcia, Hugo
Giakoumakis, Leo
Gilchrist, Beth
Guy, Wey Yuan
Holm, Michael
Holm, Michael
Liu, Jinghao
McLin, Nkenge
Myrcha, Jacek
Noriega, Fabricio
Potra, Cristina
Toyoshima, Tim
20 total names in list

```

```
Union removes duplicate names:
```

```
Aw, Kam Foo  
Bankov, Peter  
Beebe, Ann  
El Yassir, Mehdi  
Garcia, Debra  
Garcia, Hugo  
Giakoumakis, Leo  
Gilchrist, Beth  
Guy, Wey Yuan  
Holm, Michael  
Liu, Jinghao  
McLin, Nkenge  
Myrcha, Jacek  
Noriega, Fabricio  
Potra, Cristina  
Toyoshima, Tim  
16 total names in list
```

```
Merge based on intersect:
```

```
Bankov, Peter  
Holm, Michael  
Garcia, Hugo  
Beebe, Ann  
4 total names in list
```

```
Concat based on partial name match "Garcia":
```

```
Garcia, Debra  
Garcia, Hugo  
Garcia, Hugo  
3 total names in list
```

```
*/
```

## Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

## See also

- [LINQ and Strings \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)



# How to: Populate Object Collections from Multiple Sources (LINQ) (C#)

5/14/2019 • 4 minutes to read • [Edit Online](#)

This example shows how to merge data from different sources into a sequence of new types.

## NOTE

Don't try to join in-memory data or data in the file system with data that is still in a database. Such cross-domain joins can yield undefined results because of different ways in which join operations might be defined for database queries and other types of sources. Additionally, there is a risk that such an operation could cause an out-of-memory exception if the amount of data in the database is large enough. To join data from a database to in-memory data, first call `ToList` or `ToArray` on the database query, and then perform the join on the returned collection.

## To create the data file

Copy the `names.csv` and `scores.csv` files into your project folder, as described in [How to: Join Content from Dissimilar Files \(LINQ\) \(C#\)](#).

## Example

The following example shows how to use a named type `Student` to store merged data from two in-memory collections of strings that simulate spreadsheet data in `.csv` format. The first collection of strings represents the student names and IDs, and the second collection represents the student ID (in the first column) and four exam scores. The ID is used as the foreign key.

```
using System;
using System.Collections.Generic;
using System.Linq;

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public List<int> ExamScores { get; set; }
}

class PopulateCollection
{
    static void Main()
    {
        // These data files are defined in How to: Join Content from
        // Dissimilar Files (LINQ).

        // Each line of names.csv consists of a last name, a first name, and an
        // ID number, separated by commas. For example, Omelchenko,Svetlana,111
        string[] names = System.IO.File.ReadAllLines(@"../../names.csv");

        // Each line of scores.csv consists of an ID number and four test
        // scores, separated by commas. For example, 111, 97, 92, 81, 60
        string[] scores = System.IO.File.ReadAllLines(@"../../scores.csv");

        // Merge the data sources using a named type.
        // var could be used instead of an explicit type. Note the dynamic
```

```

// creation of a list of ints for the ExamScores member. The first item
// is skipped in the split string because it is the student ID,
// not an exam score.
IEnumerable<Student> queryNamesScores =
    from nameLine in names
    let splitName = nameLine.Split(',')
    from scoreLine in scores
    let splitScoreLine = scoreLine.Split(',')
    where Convert.ToInt32(splitName[2]) == Convert.ToInt32(splitScoreLine[0])
    select new Student()
    {
        FirstName = splitName[0],
        LastName = splitName[1],
        ID = Convert.ToInt32(splitName[2]),
        ExamScores = (from scoreAsText in splitScoreLine.Skip(1)
                      select Convert.ToInt32(scoreAsText)).
                      ToList()
    };

// Optional. Store the newly created student objects in memory
// for faster access in future queries. This could be useful with
// very large data files.
List<Student> students = queryNamesScores.ToList();

// Display each student's name and exam score average.
foreach (var student in students)
{
    Console.WriteLine("The average score of {0} {1} is {2}.",
        student.FirstName, student.LastName,
        student.ExamScores.Average());
}

//Keep console window open in debug mode
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}
}
/* Output:
The average score of Omelchenko Svetlana is 82.5.
The average score of O'Donnell Claire is 72.25.
The average score of Mortensen Sven is 84.5.
The average score of Garcia Cesar is 88.25.
The average score of Garcia Debra is 67.
The average score of Fakhouri Fadi is 92.25.
The average score of Feng Hanying is 88.
The average score of Garcia Hugo is 85.75.
The average score of Tucker Lance is 81.75.
The average score of Adams Terry is 85.25.
The average score of Zabokritski Eugene is 83.
The average score of Tucker Michael is 92.
*/

```

In the [select](#) clause, an object initializer is used to instantiate each new `Student` object by using the data from the two sources.

If you don't have to store the results of a query, anonymous types can be more convenient than named types. Named types are required if you pass the query results outside the method in which the query is executed. The following example executes the same task as the previous example, but uses anonymous types instead of named types:

```

// Merge the data sources by using an anonymous type.
// Note the dynamic creation of a list of ints for the
// ExamScores member. We skip 1 because the first string
// in the array is the student ID, not an exam score.
var queryNamesScores2 =
    from nameLine in names
    let splitName = nameLine.Split(',')
    from scoreLine in scores
    let splitScoreLine = scoreLine.Split(',')
    where Convert.ToInt32(splitName[2]) == Convert.ToInt32(splitScoreLine[0])
    select new
    {
        First = splitName[0],
        Last = splitName[1],
        ExamScores = (from scoreAsText in splitScoreLine.Skip(1)
                      select Convert.ToInt32(scoreAsText))
                      .ToList()
    };

// Display each student's name and exam score average.
foreach (var student in queryNamesScores2)
{
    Console.WriteLine("The average score of {0} {1} is {2}.",
        student.First, student.Last, student.ExamScores.Average());
}

```

## See also

- [LINQ and Strings \(C#\)](#)
- [Object and Collection Initializers](#)
- [Anonymous Types](#)

# How to: Split a File Into Many Files by Using Groups (LINQ) (C#)

5/14/2019 • 2 minutes to read • [Edit Online](#)

This example shows one way to merge the contents of two files and then create a set of new files that organize the data in a new way.

## To create the data files

1. Copy these names into a text file that is named names1.txt and save it in your project folder:

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

2. Copy these names into a text file that is named names2.txt and save it in your project folder: Note that the two files have some names in common.

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

## Example

```
class SplitWithGroups
{
    static void Main()
    {
        string[] fileA = System.IO.File.ReadAllLines(@"../.././names1.txt");
        string[] fileB = System.IO.File.ReadAllLines(@"../.././names2.txt");

        // Concatenate and remove duplicate names based on
        // default string comparer
        var mergeQuery = fileA.Union(fileB);

        // Group the names by the first letter in the last name.
        var groupQuery = from name in mergeQuery
            let n = name.Split(',')
            group name by n[0][0] into g
            orderby g.Key
            select g;
    }
}
```

```

// Create a new file for each group that was created
// Note that nested foreach loops are required to access
// individual items with each group.
foreach (var g in groupQuery)
{
    // Create the new file name.
    string fileName = @"../../testFile_" + g.Key + ".txt";

    // Output to display.
    Console.WriteLine(g.Key);

    // Write file.
    using (System.IO.StreamWriter sw = new System.IO.StreamWriter(fileName))
    {
        foreach (var item in g)
        {
            sw.WriteLine(item);
            // Output to console for example purposes.
            Console.WriteLine("    {0}", item);
        }
    }

    // Keep console window open in debug mode.
    Console.WriteLine("Files have been written. Press any key to exit");
    Console.ReadKey();
}
}
/* Output:
A
    Aw, Kam Foo
B
    Bankov, Peter
    Beebe, Ann
E
    El Yassir, Mehdi
G
    Garcia, Hugo
    Guy, Wey Yuan
    Garcia, Debra
    Gilchrist, Beth
    Giakoumakis, Leo
H
    Holm, Michael
L
    Liu, Jinghao
M
    Myrcha, Jacek
    McLin, Nkenge
N
    Noriega, Fabricio
P
    Potra, Cristina
T
    Toyoshima, Tim
*/

```

The program writes a separate file for each group in the same folder as the data files.

## Compiling the Code

Create a C# console application project, with `using` directives for the System.Linq and System.IO namespaces.

## See also

- [LINQ and Strings \(C#\)](#)

- [LINQ and File Directories \(C#\)](#)

# How to: Join Content from Dissimilar Files (LINQ) (C#)

5/14/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to join data from two comma-delimited files that share a common value that is used as a matching key. This technique can be useful if you have to combine data from two spreadsheets, or from a spreadsheet and from a file that has another format, into a new file. You can modify the example to work with any kind of structured text.

## To create the data files

1. Copy the following lines into a file that is named `scores.csv` and save it to your project folder. The file represents spreadsheet data. Column 1 is the student's ID, and columns 2 through 5 are test scores.

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

2. Copy the following lines into a file that is named `names.csv` and save it to your project folder. The file represents a spreadsheet that contains the student's last name, first name, and student ID.

```
Omelchenko,Svetlana,111
O'Donnell,Claire,112
Mortensen,Sven,113
Garcia,Cesar,114
Garcia,Debra,115
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Hugo,118
Tucker,Lance,119
Adams,Terry,120
Zabokritski,Eugene,121
Tucker,Michael,122
```

## Example

```
using System;
using System.Collections.Generic;
using System.Linq;

class JoinStrings
{
    static void Main()
    {
        // Join content from dissimilar files that contain
```

```

// Join records from dissimilar files and create
// related information. File names.csv contains the student
// name plus an ID number. File scores.csv contains the ID
// and a set of four test scores. The following query joins
// the scores to the student names by using ID as a
// matching key.

string[] names = System.IO.File.ReadAllLines(@"../.././names.csv");
string[] scores = System.IO.File.ReadAllLines(@"../.././scores.csv");

// Name:      Last[0],      First[1],  ID[2]
//           Omelchenko,    Svetlana,  11
// Score:      StudentID[0], Exam1[1]   Exam2[2],  Exam3[3],  Exam4[4]
//           111,           97,         92,       81,       60

// This query joins two dissimilar spreadsheets based on common ID value.
// Multiple from clauses are used instead of a join clause
// in order to store results of id.Split.
IEnumerable<string> scoreQuery1 =
    from name in names
    let nameFields = name.Split(',')
    from id in scores
    let scoreFields = id.Split(',')
    where Convert.ToInt32(nameFields[2]) == Convert.ToInt32(scoreFields[0])
    select nameFields[0] + "," + scoreFields[1] + "," + scoreFields[2]
        + "," + scoreFields[3] + "," + scoreFields[4];

// Pass a query variable to a method and execute it
// in the method. The query itself is unchanged.
OutputQueryResults(scoreQuery1, "Merge two spreadsheets:");

// Keep console window open in debug mode.
Console.WriteLine("Press any key to exit");
Console.ReadKey();
}

static void OutputQueryResults(IEnumerable<string> query, string message)
{
    Console.WriteLine(System.Environment.NewLine + message);
    foreach (string item in query)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("{0} total names in list", query.Count());
}
}

/* Output:
Merge two spreadsheets:
Omelchenko, 97, 92, 81, 60
O'Donnell, 75, 84, 91, 39
Mortensen, 88, 94, 65, 91
Garcia, 97, 89, 85, 82
Garcia, 35, 72, 91, 70
Fakhouri, 99, 86, 90, 94
Feng, 93, 92, 80, 87
Garcia, 92, 90, 83, 78
Tucker, 68, 79, 88, 92
Adams, 99, 82, 81, 79
Zabokritski, 96, 85, 91, 60
Tucker, 94, 92, 91, 91
12 total names in list
*/

```

## See also

- [LINQ and Strings \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)





# How to: Compute Column Values in a CSV Text File (LINQ) (C#)

5/14/2019 • 3 minutes to read • [Edit Online](#)

This example shows how to perform aggregate computations such as Sum, Average, Min, and Max on the columns of a .csv file. The example principles that are shown here can be applied to other types of structured text.

## To create the source file

1. Copy the following lines into a file that is named scores.csv and save it in your project folder. Assume that the first column represents a student ID, and subsequent columns represent scores from four exams.

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

## Example

```
class SumColumns
{
    static void Main(string[] args)
    {
        string[] lines = System.IO.File.ReadAllLines(@"../.././scores.csv");

        // Specifies the column to compute.
        int exam = 3;

        // Spreadsheet format:
        // Student ID    Exam#1  Exam#2  Exam#3  Exam#4
        // 111,          97,     92,      81,     60

        // Add one to exam to skip over the first column,
        // which holds the student ID.
        SingleColumn(lines, exam + 1);
        Console.WriteLine();
        MultiColumns(lines);

        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    static void SingleColumn(IEnumerable<string> strs, int examNum)
    {
        Console.WriteLine("Single Column Query:");

        // Parameter examNum specifies the column to
        // run the calculations on. This value could be
        // passed in dynamically at runtime.
```

```

// Variable columnQuery is an IEnumerable<int>.
// The following query performs two steps:
// 1) use Split to break each row (a string) into an array
//    of strings,
// 2) convert the element at position examNum to an int
//    and select it.
var columnQuery =
    from line in strs
    let elements = line.Split(',')
    select Convert.ToInt32(elements[examNum]);

// Execute the query and cache the results to improve
// performance. This is helpful only with very large files.
var results = columnQuery.ToList();

// Perform aggregate calculations Average, Max, and
// Min on the column specified by examNum.
double average = results.Average();
int max = results.Max();
int min = results.Min();

Console.WriteLine("Exam #{0}: Average:{1:##.##} High Score:{2} Low Score:{3}",
    examNum, average, max, min);
}

static void MultiColumns(IEnumerable<string> strs)
{
    Console.WriteLine("Multi Column Query:");

    // Create a query, multiColQuery. Explicit typing is used
    // to make clear that, when executed, multiColQuery produces
    // nested sequences. However, you get the same results by
    // using 'var'.

    // The multiColQuery query performs the following steps:
    // 1) use Split to break each row (a string) into an array
    //    of strings,
    // 2) use Skip to skip the "Student ID" column, and store the
    //    rest of the row in scores.
    // 3) convert each score in the current row from a string to
    //    an int, and select that entire sequence as one row
    //    in the results.
    IEnumerable<IEnumerable<int>> multiColQuery =
        from line in strs
        let elements = line.Split(',')
        let scores = elements.Skip(1)
        select (from str in scores
            select Convert.ToInt32(str));

    // Execute the query and cache the results to improve
    // performance.
    // ToArray could be used instead of ToList.
    var results = multiColQuery.ToList();

    // Find out how many columns you have in results.
    int columnCount = results[0].Count();

    // Perform aggregate calculations Average, Max, and
    // Min on each column.
    // Perform one iteration of the loop for each column
    // of scores.
    // You can use a for loop instead of a foreach loop
    // because you already executed the multiColQuery
    // query by calling ToList.
    for (int column = 0; column < columnCount; column++)
    {
        var results2 = from row in results
            select row.ElementAt(column);
        double average = results2.Average();
    }
}

```

```

double average = results2.Average();
int max = results2.Max();
int min = results2.Min();

// Add one to column because the first exam is Exam #1,
// not Exam #0.
Console.WriteLine("Exam #{0} Average: {1:##.##} High Score: {2} Low Score: {3}",
    column + 1, average, max, min);
}
}
}
/* Output:
Single Column Query:
Exam #4: Average:76.92 High Score:94 Low Score:39

Multi Column Query:
Exam #1 Average: 86.08 High Score: 99 Low Score: 35
Exam #2 Average: 86.42 High Score: 94 Low Score: 72
Exam #3 Average: 84.75 High Score: 91 Low Score: 65
Exam #4 Average: 76.92 High Score: 94 Low Score: 39
*/

```

The query works by using the [Split](#) method to convert each line of text into an array. Each array element represents a column. Finally, the text in each column is converted to its numeric representation. If your file is a tab-separated file, just update the argument in the `Split` method to `\t`.

## Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

## See also

- [LINQ and Strings \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

2 minutes to read

# How to: Query An Assembly's Metadata with Reflection (LINQ) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

The .NET Framework class library reflection APIs can be used to examine the metadata in a .NET assembly and create collections of types, type members, parameters, and so on that are in that assembly. Because these collections support the generic `IEnumerable` interface, they can be queried by using LINQ.

The following example shows how LINQ can be used with reflection to retrieve specific metadata about methods that match a specified search criterion. In this case, the query will find the names of all the methods in the assembly that return enumerable types such as arrays.

## Example

```
using System.Reflection;
using System.IO;
namespace LINQReflection
{
    class ReflectionHowTO
    {
        static void Main(string[] args)
        {
            Assembly assembly = Assembly.Load("System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089");
            var pubTypesQuery = from type in assembly.GetTypes()
                                where type.IsPublic
                                from method in type.GetMethods()
                                where method.ReturnType.IsArray == true
                                || ( method.ReturnType.GetInterface(
                                    typeof(System.Collections.Generic.IEnumerable<>).FullName ) != null
                                    && method.ReturnType.FullName != "System.String" )
                                group method.ToString() by type.ToString();

            foreach (var groupOfMethods in pubTypesQuery)
            {
                Console.WriteLine("Type: {0}", groupOfMethods.Key);
                foreach (var method in groupOfMethods)
                {
                    Console.WriteLine("  {0}", method);
                }
            }

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }
    }
}
```

The example uses the `GetTypes` method to return an array of types in the specified assembly. The `where` filter is applied so that only public types are returned. For each public type, a subquery is generated by using the `MethodInfo` array that is returned from the `GetMethods` call. These results are filtered to return only those methods whose return type is an array or else a type that implements `IEnumerable<T>`. Finally, these results are grouped by using the type name as a key.

## Compiling the Code

Create a C# console application project with `using` directives for the System.Linq and System.IO namespaces.

## See also

- [LINQ to Objects \(C#\)](#)

# LINQ and File Directories (C#)

1/23/2019 • 2 minutes to read • [Edit Online](#)

Many file system operations are essentially queries and are therefore well-suited to the LINQ approach.

Note that the queries in this section are non-destructive. They are not used to change the contents of the original files or folders. This follows the rule that queries should not cause any side-effects. In general, any code (including queries that perform create / update / delete operators) that modifies source data should be kept separate from the code that just queries the data.

This section contains the following topics:

## [How to: Query for Files with a Specified Attribute or Name \(C#\)](#)

Shows how to search for files by examining one or more properties of its [FileInfo](#) object.

## [How to: Group Files by Extension \(LINQ\) \(C#\)](#)

Shows how to return groups of [FileInfo](#) object based on their file name extension.

## [How to: Query for the Total Number of Bytes in a Set of Folders \(LINQ\) \(C#\)](#)

Shows how to return the total number of bytes in all the files in a specified directory tree.

## [How to: Compare the Contents of Two Folders \(LINQ\) \(C#\)](#)

Shows how to return all the files that are present in two specified folders, and also all the files that are present in one folder but not the other.

## [How to: Query for the Largest File or Files in a Directory Tree \(LINQ\) \(C#\)](#)

Shows how to return the largest or smallest file, or a specified number of files, in a directory tree.

## [How to: Query for Duplicate Files in a Directory Tree \(LINQ\) \(C#\)](#)

Shows how to group for all file names that occur in more than one location in a specified directory tree. Also shows how to perform more complex comparisons based on a custom comparer.

## [How to: Query the Contents of Files in a Folder \(LINQ\) \(C#\)](#)

Shows how to iterate through folders in a tree, open each file, and query the file's contents.

## Comments

There is some complexity involved in creating a data source that accurately represents the contents of the file system and handles exceptions gracefully. The examples in this section create a snapshot collection of [FileInfo](#) objects that represents all the files under a specified root folder and all its subfolders. The actual state of each [FileInfo](#) may change in the time between when you begin and end executing a query. For example, you can create a list of [FileInfo](#) objects to use as a data source. If you try to access the `Length` property in a query, the [FileInfo](#) object will try to access the file system to update the value of `Length`. If the file no longer exists, you will get a [FileNotFoundException](#) in your query, even though you are not querying the file system directly. Some queries in this section use a separate method that consumes these particular exceptions in certain cases. Another option is to keep your data source updated dynamically by using the [FileSystemWatcher](#).

## See also

- [LINQ to Objects \(C#\)](#)



# How to: Query for Files with a Specified Attribute or Name (C#)

5/14/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to find all files that have a specified file name extension (for example ".txt") in a specified directory tree. It also shows how to return either the newest or oldest file in the tree based on the creation time.

## Example

```
class FindFileByExtension
{
    // This query will produce the full path for all .txt files
    // under the specified folder including subfolders.
    // It orders the list according to the file name.
    static void Main()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*", System.IO.SearchOption.AllDirectories);

        //Create the query
        IEnumerable<System.IO.FileInfo> fileQuery =
            from file in fileList
            where file.Extension == ".txt"
            orderby file.Name
            select file;

        //Execute the query. This might write out a lot of files!
        foreach (System.IO.FileInfo fi in fileQuery)
        {
            Console.WriteLine(fi.FullName);
        }

        // Create and execute a new query by using the previous
        // query as a starting point. fileQuery is not
        // executed again until the call to Last()
        var newestFile =
            (from file in fileQuery
            orderby file.CreationTime
            select new { file.FullName, file.CreationTime })
            .Last();

        Console.WriteLine("\r\nThe newest .txt file is {0}. Creation time: {1}",
            newestFile.FullName, newestFile.CreationTime);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

## Compiling the Code

Create a C# console application project, with `using` directives for the System.Linq and System.IO namespaces.

## See also

- [LINQ to Objects \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

# How to: Group Files by Extension (LINQ) (C#)

5/14/2019 • 2 minutes to read • [Edit Online](#)

This example shows how LINQ can be used to perform advanced grouping and sorting operations on lists of files or folders. It also shows how to page output in the console window by using the [Skip](#) and [Take](#) methods.

## Example

The following query shows how to group the contents of a specified directory tree by the file name extension.

```
class GroupByExtension
{
    // This query will sort all the files under the specified folder
    // and subfolder into groups keyed by the file extension.
    private static void Main()
    {
        // Take a snapshot of the file system.
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\Common7";

        // Used in WriteLine to trim output lines.
        int trimLength = startFolder.Length;

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*", System.IO.SearchOption.AllDirectories);

        // Create the query.
        var queryGroupByExt =
            from file in fileList
            group file by file.Extension.ToLower() into fileGroup
            orderby fileGroup.Key
            select fileGroup;

        // Display one group at a time. If the number of
        // entries is greater than the number of lines
        // in the console window, then page the output.
        PageOutput(trimLength, queryGroupByExt);
    }

    // This method specifically handles group queries of FileInfo objects with string keys.
    // It can be modified to work for any long listings of data. Note that explicit typing
    // must be used in method signatures. The groupbyExtList parameter is a query that produces
    // groups of FileInfo objects with string keys.
    private static void PageOutput(int rootLength,
                                    IEnumerable<System.Linq.IGrouping<string, System.IO.FileInfo>>
groupbyExtList)
    {
        // Flag to break out of paging loop.
        bool goAgain = true;

        // "3" = 1 line for extension + 1 for "Press any key" + 1 for input cursor.
        int numLines = Console.WindowHeight - 3;

        // Iterate through the outer collection of groups.
        foreach (var filegroup in groupbyExtList)
        {
            // Start a new extension at the top of a page.
            int currentLine = 0;
```

```

// Output only as many lines of the current group as will fit in the window.
do
{
    Console.Clear();
    Console.WriteLine(filegroup.Key == String.Empty ? "[none]" : filegroup.Key);

    // Get 'numLines' number of items starting at number 'currentLine'.
    var resultPage = filegroup.Skip(currentLine).Take(numLines);

    //Execute the resultPage query
    foreach (var f in resultPage)
    {
        Console.WriteLine("\t{0}", f.FullName.Substring(rootLength));
    }

    // Increment the line counter.
    currentLine += numLines;

    // Give the user a chance to escape.
    Console.WriteLine("Press any key to continue or the 'End' key to break...");
    ConsoleKey key = Console.ReadKey().Key;
    if (key == ConsoleKey.End)
    {
        goAgain = false;
        break;
    }
} while (currentLine < filegroup.Count());

if (goAgain == false)
    break;
}
}
}

```

The output from this program can be long, depending on the details of the local file system and what the `startFolder` is set to. To enable viewing of all results, this example shows how to page through results. The same techniques can be applied to Windows and Web applications. Notice that because the code pages the items in a group, a nested `foreach` loop is required. There is also some additional logic to compute the current position in the list, and to enable the user to stop paging and exit the program. In this particular case, the paging query is run against the cached results from the original query. In other contexts, such as LINQ to SQL, such caching is not required.

## Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

## See also

- [LINQ to Objects \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

# How to: Query for the Total Number of Bytes in a Set of Folders (LINQ) (C#)

5/14/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to retrieve the total number of bytes used by all the files in a specified folder and all its subfolders.

## Example

The [Sum](#) method adds the values of all the items selected in the `select` clause. You can easily modify this query to retrieve the biggest or smallest file in the specified directory tree by calling the [Min](#) or [Max](#) method instead of [Sum](#).

```

class QuerySize
{
    public static void Main()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\VC#";

        // Take a snapshot of the file system.
        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<string> fileList = System.IO.Directory.GetFiles(startFolder, "**.*",
System.IO.SearchOption.AllDirectories);

        var fileQuery = from file in fileList
                        select GetFileLength(file);

        // Cache the results to avoid multiple trips to the file system.
        long[] fileLengths = fileQuery.ToArray();

        // Return the size of the largest file
        long largestFile = fileLengths.Max();

        // Return the total number of bytes in all the files under the specified folder.
        long totalBytes = fileLengths.Sum();

        Console.WriteLine("There are {0} bytes in {1} files under {2}",
            totalBytes, fileList.Count(), startFolder);
        Console.WriteLine("The largest files is {0} bytes.", largestFile);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // This method is used to swallow the possible exception
    // that can be raised when accessing the System.IO.FileInfo.Length property.
    static long GetFileLength(string filename)
    {
        long retval;
        try
        {
            System.IO.FileInfo fi = new System.IO.FileInfo(filename);
            retval = fi.Length;
        }
        catch (System.IO.FileNotFoundException)
        {
            // If a file is no longer present,
            // just add zero bytes to the total.
            retval = 0;
        }
        return retval;
    }
}

```

If you only have to count the number of bytes in a specified directory tree, you can do this more efficiently without creating a LINQ query, which incurs the overhead of creating the list collection as a data source. The usefulness of the LINQ approach increases as the query becomes more complex, or when you have to run multiple queries against the same data source.

The query calls out to a separate method to obtain the file length. It does this in order to consume the possible exception that will be raised if the file was deleted on another thread after the [FileInfo](#) object was created in the call to `GetFiles`. Even though the [FileInfo](#) object has already been created, the exception can occur because a [FileInfo](#) object will try to refresh its [Length](#) property with the most current length the first time the property is accessed. By putting this operation in a try-catch block outside the query, the code follows the rule of avoiding operations in queries that can cause side-effects. In general, great care must be taken when you consume exceptions to make

sure that an application is not left in an unknown state.

## Compiling the Code

Create a C# console application project, with `using` directives for the System.Linq and System.IO namespaces.

## See also

- [LINQ to Objects \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

# How to: Compare the Contents of Two Folders (LINQ) (C#)

5/14/2019 • 2 minutes to read • [Edit Online](#)

This example demonstrates three ways to compare two file listings:

- By querying for a Boolean value that specifies whether the two file lists are identical.
- By querying for the intersection to retrieve the files that are in both folders.
- By querying for the set difference to retrieve the files that are in one folder but not the other.

## NOTE

The techniques shown here can be adapted to compare sequences of objects of any type.

The `FileComparer` class shown here demonstrates how to use a custom comparer class together with the Standard Query Operators. The class is not intended for use in real-world scenarios. It just uses the name and length in bytes of each file to determine whether the contents of each folder are identical or not. In a real-world scenario, you should modify this comparer to perform a more rigorous equality check.

## Example

```
namespace QueryCompareTwoDirs
{
    class CompareDirs
    {
        static void Main(string[] args)
        {
            // Create two identical or different temporary folders
            // on a local drive and change these file paths.
            string pathA = @"C:\TestDir";
            string pathB = @"C:\TestDir2";

            System.IO.DirectoryInfo dir1 = new System.IO.DirectoryInfo(pathA);
            System.IO.DirectoryInfo dir2 = new System.IO.DirectoryInfo(pathB);

            // Take a snapshot of the file system.
            IEnumerable<System.IO.FileInfo> list1 = dir1.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);
            IEnumerable<System.IO.FileInfo> list2 = dir2.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

            //A custom file comparer defined below
            FileCompare myFileCompare = new FileCompare();

            // This query determines whether the two folders contain
            // identical file lists, based on the custom file comparer
            // that is defined in the FileCompare class.
            // The query executes immediately because it returns a bool.
            bool areIdentical = list1.SequenceEqual(list2, myFileCompare);

            if (areIdentical == true)
            {
                Console.WriteLine("the two folders are the same");
            }
        }
    }
}
```



```

        Console.WriteLine("The two folders are the same ");
    }
    else
    {
        Console.WriteLine("The two folders are not the same");
    }

    // Find the common files. It produces a sequence and doesn't
    // execute until the foreach statement.
    var queryCommonFiles = list1.Intersect(list2, myFileCompare);

    if (queryCommonFiles.Count() > 0)
    {
        Console.WriteLine("The following files are in both folders:");
        foreach (var v in queryCommonFiles)
        {
            Console.WriteLine(v.FullName); //shows which items end up in result list
        }
    }
    else
    {
        Console.WriteLine("There are no common files in the two folders.");
    }

    // Find the set difference between the two folders.
    // For this example we only check one way.
    var queryList1Only = (from file in list1
                          select file).Except(list2, myFileCompare);

    Console.WriteLine("The following files are in list1 but not list2:");
    foreach (var v in queryList1Only)
    {
        Console.WriteLine(v.FullName);
    }

    // Keep the console window open in debug mode.
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
}

// This implementation defines a very simple comparison
// between two FileInfo objects. It only compares the name
// of the files being compared and their length in bytes.
class FileCompare : System.Collections.Generic.IEqualityComparer<System.IO.FileInfo>
{
    public FileCompare() { }

    public bool Equals(System.IO.FileInfo f1, System.IO.FileInfo f2)
    {
        return (f1.Name == f2.Name &&
                f1.Length == f2.Length);
    }

    // Return a hash that reflects the comparison criteria. According to the
    // rules for IEqualityComparer<T>, if Equals is true, then the hash codes must
    // also be equal. Because equality as defined here is a simple value equality, not
    // reference identity, it is possible that two or more objects will produce the same
    // hash code.
    public int GetHashCode(System.IO.FileInfo fi)
    {
        string s = $"{fi.Name}{fi.Length}";
        return s.GetHashCode();
    }
}
}

```

## Compiling the Code

Create a C# console application project, with `using` directives for the System.Linq and System.IO namespaces.

## See also

- [LINQ to Objects \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

# How to: Query for the Largest File or Files in a Directory Tree (LINQ) (C#)

5/14/2019 • 3 minutes to read • [Edit Online](#)

This example shows five queries related to file size in bytes:

- How to retrieve the size in bytes of the largest file.
- How to retrieve the size in bytes of the smallest file.
- How to retrieve the [FileInfo](#) object largest or smallest file from one or more folders under a specified root folder.
- How to retrieve a sequence such as the 10 largest files.
- How to order files into groups based on their file size in bytes, ignoring files that are less than a specified size.

## Example

The following example contains five separate queries that show how to query and group files, depending on their file size in bytes. You can easily modify these examples to base the query on some other property of the [FileInfo](#) object.

```
class QueryBySize
{
    static void Main(string[] args)
    {
        QueryFilesBySize();
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    private static void QueryFilesBySize()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*", System.IO.SearchOption.AllDirectories);

        //Return the size of the largest file
        long maxSize =
            (from file in fileList
             let len = GetFileLength(file)
             select len)
            .Max();

        Console.WriteLine("The length of the largest file under {0} is {1}",
            startFolder, maxSize);

        // Return the FileInfo object for the largest file
        // by sorting and selecting from beginning of list
        System.IO.FileInfo longestFile =
            (from file in fileList
```

```

        let len = GetFileLength(file)
        where len > 0
        orderby len descending
        select file)
        .First();

Console.WriteLine("The largest file under {0} is {1} with a length of {2} bytes",
    startFolder, longestFile.FullName, longestFile.Length);

//Return the FileInfo of the smallest file
System.IO.FileInfo smallestFile =
    (from file in fileList
        let len = GetFileLength(file)
        where len > 0
        orderby len ascending
        select file).First();

Console.WriteLine("The smallest file under {0} is {1} with a length of {2} bytes",
    startFolder, smallestFile.FullName, smallestFile.Length);

//Return the FileInfos for the 10 largest files
// queryTenLargest is an IEnumerable<System.IO.FileInfo>
var queryTenLargest =
    (from file in fileList
        let len = GetFileLength(file)
        orderby len descending
        select file).Take(10);

Console.WriteLine("The 10 largest files under {0} are:", startFolder);

foreach (var v in queryTenLargest)
{
    Console.WriteLine("{0}: {1} bytes", v.FullName, v.Length);
}

// Group the files according to their size, leaving out
// files that are less than 200000 bytes.
var querySizeGroups =
    from file in fileList
    let len = GetFileLength(file)
    where len > 0
    group file by (len / 100000) into fileGroup
    where fileGroup.Key >= 2
    orderby fileGroup.Key descending
    select fileGroup;

foreach (var filegroup in querySizeGroups)
{
    Console.WriteLine(filegroup.Key.ToString() + "00000");
    foreach (var item in filegroup)
    {
        Console.WriteLine("\t{0}: {1}", item.Name, item.Length);
    }
}
}

// This method is used to swallow the possible exception
// that can be raised when accessing the FileInfo.Length property.
// In this particular case, it is safe to swallow the exception.
static long GetFileLength(System.IO.FileInfo fi)
{
    long retval;
    try
    {
        retval = fi.Length;
    }
    catch (System.IO.FileNotFoundException)
    {
        // If a file is no longer present,

```

```
        // just add zero bytes to the total.  
        retval = 0;  
    }  
    return retval;  
}  
  
}
```

To return one or more complete [FileInfo](#) objects, the query first must examine each one in the data source, and then sort them by the value of their `Length` property. Then it can return the single one or the sequence with the greatest lengths. Use [First](#) to return the first element in a list. Use [Take](#) to return the first n number of elements. Specify a descending sort order to put the smallest elements at the start of the list.

The query calls out to a separate method to obtain the file size in bytes in order to consume the possible exception that will be raised in the case where a file was deleted on another thread in the time period since the [FileInfo](#) object was created in the call to `GetFiles`. Even though the [FileInfo](#) object has already been created, the exception can occur because a [FileInfo](#) object will try to refresh its `Length` property by using the most current size in bytes the first time the property is accessed. By putting this operation in a try-catch block outside the query, we follow the rule of avoiding operations in queries that can cause side-effects. In general, great care must be taken when consuming exceptions, to make sure that an application is not left in an unknown state.

## Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

## See also

- [LINQ to Objects \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)

# How to: Query for Duplicate Files in a Directory Tree (LINQ) (C#)

5/14/2019 • 3 minutes to read • [Edit Online](#)

Sometimes files that have the same name may be located in more than one folder. For example, under the Visual Studio installation folder, several folders have a readme.htm file. This example shows how to query for such duplicate file names under a specified root folder. The second example shows how to query for files whose size and creation times also match.

## Example

```
class QueryDuplicateFileNames
{
    static void Main(string[] args)
    {
        // Uncomment QueryDuplicates2 to run that query.
        QueryDuplicates();
        // QueryDuplicates2();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void QueryDuplicates()
    {
        // Change the root drive or folder if necessary
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*", System.IO.SearchOption.AllDirectories);

        // used in WriteLine to keep the lines shorter
        int charsToSkip = startFolder.Length;

        // var can be used for convenience with groups.
        var queryDupNames =
            from file in fileList
            group file.FullName.Substring(charsToSkip) by file.Name into fileGroup
            where fileGroup.Count() > 1
            select fileGroup;

        // Pass the query to a method that will
        // output one page at a time.
        PageOutput<string, string>(queryDupNames);
    }

    // A Group key that can be passed to a separate method.
    // Override Equals and GetHashCode to define equality for the key.
    // Override ToString to provide a friendly name for Key.ToString()
    class PortableKey
    {
        public string Name { get; set; }
        public DateTime CreationTime { get; set; }
        public long Length { get; set; }
    }
}
```

```

public override bool Equals(object obj)
{
    PortableKey other = (PortableKey)obj;
    return other.CreationTime == this.CreationTime &&
        other.Length == this.Length &&
        other.Name == this.Name;
}

public override int GetHashCode()
{
    string str = $"{this.CreationTime}{this.Length}{this.Name}";
    return str.GetHashCode();
}

public override string ToString()
{
    return $"{this.Name} {this.Length} {this.CreationTime}";
}
}

static void QueryDuplicates2()
{
    // Change the root drive or folder if necessary.
    string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\Common7";

    // Make the lines shorter for the console display
    int charsToSkip = startFolder.Length;

    // Take a snapshot of the file system.
    System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);
    IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*", System.IO.SearchOption.AllDirectories);

    // Note the use of a compound key. Files that match
    // all three properties belong to the same group.
    // A named type is used to enable the query to be
    // passed to another method. Anonymous types can also be used
    // for composite keys but cannot be passed across method boundaries
    //
    var queryDupFiles =
        from file in fileList
        group file.FullName.Substring(charsToSkip) by
            new PortableKey { Name = file.Name, CreationTime = file.CreationTime, Length = file.Length }
into fileGroup
    where fileGroup.Count() > 1
    select fileGroup;

    var list = queryDupFiles.ToList();

    int i = queryDupFiles.Count();

    PageOutput<PortableKey, string>(queryDupFiles);
}

// A generic method to page the output of the QueryDuplications methods
// Here the type of the group must be specified explicitly. "var" cannot
// be used in method signatures. This method does not display more than one
// group per page.
private static void PageOutput<K, V>(IEnumerable<System.Linq.IGrouping<K, V>> groupByExtList)
{
    // Flag to break out of paging loop.
    bool goAgain = true;

    // "3" = 1 line for extension + 1 for "Press any key" + 1 for input cursor.
    int numLines = Console.WindowHeight - 3;

    // Iterate through the outer collection of groups.
    foreach (var filegroup in groupByExtList)
    {
        // Start a new extension at the top of a page.
        int currentLine = 0;

```

```

        // Output only as many lines of the current group as will fit in the window.
        do
        {
            Console.Clear();
            Console.WriteLine("Filename = {0}", filegroup.Key.ToString() == String.Empty ? "[none]" :
filegroup.Key.ToString());

            // Get 'numLines' number of items starting at number 'currentLine'.
            var resultPage = filegroup.Skip(currentLine).Take(numLines);

            //Execute the resultPage query
            foreach (var fileName in resultPage)
            {
                Console.WriteLine("\t{0}", fileName);
            }

            // Increment the line counter.
            currentLine += numLines;

            // Give the user a chance to escape.
            Console.WriteLine("Press any key to continue or the 'End' key to break...");
            ConsoleKey key = Console.ReadKey().Key;
            if (key == ConsoleKey.End)
            {
                goAgain = false;
                break;
            }
        } while (currentLine < filegroup.Count());

        if (goAgain == false)
            break;
    }
}
}

```

The first query uses a simple key to determine a match; this finds files that have the same name but whose contents might be different. The second query uses a compound key to match against three properties of the [FileInfo](#) object. This query is much more likely to find files that have the same name and similar or identical content.

## Compiling the Code

Create a C# console application project, with `using` directives for the `System.Linq` and `System.IO` namespaces.

## See also

- [LINQ to Objects \(C#\)](#)
- [LINQ and File Directories \(C#\)](#)



# How to: Query the Contents of Text Files in a Folder (LINQ) (C#)

5/14/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to query over all the files in a specified directory tree, open each file, and inspect its contents. This type of technique could be used to create indexes or reverse indexes of the contents of a directory tree. A simple string search is performed in this example. However, more complex types of pattern matching can be performed with a regular expression. For more information, see [How to: Combine LINQ Queries with Regular Expressions \(C#\)](#).

## Example

```

class QueryContents
{
    public static void Main()
    {
        // Modify this path as necessary.
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*", System.IO.SearchOption.AllDirectories);

        string searchTerm = @"Visual Studio";

        // Search the contents of each file.
        // A regular expression created with the RegEx class
        // could be used instead of the Contains method.
        // queryMatchingFiles is an IEnumerable<string>.
        var queryMatchingFiles =
            from file in fileList
            where file.Extension == ".htm"
            let fileText = GetFileText(file.FullName)
            where fileText.Contains(searchTerm)
            select file.FullName;

        // Execute the query.
        Console.WriteLine("The term \"{0}\" was found in:", searchTerm);
        foreach (string filename in queryMatchingFiles)
        {
            Console.WriteLine(filename);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Read the contents of the file.
    static string GetFileText(string name)
    {
        string fileContents = String.Empty;

        // If the file has been deleted since we took
        // the snapshot, ignore it and return the empty string.
        if (System.IO.File.Exists(name))
        {
            fileContents = System.IO.File.ReadAllText(name);
        }
        return fileContents;
    }
}

```

## Compiling the Code

Create a C# console application project, with `using` directives for the System.Linq and System.IO namespaces.

## See also

- [LINQ and File Directories \(C#\)](#)
- [LINQ to Objects \(C#\)](#)

# How to: Query an ArrayList with LINQ (C#)

1/23/2019 • 2 minutes to read • [Edit Online](#)

When using LINQ to query non-generic [IEnumerable](#) collections such as [ArrayList](#), you must explicitly declare the type of the range variable to reflect the specific type of the objects in the collection. For example, if you have an [ArrayList](#) of `Student` objects, your [from clause](#) should look like this:

```
var query = from Student s in arrList
...
```

By specifying the type of the range variable, you are casting each item in the [ArrayList](#) to a `Student`.

The use of an explicitly typed range variable in a query expression is equivalent to calling the [Cast](#) method. [Cast](#) throws an exception if the specified cast cannot be performed. [Cast](#) and [OfType](#) are the two Standard Query Operator methods that operate on non-generic [IEnumerable](#) types. For more information, see [Type Relationships in LINQ Query Operations](#).

## Example

The following example shows a simple query over an [ArrayList](#). Note that this example uses object initializers when the code calls the [Add](#) method, but this is not a requirement.

```

using System;
using System.Collections;
using System.Linq;

namespace NonGenericLINQ
{
    public class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int[] Scores { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arrList = new ArrayList();
            arrList.Add(
                new Student
                {
                    FirstName = "Svetlana", LastName = "Omelchenko", Scores = new int[] { 98, 92, 81, 60 }
                });
            arrList.Add(
                new Student
                {
                    FirstName = "Claire", LastName = "O'Donnell", Scores = new int[] { 75, 84, 91, 39 }
                });
            arrList.Add(
                new Student
                {
                    FirstName = "Sven", LastName = "Mortensen", Scores = new int[] { 88, 94, 65, 91 }
                });
            arrList.Add(
                new Student
                {
                    FirstName = "Cesar", LastName = "Garcia", Scores = new int[] { 97, 89, 85, 82 }
                });

            var query = from Student student in arrList
                        where student.Scores[0] > 95
                        select student;

            foreach (Student s in query)
                Console.WriteLine(s.LastName + ": " + s.Scores[0]);

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}

/* Output:
    Omelchenko: 98
    Garcia: 97
*/

```

## See also

- [LINQ to Objects \(C#\)](#)

# How to: Add Custom Methods for LINQ Queries (C#)

4/16/2019 • 5 minutes to read • [Edit Online](#)

You can extend the set of methods that you can use for LINQ queries by adding extension methods to the `IEnumerable<T>` interface. For example, in addition to the standard average or maximum operations, you can create a custom aggregate method to compute a single value from a sequence of values. You can also create a method that works as a custom filter or a specific data transform for a sequence of values and returns a new sequence. Examples of such methods are [Distinct](#), [Skip](#), and [Reverse](#).

When you extend the `IEnumerable<T>` interface, you can apply your custom methods to any enumerable collection. For more information, see [Extension Methods](#).

## Adding an Aggregate Method

An aggregate method computes a single value from a set of values. LINQ provides several aggregate methods, including [Average](#), [Min](#), and [Max](#). You can create your own aggregate method by adding an extension method to the `IEnumerable<T>` interface.

The following code example shows how to create an extension method called `Median` to compute a median for a sequence of numbers of type `double`.

```
public static class LINQExtension
{
    public static double Median(this IEnumerable<double> source)
    {
        if (source.Count() == 0)
        {
            throw new InvalidOperationException("Cannot compute median for an empty set.");
        }

        var sortedList = from number in source
                        orderby number
                        select number;

        int itemIndex = (int)sortedList.Count() / 2;

        if (sortedList.Count() % 2 == 0)
        {
            // Even number of items.
            return (sortedList.ElementAt(itemIndex) + sortedList.ElementAt(itemIndex - 1)) / 2;
        }
        else
        {
            // Odd number of items.
            return sortedList.ElementAt(itemIndex);
        }
    }
}
```

You call this extension method for any enumerable collection in the same way you call other aggregate methods from the `IEnumerable<T>` interface.

The following code example shows how to use the `Median` method for an array of type `double`.

```
double[] numbers1 = { 1.9, 2, 8, 4, 5.7, 6, 7.2, 0 };

var query1 = numbers1.Median();

Console.WriteLine("double: Median = " + query1);
```

```
/*
  This code produces the following output:

  Double: Median = 4.85
*/
```

## Overloading an Aggregate Method to Accept Various Types

You can overload your aggregate method so that it accepts sequences of various types. The standard approach is to create an overload for each type. Another approach is to create an overload that will take a generic type and convert it to a specific type by using a delegate. You can also combine both approaches.

### To create an overload for each type

You can create a specific overload for each type that you want to support. The following code example shows an overload of the `Median` method for the `integer` type.

```
//int overload

public static double Median(this IEnumerable<int> source)
{
    return (from num in source select (double)num).Median();
}
```

You can now call the `Median` overloads for both `integer` and `double` types, as shown in the following code:

```
double[] numbers1 = { 1.9, 2, 8, 4, 5.7, 6, 7.2, 0 };

var query1 = numbers1.Median();

Console.WriteLine("double: Median = " + query1);
```

```
int[] numbers2 = { 1, 2, 3, 4, 5 };

var query2 = numbers2.Median();

Console.WriteLine("int: Median = " + query2);
```

```
/*
  This code produces the following output:

  Double: Median = 4.85
  Integer: Median = 3
*/
```

### To create a generic overload

You can also create an overload that accepts a sequence of generic objects. This overload takes a delegate as a parameter and uses it to convert a sequence of objects of a generic type to a specific type.

The following code shows an overload of the `Median` method that takes the `Func<T,TResult>` delegate as a parameter. This delegate takes an object of generic type `T` and returns an object of type `double`.

```
// Generic overload.

public static double Median<T>(this IEnumerable<T> numbers,
                               Func<T, double> selector)
{
    return (from num in numbers select selector(num)).Median();
}
```

You can now call the `Median` method for a sequence of objects of any type. If the type does not have its own method overload, you have to pass a delegate parameter. In C#, you can use a lambda expression for this purpose. Also, in Visual Basic only, if you use the `Aggregate` or `Group By` clause instead of the method call, you can pass any value or expression that is in the scope this clause.

The following example code shows how to call the `Median` method for an array of integers and an array of strings. For strings, the median for the lengths of strings in the array is calculated. The example shows how to pass the `Func<T,TResult>` delegate parameter to the `Median` method for each case.

```
int[] numbers3 = { 1, 2, 3, 4, 5 };

/*
    You can use the num=>num lambda expression as a parameter for the Median method
    so that the compiler will implicitly convert its value to double.
    If there is no implicit conversion, the compiler will display an error message.
*/

var query3 = numbers3.Median(num => num);

Console.WriteLine("int: Median = " + query3);

string[] numbers4 = { "one", "two", "three", "four", "five" };

// With the generic overload, you can also use numeric properties of objects.

var query4 = numbers4.Median(str => str.Length);

Console.WriteLine("String: Median = " + query4);

/*
    This code produces the following output:

    Integer: Median = 3
    String: Median = 4
*/
```

## Adding a Method That Returns a Collection

You can extend the `IEnumerable<T>` interface with a custom query method that returns a sequence of values. In this case, the method must return a collection of type `IEnumerable<T>`. Such methods can be used to apply filters or data transforms to a sequence of values.

The following example shows how to create an extension method named `AlternateElements` that returns every other element in a collection, starting from the first element.

```
// Extension method for the IEnumerable<T> interface.
// The method returns every other element of a sequence.

public static IEnumerable<T> AlternateElements<T>(this IEnumerable<T> source)
{
    List<T> list = new List<T>();

    int i = 0;

    foreach (var element in source)
    {
        if (i % 2 == 0)
        {
            list.Add(element);
        }

        i++;
    }

    return list;
}
```

You can call this extension method for any enumerable collection just as you would call other methods from the [IEnumerable<T>](#) interface, as shown in the following code:

```
string[] strings = { "a", "b", "c", "d", "e" };

var query = strings.AlternateElements();

foreach (var element in query)
{
    Console.WriteLine(element);
}
/*
This code produces the following output:

a
c
e
*/
```

## See also

- [IEnumerable<T>](#)
- [Extension Methods](#)



# LINQ to XML Overview (C#)

6/4/2019 • 4 minutes to read • [Edit Online](#)

LINQ to XML provides an in-memory XML programming interface that leverages the .NET Language-Integrated Query (LINQ) Framework. LINQ to XML uses .NET capabilities and is comparable to an updated, redesigned Document Object Model (DOM) XML programming interface.

XML has been widely adopted as a way to format data in many contexts. For example, you can find XML on the Web, in configuration files, in Microsoft Office Word files, and in databases.

LINQ to XML is an up-to-date, redesigned approach to programming with XML. It provides the in-memory document modification capabilities of the Document Object Model (DOM), and supports LINQ query expressions. Although these query expressions are syntactically different from XPath, they provide similar functionality.

## LINQ to XML Developers

LINQ to XML targets a variety of developers. For an average developer who just wants to get something done, LINQ to XML makes XML easier by providing a query experience that is similar to SQL. With just a bit of study, programmers can learn to write succinct and powerful queries in their programming language of choice.

Professional developers can use LINQ to XML to greatly increase their productivity. With LINQ to XML, they can write less code that is more expressive, more compact, and more powerful. They can use query expressions from multiple data domains at the same time.

## What Is LINQ to XML?

LINQ to XML is a LINQ-enabled, in-memory XML programming interface that enables you to work with XML from within the .NET Framework programming languages.

LINQ to XML is like the Document Object Model (DOM) in that it brings the XML document into memory. You can query and modify the document, and after you modify it you can save it to a file or serialize it and send it over the Internet. However, LINQ to XML differs from DOM: It provides a new object model that is lighter weight and easier to work with, and that takes advantage of language features in C#.

The most important advantage of LINQ to XML is its integration with Language-Integrated Query (LINQ). This integration enables you to write queries on the in-memory XML document to retrieve collections of elements and attributes. The query capability of LINQ to XML is comparable in functionality (although not in syntax) to XPath and XQuery. The integration of LINQ in C# provides stronger typing, compile-time checking, and improved debugger support.

Another advantage of LINQ to XML is the ability to use query results as parameters to [XElement](#) and [XAttribute](#) object constructors enables a powerful approach to creating XML trees. This approach, called *functional construction*, enables developers to easily transform XML trees from one shape to another.

For example, you might have a typical XML purchase order as described in [Sample XML File: Typical Purchase Order \(LINQ to XML\)](#). By using LINQ to XML, you could run the following query to obtain the part number attribute value for every item element in the purchase order:

```
// Load the XML file from our project directory containing the purchase orders
var filename = "PurchaseOrder.xml";
var currentDirectory = Directory.GetCurrentDirectory();
var purchaseOrderFilepath = Path.Combine(currentDirectory, filename);

XElement purchaseOrder = XElement.Load($"{purchaseOrderFilepath}");

IEnumerable<string> partNos = from item in purchaseOrder.Descendants("Item")
                             select (string) item.Attribute("PartNumber");
```

This can be rewritten in method syntax form:

```
IEnumerable<string> partNos = purchaseOrder.Descendants("Item").Select(x => (string)
x.Attribute("PartNumber"));
```

As another example, you might want a list, sorted by part number, of the items with a value greater than \$100. To obtain this information, you could run the following query:

```
// Load the XML file from our project directory containing the purchase orders
var filename = "PurchaseOrder.xml";
var currentDirectory = Directory.GetCurrentDirectory();
var purchaseOrderFilepath = Path.Combine(currentDirectory, filename);

XElement purchaseOrder = XElement.Load($"{purchaseOrderFilepath}");

IEnumerable<XElement> pricesByPartNos = from item in purchaseOrder.Descendants("Item")
                                         where (int) item.Element("Quantity") * (decimal) item.Element("USPrice") >
100
                                         orderby (string)item.Element("PartNumber")
                                         select item;
```

Again, this can be rewritten in method syntax form:

```
IEnumerable<XElement> pricesByPartNos = purchaseOrder.Descendants("Item")
                                         .Where(item => (int)item.Element("Quantity") *
(decimal)item.Element("USPrice") > 100)
                                         .OrderBy(order => order.Element("PartNumber"));
```

In addition to these LINQ capabilities, LINQ to XML provides an improved XML programming interface. Using LINQ to XML, you can:

- Load XML from [files](#) or [streams](#).
- Serialize XML to files or streams.
- Create XML from scratch by using functional construction.
- Query XML using XPath-like axes.
- Manipulate the in-memory XML tree by using methods such as [Add](#), [Remove](#), [ReplaceWith](#), and [SetValue](#).
- Validate XML trees using XSD.
- Use a combination of these features to transform XML trees from one shape into another.

## Creating XML Trees

One of the most significant advantages of programming with LINQ to XML is that it is easy to create XML trees.

For example, to create a small XML tree, you can write code as follows:

```
XElement contacts =
new XElement("Contacts",
    new XElement("Contact",
        new XElement("Name", "Patrick Hines"),
        new XElement("Phone", "206-555-0144",
            new XAttribute("Type", "Home")),
        new XElement("phone", "425-555-0145",
            new XAttribute("Type", "Work")),
        new XElement("Address",
            new XElement("Street1", "123 Main St"),
            new XElement("City", "Mercer Island"),
            new XElement("State", "WA"),
            new XElement("Postal", "68042")
        )
    )
);
```

For more information, see [Creating XML Trees \(C#\)](#).

## See also

- [Reference \(LINQ to XML\)](#)
- [LINQ to XML vs. DOM \(C#\)](#)
- [LINQ to XML vs. Other XML Technologies](#)
- [System.Xml.Linq](#)

# LINQ to XML vs. DOM (C#)

6/4/2019 • 5 minutes to read • [Edit Online](#)

This section describes some key differences between LINQ to XML and the current predominant XML programming API, the W3C Document Object Model (DOM).

## New Ways to Construct XML Trees

In the W3C DOM, you build an XML tree from the bottom up; that is, you create a document, you create elements, and then you add the elements to the document.

For example, the following would be a typical way to create an XML tree using the Microsoft implementation of DOM, [XmlDocument](#):

```
XmlDocument doc = new XmlDocument();
XmlElement name = doc.CreateElement("Name");
name.InnerText = "Patrick Hines";
XmlElement phone1 = doc.CreateElement("Phone");
phone1.SetAttribute("Type", "Home");
phone1.InnerText = "206-555-0144";
XmlElement phone2 = doc.CreateElement("Phone");
phone2.SetAttribute("Type", "Work");
phone2.InnerText = "425-555-0145";
XmlElement street1 = doc.CreateElement("Street1");
street1.InnerText = "123 Main St";
XmlElement city = doc.CreateElement("City");
city.InnerText = "Mercer Island";
XmlElement state = doc.CreateElement("State");
state.InnerText = "WA";
XmlElement postal = doc.CreateElement("Postal");
postal.InnerText = "68042";
XmlElement address = doc.CreateElement("Address");
address.AppendChild(street1);
address.AppendChild(city);
address.AppendChild(state);
address.AppendChild(postal);
XmlElement contact = doc.CreateElement("Contact");
contact.AppendChild(name);
contact.AppendChild(phone1);
contact.AppendChild(phone2);
contact.AppendChild(address);
XmlElement contacts = doc.CreateElement("Contacts");
contacts.AppendChild(contact);
doc.AppendChild(contacts);
```

This style of coding does not visually provide much information about the structure of the XML tree. LINQ to XML supports this approach to constructing an XML tree, but also supports an alternative approach, *functional construction*. Functional construction uses the [XElement](#) and [XAttribute](#) constructors to build an XML tree.

Here is how you would construct the same XML tree by using LINQ to XML functional construction:

```

XElement contacts =
    new XElement("Contacts",
        new XElement("Contact",
            new XElement("Name", "Patrick Hines"),
            new XElement("Phone", "206-555-0144",
                new XAttribute("Type", "Home")),
            new XElement("phone", "425-555-0145",
                new XAttribute("Type", "Work")),
            new XElement("Address",
                new XElement("Street1", "123 Main St"),
                new XElement("City", "Mercer Island"),
                new XElement("State", "WA"),
                new XElement("Postal", "68042")
            )
        )
    );

```

Notice that indenting the code to construct the XML tree shows the structure of the underlying XML.

For more information, see [Creating XML Trees \(C#\)](#).

## Working Directly with XML Elements

When you program with XML, your primary focus is usually on XML elements and perhaps on attributes. In LINQ to XML, you can work directly with XML elements and attributes. For example, you can do the following:

- Create XML elements without using a document object at all. This simplifies programming when you have to work with fragments of XML trees.
- Load `T:System.Xml.Linq.XElement` objects directly from an XML file.
- Serialize `T:System.Xml.Linq.XElement` objects to a file or a stream.

Compare this to the W3C DOM, in which the XML document is used as a logical container for the XML tree. In DOM, XML nodes, including elements and attributes, must be created in the context of an XML document. Here is a fragment of the code to create a name element in DOM:

```

XmlDocument doc = new XmlDocument();
XmlElement name = doc.CreateElement("Name");
name.InnerText = "Patrick Hines";
doc.AppendChild(name);

```

If you want to use an element across multiple documents, you must import the nodes across documents. LINQ to XML avoids this layer of complexity.

When using LINQ to XML, you use the [XDocument](#) class only if you want to add a comment or processing instruction at the root level of the document.

## Simplified Handling of Names and Namespaces

Handling names, namespaces, and namespace prefixes is generally a complex part of XML programming. LINQ to XML simplifies names and namespaces by eliminating the requirement to deal with namespace prefixes. If you want to control namespace prefixes, you can. But if you decide to not explicitly control namespace prefixes, LINQ to XML will assign namespace prefixes during serialization if they are required, or will serialize using default namespaces if they are not. If default namespaces are used, there will be no namespace prefixes in the resulting document. For more information, see [Working with XML Namespaces \(C#\)](#).

Another problem with the DOM is that it does not let you change the name of a node. Instead, you have to create a

new node and copy all the child nodes to it, losing the original node identity. LINQ to XML avoids this problem by enabling you to set the [XName](#) property on a node.

## Static Method Support for Loading XML

LINQ to XML lets you load XML by using static methods, instead of instance methods. This simplifies loading and parsing. For more information, see [How to: Load XML from a File \(C#\)](#).

## Removal of Support for DTD Constructs

LINQ to XML further simplifies XML programming by removing support for entities and entity references. The management of entities is complex, and is rarely used. Removing their support increases performance and simplifies the programming interface. When a LINQ to XML tree is populated, all DTD entities are expanded.

## Support for Fragments

LINQ to XML does not provide an equivalent for the `XmlDocumentFragment` class. In many cases, however, the `XmlDocumentFragment` concept can be handled by the result of a query that is typed as `IEnumerable<T>` of `XNode`, or `IEnumerable<T>` of `XElement`.

## Support for XPathNavigator

LINQ to XML provides support for [XPathNavigator](#) through extension methods in the `System.Xml.XPath` namespace. For more information, see [System.Xml.XPath.Extensions](#).

## Support for White Space and Indentation

LINQ to XML handles white space more simply than the DOM.

A common scenario is to read indented XML, create an in-memory XML tree without any white space text nodes (that is, not preserving white space), perform some operations on the XML, and then save the XML with indentation. When you serialize the XML with formatting, only significant white space in the XML tree is preserved. This is the default behavior for LINQ to XML.

Another common scenario is to read and modify XML that has already been intentionally indented. You might not want to change this indentation in any way. In LINQ to XML, you can do this by preserving white space when you load or parse the XML and disabling formatting when you serialize the XML.

LINQ to XML stores white space as an [XText](#) node, instead of having a specialized [Whitespace](#) node type, as the DOM does.

## Support for Annotations

LINQ to XML elements support an extensible set of annotations. This is useful for tracking miscellaneous information about an element, such as schema information, information about whether the element is bound to a UI, or any other kind of application-specific information. For more information, see [LINQ to XML Annotations](#).

## Support for Schema Information

LINQ to XML provides support for XSD validation through extension methods in the `System.Xml.Schema` namespace. You can validate that an XML tree complies with an XSD. You can populate the XML tree with the post-schema-validation info set (PSVI). For more information, see [How to: Validate Using XSD](#) and [Extensions](#).

## See also

- [Getting Started \(LINQ to XML\)](#)

# LINQ to XML vs. Other XML Technologies

6/4/2019 • 3 minutes to read • [Edit Online](#)

This topic compares LINQ to XML to the following XML technologies: [XmlReader](#), XSLT, MSXML, and XmlLite. This information can help you decide which technology to use.

For a comparison of LINQ to XML to the Document Object Model (DOM), see [LINQ to XML vs. DOM \(C#\)](#).

## LINQ to XML vs. XmlReader

[XmlReader](#) is a fast, forward-only, non-caching parser.

LINQ to XML is implemented on top of [XmlReader](#), and they are tightly integrated. However, you can also use [XmlReader](#) by itself.

For example, suppose you are building a Web service that will parse hundreds of XML documents per second, and the documents have the same structure, meaning that you only have to write one implementation of the code to parse the XML. In this case, you would probably want to use [XmlReader](#) by itself.

In contrast, if you are building a system that parses many smaller XML documents, and each one is different, you would want to take advantage of the productivity improvements that LINQ to XML provides.

## LINQ to XML vs. XSLT

Both LINQ to XML and XSLT provide extensive XML document transformation capabilities. XSLT is a rule-based, declarative approach. Advanced XSLT programmers write XSLT in a functional programming style that emphasizes a stateless approach. Transformations can be written using pure functions that are implemented without side effects. This rule-based or functional approach is unfamiliar to many developers, and can be difficult and time-consuming to learn.

XSLT can be a very productive system that yields high-performance applications. For example, some big Web companies use XSLT as a way to generate HTML from XML that has been pulled from a variety of data stores. The managed XSLT engine compiles XSLT to CLR code, and performs even better in some scenarios than the native XSLT engine.

However, XSLT does not take advantage of the C# and Visual Basic knowledge that many developers have. It requires developers to write code in a different and complex programming language. Using two non-integrated development systems such as C# (or Visual Basic) and XSLT results in software systems that are more difficult to develop and maintain.

After you have mastered LINQ to XML query expressions, LINQ to XML transformations are a powerful technology that is easy to use. Basically, you form your XML document by using functional construction, pulling in data from various sources, constructing [XElement](#) objects dynamically, and assembling the whole into a new XML tree. The transformation can generate a completely new document. Constructing transformations in LINQ to XML is relatively easy and intuitive, and the resulting code is readable. This reduces development and maintenance costs.

LINQ to XML is not intended to replace XSLT. XSLT is still the tool of choice for complicated and document-centric XML transformations, especially if the structure of the document is not well defined.

XSLT has the advantage of being a World Wide Web Consortium (W3C) standard. If you have a requirement that you use only technologies that are standards, XSLT might be more appropriate.



XSLT is XML, and therefore can be programmatically manipulated.

## LINQ to XML vs. MSXML

MSXML is the COM-based technology for processing XML that is included with Microsoft Windows. MSXML provides a native implementation of the DOM with support for XPath and XSLT. It also contains the SAX2 non-caching, event-based parser.

MSXML performs well, is secure by default in most scenarios, and can be accessed in Internet Explorer for performing client-side XML processing in AJAX-style applications. MSXML can be used from any programming language that supports COM, including C++, JavaScript, and Visual Basic 6.0.

MSXML is not recommended for use in managed code based on the common language runtime (CLR).

## LINQ to XML vs. XmlLite

XmlLite is a non-caching, forward only, pull parser. Developers primarily use XmlLite with C++. It is not recommended for developers to use XmlLite with managed code.

The main advantage of XmlLite is that it is a lightweight, fast XML parser that is secure in most scenarios. Its threat surface area is very small. If you have to parse untrusted documents and you want to protect against attacks such as denial of service or exposure of data, XmlLite might be a good option.

XmlLite is not integrated with Language-Integrated Query (LINQ). It does not yield the programmer productivity improvements that are the motivating force behind LINQ.

## See also

- [Getting Started \(LINQ to XML\)](#)

# Functional vs. Procedural Programming (LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

There are various types of XML applications:

- Some applications take source XML documents, and produce new XML documents that are in a different shape than the source documents.
- Some applications take source XML documents, and produce result documents in an entirely different form, such as HTML or CSV text files.
- Some applications take source XML documents, and insert records into a database.
- Some applications take data from another source, such as a database, and create XML documents from it.

These are not all of the types of XML applications, but these are a representative set of the types of functionality that an XML programmer has to implement.

With all of these types of applications, there are two contrasting approaches that a developer can take:

- Functional construction using a declarative approach.
- In-memory XML tree modification using procedural code.

LINQ to XML supports both approaches.

When using the functional approach, you write transformations that take the source documents and generate completely new result documents with the desired shape.

When modifying an XML tree in place, you write code that traverses and navigates through nodes in an in-memory XML tree, inserting, deleting, and modifying nodes as necessary.

You can use LINQ to XML with either approach. You use the same classes, and in some cases the same methods. However, the structure and goals of the two approaches are very different. For example, in different situations, one or the other approach will often have better performance, and use more or less memory. In addition, one or the other approach will be easier to write and yield more maintainable code.

To see the two approaches contrasted, see [In-Memory XML Tree Modification vs. Functional Construction \(LINQ to XML\) \(C#\)](#).

For a tutorial on writing functional transformations, see [Pure Functional Transformations of XML \(C#\)](#).

## See also

- [LINQ to XML Programming Overview \(C#\)](#)

# LINQ to XML Classes Overview (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic provides a list of the LINQ to XML classes in the [System.Xml.Linq](#) namespace, and a short description of each.

## LINQ to XML Classes

### XAttribute Class

[XAttribute](#) represents an XML attribute. For detailed information and examples, see [XAttribute Class Overview \(C#\)](#).

### XCDATA Class

[XCDATA](#) represents a CDATA text node.

### XComment Class

[XComment](#) represents an XML comment.

### XContainer Class

[XContainer](#) is an abstract base class for all nodes that can have child nodes. The following classes derive from the [XContainer](#) class:

- [XElement](#)
- [XDocument](#)

### XDeclaration Class

[XDeclaration](#) represents an XML declaration. An XML declaration is used to declare the XML version and the encoding of a document. In addition, an XML declaration specifies whether the XML document is stand-alone. If a document is stand-alone, there are no external markup declarations, either in an external DTD, or in an external parameter entity referenced from the internal subset.

### XDocument Class

[XDocument](#) represents an XML document. For detailed information and examples, see [XDocument Class Overview \(C#\)](#).

### XDocumentType Class

[XDocumentType](#) represents an XML Document Type Definition (DTD).

### XElement Class

[XElement](#) represents an XML element. For detailed information and examples, see [XElement Class Overview \(C#\)](#).

### XName Class

[XName](#) represents names of elements ([XElement](#)) and attributes ([XAttribute](#)). For detailed information and examples, see [XDocument Class Overview \(C#\)](#).

LINQ to XML is designed to make XML names as straightforward as possible. Due to their complexity, XML names are often considered to be an advanced topic in XML. Arguably, this complexity comes not from namespaces, which developers use regularly in programming, but from namespace prefixes. Namespace prefixes can be useful to reduce the keystrokes required when you input XML, or to make XML easier to read. However, prefixes are often just a shortcut for using the full XML namespace, and are not required in most cases. LINQ to XML simplifies XML names by resolving all prefixes to their corresponding XML namespace. Prefixes are available, if they are required,

through the [GetPrefixOfNamespace](#) method.

It is possible, if necessary, to control namespace prefixes. In some circumstances, if you are working with other XML systems, such as XSLT or XAML, you need to control namespace prefixes. For example, if you have an XPath expression that uses namespace prefixes and is embedded in an XSLT stylesheet, you must make sure that your XML document is serialized with namespace prefixes that match those used in the XPath expression.

### **XNamespace Class**

[XNamespace](#) represents a namespace for an [XElement](#) or [XAttribute](#). Namespaces are a component of an [XName](#).

### **XNode Class**

[XNode](#) is an abstract class that represents the nodes of an XML tree. The following classes derive from the [XNode](#) class:

- [XText](#)
- [XContainer](#)
- [XComment](#)
- [XProcessingInstruction](#)
- [XDocumentType](#)

### **XNodeDocumentOrderComparer Class**

[XNodeDocumentOrderComparer](#) provides functionality to compare nodes for their document order.

### **XNodeEqualityComparer Class**

[XNodeEqualityComparer](#) provides functionality to compare nodes for value equality.

### **XObject Class**

[XObject](#) is an abstract base class of [XNode](#) and [XAttribute](#). It provides annotation and event functionality.

### **XObjectChange Class**

[XObjectChange](#) specifies the event type when an event is raised for an [XObject](#).

### **XObjectChangeEventArgs Class**

[XObjectChangeEventArgs](#) provides data for the [Changing](#) and [Changed](#) events.

### **XProcessingInstruction Class**

[XProcessingInstruction](#) represents an XML processing instruction. A processing instruction communicates information to an application that processes the XML.

### **XText Class**

[XText](#) represents a text node. In most cases, you do not have to use this class. This class is primarily used for mixed content.

## **See also**

- [LINQ to XML Programming Overview \(C#\)](#)

# XElement Class Overview (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

The [XElement](#) class is one of the fundamental classes in LINQ to XML. It represents an XML element. You can use this class to create elements; change the content of the element; add, change, or delete child elements; add attributes to an element; or serialize the contents of an element in text form. You can also interoperate with other classes in [System.Xml](#), such as [XmlReader](#), [XmlWriter](#), and [XslCompiledTransform](#).

This topic describes the functionality provided by the [XElement](#) class.

## Constructing XML Trees

You can construct XML trees in a variety of ways, including the following:

- You can construct an XML tree in code. For more information, see [Creating XML Trees \(C#\)](#).
- You can parse XML from various sources, including a [TextReader](#), text files, or a Web address (URL). For more information, see [Parsing XML \(C#\)](#).
- You can use an [XmlReader](#) to populate the tree. For more information, see [ReadFrom](#).
- If you have a module that can write content to an [XmlWriter](#), you can use the [CreateWriter](#) method to create a writer, pass the writer to the module, and then use the content that is written to the [XmlWriter](#) to populate the XML tree.

However, the most common way to create an XML tree is as follows:

```
XElement contacts =  
    new XElement("Contacts",  
        new XElement("Contact",  
            new XElement("Name", "Patrick Hines"),  
            new XElement("Phone", "206-555-0144"),  
            new XElement("Address",  
                new XElement("Street1", "123 Main St"),  
                new XElement("City", "Mercer Island"),  
                new XElement("State", "WA"),  
                new XElement("Postal", "68042")  
            )  
        )  
    );
```

Another very common technique for creating an XML tree involves using the results of a LINQ query to populate an XML tree, as shown in the following example:

```

XElement srcTree = new XElement("Root",
    new XElement("Element", 1),
    new XElement("Element", 2),
    new XElement("Element", 3),
    new XElement("Element", 4),
    new XElement("Element", 5)
);
XElement xmlTree = new XElement("Root",
    new XElement("Child", 1),
    new XElement("Child", 2),
    from el in srcTree.Elements()
    where (int)el > 2
    select el
);
Console.WriteLine(xmlTree);

```

This example produces the following output:

```

<Root>
  <Child>1</Child>
  <Child>2</Child>
  <Element>3</Element>
  <Element>4</Element>
  <Element>5</Element>
</Root>

```

## Serializing XML Trees

You can serialize the XML tree to a [File](#), a [TextWriter](#), or an [XmlWriter](#).

For more information, see [Serializing XML Trees \(C#\)](#).

## Retrieving XML Data via Axis Methods

You can use axis methods to retrieve attributes, child elements, descendant elements, and ancestor elements. LINQ queries operate on axis methods, and provide several flexible and powerful ways to navigate through and process an XML tree.

For more information, see [LINQ to XML Axes \(C#\)](#).

## Querying XML Trees

You can write LINQ queries that extract data from an XML tree.

For more information, see [Querying XML Trees \(C#\)](#).

## Modifying XML Trees

You can modify an element in a variety of ways, including changing its content or attributes. You can also remove an element from its parent.

For more information, see [Modifying XML Trees \(LINQ to XML\) \(C#\)](#).

## See also

- [LINQ to XML Programming Overview \(C#\)](#)

# XAttribute Class Overview (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

Attributes are name/value pairs that are associated with an element. The [XAttribute](#) class represents XML attributes.

## Overview

Working with attributes in LINQ to XML is similar to working with elements. Their constructors are similar. The methods that you use to retrieve collections of them are similar. A LINQ query expression for a collection of attributes looks very similar to a LINQ query expression for a collection of elements.

The order in which attributes were added to an element is preserved. That is, when you iterate through the attributes, you see them in the same order that they were added.

## The XAttribute Constructor

The following constructor of the [XAttribute](#) class is the one that you will most commonly use:

CONSTRUCTOR	DESCRIPTION
<code>XAttribute(XName name, object content)</code>	Creates an <a href="#">XAttribute</a> object. The <code>name</code> argument specifies the name of the attribute; <code>content</code> specifies the content of the attribute.

### Creating an Element with an Attribute

The following code shows the common task of creating an element that contains an attribute:

```
XElement phone = new XElement("Phone",  
    new XAttribute("Type", "Home"),  
    "555-555-5555");  
Console.WriteLine(phone);
```

This example produces the following output:

```
<Phone Type="Home">555-555-5555</Phone>
```

### Functional Construction of Attributes

You can construct [XAttribute](#) objects in-line with the construction of [XElement](#) objects, as follows:

```

XElement c = new XElement("Customers",
    new XElement("Customer",
        new XElement("Name", "John Doe"),
        new XElement("PhoneNumbers",
            new XElement("Phone",
                new XAttribute("type", "home"),
                "555-555-5555"),
            new XElement("Phone",
                new XAttribute("type", "work"),
                "666-666-6666")
        )
    )
);
Console.WriteLine(c);

```

This example produces the following output:

```

<Customers>
  <Customer>
    <Name>John Doe</Name>
    <PhoneNumbers>
      <Phone type="home">555-555-5555</Phone>
      <Phone type="work">666-666-6666</Phone>
    </PhoneNumbers>
  </Customer>
</Customers>

```

### Attributes Are Not Nodes

There are some differences between attributes and elements. [XAttribute](#) objects are not nodes in the XML tree. They are name/value pairs associated with an XML element. In contrast to the Document Object Model (DOM), this more closely reflects the structure of XML. Although [XAttribute](#) objects are not actually nodes in the XML tree, working with [XAttribute](#) objects is very similar to working with [XElement](#) objects.

This distinction is primarily important only to developers who are writing code that works with XML trees at the node level. Many developers will not be concerned with this distinction.

## See also

- [LINQ to XML Programming Overview \(C#\)](#)



# XDocument Class Overview (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic introduces the [XDocument](#) class.

## Overview of the XDocument class

The [XDocument](#) class contains the information necessary for a valid XML document. This includes an XML declaration, processing instructions, and comments.

Note that you only have to create [XDocument](#) objects if you require the specific functionality provided by the [XDocument](#) class. In many circumstances, you can work directly with [XElement](#). Working directly with [XElement](#) is a simpler programming model.

[XDocument](#) derives from [XContainer](#). Therefore, it can contain child nodes. However, [XDocument](#) objects can have only one child [XElement](#) node. This reflects the XML standard that there can be only one root element in an XML document.

## Components of XDocument

An [XDocument](#) can contain the following elements:

- One [XDeclaration](#) object. [XDeclaration](#) enables you to specify the pertinent parts of an XML declaration: the XML version, the encoding of the document, and whether the XML document is stand-alone.
- One [XElement](#) object. This is the root node of the XML document.
- Any number of [XProcessingInstruction](#) objects. A processing instruction communicates information to an application that processes the XML.
- Any number of [XComment](#) objects. The comments will be siblings to the root element. The [XComment](#) object cannot be the first argument in the list, because it is not valid for an XML document to start with a comment.
- One [XDocumentType](#) for the DTD.

When you serialize an [XDocument](#), even if `xDocument.Declaration` is `null`, the output will have an XML declaration if the writer has `Writer.Settings.OmitXmlDeclaration` set to `false` (the default).

By default, LINQ to XML sets the version to "1.0", and sets the encoding to "utf-8".

## Using XElement without XDocument

As previously mentioned, the [XElement](#) class is the main class in the LINQ to XML programming interface. In many cases, your application will not require that you create a document. By using the [XElement](#) class, you can create an XML tree, add other XML trees to it, modify the XML tree, and save it.

## Using XDocument

To construct an [XDocument](#), use functional construction, just like you do to construct [XElement](#) objects.

The following code creates an [XDocument](#) object and its associated contained objects.

```

XDocument d = new XDocument(
    new XComment("This is a comment."),
    new XProcessingInstruction("xml-stylesheet",
        "href='mystyle.css' title='Compact' type='text/css'"),
    new XElement("Pubs",
        new XElement("Book",
            new XElement("Title", "Artifacts of Roman Civilization"),
            new XElement("Author", "Moreno, Jordao")
        ),
        new XElement("Book",
            new XElement("Title", "Midieval Tools and Implements"),
            new XElement("Author", "Gazit, Inbar")
        )
    ),
    new XComment("This is another comment.")
);
d.Declaration = new XDeclaration("1.0", "utf-8", "true");
Console.WriteLine(d);

d.Save("test.xml");

```

When you examine the file test.xml, you get the following output:

```

<?xml version="1.0" encoding="utf-8"?>
<!--This is a comment.-->
<?xml-stylesheet href='mystyle.css' title='Compact' type='text/css'?>
<Pubs>
  <Book>
    <Title>Artifacts of Roman Civilization</Title>
    <Author>Moreno, Jordao</Author>
  </Book>
  <Book>
    <Title>Midieval Tools and Implements</Title>
    <Author>Gazit, Inbar</Author>
  </Book>
</Pubs>
<!--This is another comment.-->

```

## See also

- [LINQ to XML Programming Overview \(C#\)](#)

# How to: Build LINQ to XML Examples (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

The various snippets and examples in this documentation use classes and types from a variety of namespaces. When compiling C# code, you need to supply appropriate `using` directives.

## Example

The following code contains the `using` directives that the C# examples require to build and run. Not all `using` directives are required for every example.

```
using System;
using System.Diagnostics;
using System.Collections;
using System.Collections.Generic;
using System.Collections.Specialized;
using System.Text;
using System.Linq;
using System.Xml;
using System.Xml.Linq;
using System.Xml.Schema;
using System.Xml.XPath;
using System.Xml.Xsl;
using System.IO;
using System.Threading;
using System.Reflection;
using System.IO.Packaging;
```

## See also

- [LINQ to XML Programming Overview \(C#\)](#)

# Functional Construction (LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

LINQ to XML provides a powerful way to create XML elements called *functional construction*. Functional construction is the ability to create an XML tree in a single statement.

There are several key features of the LINQ to XML programming interface that enable functional construction:

- The `XElement` constructor takes various types of arguments for content. For example, you can pass another `XElement` object, which becomes a child element. You can pass an `XAttribute` object, which becomes an attribute of the element. Or you can pass any other type of object, which is converted to a string and becomes the text content of the element.
- The `XElement` constructor takes a `params` array of type `Object`, so that you can pass any number of objects to the constructor. This enables you to create an element that has complex content.
- If an object implements `IEnumerable<T>`, the collection in the object is enumerated, and all items in the collection are added. If the collection contains `XElement` or `XAttribute` objects, each item in the collection is added separately. This is important because it lets you pass the results of a LINQ query to the constructor.

These features enable you to write code to create an XML tree. The following is an example:

```
XElement contacts =
    new XElement("Contacts",
        new XElement("Contact",
            new XElement("Name", "Patrick Hines"),
            new XElement("Phone", "206-555-0144"),
            new XElement("Address",
                new XElement("Street1", "123 Main St"),
                new XElement("City", "Mercer Island"),
                new XElement("State", "WA"),
                new XElement("Postal", "68042")
            )
        )
    );
```

These features also enable you to write code that uses the results of LINQ queries when you create an XML tree, as follows:

```
XElement srcTree = new XElement("Root",
    new XElement("Element", 1),
    new XElement("Element", 2),
    new XElement("Element", 3),
    new XElement("Element", 4),
    new XElement("Element", 5)
);
XElement xmlTree = new XElement("Root",
    new XElement("Child", 1),
    new XElement("Child", 2),
    from el in srcTree.Elements()
    where (int)el > 2
    select el
);
Console.WriteLine(xmlTree);
```

This example produces the following output:

```
<Root>  
  <Child>1</Child>  
  <Child>2</Child>  
  <Element>3</Element>  
  <Element>4</Element>  
  <Element>5</Element>  
</Root>
```

# Creating XML trees in C# (LINQ to XML)

6/4/2019 • 4 minutes to read • [Edit Online](#)

This section provides information about creating XML trees in C#.

For information about using the results of LINQ queries as the content for an [XElement](#), see [Functional Construction \(LINQ to XML\) \(C#\)](#).

## Constructing elements

The signatures of the [XElement](#) and [XAttribute](#) constructors let you pass the contents of the element or attribute as arguments to the constructor. Because one of the constructors takes a variable number of arguments, you can pass any number of child elements. Of course, each of those child elements can contain their own child elements. For any element, you can add any number of attributes.

When adding [XNode](#) (including [XElement](#)) or [XAttribute](#) objects, if the new content has no parent, the objects are simply attached to the XML tree. If the new content already is parented, and is part of another XML tree, the new content is cloned, and the newly cloned content is attached to the XML tree. The last example in this topic demonstrates this.

To create a `contacts` [XElement](#), you could use the following code:

```
XElement contacts =
    new XElement("Contacts",
        new XElement("Contact",
            new XElement("Name", "Patrick Hines"),
            new XElement("Phone", "206-555-0144"),
            new XElement("Address",
                new XElement("Street1", "123 Main St"),
                new XElement("City", "Mercer Island"),
                new XElement("State", "WA"),
                new XElement("Postal", "68042")
            )
        )
    );
```

If indented properly, the code to construct [XElement](#) objects closely resembles the structure of the underlying XML.

## XElement constructors

The [XElement](#) class uses the following constructors for functional construction. Note that there are some other constructors for [XElement](#), but because they are not used for functional construction they are not listed here.

CONSTRUCTOR	DESCRIPTION
<code>XElement(XName name, object content)</code>	Creates an <a href="#">XElement</a> . The <code>name</code> parameter specifies the name of the element; <code>content</code> specifies the content of the element.
<code>XElement(XName name)</code>	Creates an <a href="#">XElement</a> with its <a href="#">XName</a> initialized to the specified name.

CONSTRUCTOR	DESCRIPTION
<code>XElement(XName name, params object[] content)</code>	Creates an <a href="#">XElement</a> with its <a href="#">XName</a> initialized to the specified name. The attributes and/or child elements are created from the contents of the parameter list.

The `content` parameter is extremely flexible. It supports any type of object that is a valid child of an [XElement](#). The following rules apply to different types of objects passed in this parameter:

- A string is added as text content.
- An [XElement](#) is added as a child element.
- An [XAttribute](#) is added as an attribute.
- An [XProcessingInstruction](#), [XComment](#), or [XText](#) is added as child content.
- An [IEnumerable](#) is enumerated, and these rules are applied recursively to the results.
- For any other type, its `ToString` method is called and the result is added as text content.

### Creating an XElement with content

You can create an [XElement](#) that contains simple content with a single method call. To do this, specify the content as the second parameter, as follows:

```
XElement n = new XElement("Customer", "Adventure Works");
Console.WriteLine(n);
```

This example produces the following output:

```
<Customer>Adventure Works</Customer>
```

You can pass any type of object as the content. For example, the following code creates an element that contains a floating point number as content:

```
XElement n = new XElement("Cost", 324.50);
Console.WriteLine(n);
```

This example produces the following output:

```
<Cost>324.5</Cost>
```

The floating point number is boxed and passed in to the constructor. The boxed number is converted to a string and used as the content of the element.

### Creating an XElement with a child element

If you pass an instance of the [XElement](#) class for the content argument, the constructor creates an element with a child element:

```
XElement shippingUnit = new XElement("ShippingUnit",
    new XElement("Cost", 324.50)
);
Console.WriteLine(shippingUnit);
```

This example produces the following output:

```
<ShippingUnit>
  <Cost>324.5</Cost>
</ShippingUnit>
```

### Creating an XElement with multiple child elements

You can pass in a number of [XElement](#) objects for the content. Each of the [XElement](#) objects is included as a child element.

```
XElement address = new XElement("Address",
    new XElement("Street1", "123 Main St"),
    new XElement("City", "Mercer Island"),
    new XElement("State", "WA"),
    new XElement("Postal", "68042")
);
Console.WriteLine(address);
```

This example produces the following output:

```
<Address>
  <Street1>123 Main St</Street1>
  <City>Mercer Island</City>
  <State>WA</State>
  <Postal>68042</Postal>
</Address>
```

By extending the above example, you can create an entire XML tree, as follows:

```
XElement contacts =
    new XElement("Contacts",
        new XElement("Contact",
            new XElement("Name", "Patrick Hines"),
            new XElement("Phone", "206-555-0144"),
            new XElement("Address",
                new XElement("Street1", "123 Main St"),
                new XElement("City", "Mercer Island"),
                new XElement("State", "WA"),
                new XElement("Postal", "68042")
            )
        )
    );
Console.WriteLine(contacts);
```

This example produces the following output:

```
<Contacts>
  <Contact>
    <Name>Patrick Hines</Name>
    <Phone>206-555-0144</Phone>
    <Address>
      <Street1>123 Main St</Street1>
      <City>Mercer Island</City>
      <State>WA</State>
      <Postal>68042</Postal>
    </Address>
  </Contact>
</Contacts>
```



## Creating an XElement with an XAttribute

If you pass an instance of the [XAttribute](#) class for the content argument, the constructor creates an element with an attribute:

```
XElement phone = new XElement("Phone",  
    new XAttribute("Type", "Home"),  
    "555-555-5555");  
Console.WriteLine(phone);
```

This example produces the following output:

```
<Phone Type="Home">555-555-5555</Phone>
```

## Creating an empty element

To create an empty [XElement](#), you do not pass any content to the constructor. The following example creates an empty element:

```
XElement n = new XElement("Customer");  
Console.WriteLine(n);
```

This example produces the following output:

```
<Customer />
```

## Attaching vs. cloning

As mentioned previously, when adding [XNode](#) (including [XElement](#)) or [XAttribute](#) objects, if the new content has no parent, the objects are simply attached to the XML tree. If the new content already is parented and is part of another XML tree, the new content is cloned, and the newly cloned content is attached to the XML tree.

The following example demonstrates the behavior when you add a parented element to a tree, and when you add an element with no parent to a tree.

```

// Create a tree with a child element.
XElement xmlTree1 = new XElement("Root",
    new XElement("Child1", 1)
);

// Create an element that is not parented.
XElement child2 = new XElement("Child2", 2);

// Create a tree and add Child1 and Child2 to it.
XElement xmlTree2 = new XElement("Root",
    xmlTree1.Element("Child1"),
    child2
);

// Compare Child1 identity.
Console.WriteLine("Child1 was {0}",
    xmlTree1.Element("Child1") == xmlTree2.Element("Child1") ?
    "attached" : "cloned");

// Compare Child2 identity.
Console.WriteLine("Child2 was {0}",
    child2 == xmlTree2.Element("Child2") ?
    "attached" : "cloned");

// The example displays the following output:
//   Child1 was cloned
//   Child2 was attached

```

## See also

- [Creating XML Trees \(C#\)](#)

# How to: Parse a String (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how to parse a string to create an XML tree in C#.

## Example

The following C# code shows how to parse a string.

```
XElement contacts = XElement.Parse(
    @"<Contacts>
      <Contact>
        <Name>Patrick Hines</Name>
        <Phone Type=""home"">206-555-0144</Phone>
        <Phone type=""work"">425-555-0145</Phone>
        <Address>
          <Street1>123 Main St</Street1>
          <City>Mercer Island</City>
          <State>WA</State>
          <Postal>68042</Postal>
        </Address>
        <NetWorth>10</NetWorth>
      </Contact>
      <Contact>
        <Name>Gretchen Rivas</Name>
        <Phone Type=""mobile"">206-555-0163</Phone>
        <Address>
          <Street1>123 Main St</Street1>
          <City>Mercer Island</City>
          <State>WA</State>
          <Postal>68042</Postal>
        </Address>
        <NetWorth>11</NetWorth>
      </Contact>
    </Contacts>");
Console.WriteLine(contacts);
```

# How to: Load XML from a File (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how to load XML from a URI by using the [XElement.Load](#) method.

## Example

The following example shows how to load an XML document from a file. The following example loads books.xml and outputs the XML tree to the console.

This example uses the following XML document: [Sample XML File: Books \(LINQ to XML\)](#).

```
XElement booksFromFile = XElement.Load(@"books.xml");
Console.WriteLine(booksFromFile);
```

This code produces the following output:

```
<Catalog>
  <Book id="bk101">
    <Author>Garghentini, Davide</Author>
    <Title>XML Developer's Guide</Title>
    <Genre>Computer</Genre>
    <Price>44.95</Price>
    <PublishDate>2000-10-01</PublishDate>
    <Description>An in-depth look at creating applications
      with XML.</Description>
  </Book>
  <Book id="bk102">
    <Author>Garcia, Debra</Author>
    <Title>Midnight Rain</Title>
    <Genre>Fantasy</Genre>
    <Price>5.95</Price>
    <PublishDate>2000-12-16</PublishDate>
    <Description>A former architect battles corporate zombies,
      an evil sorceress, and her own childhood to become queen
      of the world.</Description>
  </Book>
</Catalog>
```

# Preserving White Space while Loading or Parsing XML

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic describes how to control the white-space behavior of LINQ to XML.

A common scenario is to read indented XML, create an in-memory XML tree without any white space text nodes (that is, not preserving white space), perform some operations on the XML, and then save the XML with indentation. When you serialize the XML with formatting, only significant white space in the XML tree is preserved. This is the default behavior for LINQ to XML.

Another common scenario is to read and modify XML that has already been intentionally indented. You might not want to change this indentation in any way. To do this in LINQ to XML, you preserve white space when you load or parse the XML and disable formatting when you serialize the XML.

This topic describes the white-space behavior of methods that populate XML trees. For information about controlling white space when you serialize XML trees, see [Preserving White Space While Serializing](#).

## Behavior of Methods that Populate XML Trees

The following methods in the [XElement](#) and [XDocument](#) classes populate an XML tree. You can populate an XML tree from a file, a [TextReader](#), an [XmlReader](#), or a string:

- [XElement.Load](#)
- [XElement.Parse](#)
- [XDocument.Load](#)
- [XDocument.Parse](#)

If the method does not take [LoadOptions](#) as an argument, the method will not preserve insignificant white space.

In most cases, if the method takes [LoadOptions](#) as an argument, you can optionally preserve insignificant white space as text nodes in the XML tree. However, if the method is loading the XML from an [XmlReader](#), then the [XmlReader](#) determines whether white space will be preserved or not. Setting [PreserveWhitespace](#) will have no effect.

With these methods, if white space is preserved, insignificant white space is inserted into the XML tree as [XText](#) nodes. If white space is not preserved, text nodes are not inserted.

You can create an XML tree by using an [XmlWriter](#). Nodes that are written to the [XmlWriter](#) are populated in the tree. However, when you build an XML tree using this method, all nodes are preserved, regardless of whether the node is white space or not, or whether the white space is significant or not.

# How to: Catch Parsing Errors (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how to detect badly formed or invalid XML.

LINQ to XML is implemented using [XmlReader](#). If badly formed or invalid XML is passed to LINQ to XML, the underlying [XmlReader](#) class will throw an exception. The various methods that parse XML, such as [XElement.Parse](#), do not catch the exception; the exception can then be caught by your application.

## Example

The following code tries to parse invalid XML:

```
try {
    XElement contacts = XElement.Parse(
        @"<Contacts>
        <Contact>
            <Name>Jim Wilson</Name>
        </Contact>
        </Contcts>");

    Console.WriteLine(contacts);
}
catch (System.Xml.XmlException e)
{
    Console.WriteLine(e.Message);
}
```

When you run this code, it throws the following exception:

```
The 'Contacts' start tag on line 1 does not match the end tag of 'Contcts'. Line 5, position 13.
```

For information about the exceptions that you can expect the [XElement.Parse](#), [XDocument.Parse](#), [XElement.Load](#), and [XDocument.Load](#) methods to throw, see the [XmlReader](#) documentation.

# How to: Create a Tree from an XmlReader (C#)

1/23/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how to create an XML tree directly from an [XmlReader](#). To create an [XElement](#) from an [XmlReader](#), you must position the [XmlReader](#) on an element node. The [XmlReader](#) will skip comments and processing instructions, but if the [XmlReader](#) is positioned on a text node, an error will be thrown. To avoid such errors, always position the [XmlReader](#) on an element before you create an XML tree from the [XmlReader](#).

## Example

This example uses the following XML document: [Sample XML File: Books \(LINQ to XML\)](#).

The following code creates an `T:System.Xml.XmlReader` object, and then reads nodes until it finds the first element node. It then loads the [XElement](#) object.

```
XmlReader r = XmlReader.Create("books.xml");
while (r.NodeType != XmlNodeType.Element)
    r.Read();
XElement e = XElement.Load(r);
Console.WriteLine(e);
```

This example produces the following output:

```
<Catalog>
  <Book id="bk101">
    <Author>Garghentini, Davide</Author>
    <Title>XML Developer's Guide</Title>
    <Genre>Computer</Genre>
    <Price>44.95</Price>
    <PublishDate>2000-10-01</PublishDate>
    <Description>An in-depth look at creating applications
    with XML.</Description>
  </Book>
  <Book id="bk102">
    <Author>Garcia, Debra</Author>
    <Title>Midnight Rain</Title>
    <Genre>Fantasy</Genre>
    <Price>5.95</Price>
    <PublishDate>2000-12-16</PublishDate>
    <Description>A former architect battles corporate zombies,
    an evil sorceress, and her own childhood to become queen
    of the world.</Description>
  </Book>
</Catalog>
```

## See also

- [Parsing XML \(C#\)](#)

# How to: Stream XML Fragments from an XmlReader (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

When you have to process large XML files, it might not be feasible to load the whole XML tree into memory. This topic shows how to stream fragments using an [XmlReader](#).

One of the most effective ways to use an [XmlReader](#) to read [XElement](#) objects is to write your own custom axis method. An axis method typically returns a collection such as [IEnumerable<T>](#) of [XElement](#), as shown in the example in this topic. In the custom axis method, after you create the XML fragment by calling the [ReadFrom](#) method, return the collection using `yield return`. This provides deferred execution semantics to your custom axis method.

When you create an XML tree from an [XmlReader](#) object, the [XmlReader](#) must be positioned on an element. The [ReadFrom](#) method does not return until it has read the close tag of the element.

If you want to create a partial tree, you can instantiate an [XmlReader](#), position the reader on the node that you want to convert to an [XElement](#) tree, and then create the [XElement](#) object.

The topic [How to: Stream XML Fragments with Access to Header Information \(C#\)](#) contains information and an example on how to stream a more complex document.

The topic [How to: Perform Streaming Transform of Large XML Documents \(C#\)](#) contains an example of using LINQ to XML to transform extremely large XML documents while maintaining a small memory footprint.

## Example

This example creates a custom axis method. You can query it by using a LINQ query. The custom axis method, `StreamRootChildDoc`, is a method that is designed specifically to read a document that has a repeating `Child` element.



```

static IEnumerable<XElement> StreamRootChildDoc(StringReader stringReader)
{
    using (XmlReader reader = XmlReader.Create(stringReader))
    {
        reader.MoveToContent();
        // Parse the file and display each of the nodes.
        while (reader.Read())
        {
            switch (reader.NodeType)
            {
                case XmlNodeType.Element:
                    if (reader.Name == "Child") {
                        XElement el = XElement.ReadFrom(reader) as XElement;
                        if (el != null)
                            yield return el;
                    }
                    break;
            }
        }
    }
}

static void Main(string[] args)
{
    string markup = @"<Root>
    <Child Key=""01"">
        <GrandChild>aaa</GrandChild>
    </Child>
    <Child Key=""02"">
        <GrandChild>bbb</GrandChild>
    </Child>
    <Child Key=""03"">
        <GrandChild>ccc</GrandChild>
    </Child>
</Root>";

    IEnumerable<string> grandChildData =
        from el in StreamRootChildDoc(new StringReader(markup))
        where (int)el.Attribute("Key") > 1
        select (string)el.Element("GrandChild");

    foreach (string str in grandChildData) {
        Console.WriteLine(str);
    }
}

```

This example produces the following output:

```

bbb
ccc

```

In this example, the source document is very small. However, even if there were millions of `Child` elements, this example would still have a small memory footprint.

# How to: Populate an XML Tree with an XmlWriter (LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

One way to populate an XML tree is to use [CreateWriter](#) to create an [XmlWriter](#), and then write to the [XmlWriter](#). The XML tree is populated with all nodes that are written to the [XmlWriter](#).

You would typically use this method when you use LINQ to XML with another class that expects to write to an [XmlWriter](#), such as [XslCompiledTransform](#).

## Example

One possible use for [CreateWriter](#) is when invoking an XSLT transformation. This example creates an XML tree, creates an [XmlReader](#) from the XML tree, creates a new document, and then creates an [XmlWriter](#) to write into the new document. It then invokes the XSLT transformation, passing in [XmlReader](#) and [XmlWriter](#). After the transformation successfully completes, the new XML tree is populated with the results of the transformation.

```
string xslMarkup = @"<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl='http://www.w3.org/1999/XSL/Transform' version='1.0'>
  <xsl:template match='/Parent'>
    <Root>
      <C1>
        <xsl:value-of select='Child1' />
      </C1>
      <C2>
        <xsl:value-of select='Child2' />
      </C2>
    </Root>
  </xsl:template>
</xsl:stylesheet>";

XDocument xmlTree = new XDocument(
    new XElement("Parent",
        new XElement("Child1", "Child1 data"),
        new XElement("Child2", "Child2 data")
    )
);

XDocument newTree = new XDocument();
using (XmlWriter writer = newTree.CreateWriter())
{
    // Load the style sheet.
    XslCompiledTransform xslt = new XslCompiledTransform();
    xslt.Load(XmlReader.Create(new StringReader(xslMarkup)));

    // Execute the transformation and output the results to a writer.
    xslt.Transform(xmlTree.CreateReader(), writer);
}

Console.WriteLine(newTree);
```

This example produces the following output:

```
<Root>  
  <C1>Child1 data</C1>  
  <C2>Child2 data</C2>  
</Root>
```

## See also

- [CreateWriter](#)
- [XmlWriter](#)
- [XslCompiledTransform](#)
- [Creating XML Trees \(C#\)](#)

# How to: Validate Using XSD (LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

The [System.Xml.Schema](#) namespace contains extension methods that make it easy to validate an XML tree against an XML Schema Definition Language (XSD) file. For more information, see the [Validate](#) method documentation.

## Example

The following example creates an [XmlSchemaSet](#), then validates two [XDocument](#) objects against the schema set. One of the documents is valid, the other is not.

```
string xsdMarkup =
    @"<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
      <xsd:element name='Root'>
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name='Child1' minOccurs='1' maxOccurs='1'/>
            <xsd:element name='Child2' minOccurs='1' maxOccurs='1'/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>";
XmlSchemaSet schemas = new XmlSchemaSet();
schemas.Add("", XmlReader.Create(new StringReader(xsdMarkup)));

XDocument doc1 = new XDocument(
    new XElement("Root",
        new XElement("Child1", "content1"),
        new XElement("Child2", "content1")
    )
);

XDocument doc2 = new XDocument(
    new XElement("Root",
        new XElement("Child1", "content1"),
        new XElement("Child3", "content1")
    )
);

Console.WriteLine("Validating doc1");
bool errors = false;
doc1.Validate(schemas, (o, e) =>
{
    Console.WriteLine("{0}", e.Message);
    errors = true;
});
Console.WriteLine("doc1 {0}", errors ? "did not validate" : "validated");

Console.WriteLine();
Console.WriteLine("Validating doc2");
errors = false;
doc2.Validate(schemas, (o, e) =>
{
    Console.WriteLine("{0}", e.Message);
    errors = true;
});
Console.WriteLine("doc2 {0}", errors ? "did not validate" : "validated");
```

This example produces the following output:

```
Validating doc1
doc1 validated

Validating doc2
The element 'Root' has invalid child element 'Child3'. List of possible elements expected: 'Child2'.
doc2 did not validate
```

## Example

The following example validates that the XML document from [Sample XML File: Customers and Orders \(LINQ to XML\)](#) is valid per the schema from [Sample XSD File: Customers and Orders](#). It then modifies the source XML document. It changes the `CustomerID` attribute on the first customer. After the change, orders will then refer to a customer that does not exist, so the XML document will no longer validate.

This example uses the following XML document: [Sample XML File: Customers and Orders \(LINQ to XML\)](#).

This example uses the following XSD schema: [Sample XSD File: Customers and Orders](#).

```
XmlSchemaSet schemas = new XmlSchemaSet();
schemas.Add("", "CustomersOrders.xsd");

Console.WriteLine("Attempting to validate");
XDocument custOrdDoc = XDocument.Load("CustomersOrders.xml");
bool errors = false;
custOrdDoc.Validate(schemas, (o, e) =>
{
    Console.WriteLine("{0}", e.Message);
    errors = true;
});
Console.WriteLine("custOrdDoc {0}", errors ? "did not validate" : "validated");

Console.WriteLine();
// Modify the source document so that it will not validate.
custOrdDoc.Root.Element("Orders").Element("Order").Element("CustomerID").Value = "AAAAA";
Console.WriteLine("Attempting to validate after modification");
errors = false;
custOrdDoc.Validate(schemas, (o, e) =>
{
    Console.WriteLine("{0}", e.Message);
    errors = true;
});
Console.WriteLine("custOrdDoc {0}", errors ? "did not validate" : "validated");
```

This example produces the following output:

```
Attempting to validate
custOrdDoc validated

Attempting to validate after modification
The key sequence 'AAAAA' in Keyref fails to refer to some key.
custOrdDoc did not validate
```

## See also

- [Validate](#)
- [Creating XML Trees \(C#\)](#)

# Valid Content of XElement and XDocument Objects

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic describes the valid arguments that can be passed to constructors and methods that you use to add content to elements and documents.

## Valid Content

Queries often evaluate to [IEnumerable<T>](#) of [XElement](#) or [IEnumerable<T>](#) of [XAttribute](#). You can pass collections of [XElement](#) or [XAttribute](#) objects to the [XElement](#) constructor. Therefore, it is convenient to pass the results of a query as content into methods and constructors that you use to populate XML trees.

When adding simple content, various types can be passed to this method. Valid types include the following:

- [String](#)
- [Double](#)
- [Single](#)
- [Decimal](#)
- [Boolean](#)
- [DateTime](#)
- [TimeSpan](#)
- [DateTimeOffset](#)
- Any type that implements `Object.ToString`.
- Any type that implements [IEnumerable<T>](#).

When adding complex content, various types can be passed to this method:

- [XObject](#)
- [XNode](#)
- [XAttribute](#)
- Any type that implements [IEnumerable<T>](#)

If an object implements [IEnumerable<T>](#), the collection in the object is enumerated, and all items in the collection are added. If the collection contains [XNode](#) or [XAttribute](#) objects, each item in the collection is added separately. If the collection contains text (or objects that are converted to text), the text in the collection is concatenated and added as a single text node.

If content is `null`, nothing is added. When passing a collection items in the collection can be `null`. A `null` item in the collection has no effect on the tree.

An added attribute must have a unique name within its containing element.

When adding [XNode](#) or [XAttribute](#) objects, if the new content has no parent, then the objects are simply attached to the XML tree. If the new content already is parented and is part of another XML tree, then the new content is cloned, and the newly cloned content is attached to the XML tree.

# Valid Content for Documents

Attributes and simple content cannot be added to a document.

There are not many scenarios that require you to create an [XDocument](#). Instead, you can usually create your XML trees with an [XElement](#) root node. Unless you have a specific requirement to create a document (for example, because you have to create processing instructions and comments at the top level, or you have to support document types), it is often more convenient to use [XElement](#) as your root node.

Valid content for a document includes the following:

- Zero or one [XDocumentType](#) objects. The document types must come before the element.
- Zero or one element.
- Zero or more comments.
- Zero or more processing instructions.
- Zero or more text nodes that contain only white space.

## Constructors and Functions that Allow Adding Content

The following methods allow you to add child content to an [XElement](#) or an [XDocument](#):

METHOD	DESCRIPTION
<a href="#">XElement</a>	Constructs an <a href="#">XElement</a> .
<a href="#">XDocument</a>	Constructs a <a href="#">XDocument</a> .
<a href="#">Add</a>	Adds to the end of the child content of the <a href="#">XElement</a> or <a href="#">XDocument</a> .
<a href="#">AddAfterSelf</a>	Adds content after the <a href="#">XNode</a> .
<a href="#">AddBeforeSelf</a>	Adds content before the <a href="#">XNode</a> .
<a href="#">AddFirst</a>	Adds content at the beginning of the child content of the <a href="#">XContainer</a> .
<a href="#">ReplaceAll</a>	Replaces all content (child nodes and attributes) of an <a href="#">XElement</a> .
<a href="#">ReplaceAttributes</a>	Replaces the attributes of an <a href="#">XElement</a> .
<a href="#">ReplaceNodes</a>	Replaces the children nodes with new content.
<a href="#">ReplaceWith</a>	Replaces a node with new content.

## See also

- [Creating XML Trees \(C#\)](#)

# Namespaces Overview (LINQ to XML)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic introduces namespaces, the [XName](#) class, and the [XNamespace](#) class.

## XML Names

XML names are often a source of complexity in XML programming. An XML name consists of an XML namespace (also called an XML namespace URI) and a local name. An XML namespace is similar to a namespace in a .NET Framework-based program. It enables you to uniquely qualify the names of elements and attributes. This helps avoid name conflicts between various parts of an XML document. When you have declared an XML namespace, you can select a local name that only has to be unique within that namespace.

Another aspect of XML names is XML *namespace prefixes*. XML prefixes cause most of the complexity of XML names. These prefixes enable you to create a shortcut for an XML namespace, which makes the XML document more concise and understandable. However, XML prefixes depend on their context to have meaning, which adds complexity. For example, the XML prefix `aw` could be associated with one XML namespace in one part of an XML tree, and with a different XML namespace in a different part of the XML tree.

One of the advantages of using LINQ to XML with C# is that you do not have to use XML prefixes. When LINQ to XML loads or parses an XML document, each XML prefix is resolved to its corresponding XML namespace. After that, when you work with a document that uses namespaces, you almost always access the namespaces through the namespace URI, and not through the namespace prefix. When developers work with XML names in LINQ to XML they always work with a fully-qualified XML name (that is, an XML namespace and a local name). However, when necessary, LINQ to XML allows you to work with and control namespace prefixes.

In LINQ to XML, the class that represents XML names is [XName](#). XML names appear frequently throughout the LINQ to XML API, and wherever an XML name is required, you will find an [XName](#) parameter. However, you rarely work directly with an [XName](#). [XName](#) contains an implicit conversion from string.

For more information, see [XNamespace](#) and [XName](#).

## See also

- [Working with XML Namespaces \(C#\)](#)



# How to: Create a Document with Namespaces (C#) (LINQ to XML)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how to create documents with namespaces.

## Example

To create an element or an attribute that is in a namespace, you first declare and initialize an [XNamespace](#) object. You then use the addition operator overload to combine the namespace with the local name, expressed as a string.

The following example creates a document with one namespace. By default, LINQ to XML serializes this document with a default namespace.

```
// Create an XML tree in a namespace.
XNamespace aw = "http://www.adventure-works.com";
XElement root = new XElement(aw + "Root",
    new XElement(aw + "Child", "child content")
);
Console.WriteLine(root);
```

This example produces the following output:

```
<Root xmlns="http://www.adventure-works.com">
  <Child>child content</Child>
</Root>
```

## Example

The following example creates a document with one namespace. It also creates an attribute that declares the namespace with a namespace prefix. To create an attribute that declares a namespace with a prefix, you create an attribute where the name of the attribute is the namespace prefix, and this name is in the [Xmlns](#) namespace. The value of this attribute is the URI of the namespace.

```
// Create an XML tree in a namespace, with a specified prefix
XNamespace aw = "http://www.adventure-works.com";
XElement root = new XElement(aw + "Root",
    new XAttribute(XNamespace.Xmlns + "aw", "http://www.adventure-works.com"),
    new XElement(aw + "Child", "child content")
);
Console.WriteLine(root);
```

This example produces the following output:

```
<aw:Root xmlns:aw="http://www.adventure-works.com">
  <aw:Child>child content</aw:Child>
</aw:Root>
```

## Example

The following example shows the creation of a document that contains two namespaces. One is the default namespace. Another is a namespace with a prefix.

By including namespace attributes in the root element, the namespaces are serialized so that

`http://www.adventure-works.com` is the default namespace, and `www.fourthcoffee.com` is serialized with a prefix of "fc". To create an attribute that declares a default namespace, you create an attribute with the name "xmlns", without a namespace. The value of the attribute is the default namespace URI.

```
// The http://www.adventure-works.com namespace is forced to be the default namespace.
XNamespace aw = "http://www.adventure-works.com";
XNamespace fc = "www.fourthcoffee.com";
XElement root = new XElement(aw + "Root",
    new XAttribute("xmlns", "http://www.adventure-works.com"),
    new XAttribute(XNamespace.Xmlns + "fc", "www.fourthcoffee.com"),
    new XElement(fc + "Child",
        new XElement(aw + "DifferentChild", "other content")
    ),
    new XElement(aw + "Child2", "c2 content"),
    new XElement(fc + "Child3", "c3 content")
);
Console.WriteLine(root);
```

This example produces the following output:

```
<Root xmlns="http://www.adventure-works.com" xmlns:fc="www.fourthcoffee.com">
  <fc:Child>
    <DifferentChild>other content</DifferentChild>
  </fc:Child>
  <Child2>c2 content</Child2>
  <fc:Child3>c3 content</fc:Child3>
</Root>
```

## Example

The following example creates a document that contains two namespaces, both with namespace prefixes.

```
XNamespace aw = "http://www.adventure-works.com";
XNamespace fc = "www.fourthcoffee.com";
XElement root = new XElement(aw + "Root",
    new XAttribute(XNamespace.Xmlns + "aw", aw.NamespaceName),
    new XAttribute(XNamespace.Xmlns + "fc", fc.NamespaceName),
    new XElement(fc + "Child",
        new XElement(aw + "DifferentChild", "other content")
    ),
    new XElement(aw + "Child2", "c2 content"),
    new XElement(fc + "Child3", "c3 content")
);
Console.WriteLine(root);
```

This example produces the following output:

```
<aw:Root xmlns:aw="http://www.adventure-works.com" xmlns:fc="www.fourthcoffee.com">
  <fc:Child>
    <aw:DifferentChild>other content</aw:DifferentChild>
  </fc:Child>
  <aw:Child2>c2 content</aw:Child2>
  <fc:Child3>c3 content</fc:Child3>
</aw:Root>
```

## Example

Another way to accomplish the same result is to use expanded names instead of declaring and creating an [XNamespace](#) object.

This approach has performance implications. Each time you pass a string that contains an expanded name to LINQ to XML, LINQ to XML must parse the name, find the atomized namespace, and find the atomized name. This process takes CPU time. If performance is important, you might want to declare and use an [XNamespace](#) object explicitly.

If performance is an important issue, see [Pre-Atomization of XName Objects \(LINQ to XML\) \(C#\)](#) for more information

```
// Create an XML tree in a namespace, with a specified prefix
XElement root = new XElement("{http://www.adventure-works.com}Root",
    new XAttribute(XNamespace.Xmlns + "aw", "http://www.adventure-works.com"),
    new XElement("{http://www.adventure-works.com}Child", "child content")
);
Console.WriteLine(root);
```

This example produces the following output:

```
<aw:Root xmlns:aw="http://www.adventure-works.com">
  <aw:Child>child content</aw:Child>
</aw:Root>
```

## See also

- [Working with XML Namespaces \(C#\)](#)

# How to: Control Namespace Prefixes (C#) (LINQ to XML)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic describes how you can control namespace prefixes when serializing an XML tree.

In many situations, it is not necessary to control namespace prefixes.

However, certain XML programming tools require specific control of namespace prefixes. For example, you might be manipulating an XSLT style sheet or a XAML document that contains embedded XPath expressions that refer to specific namespace prefixes; in this case, it is important that the document be serialized with those specific prefixes.

This is the most common reason for controlling namespace prefixes.

Another common reason for controlling namespace prefixes is that you want users to edit the XML document manually, and you want to create namespace prefixes that are convenient for the user to type. For example, you might be generating an XSD document. Conventions for schemas suggest that you use either `xs` or `xsd` as the prefix for the schema namespace.

To control namespace prefixes, you insert attributes that declare namespaces. If you declare the namespaces with specific prefixes, LINQ to XML will attempt to honor the namespace prefixes when serializing.

To create an attribute that declares a namespace with a prefix, you create an attribute where the namespace of the name of the attribute is `Xmlns`, and the name of the attribute is the namespace prefix. The value of the attribute is the URI of the namespace.

## Example

This example declares two namespaces. It specifies that the `http://www.adventure-works.com` namespace has the prefix of `aw`, and that the `www.fourthcoffee.com` namespace has the prefix of `fc`.

```
XNamespace aw = "http://www.adventure-works.com";
XNamespace fc = "www.fourthcoffee.com";
XElement root = new XElement(aw + "Root",
    new XAttribute(XNamespace.Xmlns + "aw", "http://www.adventure-works.com"),
    new XAttribute(XNamespace.Xmlns + "fc", "www.fourthcoffee.com"),
    new XElement(fc + "Child",
        new XElement(aw + "DifferentChild", "other content")
    ),
    new XElement(aw + "Child2", "c2 content"),
    new XElement(fc + "Child3", "c3 content")
);
Console.WriteLine(root);
```

This example produces the following output:

```
<aw:Root xmlns:aw="http://www.adventure-works.com" xmlns:fc="www.fourthcoffee.com">
  <fc:Child>
    <aw:DifferentChild>other content</aw:DifferentChild>
  </fc:Child>
  <aw:Child2>c2 content</aw:Child2>
  <fc:Child3>c3 content</fc:Child3>
</aw:Root>
```

## See also

- [Working with XML Namespaces \(C#\)](#)

# Scope of Default Namespaces in C#

6/4/2019 • 2 minutes to read • [Edit Online](#)

Default namespaces as represented in the XML tree are not in scope for queries. If you have XML that is in a default namespace, you still must declare an [XNamespace](#) variable, and combine it with the local name to make a qualified name to be used in the query.

One of the most common problems when querying XML trees is that if the XML tree has a default namespace, the developer sometimes writes the query as though the XML were not in a namespace.

The first set of examples in this topic shows a typical way that XML in a default namespace is loaded, but is queried improperly.

The second set of examples show the necessary corrections so that you can query XML in a namespace.

## Example

This example shows the creation of XML in a namespace, and a query that returns an empty result set.

### Code

```
XElement root = XElement.Parse(
@"<Root xmlns='http://www.adventure-works.com'>
  <Child>1</Child>
  <Child>2</Child>
  <Child>3</Child>
  <AnotherChild>4</AnotherChild>
  <AnotherChild>5</AnotherChild>
  <AnotherChild>6</AnotherChild>
</Root>");
IEnumerable<XElement> c1 =
    from el in root.Elements("Child")
    select el;
Console.WriteLine("Result set follows:");
foreach (XElement el in c1)
    Console.WriteLine((int)el);
Console.WriteLine("End of result set");
```

### Comments

This example produces the following result:

```
Result set follows:
End of result set
```

## Example

This example shows the creation of XML in a namespace, and a query that is coded properly.

In contrast to the incorrectly coded example above, the correct approach when using C# is to declare and initialize an [XNamespace](#) object, and to use it when specifying [XName](#) objects. In this case, the argument to the [Elements](#) method is an [XName](#) object.

### Code

```
XElement root = XElement.Parse(
@"<Root xmlns='http://www.adventure-works.com'>
  <Child>1</Child>
  <Child>2</Child>
  <Child>3</Child>
  <AnotherChild>4</AnotherChild>
  <AnotherChild>5</AnotherChild>
  <AnotherChild>6</AnotherChild>
</Root>");
XNamespace aw = "http://www.adventure-works.com";
IEnumerable<XElement> c1 =
    from el in root.Elements(aw + "Child")
    select el;
Console.WriteLine("Result set follows:");
foreach (XElement el in c1)
    Console.WriteLine((int)el);
Console.WriteLine("End of result set");
```

## Comments

This example produces the following result:

```
Result set follows:
1
2
3
End of result set
```

## See also

- [Working with XML Namespaces \(C#\)](#)

# How to: Write Queries on XML in Namespaces (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

To write a query on XML that is in a namespace, you must use [XName](#) objects that have the correct namespace.

For C#, the most common approach is to initialize an [XNamespace](#) using a string that contains the URI, then use the addition operator overload to combine the namespace with the local name.

The first set of examples in this topic shows how to create an XML tree in a default namespace. The second set shows how to create an XML tree in a namespace with a prefix.

## Example

The following example creates an XML tree that is in a default namespace. It then retrieves a collection of elements.

```
XNamespace aw = "http://www.adventure-works.com";
XElement root = XElement.Parse(
@"<Root xmlns='http://www.adventure-works.com'>
  <Child>1</Child>
  <Child>2</Child>
  <Child>3</Child>
  <AnotherChild>4</AnotherChild>
  <AnotherChild>5</AnotherChild>
  <AnotherChild>6</AnotherChild>
</Root>");
IEnumerable<XElement> c1 =
    from el in root.Elements(aw + "Child")
    select el;
foreach (XElement el in c1)
    Console.WriteLine((int)el);
```

This example produces the following output:

```
1
2
3
```

## Example

In C#, you write queries in the same way regardless of whether you are writing queries on an XML tree that uses a namespace with a prefix or on an XML tree with a default namespace.

The following example creates an XML tree that is in a namespace with a prefix. It then retrieves a collection of elements.



```
XNamespace aw = "http://www.adventure-works.com";
XElement root = XElement.Parse(
@"<aw:Root xmlns:aw='http://www.adventure-works.com'>
  <aw:Child>1</aw:Child>
  <aw:Child>2</aw:Child>
  <aw:Child>3</aw:Child>
  <aw:AnotherChild>4</aw:AnotherChild>
  <aw:AnotherChild>5</aw:AnotherChild>
  <aw:AnotherChild>6</aw:AnotherChild>
</aw:Root>");
IEnumerable<XElement> c1 =
    from el in root.Elements(aw + "Child")
    select el;
foreach (XElement el in c1)
    Console.WriteLine((int)el);
```

This example produces the following output:

```
1
2
3
```

## See also

- [Working with XML Namespaces \(C#\)](#)

# Preserving White Space While Serializing

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic describes how to control white space when serializing an XML tree.

A common scenario is to read indented XML, create an in-memory XML tree without any white-space text nodes (that is, not preserving white space), perform some operations on the XML, and then save the XML with indentation. When you serialize the XML with formatting, only significant white space in the XML tree is preserved. This is the default behavior for LINQ to XML.

Another common scenario is to read and modify XML that has already been intentionally indented. You might not want to change this indentation in any way. To do this in LINQ to XML, you preserve white space when you load or parse the XML and disable formatting when you serialize the XML.

## White-Space Behavior of Methods that Serialize XML Trees

The following methods in the [XElement](#) and [XDocument](#) classes serialize an XML tree. You can serialize an XML tree to a file, a [TextReader](#), or an [XmlReader](#). The `ToString` method serializes to a string.

- [XElement.Save](#)
- [XDocument.Save](#)
- [XElement.ToString\(\)](#)
- [XDocument.ToString\(\)](#)

If the method does not take [SaveOptions](#) as an argument, then the method will format (indent) the serialized XML. In this case, all insignificant white space in the XML tree is discarded.

If the method does take [SaveOptions](#) as an argument, then you can specify that the method not format (indent) the serialized XML. In this case, all white space in the XML tree is preserved.

# Serializing with an XML Declaration (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic describes how to control whether serialization generates an XML declaration.

## XML Declaration Generation

Serializing to a [File](#) or a [TextWriter](#) using the [XElement.Save](#) method or the [XDocument.Save](#) method generates an XML declaration. When you serialize to an [XmlWriter](#), the writer settings (specified in an [XmlWriterSettings](#) object) determine whether an XML declaration is generated or not.

If you are serializing to a string using the `ToString` method, the resulting XML will not include an XML declaration.

### Serializing with an XML Declaration

The following example creates an [XElement](#), saves the document to a file, and then prints the file to the console:

```
XElement root = new XElement("Root",
    new XElement("Child", "child content")
);
root.Save("Root.xml");
string str = File.ReadAllText("Root.xml");
Console.WriteLine(str);
```

This example produces the following output:

```
<?xml version="1.0" encoding="utf-8"?>
<Root>
  <Child>child content</Child>
</Root>
```

### Serializing without an XML Declaration

The following example shows how to save an [XElement](#) to an [XmlWriter](#).

```
StringBuilder sb = new StringBuilder();
XmlWriterSettings xws = new XmlWriterSettings();
xws.OmitXmlDeclaration = true;

using (XmlWriter xw = XmlWriter.Create(sb, xws)) {
    XElement root = new XElement("Root",
        new XElement("Child", "child content")
    );
    root.Save(xw);
}
Console.WriteLine(sb.ToString());
```

This example produces the following output:

```
<Root><Child>child content</Child></Root>
```

See also

- [Serializing XML Trees \(C#\)](#)

# Serializing to Files, TextWriters, and XmlWriters

6/4/2019 • 2 minutes to read • [Edit Online](#)

You can serialize XML trees to a [File](#), a [TextWriter](#), or an [XmlWriter](#).

You can serialize any XML component, including [XDocument](#) and [XElement](#), to a string by using the `ToString` method.

If you want to suppress formatting when serializing to a string, you can use the [XNode.ToString](#) method.

The default behavior when serializing to a file is to format (indent) the resulting XML document. When you indent, the insignificant white space in the XML tree is not preserved. To serialize with formatting, use one of the overloads of the following methods that do not take [SaveOptions](#) as an argument:

- [XDocument.Save](#)
- [XElement.Save](#)

If you want the option not to indent and to preserve the insignificant white space in the XML tree, use one of the overloads of the following methods that takes [SaveOptions](#) as an argument:

- [XDocument.Save](#)
- [XElement.Save](#)

For examples, see the appropriate reference topic.

## See also

- [Serializing XML Trees \(C#\)](#)

# Serializing to an XmlReader (Invoking XSLT) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

When you use the [System.Xml](#) interoperability capabilities of LINQ to XML, you can use [CreateReader](#) to create an [XmlReader](#). The module that reads from this [XmlReader](#) reads the nodes from the XML tree and processes them accordingly.

## Invoking an XSLT Transformation

One possible use for this method is when invoking an XSLT transformation. You can create an XML tree, create an [XmlReader](#) from the XML tree, create a new document, and then create an [XmlWriter](#) to write into the new document. Then, you can invoke the XSLT transformation, passing in [XmlReader](#) and [XmlWriter](#). After the transformation successfully completes, the new XML tree is populated with the results of the transformation.

```
string xslMarkup = @"<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl='http://www.w3.org/1999/XSL/Transform' version='1.0'>
  <xsl:template match='/Parent'>
    <Root>
      <C1>
        <xsl:value-of select='Child1' />
      </C1>
      <C2>
        <xsl:value-of select='Child2' />
      </C2>
    </Root>
  </xsl:template>
</xsl:stylesheet>";

XDocument xmlTree = new XDocument(
    new XElement("Parent",
        new XElement("Child1", "Child1 data"),
        new XElement("Child2", "Child2 data")
    )
);

XDocument newTree = new XDocument();
using (XmlWriter writer = newTree.CreateWriter()) {
    // Load the style sheet.
    XslCompiledTransform xslt = new XslCompiledTransform();
    xslt.Load(XmlReader.Create(new StringReader(xslMarkup)));

    // Execute the transformation and output the results to a writer.
    xslt.Transform(xmlTree.CreateReader(), writer);
}

Console.WriteLine(newTree);
```

This example produces the following output:

```
<Root>
  <C1>Child1 data</C1>
  <C2>Child2 data</C2>
</Root>
```

See also

- [Serializing XML Trees \(C#\)](#)

# LINQ to XML Axes Overview (C#)

1/23/2019 • 3 minutes to read • [Edit Online](#)

After you have created an XML tree or loaded an XML document into an XML tree, you can query it to find elements and attributes and retrieve their values. You retrieve collections through the *axis methods*, also called *axes*. Some of the axes are methods in the [XElement](#) and [XDocument](#) classes that return [IEnumerable<T>](#) collections. Some of the axes are extension methods in the [Extensions](#) class. The axes that are implemented as extension methods operate on collections, and return collections.

As described in [XElement Class Overview](#), an [XElement](#) object represents a single element node. The content of an element can be complex (sometimes called structured content), or it can be a simple element. A simple element can be empty or can contain a value. If the node contains structured content, you can use the various axis methods to retrieve enumerations of descendant elements. The most commonly used axis methods are [Elements](#) and [Descendants](#).

In addition to the axis methods, which return collections, there are two more methods that you will commonly use in LINQ to XML queries. The [Element](#) method returns a single [XElement](#). The [Attribute](#) method returns a single [XAttribute](#).

For many purposes, LINQ queries provide the most powerful way to examine a tree, extract data from it, and transform it. LINQ queries operate on objects that implement [IEnumerable<T>](#), and the LINQ to XML axes return [IEnumerable<T>](#) of [XElement](#) collections, and [IEnumerable<T>](#) of [XAttribute](#) collections. You need these collections to perform your queries.

In addition to the axis methods that retrieve collections of elements and attributes, there are axis methods that allow you to iterate through the tree in great detail. For example, instead of dealing with elements and attributes, you can work with the nodes of the tree. Nodes are a finer level of granularity than elements and attributes. When working with nodes, you can examine XML comments, text nodes, processing instructions, and more. This functionality is important, for example, to someone who is writing a word processor and wants to save documents as XML. However, the majority of XML programmers are primarily concerned with elements, attributes, and their values.

## Methods for Retrieving a Collection of Elements

The following is a summary of the methods of the [XElement](#) class (or its base classes) that you call on an [XElement](#) to return a collection of elements.

METHOD	DESCRIPTION
<a href="#">XNode.Ancestors</a>	Returns an <a href="#">IEnumerable&lt;T&gt;</a> of <a href="#">XElement</a> of the ancestors of this element. An overload returns an <a href="#">IEnumerable&lt;T&gt;</a> of <a href="#">XElement</a> of the ancestors that have the specified <a href="#">XName</a> .
<a href="#">XContainer.Descendants</a>	Returns an <a href="#">IEnumerable&lt;T&gt;</a> of <a href="#">XElement</a> of the descendants of this element. An overload returns an <a href="#">IEnumerable&lt;T&gt;</a> of <a href="#">XElement</a> of the descendants that have the specified <a href="#">XName</a> .
<a href="#">XContainer.Elements</a>	Returns an <a href="#">IEnumerable&lt;T&gt;</a> of <a href="#">XElement</a> of the child elements of this element. An overload returns an <a href="#">IEnumerable&lt;T&gt;</a> of <a href="#">XElement</a> of the child elements that have the specified <a href="#">XName</a> .



METHOD	DESCRIPTION
<a href="#">XNode.ElementsAfterSelf</a>	Returns an <a href="#">IEnumerable&lt;T&gt;</a> of <a href="#">XElement</a> of the elements that come after this element. An overload returns an <a href="#">IEnumerable&lt;T&gt;</a> of <a href="#">XElement</a> of the elements after this element that have the specified <a href="#">XName</a> .
<a href="#">XNode.ElementsBeforeSelf</a>	Returns an <a href="#">IEnumerable&lt;T&gt;</a> of <a href="#">XElement</a> of the elements that come before this element. An overload returns an <a href="#">IEnumerable&lt;T&gt;</a> of <a href="#">XElement</a> of the elements before this element that have the specified <a href="#">XName</a> .
<a href="#">XElement.AncestorsAndSelf</a>	Returns an <a href="#">IEnumerable&lt;T&gt;</a> of <a href="#">XElement</a> of this element and its ancestors. An overload returns an <a href="#">IEnumerable&lt;T&gt;</a> of <a href="#">XElement</a> of the elements that have the specified <a href="#">XName</a> .
<a href="#">XElement.DescendantsAndSelf</a>	Returns an <a href="#">IEnumerable&lt;T&gt;</a> of <a href="#">XElement</a> of this element and its descendants. An overload returns an <a href="#">IEnumerable&lt;T&gt;</a> of <a href="#">XElement</a> of the elements that have the specified <a href="#">XName</a> .

## Method for Retrieving a Single Element

The following method retrieves a single child from an [XElement](#) object.

METHOD	DESCRIPTION
<a href="#">XContainer.Element</a>	Returns the first child <a href="#">XElement</a> object that has the specified <a href="#">XName</a> .

## Method for Retrieving a Collection of Attributes

The following method retrieves attributes from an [XElement](#) object.

METHOD	DESCRIPTION
<a href="#">XElement.Attributes</a>	Returns an <a href="#">IEnumerable&lt;T&gt;</a> of <a href="#">XAttribute</a> of all of the attributes.

## Method for Retrieving a Single Attribute

The following method retrieves a single attribute from an [XElement](#) object.

METHOD	DESCRIPTION
<a href="#">XElement.Attribute</a>	Returns the <a href="#">XAttribute</a> that has the specified <a href="#">XName</a> .

## See also

- [LINQ to XML Axes \(C#\)](#)

# How to: Retrieve a Collection of Elements (LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic demonstrates the [Elements](#) method. This method retrieves a collection of the child elements of an element.

## Example

This example iterates through the child elements of the `purchaseOrder` element.

This example uses the following XML document: [Sample XML File: Typical Purchase Order \(LINQ to XML\)](#).

```
XElement po = XElement.Load("PurchaseOrder.xml");
IEnumerable<XElement> childElements =
    from el in po.Elements()
    select el;
foreach (XElement el in childElements)
    Console.WriteLine("Name: " + el.Name);
```

This example produces the following output.

```
Name: Address
Name: Address
Name: DeliveryNotes
Name: Items
```

## See also

- [LINQ to XML Axes \(C#\)](#)

# How to: Retrieve the Value of an Element (LINQ to XML) (C#)

6/4/2019 • 3 minutes to read • [Edit Online](#)

This topic shows how to get the value of elements. There are two main ways to do this. One way is to cast an [XElement](#) or an [XAttribute](#) to the desired type. The explicit conversion operator then converts the contents of the element or attribute to the specified type and assigns it to your variable. Alternatively, you can use the [XElement.Value](#) property or the [XAttribute.Value](#) property.

With C#, however, casting is generally the better approach. If you cast the element or attribute to a nullable type, the code is simpler to write when retrieving the value of an element (or attribute) that might or might not exist. The last example in this topic demonstrates this. However, you cannot set the contents of an element through casting, as you can through [XElement.Value](#) property.

## Example

To retrieve the value of an element, you just cast the [XElement](#) object to your desired type. You can always cast an element to a string, as follows:

```
XElement e = new XElement("StringElement", "abcde");
Console.WriteLine(e);
Console.WriteLine("Value of e:" + (string)e);
```

This example produces the following output:

```
<StringElement>abcde</StringElement>
Value of e:abcde
```

## Example

You can also cast elements to types other than string. For example, if you have an element that contains an integer, you can cast it to `int`, as shown in the following code:

```
XElement e = new XElement("Age", "44");
Console.WriteLine(e);
Console.WriteLine("Value of e:" + (int)e);
```

This example produces the following output:

```
<Age>44</Age>
Value of e:44
```

LINQ to XML provides explicit cast operators for the following data types: `string`, `bool`, `bool?`, `int`, `int?`, `uint`, `uint?`, `long`, `long?`, `ulong`, `ulong?`, `float`, `float?`, `double`, `double?`, `decimal`, `decimal?`, `DateTime`, `DateTime?`, `TimeSpan`, `TimeSpan?`, `GUID`, and `GUID?`.

LINQ to XML provides the same cast operators for [XAttribute](#) objects.

## Example

You can use the [Value](#) property to retrieve the contents of an element:

```
XElement e = new XElement("StringElement", "abcde");  
Console.WriteLine(e);  
Console.WriteLine("Value of e:" + e.Value);
```

This example produces the following output:

```
<StringElement>abcde</StringElement>  
Value of e:abcde
```

## Example

Sometimes you try to retrieve the value of an element even though you are not sure it exists. In this case, when you assign the casted element to a nullable type (either `string` or one of the nullable types in the .NET Framework), if the element does not exist the assigned variable is just set to `null`. The following code shows that when the element might or might not exist, it is easier to use casting than to use the [Value](#) property.

```

XElement root = new XElement("Root",
    new XElement("Child1", "child 1 content"),
    new XElement("Child2", "2")
);

// The following assignments show why it is easier to use
// casting when the element might or might not exist.

string c1 = (string)root.Element("Child1");
Console.WriteLine("c1:{0}", c1 == null ? "element does not exist" : c1);

int? c2 = (int?)root.Element("Child2");
Console.WriteLine("c2:{0}", c2 == null ? "element does not exist" : c2.ToString());

string c3 = (string)root.Element("Child3");
Console.WriteLine("c3:{0}", c3 == null ? "element does not exist" : c3);

int? c4 = (int?)root.Element("Child4");
Console.WriteLine("c4:{0}", c4 == null ? "element does not exist" : c4.ToString());

Console.WriteLine();

// The following assignments show the required code when using
// the Value property when the element might or might not exist.
// Notice that this is more difficult than the casting approach.

XElement e1 = root.Element("Child1");
string v1;
if (e1 == null)
    v1 = null;
else
    v1 = e1.Value;
Console.WriteLine("v1:{0}", v1 == null ? "element does not exist" : v1);

XElement e2 = root.Element("Child2");
int? v2;
if (e2 == null)
    v2 = null;
else
    v2 = Int32.Parse(e2.Value);
Console.WriteLine("v2:{0}", v2 == null ? "element does not exist" : v2.ToString());

XElement e3 = root.Element("Child3");
string v3;
if (e3 == null)
    v3 = null;
else
    v3 = e3.Value;
Console.WriteLine("v3:{0}", v3 == null ? "element does not exist" : v3);

XElement e4 = root.Element("Child4");
int? v4;
if (e4 == null)
    v4 = null;
else
    v4 = Int32.Parse(e4.Value);
Console.WriteLine("v4:{0}", v4 == null ? "element does not exist" : v4.ToString());

```

This code produces the following output:

```
c1:child 1 content
c2:2
c3:element does not exist
c4:element does not exist

v1:child 1 content
v2:2
v3:element does not exist
v4:element does not exist
```

In general, you can write simpler code when using casting to retrieve the contents of elements and attributes.

## See also

- [LINQ to XML Axes \(C#\)](#)

# How to: Filter on Element Names (LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

When you call one of the methods that return `IEnumerable<T>` of `XElement`, you can filter on the element name.

## Example

This example retrieves a collection of descendants that is filtered to contain only descendants with the specified name.

This example uses the following XML document: [Sample XML File: Typical Purchase Order \(LINQ to XML\)](#).

```
XElement po = XElement.Load("PurchaseOrder.xml");
IEnumerable<XElement> items =
    from el in po.Descendants("ProductName")
    select el;
foreach(XElement prdName in items)
    Console.WriteLine(prdName.Name + ":" + (string) prdName);
```

This code produces the following output:

```
ProductName:Lawnmower
ProductName:Baby Monitor
```

The other methods that return `IEnumerable<T>` of `XElement` collections follow the same pattern. Their signatures are similar to [Elements](#) and [Descendants](#). The following is the complete list of methods that have similar method signatures:

- [Ancestors](#)
- [Descendants](#)
- [Elements](#)
- [ElementsAfterSelf](#)
- [ElementsBeforeSelf](#)
- [AncestorsAndSelf](#)
- [DescendantsAndSelf](#)

## Example

The following example shows the same query for XML that is in a namespace. For more information, see [Working with XML Namespaces \(C#\)](#).

This example uses the following XML document: [Sample XML File: Typical Purchase Order in a Namespace](#).

```
XNamespace aw = "http://www.adventure-works.com";
XElement po = XElement.Load("PurchaseOrderInNamespace.xml");
IEnumerable<XElement> items =
    from el in po.Descendants(aw + "ProductName")
    select el;
foreach (XElement prdName in items)
    Console.WriteLine(prdName.Name + ":" + (string)prdName);
```

This code produces the following output:

```
{http://www.adventure-works.com}ProductName:Lawnmower
{http://www.adventure-works.com}ProductName:Baby Monitor
```

## See also

- [LINQ to XML Axes \(C#\)](#)



# How to: Chain Axis Method Calls (LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

A common pattern that you will use in your code is to call an axis method, then call one of the extension method axes.

There are two axes with the name of `Elements` that return a collection of elements: the `XContainer.Elements` method and the `Extensions.Elements` method. You can combine these two axes to find all elements of a specified name at a given depth in the tree.

## Example

This example uses `XContainer.Elements` and `Extensions.Elements` to find all `Name` elements in all `Address` elements in all `PurchaseOrder` elements.

This example uses the following XML document: [Sample XML File: Multiple Purchase Orders \(LINQ to XML\)](#).

```
XElement purchaseOrders = XElement.Load("PurchaseOrders.xml");
IEnumerable<XElement> names =
    from el in purchaseOrders
        .Elements("PurchaseOrder")
        .Elements("Address")
        .Elements("Name")
    select el;
foreach (XElement e in names)
    Console.WriteLine(e);
```

This example produces the following output:

```
<Name>Ellen Adams</Name>
<Name>Tai Yee</Name>
<Name>Cristian Osorio</Name>
<Name>Cristian Osorio</Name>
<Name>Jessica Arnold</Name>
<Name>Jessica Arnold</Name>
```

This works because one of the implementations of the `Elements` axis is as an extension method on `IEnumerable<T>` of `XContainer`. `XElement` derives from `XContainer`, so you can call the `Extensions.Elements` method on the results of a call to the `XContainer.Elements` method.

## Example

Sometimes you want to retrieve all elements at a particular element depth when there might or might not be intervening ancestors. For example, in the following document, you might want to retrieve all the `ConfigParameter` elements that are children of the `Customer` element, but not the `ConfigParameter` that is a child of the `Root` element.

```

<Root>
  <ConfigParameter>RootConfigParameter</ConfigParameter>
  <Customer>
    <Name>Frank</Name>
    <Config>
      <ConfigParameter>FirstConfigParameter</ConfigParameter>
    </Config>
  </Customer>
  <Customer>
    <Name>Bob</Name>
    <!--This customer doesn't have a Config element-->
  </Customer>
  <Customer>
    <Name>Bill</Name>
    <Config>
      <ConfigParameter>SecondConfigParameter</ConfigParameter>
    </Config>
  </Customer>
</Root>

```

To do this, you can use the [Extensions.Elements](#) axis, as follows:

```

XElement root = XElement.Load("Irregular.xml");
IEnumerable<XElement> configParameters =
    root.Elements("Customer").Elements("Config").
    Elements("ConfigParameter");
foreach (XElement cp in configParameters)
    Console.WriteLine(cp);

```

This example produces the following output:

```

<ConfigParameter>FirstConfigParameter</ConfigParameter>
<ConfigParameter>SecondConfigParameter</ConfigParameter>

```

## Example

The following example shows the same technique for XML that is in a namespace. For more information, see [Working with XML Namespaces \(C#\)](#).

This example uses the following XML document: [Sample XML File: Multiple Purchase Orders in a Namespace](#).

```

XNamespace aw = "http://www.adventure-works.com";
XElement purchaseOrders = XElement.Load("PurchaseOrdersInNamespace.xml");
IEnumerable<XElement> names =
    from el in purchaseOrders
        .Elements(aw + "PurchaseOrder")
        .Elements(aw + "Address")
        .Elements(aw + "Name")
    select el;
foreach (XElement e in names)
    Console.WriteLine(e);

```

This example produces the following output:

```
<aw:Name xmlns:aw="http://www.adventure-works.com">Ellen Adams</aw:Name>  
<aw:Name xmlns:aw="http://www.adventure-works.com">Tai Yee</aw:Name>  
<aw:Name xmlns:aw="http://www.adventure-works.com">Cristian Osorio</aw:Name>  
<aw:Name xmlns:aw="http://www.adventure-works.com">Cristian Osorio</aw:Name>  
<aw:Name xmlns:aw="http://www.adventure-works.com">Jessica Arnold</aw:Name>  
<aw:Name xmlns:aw="http://www.adventure-works.com">Jessica Arnold</aw:Name>
```

## See also

- [LINQ to XML Axes \(C#\)](#)

# How to: Retrieve a Single Child Element (LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic explains how to retrieve a single child element, given the name of the child element. When you know the name of the child element and that there is only one element that has this name, it can be convenient to retrieve just one element, instead of a collection.

The [Element](#) method returns the first child [XElement](#) with the specified [XName](#).

If you want to retrieve a single child element in Visual Basic, a common approach is to use the XML property, and then retrieve the first element using array indexer notation.

## Example

The following example demonstrates the use of the [Element](#) method. This example takes the XML tree named `po` and finds the first element named `Comment`.

The Visual Basic example shows using array indexer notation to retrieve a single element.

This example uses the following XML document: [Sample XML File: Typical Purchase Order \(LINQ to XML\)](#).

```
XElement po = XElement.Load("PurchaseOrder.xml");
XElement e = po.Element("DeliveryNotes");
Console.WriteLine(e);
```

This example produces the following output:

```
<DeliveryNotes>Please leave packages in shed by driveway.</DeliveryNotes>
```

## Example

The following example shows the same code for XML that is in a namespace. For more information, see [Working with XML Namespaces \(C#\)](#).

This example uses the following XML document: [Sample XML File: Typical Purchase Order in a Namespace](#).

```
XElement po = XElement.Load("PurchaseOrderInNamespace.xml");
XNamespace aw = "http://www.adventure-works.com";
XElement e = po.Element(aw + "DeliveryNotes");
Console.WriteLine(e);
```

This example produces the following output:

```
<aw:DeliveryNotes xmlns:aw="http://www.adventure-works.com">Please leave packages in shed by driveway.
</aw:DeliveryNotes>
```

## See also

- [LINQ to XML Axes \(C#\)](#)

# How to: Retrieve a Collection of Attributes (LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic introduces the [Attributes](#) method. This method retrieves the attributes of an element.

## Example

The following example shows how to iterate through the collection of attributes of an element.

```
XElement val = new XElement("Value",
    new XAttribute("ID", "1243"),
    new XAttribute("Type", "int"),
    new XAttribute("ConvertibleTo", "double"),
    "100");
IEnumerable<XAttribute> listOfAttributes =
    from att in val.Attributes()
    select att;
foreach (XAttribute a in listOfAttributes)
    Console.WriteLine(a);
```

This code produces the following output:

```
ID="1243"
Type="int"
ConvertibleTo="double"
```

## See also

- [LINQ to XML Axes \(C#\)](#)

# How to: Retrieve a Single Attribute (LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic explains how to retrieve a single attribute of an element, given the attribute name. This is useful for writing query expressions where you want to find an element that has a particular attribute.

The [Attribute](#) method of the [XElement](#) class returns the [XAttribute](#) with the specified name.

## Example

The following example uses the [Attribute](#) method.

```
XElement cust = new XElement("PhoneNumbers",
    new XElement("Phone",
        new XAttribute("type", "home"),
        "555-555-5555"),
    new XElement("Phone",
        new XAttribute("type", "work"),
        "555-555-6666")
);
IEnumerable<XElement> elList =
    from el in cust.Descendants("Phone")
    select el;
foreach (XElement el in elList)
    Console.WriteLine((string)el.Attribute("type"));
```

This example finds all the descendants in the tree named `Phone`, and then finds the attribute named `type`.

This code produces the following output:

```
home
work
```

## Example

If you want to retrieve the value of the attribute, you can cast it, just as you do for with [XElement](#) objects. The following example demonstrates this.

```
XElement cust = new XElement("PhoneNumbers",
    new XElement("Phone",
        new XAttribute("type", "home"),
        "555-555-5555"),
    new XElement("Phone",
        new XAttribute("type", "work"),
        "555-555-6666")
);
IEnumerable<XElement> elList =
    from el in cust.Descendants("Phone")
    select el;
foreach (XElement el in elList)
    Console.WriteLine((string)el.Attribute("type"));
```

This code produces the following output:

```
home
work
```

LINQ to XML provides explicit cast operators for the [XAttribute](#) class to `string`, `bool`, `bool?`, `int`, `int?`, `uint`, `uint?`, `long`, `long?`, `ulong`, `ulong?`, `float`, `float?`, `double`, `double?`, `decimal`, `decimal?`, `DateTime`, `DateTime?`, `TimeSpan`, `TimeSpan?`, `GUID`, and `GUID?`.

## Example

The following example shows the same code for an attribute that is in a namespace. For more information, see [Working with XML Namespaces \(C#\)](#).

```
XNamespace aw = "http://www.adventure-works.com";
XElement cust = new XElement(aw + "PhoneNumbers",
    new XElement(aw + "Phone",
        new XAttribute(aw + "type", "home"),
        "555-555-5555"),
    new XElement(aw + "Phone",
        new XAttribute(aw + "type", "work"),
        "555-555-6666")
);
IEnumerable<XElement> ellist =
    from el in cust.Descendants(aw + "Phone")
    select el;
foreach (XElement el in ellist)
    Console.WriteLine((string)el.Attribute(aw + "type"));
```

This code produces the following output:

```
home
work
```

## See also

- [LINQ to XML Axes \(C#\)](#)



# How to: Retrieve the Value of an Attribute (LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how to obtain the value of attributes. There are two main ways: You can cast an [XAttribute](#) to the desired type; the explicit conversion operator then converts the contents of the element or attribute to the specified type. Alternatively, you can use the [Value](#) property. However, casting is generally the better approach. If you cast the attribute to a nullable type, the code is simpler to write when retrieving the value of an attribute that might or might not exist. For examples of this technique, see [How to: Retrieve the Value of an Element \(LINQ to XML\) \(C#\)](#).

## Example

To retrieve the value of an attribute, you just cast the [XAttribute](#) object to your desired type.

```
XElement root = new XElement("Root",
    new XAttribute("Attr", "abcde")
);
Console.WriteLine(root);
string str = (string)root.Attribute("Attr");
Console.WriteLine(str);
```

This example produces the following output:

```
<Root Attr="abcde" />
abcde
```

## Example

The following example shows how to retrieve the value of an attribute where the attribute is in a namespace. For more information, see [Working with XML Namespaces \(C#\)](#).

```
XNamespace aw = "http://www.adventure-works.com";
XElement root = new XElement(aw + "Root",
    new XAttribute(aw + "Attr", "abcde")
);
string str = (string)root.Attribute(aw + "Attr");
Console.WriteLine(str);
```

This example produces the following output:

```
abcde
```

## See also

- [LINQ to XML Axes \(C#\)](#)

# How to: Retrieve the Shallow Value of an Element (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how to get the shallow value of an element. The shallow value is the value of the specific element only, as opposed to the deep value, which includes the values of all descendent elements concatenated into a single string.

When you retrieve an element value by using either casting or the [XElement.Value](#) property, you retrieve the deep value. To retrieve the shallow value, you can use the `ShallowValue` extension method, as shown in the following example. Retrieving the shallow value is useful when you want to select elements based on their content.

The following example declares an extension method that retrieves the shallow value of an element. It then uses the extension method in a query to list all elements that contain a calculated value.

## Example

The following text file, Report.xml, is the source for this example.

```
<?xml version="1.0" encoding="utf-8" ?>
<Report>
  <Section>
    <Heading>
      <Column Name="CustomerId">=Customer.CustomerId.Heading</Column>
      <Column Name="Name">=Customer.Name.Heading</Column>
    </Heading>
    <Detail>
      <Column Name="CustomerId">=Customer.CustomerId</Column>
      <Column Name="Name">=Customer.Name</Column>
    </Detail>
  </Section>
</Report>
```

```

public static class MyExtensions
{
    public static string ShallowValue(this XElement xe)
    {
        return xe
            .Nodes()
            .OfType<XText>()
            .Aggregate(new StringBuilder(),
                (s, c) => s.Append(c),
                s => s.ToString());
    }
}

class Program
{
    static void Main(string[] args)
    {
        XElement root = XElement.Load("Report.xml");

        IEnumerable<XElement> query = from el in root.Descendants()
                                        where el.ShallowValue().StartsWith("=")
                                        select el;

        foreach (var q in query)
        {
            Console.WriteLine("{0}{1}{2}",
                q.Name.ToString().PadRight(8),
                q.Attribute("Name").ToString().PadRight(20),
                q.ShallowValue());
        }
    }
}

```

This example produces the following output:

```

Column  Name="CustomerId"  =Customer.CustomerId.Heading
Column  Name="Name"      =Customer.Name.Heading
Column  Name="CustomerId" =Customer.CustomerId
Column  Name="Name"      =Customer.Name

```

## See also

- [LINQ to XML Axes \(C#\)](#)

# How to: Find an Element with a Specific Attribute (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how to find an element that has an attribute that has a specific value.

## Example

The example shows how to find the `Address` element that has a `Type` attribute with a value of "Billing".

This example uses the following XML document: [Sample XML File: Typical Purchase Order \(LINQ to XML\)](#).

```
XElement root = XElement.Load("PurchaseOrder.xml");
IEnumerable<XElement> address =
    from el in root.Elements("Address")
    where (string)el.Attribute("Type") == "Billing"
    select el;
foreach (XElement el in address)
    Console.WriteLine(el);
```

This code produces the following output:

```
<Address Type="Billing">
  <Name>Tai Yee</Name>
  <Street>8 Oak Avenue</Street>
  <City>Old Town</City>
  <State>PA</State>
  <Zip>95819</Zip>
  <Country>USA</Country>
</Address>
```

## Example

The following example shows the same query for XML that is in a namespace. For more information, see [Working with XML Namespaces \(C#\)](#).

This example uses the following XML document: [Sample XML File: Typical Purchase Order in a Namespace](#).

```
XElement root = XElement.Load("PurchaseOrderInNamespace.xml");
XNamespace aw = "http://www.adventure-works.com";
IEnumerable<XElement> address =
    from el in root.Elements(aw + "Address")
    where (string)el.Attribute(aw + "Type") == "Billing"
    select el;
foreach (XElement el in address)
    Console.WriteLine(el);
```

This code produces the following output:

```
<aw:Address aw:Type="Billing" xmlns:aw="http://www.adventure-works.com">
  <aw:Name>Tai Yee</aw:Name>
  <aw:Street>8 Oak Avenue</aw:Street>
  <aw:City>Old Town</aw:City>
  <aw:State>PA</aw:State>
  <aw:Zip>95819</aw:Zip>
  <aw:Country>USA</aw:Country>
</aw:Address>
```

## See also

- [Attribute](#)
- [Elements](#)
- [Standard Query Operators Overview \(C#\)](#)
- [Projection Operations \(C#\)](#)

# How to: Find an Element with a Specific Child Element (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how to find a particular element that has a child element with a specific value.

## Example

The example finds the `Test` element that has a `CommandLine` child element with the value of "Examp2.EXE".

This example uses the following XML document: [Sample XML File: Test Configuration \(LINQ to XML\)](#).

```
XElement root = XElement.Load("TestConfig.xml");
IEnumerable<XElement> tests =
    from el in root.Elements("Test")
    where (string)el.Element("CommandLine") == "Examp2.EXE"
    select el;
foreach (XElement el in tests)
    Console.WriteLine((string)el.Attribute("TestId"));
```

This code produces the following output:

```
0002
0006
```

## Example

The following example shows the same query for XML that is in a namespace. For more information, see [Working with XML Namespaces \(C#\)](#).

This example uses the following XML document: [Sample XML File: Test Configuration in a Namespace](#).

```
XElement root = XElement.Load("TestConfigInNamespace.xml");
XNamespace ad = "http://www.adatum.com";
IEnumerable<XElement> tests =
    from el in root.Elements(ad + "Test")
    where (string)el.Element(ad + "CommandLine") == "Examp2.EXE"
    select el;
foreach (XElement el in tests)
    Console.WriteLine((string)el.Attribute("TestId"));
```

This code produces the following output:

```
0002
0006
```

## See also

- [Attribute](#)
- [Elements](#)

- [Standard Query Operators Overview \(C#\)](#)
- [Projection Operations \(C#\)](#)

# Querying an XDocument vs. Querying an XElement (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

When you load a document via [XDocument.Load](#), you will notice that you have to write queries slightly differently than when you load via [XElement.Load](#).

## Comparison of XDocument.Load and XElement.Load

When you load an XML document into an [XElement](#) via [XElement.Load](#), the [XElement](#) at the root of the XML tree contains the root element of the loaded document. However, when you load the same XML document into an [XDocument](#) via [XDocument.Load](#), the root of the tree is an [XDocument](#) node, and the root element of the loaded document is the one allowed child [XElement](#) node of the [XDocument](#). The LINQ to XML axes operate relative to the root node.

This first example loads an XML tree using [Load](#). It then queries for the child elements of the root of the tree.

```
// Create a simple document and write it to a file
File.WriteAllText("Test.xml", @"<Root>
  <Child1>1</Child1>
  <Child2>2</Child2>
  <Child3>3</Child3>
</Root>");

Console.WriteLine("Querying tree loaded with XElement.Load");
Console.WriteLine("----");
XElement doc = XElement.Load("Test.xml");
IEnumerable<XElement> childList =
    from el in doc.Elements()
    select el;
foreach (XElement e in childList)
    Console.WriteLine(e);
```

As expected, this example produces the following output:

```
Querying tree loaded with XElement.Load
----
<Child1>1</Child1>
<Child2>2</Child2>
<Child3>3</Child3>
```

The following example is the same as the one above, with the exception that the XML tree is loaded into an [XDocument](#) instead of an [XElement](#).



```
// Create a simple document and write it to a file
File.WriteAllText("Test.xml", @"<Root>
    <Child1>1</Child1>
    <Child2>2</Child2>
    <Child3>3</Child3>
</Root>");

Console.WriteLine("Querying tree loaded with XDocument.Load");
Console.WriteLine("----");
XDocument doc = XDocument.Load("Test.xml");
IEnumerable<XElement> childList =
    from el in doc.Elements()
    select el;
foreach (XElement e in childList)
    Console.WriteLine(e);
```

This example produces the following output:

```
Querying tree loaded with XDocument.Load
----
<Root>
  <Child1>1</Child1>
  <Child2>2</Child2>
  <Child3>3</Child3>
</Root>
```

Notice that the same query returned the one `Root` node instead of the three child nodes.

One approach to dealing with this is to use the `Root` property before accessing the axes methods, as follows:

```
// Create a simple document and write it to a file
File.WriteAllText("Test.xml", @"<Root>
    <Child1>1</Child1>
    <Child2>2</Child2>
    <Child3>3</Child3>
</Root>");

Console.WriteLine("Querying tree loaded with XDocument.Load");
Console.WriteLine("----");
XDocument doc = XDocument.Load("Test.xml");
IEnumerable<XElement> childList =
    from el in doc.Root.Elements()
    select el;
foreach (XElement e in childList)
    Console.WriteLine(e);
```

This query now performs in the same way as the query on the tree rooted in `XElement`. The example produces the following output:

```
Querying tree loaded with XDocument.Load
----
<Child1>1</Child1>
<Child2>2</Child2>
<Child3>3</Child3>
```

# How to: Find Descendants with a Specific Element Name (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

Sometimes you want to find all descendants with a particular name. You could write code to iterate through all of the descendants, but it is easier to use the [Descendants](#) axis.

## Example

The following example shows how to find descendants based on the element name.

```
XElement root = XElement.Parse(@"<root>
  <para>
    <r>
      <t>Some text </t>
    </r>
    <n>
      <r>
        <t>that is broken up into </t>
      </r>
    </n>
    <n>
      <r>
        <t>multiple segments.</t>
      </r>
    </n>
  </para>
</root>");
IEnumerable<string> textSegs =
    from seg in root.Descendants("t")
    select (string)seg;

string str = textSegs.Aggregate(new StringBuilder(),
    (sb, i) => sb.Append(i),
    sp => sp.ToString()
);

Console.WriteLine(str);
```

This code produces the following output:

```
Some text that is broken up into multiple segments.
```

## Example

The following example shows the same query for XML that is in a namespace. For more information, see [Working with XML Namespaces \(C#\)](#).

```

XElement root = XElement.Parse(@"<root xmlns='http://www.adatum.com'>
  <para>
    <r>
      <t>Some text </t>
    </r>
    <n>
      <r>
        <t>that is broken up into </t>
      </r>
    </n>
    <n>
      <r>
        <t>multiple segments.</t>
      </r>
    </n>
  </para>
</root>");
XNamespace ad = "http://www.adatum.com";
IEnumerable<string> textSegs =
  from seg in root.Descendants(ad + "t")
  select (string)seg;

string str = textSegs.Aggregate(new StringBuilder(),
  (sb, i) => sb.Append(i),
  sp => sp.ToString()
);

Console.WriteLine(str);

```

This code produces the following output:

```
Some text that is broken up into multiple segments.
```

## See also

- [Descendants](#)

# How to: Find a Single Descendant Using the Descendants Method (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

You can use the [Descendants](#) axis method to quickly write code to find a single uniquely named element. This technique is especially useful when you want to find a particular descendant with a specific name. You could write the code to navigate to the desired element, but it is often faster and easier to write the code using the [Descendants](#) axis.

## Example

This example uses the [First](#) standard query operator.

```
XElement root = XElement.Parse(@"<Root>
  <Child1>
    <GrandChild1>GC1 Value</GrandChild1>
  </Child1>
  <Child2>
    <GrandChild2>GC2 Value</GrandChild2>
  </Child2>
  <Child3>
    <GrandChild3>GC3 Value</GrandChild3>
  </Child3>
  <Child4>
    <GrandChild4>GC4 Value</GrandChild4>
  </Child4>
</Root>");
string grandChild3 = (string)
  (from el in root.Descendants("GrandChild3")
   select el).First();
Console.WriteLine(grandChild3);
```

This code produces the following output:

```
GC3 Value
```

## Example

The following example shows the same query for XML that is in a namespace. For more information, see [Working with XML Namespaces \(C#\)](#).

```
XElement root = XElement.Parse(@"<aw:Root xmlns:aw='http://www.adventure-works.com'>
  <aw:Child1>
    <aw:GrandChild1>GC1 Value</aw:GrandChild1>
  </aw:Child1>
  <aw:Child2>
    <aw:GrandChild2>GC2 Value</aw:GrandChild2>
  </aw:Child2>
  <aw:Child3>
    <aw:GrandChild3>GC3 Value</aw:GrandChild3>
  </aw:Child3>
  <aw:Child4>
    <aw:GrandChild4>GC4 Value</aw:GrandChild4>
  </aw:Child4>
</aw:Root>");
XNamespace aw = "http://www.adventure-works.com";
string grandChild3 = (string)
  (from el in root.Descendants(aw + "GrandChild3")
   select el).First();
Console.WriteLine(grandChild3);
```

This code produces the following output:

```
GC3 Value
```

# How to: Write Queries with Complex Filtering (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

Sometimes you want to write LINQ to XML queries with complex filters. For example, you might have to find all elements that have a child element with a particular name and value. This topic gives an example of writing a query with complex filtering.

## Example

This example shows how to find all `PurchaseOrder` elements that have a child `Address` element that has a `Type` attribute equal to "Shipping" and a child `State` element equal to "NY". It uses a nested query in the `Where` clause, and the `Any` operator returns `true` if the collection has any elements in it. For information about using method-based query syntax, see [Query Syntax and Method Syntax in LINQ](#).

This example uses the following XML document: [Sample XML File: Multiple Purchase Orders \(LINQ to XML\)](#).

For more information about the `Any` operator, see [Quantifier Operations \(C#\)](#).

```
XElement root = XElement.Load("PurchaseOrders.xml");
IEnumerable<XElement> purchaseOrders =
    from el in root.Elements("PurchaseOrder")
    where
        (from add in el.Elements("Address")
         where
             (string)add.Attribute("Type") == "Shipping" &&
             (string)add.Element("State") == "NY"
         select add)
        .Any()
    select el;
foreach (XElement el in purchaseOrders)
    Console.WriteLine((string)el.Attribute("PurchaseOrderNumber"));
```

This code produces the following output:

```
99505
```

## Example

The following example shows the same query for XML that is in a namespace. For more information, see [Working with XML Namespaces \(C#\)](#).

This example uses the following XML document: [Sample XML File: Multiple Purchase Orders in a Namespace](#).

```
XElement root = XElement.Load("PurchaseOrdersInNamespace.xml");
XNamespace aw = "http://www.adventure-works.com";
IEnumerable<XElement> purchaseOrders =
    from el in root.Elements(aw + "PurchaseOrder")
    where
        (from add in el.Elements(aw + "Address")
        where
            (string)add.Attribute(aw + "Type") == "Shipping" &&
            (string)add.Element(aw + "State") == "NY"
        select add)
        .Any()
    select el;
foreach (XElement el in purchaseOrders)
    Console.WriteLine((string)el.Attribute(aw + "PurchaseOrderNumber"));
```

This code produces the following output:

```
99505
```

## See also

- [Attribute](#)
- [Elements](#)
- [Projection Operations \(C#\)](#)
- [Quantifier Operations \(C#\)](#)

# How to: Filter on an Optional Element (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

Sometimes you want to filter for an element even though you are not sure it exists in your XML document. The search should be executed so that if the particular element does not have the child element, you do not trigger a null reference exception by filtering for it. In the following example, the `Child5` element does not have a `Type` child element, but the query still executes correctly.

## Example

This example uses the [Elements](#) extension method.

```
XElement root = XElement.Parse(@"<Root>
  <Child1>
    <Text>Child One Text</Text>
    <Type Value=""Yes""/>
  </Child1>
  <Child2>
    <Text>Child Two Text</Text>
    <Type Value=""Yes""/>
  </Child2>
  <Child3>
    <Text>Child Three Text</Text>
    <Type Value=""No""/>
  </Child3>
  <Child4>
    <Text>Child Four Text</Text>
    <Type Value=""Yes""/>
  </Child4>
  <Child5>
    <Text>Child Five Text</Text>
  </Child5>
</Root>");
var cList =
    from typeElement in root.Elements().Elements("Type")
    where (string)typeElement.Attribute("Value") == "Yes"
    select (string)typeElement.Parent.Element("Text");
foreach(string str in cList)
    Console.WriteLine(str);
```

This code produces the following output:

```
Child One Text
Child Two Text
Child Four Text
```

## Example

The following example shows the same query for XML that is in a namespace. For more information, see [Working with XML Namespaces \(C#\)](#).



```

XElement root = XElement.Parse(@"<Root xmlns='http://www.adatum.com'>
  <Child1>
    <Text>Child One Text</Text>
    <Type Value=""Yes""/>
  </Child1>
  <Child2>
    <Text>Child Two Text</Text>
    <Type Value=""Yes""/>
  </Child2>
  <Child3>
    <Text>Child Three Text</Text>
    <Type Value=""No""/>
  </Child3>
  <Child4>
    <Text>Child Four Text</Text>
    <Type Value=""Yes""/>
  </Child4>
  <Child5>
    <Text>Child Five Text</Text>
  </Child5>
</Root>");
XNamespace ad = "http://www.adatum.com";
var cList =
    from typeElement in root.Elements().Elements(ad + "Type")
    where (string)typeElement.Attribute("Value") == "Yes"
    select (string)typeElement.Parent.Element(ad + "Text");
foreach (string str in cList)
    Console.WriteLine(str);

```

This code produces the following output:

```

Child One Text
Child Two Text
Child Four Text

```

## See also

- [XElement.Attribute](#)
- [XContainer.Elements](#)
- [Extensions.Elements](#)
- [Standard Query Operators Overview \(C#\)](#)
- [Projection Operations \(C#\)](#)

# How to: Find All Nodes in a Namespace (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

You can filter on the namespace of each element or attribute to find all nodes in that particular namespace.

## Example

The following example creates an XML tree with two namespaces. It then iterates through the tree and prints the names of all the elements and attributes in one of those namespaces.

```
string markup = @"<aw:Root xmlns:aw='http://www.adventure-works.com' xmlns:fc='www.fourthcoffee.com'>
  <fc:Child1>abc</fc:Child1>
  <fc:Child2>def</fc:Child2>
  <aw:Child3>ghi</aw:Child3>
  <fc:Child4>
    <fc:GrandChild1>jkl</fc:GrandChild1>
    <aw:GrandChild2>mno</aw:GrandChild2>
  </fc:Child4>
</aw:Root>";
XElement xmlTree = XElement.Parse(markup);
Console.WriteLine("Nodes in the http://www.adventure-works.com namespace");
IEnumerable<XElement> awElements =
    from el in xmlTree.Descendants()
    where el.Name.Namespace == "http://www.adventure-works.com"
    select el;
foreach (XElement el in awElements)
    Console.WriteLine(el.Name.ToString());
```

This code produces the following output:

```
Nodes in the http://www.adventure-works.com namespace
{http://www.adventure-works.com}Child3
{http://www.adventure-works.com}GrandChild2
```

## Example

The XML file accessed by the following query contains purchase orders in two different namespaces. The query creates a new tree with just the elements in one of the namespaces.

This example uses the following XML document: [Sample XML File: Consolidated Purchase Orders](#).

```
XDocument cpo = XDocument.Load("ConsolidatedPurchaseOrders.xml");
XNamespace aw = "http://www.adventure-works.com";
XElement newTree = new XElement("Root",
    from el in cpo.Root.Elements()
    where el.Name.Namespace == aw
    select el
);
Console.WriteLine(newTree);
```

This code produces the following output:

```
<Root>
  <aw:PurchaseOrder PONumber="11223" Date="2000-01-15" xmlns:aw="http://www.adventure-works.com">
    <aw:ShippingAddress>
      <aw:Name>Chris Preston</aw:Name>
      <aw:Street>123 Main St.</aw:Street>
      <aw:City>Seattle</aw:City>
      <aw:State>WA</aw:State>
      <aw:Zip>98113</aw:Zip>
      <aw:Country>USA</aw:Country>
    </aw:ShippingAddress>
    <aw:BillingAddress>
      <aw:Name>Chris Preston</aw:Name>
      <aw:Street>123 Main St.</aw:Street>
      <aw:City>Seattle</aw:City>
      <aw:State>WA</aw:State>
      <aw:Zip>98113</aw:Zip>
      <aw:Country>USA</aw:Country>
    </aw:BillingAddress>
    <aw:DeliveryInstructions>Ship only complete order.</aw:DeliveryInstructions>
    <aw:Item PartNum="LIT-01">
      <aw:ProductID>Litware Networking Card</aw:ProductID>
      <aw:Qty>1</aw:Qty>
      <aw:Price>20.99</aw:Price>
    </aw:Item>
    <aw:Item PartNum="LIT-25">
      <aw:ProductID>Litware 17in LCD Monitor</aw:ProductID>
      <aw:Qty>1</aw:Qty>
      <aw:Price>199.99</aw:Price>
    </aw:Item>
  </aw:PurchaseOrder>
</Root>
```

# How to: Sort Elements (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to write a query that sorts its results.

## Example

This example uses the following XML document: [Sample XML File: Numerical Data \(LINQ to XML\)](#).

```
XElement root = XElement.Load("Data.xml");
IEnumerable<decimal> prices =
    from el in root.Elements("Data")
    let price = (decimal)el.Element("Price")
    orderby price
    select price;
foreach (decimal el in prices)
    Console.WriteLine(el);
```

This code produces the following output:

```
0.99
4.95
6.99
24.50
29.00
66.00
89.99
```

## Example

The following example shows the same query for XML that is in a namespace. For more information, see [Working with XML Namespaces \(C#\)](#).

This example uses the following XML document: [Sample XML File: Numerical Data in a Namespace](#).

```
XElement root = XElement.Load("DataInNamespace.xml");
XNamespace aw = "http://www.adatum.com";
IEnumerable<decimal> prices =
    from el in root.Elements(aw + "Data")
    let price = (decimal)el.Element(aw + "Price")
    orderby price
    select price;
foreach (decimal el in prices)
    Console.WriteLine(el);
```

This code produces the following output:

```
0.99
4.95
6.99
24.50
29.00
66.00
89.99
```

## See also

- [Sorting Data \(C#\)](#)

# How to: Sort Elements on Multiple Keys (C#)

6/5/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how to sort on multiple keys.

## Example

In this example, the results are ordered first by the shipping postal code, then by the order date.

This example uses the following XML document: [Sample XML File: Customers and Orders \(LINQ to XML\)](#).

```
XElement co = XElement.Load("CustomersOrders.xml");
var sortedElements =
    from c in co.Element("Orders").Elements("Order")
    orderby (string)c.Element("ShipInfo").Element("ShipPostalCode"),
            (DateTime)c.Element("OrderDate")
    select new {
        CustomerID = (string)c.Element("CustomerID"),
        EmployeeID = (string)c.Element("EmployeeID"),
        ShipPostalCode = (string)c.Element("ShipInfo").Element("ShipPostalCode"),
        OrderDate = (DateTime)c.Element("OrderDate")
    };
foreach (var r in sortedElements)
    Console.WriteLine("CustomerID:{0} EmployeeID:{1} ShipPostalCode:{2} OrderDate:{3:d}",
        r.CustomerID, r.EmployeeID, r.ShipPostalCode, r.OrderDate);
```

This code produces the following output:

```
CustomerID:LETSS EmployeeID:1 ShipPostalCode:94117 OrderDate:6/25/1997
CustomerID:LETSS EmployeeID:8 ShipPostalCode:94117 OrderDate:10/27/1997
CustomerID:LETSS EmployeeID:6 ShipPostalCode:94117 OrderDate:11/10/1997
CustomerID:LETSS EmployeeID:4 ShipPostalCode:94117 OrderDate:2/12/1998
CustomerID:GREAL EmployeeID:6 ShipPostalCode:97403 OrderDate:5/6/1997
CustomerID:GREAL EmployeeID:8 ShipPostalCode:97403 OrderDate:7/4/1997
CustomerID:GREAL EmployeeID:1 ShipPostalCode:97403 OrderDate:7/31/1997
CustomerID:GREAL EmployeeID:4 ShipPostalCode:97403 OrderDate:7/31/1997
CustomerID:GREAL EmployeeID:6 ShipPostalCode:97403 OrderDate:9/4/1997
CustomerID:GREAL EmployeeID:3 ShipPostalCode:97403 OrderDate:9/25/1997
CustomerID:GREAL EmployeeID:4 ShipPostalCode:97403 OrderDate:1/6/1998
CustomerID:GREAL EmployeeID:3 ShipPostalCode:97403 OrderDate:3/9/1998
CustomerID:GREAL EmployeeID:3 ShipPostalCode:97403 OrderDate:4/7/1998
CustomerID:GREAL EmployeeID:4 ShipPostalCode:97403 OrderDate:4/22/1998
CustomerID:GREAL EmployeeID:4 ShipPostalCode:97403 OrderDate:4/30/1998
CustomerID:HUNGC EmployeeID:3 ShipPostalCode:97827 OrderDate:12/6/1996
CustomerID:HUNGC EmployeeID:1 ShipPostalCode:97827 OrderDate:12/25/1996
CustomerID:HUNGC EmployeeID:3 ShipPostalCode:97827 OrderDate:1/15/1997
CustomerID:HUNGC EmployeeID:4 ShipPostalCode:97827 OrderDate:7/16/1997
CustomerID:HUNGC EmployeeID:8 ShipPostalCode:97827 OrderDate:9/8/1997
CustomerID:LAZYK EmployeeID:1 ShipPostalCode:99362 OrderDate:3/21/1997
CustomerID:LAZYK EmployeeID:8 ShipPostalCode:99362 OrderDate:5/22/1997
```

## Example

The following example shows the same query for XML that is in a namespace. For more information, see [Working with XML Namespaces \(C#\)](#).

This example uses the following XML document: [Sample XML File: Customers and Orders in a Namespace](#).

```
XElement co = XElement.Load("CustomersOrdersInNamespace.xml");
XNamespace aw = "http://www.adventure-works.com";
var sortedElements =
    from c in co.Element(aw + "Orders").Elements(aw + "Order")
    orderby (string)c.Element(aw + "ShipInfo").Element(aw + "ShipPostalCode"),
            (DateTime)c.Element(aw + "OrderDate")
    select new
    {
        CustomerID = (string)c.Element(aw + "CustomerID"),
        EmployeeID = (string)c.Element(aw + "EmployeeID"),
        ShipPostalCode = (string)c.Element(aw + "ShipInfo").Element(aw + "ShipPostalCode"),
        OrderDate = (DateTime)c.Element(aw + "OrderDate")
    };
foreach (var r in sortedElements)
    Console.WriteLine("CustomerID:{0} EmployeeID:{1} ShipPostalCode:{2} OrderDate:{3:d}",
        r.CustomerID, r.EmployeeID, r.ShipPostalCode, r.OrderDate);
```

This code produces the following output:

```
CustomerID:LETSS EmployeeID:1 ShipPostalCode:94117 OrderDate:6/25/1997
CustomerID:LETSS EmployeeID:8 ShipPostalCode:94117 OrderDate:10/27/1997
CustomerID:LETSS EmployeeID:6 ShipPostalCode:94117 OrderDate:11/10/1997
CustomerID:LETSS EmployeeID:4 ShipPostalCode:94117 OrderDate:2/12/1998
CustomerID:GREAL EmployeeID:6 ShipPostalCode:97403 OrderDate:5/6/1997
CustomerID:GREAL EmployeeID:8 ShipPostalCode:97403 OrderDate:7/4/1997
CustomerID:GREAL EmployeeID:1 ShipPostalCode:97403 OrderDate:7/31/1997
CustomerID:GREAL EmployeeID:4 ShipPostalCode:97403 OrderDate:7/31/1997
CustomerID:GREAL EmployeeID:6 ShipPostalCode:97403 OrderDate:9/4/1997
CustomerID:GREAL EmployeeID:3 ShipPostalCode:97403 OrderDate:9/25/1997
CustomerID:GREAL EmployeeID:4 ShipPostalCode:97403 OrderDate:1/6/1998
CustomerID:GREAL EmployeeID:3 ShipPostalCode:97403 OrderDate:3/9/1998
CustomerID:GREAL EmployeeID:3 ShipPostalCode:97403 OrderDate:4/7/1998
CustomerID:GREAL EmployeeID:4 ShipPostalCode:97403 OrderDate:4/22/1998
CustomerID:GREAL EmployeeID:4 ShipPostalCode:97403 OrderDate:4/30/1998
CustomerID:HUNGC EmployeeID:3 ShipPostalCode:97827 OrderDate:12/6/1996
CustomerID:HUNGC EmployeeID:1 ShipPostalCode:97827 OrderDate:12/25/1996
CustomerID:HUNGC EmployeeID:3 ShipPostalCode:97827 OrderDate:1/15/1997
CustomerID:HUNGC EmployeeID:4 ShipPostalCode:97827 OrderDate:7/16/1997
CustomerID:HUNGC EmployeeID:8 ShipPostalCode:97827 OrderDate:9/8/1997
CustomerID:LAZYK EmployeeID:1 ShipPostalCode:99362 OrderDate:3/21/1997
CustomerID:LAZYK EmployeeID:8 ShipPostalCode:99362 OrderDate:5/22/1997
```

# How to: Calculate Intermediate Values (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to calculate intermediate values that can be used in sorting, filtering, and selecting.

## Example

The following example uses the `Let` clause.

This example uses the following XML document: [Sample XML File: Numerical Data \(LINQ to XML\)](#).

```
XElement root = XElement.Load("Data.xml");
IEnumerable<decimal> extensions =
    from el in root.Elements("Data")
    let extension = (decimal)el.Element("Quantity") * (decimal)el.Element("Price")
    where extension >= 25
    orderby extension
    select extension;
foreach (decimal ex in extensions)
    Console.WriteLine(ex);
```

This code produces the following output:

```
55.92
73.50
89.99
198.00
435.00
```

## Example

The following example shows the same query for XML that is in a namespace. For more information, see [Working with XML Namespaces \(C#\)](#).

This example uses the following XML document: [Sample XML File: Numerical Data in a Namespace](#).

```
XElement root = XElement.Load("DataInNamespace.xml");
XNamespace ad = "http://www.adatum.com";
IEnumerable<decimal> extensions =
    from el in root.Elements(ad + "Data")
    let extension = (decimal)el.Element(ad + "Quantity") * (decimal)el.Element(ad + "Price")
    where extension >= 25
    orderby extension
    select extension;
foreach (decimal ex in extensions)
    Console.WriteLine(ex);
```

This code produces the following output:



55.92

73.50

89.99

198.00

435.00

# How to: Write a Query that Finds Elements Based on Context (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

Sometimes you might have to write a query that selects elements based on their context. You might want to filter based on preceding or following sibling elements. You might want to filter based on child or ancestor elements.

You can do this by writing a query and using the results of the query in the `where` clause. If you have to first test against null, and then test the value, it is more convenient to do the query in a `let` clause, and then use the results in the `where` clause.

## Example

The following example selects all `p` elements that are immediately followed by a `ul` element.

```
XElement doc = XElement.Parse(@"<Root>
  <p id=""1""/>
  <ul>abc</ul>
  <Child>
    <p id=""2""/>
    <notul/>
    <p id=""3""/>
    <ul>def</ul>
    <p id=""4""/>
  </Child>
  <Child>
    <p id=""5""/>
    <notul/>
    <p id=""6""/>
    <ul>abc</ul>
    <p id=""7""/>
  </Child>
</Root>");

IEnumerable<XElement> items =
    from e in doc.Descendants("p")
    let z = e.ElementsAfterSelf().FirstOrDefault()
    where z != null && z.Name.LocalName == "ul"
    select e;

foreach (XElement e in items)
    Console.WriteLine("id = {0}", (string)e.Attribute("id"));
```

This code produces the following output:

```
id = 1
id = 3
id = 6
```

## Example

The following example shows the same query for XML that is in a namespace. For more information, see [Working with XML Namespaces \(C#\)](#).

```

XElement doc = XElement.Parse(@"<Root xmlns='http://www.adatum.com'>
  <p id=""1""/>
  <ul>abc</ul>
  <Child>
    <p id=""2""/>
    <notul/>
    <p id=""3""/>
    <ul>def</ul>
    <p id=""4""/>
  </Child>
  <Child>
    <p id=""5""/>
    <notul/>
    <p id=""6""/>
    <ul>abc</ul>
    <p id=""7""/>
  </Child>
</Root>");

XNamespace ad = "http://www.adatum.com";

IEnumerable<XElement> items =
  from e in doc.Descendants(ad + "p")
  let z = e.ElementsAfterSelf().FirstOrDefault()
  where z != null && z.Name == ad.GetName("ul")
  select e;

foreach (XElement e in items)
  Console.WriteLine("id = {0}", (string)e.Attribute("id"));

```

This code produces the following output:

```

id = 1
id = 3
id = 6

```

## See also

- [Parse](#)
- [Descendants](#)
- [ElementsAfterSelf](#)
- [FirstOrDefault](#)

# How to: Debug Empty Query Results Sets (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

One of the most common problems when querying XML trees is that if the XML tree has a default namespace, the developer sometimes writes the query as though the XML were not in a namespace.

The first set of examples in this topic shows a typical way that XML in a default namespace is loaded, and is queried improperly.

The second set of examples show the necessary corrections so that you can query XML in a namespace.

For more information, see [Working with XML Namespaces \(C#\)](#).

## Example

This example shows creation of XML in a namespace, and a query that returns an empty result set.

```
XElement root = XElement.Parse(
@"<Root xmlns='http://www.adventure-works.com'>
  <Child>1</Child>
  <Child>2</Child>
  <Child>3</Child>
  <AnotherChild>4</AnotherChild>
  <AnotherChild>5</AnotherChild>
  <AnotherChild>6</AnotherChild>
</Root>");
IEnumerable<XElement> c1 =
    from el in root.Elements("Child")
    select el;
Console.WriteLine("Result set follows:");
foreach (XElement el in c1)
    Console.WriteLine((int)el);
Console.WriteLine("End of result set");
```

This example produces the following result:

```
Result set follows:
End of result set
```

## Example

This example shows creation of XML in a namespace, and a query that is coded properly.

The solution is to declare and initialize an [XNamespace](#) object, and to use it when specifying [XName](#) objects. In this case, the argument to the [Elements](#) method is an [XName](#) object.

```
XElement root = XElement.Parse(
@"<Root xmlns='http://www.adventure-works.com'>
  <Child>1</Child>
  <Child>2</Child>
  <Child>3</Child>
  <AnotherChild>4</AnotherChild>
  <AnotherChild>5</AnotherChild>
  <AnotherChild>6</AnotherChild>
</Root>");
XNamespace aw = "http://www.adventure-works.com";
IEnumerable<XElement> c1 =
    from el in root.Elements(aw + "Child")
    select el;
Console.WriteLine("Result set follows:");
foreach (XElement el in c1)
    Console.WriteLine((int)el);
Console.WriteLine("End of result set");
```

This example produces the following result:

```
Result set follows:
1
2
3
End of result set
```

# How to: Work with Dictionaries Using LINQ to XML (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

It is often convenient to convert varieties of data structures to XML, and XML back to other data structures. This topic shows a specific implementation of this general approach by converting a `Dictionary<TKey,TValue>` to XML and back.

## Example

This example uses a form of functional construction in which a query projects new `XElement` objects, and the resulting collection is passed as an argument to the constructor of the Root `XElement` object.

```
Dictionary<string, string> dict = new Dictionary<string, string>();
dict.Add("Child1", "Value1");
dict.Add("Child2", "Value2");
dict.Add("Child3", "Value3");
dict.Add("Child4", "Value4");
XElement root = new XElement("Root",
    from keyValue in dict
    select new XElement(keyValue.Key, keyValue.Value)
);
Console.WriteLine(root);
```

This code produces the following output:

```
<Root>
  <Child1>Value1</Child1>
  <Child2>Value2</Child2>
  <Child3>Value3</Child3>
  <Child4>Value4</Child4>
</Root>
```

## Example

The following code creates a dictionary from XML.

```
XElement root = new XElement("Root",
    new XElement("Child1", "Value1"),
    new XElement("Child2", "Value2"),
    new XElement("Child3", "Value3"),
    new XElement("Child4", "Value4")
);

Dictionary<string, string> dict = new Dictionary<string, string>();
foreach (XElement el in root.Elements())
    dict.Add(el.Name.LocalName, el.Value);
foreach (string str in dict.Keys)
    Console.WriteLine("{0}:{1}", str, dict[str]);
```

This code produces the following output:

Child1:Value1  
Child2:Value2  
Child3:Value3  
Child4:Value4

# How to: Transform the Shape of an XML Tree (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

The *shape* of an XML document refers to its element names, attribute names, and the characteristics of its hierarchy.

Sometimes you will have to change the shape of an XML document. For example, you might have to send an existing XML document to another system that requires different element and attribute names. You could go through the document, deleting and renaming elements as required, but using functional construction results in more readable and maintainable code. For more information about functional construction, see [Functional Construction \(LINQ to XML\) \(C#\)](#).

The first example changes the organization of the XML document. It moves complex elements from one location in the tree to another.

The second example in this topic creates an XML document with a different shape than the source document. It changes the casing of the element names, renames some elements, and leaves some elements from the source tree out of the transformed tree.

## Example

The following code changes the shape of an XML file using embedded query expressions.

The source XML document in this example contains a `Customers` element under the `Root` element that contains all customers. It also contains an `Orders` element under the `Root` element that contains all orders. This example creates a new XML tree in which the orders for each customer are contained in an `Orders` element within the `Customer` element. The original document also contains a `CustomerID` element in the `Order` element; this element will be removed from the re-shaped document.

This example uses the following XML document: [Sample XML File: Customers and Orders \(LINQ to XML\)](#).

```
XElement co = XElement.Load("CustomersOrders.xml");
XElement newCustOrd =
    new XElement("Root",
        from cust in co.Element("Customers").Elements("Customer")
        select new XElement("Customer",
            cust.Attributes(),
            cust.Elements(),
            new XElement("Orders",
                from ord in co.Element("Orders").Elements("Order")
                where (string)ord.Element("CustomerID") == (string)cust.Attribute("CustomerID")
                select new XElement("Order",
                    ord.Attributes(),
                    ord.Element("EmployeeID"),
                    ord.Element("OrderDate"),
                    ord.Element("RequiredDate"),
                    ord.Element("ShipInfo")
                )
            )
        )
    );
Console.WriteLine(newCustOrd);
```

This code produces the following output:



```

<Root>
  <Customer CustomerID="GREAL">
    <CompanyName>Great Lakes Food Market</CompanyName>
    <ContactName>Howard Snyder</ContactName>
    <ContactTitle>Marketing Manager</ContactTitle>
    <Phone>(503) 555-7555</Phone>
    <FullAddress>
      <Address>2732 Baker Blvd.</Address>
      <City>Eugene</City>
      <Region>OR</Region>
      <PostalCode>97403</PostalCode>
      <Country>USA</Country>
    </FullAddress>
    <Orders />
  </Customer>
  <Customer CustomerID="HUNG">
    <CompanyName>Hungry Coyote Import Store</CompanyName>
    <ContactName>Yoshi Latimer</ContactName>
    <ContactTitle>Sales Representative</ContactTitle>
    <Phone>(503) 555-6874</Phone>
    <Fax>(503) 555-2376</Fax>
    <FullAddress>
      <Address>City Center Plaza 516 Main St.</Address>
      <City>Elgin</City>
      <Region>OR</Region>
      <PostalCode>97827</PostalCode>
      <Country>USA</Country>
    </FullAddress>
    <Orders />
  </Customer>
  . . .

```

## Example

This example renames some elements and converts some attributes to elements.

The code calls `ConvertAddress`, which returns a list of `XElement` objects. The argument to the method is a query that determines the `Address` complex element where the `Type` attribute has a value of `"Shipping"`.

This example uses the following XML document: [Sample XML File: Typical Purchase Order \(LINQ to XML\)](#).

```

static IEnumerable<XElement> ConvertAddress(XElement add)
{
    List<XElement> fragment = new List<XElement>() {
        new XElement("NAME", (string)add.Element("Name")),
        new XElement("STREET", (string)add.Element("Street")),
        new XElement("CITY", (string)add.Element("City")),
        new XElement("ST", (string)add.Element("State")),
        new XElement("POSTALCODE", (string)add.Element("Zip")),
        new XElement("COUNTRY", (string)add.Element("Country"))
    };
    return fragment;
}

static void Main(string[] args)
{
    XElement po = XElement.Load("PurchaseOrder.xml");
    XElement newPo = new XElement("PO",
        new XElement("ID", (string)po.Attribute("PurchaseOrderNumber")),
        new XElement("DATE", (DateTime)po.Attribute("OrderDate")),
        ConvertAddress(
            (from el in po.Elements("Address")
             where (string)el.Attribute("Type") == "Shipping"
             select el)
            .First()
        )
    );
    Console.WriteLine(newPo);
}

```

This code produces the following output:

```

<PO>
  <ID>99503</ID>
  <DATE>1999-10-20T00:00:00</DATE>
  <NAME>Ellen Adams</NAME>
  <STREET>123 Maple Street</STREET>
  <CITY>Mill Valley</CITY>
  <ST>CA</ST>
  <POSTALCODE>10999</POSTALCODE>
  <COUNTRY>USA</COUNTRY>
</PO>

```

# How to: Control the Type of a Projection (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

Projection is the process of taking one set of data, filtering it, changing its shape, and even changing its type. Most query expressions perform projections. Most of the query expressions shown in this section evaluate to `IEnumerable<T>` of `XElement`, but you can control the type of the projection to create collections of other types. This topic shows how to do this.

## Example

The following example defines a new type, `Customer`. The query expression then instantiates new `Customer` objects in the `Select` clause. This causes the type of the query expression to be `IEnumerable<T>` of `Customer`.

This example uses the following XML document: [Sample XML File: Customers and Orders \(LINQ to XML\)](#).

```
public class Customer
{
    private string customerID;
    public string CustomerID{ get{return customerID;} set{customerID = value;}}

    private string companyName;
    public string CompanyName{ get{return companyName;} set{companyName = value;}}

    private string contactName;
    public string ContactName { get{return contactName;} set{contactName = value;}}

    public Customer(string customerID, string companyName, string contactName)
    {
        CustomerID = customerID;
        CompanyName = companyName;
        ContactName = contactName;
    }

    public override string ToString()
    {
        return $"{this.customerID}:{this.companyName}:{this.contactName}";
    }
}

class Program
{
    static void Main(string[] args)
    {
        XElement custOrd = XElement.Load("CustomersOrders.xml");
        IEnumerable<Customer> custList =
            from el in custOrd.Element("Customers").Elements("Customer")
            select new Customer(
                (string)el.Attribute("CustomerID"),
                (string)el.Element("CompanyName"),
                (string)el.Element("ContactName")
            );
        foreach (Customer cust in custList)
            Console.WriteLine(cust);
    }
}
```

This code produces the following output:

GREAL:Great Lakes Food Market:Howard Snyder  
HUNGC:Hungry Coyote Import Store:Yoshi Latimer  
LAZYK:Lazy K Kountry Store:John Steel  
LETSS:Let's Stop N Shop:Jaime Yorres

## See also

- [Select](#)

# How to: Project a New Type (LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

Other examples in this section have shown queries that return results as `IEnumerable<T>` of `XElement`, `IEnumerable<T>` of `string`, and `IEnumerable<T>` of `int`. These are common result types, but they are not appropriate for every scenario. In many cases you will want your queries to return an `IEnumerable<T>` of some other type.

## Example

This example shows how to instantiate objects in the `select` clause. The code first defines a new class with a constructor, and then modifies the `select` statement so that the expression is a new instance of the new class.

This example uses the following XML document: [Sample XML File: Typical Purchase Order \(LINQ to XML\)](#).

```
class NameQty {
    public string name;
    public int qty;
    public NameQty(string n, int q)
    {
        name = n;
        qty = q;
    }
};

class Program {
    public static void Main() {
        XElement po = XElement.Load("PurchaseOrder.xml");

        IEnumerable<NameQty> nqList =
            from n in po.Descendants("Item")
            select new NameQty(
                (string)n.Element("ProductName"),
                (int)n.Element("Quantity")
            );

        foreach (NameQty n in nqList)
            Console.WriteLine(n.name + ":" + n.qty);
    }
}
```

This example uses the `M:System.Xml.Linq.XElement.Element` method that was introduced in the topic [How to: Retrieve a Single Child Element \(LINQ to XML\) \(C#\)](#). It also uses casts to retrieve the values of the elements that are returned by the `M:System.Xml.Linq.XElement.Element` method.

This example produces the following output:

```
Lawnmower:1
Baby Monitor:2
```

# How to: Project an Object Graph (C#)

6/4/2019 • 3 minutes to read • [Edit Online](#)

This topic illustrates how to project, or populate, an object graph from XML.

## Example

The following code populates an object graph with the `Address`, `PurchaseOrder`, and `PurchaseOrderItem` classes from the [Sample XML File: Typical Purchase Order \(LINQ to XML\)](#) XML document.

```
class Address
{
    public enum AddressUse
    {
        Shipping,
        Billing,
    }

    private AddressUse addressType;
    private string name;
    private string street;
    private string city;
    private string state;
    private string zip;
    private string country;

    public AddressUse AddressType {
        get { return addressType; } set { addressType = value; }
    }

    public string Name {
        get { return name; } set { name = value; }
    }

    public string Street {
        get { return street; } set { street = value; }
    }

    public string City {
        get { return city; } set { city = value; }
    }

    public string State {
        get { return state; } set { state = value; }
    }

    public string Zip {
        get { return zip; } set { zip = value; }
    }

    public string Country {
        get { return country; } set { country = value; }
    }

    public override string ToString()
    {
        StringBuilder sb = new StringBuilder();
        sb.Append($"Type: {(addressType == AddressUse.Shipping ? "Shipping" : "Billing")}\n");
        sb.Append($"Name: {name}\n");
        sb.Append($"Street: {street}\n");
        sb.Append($"City: {city}\n");
    }
}
```

```

        sb.Append($"State: {state}\n");
        sb.Append($"Zip: {zip}\n");
        sb.Append($"Country: {country}\n");
        return sb.ToString();
    }
}

class PurchaseOrderItem
{
    private string partNumber;
    private string productName;
    private int quantity;
    private Decimal usPrice;
    private string comment;
    private DateTime shipDate;

    public string PartNumber {
        get { return partNumber; } set { partNumber = value; }
    }

    public string ProductName {
        get { return productName; } set { productName = value; }
    }

    public int Quantity {
        get { return quantity; } set { quantity = value; }
    }

    public Decimal USPrice {
        get { return usPrice; } set { usPrice = value; }
    }

    public string Comment {
        get { return comment; } set { comment = value; }
    }

    public DateTime ShipDate {
        get { return shipDate; } set { shipDate = value; }
    }

    public override string ToString()
    {
        StringBuilder sb = new StringBuilder();
        sb.Append($"PartNumber: {partNumber}\n");
        sb.Append($"ProductName: {productName}\n");
        sb.Append($"Quantity: {quantity}\n");
        sb.Append($"USPrice: {usPrice}\n");
        if (comment != null)
            sb.Append($"Comment: {comment}\n");
        if (shipDate != DateTime.MinValue)
            sb.Append($"ShipDate: {shipDate:d}\n");
        return sb.ToString();
    }
}

class PurchaseOrder
{
    private string purchaseOrderNumber;
    private DateTime orderDate;
    private string comment;
    private List<Address> addresses;
    private List<PurchaseOrderItem> items;

    public string PurchaseOrderNumber {
        get { return purchaseOrderNumber; } set { purchaseOrderNumber = value; }
    }

    public DateTime OrderDate {
        get { return orderDate; } set { orderDate = value; }
    }
}

```





```

                (DateTime)i.Element( "ShipDate" ) :
                DateTime.MinValue
            }
        }).ToList()
    };
    Console.WriteLine(purchaseOrder);
}
}

```

In this example, the result of the LINQ query is returned as an [IEnumerable<T>](#) of `PurchaseOrderItem`. The items in the `PurchaseOrder` class are of type [IEnumerable<T>](#) of `PurchaseOrderItem`. The code uses the [ToList](#) extension method to create a [List<T>](#) collection from the results of the query.

The example produces the following output:

```

PurchaseOrderNumber: 99503
OrderDate: 10/20/1999

Addresses
=====
Type: Shipping
Name: Ellen Adams
Street: 123 Maple Street
City: Mill Valley
State: CA
Zip: 10999
Country: USA

Type: Billing
Name: Tai Yee
Street: 8 Oak Avenue
City: Old Town
State: PA
Zip: 95819
Country: USA

Items
=====
PartNumber: 872-AA
ProductName: Lawnmower
Quantity: 1
USPrice: 148.95
Comment: Confirm this is electric

PartNumber: 926-AA
ProductName: Baby Monitor
Quantity: 2
USPrice: 39.98
ShipDate: 5/21/1999

```

## See also

- [Select](#)
- [ToList](#)

# How to: Project an Anonymous Type (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

In some cases you might want to project a query to a new type, even though you know you will only use this type for a short while. It is a lot of extra work to create a new type just to use in the projection. A more efficient approach in this case is to project to an anonymous type. Anonymous types allow you to define a class, then declare and initialize an object of that class, without giving the class a name.

Anonymous types are the C# implementation of the mathematical concept of a *tuple*. The mathematical term tuple originated from the sequence single, double, triple, quadruple, quintuple, n-tuple. It refers to a finite sequence of objects, each of a specific type. Sometimes this is called a list of name/value pairs. For example, the contents of an address in the [Sample XML File: Typical Purchase Order \(LINQ to XML\)](#) XML document could be expressed as follows:

```
Name: Ellen Adams
Street: 123 Maple Street
City: Mill Valley
State: CA
Zip: 90952
Country: USA
```

When you create an instance of an anonymous type, it is convenient to think of it as creating a tuple of order n. If you write a query that creates a tuple in the `select` clause, the query returns an `IEnumerable` of the tuple.

## Example

In this example, the `select` clause projects an anonymous type. The example then uses `var` to create the `IEnumerable` object. Within the `foreach` loop, the iteration variable becomes an instance of the anonymous type created in the query expression.

This example uses the following XML document: [Sample XML File: Customers and Orders \(LINQ to XML\)](#).

```
XElement custOrd = XElement.Load("CustomersOrders.xml");
var custList =
    from el in custOrd.Element("Customers").Elements("Customer")
    select new {
        CustomerID = (string)el.Attribute("CustomerID"),
        CompanyName = (string)el.Element("CompanyName"),
        ContactName = (string)el.Element("ContactName")
    };
foreach (var cust in custList)
    Console.WriteLine("{0}:{1}:{2}", cust.CustomerID, cust.CompanyName, cust.ContactName);
```

This code produces the following output:

```
GREAL:Great Lakes Food Market:Howard Snyder
HUNGC:Hungry Coyote Import Store:Yoshi Latimer
LAZYK:Lazy K Kountry Store:John Steel
LETSS:Let's Stop N Shop:Jaime Yorres
```

# How to: Generate Text Files from XML (C#)

1/23/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to generate a comma-separated values (CSV) file from an XML file.

## Example

The C# version of this example uses method syntax and the `Aggregate` operator to generate a CSV file from an XML document in a single expression. For more information, see [Query Syntax and Method Syntax in LINQ](#).

This example uses the following XML document: [Sample XML File: Customers and Orders \(LINQ to XML\)](#).

```
XElement custOrd = XElement.Load("CustomersOrders.xml");
string csv =
    (from el in custOrd.Element("Customers").Elements("Customer")
     select
         String.Format("{0},{1},{2},{3},{4},{5},{6},{7},{8},{9}{10}",
             (string)el.Attribute("CustomerID"),
             (string)el.Element("CompanyName"),
             (string)el.Element("ContactName"),
             (string)el.Element("ContactTitle"),
             (string)el.Element("Phone"),
             (string)el.Element("FullAddress").Element("Address"),
             (string)el.Element("FullAddress").Element("City"),
             (string)el.Element("FullAddress").Element("Region"),
             (string)el.Element("FullAddress").Element("PostalCode"),
             (string)el.Element("FullAddress").Element("Country"),
             Environment.NewLine
         )
    )
    .Aggregate(
        new StringBuilder(),
        (sb, s) => sb.Append(s),
        sb => sb.ToString()
    );
Console.WriteLine(csv);
```

This code produces the following output:

```
GREAL,Great Lakes Food Market,Howard Snyder,Marketing Manager,(503) 555-7555,2732 Baker
Blvd.,Eugene,OR,97403,USA
HUNGC,Hungry Coyote Import Store,Yoshi Latimer,Sales Representative,(503) 555-6874,City Center Plaza 516 Main
St.,Elgin,OR,97827,USA
LAZYK,Lazy K Kountry Store,John Steel,Marketing Manager,(509) 555-7969,12 Orchestra Terrace,Walla
Walla,WA,99362,USA
LETSS,Let's Stop N Shop,Jaime Yorres,Owner,(415) 555-5938,87 Polk St. Suite 5,San Francisco,CA,94117,USA
```

## See also

- [Projections and Transformations \(LINQ to XML\) \(C#\)](#)

# How to: Generate XML from CSV Files (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to use Language-Integrated Query (LINQ) and LINQ to XML to generate an XML file from a comma-separated value (CSV) file.

## Example

The following code performs a LINQ query on an array of strings.

The query uses the `let` clause to split each string into an array of fields.

```
// Create the text file.
string csvString = @"GREAL,Great Lakes Food Market,Howard Snyder,Marketing Manager,(503) 555-7555,2732 Baker
Blvd.,Eugene,OR,97403,USA
HUNGC,Hungry Coyote Import Store,Yoshi Latimer,Sales Representative,(503) 555-6874,City Center Plaza 516 Main
St.,Elgin,OR,97827,USA
LAZYK,Lazy K Kountry Store,John Steel,Marketing Manager,(509) 555-7969,12 Orchestra Terrace,Walla
Walla,WA,99362,USA
LETSS,Let's Stop N Shop,Jaime Yorres,Owner,(415) 555-5938,87 Polk St. Suite 5,San Francisco,CA,94117,USA";
File.WriteAllText("cust.csv", csvString);

// Read into an array of strings.
string[] source = File.ReadAllLines("cust.csv");
XElement cust = new XElement("Root",
    from str in source
    let fields = str.Split(',')
    select new XElement("Customer",
        new XAttribute("CustomerID", fields[0]),
        new XElement("CompanyName", fields[1]),
        new XElement("ContactName", fields[2]),
        new XElement("ContactTitle", fields[3]),
        new XElement("Phone", fields[4]),
        new XElement("FullAddress",
            new XElement("Address", fields[5]),
            new XElement("City", fields[6]),
            new XElement("Region", fields[7]),
            new XElement("PostalCode", fields[8]),
            new XElement("Country", fields[9])
        )
    )
);
Console.WriteLine(cust);
```

This code produces the following output:

```
<Root>
  <Customer CustomerID="GREAL">
    <CompanyName>Great Lakes Food Market</CompanyName>
    <ContactName>Howard Snyder</ContactName>
    <ContactTitle>Marketing Manager</ContactTitle>
    <Phone>(503) 555-7555</Phone>
    <FullAddress>
      <Address>2732 Baker Blvd.</Address>
      <City>Eugene</City>
      <Region>OR</Region>
      <PostalCode>97403</PostalCode>
      <Country>USA</Country>
    </FullAddress>
  </Customer>
  <Customer CustomerID="HUNGC">
    <CompanyName>Hungry Coyote Import Store</CompanyName>
    <ContactName>Yoshi Latimer</ContactName>
    <ContactTitle>Sales Representative</ContactTitle>
    <Phone>(503) 555-6874</Phone>
    <FullAddress>
      <Address>City Center Plaza 516 Main St.</Address>
      <City>Elgin</City>
      <Region>OR</Region>
      <PostalCode>97827</PostalCode>
      <Country>USA</Country>
    </FullAddress>
  </Customer>
  <Customer CustomerID="LAZYK">
    <CompanyName>Lazy K Kountry Store</CompanyName>
    <ContactName>John Steel</ContactName>
    <ContactTitle>Marketing Manager</ContactTitle>
    <Phone>(509) 555-7969</Phone>
    <FullAddress>
      <Address>12 Orchestra Terrace</Address>
      <City>Walla Walla</City>
      <Region>WA</Region>
      <PostalCode>99362</PostalCode>
      <Country>USA</Country>
    </FullAddress>
  </Customer>
  <Customer CustomerID="LETSS">
    <CompanyName>Let's Stop N Shop</CompanyName>
    <ContactName>Jaime Yorres</ContactName>
    <ContactTitle>Owner</ContactTitle>
    <Phone>(415) 555-5938</Phone>
    <FullAddress>
      <Address>87 Polk St. Suite 5</Address>
      <City>San Francisco</City>
      <Region>CA</Region>
      <PostalCode>94117</PostalCode>
      <Country>USA</Country>
    </FullAddress>
  </Customer>
</Root>
```

# How to: Join Two Collections (LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

An element or attribute in an XML document can sometimes refer to another element or attribute. For example, the [Sample XML File: Customers and Orders \(LINQ to XML\)](#) XML document contains a list of customers and a list of orders. Each `Customer` element contains a `CustomerID` attribute. Each `Order` element contains a `CustomerID` element. The `CustomerID` element in each order refers to the `CustomerID` attribute in a customer.

The topic [Sample XSD File: Customers and Orders](#) contains an XSD that can be used to validate this document. It uses the `xs:key` and `xs:keyref` features of XSD to establish that the `CustomerID` attribute of the `Customer` element is a key, and to establish a relationship between the `CustomerID` element in each `Order` element and the `CustomerID` attribute in each `Customer` element.

With LINQ to XML, you can take advantage of this relationship by using the `join` clause.

Note that because there is no index available, such joining will have poor runtime performance.

For more detailed information about `join`, see [Join Operations \(C#\)](#).

## Example

The following example joins the `Customer` elements to the `Order` elements, and generates a new XML document that includes the `CompanyName` element in the orders.

Before executing the query, the example validates that the document complies with the schema in [Sample XSD File: Customers and Orders](#). This ensures that the join clause will always work.

This query first retrieves all `Customer` elements, and then joins them to the `Order` elements. It selects only the orders for customers with a `CustomerID` greater than "K". It then projects a new `Order` element that contains the customer information within each order.

This example uses the following XML document: [Sample XML File: Customers and Orders \(LINQ to XML\)](#).

This example uses the following XSD schema: [Sample XSD File: Customers and Orders](#).

Note that joining in this fashion will not perform very well. Joins are performed via a linear search. There are no hash tables or indexes to help with performance.

```

XmlSchemaSet schemas = new XmlSchemaSet();
schemas.Add("", "CustomersOrders.xsd");

Console.WriteLine("Attempting to validate, ");
XDocument custOrdDoc = XDocument.Load("CustomersOrders.xml");

bool errors = false;
custOrdDoc.Validate(schemas, (o, e) =>
{
    Console.WriteLine("{0}", e.Message);
    errors = true;
});
Console.WriteLine("custOrdDoc {0}", errors ? "did not validate" : "validated");

if (!errors)
{
    // Join customers and orders, and create a new XML document with
    // a different shape.

    // The new document contains orders only for customers with a
    // CustomerID > 'K'
    XElement custOrd = custOrdDoc.Element("Root");
    XElement newCustOrd = new XElement("Root",
        from c in custOrd.Element("Customers").Elements("Customer")
        join o in custOrd.Element("Orders").Elements("Order")
        on (string)c.Attribute("CustomerID") equals
            (string)o.Element("CustomerID")
        where ((string)c.Attribute("CustomerID")).CompareTo("K") > 0
        select new XElement("Order",
            new XElement("CustomerID", (string)c.Attribute("CustomerID")),
            new XElement("CompanyName", (string)c.Element("CompanyName")),
            new XElement("ContactName", (string)c.Element("ContactName")),
            new XElement("EmployeeID", (string)o.Element("EmployeeID")),
            new XElement("OrderDate", (DateTime)o.Element("OrderDate"))
        )
    );
    Console.WriteLine(newCustOrd);
}

```

This code produces the following output:

Attempting to validate, custOrdDoc validated

```
<Root>
  <Order>
    <CustomerID>LAZYK</CustomerID>
    <CompanyName>Lazy K Kountry Store</CompanyName>
    <ContactName>John Steel</ContactName>
    <EmployeeID>1</EmployeeID>
    <OrderDate>1997-03-21T00:00:00</OrderDate>
  </Order>
  <Order>
    <CustomerID>LAZYK</CustomerID>
    <CompanyName>Lazy K Kountry Store</CompanyName>
    <ContactName>John Steel</ContactName>
    <EmployeeID>8</EmployeeID>
    <OrderDate>1997-05-22T00:00:00</OrderDate>
  </Order>
  <Order>
    <CustomerID>LETSS</CustomerID>
    <CompanyName>Let's Stop N Shop</CompanyName>
    <ContactName>Jaime Yorres</ContactName>
    <EmployeeID>1</EmployeeID>
    <OrderDate>1997-06-25T00:00:00</OrderDate>
  </Order>
  <Order>
    <CustomerID>LETSS</CustomerID>
    <CompanyName>Let's Stop N Shop</CompanyName>
    <ContactName>Jaime Yorres</ContactName>
    <EmployeeID>8</EmployeeID>
    <OrderDate>1997-10-27T00:00:00</OrderDate>
  </Order>
  <Order>
    <CustomerID>LETSS</CustomerID>
    <CompanyName>Let's Stop N Shop</CompanyName>
    <ContactName>Jaime Yorres</ContactName>
    <EmployeeID>6</EmployeeID>
    <OrderDate>1997-11-10T00:00:00</OrderDate>
  </Order>
  <Order>
    <CustomerID>LETSS</CustomerID>
    <CompanyName>Let's Stop N Shop</CompanyName>
    <ContactName>Jaime Yorres</ContactName>
    <EmployeeID>4</EmployeeID>
    <OrderDate>1998-02-12T00:00:00</OrderDate>
  </Order>
</Root>
```



# How to: Create Hierarchy Using Grouping (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This example shows how to group data, and then generate XML based on the grouping.

## Example

This example first groups data by a category, then generates a new XML file in which the XML hierarchy reflects the grouping.

This example uses the following XML document: [Sample XML File: Numerical Data \(LINQ to XML\)](#).

```
XElement doc = XElement.Load("Data.xml");
var newData =
    new XElement("Root",
        from data in doc.Elements("Data")
        group data by (string)data.Element("Category") into groupedData
        select new XElement("Group",
            new XAttribute("ID", groupedData.Key),
            from g in groupedData
            select new XElement("Data",
                g.Element("Quantity"),
                g.Element("Price")
            )
        )
    );
Console.WriteLine(newData);
```

This example produces the following output:

```
<Root>
  <Group ID="A">
    <Data>
      <Quantity>3</Quantity>
      <Price>24.50</Price>
    </Data>
    <Data>
      <Quantity>5</Quantity>
      <Price>4.95</Price>
    </Data>
    <Data>
      <Quantity>3</Quantity>
      <Price>66.00</Price>
    </Data>
    <Data>
      <Quantity>15</Quantity>
      <Price>29.00</Price>
    </Data>
  </Group>
  <Group ID="B">
    <Data>
      <Quantity>1</Quantity>
      <Price>89.99</Price>
    </Data>
    <Data>
      <Quantity>10</Quantity>
      <Price>.99</Price>
    </Data>
    <Data>
      <Quantity>8</Quantity>
      <Price>6.99</Price>
    </Data>
  </Group>
</Root>
```

# How to: Query LINQ to XML Using XPath (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic introduces the extension methods that enable you to query an XML tree by using XPath. For detailed information about using these extension methods, see [System.Xml.XPath.Extensions](#).

Unless you have a very specific reason for querying using XPath, such as extensive use of legacy code, using XPath with LINQ to XML is not recommended. XPath queries will not perform as well as LINQ to XML queries.

## Example

The following example creates a small XML tree and uses [XPathSelectElements](#) to select a set of elements.

```
XElement root = new XElement("Root",
    new XElement("Child1", 1),
    new XElement("Child1", 2),
    new XElement("Child1", 3),
    new XElement("Child2", 4),
    new XElement("Child2", 5),
    new XElement("Child2", 6)
);
IEnumerable<XElement> list = root.XPathSelectElements("./Child2");
foreach (XElement el in list)
    Console.WriteLine(el);
```

This example produces the following output:

```
<Child2>4</Child2>
<Child2>5</Child2>
<Child2>6</Child2>
```

# How to: Write a LINQ to XML Axis Method (C#)

6/4/2019 • 3 minutes to read • [Edit Online](#)

You can write your own axis methods to retrieve collections from an XML tree. One of the best ways to do this is to write an extension method that returns a collection of elements or attributes. You can write your extension method to return specific subsets of elements or attributes, based on the requirements of your application.

## Example

The following example uses two extension methods. The first extension method, `GetXPath`, operates on `XObject`, and returns an XPath expression that when evaluated will return the node or attribute. The second extension method, `Find`, operates on `XElement`. It returns a collection of `XAttribute` objects and `XElement` objects that contain some specified text.

This example uses the following XML document: [Sample XML File: Multiple Purchase Orders \(LINQ to XML\)](#).

```
public static class MyExtensions
{
    private static string GetQName(XElement xe)
    {
        string prefix = xe.GetPrefixOfNamespace(xe.Name.Namespace);
        if (xe.Name.Namespace == XNamespace.None || prefix == null)
            return xe.Name.LocalName.ToString();
        else
            return prefix + ":" + xe.Name.LocalName.ToString();
    }

    private static string GetQName(XAttribute xa)
    {
        string prefix =
            xa.Parent.GetPrefixOfNamespace(xa.Name.Namespace);
        if (xa.Name.Namespace == XNamespace.None || prefix == null)
            return xa.Name.ToString();
        else
            return prefix + ":" + xa.Name.LocalName;
    }

    private static string NameWithPredicate(XElement el)
    {
        if (el.Parent != null && el.Parent.Elements(el.Name).Count() != 1)
            return GetQName(el) + "[" +
                (el.ElementsBeforeSelf(el.Name).Count() + 1) + "]";
        else
            return GetQName(el);
    }

    public static string StrCat<T>(this IEnumerable<T> source,
        string separator)
    {
        return source.Aggregate(new StringBuilder(),
            (sb, i) => sb
                .Append(i.ToString())
                .Append(separator),
            s => s.ToString());
    }

    public static string GetXPath(this XElement xobj)
    {
        if (xobj.Parent == null)
            return xobj.Name.ToString();
        else
            return GetXPath(xobj.Parent) + "/" + NameWithPredicate(xobj);
    }
}
```

```

        {
            XDocument doc = xobj as XDocument;
            if (doc != null)
                return ".";
            XElement el = xobj as XElement;
            if (el != null)
                return "/" + NameWithPredicate(el);
            // the XPath data model does not include white space text nodes
            // that are children of a document, so this method returns null.
            XText xt = xobj as XText;
            if (xt != null)
                return null;
            XComment com = xobj as XComment;
            if (com != null)
                return
                    "/" +
                    (
                        com
                        .Document
                        .Nodes()
                        .OfType<XComment>()
                        .Count() != 1 ?
                        "comment()[" +
                        (com
                        .NodesBeforeSelf()
                        .OfType<XComment>()
                        .Count() + 1) +
                        "]" :
                        "comment()"
                    );
            XProcessingInstruction pi = xobj as XProcessingInstruction;
            if (pi != null)
                return
                    "/" +
                    (
                        pi.Document.Nodes()
                        .OfType<XProcessingInstruction>()
                        .Count() != 1 ?
                        "processing-instruction()[" +
                        (pi
                        .NodesBeforeSelf()
                        .OfType<XProcessingInstruction>()
                        .Count() + 1) +
                        "]" :
                        "processing-instruction()"
                    );
            return null;
        }
    }
    else
    {
        XElement el = xobj as XElement;
        if (el != null)
        {
            return
                "/" +
                el
                .Ancestors()
                .InDocumentOrder()
                .Select(e => NameWithPredicate(e))
                .StrCat("/") +
                NameWithPredicate(el);
        }
        XAttribute at = xobj as XAttribute;
        if (at != null)
            return
                "/" +
                at
                .Parent
                .AncestorsAndSelf()
    }
}

```

```

        .InDocumentOrder()
        .Select(e => NameWithPredicate(e))
        .StrCat("/") +
        "@" + GetQName(at);
XComment com = xobj as XComment;
if (com != null)
    return
        "/" +
        com
        .Parent
        .AncestorsAndSelf()
        .InDocumentOrder()
        .Select(e => NameWithPredicate(e))
        .StrCat("/") +
        (
            com
            .Parent
            .Nodes()
            .OfType<XComment>()
            .Count() != 1 ?
            "comment()[" +
            (com
            .NodesBeforeSelf()
            .OfType<XComment>()
            .Count() + 1) + "]" :
            "comment()"
        );
XCData cd = xobj as XCData;
if (cd != null)
    return
        "/" +
        cd
        .Parent
        .AncestorsAndSelf()
        .InDocumentOrder()
        .Select(e => NameWithPredicate(e))
        .StrCat("/") +
        (
            cd
            .Parent
            .Nodes()
            .OfType<XText>()
            .Count() != 1 ?
            "text()[" +
            (cd
            .NodesBeforeSelf()
            .OfType<XText>()
            .Count() + 1) + "]" :
            "text()"
        );
XText tx = xobj as XText;
if (tx != null)
    return
        "/" +
        tx
        .Parent
        .AncestorsAndSelf()
        .InDocumentOrder()
        .Select(e => NameWithPredicate(e))
        .StrCat("/") +
        (
            tx
            .Parent
            .Nodes()
            .OfType<XText>()
            .Count() != 1 ?
            "text()[" +
            (tx
            .NodesBeforeSelf()

```

```

        .OfType<XText>()
        .Count() + 1) + "]" :
        "text()"
    );
    XProcessingInstruction pi = xobj as XProcessingInstruction;
    if (pi != null)
        return
            "/" +
            pi
            .Parent
            .AncestorsAndSelf()
            .InDocumentOrder()
            .Select(e => NameWithPredicate(e))
            .StrCat("/") +
            (
                pi
                .Parent
                .Nodes()
                .OfType<XProcessingInstruction>()
                .Count() != 1 ?
                "processing-instruction()[" +
                (pi
                .NodesBeforeSelf()
                .OfType<XProcessingInstruction>()
                .Count() + 1) + "]" :
                "processing-instruction()"
            );
        return null;
    }
}

public static IEnumerable<XObject> Find(this XElement source, string value)
{
    if (source.Attributes().Any())
    {
        foreach (XAttribute att in source.Attributes())
        {
            string contents = (string)att;
            if (contents.Contains(value))
                yield return att;
        }
    }
    if (source.Elements().Any())
    {
        foreach (XElement child in source.Elements())
            foreach (XObject s in child.Find(value))
                yield return s;
    }
    else
    {
        string contents = (string)source;
        if (contents.Contains(value))
            yield return source;
    }
}

}

class Program
{
    static void Main(string[] args)
    {
        XElement purchaseOrders = XElement.Load("PurchaseOrders.xml");

        IEnumerable<XObject> subset =
            from xobj in purchaseOrders.Find("1999")
            select xobj;

        foreach (XObject obj in subset)
        {

```

```
        Console.WriteLine(obj.GetXPath());
        if (obj.GetType() == typeof(XElement))
            Console.WriteLine(((XElement)obj).Value);
        else if (obj.GetType() == typeof(XAttribute))
            Console.WriteLine(((XAttribute)obj).Value);
    }
}
```

This code produces the following output:

```
/PurchaseOrders/PurchaseOrder[1]/@OrderDate
1999-10-20
/PurchaseOrders/PurchaseOrder[1]/Items/Item[2]/ShipDate
1999-05-21
/PurchaseOrders/PurchaseOrder[2]/@OrderDate
1999-10-22
/PurchaseOrders/PurchaseOrder[3]/@OrderDate
1999-10-22
```



# How to: Perform Streaming Transformations of Text to XML (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

One approach to processing a text file is to write an extension method that streams the text file a line at a time using the `yield return` construct. You then can write a LINQ query that processes the text file in a lazy deferred fashion. If you then use [XStreamingElement](#) to stream output, you then can create a transformation from the text file to XML that uses a minimal amount of memory, regardless of the size of the source text file.

There are some caveats regarding streaming transformations. A streaming transformation is best applied in situations where you can process the entire file once, and if you can process the lines in the order that they occur in the source document. If you have to process the file more than once, or if you have to sort the lines before you can process them, you will lose many of the benefits of using a streaming technique.

## Example

The following text file, `People.txt`, is the source for this example.

```
#This is a comment
1,Tai,Yee,Writer
2,Nikolay,Grachev,Programmer
3,David,Wright,Inventor
```

The following code contains an extension method that streams the lines of the text file in a deferred fashion.

```

public static class StreamReaderSequence
{
    public static IEnumerable<string> Lines(this StreamReader source)
    {
        String line;

        if (source == null)
            throw new ArgumentNullException("source");
        while ((line = source.ReadLine()) != null)
        {
            yield return line;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        StreamReader sr = new StreamReader("People.txt");
        XStreamingElement xmlTree = new XStreamingElement("Root",
            from line in sr.Lines()
            let items = line.Split(',')
            where !line.StartsWith("#")
            select new XElement("Person",
                new XAttribute("ID", items[0]),
                new XElement("First", items[1]),
                new XElement("Last", items[2]),
                new XElement("Occupation", items[3])
            )
        );
        Console.WriteLine(xmlTree);
        sr.Close();
    }
}

```

This example produces the following output:

```

<Root>
  <Person ID="1">
    <First>Tai</First>
    <Last>Yee</Last>
    <Occupation>Writer</Occupation>
  </Person>
  <Person ID="2">
    <First>Nikolay</First>
    <Last>Grachev</Last>
    <Occupation>Programmer</Occupation>
  </Person>
  <Person ID="3">
    <First>David</First>
    <Last>Wright</Last>
    <Occupation>Inventor</Occupation>
  </Person>
</Root>

```

## See also

- [XStreamingElement](#)

# How to: List All Nodes in a Tree (C#)

6/4/2019 • 4 minutes to read • [Edit Online](#)

Sometimes it is helpful to list all nodes in a tree. This can be useful when learning exactly how a method or property affects the tree. One approach to listing all nodes in a textual form is to generate an XPath expression that exactly and specifically identifies any node in the tree.

It is not particularly helpful to execute XPath expressions using LINQ to XML. XPath expressions have poorer performance than LINQ to XML queries, and LINQ to XML queries are much more powerful. However, as a way to identify nodes in the XML tree, XPath works well.

## Example

This example shows an function named `GetXPath` that generates a specific XPath expression for any node in the XML tree. It generates appropriate XPath expressions even when nodes are in a namespace. The XPath expressions are generated by using namespace prefixes.

The example then creates a small XML tree that contains an example of several types of nodes. It then iterates through the descendant nodes and prints the XPath expression for each node.

You will notice that the XML declaration is not a node in the tree.

The following is an XML file that contains several types of nodes:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<?target data?>
<Root AttName="An Attribute" xmlns:aw="http://www.adventure-works.com">
  <!--This is a comment-->
  <Child>Text</Child>
  <Child>Other Text</Child>
  <ChildWithMixedContent>text<b>BoldText</b>otherText</ChildWithMixedContent>
  <aw:ElementInNamespace>
    <aw:ChildInNamespace />
  </aw:ElementInNamespace>
</Root>
```

The following is the list of nodes in the above XML tree, expressed as XPath expressions:

```
/processing-instruction()
/Root
/Root/@AttName
/Root/@xmlns:aw
/Root/comment()
/Root/Child[1]
/Root/Child[1]/text()
/Root/Child[2]
/Root/Child[2]/text()
/Root/ChildWithMixedContent
/Root/ChildWithMixedContent/text()[1]
/Root/ChildWithMixedContent/b
/Root/ChildWithMixedContent/b/text()
/Root/ChildWithMixedContent/text()[2]
/Root/aw:ElementInNamespace
/Root/aw:ElementInNamespace/aw:ChildInNamespace
```



```

        .Count() + 1) +
        "]" :
        "comment()"
    );
XProcessingInstruction pi = xobj as XProcessingInstruction;
if (pi != null)
    return
        "/" +
        (
            pi.Document.Nodes()
            .OfType<XProcessingInstruction>()
            .Count() != 1 ?
            "processing-instruction()[" +
            (pi
            .NodesBeforeSelf()
            .OfType<XProcessingInstruction>()
            .Count() + 1) +
            "]" :
            "processing-instruction()"
        );
    return null;
}
else
{
    XElement el = xobj as XElement;
    if (el != null)
    {
        return
            "/" +
            el
            .Ancestors()
            .InDocumentOrder()
            .Select(e => NameWithPredicate(e))
            .StrCat("/") +
            NameWithPredicate(el);
    }
    XAttribute at = xobj as XAttribute;
    if (at != null)
        return
            "/" +
            at
            .Parent
            .AncestorsAndSelf()
            .InDocumentOrder()
            .Select(e => NameWithPredicate(e))
            .StrCat("/") +
            "@" + GetQName(at);
    XComment com = xobj as XComment;
    if (com != null)
        return
            "/" +
            com
            .Parent
            .AncestorsAndSelf()
            .InDocumentOrder()
            .Select(e => NameWithPredicate(e))
            .StrCat("/") +
            (
                com
                .Parent
                .Nodes()
                .OfType<XComment>()
                .Count() != 1 ?
                "comment()[" +
                (com
                .NodesBeforeSelf()
                .OfType<XComment>()
                .Count() + 1) + "]" :
                "comment()"
            );
    }
}

```

```

    );
    XCData cd = xobj as XCData;
    if (cd != null)
        return
            "/" +
            cd
            .Parent
            .AncestorsAndSelf()
            .InDocumentOrder()
            .Select(e => NameWithPredicate(e))
            .StrCat("/") +
            (
                cd
                .Parent
                .Nodes()
                .OfType<XText>()
                .Count() != 1 ?
                "text()[" +
                (cd
                .NodesBeforeSelf()
                .OfType<XText>()
                .Count() + 1) + "]" :
                "text()"
            );
    XText tx = xobj as XText;
    if (tx != null)
        return
            "/" +
            tx
            .Parent
            .AncestorsAndSelf()
            .InDocumentOrder()
            .Select(e => NameWithPredicate(e))
            .StrCat("/") +
            (
                tx
                .Parent
                .Nodes()
                .OfType<XText>()
                .Count() != 1 ?
                "text()[" +
                (tx
                .NodesBeforeSelf()
                .OfType<XText>()
                .Count() + 1) + "]" :
                "text()"
            );
    XProcessingInstruction pi = xobj as XProcessingInstruction;
    if (pi != null)
        return
            "/" +
            pi
            .Parent
            .AncestorsAndSelf()
            .InDocumentOrder()
            .Select(e => NameWithPredicate(e))
            .StrCat("/") +
            (
                pi
                .Parent
                .Nodes()
                .OfType<XProcessingInstruction>()
                .Count() != 1 ?
                "processing-instruction()[" +
                (pi
                .NodesBeforeSelf()
                .OfType<XProcessingInstruction>()
                .Count() + 1) + "]" :
                "processing-instruction()"
            );

```

```

        );
        return null;
    }
}

class Program
{
    static void Main(string[] args)
    {
        XNamespace aw = "http://www.adventure-works.com";
        XDocument doc = new XDocument(
            new XDeclaration("1.0", "utf-8", "yes"),
            new XProcessingInstruction("target", "data"),
            new XElement("Root",
                new XAttribute("AttName", "An Attribute"),
                new XAttribute(XNamespace.Xmlns + "aw", aw.ToString()),
                new XComment("This is a comment"),
                new XElement("Child",
                    new XText("Text")
                ),
                new XElement("Child",
                    new XText("Other Text")
                ),
                new XElement("ChildWithMixedContent",
                    new XText("text"),
                    new XElement("b", "BoldText"),
                    new XText("otherText")
                ),
                new XElement(aw + "ElementInNamespace",
                    new XElement(aw + "ChildInNamespace")
                )
            )
        );
        doc.Save("Test.xml");
        Console.WriteLine(File.ReadAllText("Test.xml"));
        Console.WriteLine("-----");
        foreach (XObject obj in doc.DescendantNodes())
        {
            Console.WriteLine(obj.GetXPath());
            XElement el = obj as XElement;
            if (el != null)
            {
                foreach (XAttribute at in el.Attributes())
                {
                    Console.WriteLine(at.GetXPath());
                }
            }
        }
    }
}

```

This example produces the following output:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<?target data?>
<Root AttName="An Attribute" xmlns:aw="http://www.adventure-works.com">
  <!--This is a comment-->
  <Child>Text</Child>
  <Child>Other Text</Child>
  <ChildWithMixedContent>text<b>BoldText</b>otherText</ChildWithMixedContent>
  <aw:ElementInNamespace>
    <aw:ChildInNamespace />
  </aw:ElementInNamespace>
</Root>
-----
/processing-instruction()
/Root
/Root/@AttName
/Root/@xmlns:aw
/Root/comment()
/Root/Child[1]
/Root/Child[1]/text()
/Root/Child[2]
/Root/Child[2]/text()
/Root/ChildWithMixedContent
/Root/ChildWithMixedContent/text()[1]
/Root/ChildWithMixedContent/b
/Root/ChildWithMixedContent/b/text()
/Root/ChildWithMixedContent/text()[2]
/Root/aw:ElementInNamespace
/Root/aw:ElementInNamespace/aw:ChildInNamespace
```



# How to: Retrieve Paragraphs from an Office Open XML Document (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic presents an example that opens an Office Open XML document, and retrieves a collection of all of the paragraphs in the document.

For more information on Office Open XML, see [Open XML SDK](#) and [www.ericwhite.com](http://www.ericwhite.com).

## Example

This example opens an Office Open XML package, uses the relationships within the Open XML package to find the document and the style parts. It then queries the document, projecting a collection of an anonymous type that contains the paragraph `XElement` node, the style name of each paragraph, and the text of each paragraph.

The example uses an extension method named `StringConcatenate`, which is also supplied in the example.

For a detailed tutorial that explains how this example works, see [Pure Functional Transformations of XML \(C#\)](#).

This example uses classes found in the WindowsBase assembly. It uses types in the `System.IO.Packaging` namespace.

```
public static class LocalExtensions
{
    public static string StringConcatenate(this IEnumerable<string> source)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item));
        return sb.ToString();
    }

    public static string StringConcatenate(this IEnumerable<string> source, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s).Append(separator);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item)).Append(separator);
        return sb.ToString();
    }
}
```

```

class Program
{
    public static string ParagraphText(XElement e)
    {
        XNamespace w = e.Name.Namespace;
        return e
            .Elements(w + "r")
            .Elements(w + "t")
            .StringConcatenate(element => (string)element);
    }

    static void Main(string[] args)
    {
        const string fileName = "SampleDoc.docx";

        const string documentRelationshipType =
            "http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
        const string stylesRelationshipType =
            "http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
        const string wordmlNamespace =
            "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
        XNamespace w = wordmlNamespace;

        XDocument xDoc = null;
        XDocument styleDoc = null;

        using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.Read))
        {
            PackageRelationship docPackageRelationship =
                wdPackage
                    .GetRelationshipsByType(documentRelationshipType)
                    .FirstOrDefault();
            if (docPackageRelationship != null)
            {
                Uri documentUri =
                    PackUriHelper
                        .ResolvePartUri(
                            new Uri("/", UriKind.Relative),
                            docPackageRelationship.TargetUri);
                PackagePart documentPart =
                    wdPackage.GetPart(documentUri);

                // Load the document XML in the part into an XDocument instance.
                xDoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

                // Find the styles part. There will only be one.
                PackageRelationship styleRelation =
                    documentPart.GetRelationshipsByType(stylesRelationshipType)
                        .FirstOrDefault();
                if (styleRelation != null)
                {
                    Uri styleUri = PackUriHelper.ResolvePartUri(documentUri, styleRelation.TargetUri);
                    PackagePart stylePart = wdPackage.GetPart(styleUri);

                    // Load the style XML in the part into an XDocument instance.
                    styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));
                }
            }
        }

        string defaultStyle =
            (string)(
                from style in styleDoc.Root.Elements(w + "style")
                where (string)style.Attribute(w + "type") == "paragraph" &&
                    (string)style.Attribute(w + "default") == "1"
                select style
            ).First().Attribute(w + "styleId");
    }
}

```

```

// Find all paragraphs in the document.
var paragraphs =
    from para in xDoc
        .Root
        .Element(w + "body")
        .Descendants(w + "p")

    let styleNode = para
        .Elements(w + "pPr")
        .Elements(w + "pStyle")
        .FirstOrDefault()

    select new
    {
        ParagraphNode = para,
        StyleName = styleNode != null ?
            (string)styleNode.Attribute(w + "val") :
            defaultStyle
    };

// Retrieve the text of each paragraph.
var paraWithText =
    from para in paragraphs
    select new
    {
        ParagraphNode = para.ParagraphNode,
        StyleName = para.StyleName,
        Text = ParagraphText(para.ParagraphNode)
    };

foreach (var p in paraWithText)
    Console.WriteLine("StyleName:{0} >{1}<", p.StyleName, p.Text);
}

```

When run with the sample Open XML document described in [Creating the Source Office Open XML Document \(C#\)](#), this example produces the following output:

```

StyleName:Heading1 >Parsing WordprocessingML with LINQ to XML<
StyleName:Normal ><
StyleName:Normal >The following example prints to the console.<
StyleName:Normal ><
StyleName:Code >using System;<
StyleName:Code ><
StyleName:Code >class Program {<
StyleName:Code >    public static void (string[] args) {<
StyleName:Code >        Console.WriteLine("Hello World");<
StyleName:Code >    }<
StyleName:Code >}<
StyleName:Normal ><
StyleName:Normal >This example produces the following output:<
StyleName:Normal ><
StyleName:Code >Hello World<

```

# How to: Modify an Office Open XML Document (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic presents an example that opens an Office Open XML document, modifies it, and saves it.

For more information on Office Open XML, see [Open XML SDK](#) and [www.ericwhite.com](http://www.ericwhite.com).

## Example

This example finds the first paragraph element in the document. It retrieves the text from the paragraph, and then deletes all text runs in the paragraph. It creates a new text run that consists of the first paragraph text that has been converted to upper case. It then serializes the changed XML into the Open XML package and closes it.

This example uses classes found in the WindowsBase assembly. It uses types in the [System.IO.Packaging](#) namespace.

```
public static class LocalExtensions
{
    public static string StringConcatenate(this IEnumerable<string> source)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item));
        return sb.ToString();
    }

    public static string StringConcatenate(this IEnumerable<string> source, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s).Append(separator);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item)).Append(separator);
        return sb.ToString();
    }
}

class Program
{
    public static string ParagraphText(XElement e)
    {
        XNamespace w = e.Name.Namespace;
        return e
            .Elements(w + "r")
```

```

        .Elements(w + "t")
        .StringConcatenate(element => (string)element);
    }

static void Main(string[] args)
{
    const string fileName = "SampleDoc.docx";

    const string documentRelationshipType =
        "http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
    const string stylesRelationshipType =
        "http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
    const string wordmlNamespace =
        "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
    XNamespace w = wordmlNamespace;

    using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.ReadWrite))
    {
        PackageRelationship docPackageRelationship =
            wdPackage.GetRelationshipsByType(documentRelationshipType).FirstOrDefault();
        if (docPackageRelationship != null)
        {
            Uri documentUri = PackUriHelper.ResolvePartUri(new Uri("/"), UriKind.Relative),
                docPackageRelationship.TargetUri);
            PackagePart documentPart = wdPackage.GetPart(documentUri);

            // Load the document XML in the part into an XDocument instance.
            XDocument xDoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

            // Find the styles part. There will only be one.
            PackageRelationship styleRelation =
                documentPart.GetRelationshipsByType(stylesRelationshipType).FirstOrDefault();
            PackagePart stylePart = null;
            XDocument styleDoc = null;

            if (styleRelation != null)
            {
                Uri styleUri = PackUriHelper.ResolvePartUri(documentUri, styleRelation.TargetUri);
                stylePart = wdPackage.GetPart(styleUri);

                // Load the style XML in the part into an XDocument instance.
                styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));
            }

            XElement paraNode = xDoc
                .Root
                .Element(w + "body")
                .Descendants(w + "p")
                .FirstOrDefault();

            string paraText = ParagraphText(paraNode);

            // remove all text runs
            paraNode.Descendants(w + "r").Remove();

            paraNode.Add(
                new XElement(w + "r",
                    new XElement(w + "t", paraText.ToUpper())
                )
            );

            // Save the XML into the package
            using (XmlWriter xw =
                XmlWriter.Create(documentPart.GetStream(FileMode.Create, FileAccess.Write)))
            {
                xDoc.Save(xw);
            }

            Console.WriteLine("New first paragraph: >{0}<", paraText.ToUpper());
        }
    }
}

```

```
}  
  }  
}
```

If you open `SampleDoc.docx` after running this program, you can see that this program converted the first paragraph in the document to upper case.

When run with the sample Open XML document described in [Creating the Source Office Open XML Document \(C#\)](#), this example produces the following output:

```
New first paragraph: >PARSING WORDPROCESSINGML WITH LINQ TO XML<
```

# How to: Populate an XML Tree from the File System (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

A common and useful application of XML trees is as a hierarchical name/value data store. You can populate an XML tree with hierarchical data, and then query it, transform it, and if necessary, serialize it. In this usage scenario, many of the XML specific semantics, such as namespaces and white space behavior, are not important. Instead, you are using the XML tree as a small, in memory, single user hierarchical database.

## Example

The following example populates an XML tree from the local file system using recursion. It then queries the tree, calculating the total of the sizes of all files in the tree.

```
class Program
{
    static XElement CreateFileSystemXmlTree(string source)
    {
        DirectoryInfo di = new DirectoryInfo(source);
        return new XElement("Dir",
            new XAttribute("Name", di.Name),
            from d in Directory.GetDirectories(source)
            select CreateFileSystemXmlTree(d),
            from fi in di.GetFiles()
            select new XElement("File",
                new XElement("Name", fi.Name),
                new XElement("Length", fi.Length)
            )
        );
    }

    static void Main(string[] args)
    {
        XElement fileSystemTree = CreateFileSystemXmlTree("C:/Tmp");
        Console.WriteLine(fileSystemTree);
        Console.WriteLine("-----");
        long totalFileSize =
            (from f in fileSystemTree.Descendants("File")
             select (long)f.Element("Length")).Sum();
        Console.WriteLine("Total File Size:{0}", totalFileSize);
    }
}
```

This example produces output similar to the following:

```
<Dir Name="Tmp">
  <Dir Name="ConsoleApplication1">
    <Dir Name="bin">
      <Dir Name="Debug">
        <File>
          <Name>ConsoleApplication1.exe</Name>
          <Length>4608</Length>
        </File>
        <File>
          <Name>ConsoleApplication1.pdb</Name>
          <Length>11776</Length>
        </File>
        <File>
          <Name>ConsoleApplication1.vshost.exe</Name>
          <Length>9568</Length>
        </File>
        <File>
          <Name>ConsoleApplication1.vshost.exe.manifest</Name>
          <Length>473</Length>
        </File>
      </Dir>
    </Dir>
    <Dir Name="obj">
      <Dir Name="Debug">
        <Dir Name="TempPE" />
        <File>
          <Name>ConsoleApplication1.csproj.FileListAbsolute.txt</Name>
          <Length>322</Length>
        </File>
        <File>
          <Name>ConsoleApplication1.exe</Name>
          <Length>4608</Length>
        </File>
        <File>
          <Name>ConsoleApplication1.pdb</Name>
          <Length>11776</Length>
        </File>
      </Dir>
    </Dir>
    <Dir Name="Properties">
      <File>
        <Name>AssemblyInfo.cs</Name>
        <Length>1454</Length>
      </File>
    </Dir>
    <File>
      <Name>ConsoleApplication1.csproj</Name>
      <Length>2546</Length>
    </File>
    <File>
      <Name>ConsoleApplication1.sln</Name>
      <Length>937</Length>
    </File>
    <File>
      <Name>ConsoleApplication1.suo</Name>
      <Length>10752</Length>
    </File>
    <File>
      <Name>Program.cs</Name>
      <Length>269</Length>
    </File>
  </Dir>
</Dir>
-----
Total File Size:59089
```



# Comparison of XPath and LINQ to XML

6/4/2019 • 3 minutes to read • [Edit Online](#)

XPath and LINQ to XML offer some similar functionality. Both can be used to query an XML tree, returning such results as a collection of elements, a collection of attributes, a collection of nodes, or the value of an element or attribute. However, there are also some differences.

## Differences Between XPath and LINQ to XML

XPath does not allow projection of new types. It can only return collections of nodes from the tree, whereas LINQ to XML can execute a query and project an object graph or an XML tree in a new shape. LINQ to XML queries encompass much more functionality and are much more powerful than XPath expressions.

XPath expressions exist in isolation within a string. The C# compiler cannot help parse the XPath expression at compile time. By contrast, LINQ to XML queries are parsed and compiled by the C# compiler. The compiler is able to catch many query errors.

XPath results are not strongly typed. In a number of circumstances, the result of evaluating an XPath expression is an object, and it is up to the developer to determine the proper type and cast the result as necessary. By contrast, the projections from a LINQ to XML query are strongly typed.

## Result Ordering

The XPath 1.0 Recommendation states that a collection that is the result of evaluating an XPath expression is unordered.

However, when iterating through a collection returned by a LINQ to XML XPath axis method, the nodes in the collection are returned in document order. This is the case even when accessing the XPath axes where predicates are expressed in terms of reverse document order, such as `preceding` and `preceding-sibling`.

By contrast, most of the LINQ to XML axes return collections in document order, but two of them, [Ancestors](#) and [AncestorsAndSelf](#), return collections in reverse document order. The following table enumerates the axes, and indicates collection order for each:

LINQ TO XML AXIS	ORDERING
XContainer.DescendantNodes	Document order
XContainer.Descendants	Document order
XContainer.Elements	Document order
XContainer.Nodes	Document order
XContainer.NodesAfterSelf	Document order
XContainer.NodesBeforeSelf	Document order
XElement.AncestorsAndSelf	Reverse document order

LINQ TO XML AXIS	ORDERING
XElement.Attributes	Document order
XElement.DescendantNodesAndSelf	Document order
XElement.DescendantsAndSelf	Document order
XNode.Ancestors	Reverse document order
XNode.ElementsAfterSelf	Document order
XNode.ElementsBeforeSelf	Document order
XNode.NodesAfterSelf	Document order
XNode.NodesBeforeSelf	Document order

## Positional Predicates

Within an XPath expression, positional predicates are expressed in terms of document order for many axes, but are expressed in reverse document order for reverse axes, which are `preceding`, `preceding-sibling`, `ancestor`, and `ancestor-or-self`. For example, the XPath expression `preceding-sibling::*[1]` returns the immediately preceding sibling. This is the case even though the final result set is presented in document order.

By contrast, all positional predicates in LINQ to XML are always expressed in terms of the order of the axis. For example, `anElement.ElementsBeforeSelf().ElementAt(0)` returns the first child element of the parent of the queried element, not the immediate preceding sibling. Another example: `anElement.Ancestors().ElementAt(0)` returns the parent element.

If you wanted to find the immediately preceding element in LINQ to XML, you would write the following expression:

```
ElementsBeforeSelf().Last()
```

```
ElementsBeforeSelf().Last()
```

## Performance Differences

XPath queries that use the XPath functionality in LINQ to XML will not perform as well as LINQ to XML queries.

## Comparison of Composition

Composition of a LINQ to XML query is somewhat parallel to composition of an XPath expression, although very different in syntax.

For example, if you have an element in a variable named `customers`, and you want to find a grandchild element named `CompanyName` under all child elements named `Customer`, you would write an XPath expression as follows:

```
customers.XPathSelectElements("./Customer/CompanyName")
```

```
customers.XPathSelectElements("./Customer/CompanyName")
```

The equivalent LINQ to XML query is:

```
customers.Elements("Customer").Elements("CompanyName")
```

```
customers.Elements("Customer").Elements("CompanyName")
```

There are similar parallels for each of the XPath axes.

XPATH AXIS	LINQ TO XML AXIS
child (the default axis)	<a href="#">XContainer.Elements</a>
Parent (..)	<a href="#">XObject.Parent</a>
attribute axis (@)	<a href="#">XElement.Attribute</a> or <a href="#">XElement.Attributes</a>
ancestor axis	<a href="#">XNode.Ancestors</a>
ancestor-or-self axis	<a href="#">XElement.AncestorsAndSelf</a>
descendant axis (//)	<a href="#">XContainer.Descendants</a> or <a href="#">XContainer.DescendantNodes</a>
descendant-or-self	<a href="#">XElement.DescendantsAndSelf</a> or <a href="#">XElement.DescendantNodesAndSelf</a>
following-sibling	<a href="#">XNode.ElementsAfterSelf</a> or <a href="#">XNode.NodesAfterSelf</a>
preceding-sibling	<a href="#">XNode.ElementsBeforeSelf</a> or <a href="#">XNode.NodesBeforeSelf</a>
following	No direct equivalent.
preceding	No direct equivalent.

# How to: Find a Child Element (XPath-LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic compares the XPath child element axis to the LINQ to XML [Element](#) method.

The XPath expression is `DeliveryNotes`.

## Example

This example finds the child element `DeliveryNotes`.

This example uses the following XML document: [Sample XML File: Multiple Purchase Orders \(LINQ to XML\)](#).

```
XDocument cpo = XDocument.Load("PurchaseOrders.xml");
XElement po = cpo.Root.Element("PurchaseOrder");

// LINQ to XML query
XElement el1 = po.Element("DeliveryNotes");

// XPath expression
XElement el2 = po.XPathSelectElement("DeliveryNotes");
// same as "child::DeliveryNotes"
// same as "./DeliveryNotes"

if (el1 == el2)
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
Console.WriteLine(el1);
```

This example produces the following output:

```
Results are identical
<DeliveryNotes>Please leave packages in shed by driveway.</DeliveryNotes>
```

# How to: Find a List of Child Elements (XPath-LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic compares the XPath child elements axis to the LINQ to XML [Elements](#) axis.

The XPath expression is: `./*`

## Example

This example finds all of the child elements of the `Address` element.

This example uses the following XML document: [Sample XML File: Multiple Purchase Orders \(LINQ to XML\)](#).

```
XDocument cpo = XDocument.Load("PurchaseOrders.xml");
XElement po = cpo.Root.Element("PurchaseOrder").Element("Address");

// LINQ to XML query
IEnumerable<XElement> list1 = po.Elements();

// XPath expression
IEnumerable<XElement> list2 = po.XPathSelectElements("./*");

if (list1.Count() == list2.Count() &&
    list1.Intersect(list2).Count() == list1.Count())
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
foreach (XElement el in list1)
    Console.WriteLine(el);
```

This example produces the following output:

```
Results are identical
<Name>Ellen Adams</Name>
<Street>123 Maple Street</Street>
<City>Mill Valley</City>
<State>CA</State>
<Zip>10999</Zip>
<Country>USA</Country>
```

# How to: Find the Root Element (XPath-LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how to get the root element with XPath and LINQ to XML.

The XPath expression is:

```
/PurchaseOrders
```

## Example

This example finds the root element.

This example uses the following XML document: [Sample XML File: Multiple Purchase Orders \(LINQ to XML\)](#).

```
XDocument po = XDocument.Load("PurchaseOrders.xml");

// LINQ to XML query
XElement e1 = po.Root;

// XPath expression
XElement e2 = po.XPathSelectElement("/PurchaseOrders");

if (e1 == e2)
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
Console.WriteLine(e1.Name);
```

This example produces the following output:

```
Results are identical
PurchaseOrders
```

# How to: Find Descendant Elements (XPath-LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how to get the descendant elements with a particular name.

The XPath expression is `//Name`.

## Example

This example finds all descendants named `Name`.

This example uses the following XML document: [Sample XML File: Multiple Purchase Orders \(LINQ to XML\)](#).

```
XDocument po = XDocument.Load("PurchaseOrders.xml");

// LINQ to XML query
IEnumerable<XElement> list1 = po.Root.Descendants("Name");

// XPath expression
IEnumerable<XElement> list2 = po.XPathSelectElements("//Name");

if (list1.Count() == list2.Count() &&
    list1.Intersect(list2).Count() == list1.Count())
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
foreach (XElement el in list1)
    Console.WriteLine(el);
```

This example produces the following output:

```
Results are identical
<Name>Ellen Adams</Name>
<Name>Tai Yee</Name>
<Name>Cristian Osorio</Name>
<Name>Cristian Osorio</Name>
<Name>Jessica Arnold</Name>
<Name>Jessica Arnold</Name>
```

# How to: Filter on an Attribute (XPath-LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how to get the descendant elements with a specified name, and with an attribute with a specified value.

The XPath expression is:

```
./Address[@Type='Shipping']
```

## Example

This example finds all descendants elements with the name of `Address`, and with a `Type` attribute with a value of "Shipping".

This example uses the following XML document: [Sample XML File: Multiple Purchase Orders \(LINQ to XML\)](#).

```
XDocument po = XDocument.Load("PurchaseOrders.xml");

// LINQ to XML query
IEnumerable<XElement> list1 =
    from el in po.Descendants("Address")
    where (string)el.Attribute("Type") == "Shipping"
    select el;

// XPath expression
IEnumerable<XElement> list2 = po.XPathSelectElements("./Address[@Type='Shipping']");

if (list1.Count() == list2.Count() &&
    list1.Intersect(list2).Count() == list1.Count())
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
foreach (XElement el in list1)
    Console.WriteLine(el);
```

This example produces the following output:



Results are identical

```
<Address Type="Shipping">
  <Name>Ellen Adams</Name>
  <Street>123 Maple Street</Street>
  <City>Mill Valley</City>
  <State>CA</State>
  <Zip>10999</Zip>
  <Country>USA</Country>
</Address>
<Address Type="Shipping">
  <Name>Cristian Osorio</Name>
  <Street>456 Main Street</Street>
  <City>Buffalo</City>
  <State>NY</State>
  <Zip>98112</Zip>
  <Country>USA</Country>
</Address>
<Address Type="Shipping">
  <Name>Jessica Arnold</Name>
  <Street>4055 Madison Ave</Street>
  <City>Seattle</City>
  <State>WA</State>
  <Zip>98112</Zip>
  <Country>USA</Country>
</Address>
```

# How to: Find Related Elements (XPath-LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how to get an element selecting on an attribute that is referred to by the value of another element.

The XPath expression is:

```
../Customer[@CustomerID=/Root/Orders/Order[12]/CustomerID]
```

## Example

This example finds the 12th `Order` element, and then finds the customer for that order.

Note that indexing into a list in .NET is 'zero' based. Indexing into a collection of nodes in an XPath predicate is 'one' based. This example reflects this difference.

This example uses the following XML document: [Sample XML File: Customers and Orders \(LINQ to XML\)](#).

```
XDocument co = XDocument.Load("CustomersOrders.xml");

// LINQ to XML query
XElement customer1 =
    (from el in co.Descendants("Customer")
     where (string)el.Attribute("CustomerID") ==
         (string)(co
             .Element("Root")
             .Element("Orders")
             .Elements("Order")
             .ToList()[11]
             .Element("CustomerID"))
     select el)
    .First();

// An alternate way to write the query that avoids creation
// of a System.Collections.Generic.List:
XElement customer2 =
    (from el in co.Descendants("Customer")
     where (string)el.Attribute("CustomerID") ==
         (string)(co
             .Element("Root")
             .Element("Orders")
             .Elements("Order")
             .Skip(11).First()
             .Element("CustomerID"))
     select el)
    .First();

// XPath expression
XElement customer3 = co.XPathSelectElement(
    "../Customer[@CustomerID=/Root/Orders/Order[12]/CustomerID]");

if (customer1 == customer2 && customer1 == customer3)
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
Console.WriteLine(customer1);
```

This example produces the following output:

```
Results are identical
<Customer CustomerID="HUNG">
  <CompanyName>Hungry Coyote Import Store</CompanyName>
  <ContactName>Yoshi Latimer</ContactName>
  <ContactTitle>Sales Representative</ContactTitle>
  <Phone>(503) 555-6874</Phone>
  <Fax>(503) 555-2376</Fax>
  <FullAddress>
    <Address>City Center Plaza 516 Main St.</Address>
    <City>Elgin</City>
    <Region>OR</Region>
    <PostalCode>97827</PostalCode>
    <Country>USA</Country>
  </FullAddress>
</Customer>
```

# How to: Find Elements in a Namespace (XPath-LINQ to XML) (C#)

6/5/2019 • 2 minutes to read • [Edit Online](#)

XPath expressions can find nodes in a particular namespace. XPath expressions use namespace prefixes for specifying namespaces. To parse an XPath expression that contains namespace prefixes, you must pass an object to the XPath methods that implements [IXmlNamespaceResolver](#). This example uses [XmlNamespaceManager](#).

The XPath expression is:

```
./aw:*
```

## Example

The following example reads an XML tree that contains two namespaces. It uses an [XmlReader](#) to read the XML document. It then gets an [XmlNameTable](#) from the [XmlReader](#), and an [XmlNamespaceManager](#) from the [XmlNameTable](#). It uses the [XmlNamespaceManager](#) when selecting elements.

```
XmlReader reader = XmlReader.Create("ConsolidatedPurchaseOrders.xml");
XElement root = XElement.Load(reader);
XmlNameTable nameTable = reader.NameTable;
XmlNamespaceManager namespaceManager = new XmlNamespaceManager(nameTable);
namespaceManager.AddNamespace("aw", "http://www.adventure-works.com");
IEnumerable<XElement> list1 = root.XPathSelectElements("./aw:*", namespaceManager);
IEnumerable<XElement> list2 =
    from el in root.Elements()
    where el.Name.Namespace == "http://www.adventure-works.com"
    select el;
if (list1.Count() == list2.Count() &&
    list1.Intersect(list2).Count() == list1.Count())
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
foreach (XElement el in list2)
    Console.WriteLine(el);
```

This example produces the following output:

Results are identical

```
<aw:PurchaseOrder PONumber="11223" Date="2000-01-15" xmlns:aw="http://www.adventure-works.com">
  <aw:ShippingAddress>
    <aw:Name>Chris Preston</aw:Name>
    <aw:Street>123 Main St.</aw:Street>
    <aw:City>Seattle</aw:City>
    <aw:State>WA</aw:State>
    <aw:Zip>98113</aw:Zip>
    <aw:Country>USA</aw:Country>
  </aw:ShippingAddress>
  <aw:BillingAddress>
    <aw:Name>Chris Preston</aw:Name>
    <aw:Street>123 Main St.</aw:Street>
    <aw:City>Seattle</aw:City>
    <aw:State>WA</aw:State>
    <aw:Zip>98113</aw:Zip>
    <aw:Country>USA</aw:Country>
  </aw:BillingAddress>
  <aw:DeliveryInstructions>Ship only complete order.</aw:DeliveryInstructions>
  <aw:Item PartNum="LIT-01">
    <aw:ProductID>Litware Networking Card</aw:ProductID>
    <aw:Qty>1</aw:Qty>
    <aw:Price>20.99</aw:Price>
  </aw:Item>
  <aw:Item PartNum="LIT-25">
    <aw:ProductID>Litware 17in LCD Monitor</aw:ProductID>
    <aw:Qty>1</aw:Qty>
    <aw:Price>199.99</aw:Price>
  </aw:Item>
</aw:PurchaseOrder>
```

# How to: Find Preceding Siblings (XPath-LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic compares the XPath `preceding-sibling` axis to the LINQ to XML child `XNode.ElementsBeforeSelf` axis.

The XPath expression is:

```
preceding-sibling::*
```

Note that the results of both `XPathSelectElements` and `XNode.ElementsBeforeSelf` are in document order.

## Example

The following example finds the `FullAddress` element, and then retrieves the previous elements using the `preceding-sibling` axis.

This example uses the following XML document: [Sample XML File: Customers and Orders \(LINQ to XML\)](#).

```
XElement co = XElement.Load("CustomersOrders.xml");

XElement add = co.Element("Customers").Element("Customer").Element("FullAddress");

// LINQ to XML query
IEnumerable<XElement> list1 = add.ElementsBeforeSelf();

// XPath expression
IEnumerable<XElement> list2 = add.XPathSelectElements("preceding-sibling::*");

if (list1.Count() == list2.Count() &&
    list1.Intersect(list2).Count() == list1.Count())
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
foreach (XElement el in list2)
    Console.WriteLine(el);
```

This example produces the following output:

```
Results are identical
<CompanyName>Great Lakes Food Market</CompanyName>
<ContactName>Howard Snyder</ContactName>
<ContactTitle>Marketing Manager</ContactTitle>
<Phone>(503) 555-7555</Phone>
```

# How to: Find Descendants of a Child Element (XPath-LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how to get the descendant elements of a child element with a particular name.

The XPath expression is:

```
./Paragraph//Text/text()
```

## Example

This example simulates the problems of extracting text from an XML representation of a word processing document. It first selects all `Paragraph` elements, and then it selects all `Text` descendant elements of each `Paragraph` element. This doesn't select the descendant `Text` elements of the `Comment` element.

```

XElement root = XElement.Parse(
@"<Root>
  <Paragraph>
    <Text>This is the start of</Text>
  </Paragraph>
  <Comment>
    <Text>This comment is not part of the paragraph text.</Text>
  </Comment>
  <Paragraph>
    <Annotation Emphasis='true'>
      <Text> a sentence.</Text>
    </Annotation>
  </Paragraph>
  <Paragraph>
    <Text> This is a second sentence.</Text>
  </Paragraph>
</Root>");

// LINQ to XML query
string str1 =
    root
    .Elements("Paragraph")
    .Descendants("Text")
    .Select(s => s.Value)
    .Aggregate(
        new StringBuilder(),
        (s, i) => s.Append(i),
        s => s.ToString()
    );

// XPath expression
string str2 =
    ((IEnumerable)root.XPathEvaluate("./Paragraph//Text/text()"))
    .Cast<XText>()
    .Select(s => s.Value)
    .Aggregate(
        new StringBuilder(),
        (s, i) => s.Append(i),
        s => s.ToString()
    );

if (str1 == str2)
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
Console.WriteLine(str2);

```

This example produces the following output:

```

Results are identical
This is the start of a sentence.  This is a second sentence.

```



# How to: Find a Union of Two Location Paths (XPath-LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

XPath allows you to find the union of the results of two XPath location paths.

The XPath expression is:

```
//Category|//Price
```

You can achieve the same results by using the [Concat](#) standard query operator.

## Example

This example finds all of the `Category` elements and all of the `Price` elements, and concatenates them into a single collection. Note that the LINQ to XML query calls [InDocumentOrder](#) to order the results. The results of the XPath expression evaluation are also in document order.

This example uses the following XML document: [Sample XML File: Numerical Data \(LINQ to XML\)](#).

```
XDocument data = XDocument.Load("Data.xml");

// LINQ to XML query
IEnumerable<XElement> list1 =
    data
    .Descendants("Category")
    .Concat(
        data
        .Descendants("Price")
    )
    .InDocumentOrder();

// XPath expression
IEnumerable<XElement> list2 = data.XPathSelectElements("//Category|//Price");

if (list1.Count() == list2.Count() &&
    list1.Intersect(list2).Count() == list1.Count())
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
foreach (XElement el in list1)
    Console.WriteLine(el);
```

This example produces the following output:

Results are identical  
<Category>A</Category>  
<Price>24.50</Price>  
<Category>B</Category>  
<Price>89.99</Price>  
<Category>A</Category>  
<Price>4.95</Price>  
<Category>A</Category>  
<Price>66.00</Price>  
<Category>B</Category>  
<Price>.99</Price>  
<Category>A</Category>  
<Price>29.00</Price>  
<Category>B</Category>  
<Price>6.99</Price>

# How to: Find Sibling Nodes (XPath-LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

You might want to find all siblings of a node that have a specific name. The resulting collection might include the context node if the context node also has the specific name.

The XPath expression is:

```
../Book
```

## Example

This example first finds a `Book` element, and then finds all sibling elements named `Book`. The resulting collection includes the context node.

This example uses the following XML document: [Sample XML File: Books \(LINQ to XML\)](#).

```
XDocument books = XDocument.Load("Books.xml");

XElement book =
    books
    .Root
    .Elements("Book")
    .Skip(1)
    .First();

// LINQ to XML query
IEnumerable<XElement> list1 =
    book
    .Parent
    .Elements("Book");

// XPath expression
IEnumerable<XElement> list2 = book.XPathSelectElements("../Book");

if (list1.Count() == list2.Count() &&
    list1.Intersect(list2).Count() == list1.Count())
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
foreach (XElement el in list1)
    Console.WriteLine(el);
```

This example produces the following output:

Results are identical

```
<Book id="bk101">
  <Author>Garghentini, Davide</Author>
  <Title>XML Developer's Guide</Title>
  <Genre>Computer</Genre>
  <Price>44.95</Price>
  <PublishDate>2000-10-01</PublishDate>
  <Description>An in-depth look at creating applications
    with XML.</Description>
</Book>
<Book id="bk102">
  <Author>Garcia, Debra</Author>
  <Title>Midnight Rain</Title>
  <Genre>Fantasy</Genre>
  <Price>5.95</Price>
  <PublishDate>2000-12-16</PublishDate>
  <Description>A former architect battles corporate zombies,
    an evil sorceress, and her own childhood to become queen
    of the world.</Description>
</Book>
```

# How to: Find an Attribute of the Parent (XPath-LINQ to XML) (C#)

6/5/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how to navigate to the parent element and find an attribute of it.

The XPath expression is:

```
../@id
```

## Example

This example first finds an `Author` element. It then finds the `id` attribute of the parent element.

This example uses the following XML document: [Sample XML File: Books \(LINQ to XML\)](#).

```
XDocument books = XDocument.Load("Books.xml");

XElement author =
    books
    .Root
    .Element("Book")
    .Element("Author");

// LINQ to XML query
XAttribute att1 =
    author
    .Parent
    .Attribute("id");

// XPath expression
XAttribute att2 = ((IEnumerable)author.XPathEvaluate("../@id")).Cast<XAttribute>().First();

if (att1 == att2)
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
Console.WriteLine(att1);
```

This example produces the following output:

```
Results are identical
id="bk101"
```

# How to: Find Attributes of Siblings with a Specific Name (XPath-LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how to find all attributes of the siblings of the context node. Only attributes with a specific name are returned in the collection.

The XPath expression is:

```
../Book/@id
```

## Example

This example first finds a `Book` element, and then finds all sibling elements named `Book`, and then finds all attributes named `id`. The result is a collection of attributes.

This example uses the following XML document: [Sample XML File: Books \(LINQ to XML\)](#).

```
XDocument books = XDocument.Load("Books.xml");

XElement book =
    books
    .Root
    .Element("Book");

// LINQ to XML query
IEnumerable<XAttribute> list1 =
    from el in book.Parent.Elements("Book")
    select el.Attribute("id");

// XPath expression
IEnumerable<XAttribute> list2 =
    ((IEnumerable)book.XPathEvaluate("../Book/@id")).Cast<XAttribute>();

if (list1.Count() == list2.Count() &&
    list1.Intersect(list2).Count() == list1.Count())
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
foreach (XAttribute el in list1)
    Console.WriteLine(el);
```

This example produces the following output:

```
Results are identical
id="bk101"
id="bk102"
```

# How to: Find Elements with a Specific Attribute (XPath-LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

Sometimes you want to find all elements that have a specific attribute. You are not concerned about the contents of the attribute. Instead, you want to select based on the existence of the attribute.

The XPath expression is:

```
./*[@Select]
```

## Example

The following code selects just the elements that have the `Select` attribute.

```
XElement doc = XElement.Parse(
@"<Root>
  <Child1>1</Child1>
  <Child2 Select='true'>2</Child2>
  <Child3>3</Child3>
  <Child4 Select='true'>4</Child4>
  <Child5>5</Child5>
</Root>");

// LINQ to XML query
IEnumerable<XElement> list1 =
    from el in doc.Elements()
    where el.Attribute("Select") != null
    select el;

// XPath expression
IEnumerable<XElement> list2 =
    ((IEnumerable)doc.XPathEvaluate("./*[@Select]")).Cast<XElement>();

if (list1.Count() == list2.Count() &&
    list1.Intersect(list2).Count() == list1.Count())
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
foreach (XElement el in list1)
    Console.WriteLine(el);
```

This example produces the following output:

```
Results are identical
<Child2 Select="true">2</Child2>
<Child4 Select="true">4</Child4>
```

# How to: Find Child Elements Based on Position (XPath-LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

Sometimes you want to find elements based on their position. You might want to find the second element, or you might want to find the third through the fifth element.

The XPath expression is:

```
Test[position() >= 2 and position() <= 4]
```

There are two approaches to writing this LINQ to XML query in a lazy way. You can use the [Skip](#) and [Take](#) operators, or you can use the [Where](#) overload that takes an index. When you use the [Where](#) overload, you use a lambda expression that takes two arguments. The following example shows both methods of selecting based on position.

## Example

This example finds the second through the fourth `Test` element. The result is a collection of elements.

This example uses the following XML document: [Sample XML File: Test Configuration \(LINQ to XML\)](#).

```
XElement testCfg = XElement.Load("TestConfig.xml");

// LINQ to XML query
IEnumerable<XElement> list1 =
    testCfg
        .Elements("Test")
        .Skip(1)
        .Take(3);

// LINQ to XML query
IEnumerable<XElement> list2 =
    testCfg
        .Elements("Test")
        .Where((el, idx) => idx >= 1 && idx <= 3);

// XPath expression
IEnumerable<XElement> list3 =
    testCfg.XPathSelectElements("Test[position() >= 2 and position() <= 4]");

if (list1.Count() == list2.Count() &&
    list1.Count() == list3.Count() &&
    list1.Intersect(list2).Count() == list1.Count() &&
    list1.Intersect(list3).Count() == list1.Count())
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
foreach (XElement el in list1)
    Console.WriteLine(el);
```

This example produces the following output:



Results are identical

```
<Test TestId="0002" TestType="CMD">
  <Name>Find succeeding characters</Name>
  <CommandLine>Examp2.EXE</CommandLine>
  <Input>abc</Input>
  <Output>def</Output>
</Test>
<Test TestId="0003" TestType="GUI">
  <Name>Convert multiple numbers to strings</Name>
  <CommandLine>Examp2.EXE /Verbose</CommandLine>
  <Input>123</Input>
  <Output>One Two Three</Output>
</Test>
<Test TestId="0004" TestType="GUI">
  <Name>Find correlated key</Name>
  <CommandLine>Examp3.EXE</CommandLine>
  <Input>a1</Input>
  <Output>b1</Output>
</Test>
```

# How to: Find the Immediate Preceding Sibling (XPath-LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

Sometimes you want to find the immediate preceding sibling to a node. Due to the difference in the semantics of positional predicates for the preceding sibling axes in XPath as opposed to LINQ to XML, this is one of the more interesting comparisons.

## Example

In this example, the LINQ to XML query uses the [Last](#) operator to find the last node in the collection returned by [ElementsBeforeSelf](#). By contrast, the XPath expression uses a predicate with a value of 1 to find the immediately preceding element.

```
XElement root = XElement.Parse(
    @"<Root>
    <Child1/>
    <Child2/>
    <Child3/>
    <Child4/>
</Root>");
XElement child4 = root.Element("Child4");

// LINQ to XML query
XElement e1 =
    child4
    .ElementsBeforeSelf()
    .Last();

// XPath expression
XElement e2 =
    ((IEnumerable)child4
        .XPathEvaluate("preceding-sibling::*[1]"))
    .Cast<XElement>()
    .First();

if (e1 == e2)
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
Console.WriteLine(e1);
```

This example produces the following output:

```
Results are identical
<Child3 />
```

# Introduction to Pure Functional Transformations (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This section introduces functional transformations, including the underlying concepts and supporting language constructs. It contrasts the object-oriented and functional transformation approaches to programming, including advice on how to transition to the latter. Although functional transformations can be used in many programming scenarios, XML transformation is used here as a concrete example.

The [Tutorial: Manipulating Content in a WordprocessingML Document \(C#\)](#) tutorial provides a series of examples, each building on the previous one. These examples demonstrate the pure functional transformational approach to manipulating XML. This tutorial assumes a working knowledge of C#. Detailed semantics of the language constructs are not provided in this tutorial, but links are provided to the language documentation as appropriate.

A working knowledge of basic computer science concepts and XML, including XML namespaces, is also assumed.

## In This Section

TOPIC	DESCRIPTION
<a href="#">Concepts and Terminology (Functional Transformation) (C#)</a>	Introduces the concepts and terminology of pure functional transformations.
<a href="#">Functional Programming vs. Imperative Programming (C#)</a>	Compares and contrasts functional programming to more traditional imperative (procedural) programming.
<a href="#">Refactoring Into Pure Functions (C#)</a>	Introduces pure functions, and shows examples of and pure and impure functions.
<a href="#">Applicability of Functional Transformation (C#)</a>	Describes typical scenarios for functional transformations.
<a href="#">Functional Transformation of XML (Visual Basic)</a>	Describes functional transformations in the context of transforming XML trees.

# Concepts and Terminology (Functional Transformation) (C#)

4/28/2019 • 3 minutes to read • [Edit Online](#)

This topic introduces the concepts and terminology of pure functional transformations. The functional transformation approach to transforming data yields code that is often quicker to program, more expressive, and easier to debug and maintain than more traditional, imperative programming.

Note that the topics in this section are not intended to fully explain functional programming. Instead, these topics identify some of the functional programming capabilities that make it easier to transform XML from one shape to another.

## What Is Pure Functional Transformation?

In *pure functional transformation*, a set of functions, called *pure functions*, define how to transform a set of structured data from its original form into another form. The word "pure" indicates that the functions are *composable*, which requires that they are:

- *Self-contained*, so that they can be freely ordered and rearranged without entanglement or interdependencies with the rest of the program. Pure transformations have no knowledge of or effect upon their environment. That is, the functions used in the transformation have no *side effects*.
- *Stateless*, so that executing the same function or specific set of functions on the same input will always result in the same output. Pure transformations have no memory of their prior use.

### IMPORTANT

In the rest of this tutorial, the term "pure function" is used in a general sense to indicate a programming approach, and not a specific language feature.

Note that pure functions must be implemented as methods in C#.

Also, you should not confuse pure functions with pure virtual methods in C++. The latter indicates that the containing class is abstract and that no method body is supplied.

## Functional Programming

*Functional programming* is a programming approach that directly supports pure functional transformation.

Historically, general-purpose functional programming languages, such as ML, Scheme, Haskell, and F#, have been primarily of interest to the academic community. Although it has always been possible to write pure functional transformations in C#, the difficulty of doing so has not made it an attractive option to most programmers. In recent versions of C#, however, new language constructs such as lambda expressions and type inference make it functional programming much easier and more productive.

For more information about functional programming, see [Functional Programming vs. Imperative Programming \(C#\)](#).

### Domain-Specific FP Languages

Although general functional programming languages have not been widely adopted, specific domain-specific functional programming languages have had better success. For example, Cascading Style Sheets (CSS) are used to determine the look and feel of many Web pages, and Extensible Stylesheet Language Transformations (XSLT) style sheets are used extensively in XML data manipulation. For more information about XSLT, see [XSLT](#)

## Terminology

The following table defines some terms related to functional transformations.

higher-order (first-class) function

A function that can be treated as a programmatic object. For example, a higher-order function can be passed to or returned from other functions. In C#, delegates and lambda expressions are language features that support higher-order functions. To write a higher-order function, you declare one or more arguments to take delegates, and you often use lambda expressions when calling it. Many of the standard query operators are higher-order functions.

For more information, see [Standard Query Operators Overview \(C#\)](#).

lambda expression

Essentially, an inline anonymous function that can be used wherever a delegate type is expected. This is a simplified definition of lambda expressions, but it is adequate for the purposes of this tutorial.

For more information about, see [Lambda Expressions](#).

collection

A structured set of data, usually of a uniform type. To be compatible with LINQ, a collection must implement the [IEnumerable](#) interface or the [IQueryable](#) interface (or one of their generic counterparts, [IEnumerator<T>](#) or [IQueryable<T>](#)).

tuple (anonymous types)

A mathematical concept, a tuple is a finite sequence of objects, each of a specific type. A tuple is also known as an ordered list. Anonymous types are a language implementation of this concept, which enable an unnamed class type to be declared and an object of that type to be instantiated at the same time.

For more information, see [Anonymous Types](#).

type inference (implicit typing)

The ability of a compiler to determine the type of a variable in the absence of an explicit type declaration.

For more information, see [Implicitly Typed Local Variables](#).

deferred execution and lazy evaluation

The delaying of evaluation of an expression until its resolved value is actually required. Deferred execution is supported in collections.

For more information, see [Introduction to LINQ Queries \(C#\)](#) and [Deferred Execution and Lazy Evaluation in LINQ to XML \(C#\)](#).

These language features will be used in code samples throughout this section.

## See also

- [Introduction to Pure Functional Transformations \(C#\)](#)
- [Functional Programming vs. Imperative Programming \(C#\)](#)

# Functional Programming vs. Imperative Programming (C#)

4/28/2019 • 3 minutes to read • [Edit Online](#)

This topic compares and contrasts functional programming with more traditional imperative (procedural) programming.

## Functional Programming vs. Imperative Programming

The *functional programming* paradigm was explicitly created to support a pure functional approach to problem solving. Functional programming is a form of *declarative programming*. In contrast, most mainstream languages, including object-oriented programming (OOP) languages such as C#, Visual Basic, C++, and Java, were designed to primarily support *imperative* (procedural) programming.

With an imperative approach, a developer writes code that describes in exacting detail the steps that the computer must take to accomplish the goal. This is sometimes referred to as *algorithmic* programming. In contrast, a functional approach involves composing the problem as a set of functions to be executed. You define carefully the input to each function, and what each function returns. The following table describes some of the general differences between these two approaches.

CHARACTERISTIC	IMPERATIVE APPROACH	FUNCTIONAL APPROACH
Programmer focus	How to perform tasks (algorithms) and how to track changes in state.	What information is desired and what transformations are required.
State changes	Important.	Non-existent.
Order of execution	Important.	Low importance.
Primary flow control	Loops, conditionals, and function (method) calls.	Function calls, including recursion.
Primary manipulation unit	Instances of structures or classes.	Functions as first-class objects and data collections.

Although most languages were designed to support a specific programming paradigm, many general languages are flexible enough to support multiple paradigms. For example, most languages that contain function pointers can be used to credibly support functional programming. Furthermore, C# includes explicit language extensions to support functional programming, including lambda expressions and type inference. LINQ technology is a form of declarative, functional programming.

## Functional Programming Using XSLT

Many XSLT developers are familiar with the pure functional approach. The most effective way to develop an XSLT style sheet is to treat each template as an isolated, composable transformation. The order of execution is completely de-emphasized. XSLT does not allow side effects (with the exception that escaping mechanisms for executing procedural code can introduce side effects that result in functional impurity). However, although XSLT is an effective tool, some of its characteristics are not optimal. For example, expressing programming constructs in XML makes code relatively verbose, and therefore difficult to maintain. Also, the heavy reliance on recursion for

flow control can result in code that is hard to read. For more information about XSLT, see [XSLT Transformations](#).

However, XSLT has proved the value of using a pure functional approach for transforming XML from one shape to another. Pure functional programming with LINQ to XML is similar in many ways to XSLT. However, the programming constructs introduced by LINQ to XML and C# allow you to write pure functional transformations that are more readable and maintainable than XSLT.

## Advantages of Pure Functions

The primary reason to implement functional transformations as pure functions is that pure functions are composable: that is, self-contained and stateless. These characteristics bring a number of benefits, including the following:

- Increased readability and maintainability. This is because each function is designed to accomplish a specific task given its arguments. The function does not rely on any external state.
- Easier reiterative development. Because the code is easier to refactor, changes to design are often easier to implement. For example, suppose you write a complicated transformation, and then realize that some code is repeated several times in the transformation. If you refactor through a pure method, you can call your pure method at will without worrying about side effects.
- Easier testing and debugging. Because pure functions can more easily be tested in isolation, you can write test code that calls the pure function with typical values, valid edge cases, and invalid edge cases.

## Transitioning for OOP Developers

In traditional object-oriented programming (OOP), most developers are accustomed to programming in the imperative/procedural style. To switch to developing in a pure functional style, they have to make a transition in their thinking and their approach to development.

To solve problems, OOP developers design class hierarchies, focus on proper encapsulation, and think in terms of class contracts. The behavior and state of object types are paramount, and language features, such as classes, interfaces, inheritance, and polymorphism, are provided to address these concerns.

In contrast, functional programming approaches computational problems as an exercise in the evaluation of pure functional transformations of data collections. Functional programming avoids state and mutable data, and instead emphasizes the application of functions.

Fortunately, C# doesn't require the full leap to functional programming, because it supports both imperative and functional programming approaches. A developer can choose which approach is most appropriate for a particular scenario. In fact, programs often combine both approaches.

## See also

- [Introduction to Pure Functional Transformations \(C#\)](#)
- [XSLT Transformations](#)
- [Refactoring Into Pure Functions \(C#\)](#)

# Refactoring Into Pure Functions (C#)

6/4/2019 • 3 minutes to read • [Edit Online](#)

An important aspect of pure functional transformations is learning how to refactor code using pure functions.

## NOTE

The common nomenclature in functional programming is that you refactor programs using pure functions. In Visual Basic and C++, this aligns with the use of functions in the respective languages. However, in C#, functions are called methods. For the purposes of this discussion, a pure function is implemented as a method in C#.

As noted previously in this section, a pure function has two useful characteristics:

- It has no side effects. The function does not change any variables or the data of any type outside of the function.
- It is consistent. Given the same set of input data, it will always return the same output value.

One way of transitioning to functional programming is to refactor existing code to eliminate unnecessary side effects and external dependencies. In this way, you can create pure function versions of existing code.

This topic discusses what a pure function is and what it is not. The [Tutorial: Manipulating Content in a WordprocessingML Document \(C#\)](#) tutorial shows how to manipulate a WordprocessingML document, and includes two examples of how to refactor using a pure function.

## Eliminating Side Effects and External Dependencies

The following examples contrast two non-pure functions and a pure function.

### Non-Pure Function that Changes a Class Member

In the following code, the `HyphenatedConcat` function is not a pure function, because it modifies the `aMember` data member in the class:

```
public class Program
{
    private static string aMember = "StringOne";

    public static void HyphenatedConcat(string appendStr)
    {
        aMember += '-' + appendStr;
    }

    public static void Main()
    {
        HyphenatedConcat("StringTwo");
        Console.WriteLine(aMember);
    }
}
```

This code produces the following output:

```
StringOne-StringTwo
```



Note that it is irrelevant whether the data being modified has `public` or `private` access, or is a `static` member or an instance member. A pure function does not change any data outside of the function.

### Non-Pure Function that Changes an Argument

Furthermore, the following version of this same function is not pure because it modifies the contents of its parameter, `sb`.

```
public class Program
{
    public static void HyphenatedConcat(StringBuilder sb, String appendStr)
    {
        sb.Append('-' + appendStr);
    }

    public static void Main()
    {
        StringBuilder sb1 = new StringBuilder("StringOne");
        HyphenatedConcat(sb1, "StringTwo");
        Console.WriteLine(sb1);
    }
}
```

This version of the program produces the same output as the first version, because the `HyphenatedConcat` function has changed the value (state) of its first parameter by invoking the `Append` member function. Note that this alteration occurs despite that fact that `HyphenatedConcat` uses call-by-value parameter passing.

#### IMPORTANT

For reference types, if you pass a parameter by value, it results in a copy of the reference to an object being passed. This copy is still associated with the same instance data as the original reference (until the reference variable is assigned to a new object). Call-by-reference is not necessarily required for a function to modify a parameter.

### Pure Function

This next version of the program shows how to implement the `HyphenatedConcat` function as a pure function.

```
class Program
{
    public static string HyphenatedConcat(string s, string appendStr)
    {
        return (s + '-' + appendStr);
    }

    public static void Main(string[] args)
    {
        string s1 = "StringOne";
        string s2 = HyphenatedConcat(s1, "StringTwo");
        Console.WriteLine(s2);
    }
}
```

Again, this version produces the same line of output: `StringOne-StringTwo`. Note that to retain the concatenated value, it is stored in the intermediate variable `s2`.

One approach that can be very useful is to write functions that are locally impure (that is, they declare and modify local variables) but are globally pure. Such functions have many of the desirable composability characteristics, but avoid some of the more convoluted functional programming idioms, such as having to use recursion when a simple loop would accomplish the same thing.

# Standard Query Operators

An important characteristic of the standard query operators is that they are implemented as pure functions.

For more information, see [Standard Query Operators Overview \(C#\)](#).

## See also

- [Introduction to Pure Functional Transformations \(C#\)](#)
- [Functional Programming vs. Imperative Programming \(C#\)](#)

# Applicability of Functional Transformation (C#)

4/28/2019 • 2 minutes to read • [Edit Online](#)

Pure functional transformations are applicable in a wide variety of situations.

The functional transformation approach is ideally suited for querying and manipulating structured data; therefore it fits well with LINQ technologies. However, functional transformation has a much wider applicability than use with LINQ. Any process where the main focus is on transforming data from one form to another should probably be considered as a candidate for functional transformation.

This approach is applicable to many problems that might not appear at first glance to be a candidate. Used in conjunction with or separately from LINQ, functional transformation should be considered for the following areas:

- XML-based documents. Well-formed data of any XML dialect can be easily manipulated through functional transformation. For more information, see [Functional Transformation of XML \(C#\)](#).
- Other structured file formats. From Windows.ini files to plain text documents, most files have some structure that lends itself to analysis and transformation.
- Data streaming protocols. Encoding data into and decoding data from communication protocols can often be represented by a simple functional transform.
- RDBMS and OODBMS data. Relational and object-oriented databases, just like XML, are widely-used structured data sources.
- Mathematic, statistic, and science solutions. These fields tend to manipulate large data sets to assist the user in visualizing, estimating, or actually solving non-trivial problems.

As described in [Refactoring Into Pure Functions \(C#\)](#), using pure functions is an example of functional programming. In addition to their immediate benefits, using pure functions provides valuable experience in thinking about problems from a functional transformation perspective. This approach can also have major impact on program and class design. This is especially true when a problem lends itself to a data transformation solution as described above.

Although they are beyond the scope of this tutorial, designs that are influenced by the functional transformation perspective tend to center on processes more than on objects as actors, and the resulting solution tends to be implemented as series of large-scale transformations, rather than individual object state changes.

Again, remember that C# supports both imperative and functional approaches, so the best design for your application might incorporate elements of both.

## See also

- [Introduction to Pure Functional Transformations \(C#\)](#)
- [Functional Transformation of XML \(C#\)](#)
- [Refactoring Into Pure Functions \(C#\)](#)

# Functional Transformation of XML (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic discusses the pure functional transformation approach to modifying XML documents, and contrasts it with a procedural approach.

## Modifying an XML Document

One of the most common tasks for an XML programmer is transforming XML from one shape to another. The shape of an XML document is the structure of the document, which includes the following:

- The hierarchy expressed by the document.
- The element and attribute names.
- The data types of the elements and attributes.

In general, the most effective approach to transforming XML from one shape to another is that of pure functional transformation. In this approach, the primary programmer task is to create a transformation which is applied to the entire XML document (or to one or more strictly defined nodes). Functional transformation is arguably the easiest to code (after a programmer is familiar with the approach), yields the most maintainable code, and is often more compact than alternative approaches.

### XML Functional Transformational Technologies

Microsoft offers two functional transformation technologies for use on XML documents: XSLT and LINQ to XML. XSLT is supported in the [System.Xml.Xsl](#) managed namespace and in the native COM implementation of MSXML. Although XSLT is a robust technology for manipulating XML documents, it requires expertise in a specialized domain, namely the XSLT language and its supporting APIs.

LINQ to XML provides the tools necessary to code pure functional transformations in an expressive and powerful way, within C# or Visual Basic code. For example, many of the examples in the LINQ to XML documentation use a pure functional approach. Also, in the [Tutorial: Manipulating Content in a WordprocessingML Document \(C#\)](#) tutorial, we use LINQ to XML in a functional approach to manipulate information in a Microsoft Word document.

For a more complete comparison of LINQ to XML with other Microsoft XML technologies, see [LINQ to XML vs. Other XML Technologies](#).

XSLT is the recommended tool for document-centric transformations when the source document has an irregular structure. However, LINQ to XML can also perform document-centric transforms. For more information, see [How to: Use Annotations to Transform LINQ to XML Trees in an XSLT Style \(C#\)](#).

## See also

- [Introduction to Pure Functional Transformations \(C#\)](#)
- [Tutorial: Manipulating Content in a WordprocessingML Document \(C#\)](#)
- [LINQ to XML vs. Other XML Technologies](#)

# Deferred Execution and Lazy Evaluation in LINQ to XML (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

Query and axis operations are often implemented to use deferred execution. This topic explains the requirements and advantages of deferred execution, and some implementation considerations.

## Deferred Execution

Deferred execution means that the evaluation of an expression is delayed until its *realized* value is actually required. Deferred execution can greatly improve performance when you have to manipulate large data collections, especially in programs that contain a series of chained queries or manipulations. In the best case, deferred execution enables only a single iteration through the source collection.

The LINQ technologies make extensive use of deferred execution in both the members of core [System.Linq](#) classes and in the extension methods in the various LINQ namespaces, such as [System.Xml.Linq.Extensions](#).

Deferred execution is supported directly in the C# language by the [yield](#) keyword (in the form of the `yield-return` statement) when used within an iterator block. Such an iterator must return a collection of type [IEnumerator](#) or [IEnumerator<T>](#) (or a derived type).

## Eager vs. Lazy Evaluation

When you write a method that implements deferred execution, you also have to decide whether to implement the method using lazy evaluation or eager evaluation.

- In *lazy evaluation*, a single element of the source collection is processed during each call to the iterator. This is the typical way in which iterators are implemented.
- In *eager evaluation*, the first call to the iterator will result in the entire collection being processed. A temporary copy of the source collection might also be required. For example, the [OrderBy](#) method has to sort the entire collection before it returns the first element.

Lazy evaluation usually yields better performance because it distributes overhead processing evenly throughout the evaluation of the collection and minimizes the use of temporary data. Of course, for some operations, there is no other option than to materialize intermediate results.

## Next Steps

The next topic in this tutorial illustrates deferred execution:

- [Deferred Execution Example \(C#\)](#)

## See also

- [Tutorial: Chaining Queries Together \(C#\)](#)
- [Concepts and Terminology \(Functional Transformation\) \(C#\)](#)
- [Aggregation Operations \(C#\)](#)
- [yield](#)

# Deferred Execution Example (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how deferred execution and lazy evaluation affect the execution of your LINQ to XML queries.

## Example

The following example shows the order of execution when using an extension method that uses deferred execution. The example declares an array of three strings. It then iterates through the collection returned by

`ConvertCollectionToUpperCase`.

```
public static class LocalExtensions
{
    public static IEnumerable<string>
        ConvertCollectionToUpperCase(this IEnumerable<string> source)
    {
        foreach (string str in source)
        {
            Console.WriteLine("ToUpper: source {0}", str);
            yield return str.ToUpper();
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        string[] stringArray = { "abc", "def", "ghi" };

        var q = from str in stringArray.ConvertCollectionToUpperCase()
                select str;

        foreach (string str in q)
            Console.WriteLine("Main: str {0}", str);
    }
}
```

This example produces the following output:

```
ToUpper: source abc
Main: str ABC
ToUpper: source def
Main: str DEF
ToUpper: source ghi
Main: str GHI
```

Notice that when iterating through the collection returned by `ConvertCollectionToUpperCase`, each item is retrieved from the source string array and converted to uppercase before the next item is retrieved from the source string array.

You can see that the entire array of strings is not converted to uppercase before each item in the returned collection is processed in the `foreach` loop in `Main`.

The next topic in this tutorial illustrates chaining queries together:

- [Chaining Queries Example \(C#\)](#)

## See also

- [Tutorial: Chaining Queries Together \(C#\)](#)

# Chaining Queries Example (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This example builds on the previous example and shows what happens when you chain together two queries that both use deferred execution and lazy evaluation.

## Example

In this example, another extension method is introduced, `AppendString`, which appends a specified string onto every string in the source collection, and then yields the new strings.

```
public static class LocalExtensions
{
    public static IEnumerable<string>
        ConvertCollectionToUpperCase(this IEnumerable<string> source)
    {
        foreach (string str in source)
        {
            Console.WriteLine("ToUpper: source >{0}<", str);
            yield return str.ToUpper();
        }
    }

    public static IEnumerable<string>
        AppendString(this IEnumerable<string> source, string stringToAppend)
    {
        foreach (string str in source)
        {
            Console.WriteLine("AppendString: source >{0}<", str);
            yield return str + stringToAppend;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        string[] stringArray = { "abc", "def", "ghi" };

        IEnumerable<string> q1 =
            from s in stringArray.ConvertCollectionToUpperCase()
            select s;

        IEnumerable<string> q2 =
            from s in q1.AppendString("!!!")
            select s;

        foreach (string str in q2)
        {
            Console.WriteLine("Main: str >{0}<", str);
            Console.WriteLine();
        }
    }
}
```

This example produces the following output:



```
ToUpper: source >abc<
AppendString: source >ABC<
Main: str >ABC!!!<

ToUpper: source >def<
AppendString: source >DEF<
Main: str >DEF!!!<

ToUpper: source >ghi<
AppendString: source >GHI<
Main: str >GHI!!!<
```

In this example, you can see that each extension method operates one at a time for each item in the source collection.

What should be clear from this example is that even though we have chained together queries that yield collections, no intermediate collections are materialized. Instead, each item is passed from one lazy method to the next. This results in a much smaller memory footprint than an approach that would first take one array of strings, then create a second array of strings that have been converted to uppercase, and finally create a third array of strings where each string has the exclamation points appended to it.

The next topic in this tutorial illustrates intermediate materialization:

- [Intermediate Materialization \(C#\)](#)

## See also

- [Tutorial: Chaining Queries Together \(C#\)](#)

# Intermediate Materialization (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

If you are not careful, in some situations you can drastically alter the memory and performance profile of your application by causing premature materialization of collections in your queries. Some standard query operators cause materialization of their source collection before yielding a single element. For example, [Enumerable.OrderBy](#) first iterates through its entire source collection, then sorts all items, and then finally yields the first item. This means that it is expensive to get the first item of an ordered collection; each item thereafter is not expensive. This makes sense: It would be impossible for that query operator to do otherwise.

## Example

This example alters the previous example. The `AppendString` method calls [ToList](#) before iterating through the source. This causes materialization.

```

public static class LocalExtensions
{
    public static IEnumerable<string>
        ConvertCollectionToUpperCase(this IEnumerable<string> source)
    {
        foreach (string str in source)
        {
            Console.WriteLine("ToUpper: source >{0}<", str);
            yield return str.ToUpper();
        }
    }

    public static IEnumerable<string>
        AppendString(this IEnumerable<string> source, string stringToAppend)
    {
        // the following statement materializes the source collection in a List<T>
        // before iterating through it
        foreach (string str in source.ToList())
        {
            Console.WriteLine("AppendString: source >{0}<", str);
            yield return str + stringToAppend;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        string[] stringArray = { "abc", "def", "ghi" };

        IEnumerable<string> q1 =
            from s in stringArray.ConvertCollectionToUpperCase()
            select s;

        IEnumerable<string> q2 =
            from s in q1.AppendString("!!!")
            select s;

        foreach (string str in q2)
        {
            Console.WriteLine("Main: str >{0}<", str);
            Console.WriteLine();
        }
    }
}

```

This example produces the following output:

```

ToUpper: source >abc<
ToUpper: source >def<
ToUpper: source >ghi<
AppendString: source >ABC<
Main: str >ABC!!!<

AppendString: source >DEF<
Main: str >DEF!!!<

AppendString: source >GHI<
Main: str >GHI!!!<

```

In this example, you can see that the call to [ToList](#) causes `AppendString` to enumerate its entire source before yielding the first item. If the source were a large array, this would significantly alter the memory profile of the application.

Standard query operators can also be chained together. The final topic in this tutorial illustrates this.

- [Chaining Standard Query Operators Together \(C#\)](#)

## See also

- [Tutorial: Chaining Queries Together \(C#\)](#)

# Chaining Standard Query Operators Together (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This is the final topic in the [Tutorial: Chaining Queries Together \(C#\)](#) tutorial.

The standard query operators can also be chained together. For example, you can interject the [Enumerable.Where](#) operator, and it also operates in a lazy fashion. No intermediate results are materialized by it.

## Example

In this example, the [Where](#) method is called before calling `ConvertCollectionToUpperCase`. The [Where](#) method operates in almost exactly the same way as the lazy methods used in previous examples in this tutorial, `ConvertCollectionToUpperCase` and `AppendString`.

One difference is that in this case, the [Where](#) method iterates through its source collection, determines that the first item does not pass the predicate, and then gets the next item, which does pass. It then yields the second item.

However, the basic idea is the same: Intermediate collections are not materialized unless they have to be.

When query expressions are used, they are converted to calls to the standard query operators, and the same principles apply.

All of the examples in this section that are querying Office Open XML documents use the same principle. Deferred execution and lazy evaluation are some of the fundamental concepts that you must understand to use LINQ (and LINQ to XML) effectively.

```

public static class LocalExtensions
{
    public static IEnumerable<string>
        ConvertCollectionToUpperCase(this IEnumerable<string> source)
    {
        foreach (string str in source)
        {
            Console.WriteLine("ToUpper: source >{0}<", str);
            yield return str.ToUpper();
        }
    }

    public static IEnumerable<string>
        AppendString(this IEnumerable<string> source, string stringToAppend)
    {
        foreach (string str in source)
        {
            Console.WriteLine("AppendString: source >{0}<", str);
            yield return str + stringToAppend;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        string[] stringArray = { "abc", "def", "ghi" };

        IEnumerable<string> q1 =
            from s in stringArray.ConvertCollectionToUpperCase()
            where s.CompareTo("D") >= 0
            select s;

        IEnumerable<string> q2 =
            from s in q1.AppendString("!!!")
            select s;

        foreach (string str in q2)
        {
            Console.WriteLine("Main: str >{0}<", str);
            Console.WriteLine();
        }
    }
}

```

This example produces the following output:

```

ToUpper: source >abc<
ToUpper: source >def<
AppendString: source >DEF<
Main: str >DEF!!!<

ToUpper: source >ghi<
AppendString: source >GHI<
Main: str >GHI!!!<

```

# Deferred Execution and Lazy Evaluation in LINQ to XML (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

Query and axis operations are often implemented to use deferred execution. This topic explains the requirements and advantages of deferred execution, and some implementation considerations.

## Deferred Execution

Deferred execution means that the evaluation of an expression is delayed until its *realized* value is actually required. Deferred execution can greatly improve performance when you have to manipulate large data collections, especially in programs that contain a series of chained queries or manipulations. In the best case, deferred execution enables only a single iteration through the source collection.

The LINQ technologies make extensive use of deferred execution in both the members of core [System.Linq](#) classes and in the extension methods in the various LINQ namespaces, such as [System.Xml.Linq.Extensions](#).

Deferred execution is supported directly in the C# language by the [yield](#) keyword (in the form of the `yield-return` statement) when used within an iterator block. Such an iterator must return a collection of type [IEnumerator](#) or [IEnumerator<T>](#) (or a derived type).

## Eager vs. Lazy Evaluation

When you write a method that implements deferred execution, you also have to decide whether to implement the method using lazy evaluation or eager evaluation.

- In *lazy evaluation*, a single element of the source collection is processed during each call to the iterator. This is the typical way in which iterators are implemented.
- In *eager evaluation*, the first call to the iterator will result in the entire collection being processed. A temporary copy of the source collection might also be required. For example, the [OrderBy](#) method has to sort the entire collection before it returns the first element.

Lazy evaluation usually yields better performance because it distributes overhead processing evenly throughout the evaluation of the collection and minimizes the use of temporary data. Of course, for some operations, there is no other option than to materialize intermediate results.

## Next Steps

The next topic in this tutorial illustrates deferred execution:

- [Deferred Execution Example \(C#\)](#)

## See also

- [Tutorial: Chaining Queries Together \(C#\)](#)
- [Concepts and Terminology \(Functional Transformation\) \(C#\)](#)
- [Aggregation Operations \(C#\)](#)
- [yield](#)

# Deferred Execution Example (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how deferred execution and lazy evaluation affect the execution of your LINQ to XML queries.

## Example

The following example shows the order of execution when using an extension method that uses deferred execution. The example declares an array of three strings. It then iterates through the collection returned by

`ConvertCollectionToUpperCase`.

```
public static class LocalExtensions
{
    public static IEnumerable<string>
        ConvertCollectionToUpperCase(this IEnumerable<string> source)
    {
        foreach (string str in source)
        {
            Console.WriteLine("ToUpper: source {0}", str);
            yield return str.ToUpper();
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        string[] stringArray = { "abc", "def", "ghi" };

        var q = from str in stringArray.ConvertCollectionToUpperCase()
                select str;

        foreach (string str in q)
            Console.WriteLine("Main: str {0}", str);
    }
}
```

This example produces the following output:

```
ToUpper: source abc
Main: str ABC
ToUpper: source def
Main: str DEF
ToUpper: source ghi
Main: str GHI
```

Notice that when iterating through the collection returned by `ConvertCollectionToUpperCase`, each item is retrieved from the source string array and converted to uppercase before the next item is retrieved from the source string array.

You can see that the entire array of strings is not converted to uppercase before each item in the returned collection is processed in the `foreach` loop in `Main`.

The next topic in this tutorial illustrates chaining queries together:



- [Chaining Queries Example \(C#\)](#)

## See also

- [Tutorial: Chaining Queries Together \(C#\)](#)

# Chaining Queries Example (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This example builds on the previous example and shows what happens when you chain together two queries that both use deferred execution and lazy evaluation.

## Example

In this example, another extension method is introduced, `AppendString`, which appends a specified string onto every string in the source collection, and then yields the new strings.

```
public static class LocalExtensions
{
    public static IEnumerable<string>
        ConvertCollectionToUpperCase(this IEnumerable<string> source)
    {
        foreach (string str in source)
        {
            Console.WriteLine("ToUpper: source >{0}<", str);
            yield return str.ToUpper();
        }
    }

    public static IEnumerable<string>
        AppendString(this IEnumerable<string> source, string stringToAppend)
    {
        foreach (string str in source)
        {
            Console.WriteLine("AppendString: source >{0}<", str);
            yield return str + stringToAppend;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        string[] stringArray = { "abc", "def", "ghi" };

        IEnumerable<string> q1 =
            from s in stringArray.ConvertCollectionToUpperCase()
            select s;

        IEnumerable<string> q2 =
            from s in q1.AppendString("!!!")
            select s;

        foreach (string str in q2)
        {
            Console.WriteLine("Main: str >{0}<", str);
            Console.WriteLine();
        }
    }
}
```

This example produces the following output:

```
ToUpper: source >abc<
AppendString: source >ABC<
Main: str >ABC!!!<

ToUpper: source >def<
AppendString: source >DEF<
Main: str >DEF!!!<

ToUpper: source >ghi<
AppendString: source >GHI<
Main: str >GHI!!!<
```

In this example, you can see that each extension method operates one at a time for each item in the source collection.

What should be clear from this example is that even though we have chained together queries that yield collections, no intermediate collections are materialized. Instead, each item is passed from one lazy method to the next. This results in a much smaller memory footprint than an approach that would first take one array of strings, then create a second array of strings that have been converted to uppercase, and finally create a third array of strings where each string has the exclamation points appended to it.

The next topic in this tutorial illustrates intermediate materialization:

- [Intermediate Materialization \(C#\)](#)

## See also

- [Tutorial: Chaining Queries Together \(C#\)](#)

# Intermediate Materialization (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

If you are not careful, in some situations you can drastically alter the memory and performance profile of your application by causing premature materialization of collections in your queries. Some standard query operators cause materialization of their source collection before yielding a single element. For example, [Enumerable.OrderBy](#) first iterates through its entire source collection, then sorts all items, and then finally yields the first item. This means that it is expensive to get the first item of an ordered collection; each item thereafter is not expensive. This makes sense: It would be impossible for that query operator to do otherwise.

## Example

This example alters the previous example. The `AppendString` method calls [ToList](#) before iterating through the source. This causes materialization.

```

public static class LocalExtensions
{
    public static IEnumerable<string>
        ConvertCollectionToUpperCase(this IEnumerable<string> source)
    {
        foreach (string str in source)
        {
            Console.WriteLine("ToUpper: source >{0}<", str);
            yield return str.ToUpper();
        }
    }

    public static IEnumerable<string>
        AppendString(this IEnumerable<string> source, string stringToAppend)
    {
        // the following statement materializes the source collection in a List<T>
        // before iterating through it
        foreach (string str in source.ToList())
        {
            Console.WriteLine("AppendString: source >{0}<", str);
            yield return str + stringToAppend;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        string[] stringArray = { "abc", "def", "ghi" };

        IEnumerable<string> q1 =
            from s in stringArray.ConvertCollectionToUpperCase()
            select s;

        IEnumerable<string> q2 =
            from s in q1.AppendString("!!!")
            select s;

        foreach (string str in q2)
        {
            Console.WriteLine("Main: str >{0}<", str);
            Console.WriteLine();
        }
    }
}

```

This example produces the following output:

```

ToUpper: source >abc<
ToUpper: source >def<
ToUpper: source >ghi<
AppendString: source >ABC<
Main: str >ABC!!!<

AppendString: source >DEF<
Main: str >DEF!!!<

AppendString: source >GHI<
Main: str >GHI!!!<

```

In this example, you can see that the call to [ToList](#) causes `AppendString` to enumerate its entire source before yielding the first item. If the source were a large array, this would significantly alter the memory profile of the application.

Standard query operators can also be chained together. The final topic in this tutorial illustrates this.

- [Chaining Standard Query Operators Together \(C#\)](#)

## See also

- [Tutorial: Chaining Queries Together \(C#\)](#)

# Chaining Standard Query Operators Together (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This is the final topic in the [Tutorial: Chaining Queries Together \(C#\)](#) tutorial.

The standard query operators can also be chained together. For example, you can interject the [Enumerable.Where](#) operator, and it also operates in a lazy fashion. No intermediate results are materialized by it.

## Example

In this example, the [Where](#) method is called before calling `ConvertCollectionToUpperCase`. The [Where](#) method operates in almost exactly the same way as the lazy methods used in previous examples in this tutorial, `ConvertCollectionToUpperCase` and `AppendString`.

One difference is that in this case, the [Where](#) method iterates through its source collection, determines that the first item does not pass the predicate, and then gets the next item, which does pass. It then yields the second item.

However, the basic idea is the same: Intermediate collections are not materialized unless they have to be.

When query expressions are used, they are converted to calls to the standard query operators, and the same principles apply.

All of the examples in this section that are querying Office Open XML documents use the same principle. Deferred execution and lazy evaluation are some of the fundamental concepts that you must understand to use LINQ (and LINQ to XML) effectively.

```

public static class LocalExtensions
{
    public static IEnumerable<string>
        ConvertCollectionToUpperCase(this IEnumerable<string> source)
    {
        foreach (string str in source)
        {
            Console.WriteLine("ToUpper: source >{0}<", str);
            yield return str.ToUpper();
        }
    }

    public static IEnumerable<string>
        AppendString(this IEnumerable<string> source, string stringToAppend)
    {
        foreach (string str in source)
        {
            Console.WriteLine("AppendString: source >{0}<", str);
            yield return str + stringToAppend;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        string[] stringArray = { "abc", "def", "ghi" };

        IEnumerable<string> q1 =
            from s in stringArray.ConvertCollectionToUpperCase()
            where s.CompareTo("D") >= 0
            select s;

        IEnumerable<string> q2 =
            from s in q1.AppendString("!!!")
            select s;

        foreach (string str in q2)
        {
            Console.WriteLine("Main: str >{0}<", str);
            Console.WriteLine();
        }
    }
}

```

This example produces the following output:

```

ToUpper: source >abc<
ToUpper: source >def<
AppendString: source >DEF<
Main: str >DEF!!!<

ToUpper: source >ghi<
AppendString: source >GHI<
Main: str >GHI!!!<

```



# Shape of WordprocessingML Documents (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic introduces the XML shape of a WordprocessingML document.

## Microsoft Office Formats

The native file format for the 2007 Microsoft Office system is Office Open XML (commonly called Open XML). Open XML is an XML-based format that an Ecma standard and is currently going through the ISO-IEC standards process. The markup language for word processing files within Open XML is called WordprocessingML. This tutorial uses WordprocessingML source files as input for the examples.

If you are using Microsoft Office 2003, you can save documents in the Office Open XML format if you have installed the Microsoft Office Compatibility Pack for Word, Excel, and PowerPoint 2007 File Formats.

## The Shape of WordprocessingML Documents

The first thing to understand is the shape of WordprocessingML documents. A WordprocessingML document contains a body element (named `w:body`) that contains the paragraphs of the document. Each paragraph contains one or more text runs (named `w:r`). Each text run contains one or more text pieces (named `w:t`).

The following is a very simple WordprocessingML document:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<w:document
  xmlns:ve="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:o="urn:schemas-microsoft-com:office:office"
  xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships"
  xmlns:m="http://schemas.openxmlformats.org/officeDocument/2006/math"
  xmlns:v="urn:schemas-microsoft-com:vml"
  xmlns:wp="http://schemas.openxmlformats.org/drawingml/2006/wordprocessingDrawing"
  xmlns:w10="urn:schemas-microsoft-com:office:word"
  xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main"
  xmlns:wne="http://schemas.microsoft.com/office/word/2006/wordml">
  <w:body>
    <w:p w:rsidR="00E22EB6"
      w:rsidRDefault="00E22EB6">
      <w:r>
        <w:t>This is a paragraph.</w:t>
      </w:r>
    </w:p>
    <w:p w:rsidR="00E22EB6"
      w:rsidRDefault="00E22EB6">
      <w:r>
        <w:t>This is another paragraph.</w:t>
      </w:r>
    </w:p>
  </w:body>
</w:document>
```

This document contains two paragraphs. They both contain a single text run, and each text run contains a single text piece.

The easiest way to see the contents of a WordprocessingML document in XML form is to create one using Microsoft Word, save it, and then run the following program that prints the XML to the console.

This example uses classes found in the WindowsBase assembly. It uses types in the [System.IO.Packaging](#) namespace.

```
const string documentRelationshipType =
    "http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
const string wordmlNamespace =
    "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
XNamespace w = wordmlNamespace;

using (Package wdPackage = Package.Open("SampleDoc.docx", FileMode.Open, FileAccess.Read))
{
    PackageRelationship relationship =
        wdPackage
            .GetRelationshipsByType(documentRelationshipType)
            .FirstOrDefault();
    if (relationship != null)
    {
        Uri documentUri =
            PackUriHelper.ResolvePartUri(
                new Uri("/", UriKind.Relative),
                relationship.TargetUri);
        PackagePart documentPart = wdPackage.GetPart(documentUri);

        // Get the officeDocument part from the package.
        // Load the XML in the part into an XDocument instance.
        XDocument xdoc =
            XDocument.Load(XmlReader.Create(documentPart.GetStream()));
        Console.WriteLine(xdoc.Root);
    }
}
```

## External Resources

[Introducing the Office \(2007\) Open XML File Formats](#)

[Overview of WordprocessingML](#)

[Anatomy of a WordProcessingML File](#)

[Introduction to WordprocessingML](#)

[Office 2003: XML Reference Schemas Download page](#)

## See also

- [Tutorial: Manipulating Content in a WordprocessingML Document \(C#\)](#)

# Creating the Source Office Open XML Document (C#)

6/5/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how to create the Office Open XML WordprocessingML document that the other examples in this tutorial use. If you follow these instructions, your output will match the output provided in each example.

However, the examples in this tutorial will work with any valid WordprocessingML document.

To create the document that this tutorial uses, you must either have Microsoft Office 2007 or later installed, or you must have Microsoft Office 2003 with the Microsoft Office Compatibility Pack for Word, Excel, and PowerPoint 2007 File Formats.

## Creating the WordprocessingML Document

**To create the WordprocessingML document**

1. Create a new Microsoft Word document.
2. Paste the following text into the new document:

```
Parsing WordprocessingML with LINQ to XML

The following example prints to the console.

using System;

class Program {
    public static void Main(string[] args) {
        Console.WriteLine("Hello World");
    }
}

This example produces the following output:

Hello World
```

3. Format the first line with the style "Heading 1".
4. Select the lines that contain the C# code. The first line starts with the `using` keyword. The last line is the last closing brace. Format the lines with the courier font. Format them with a new style, and name the new style "Code".
5. Finally, select the entire line that contains the output, and format it with the `Code` style.
6. Save the document, and name it SampleDoc.docx.

### NOTE

If you are using Microsoft Word 2003, select **Word 2007 Document** in the **Save as Type** drop-down list.

# Finding the Default Paragraph Style (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

The first task in the Manipulating Information in a WordprocessingML Document tutorial is to find the default style of paragraphs in the document.

## Example

### Description

The following example opens an Office Open XML WordprocessingML document, finds the document and style parts of the package, and then executes a query that finds the default style name. For information about Office Open XML document packages, and the parts they consist of, see [Details of Office Open XML WordprocessingML Documents \(C#\)](#).

The query finds a node named `w:style` that has an attribute named `w:type` with a value of "paragraph", and also has an attribute named `w:default` with a value of "1". Because there will be only one XML node with these attributes, the query uses the [Enumerable.First](#) operator to convert a collection to a singleton. It then gets the value of the attribute with the name `w:styleId`.

This example uses classes from the WindowsBase assembly. It uses types in the [System.IO.Packaging](#) namespace.

### Code

```

const string fileName = "SampleDoc.docx";

const string documentRelationshipType =
    "http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
const string stylesRelationshipType =
    "http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
const string wordmlNamespace =
    "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
XNamespace w = wordmlNamespace;

XDocument xDoc = null;
XDocument styleDoc = null;

using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.Read))
{
    PackageRelationship docPackageRelationship =
        wdPackage.GetRelationshipsByType(documentRelationshipType).FirstOrDefault();
    if (docPackageRelationship != null)
    {
        Uri documentUri = PackUriHelper.ResolvePartUri(new Uri("/"), UriKind.Relative),
            docPackageRelationship.TargetUri);
        PackagePart documentPart = wdPackage.GetPart(documentUri);

        // Load the document XML in the part into an XDocument instance.
        xDoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

        // Find the styles part. There will only be one.
        PackageRelationship styleRelation =
            documentPart.GetRelationshipsByType(stylesRelationshipType).FirstOrDefault();
        if (styleRelation != null)
        {
            Uri styleUri = PackUriHelper.ResolvePartUri(documentUri, styleRelation.TargetUri);
            PackagePart stylePart = wdPackage.GetPart(styleUri);

            // Load the style XML in the part into an XDocument instance.
            styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));
        }
    }
}

// The following query finds all the paragraphs that have the default style.
string defaultStyle =
    (string)(
        from style in styleDoc.Root.Elements(w + "style")
        where (string)style.Attribute(w + "type") == "paragraph" &&
            (string)style.Attribute(w + "default") == "1"
        select style
    ).First().Attribute(w + "styleId");

Console.WriteLine("The default style is: {0}", defaultStyle);

```

## Comments

This example produces the following output:

```
The default style is: Normal
```

## Next Steps

In the next example, you'll create a similar query that finds all the paragraphs in a document and their styles:

- [Retrieving the Paragraphs and Their Styles \(C#\)](#)

# Retrieving the Paragraphs and Their Styles (C#)

6/4/2019 • 3 minutes to read • [Edit Online](#)

In this example, we write a query that retrieves the paragraph nodes from a WordprocessingML document. It also identifies the style of each paragraph.

This query builds on the query in the previous example, [Finding the Default Paragraph Style \(C#\)](#), which retrieves the default style from the list of styles. This information is required so that the query can identify the style of paragraphs that do not have a style explicitly set. Paragraph styles are set through the `w:pPr` element; if a paragraph does not contain this element, it is formatted with the default style.

This topic explains the significance of some pieces of the query, then shows the query as part of a complete, working example.

## Example

The source of the query to retrieve all the paragraphs in a document and their styles is as follows:

```
xDoc.Root.Element(w + "body").Descendants(w + "p")
```

This expression is similar to the source of the query in the previous example, [Finding the Default Paragraph Style \(C#\)](#). The main difference is that it uses the `Descendants` axis instead of the `Elements` axis. The query uses the `Descendants` axis because in documents that have sections, the paragraphs will not be the direct children of the body element; rather, the paragraphs will be two levels down in the hierarchy. By using the `Descendants` axis, the code will work of whether or not the document uses sections.

## Example

The query uses a `let` clause to determine the element that contains the style node. If there is no element, then `styleNode` is set to `null`:

```
let styleNode = para.Elements(w + "pPr").Elements(w + "pStyle").FirstOrDefault()
```

The `let` clause first uses the `Elements` axis to find all elements named `pPr`, then uses the `Elements` extension method to find all child elements named `pStyle`, and finally uses the `FirstOrDefault` standard query operator to convert the collection to a singleton. If the collection is empty, `styleNode` is set to `null`. This is a useful idiom to look for the `pStyle` descendant node. Note that if the `pPr` child node does not exist, the code does not fail by throwing an exception; instead, `styleNode` is set to `null`, which is the desired behavior of this `let` clause.

The query projects a collection of an anonymous type with two members, `StyleName` and `ParagraphNode`.

## Example

This example processes a WordprocessingML document, retrieving the paragraph nodes from a WordprocessingML document. It also identifies the style of each paragraph. This example builds on the previous examples in this tutorial. The new query is called out in comments in the code below.

You can find instructions for creating the source document for this example in [Creating the Source Office Open XML Document \(C#\)](#).

This example uses classes found in the WindowsBase assembly. It uses types in the [System.IO.Packaging](#) namespace.

```
const string fileName = "SampleDoc.docx";

const string documentRelationshipType =
"http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
const string stylesRelationshipType =
"http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
const string wordmlNamespace = "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
XNamespace w = wordmlNamespace;

XDocument xDoc = null;
XDocument styleDoc = null;

using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.Read))
{
    PackageRelationship docPackageRelationship =
wdPackage.GetRelationshipsByType(documentRelationshipType).FirstOrDefault();
    if (docPackageRelationship != null)
    {
        Uri documentUri = PackUriHelper.ResolvePartUri(new Uri("/", UriKind.Relative),
docPackageRelationship.TargetUri);
        PackagePart documentPart = wdPackage.GetPart(documentUri);

        // Load the document XML in the part into an XDocument instance.
        xDoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

        // Find the styles part. There will only be one.
        PackageRelationship styleRelation =
documentPart.GetRelationshipsByType(stylesRelationshipType).FirstOrDefault();
        if (styleRelation != null)
        {
            Uri styleUri = PackUriHelper.ResolvePartUri(documentUri, styleRelation.TargetUri);
            PackagePart stylePart = wdPackage.GetPart(styleUri);

            // Load the style XML in the part into an XDocument instance.
            styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));
        }
    }
}

string defaultStyle =
    (string)(
        from style in styleDoc.Root.Elements(w + "style")
        where (string)style.Attribute(w + "type") == "paragraph"&&
            (string)style.Attribute(w + "default") == "1"
        select style
    ).First().Attribute(w + "styleId");

// Following is the new query that finds all paragraphs in the
// document and their styles.
var paragraphs =
    from para in xDoc
        .Root
        .Element(w + "body")
        .Descendants(w + "p")
    let styleNode = para
        .Elements(w + "pPr")
        .Elements(w + "pStyle")
        .FirstOrDefault()
    select new
    {
        ParagraphNode = para,
        StyleName = styleNode != null ?
            (string)styleNode.Attribute(w + "val") :
            defaultStyle
    }
}
```

```
};  
  
foreach (var p in paragraphs)  
    Console.WriteLine("StyleName:{0}", p.StyleName);
```

This example produces the following output when applied to the document described in [Creating the Source Office Open XML Document \(C#\)](#).

```
StyleName:Heading1  
StyleName:Normal  
StyleName:Normal  
StyleName:Normal  
StyleName:Code  
StyleName:Code  
StyleName:Code  
StyleName:Code  
StyleName:Code  
StyleName:Code  
StyleName:Code  
StyleName:Code  
StyleName:Code  
StyleName:Normal  
StyleName:Normal  
StyleName:Normal  
StyleName:Code
```

## Next Steps

In the next topic, [Retrieving the Text of the Paragraphs \(C#\)](#), you'll create a query to retrieve the text of paragraphs.

## See also

- [Tutorial: Manipulating Content in a WordprocessingML Document \(C#\)](#)



# Retrieving the Text of the Paragraphs (C#)

4/28/2019 • 3 minutes to read • [Edit Online](#)

This example builds on the previous example, [Retrieving the Paragraphs and Their Styles \(C#\)](#). This new example retrieves the text of each paragraph as a string.

To retrieve the text, this example adds an additional query that iterates through the collection of anonymous types and projects a new collection of an anonymous type with the addition of a new member, `Text`. It uses the [Aggregate](#) standard query operator to concatenate multiple strings into one string.

This technique (that is, first projecting to a collection of an anonymous type, then using this collection to project to a new collection of an anonymous type) is a common and useful idiom. This query could have been written without projecting to the first anonymous type. However, because of lazy evaluation, doing so does not use much additional processing power. The idiom creates more short lived objects on the heap, but this does not substantially degrade performance.

Of course, it would be possible to write a single query that contains the functionality to retrieve the paragraphs, the style of each paragraph, and the text of each paragraph. However, it often is useful to break up a more complicated query into multiple queries because the resulting code is more modular and easier to maintain. Furthermore, if you need to reuse a portion of the query, it is easier to refactor if the queries are written in this manner.

These queries, which are chained together, use the processing model that is examined in detail in the topic [Tutorial: Chaining Queries Together \(C#\)](#).

## Example

This example processes a WordprocessingML document, determining the element node, the style name, and the text of each paragraph. This example builds on the previous examples in this tutorial. The new query is called out in comments in the code below.

For instructions for creating the source document for this example, see [Creating the Source Office Open XML Document \(C#\)](#).

This example uses classes from the WindowsBase assembly. It uses types in the [System.IO.Packaging](#) namespace.

```
const string fileName = "SampleDoc.docx";

const string documentRelationshipType =
    "http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
const string stylesRelationshipType =
    "http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
const string wordmlNamespace =
    "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
XNamespace w = wordmlNamespace;

XDocument xDoc = null;
XDocument styleDoc = null;

using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.Read))
{
    PackageRelationship docPackageRelationship =
        wdPackage.GetRelationshipsByType(documentRelationshipType).FirstOrDefault();
    if (docPackageRelationship != null)
    {
        Uri documentUri = PackUriHelper.ResolvePartUri(new Uri("/", UriKind.Relative),
            docPackageRelationship.TargetUri);
        XDocument documentDoc = Package.Open(documentUri, FileMode.Open, FileAccess.Read);
    }
}
```

```

PackagePart documentPart = wdPackage.GetPart(documentUri);

// Load the document XML in the part into an XDocument instance.
xDoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

// Find the styles part. There will only be one.
PackageRelationship styleRelation =
    documentPart.GetRelationshipsByType(stylesRelationshipType).FirstOrDefault();
if (styleRelation != null)
{
    Uri styleUri = PackUriHelper.ResolvePartUri(documentUri, styleRelation.TargetUri);
    PackagePart stylePart = wdPackage.GetPart(styleUri);

    // Load the style XML in the part into an XDocument instance.
    styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));
}
}

string defaultStyle =
    (string)(
        from style in styleDoc.Root.Elements(w + "style")
        where (string)style.Attribute(w + "type") == "paragraph" &&
            (string)style.Attribute(w + "default") == "1"
        select style
    ).First().Attribute(w + "styleId");

// Find all paragraphs in the document.
var paragraphs =
    from para in xDoc
        .Root
        .Element(w + "body")
        .Descendants(w + "p")
    let styleNode = para
        .Elements(w + "pPr")
        .Elements(w + "pStyle")
        .FirstOrDefault()
    select new
    {
        ParagraphNode = para,
        StyleName = styleNode != null ?
            (string)styleNode.Attribute(w + "val") :
            defaultStyle
    };

// Following is the new query that retrieves the text of
// each paragraph.
var paraWithText =
    from para in paragraphs
    select new
    {
        ParagraphNode = para.ParagraphNode,
        StyleName = para.StyleName,
        Text = para
            .ParagraphNode
            .Elements(w + "r")
            .Elements(w + "t")
            .Aggregate(
                new StringBuilder(),
                (s, i) => s.Append((string)i),
                s => s.ToString()
            )
    };

foreach (var p in paraWithText)
    Console.WriteLine("StyleName:{0} >{1}<", p.StyleName, p.Text);

```

This example produces the following output when applied to the document described in [Creating the Source Office](#)

[Open XML Document \(C#\)](#).

```
StyleName:Heading1 >Parsing WordprocessingML with LINQ to XML<
StyleName:Normal ><
StyleName:Normal >The following example prints to the console.<
StyleName:Normal ><
StyleName:Code >using System;<
StyleName:Code ><
StyleName:Code >class Program {<
StyleName:Code >    public static void (string[] args) {<
StyleName:Code >        Console.WriteLine("Hello World");<
StyleName:Code >    }<
StyleName:Code >}<
StyleName:Normal ><
StyleName:Normal >This example produces the following output:<
StyleName:Normal ><
StyleName:Code >Hello World<
```

## Next Steps

The next example shows how to use an extension method, instead of [Aggregate](#), to concatenate multiple strings into a single string.

- [Refactoring Using an Extension Method \(C#\)](#)

## See also

- [Tutorial: Manipulating Content in a WordprocessingML Document \(C#\)](#)
- [Deferred Execution and Lazy Evaluation in LINQ to XML \(C#\)](#)

# Refactoring Using an Extension Method (C#)

6/4/2019 • 4 minutes to read • [Edit Online](#)

This example builds on the previous example, [Retrieving the Text of the Paragraphs \(C#\)](#), by refactoring the concatenation of strings using a pure function that is implemented as an extension method.

The previous example used the [Aggregate](#) standard query operator to concatenate multiple strings into one string. However, it is more convenient to write an extension method to do this, because the resulting query is smaller and more simple.

## Example

This example processes a WordprocessingML document, retrieving the paragraphs, the style of each paragraph, and the text of each paragraph. This example builds on the previous examples in this tutorial.

The example contains multiple overloads of the `StringConcatenate` method.

You can find instructions for creating the source document for this example in [Creating the Source Office Open XML Document \(C#\)](#).

This example uses classes from the WindowsBase assembly. It uses types in the [System.IO.Packaging](#) namespace.

```
public static class LocalExtensions
{
    public static string StringConcatenate(this IEnumerable<string> source)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item));
        return sb.ToString();
    }

    public static string StringConcatenate(this IEnumerable<string> source, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s).Append(separator);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item)).Append(separator);
        return sb.ToString();
    }
}
```

## Example

There are four overloads of the `StringConcatenate` method. One overload simply takes a collection of strings and returns a single string. Another overload can take a collection of any type, and a delegate that projects from a singleton of the collection to a string. There are two more overloads that allow you to specify a separator string.

The following code uses all four overloads.

```
string[] numbers = { "one", "two", "three" };

Console.WriteLine("{0}", numbers.StringConcatenate());
Console.WriteLine("{0}", numbers.StringConcatenate(":"));

int[] intNumbers = { 1, 2, 3 };
Console.WriteLine("{0}", intNumbers.StringConcatenate(i => i.ToString()));
Console.WriteLine("{0}", intNumbers.StringConcatenate(i => i.ToString(), ":"));
```

This example produces the following output:

```
onetwothree
one:two:three:
123
1:2:3:
```

## Example

Now, the example can be modified to take advantage of the new extension method:

```
public static class LocalExtensions
{
    public static string StringConcatenate(this IEnumerable<string> source)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item));
        return sb.ToString();
    }

    public static string StringConcatenate(this IEnumerable<string> source, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s).Append(separator);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item)).Append(separator);
        return sb.ToString();
    }
}
```

```

    }
}

class Program
{
    static void Main(string[] args)
    {
        const string fileName = "SampleDoc.docx";

        const string documentRelationshipType =
            "http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
        const string stylesRelationshipType =
            "http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
        const string wordmlNamespace =
            "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
        XNamespace w = wordmlNamespace;

        XDocument xDoc = null;
        XDocument styleDoc = null;

        using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.Read))
        {
            PackageRelationship docPackageRelationship =
                wdPackage.GetRelationshipsByType(documentRelationshipType).FirstOrDefault();
            if (docPackageRelationship != null)
            {
                Uri documentUri = PackUriHelper.ResolvePartUri(new Uri("/", UriKind.Relative),
                    docPackageRelationship.TargetUri);
                PackagePart documentPart = wdPackage.GetPart(documentUri);

                // Load the document XML in the part into an XDocument instance.
                xDoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

                // Find the styles part. There will only be one.
                PackageRelationship styleRelation =
                    documentPart.GetRelationshipsByType(stylesRelationshipType).FirstOrDefault();
                if (styleRelation != null)
                {
                    Uri styleUri =
                        PackUriHelper.ResolvePartUri(documentUri, styleRelation.TargetUri);
                    PackagePart stylePart = wdPackage.GetPart(styleUri);

                    // Load the style XML in the part into an XDocument instance.
                    styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));
                }
            }
        }

        string defaultStyle =
            (string)(
                from style in styleDoc.Root.Elements(w + "style")
                where (string)style.Attribute(w + "type") == "paragraph" &&
                    (string)style.Attribute(w + "default") == "1"
                select style
            ).First().Attribute(w + "styleId");

        // Find all paragraphs in the document.
        var paragraphs =
            from para in xDoc
            .Root
            .Element(w + "body")
            .Descendants(w + "p")
            let styleNode = para
                .Elements(w + "pPr")
                .Elements(w + "pStyle")
                .FirstOrDefault()
            select new
            {
                ParagraphNode = para,

```

```

        StyleName = styleNode != null ?
            (string)styleNode.Attribute(w + "val") :
            defaultStyle
    };

    // Retrieve the text of each paragraph.
    var paraWithText =
        from para in paragraphs
        select new
        {
            ParagraphNode = para.ParagraphNode,
            StyleName = para.StyleName,
            Text = para
                .ParagraphNode
                .Elements(w + "r")
                .Elements(w + "t")
                .StringConcatenate(e => (string)e)
        };

    foreach (var p in paraWithText)
        Console.WriteLine("StyleName:{0} >{1}<", p.StyleName, p.Text);
}

```

This example produces the following output when applied to the document described in [Creating the Source Office Open XML Document \(C#\)](#).

```

StyleName:Heading1 >Parsing WordprocessingML with LINQ to XML<
StyleName:Normal ><
StyleName:Normal >The following example prints to the console.<
StyleName:Normal ><
StyleName:Code >using System;<
StyleName:Code ><
StyleName:Code >class Program {<
StyleName:Code >    public static void (string[] args) {<
StyleName:Code >        Console.WriteLine("Hello World");<
StyleName:Code >    }<
StyleName:Code >}<
StyleName:Normal ><
StyleName:Normal >This example produces the following output:<
StyleName:Normal ><
StyleName:Code >Hello World<

```

Note that this refactoring is a variant of refactoring into a pure function. The next topic will introduce the idea of factoring into pure functions in more detail.

## Next Steps

The next example shows how to refactor this code in another way, by using pure functions:

- [Refactoring Using a Pure Function \(Visual Basic\)](#)

## See also

- [Tutorial: Manipulating Content in a WordprocessingML Document \(C#\)](#)
- [Refactoring Into Pure Functions \(C#\)](#)

# Refactoring Using a Pure Function (C#)

6/4/2019 • 3 minutes to read • [Edit Online](#)

The following example refactors the previous example, [Refactoring Using an Extension Method \(C#\)](#), to use a pure function. In this example, the code to find the text of a paragraph is moved to the pure static method `ParagraphText`.

## Example

This example processes a WordprocessingML document, retrieving the paragraph nodes from a WordprocessingML document. It also identifies the style of each paragraph. This example builds on the previous examples in this tutorial. The refactored code is called out in comments in the code below.

For instructions for creating the source document for this example, see [Creating the Source Office Open XML Document \(C#\)](#).

This example uses classes from the WindowsBase assembly. It uses types in the `System.IO.Packaging` namespace.

```
public static class LocalExtensions
{
    public static string StringConcatenate(this IEnumerable<string> source)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item));
        return sb.ToString();
    }

    public static string StringConcatenate(this IEnumerable<string> source, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s).Append(separator);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item)).Append(separator);
        return sb.ToString();
    }
}

class Program
{
    // This is a new method that assembles the paragraph text.
    public static string ParagraphText(XElement e)
    {
        XNamespace w = e.Name.Namespace;
```



```

        return e
            .Elements(w + "r")
            .Elements(w + "t")
            .StringConcatenate(element => (string)element);
    }

static void Main(string[] args)
{
    const string fileName = "SampleDoc.docx";

    const string documentRelationshipType =
        "http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
    const string stylesRelationshipType =
        "http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
    const string wordmlNamespace =
        "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
    XNamespace w = wordmlNamespace;

    XDocument xDoc = null;
    XDocument styleDoc = null;

    using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.Read))
    {
        PackageRelationship docPackageRelationship =
            wdPackage.GetRelationshipsByType(documentRelationshipType).FirstOrDefault();
        if (docPackageRelationship != null)
        {
            Uri documentUri = PackUriHelper.ResolvePartUri(new Uri("/", UriKind.Relative),
                docPackageRelationship.TargetUri);
            PackagePart documentPart = wdPackage.GetPart(documentUri);

            // Load the document XML in the part into an XDocument instance.
            xDoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

            // Find the styles part. There will only be one.
            PackageRelationship styleRelation =
                documentPart.GetRelationshipsByType(stylesRelationshipType).FirstOrDefault();
            if (styleRelation != null)
            {
                Uri styleUri = PackUriHelper.ResolvePartUri(documentUri,
                    styleRelation.TargetUri);
                PackagePart stylePart = wdPackage.GetPart(styleUri);

                // Load the style XML in the part into an XDocument instance.
                styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));
            }
        }
    }

    string defaultStyle =
        (string)(
            from style in styleDoc.Root.Elements(w + "style")
            where (string)style.Attribute(w + "type") == "paragraph" &&
                (string)style.Attribute(w + "default") == "1"
            select style
        ).First().Attribute(w + "styleId");

    // Find all paragraphs in the document.
    var paragraphs =
        from para in xDoc
            .Root
            .Element(w + "body")
            .Descendants(w + "p")
        let styleNode = para
            .Elements(w + "pPr")
            .Elements(w + "pStyle")
            .FirstOrDefault()
        select new
    {

```

```

        ParagraphNode = para,
        StyleName = styleNode != null ?
            (string)styleNode.Attribute(w + "val") :
            defaultStyle
    };

    // Retrieve the text of each paragraph.
    var paraWithText =
        from para in paragraphs
        select new
        {
            ParagraphNode = para.ParagraphNode,
            StyleName = para.StyleName,
            Text = ParagraphText(para.ParagraphNode)
        };

    foreach (var p in paraWithText)
        Console.WriteLine("StyleName:{0} >{1}<", p.StyleName, p.Text);
}

```

This example produces the same output as before the refactoring:

```

StyleName:Heading1 >Parsing WordprocessingML with LINQ to XML<
StyleName:Normal ><
StyleName:Normal >The following example prints to the console.<
StyleName:Normal ><
StyleName:Code >using System;<
StyleName:Code ><
StyleName:Code >class Program {<
StyleName:Code >    public static void (string[] args) {<
StyleName:Code >        Console.WriteLine("Hello World");<
StyleName:Code >    }<
StyleName:Code >}<
StyleName:Normal ><
StyleName:Normal >This example produces the following output:<
StyleName:Normal ><
StyleName:Code >Hello World<

```

## Next Steps

The next example shows how to project XML into a different shape:

- [Projecting XML in a Different Shape \(C#\)](#)

## See also

- [Tutorial: Manipulating Content in a WordprocessingML Document \(C#\)](#)
- [Refactoring Using an Extension Method \(C#\)](#)
- [Refactoring Into Pure Functions \(C#\)](#)

# Projecting XML in a Different Shape (C#)

6/4/2019 • 3 minutes to read • [Edit Online](#)

This topic shows an example of projecting XML that is in a different shape than the source XML.

Many typical XML transformations consist of chained queries, as in this example. It is common to start with some form of XML, project intermediate results as collections of anonymous types or named types, and then finally to project the results back into XML that is in an entirely different shape than the source XML.

## Example

This example processes a WordprocessingML document, retrieving the paragraph nodes from a WordprocessingML document. It also identifies the style and text of each paragraph. Finally, the example projects XML with a different shape. This example builds on the previous examples in this tutorial. The new statement that does the projection is called out in comments in the code below.

For instructions for creating the source document for this example, see [Creating the Source Office Open XML Document \(C#\)](#).

This example uses classes from the WindowsBase assembly. It uses types in the [System.IO.Packaging](#) namespace.

```
public static class LocalExtensions
{
    public static string StringConcatenate(this IEnumerable<string> source)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item));
        return sb.ToString();
    }

    public static string StringConcatenate(this IEnumerable<string> source, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s).Append(separator);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item)).Append(separator);
        return sb.ToString();
    }
}

class Program
```

```

{
    public static string ParagraphText(XElement e)
    {
        XNamespace w = e.Name.Namespace;
        return e
            .Elements(w + "r")
            .Elements(w + "t")
            .StringConcatenate(element => (string)element);
    }

    static void Main(string[] args)
    {
        const string fileName = "SampleDoc.docx";

        const string documentRelationshipType =
            "http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
        const string stylesRelationshipType =
            "http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
        const string wordmlNamespace =
            "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
        XNamespace w = wordmlNamespace;

        XDocument xDoc = null;
        XDocument styleDoc = null;

        using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.Read))
        {
            PackageRelationship docPackageRelationship =
                wdPackage.GetRelationshipsByType(documentRelationshipType).FirstOrDefault();
            if (docPackageRelationship != null)
            {
                Uri documentUri = PackUriHelper.ResolvePartUri(new Uri("/"), UriKind.Relative,
                    docPackageRelationship.TargetUri);
                PackagePart documentPart = wdPackage.GetPart(documentUri);

                // Load the document XML in the part into an XDocument instance.
                xDoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

                // Find the styles part. There will only be one.
                PackageRelationship styleRelation =
                    documentPart.GetRelationshipsByType(stylesRelationshipType).FirstOrDefault();
                if (styleRelation != null)
                {
                    Uri styleUri =
                        PackUriHelper.ResolvePartUri(documentUri, styleRelation.TargetUri);
                    PackagePart stylePart = wdPackage.GetPart(styleUri);

                    // Load the style XML in the part into an XDocument instance.
                    styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));
                }
            }
        }

        string defaultStyle =
            (string)(
                from style in styleDoc.Root.Elements(w + "style")
                where (string)style.Attribute(w + "type") == "paragraph" &&
                    (string)style.Attribute(w + "default") == "1"
                select style
            ).First().Attribute(w + "styleId");

        // Find all paragraphs in the document.
        var paragraphs =
            from para in xDoc
                .Root
                .Element(w + "body")
                .Descendants(w + "p")
            let styleNode = para
                .Elements(w + "pPr")

```

```

        .Elements(w + "pStyle")
        .FirstOrDefault()

select new
{
    ParagraphNode = para,
    StyleName = styleNode != null ?
        (string)styleNode.Attribute(w + "val") :
        defaultStyle
};

// Retrieve the text of each paragraph.
var paraWithText =
    from para in paragraphs
    select new
    {
        ParagraphNode = para.ParagraphNode,
        StyleName = para.StyleName,
        Text = ParagraphText(para.ParagraphNode)
    };

// The following is the new code that projects XML in a new shape.
XElement root = new XElement("Root",
    from p in paraWithText
    select new XElement("Paragraph",
        new XElement("StyleName", p.StyleName),
        new XElement("Text", p.Text)
    )
);

Console.WriteLine(root);
}
}

```

This example produces the following output:

```

<Root>
  <Paragraph>
    <StyleName>Heading1</StyleName>
    <Text>Parsing WordprocessingML with LINQ to XML</Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Normal</StyleName>
    <Text></Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Normal</StyleName>
    <Text>The following example prints to the console.</Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Normal</StyleName>
    <Text></Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Code</StyleName>
    <Text>using System;</Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Code</StyleName>
    <Text></Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Code</StyleName>
    <Text>class Program {</Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Code</StyleName>
    <Text>    public static void (string[] args) {</Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Code</StyleName>
    <Text>        Console.WriteLine("Hello World");</Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Code</StyleName>
    <Text>    }</Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Code</StyleName>
    <Text>}</Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Normal</StyleName>
    <Text></Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Normal</StyleName>
    <Text>This example produces the following output:</Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Normal</StyleName>
    <Text></Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Code</StyleName>
    <Text>Hello World</Text>
  </Paragraph>
</Root>

```

## Next Steps

In the next example, you'll query to find all the text in a Word document:

- [Finding Text in Word Documents \(C#\)](#)

# Finding Text in Word Documents (C#)

6/4/2019 • 5 minutes to read • [Edit Online](#)

This topic extends the previous queries to do something useful: find all occurrences of a string in the document.

## Example

This example processes a WordprocessingML document, to find all the occurrences of a specific piece of text in the document. To do this, we use a query that finds the string "Hello". This example builds on the previous examples in this tutorial. The new query is called out in comments in the code below.

For instructions for creating the source document for this example, see [Creating the Source Office Open XML Document \(C#\)](#).

This example uses classes found in the WindowsBase assembly. It uses types in the [System.IO.Packaging](#) namespace.

```
public static class LocalExtensions
{
    public static string StringConcatenate(this IEnumerable<string> source)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item));
        return sb.ToString();
    }

    public static string StringConcatenate(this IEnumerable<string> source, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s).Append(separator);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item)).Append(separator);
        return sb.ToString();
    }
}

class Program
{
    public static string ParagraphText(XElement e)
    {
        XNamespace w = e.Name.Namespace;
        return e
```



```

        .Elements(w + "r")
        .Elements(w + "t")
        .StringConcatenate(element => (string)element);
    }

static void Main(string[] args)
{
    const string fileName = "SampleDoc.docx";

    const string documentRelationshipType =
        "http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
    const string stylesRelationshipType =
        "http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
    const string wordmlNamespace =
        "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
    XNamespace w = wordmlNamespace;

    XDocument xDoc = null;
    XDocument styleDoc = null;

    using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.Read))
    {
        PackageRelationship docPackageRelationship =
            wdPackage.GetRelationshipsByType(documentRelationshipType).FirstOrDefault();
        if (docPackageRelationship != null)
        {
            Uri documentUri = PackUriHelper.ResolvePartUri(new Uri("/", UriKind.Relative),
                docPackageRelationship.TargetUri);
            PackagePart documentPart = wdPackage.GetPart(documentUri);

            // Load the document XML in the part into an XDocument instance.
            xDoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

            // Find the styles part. There will only be one.
            PackageRelationship styleRelation =
                documentPart.GetRelationshipsByType(stylesRelationshipType).FirstOrDefault();
            if (styleRelation != null)
            {
                Uri styleUri =
                    PackUriHelper.ResolvePartUri(documentUri, styleRelation.TargetUri);
                PackagePart stylePart = wdPackage.GetPart(styleUri);

                // Load the style XML in the part into an XDocument instance.
                styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));
            }
        }
    }

    string defaultStyle =
        (string)(
            from style in styleDoc.Root.Elements(w + "style")
            where (string)style.Attribute(w + "type") == "paragraph" &&
                (string)style.Attribute(w + "default") == "1"
            select style
        ).First().Attribute(w + "styleId");

    // Find all paragraphs in the document.
    var paragraphs =
        from para in xDoc
            .Root
            .Element(w + "body")
            .Descendants(w + "p")
        let styleNode = para
            .Elements(w + "pPr")
            .Elements(w + "pStyle")
            .FirstOrDefault()
        select new
        {
            ParagraphNode = para,

```

```

        ParagraphNode = para,
        StyleName = styleNode != null ?
            (string)styleNode.Attribute(w + "val") :
            defaultStyle
    };

    // Retrieve the text of each paragraph.
    var paraWithText =
        from para in paragraphs
        select new
        {
            ParagraphNode = para.ParagraphNode,
            StyleName = para.StyleName,
            Text = ParagraphText(para.ParagraphNode)
        };

    // Following is the new query that retrieves all paragraphs
    // that have specific text in them.
    var helloParagraphs =
        from para in paraWithText
        where para.Text.Contains("Hello")
        select new
        {
            ParagraphNode = para.ParagraphNode,
            StyleName = para.StyleName,
            Text = para.Text
        };

    foreach (var p in helloParagraphs)
        Console.WriteLine("StyleName:{0} >{1}<", p.StyleName, p.Text);
}
}

```

This example produces the following output:

```

StyleName:Code >      Console.WriteLine("Hello World");<
StyleName:Code >Hello World<

```

You can, of course, modify the search so that it searches for lines with a specific style. The following query finds all blank lines that have the Code style:

```

public static class LocalExtensions
{
    public static string StringConcatenate(this IEnumerable<string> source)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item));
        return sb.ToString();
    }

    public static string StringConcatenate(this IEnumerable<string> source, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s).Append(separator);
        return sb.ToString();
    }
}

```

```

        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item)).Append(separator);
        return sb.ToString();
    }
}

class Program
{
    public static string ParagraphText(XElement e)
    {
        XNamespace w = e.Name.Namespace;
        return e
            .Elements(w + "p")
            .Elements(w + "t")
            .StringConcatenate(element => (string)element);
    }

    static void Main(string[] args)
    {
        const string fileName = "SampleDoc.docx";

        const string documentRelationshipType =
"http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
        const string stylesRelationshipType =
"http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
        const string wordmlNamespace = "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
        XNamespace w = wordmlNamespace;

        XDocument xDoc = null;
        XDocument styleDoc = null;

        using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.Read))
        {
            PackageRelationship docPackageRelationship =
wdPackage.GetRelationshipsByType(documentRelationshipType).FirstOrDefault();
            if (docPackageRelationship != null)
            {
                Uri documentUri = PackUriHelper.ResolvePartUri(new Uri("/"), UriKind.Relative),
docPackageRelationship.TargetUri);
                PackagePart documentPart = wdPackage.GetPart(documentUri);

                // Load the document XML in the part into an XDocument instance.
                xDoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

                // Find the styles part. There will only be one.
                PackageRelationship styleRelation =
documentPart.GetRelationshipsByType(stylesRelationshipType).FirstOrDefault();
                if (styleRelation != null)
                {
                    Uri styleUri = PackUriHelper.ResolvePartUri(documentUri, styleRelation.TargetUri);
                    PackagePart stylePart = wdPackage.GetPart(styleUri);

                    // Load the style XML in the part into an XDocument instance.
                    styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));
                }
            }
        }

        string defaultStyle =
            (string)(
                from style in styleDoc.Root.Elements(w + "style")
                where (string)style.Attribute(w + "type") == "paragraph" &&
                (string)style.Attribute(w + "id") == "1"
            );
    }
}

```

```

        (string)style.Attribute(w + "default") == "1"
        select style
    ).First().Attribute(w + "styleId");

// Find all paragraphs in the document.
var paragraphs =
    from para in xDoc
        .Root
        .Element(w + "body")
        .Descendants(w + "p")
    let styleNode = para
        .Elements(w + "pPr")
        .Elements(w + "pStyle")
        .FirstOrDefault()
    select new
    {
        ParagraphNode = para,
        StyleName = styleNode != null ?
            (string)styleNode.Attribute(w + "val") :
            defaultStyle
    };

// Retrieve the text of each paragraph.
var paraWithText =
    from para in paragraphs
    select new
    {
        ParagraphNode = para.ParagraphNode,
        StyleName = para.StyleName,
        Text = ParagraphText(para.ParagraphNode)
    };

// Retrieve all paragraphs that have no text and are styled Code.
var blankCodeParagraphs =
    from para in paraWithText
    where para.Text == "" && para.StyleName == "Code"
    select new
    {
        ParagraphNode = para.ParagraphNode,
        StyleName = para.StyleName,
        Text = para.Text
    };

foreach (var p in blankCodeParagraphs)
    Console.WriteLine("StyleName:{0} >{1}<", p.StyleName, p.Text);
}

```

This example produces the following output:

```
StyleName:Code ><
```

Of course, this example could be enhanced in a number of ways. For example, we could use regular expressions to search for text, we could iterate through all the Word files in a particular directory, and so on.

Note that this example performs approximately as well as if it were written as a single query. Because each query is implemented in a lazy, deferred fashion, each query does not yield its results until the query is iterated. For more information about execution and lazy evaluation, see [Deferred Execution and Lazy Evaluation in LINQ to XML \(C#\)](#).

## Next Steps

The next section provides more information about WordprocessingML documents:

- [Details of Office Open XML WordprocessingML Documents \(C#\)](#)

## See also

- [Tutorial: Manipulating Content in a WordprocessingML Document \(C#\)](#)
- [Refactoring Using a Pure Function \(C#\)](#)
- [Deferred Execution and Lazy Evaluation in LINQ to XML \(C#\)](#)

# WordprocessingML Document with Styles

6/4/2019 • 2 minutes to read • [Edit Online](#)

More complicated WordprocessingML documents have paragraphs that are formatted with styles.

A few notes about the makeup of WordprocessingML documents are helpful. WordprocessingML documents are stored in packages. Packages have multiple parts (parts have an explicit meaning when used in the context of packages; essentially, parts are files that are zipped together to comprise a package). If a document contains paragraphs that are formatted with styles, there will be a document part that contains paragraphs that have styles applied to them. There will also be a style part that contains the styles that are referred to by the document.

When accessing packages, it is important that you do so through the relationships between parts, rather than using an arbitrary path. This issue is beyond the scope of the Manipulating Content in a WordprocessingML Document tutorial, but the example programs that are included in this tutorial demonstrate the correct approach.

## A Document that Uses Styles

The WordML example presented in the [Shape of WordprocessingML Documents \(C#\)](#) topic is a very simple one. The following document is more complicated: It has paragraphs that are formatted with styles. The easiest way to see the XML that makes up an Office Open XML document is to run the [Example that Outputs Office Open XML Document Parts \(C#\)](#).

In the following document, the first paragraph has the style `Heading1`. There are a number of paragraphs that have the default style. There are also a number of paragraphs that have the style `Code`. Because of this relative complexity, this is a more interesting document to parse with LINQ to XML.

In those paragraphs with non-default styles, the paragraph elements have a child element named `w:pPr`, which in turn has a child element `w:pStyle`. That element has an attribute, `w:val`, which contains the style name. If the paragraph has the default style, it means that the paragraph element does not have a `w:pPr` child element.

```
<?xml version="1.0" encoding="utf-8"?>
<w:document
  xmlns:ve="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:o="urn:schemas-microsoft-com:office:office"
  xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships"
  xmlns:m="http://schemas.openxmlformats.org/officeDocument/2006/math"
  xmlns:v="urn:schemas-microsoft-com:vml"
  xmlns:wp="http://schemas.openxmlformats.org/drawingml/2006/wordprocessingDrawing"
  xmlns:w10="urn:schemas-microsoft-com:office:word"
  xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main"
  xmlns:wne="http://schemas.microsoft.com/office/word/2006/wordml">
  <w:body>
    <w:p w:rsidR="00A75AE0" w:rsidRDefault="00A75AE0" w:rsidP="006027C7">
      <w:pPr>
        <w:pStyle w:val="Heading1" />
      </w:pPr>
      <w:r>
        <w:t>Parsing WordprocessingML with LINQ to XML</w:t>
      </w:r>
    </w:p>
    <w:p w:rsidR="00A75AE0" w:rsidRDefault="00A75AE0" />
    <w:p w:rsidR="00A75AE0" w:rsidRDefault="00A75AE0">
      <w:r>
        <w:t>The following example prints to the console.</w:t>
      </w:r>
    </w:p>
    <w:p w:rsidR="00A75AE0" w:rsidRDefault="00A75AE0" />
```

```

<w:p w:rsidR="00A75AE0" w:rsidRDefault="00A75AE0" w:rsidP="006027C7">
  <w:pPr>
    <w:pStyle w:val="Code" />
  </w:pPr>
  <w:r>
    <w:t>using System;</w:t>
  </w:r>
</w:p>
<w:p w:rsidR="00A75AE0" w:rsidRDefault="00A75AE0" w:rsidP="006027C7">
  <w:pPr>
    <w:pStyle w:val="Code" />
  </w:pPr>
</w:p>
<w:p w:rsidR="00A75AE0" w:rsidRPr="00876F34" w:rsidRDefault="00A75AE0" w:rsidP="006027C7">
  <w:pPr>
    <w:pStyle w:val="Code" />
  </w:pPr>
  <w:r w:rsidRPr="00876F34">
    <w:t>class Program {</w:t>
  </w:r>
</w:p>
<w:p w:rsidR="00A75AE0" w:rsidRPr="00876F34" w:rsidRDefault="00A75AE0" w:rsidP="006027C7">
  <w:pPr>
    <w:pStyle w:val="Code" />
  </w:pPr>
  <w:r w:rsidRPr="00876F34">
    <w:t xml:space="preserve">    public static void </w:t>
  </w:r>
  <w:smartTag w:uri="urn:schemas-microsoft-com:office:smarts" w:element="place">
    <w:r w:rsidRPr="00876F34">
      <w:t>Main</w:t>
    </w:r>
  </w:smartTag>
  <w:r w:rsidRPr="00876F34">
    <w:t>(string[] args) {</w:t>
  </w:r>
</w:p>
<w:p w:rsidR="00A75AE0" w:rsidRPr="00876F34" w:rsidRDefault="00A75AE0" w:rsidP="006027C7">
  <w:pPr>
    <w:pStyle w:val="Code" />
  </w:pPr>
  <w:r w:rsidRPr="00876F34">
    <w:t xml:space="preserve">        Console.WriteLine("Hello World");</w:t>
  </w:r>
</w:p>
<w:p w:rsidR="00A75AE0" w:rsidRPr="00876F34" w:rsidRDefault="00A75AE0" w:rsidP="006027C7">
  <w:pPr>
    <w:pStyle w:val="Code" />
  </w:pPr>
  <w:r w:rsidRPr="00876F34">
    <w:t xml:space="preserve">    }</w:t>
  </w:r>
</w:p>
<w:p w:rsidR="00A75AE0" w:rsidRPr="00876F34" w:rsidRDefault="00A75AE0" w:rsidP="006027C7">
  <w:pPr>
    <w:pStyle w:val="Code" />
  </w:pPr>
  <w:r w:rsidRPr="00876F34">
    <w:t>}</w:t>
  </w:r>
</w:p>
<w:p w:rsidR="00A75AE0" w:rsidRDefault="00A75AE0" />
<w:p w:rsidR="00A75AE0" w:rsidRDefault="00A75AE0">
  <w:r>
    <w:t>This example produces the following output:</w:t>
  </w:r>
</w:p>
<w:p w:rsidR="00A75AE0" w:rsidRDefault="00A75AE0" />
<w:p w:rsidR="00A75AE0" w:rsidRDefault="00A75AE0" />

```

```
<w:p w:rsidK="00A75AE0" w:rsidKDefault="00A75AE0" w:rsidP="006027C7">
  <w:pPr>
    <w:pStyle w:val="Code" />
  </w:pPr>
  <w:r>
    <w:t>Hello World</w:t>
  </w:r>
</w:p>
<w:sectPr w:rsidR="00A75AE0" w:rsidSect="00A75AE0">
  <w:pgSz w:w="12240" w:h="15840" />
  <w:pgMar w:top="1440" w:right="1800" w:bottom="1440" w:left="1800" w:header="720" w:footer="720"
w:gutter="0" />
  <w:cols w:space="720" />
  <w:docGrid w:linePitch="360" />
</w:sectPr>
</w:body>
</w:document>
```



# Style Part of a WordprocessingML Document

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic shows an example of the style part of the Office Open XML WordprocessingML document.

## Example

The following example is the XML that makes up the style part of an Office Open XML WordprocessingML document.

The default paragraph style has an element with the following opening tag:

```
<w:style w:type="paragraph" w:default="1" w:styleId="Normal">
```

You need to know this information when you write the query to find the default style identifier, so that the query can identify the style of paragraphs that have the default style.

Note that these documents are very simple when compared to typical documents that Microsoft Word generates.

In many cases, Word saves a great deal of additional information, additional formatting and metadata.

Furthermore, Word does not format the lines to be easily readable as in this example; instead, the XML is saved without indentation. However, all WordprocessingML documents share the same basic XML shape. Because of this, the queries presented in this tutorial will work with more complicated documents.

```
<?xml version="1.0" encoding="utf-8"?>
<w:styles
  xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships"
  xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main">
  <w:docDefaults>
    <w:rPrDefault>
      <w:rPr>
        <w:rFonts
          w:ascii="Times New Roman"
          w:eastAsia="Times New Roman"
          w:hAnsi="Times New Roman"
          w:cs="Times New Roman" />
        <w:sz w:val="22" />
        <w:szCs w:val="22" />
        <w:lang w:val="en-US" w:eastAsia="en-US" w:bidirectional="ar-SA" />
      </w:rPr>
    </w:rPrDefault>
    <w:pPrDefault />
  </w:docDefaults>
  <w:style w:type="paragraph" w:default="1" w:styleId="Normal">
    <w:name w:val="Normal" />
    <w:qFormat />
    <w:rPr>
      <w:sz w:val="24" />
      <w:szCs w:val="24" />
    </w:rPr>
  </w:style>
  <w:style w:type="paragraph" w:styleId="Heading1">
    <w:name w:val="heading 1" />
    <w:basedOn w:val="Normal" />
    <w:next w:val="Normal" />
    <w:link w:val="Heading1Char" />
    <w:uiPriority w:val="99" />
    <w:qFormat />
  </w:style>
```

```

<w:rsid w:val="006027C7" />
<w:pPr>
  <w:keepNext />
  <w:spacing w:before="240" w:after="60" />
  <w:outlineLvl w:val="0" />
</w:pPr>
<w:rPr>
  <w:rFonts w:ascii="Arial" w:hAnsi="Arial" w:cs="Arial" />
  <w:b />
  <w:bCs />
  <w:kern w:val="32" />
  <w:sz w:val="32" />
  <w:szCs w:val="32" />
</w:rPr>
</w:style>
<w:style w:type="character" w:default="1" w:styleId="DefaultParagraphFont">
  <w:name w:val="Default Paragraph Font" />
  <w:uiPriority w:val="99" />
  <w:semiHidden />
</w:style>
<w:style w:type="table" w:default="1" w:styleId="TableNormal">
  <w:name w:val="Normal Table" />
  <w:uiPriority w:val="99" />
  <w:semiHidden />
  <w:unhideWhenUsed />
  <w:qFormat />
  <w:tblPr>
    <w:tblInd w:w="0" w:type="dxa" />
    <w:tblCellMar>
      <w:top w:w="0" w:type="dxa" />
      <w:left w:w="108" w:type="dxa" />
      <w:bottom w:w="0" w:type="dxa" />
      <w:right w:w="108" w:type="dxa" />
    </w:tblCellMar>
  </w:tblPr>
</w:style>
<w:style w:type="numbering" w:default="1" w:styleId="NoList">
  <w:name w:val="No List" />
  <w:uiPriority w:val="99" />
  <w:semiHidden />
  <w:unhideWhenUsed />
</w:style>
<w:style w:type="character" w:customStyle="1" w:styleId="Heading1Char">
  <w:name w:val="Heading 1 Char" />
  <w:basedOn w:val="DefaultParagraphFont" />
  <w:link w:val="Heading1" />
  <w:uiPriority w:val="9" />
  <w:rsid w:val="009765E3" />
  <w:rPr>
    <w:rFonts
      w:asciiTheme="majorHAnsi"
      w:eastAsiaTheme="majorEastAsia"
      w:hAnsiTheme="majorHAnsi"
      w:cstheme="majorBidi" />
    <w:b />
    <w:bCs />
    <w:kern w:val="32" />
    <w:sz w:val="32" />
    <w:szCs w:val="32" />
  </w:rPr>
</w:style>
<w:style w:type="paragraph" w:customStyle="1" w:styleId="Code">
  <w:name w:val="Code" />
  <w:aliases w:val="c" />
  <w:uiPriority w:val="99" />
  <w:rsid w:val="006027C7" />
  <w:pPr>
    <w:spacing w:after="60" w:line="300" w:lineRule="exact" />
  </w:pPr>

```

```
<w:rPr>  
  <w:rFonts w:ascii="Courier New" w:hAnsi="Courier New" />  
  <w:noProof />  
  <w:color w:val="000080" />  
  <w:sz w:val="20" />  
  <w:szCs w:val="20" />  
</w:rPr>  
</w:style>  
</w:styles>
```

# Example that Outputs Office Open XML Document Parts (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic shows how to open an Office Open XML document and access parts within it.

## Example

The following example opens an Office Open XML document, and prints the document part and the style part to the console.

This example uses classes from the WindowsBase assembly. It uses types in the [System.IO.Packaging](#) namespace.

```

const string fileName = "SampleDoc.docx";

const string documentRelationshipType =
    "http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
const string stylesRelationshipType =
    "http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
const string wordmlNamespace =
    "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
XNamespace w = wordmlNamespace;

using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.Read))
{
    PackageRelationship docPackageRelationship =
        wdPackage.GetRelationshipsByType(documentRelationshipType).FirstOrDefault();
    if (docPackageRelationship != null)
    {
        Uri documentUri = PackUriHelper.ResolvePartUri(new Uri("/", UriKind.Relative),
            docPackageRelationship.TargetUri);
        PackagePart documentPart = wdPackage.GetPart(documentUri);

        // Load the document XML in the part into an XDocument instance.
        XDocument xdoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

        Console.WriteLine("TargetUri:{0}", docPackageRelationship.TargetUri);
        Console.WriteLine("=====");
        Console.WriteLine(xdoc.Root);
        Console.WriteLine();

        // Find the styles part. There will only be one.
        PackageRelationship styleRelation =
            documentPart.GetRelationshipsByType(stylesRelationshipType).FirstOrDefault();
        if (styleRelation != null)
        {
            Uri styleUri = PackUriHelper.ResolvePartUri(documentUri, styleRelation.TargetUri);
            PackagePart stylePart = wdPackage.GetPart(styleUri);

            // Load the style XML in the part into an XDocument instance.
            XDocument styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));

            Console.WriteLine("TargetUri:{0}", styleRelation.TargetUri);
            Console.WriteLine("=====");
            Console.WriteLine(styleDoc.Root);
            Console.WriteLine();
        }
    }
}

```

# In-Memory XML Tree Modification vs. Functional Construction (LINQ to XML) (C#)

6/4/2019 • 3 minutes to read • [Edit Online](#)

Modifying an XML tree in place is a traditional approach to changing the shape of an XML document. A typical application loads a document into a data store such as DOM or LINQ to XML; uses a programming interface to insert nodes, delete nodes, or change the content of nodes; and then saves the XML to a file or transmits it over a network.

LINQ to XML enables another approach that is useful in many scenarios: *functional construction*. Functional construction treats modifying data as a problem of transformation, rather than as detailed manipulation of a data store. If you can take a representation of data and transform it efficiently from one form to another, the result is the same as if you took one data store and manipulated it in some way to take another shape. A key to the functional construction approach is to pass the results of queries to [XDocument](#) and [XElement](#) constructors.

In many cases you can write the transformational code in a fraction of the time that it would take to manipulate the data store, and that code is more robust and easier to maintain. In these cases, even though the transformational approach can take more processing power, it is a more effective way to modify data. If a developer is familiar with the functional approach, the resulting code in many cases is easier to understand. It is easy to find the code that modifies each part of the tree.

The approach where you modify an XML tree in-place is more familiar to many DOM programmers, whereas code written using the functional approach might look unfamiliar to a developer who doesn't yet understand that approach. If you have to only make a small modification to a large XML tree, the approach where you modify a tree in place in many cases will take less CPU time.

This topic provides an example that is implemented with both approaches.

## Transforming Attributes into Elements

For this example, suppose you want to modify the following simple XML document so that the attributes become elements. This topic first presents the traditional in-place modification approach. It then shows the functional construction approach.

```
<?xml version="1.0" encoding="utf-8" ?>
<Root Data1="123" Data2="456">
  <Child1>Content</Child1>
</Root>
```

### Modifying the XML Tree

You can write some procedural code to create elements from the attributes, and then delete the attributes, as follows:

```
XElement root = XElement.Load("Data.xml");
foreach (XAttribute att in root.Attributes()) {
    root.Add(new XElement(att.Name, (string)att));
}
root.Attributes().Remove();
Console.WriteLine(root);
```

This code produces the following output:

```
<Root>
  <Child1>Content</Child1>
  <Data1>123</Data1>
  <Data2>456</Data2>
</Root>
```

## Functional Construction Approach

By contrast, a functional approach consists of code to form a new tree, picking and choosing elements and attributes from the source tree, and transforming them as appropriate as they are added to the new tree. The functional approach looks like the following:

```
XElement root = XElement.Load("Data.xml");
XElement newTree = new XElement("Root",
    root.Element("Child1"),
    from att in root.Attributes()
    select new XElement(att.Name, (string)att)
);
Console.WriteLine(newTree);
```

This example outputs the same XML as the first example. However, notice that you can actually see the resulting structure of the new XML in the functional approach. You can see the creation of the `Root` element, the code that pulls the `Child1` element from the source tree, and the code that transforms the attributes from the source tree to elements in the new tree.

The functional example in this case is not any shorter than the first example, and it is not really any simpler. However, if you have many changes to make to an XML tree, the non functional approach will become quite complex and somewhat obtuse. In contrast, when using the functional approach, you still just form the desired XML, embedding queries and expressions as appropriate, to pull in the desired content. The functional approach yields code that is easier to maintain.

Notice that in this case the functional approach probably would not perform quite as well as the tree manipulation approach. The main issue is that the functional approach creates more short lived objects. However, the tradeoff is an effective one if using the functional approach allows for greater programmer productivity.

This is a very simple example, but it serves to show the difference in philosophy between the two approaches. The functional approach yields greater productivity gains for transforming larger XML documents.

# Adding Elements, Attributes, and Nodes to an XML Tree (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

You can add content (elements, attributes, comments, processing instructions, text, and CDATA) to an existing XML tree.

## Methods for Adding Content

The following methods add child content to an [XElement](#) or an [XDocument](#):

METHOD	DESCRIPTION
<a href="#">Add</a>	Adds content at the end of the child content of the <a href="#">XContainer</a> .
<a href="#">AddFirst</a>	Adds content at the beginning of the child content of the <a href="#">XContainer</a> .

The following methods add content as sibling nodes of an [XNode](#). The most common node to which you add sibling content is [XElement](#), although you can add valid sibling content to other types of nodes such as [XText](#) or [XComment](#).

METHOD	DESCRIPTION
<a href="#">AddAfterSelf</a>	Adds content after the <a href="#">XNode</a> .
<a href="#">AddBeforeSelf</a>	Adds content before the <a href="#">XNode</a> .

## Example

### Description

The following example creates two XML trees, and then modifies one of the trees.

### Code



```

XElement srcTree = new XElement("Root",
    new XElement("Element1", 1),
    new XElement("Element2", 2),
    new XElement("Element3", 3),
    new XElement("Element4", 4),
    new XElement("Element5", 5)
);
XElement xmlTree = new XElement("Root",
    new XElement("Child1", 1),
    new XElement("Child2", 2),
    new XElement("Child3", 3),
    new XElement("Child4", 4),
    new XElement("Child5", 5)
);
xmlTree.Add(new XElement("NewChild", "new content"));
xmlTree.Add(
    from el in srcTree.Elements()
    where (int)el > 3
    select el
);
// Even though Child9 does not exist in srcTree, the following statement will not
// throw an exception, and nothing will be added to xmlTree.
xmlTree.Add(srcTree.Element("Child9"));
Console.WriteLine(xmlTree);

```

## Comments

This code produces the following output:

```

<Root>
  <Child1>1</Child1>
  <Child2>2</Child2>
  <Child3>3</Child3>
  <Child4>4</Child4>
  <Child5>5</Child5>
  <NewChild>new content</NewChild>
  <Element4>4</Element4>
  <Element5>5</Element5>
</Root>

```

# Modifying Elements, Attributes, and Nodes in an XML Tree

6/4/2019 • 2 minutes to read • [Edit Online](#)

The following table summarizes the methods and properties that you can use to modify an element, its child elements, or its attributes.

The following methods modify an [XElement](#).

METHOD	DESCRIPTION
<a href="#">XElement.Parse</a>	Replaces an element with parsed XML.
<a href="#">XElement.RemoveAll</a>	Removes all content (child nodes and attributes) of an element.
<a href="#">XElement.RemoveAttributes</a>	Removes the attributes of an element.
<a href="#">XElement.ReplaceAll</a>	Replaces all content (child nodes and attributes) of an element.
<a href="#">XElement.ReplaceAttributes</a>	Replaces the attributes of an element.
<a href="#">XElement.SetAttributeValue</a>	Sets the value of an attribute. Creates the attribute if it doesn't exist. If the value is set to <code>null</code> , removes the attribute.
<a href="#">XElement.SetElementValue</a>	Sets the value of a child element. Creates the element if it doesn't exist. If the value is set to <code>null</code> , removes the element.
<a href="#">XElement.Value</a>	Replaces the content (child nodes) of an element with the specified text.
<a href="#">XElement.SetValue</a>	Sets the value of an element.

The following methods modify an [XAttribute](#).

METHOD	DESCRIPTION
<a href="#">XAttribute.Value</a>	Sets the value of an attribute.
<a href="#">XAttribute.SetValue</a>	Sets the value of an attribute.

The following methods modify an [XNode](#) (including an [XElement](#) or [XDocument](#)).

METHOD	DESCRIPTION
<a href="#">XNode.ReplaceWith</a>	Replaces a node with new content.

The following methods modify an [XContainer](#) (an [XElement](#) or [XDocument](#)).

METHOD	DESCRIPTION
<a href="#">XContainer.ReplaceNodes</a>	Replaces the children nodes with new content.

# Removing Elements, Attributes, and Nodes from an XML Tree (C#)

6/5/2019 • 2 minutes to read • [Edit Online](#)

You can modify an XML tree, removing elements, attributes, and other types of nodes.

Removing a single element or a single attribute from an XML document is straightforward. However, when removing collections of elements or attributes, you should first materialize a collection into a list, and then delete the elements or attributes from the list. The best approach is to use the [Remove](#) extension method, which will do this for you.

The main reason for doing this is that most of the collections you retrieve from an XML tree are yielded using deferred execution. If you do not first materialize them into a list, or if you do not use the extension methods, it is possible to encounter a certain class of bugs. For more information, see [Mixed Declarative Code/Imperative Code Bugs \(LINQ to XML\) \(C#\)](#).

The following methods remove nodes and attributes from an XML tree.

METHOD	DESCRIPTION
<a href="#">XAttribute.Remove</a>	Removes an <a href="#">XAttribute</a> from its parent.
<a href="#">XContainer.RemoveNodes</a>	Removes the child nodes from an <a href="#">XContainer</a> .
<a href="#">XElement.RemoveAll</a>	Removes content and attributes from an <a href="#">XElement</a> .
<a href="#">XElement.RemoveAttributes</a>	Removes the attributes of an <a href="#">XElement</a> .
<a href="#">XElement.SetAttributeValue</a>	If you pass <code>null</code> for value, then removes the attribute.
<a href="#">XElement.SetElementValue</a>	If you pass <code>null</code> for value, then removes the child element.
<a href="#">XNode.Remove</a>	Removes an <a href="#">XNode</a> from its parent.
<a href="#">Extensions.Remove</a>	Removes every attribute or element in the source collection from its parent element.

## Example

### Description

This example demonstrates three approaches to removing elements. First, it removes a single element. Second, it retrieves a collection of elements, materializes them using the [Enumerable.ToList](#) operator, and removes the collection. Finally, it retrieves a collection of elements and removes them using the [Remove](#) extension method.

For more information on the [ToList](#) operator, see [Converting Data Types \(C#\)](#).

### Code

```
XElement root = XElement.Parse(@"<Root>
  <Child1>
    <GrandChild1/>
    <GrandChild2/>
    <GrandChild3/>
  </Child1>
  <Child2>
    <GrandChild4/>
    <GrandChild5/>
    <GrandChild6/>
  </Child2>
  <Child3>
    <GrandChild7/>
    <GrandChild8/>
    <GrandChild9/>
  </Child3>
</Root>");
root.Element("Child1").Element("GrandChild1").Remove();
root.Element("Child2").Elements().ToList().Remove();
root.Element("Child3").Elements().Remove();
Console.WriteLine(root);
```

## Comments

This code produces the following output:

```
<Root>
  <Child1>
    <GrandChild2 />
    <GrandChild3 />
  </Child1>
  <Child2 />
  <Child3 />
</Root>
```

Notice that the first grandchild element has been removed from `child1`. All grandchildren elements have been removed from `child2` and from `child3`.

# Maintaining Name/Value Pairs (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

Many applications have to maintain information that is best kept as name/value pairs. This information might be configuration information or global settings. LINQ to XML contains some methods that make it easy to keep a set of name/value pairs. You can either keep the information as attributes or as a set of child elements.

One difference between keeping the information as attributes or as child elements is that attributes have the constraint that there can be only one attribute with a particular name for an element. This limitation does not apply to child elements.

## SetAttributeValue and SetElementValue

The two methods that facilitate keeping name/value pairs are [SetAttributeValue](#) and [SetElementValue](#). These two methods have similar semantics.

[SetAttributeValue](#) can add, modify, or remove attributes of an element.

- If you call [SetAttributeValue](#) with a name of an attribute that does not exist, the method creates a new attribute and adds it to the specified element.
- If you call [SetAttributeValue](#) with a name of an existing attribute and with some specified content, the contents of the attribute are replaced with the specified content.
- If you call [SetAttributeValue](#) with a name of an existing attribute, and specify null for the content, the attribute is removed from its parent.

[SetElementValue](#) can add, modify, or remove child elements of an element.

- If you call [SetElementValue](#) with a name of a child element that does not exist, the method creates a new element and adds it to the specified element.
- If you call [SetElementValue](#) with a name of an existing element and with some specified content, the contents of the element are replaced with the specified content.
- If you call [SetElementValue](#) with a name of an existing element, and specify null for the content, the element is removed from its parent.

## Example

The following example creates an element with no attributes. It then uses the [SetAttributeValue](#) method to create and maintain a list of name/value pairs.

```
// Create an element with no content.
XElement root = new XElement("Root");

// Add a number of name/value pairs as attributes.
root.SetAttributeValue("Top", 22);
root.SetAttributeValue("Left", 20);
root.SetAttributeValue("Bottom", 122);
root.SetAttributeValue("Right", 300);
root.SetAttributeValue("DefaultColor", "Color.Red");
Console.WriteLine(root);

// Replace the value of Top.
root.SetAttributeValue("Top", 10);
Console.WriteLine(root);

// Remove DefaultColor.
root.SetAttributeValue("DefaultColor", null);
Console.WriteLine(root);
```

This example produces the following output:

```
<Root Top="22" Left="20" Bottom="122" Right="300" DefaultColor="Color.Red" />
<Root Top="10" Left="20" Bottom="122" Right="300" DefaultColor="Color.Red" />
<Root Top="10" Left="20" Bottom="122" Right="300" />
```

## Example

The following example creates an element with no child elements. It then uses the [SetElementValue](#) method to create and maintain a list of name/value pairs.

```
// Create an element with no content.
XElement root = new XElement("Root");

// Add a number of name/value pairs as elements.
root.SetElementValue("Top", 22);
root.SetElementValue("Left", 20);
root.SetElementValue("Bottom", 122);
root.SetElementValue("Right", 300);
root.SetElementValue("DefaultColor", "Color.Red");
Console.WriteLine(root);
Console.WriteLine("----");

// Replace the value of Top.
root.SetElementValue("Top", 10);
Console.WriteLine(root);
Console.WriteLine("----");

// Remove DefaultColor.
root.SetElementValue("DefaultColor", null);
Console.WriteLine(root);
```

This example produces the following output:

```
<Root>
  <Top>22</Top>
  <Left>20</Left>
  <Bottom>122</Bottom>
  <Right>300</Right>
  <DefaultColor>Color.Red</DefaultColor>
</Root>
----
<Root>
  <Top>10</Top>
  <Left>20</Left>
  <Bottom>122</Bottom>
  <Right>300</Right>
  <DefaultColor>Color.Red</DefaultColor>
</Root>
----
<Root>
  <Top>10</Top>
  <Left>20</Left>
  <Bottom>122</Bottom>
  <Right>300</Right>
</Root>
```

## See also

- [SetAttributeValue](#)
- [SetElementValue](#)
- [Modifying XML Trees \(LINQ to XML\) \(C#\)](#)



# How to: Change the Namespace for an Entire XML Tree (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

You sometimes have to programmatically change the namespace for an element or an attribute. LINQ to XML makes this easy. The `XElement.Name` property can be set. The `XAttribute.Name` property cannot be set, but you can easily copy the attributes into a `System.Collections.Generic.List<T>`, remove the existing attributes, and then add new attributes that are in the new desired namespace.

For more information, see [Working with XML Namespaces \(C#\)](#).

## Example

The following code creates two XML trees in no namespace. It then changes the namespace of each of the trees, and combines them into a single tree.

```

XElement tree1 = new XElement("Data",
    new XElement("Child", "content",
        new XAttribute("MyAttr", "content")
    )
);
XElement tree2 = new XElement("Data",
    new XElement("Child", "content",
        new XAttribute("MyAttr", "content")
    )
);
XNamespace aw = "http://www.adventure-works.com";
XNamespace ad = "http://www.adatum.com";
// change the namespace of every element and attribute in the first tree
foreach (XElement el in tree1.DescendantsAndSelf())
{
    el.Name = aw.GetName(el.Name.LocalName);
    List<XAttribute> atList = el.Attributes().ToList();
    el.Attributes().Remove();
    foreach (XAttribute at in atList)
        el.Add(new XAttribute(aw.GetName(at.Name.LocalName), at.Value));
}
// change the namespace of every element and attribute in the second tree
foreach (XElement el in tree2.DescendantsAndSelf())
{
    el.Name = ad.GetName(el.Name.LocalName);
    List<XAttribute> atList = el.Attributes().ToList();
    el.Attributes().Remove();
    foreach (XAttribute at in atList)
        el.Add(new XAttribute(ad.GetName(at.Name.LocalName), at.Value));
}
// add attribute namespaces so that the tree will be serialized with
// the aw and ad namespace prefixes
tree1.Add(
    new XAttribute(XNamespace.Xmlns + "aw", "http://www.adventure-works.com")
);
tree2.Add(
    new XAttribute(XNamespace.Xmlns + "ad", "http://www.adatum.com")
);
// create a new composite tree
XElement root = new XElement("Root",
    tree1,
    tree2
);
Console.WriteLine(root);

```

This example produces the following output:

```

<Root>
  <aw:Data xmlns:aw="http://www.adventure-works.com">
    <aw:Child aw:MyAttr="content">content</aw:Child>
  </aw:Data>
  <ad:Data xmlns:ad="http://www.adatum.com">
    <ad:Child ad:MyAttr="content">content</ad:Child>
  </ad:Data>
</Root>

```

# Performance of Chained Queries (LINQ to XML) (C#)

6/5/2019 • 2 minutes to read • [Edit Online](#)

One of the most important benefits of LINQ (and LINQ to XML) is that chained queries can perform as well as a single larger, more complicated query.

A chained query is a query that uses another query as its source. For example, in the following simple code,

`query2` has `query1` as its source:

```
XElement root = new XElement("Root",
    new XElement("Child", 1),
    new XElement("Child", 2),
    new XElement("Child", 3),
    new XElement("Child", 4)
);

var query1 = from x in root.Elements("Child")
             where (int)x >= 3
             select x;

var query2 = from e in query1
             where (int)e % 2 == 0
             select e;

foreach (var i in query2)
    Console.WriteLine("{0}", (int)i);
```

This example produces the following output:

```
4
```

This chained query provides the same performance profile as iterating through a linked list.

- The `Elements` axis has essentially the same performance as iterating through a linked list. `Elements` is implemented as an iterator with deferred execution. This means that it does some work in addition to iterating through the linked list, such as allocating the iterator object and keeping track of execution state. This work can be divided into two categories: the work that is done at the time the iterator is set up, and the work that is done during each iteration. The setup work is a small, fixed amount of work and the work done during each iteration is proportional to the number of items in the source collection.
- In `query1`, the `where` clause causes the query to call the `Where` method. This method is also implemented as an iterator. The setup work consists of instantiating the delegate that will reference the lambda expression, plus the normal setup for an iterator. With each iteration, the delegate is called to execute the predicate. The setup work and the work done during each iteration is the similar to the work done while iterating through the axis.
- In `query1`, the `select` clause causes the query to call the `Select` method. This method has the same performance profile as the `Where` method.
- In `query2`, both the `where` clause and the `select` clause have the same performance profile as in `query1`.

The iteration through `query2` is therefore directly proportional to the number of items in the source of the first query, in other words, linear time. A corresponding Visual Basic example would have the same performance profile.

For more information on iterators, see [yield](#).

For a more detailed tutorial on chaining queries together, see [Tutorial: Chaining Queries Together](#).

# Atomized XName and XNamespace Objects (LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

[XName](#) and [XNamespace](#) objects are *atomized*; that is, if they contain the same qualified name, they refer to the same object. This yields performance benefits for queries: When you compare two atomized names for equality, the underlying intermediate language only has to determine whether the two references point to the same object. The underlying code does not have to do string comparisons, which would be time consuming.

## Atomization Semantics

Atomization means that if two [XName](#) objects have the same local name, and they are in the same namespace, they share the same instance. In the same way, if two [XNamespace](#) objects have the same namespace URI, they share the same instance.

For a class to enable atomized objects, the constructor for the class must be private, not public. This is because if the constructor were public, you could create a non-atomized object. The [XName](#) and [XNamespace](#) classes implement an implicit conversion operator to convert a string into an [XName](#) or [XNamespace](#). This is how you get an instance of these objects. You cannot get an instance by using a constructor, because the constructor is inaccessible.

[XName](#) and [XNamespace](#) also implement the equality and inequality operators, to determine whether the two objects being compared are references to the same instance.

## Example

The following code creates some [XElement](#) objects and demonstrates that identical names share the same instance.

```
XElement r1 = new XElement("Root", "data1");
XElement r2 = XElement.Parse("<Root>data2</Root>");

if ((object)r1.Name == (object)r2.Name)
    Console.WriteLine("r1 and r2 have names that refer to the same instance.");
else
    Console.WriteLine("Different");

XName n = "Root";

if ((object)n == (object)r1.Name)
    Console.WriteLine("The name of r1 and the name in 'n' refer to the same instance.");
else
    Console.WriteLine("Different");
```

This example produces the following output:

```
r1 and r2 have names that refer to the same instance.
The name of r1 and the name in 'n' refer to the same instance.
```

As mentioned earlier, the benefit of atomized objects is that when you use one of the axis methods that take an [XName](#) as a parameter, the axis method only has to determine that two names reference the same instance to select the desired elements.

The following example passes an [XName](#) to the [Descendants](#) method call, which then has better performance because of the atomization pattern.

```
XElement root = new XElement("Root",
    new XElement("C1", 1),
    new XElement("Z1",
        new XElement("C1", 2),
        new XElement("C1", 1)
    )
);

var query = from e in root.Descendants("C1")
            where (int)e == 1
            select e;

foreach (var z in query)
    Console.WriteLine(z);
```

This example produces the following output:

```
<C1>1</C1>
<C1>1</C1>
```

# Pre-Atomization of XName Objects (LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

One way to improve performance in LINQ to XML is to pre-atomize [XName](#) objects. Pre-atomization means that you assign a string to an [XName](#) object before you create the XML tree by using the constructors of the [XElement](#) and [XAttribute](#) classes. Then, instead of passing a string to the constructor, which would use the implicit conversion from string to [XName](#), you pass the initialized [XName](#) object.

This improves performance when you create a large XML tree in which specific names are repeated. To do this, you declare and initialize [XName](#) objects before you construct the XML tree, and then use the [XName](#) objects instead of specifying strings for the element and attribute names. This technique can yield significant performance gains if you are creating a large number of elements (or attributes) with the same name.

You should test pre-atomization with your scenario to decide if you should use it.

## Example

The following example demonstrates this.

```
XName Root = "Root";
XName Data = "Data";
XName ID = "ID";

XElement root = new XElement(Root,
    new XElement(Data,
        new XAttribute(ID, "1"),
        "4,100,000"),
    new XElement(Data,
        new XAttribute(ID, "2"),
        "3,700,000"),
    new XElement(Data,
        new XAttribute(ID, "3"),
        "1,150,000")
);

Console.WriteLine(root);
```

This example produces the following output:

```
<Root>
  <Data ID="1">4,100,000</Data>
  <Data ID="2">3,700,000</Data>
  <Data ID="3">1,150,000</Data>
</Root>
```

The following example shows the same technique where the XML document is in a namespace:

```

XNamespace aw = "http://www.adventure-works.com";
XName Root = aw + "Root";
XName Data = aw + "Data";
XName ID = "ID";

XElement root = new XElement(Root,
    new XAttribute(XNamespace.Xmlns + "aw", aw),
    new XElement(Data,
        new XAttribute(ID, "1"),
        "4,100,000"),
    new XElement(Data,
        new XAttribute(ID, "2"),
        "3,700,000"),
    new XElement(Data,
        new XAttribute(ID, "3"),
        "1,150,000")
);

Console.WriteLine(root);

```

This example produces the following output:

```

<aw:Root xmlns:aw="http://www.adventure-works.com">
  <aw:Data ID="1">4,100,000</aw:Data>
  <aw:Data ID="2">3,700,000</aw:Data>
  <aw:Data ID="3">1,150,000</aw:Data>
</aw:Root>

```

The following example is more similar to what you will likely encounter in the real world. In this example, the content of the element is supplied by a query:

```

XName Root = "Root";
XName Data = "Data";
XName ID = "ID";

DateTime t1 = DateTime.Now;
XElement root = new XElement(Root,
    from i in System.Linq.Enumerable.Range(1, 100000)
    select new XElement(Data,
        new XAttribute(ID, i),
        i * 5)
);
DateTime t2 = DateTime.Now;

Console.WriteLine("Time to construct:{0}", t2 - t1);

```

The previous example performs better than the following example, in which names are not pre-atomized:

```

DateTime t1 = DateTime.Now;
XElement root = new XElement("Root",
    from i in System.Linq.Enumerable.Range(1, 100000)
    select new XElement("Data",
        new XAttribute("ID", i),
        i * 5)
);
DateTime t2 = DateTime.Now;

Console.WriteLine("Time to construct:{0}", t2 - t1);

```



## See also

- [Atomized XName and XNamespace Objects \(LINQ to XML\) \(C#\)](#)

# Statically Compiled Queries (LINQ to XML) (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

One of the most important performance benefits LINQ to XML, as opposed to [XmlDocument](#), is that queries in LINQ to XML are statically compiled, whereas XPath queries must be interpreted at run time. This feature is built in to LINQ to XML, so you do not have to perform extra steps to take advantage of it, but it is helpful to understand the distinction when choosing between the two technologies. This topic explains the difference.

## Statically Compiled Queries vs. XPath

The following example shows how to get the descendant elements with a specified name, and with an attribute with a specified value.

The following is the equivalent XPath expression:

```
//Address[@Type='Shipping']
```

```
XDocument po = XDocument.Load("PurchaseOrders.xml");

IEnumerable<XElement> list1 =
    from el in po.Descendants("Address")
    where (string)el.Attribute("Type") == "Shipping"
    select el;

foreach (XElement el in list1)
    Console.WriteLine(el);
```

The query expression in this example is re-written by the compiler to method-based query syntax. The following example, which is written in method-based query syntax, produces the same results as the previous one:

```
XDocument po = XDocument.Load("PurchaseOrders.xml");

IEnumerable<XElement> list1 =
    po
    .Descendants("Address")
    .Where(el => (string)el.Attribute("Type") == "Shipping");

foreach (XElement el in list1)
    Console.WriteLine(el);
```

The [Where](#) method is an extension method. For more information, see [Extension Methods](#). Because [Where](#) is an extension method, the query above is compiled as though it were written as follows:

```
XDocument po = XDocument.Load("PurchaseOrders.xml");

IEnumerable<XElement> list1 =
    System.Linq.Enumerable.Where(
        po.Descendants("Address"),
        el => (string)el.Attribute("Type") == "Shipping");

foreach (XElement el in list1)
    Console.WriteLine(el);
```

This example produces exactly the same results as the previous two examples. This illustrates the fact that queries are effectively compiled into statically linked method calls. This, combined with the deferred execution semantics of iterators, improves performance. For more information about the deferred execution semantics of iterators, see [Deferred Execution and Lazy Evaluation in LINQ to XML \(C#\)](#).

#### NOTE

These examples are representative of the code that the compiler might write. The actual implementation might differ slightly from these examples, but the performance will be the same or similar to these examples.

## Executing XPath Expressions with XmlDocument

The following example uses [XmlDocument](#) to accomplish the same results as the previous examples:

```
XmlReader reader = XmlReader.Create("PurchaseOrders.xml");
XmlDocument doc = new XmlDocument();
doc.Load(reader);
XmlNodeList nl = doc.SelectNodes(".//Address[@Type='Shipping']");
foreach (XmlNode n in nl)
    Console.WriteLine(n.OuterXml);
reader.Close();
```

This query returns the same output as the examples that use LINQ to XML; the only difference is that LINQ to XML indents the printed XML, whereas [XmlDocument](#) does not.

However, the [XmlDocument](#) approach generally does not perform as well as LINQ to XML, because the [SelectNodes](#) method must do the following internally every time it is called:

- It parses the string that contains the XPath expression, breaking the string into tokens.
- It validates the tokens to make sure that the XPath expression is valid.
- It translates the expression into an internal expression tree.
- It iterates through the nodes, appropriately selecting the nodes for the result set based on the evaluation of the expression.

This is significantly more than the work done by the corresponding LINQ to XML query. The specific performance difference varies for different types of queries, but in general LINQ to XML queries do less work, and therefore perform better, than evaluating XPath expressions using [XmlDocument](#).

# LINQ to XML Annotations

6/5/2019 • 2 minutes to read • [Edit Online](#)

Annotations in LINQ to XML enable you to associate any arbitrary object of any arbitrary type with any XML component in an XML tree.

To add an annotation to an XML component, such as an [XElement](#) or [XAttribute](#), you call the [AddAnnotation](#) method. You retrieve annotations by type.

Note that annotations are not part of the XML infoset; they are not serialized or deserialized.

## Methods

You can use the following methods when working with annotations:

METHOD	DESCRIPTION
<a href="#">AddAnnotation</a>	Adds an object to the annotation list of an <a href="#">XObject</a> .
<a href="#">Annotation</a>	Gets the first annotation object of the specified type from an <a href="#">XObject</a> .
<a href="#">Annotations</a>	Gets a collection of annotations of the specified type for an <a href="#">XObject</a> .
<a href="#">RemoveAnnotations</a>	Removes the annotations of the specified type from an <a href="#">XObject</a> .

# LINQ to XML Events (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

LINQ to XML events enable you to be notified when an XML tree is altered.

You can add events to an instance of any [XObject](#). The event handler will then receive events for modifications to that [XObject](#) and any of its descendants. For example, you can add an event handler to the root of the tree, and handle all modifications to the tree from that event handler.

For examples of LINQ to XML events, see [Changing](#) and [Changed](#).

## Types and Events

You use the following types when working with events:

TYPE	DESCRIPTION
<a href="#">XObjectChange</a>	Specifies the event type when an event is raised for an <a href="#">XObject</a> .
<a href="#">XObjectChangeEventArgs</a>	Provides data for the <a href="#">Changing</a> and <a href="#">Changed</a> events.

The following events are raised when you modify an XML tree:

EVENT	DESCRIPTION
<a href="#">Changing</a>	Occurs just before this <a href="#">XObject</a> or any of its descendants is going to change.
<a href="#">Changed</a>	Occurs when an <a href="#">XObject</a> has changed or any of its descendants have changed.

## Example

### Description

Events are useful when you want to maintain some aggregate information in an XML tree. For example, you may want maintain an invoice total that is the sum of the line items of the invoice. This example uses events to maintain the total of all of the child elements under the complex element `Items`.

### Code

```

XElement root = new XElement("Root",
    new XElement("Total", "0"),
    new XElement("Items")
);
XElement total = root.Element("Total");
XElement items = root.Element("Items");
items.Changed += (object sender, XObjectChangeEventArgs cea) =>
{
    switch (cea.ObjectChange)
    {
        case XObjectChange.Add:
            if (sender is XElement)
                total.Value = ((int)total + (int)(XElement)sender).ToString();
            if (sender is XText)
                total.Value = ((int)total + (int)((XText)sender).Parent).ToString();
            break;
        case XObjectChange.Remove:
            if (sender is XElement)
                total.Value = ((int)total - (int)(XElement)sender).ToString();
            if (sender is XText)
                total.Value = ((int)total - Int32.Parse(((XText)sender).Value)).ToString();
            break;
    }
    Console.WriteLine("Changed {0} {1}",
        sender.GetType().ToString(), cea.ObjectChange.ToString());
};
items.SetElementValue("Item1", 25);
items.SetElementValue("Item2", 50);
items.SetElementValue("Item2", 75);
items.SetElementValue("Item3", 133);
items.SetElementValue("Item1", null);
items.SetElementValue("Item4", 100);
Console.WriteLine("Total:{0}", (int)total);
Console.WriteLine(root);

```

## Comments

This code produces the following output:

```

Changed System.Xml.Linq.XElement Add
Changed System.Xml.Linq.XElement Add
Changed System.Xml.Linq.XText Remove
Changed System.Xml.Linq.XText Add
Changed System.Xml.Linq.XElement Add
Changed System.Xml.Linq.XElement Remove
Changed System.Xml.Linq.XElement Add
Total:308
<Root>
  <Total>308</Total>
  <Items>
    <Item2>75</Item2>
    <Item3>133</Item3>
    <Item4>100</Item4>
  </Items>
</Root>

```

# Programming with Nodes (C#)

6/4/2019 • 3 minutes to read • [Edit Online](#)

LINQ to XML developers who need to write programs such as an XML editor, a transform system, or a report writer often need to write programs that work at a finer level of granularity than elements and attributes. They often need to work at the node level, manipulating text nodes, processing instructions, and comments. This topic provides some details about programming at the node level.

## Node Details

There are a number of details of programming that a programmer working at the node level should know.

### Parent Property of Children Nodes of XDocument is Set to Null

The [Parent](#) property contains the parent [XElement](#), not the parent node. Child nodes of [XDocument](#) have no parent [XElement](#). Their parent is the document, so the [Parent](#) property for those nodes is set to null.

The following example demonstrates this:

```
XDocument doc = XDocument.Parse(@"<!-- a comment --><Root/>");
Console.WriteLine(doc.Nodes().OfType<XComment>().First().Parent == null);
Console.WriteLine(doc.Root.Parent == null);
```

This example produces the following output:

```
True
True
```

### Adjacent Text Nodes are Possible

In a number of XML programming models, adjacent text nodes are always merged. This is sometimes called normalization of text nodes. LINQ to XML does not normalize text nodes. If you add two text nodes to the same element, it will result in adjacent text nodes. However, if you add content specified as a string rather than as an [XText](#) node, LINQ to XML might merge the string with an adjacent text node.

The following example demonstrates this:

```
XElement xmlTree = new XElement("Root", "Content");

Console.WriteLine(xmlTree.Nodes().OfType<XText>().Count());

// this does not add a new text node
xmlTree.Add("new content");
Console.WriteLine(xmlTree.Nodes().OfType<XText>().Count());

// this does add a new, adjacent text node
xmlTree.Add(new XText("more text"));
Console.WriteLine(xmlTree.Nodes().OfType<XText>().Count());
```

This example produces the following output:

1  
1  
2

## Empty Text Nodes are Possible

In some XML programming models, text nodes are guaranteed to not contain the empty string. The reasoning is that such a text node has no impact on serialization of the XML. However, for the same reason that adjacent text nodes are possible, if you remove the text from a text node by setting its value to the empty string, the text node itself will not be deleted.

```
XElement xmlTree = new XElement("Root", "Content");
XText textNode = xmlTree.Nodes().OfType<XText>().First();

// the following line does not cause the removal of the text node.
textNode.Value = "";

XText textNode2 = xmlTree.Nodes().OfType<XText>().First();
Console.WriteLine(">>{0}<<", textNode2);
```

This example produces the following output:

```
>><<
```

## An Empty Text Node Impacts Serialization

If an element contains only a child text node that is empty, it is serialized with the long tag syntax: `<Child></Child>`.  
If an element contains no child nodes whatsoever, it is serialized with the short tag syntax: `<Child />`.

```
XElement child1 = new XElement("Child1",
    new XText("")
);
XElement child2 = new XElement("Child2");
Console.WriteLine(child1);
Console.WriteLine(child2);
```

This example produces the following output:

```
<Child1></Child1>
<Child2 />
```

## Namespaces are Attributes in the LINQ to XML Tree

Even though namespace declarations have identical syntax to attributes, in some programming interfaces, such as XSLT and XPath, namespace declarations are not considered to be attributes. However, in LINQ to XML, namespaces are stored as [XAttribute](#) objects in the XML tree. If you iterate through the attributes for an element that contains a namespace declaration, you will see the namespace declaration as one of the items in the returned collection.

The [IsNamespaceDeclaration](#) property indicates whether an attribute is a namespace declaration.



```

XElement root = XElement.Parse(
@"<Root
  xmlns='http://www.adventure-works.com'
  xmlns:fc='www.fourthcoffee.com'
  AnAttribute='abc' />");
foreach (XAttribute att in root.Attributes())
  Console.WriteLine("{0} IsNamespaceDeclaration:{1}", att, att.IsNamespaceDeclaration);

```

This example produces the following output:

```

xmlns="http://www.adventure-works.com" IsNamespaceDeclaration:True
xmlns:fc="www.fourthcoffee.com" IsNamespaceDeclaration:True
AnAttribute="abc" IsNamespaceDeclaration:False

```

### XPath Axis Methods Do Not Return Child White Space of XDocument

LINQ to XML allows for child text nodes of an [XDocument](#), as long as the text nodes contain only white space. However, the XPath object model does not include white space as child nodes of a document, so when you iterate through the children of an [XDocument](#) using the [Nodes](#) axis, white-space text nodes will be returned. However, when you iterate through the children of an [XDocument](#) using the XPath axis methods, white-space text nodes will not be returned.

```

// Create a document with some white-space child nodes of the document.
XDocument root = XDocument.Parse(
@"<?xml version='1.0' encoding='utf-8' standalone='yes'?>

<Root/>

<!--a comment-->
", LoadOptions.PreserveWhitespace);

// count the white-space child nodes using LINQ to XML
Console.WriteLine(root.Nodes().OfType<XText>().Count());

// count the white-space child nodes using XPathEvaluate
Console.WriteLine(((IEnumerable)root.XPathEvaluate("text()")).OfType<XText>().Count());

```

This example produces the following output:

```

3
0

```

### XDeclaration Objects are not Nodes

When you iterate through the children nodes of an [XDocument](#), you will not see the XML declaration object. It is a property of the document, not a child node of it.

```

XDocument doc = new XDocument(
  new XDeclaration("1.0", "utf-8", "yes"),
  new XElement("Root")
);
doc.Save("Temp.xml");
Console.WriteLine(File.ReadAllText("Temp.xml"));

// this shows that there is only one child node of the document
Console.WriteLine(doc.Nodes().Count());

```

This example produces the following output:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
```

```
<Root />
```

```
1
```

# Mixed Declarative Code/Imperative Code Bugs (LINQ to XML) (C#)

6/4/2019 • 5 minutes to read • [Edit Online](#)

LINQ to XML contains various methods that allow you to modify an XML tree directly. You can add elements, delete elements, change the contents of an element, add attributes, and so on. This programming interface is described in [Modifying XML Trees \(LINQ to XML\) \(C#\)](#). If you are iterating through one of the axes, such as [Elements](#), and you are modifying the XML tree as you iterate through the axis, you can end up with some strange bugs.

This problem is sometimes known as "The Halloween Problem".

## Definition of the Problem

When you write some code using LINQ that iterates through a collection, you are writing code in a declarative style. It is more akin to describing *what* you want, rather than *how* you want to get it done. If you write code that 1) gets the first element, 2) tests it for some condition, 3) modifies it, and 4) puts it back into the list, then this would be imperative code. You are telling the computer *how* to do what you want done.

Mixing these styles of code in the same operation is what leads to problems. Consider the following:

Suppose you have a linked list with three items in it (a, b, and c):

```
a -> b -> c
```

Now, suppose that you want to move through the linked list, adding three new items (a', b', and c'). You want the resulting linked list to look like this:

```
a -> a' -> b -> b' -> c -> c'
```

So you write code that iterates through the list, and for every item, adds a new item right after it. What happens is that your code will first see the `a` element, and insert `a'` after it. Now, your code will move to the next node in the list, which is now `a'`! It happily adds a new item to the list, `a''`.

How would you solve this in the real world? Well, you might make a copy of the original linked list, and create a completely new list. Or if you are writing purely imperative code, you might find the first item, add the new item, and then advance twice in the linked list, advancing over the element that you just added.

## Adding While Iterating

For example, suppose you want to write some code that for every element in a tree, you want to create a duplicate element:

```
XElement root = new XElement("Root",
    new XElement("A", "1"),
    new XElement("B", "2"),
    new XElement("C", "3")
);
foreach (XElement e in root.Elements())
    root.Add(new XElement(e.Name, (string)e));
```

This code goes into an infinite loop. The `foreach` statement iterates through the `Elements()` axis, adding new

elements to the `doc` element. It ends up iterating also through the elements it just added. And because it allocates new objects with every iteration of the loop, it will eventually consume all available memory.

You can fix this problem by pulling the collection into memory using the [ToList](#) standard query operator, as follows:

```
XElement root = new XElement("Root",
    new XElement("A", "1"),
    new XElement("B", "2"),
    new XElement("C", "3")
);
foreach (XElement e in root.Elements().ToList())
    root.Add(new XElement(e.Name, (string)e));
Console.WriteLine(root);
```

Now the code works. The resulting XML tree is the following:

```
<Root>
  <A>1</A>
  <B>2</B>
  <C>3</C>
  <A>1</A>
  <B>2</B>
  <C>3</C>
</Root>
```

## Deleting While Iterating

If you want to delete all nodes at a certain level, you might be tempted to write code like the following:

```
XElement root = new XElement("Root",
    new XElement("A", "1"),
    new XElement("B", "2"),
    new XElement("C", "3")
);
foreach (XElement e in root.Elements())
    e.Remove();
Console.WriteLine(root);
```

However, this does not do what you want. In this situation, after you have removed the first element, A, it is removed from the XML tree contained in `root`, and the code in the `Elements` method that is doing the iterating cannot find the next element.

The preceding code produces the following output:

```
<Root>
  <B>2</B>
  <C>3</C>
</Root>
```

The solution again is to call [ToList](#) to materialize the collection, as follows:

```

XElement root = new XElement("Root",
    new XElement("A", "1"),
    new XElement("B", "2"),
    new XElement("C", "3")
);
foreach (XElement e in root.Elements().ToList())
    e.Remove();
Console.WriteLine(root);

```

This produces the following output:

```
<Root />
```

Alternatively, you can eliminate the iteration altogether by calling [RemoveAll](#) on the parent element:

```

XElement root = new XElement("Root",
    new XElement("A", "1"),
    new XElement("B", "2"),
    new XElement("C", "3")
);
root.RemoveAll();
Console.WriteLine(root);

```

## Why Can't LINQ Automatically Handle This?

One approach would be to always bring everything into memory instead of doing lazy evaluation. However, it would be very expensive in terms of performance and memory use. In fact, if LINQ and (LINQ to XML) were to take this approach, it would fail in real-world situations.

Another possible approach would be to put in some sort of transaction syntax into LINQ, and have the compiler attempt to analyze the code and determine if any particular collection needed to be materialized. However, attempting to determine all code that has side-effects is incredibly complex. Consider the following code:

```

var z =
    from e in root.Elements()
    where TestSomeCondition(e)
    select DoMyProjection(e);

```

Such analysis code would need to analyze the methods `TestSomeCondition` and `DoMyProjection`, and all methods that those methods called, to determine if any code had side-effects. But the analysis code could not just look for any code that had side-effects. It would need to select for just the code that had side-effects on the child elements of `root` in this situation.

LINQ to XML does not attempt to do any such analysis.

It is up to you to avoid these problems.

## Guidance

First, do not mix declarative and imperative code.

Even if you know exactly the semantics of your collections and the semantics of the methods that modify the XML tree, if you write some clever code that avoids these categories of problems, your code will need to be maintained by other developers in the future, and they may not be as clear on the issues. If you mix declarative and imperative coding styles, your code will be more brittle.

If you write code that materializes a collection so that these problems are avoided, note it with comments as appropriate in your code, so that maintenance programmers will understand the issue.

Second, if performance and other considerations allow, use only declarative code. Don't modify your existing XML tree. Generate a new one.

```
XElement root = new XElement("Root",
    new XElement("A", "1"),
    new XElement("B", "2"),
    new XElement("C", "3")
);
XElement newRoot = new XElement("Root",
    root.Elements(),
    root.Elements()
);
Console.WriteLine(newRoot);
```

# How to: Stream XML Fragments with Access to Header Information (C#)

6/4/2019 • 3 minutes to read • [Edit Online](#)

Sometimes you have to read arbitrarily large XML files, and write your application so that the memory footprint of the application is predictable. If you attempt to populate an XML tree with a large XML file, your memory usage will be proportional to the size of the file—that is, excessive. Therefore, you should use a streaming technique instead.

One option is to write your application using [XmlReader](#). However, you might want to use LINQ to query the XML tree. If this is the case, you can write your own custom axis method. For more information, see [How to: Write a LINQ to XML Axis Method \(C#\)](#).

To write your own axis method, you write a small method that uses the [XmlReader](#) to read nodes until it reaches one of the nodes in which you are interested. The method then calls [ReadFrom](#), which reads from the [XmlReader](#) and instantiates an XML fragment. It then yields each fragment through `yield return` to the method that is enumerating your custom axis method. You can then write LINQ queries on your custom axis method.

Streaming techniques are best applied in situations where you need to process the source document only once, and you can process the elements in document order. Certain standard query operators, such as [OrderBy](#), iterate their source, collect all of the data, sort it, and then finally yield the first item in the sequence. Note that if you use a query operator that materializes its source before yielding the first item, you will not retain a small memory footprint.

## Example

Sometimes the problem gets just a little more interesting. In the following XML document, the consumer of your custom axis method also has to know the name of the customer that each item belongs to.

```

<?xml version="1.0" encoding="utf-8" ?>
<Root>
  <Customer>
    <Name>A. Datum Corporation</Name>
    <Item>
      <Key>0001</Key>
    </Item>
    <Item>
      <Key>0002</Key>
    </Item>
    <Item>
      <Key>0003</Key>
    </Item>
    <Item>
      <Key>0004</Key>
    </Item>
  </Customer>
  <Customer>
    <Name>Fabrikam, Inc.</Name>
    <Item>
      <Key>0005</Key>
    </Item>
    <Item>
      <Key>0006</Key>
    </Item>
    <Item>
      <Key>0007</Key>
    </Item>
    <Item>
      <Key>0008</Key>
    </Item>
  </Customer>
  <Customer>
    <Name>Southridge Video</Name>
    <Item>
      <Key>0009</Key>
    </Item>
    <Item>
      <Key>0010</Key>
    </Item>
  </Customer>
</Root>

```

The approach that this example takes is to also watch for this header information, save the header information, and then build a small XML tree that contains both the header information and the detail that you are enumerating. The axis method then yields this new, small XML tree. The query then has access to the header information as well as the detail information.

This approach has a small memory footprint. As each detail XML fragment is yielded, no references are kept to the previous fragment, and it is available for garbage collection. Note that this technique creates many short lived objects on the heap.

The following example shows how to implement and use a custom axis method that streams XML fragments from the file specified by the URI. This custom axis is specifically written such that it expects a document that has `Customer`, `Name`, and `Item` elements, and that those elements will be arranged as in the above `Source.xml` document. It is a simplistic implementation. A more robust implementation would be prepared to parse an invalid document.



```

static IEnumerable<XElement> StreamCustomerItem(string uri)
{
    using (XmlReader reader = XmlReader.Create(uri))
    {
        XElement name = null;
        XElement item = null;

        reader.MoveToContent();

        // Parse the file, save header information when encountered, and yield the
        // Item XElement objects as they are created.

        // loop through Customer elements
        while (reader.Read())
        {
            if (reader.NodeType == XmlNodeType.Element
                && reader.Name == "Customer")
            {
                // move to Name element
                while (reader.Read())
                {
                    if (reader.NodeType == XmlNodeType.Element &&
                        reader.Name == "Name")
                    {
                        name = XElement.ReadFrom(reader) as XElement;
                        break;
                    }
                }

                // loop through Item elements
                while (reader.Read())
                {
                    if (reader.NodeType == XmlNodeType.EndElement)
                        break;
                    if (reader.NodeType == XmlNodeType.Element
                        && reader.Name == "Item")
                    {
                        item = XElement.ReadFrom(reader) as XElement;
                        if (item != null) {
                            XElement tempRoot = new XElement("Root",
                                new XElement(name)
                            );
                            tempRoot.Add(item);
                            yield return item;
                        }
                    }
                }
            }
        }
    }
}

static void Main(string[] args)
{
    XElement xmlTree = new XElement("Root",
        from el in StreamCustomerItem("Source.xml")
        where (int)el.Element("Key") >= 3 && (int)el.Element("Key") <= 7
        select new XElement("Item",
            new XElement("Customer", (string)el.Parent.Element("Name")),
            new XElement(el.Element("Key"))
        )
    );
    Console.WriteLine(xmlTree);
}

```

This code produces the following output:

```
<Root>
  <Item>
    <Customer>A. Datum Corporation</Customer>
    <Key>0003</Key>
  </Item>
  <Item>
    <Customer>A. Datum Corporation</Customer>
    <Key>0004</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0005</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0006</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0007</Key>
  </Item>
</Root>
```

# How to: Perform Streaming Transform of Large XML Documents (C#)

6/4/2019 • 4 minutes to read • [Edit Online](#)

Sometimes you have to transform large XML files, and write your application so that the memory footprint of the application is predictable. If you try to populate an XML tree with a very large XML file, your memory usage will be proportional to the size of the file (that is, excessive). Therefore, you should use a streaming technique instead.

Streaming techniques are best applied in situations where you need to process the source document only once, and you can process the elements in document order. Certain standard query operators, such as [OrderBy](#), iterate their source, collect all of the data, sort it, and then finally yield the first item in the sequence. Note that if you use a query operator that materializes its source before yielding the first item, you will not retain a small memory footprint for your application.

Even if you use the technique described in [How to: Stream XML Fragments with Access to Header Information \(C#\)](#), if you try to assemble an XML tree that contains the transformed document, memory usage will be too great.

There are two main approaches. One approach is to use the deferred processing characteristics of [XStreamingElement](#). Another approach is to create an [XmlWriter](#), and use the capabilities of LINQ to XML to write elements to an [XmlWriter](#). This topic demonstrates both approaches.

## Example

The following example builds on the example in [How to: Stream XML Fragments with Access to Header Information \(C#\)](#).

This example uses the deferred execution capabilities of [XStreamingElement](#) to stream the output. This example can transform a very large document while maintaining a small memory footprint.

Note that the custom axis (`StreamCustomerItem`) is specifically written so that it expects a document that has `Customer`, `Name`, and `Item` elements, and that those elements will be arranged as in the following Source.xml document. A more robust implementation, however, would be prepared to parse an invalid document.

The following is the source document, Source.xml:

```
<?xml version="1.0" encoding="utf-8" ?>
<Root>
  <Customer>
    <Name>A. Datum Corporation</Name>
    <Item>
      <Key>0001</Key>
    </Item>
    <Item>
      <Key>0002</Key>
    </Item>
    <Item>
      <Key>0003</Key>
    </Item>
    <Item>
      <Key>0004</Key>
    </Item>
  </Customer>
  <Customer>
    <Name>Fabrikam, Inc.</Name>
    <Item>
      <Key>0005</Key>
    </Item>
    <Item>
      <Key>0006</Key>
    </Item>
    <Item>
      <Key>0007</Key>
    </Item>
    <Item>
      <Key>0008</Key>
    </Item>
  </Customer>
  <Customer>
    <Name>Southridge Video</Name>
    <Item>
      <Key>0009</Key>
    </Item>
    <Item>
      <Key>0010</Key>
    </Item>
  </Customer>
</Root>
```

```

static IEnumerable<XElement> StreamCustomerItem(string uri)
{
    using (XmlReader reader = XmlReader.Create(uri))
    {
        XElement name = null;
        XElement item = null;

        reader.MoveToContent();

        // Parse the file, save header information when encountered, and yield the
        // Item XElement objects as they are created.

        // loop through Customer elements
        while (reader.Read())
        {
            if (reader.NodeType == XmlNodeType.Element
                && reader.Name == "Customer")
            {
                // move to Name element
                while (reader.Read())
                {
                    if (reader.NodeType == XmlNodeType.Element &&
                        reader.Name == "Name")
                    {
                        name = XElement.ReadFrom(reader) as XElement;
                        break;
                    }
                }

                // loop through Item elements
                while (reader.Read())
                {
                    if (reader.NodeType == XmlNodeType.EndElement)
                        break;
                    if (reader.NodeType == XmlNodeType.Element
                        && reader.Name == "Item")
                    {
                        item = XElement.ReadFrom(reader) as XElement;
                        if (item != null)
                        {
                            XElement tempRoot = new XElement("Root",
                                new XElement(name)
                            );
                            tempRoot.Add(item);
                            yield return item;
                        }
                    }
                }
            }
        }
    }
}

static void Main(string[] args)
{
    XStreamingElement root = new XStreamingElement("Root",
        from el in StreamCustomerItem("Source.xml")
        select new XElement("Item",
            new XElement("Customer", (string)el.Parent.Element("Name")),
            new XElement(el.Element("Key"))
        )
    );
    root.Save("Test.xml");
    Console.WriteLine(File.ReadAllText("Test.xml"));
}

```

This code produces the following output:

```
<?xml version="1.0" encoding="utf-8"?>
<Root>
  <Item>
    <Customer>A. Datum Corporation</Customer>
    <Key>0001</Key>
  </Item>
  <Item>
    <Customer>A. Datum Corporation</Customer>
    <Key>0002</Key>
  </Item>
  <Item>
    <Customer>A. Datum Corporation</Customer>
    <Key>0003</Key>
  </Item>
  <Item>
    <Customer>A. Datum Corporation</Customer>
    <Key>0004</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0005</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0006</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0007</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0008</Key>
  </Item>
  <Item>
    <Customer>Southridge Video</Customer>
    <Key>0009</Key>
  </Item>
  <Item>
    <Customer>Southridge Video</Customer>
    <Key>0010</Key>
  </Item>
</Root>
```

## Example

The following example also builds on the example in [How to: Stream XML Fragments with Access to Header Information \(C#\)](#).

This example uses the capability of LINQ to XML to write elements to an [XmlWriter](#). This example can transform a very large document while maintaining a small memory footprint.

Note that the custom axis (`StreamCustomerItem`) is specifically written so that it expects a document that has `Customer`, `Name`, and `Item` elements, and that those elements will be arranged as in the following Source.xml document. A more robust implementation, however, would either validate the source document with an XSD, or would be prepared to parse an invalid document.

This example uses the same source document, Source.xml, as the previous example in this topic. It also produces exactly the same output.

Using [XStreamingElement](#) for streaming the output XML is preferred over writing to an [XmlWriter](#).

```

static IEnumerable<XElement> StreamCustomerItem(string uri)
{
    using (XmlReader reader = XmlReader.Create(uri))
    {
        XElement name = null;
        XElement item = null;

        reader.MoveToContent();

        // Parse the file, save header information when encountered, and yield the
        // Item XElement objects as they are created.

        // loop through Customer elements
        while (reader.Read())
        {
            if (reader.NodeType == XmlNodeType.Element
                && reader.Name == "Customer")
            {
                // move to Name element
                while (reader.Read())
                {
                    if (reader.NodeType == XmlNodeType.Element &&
                        reader.Name == "Name")
                    {
                        name = XElement.ReadFrom(reader) as XElement;
                        break;
                    }
                }

                // loop through Item elements
                while (reader.Read())
                {
                    if (reader.NodeType == XmlNodeType.EndElement)
                        break;
                    if (reader.NodeType == XmlNodeType.Element
                        && reader.Name == "Item")
                    {
                        item = XElement.ReadFrom(reader) as XElement;
                        if (item != null) {
                            XElement tempRoot = new XElement("Root",
                                new XElement(name)
                            );
                            tempRoot.Add(item);
                            yield return item;
                        }
                    }
                }
            }
        }
    }
}

static void Main(string[] args)
{
    IEnumerable<XElement> srcTree =
        from el in StreamCustomerItem("Source.xml")
        select new XElement("Item",
            new XElement("Customer", (string)el.Parent.Element("Name")),
            new XElement(el.Element("Key"))
        );
    XmlWriterSettings xws = new XmlWriterSettings();
    xws.OmitXmlDeclaration = true;
    xws.Indent = true;
    using (XmlWriter xw = XmlWriter.Create("Output.xml", xws)) {
        xw.WriteStartElement("Root");
        foreach (XElement el in srcTree)
            el.WriteTo(xw);
        xw.WriteEndElement();
    }
}

```

```

    }

    string str = File.ReadAllText("Output.xml");
    Console.WriteLine(str);
}

```

This code produces the following output:

```

<Root>
  <Item>
    <Customer>A. Datum Corporation</Customer>
    <Key>0001</Key>
  </Item>
  <Item>
    <Customer>A. Datum Corporation</Customer>
    <Key>0002</Key>
  </Item>
  <Item>
    <Customer>A. Datum Corporation</Customer>
    <Key>0003</Key>
  </Item>
  <Item>
    <Customer>A. Datum Corporation</Customer>
    <Key>0004</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0005</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0006</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0007</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0008</Key>
  </Item>
  <Item>
    <Customer>Southridge Video</Customer>
    <Key>0009</Key>
  </Item>
  <Item>
    <Customer>Southridge Video</Customer>
    <Key>0010</Key>
  </Item>
</Root>

```



# How to: Read and Write an Encoded Document (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

To create an encoded XML document, you add an [XDeclaration](#) to the XML tree, setting the encoding to the desired code page name.

Any value returned by [WebName](#) is a valid value.

If you read an encoded document, the [Encoding](#) property will be set to the code page name.

If you set [Encoding](#) to a valid code page name, LINQ to XML will serialize with the specified encoding.

## Example

The following example creates two documents, one with utf-8 encoding, and one with utf-16 encoding. It then loads the documents and prints the encoding to the console.

```
Console.WriteLine("Creating a document with utf-8 encoding");
XDocument encodedDoc8 = new XDocument(
    new XDeclaration("1.0", "utf-8", "yes"),
    new XElement("Root", "Content")
);
encodedDoc8.Save("EncodedUtf8.xml");
Console.WriteLine("Encoding is:{0}", encodedDoc8.Declaration.Encoding);
Console.WriteLine();

Console.WriteLine("Creating a document with utf-16 encoding");
XDocument encodedDoc16 = new XDocument(
    new XDeclaration("1.0", "utf-16", "yes"),
    new XElement("Root", "Content")
);
encodedDoc16.Save("EncodedUtf16.xml");
Console.WriteLine("Encoding is:{0}", encodedDoc16.Declaration.Encoding);
Console.WriteLine();

XDocument newDoc8 = XDocument.Load("EncodedUtf8.xml");
Console.WriteLine("Encoded document:");
Console.WriteLine(File.ReadAllText("EncodedUtf8.xml"));
Console.WriteLine();
Console.WriteLine("Encoding of loaded document is:{0}", newDoc8.Declaration.Encoding);
Console.WriteLine();

XDocument newDoc16 = XDocument.Load("EncodedUtf16.xml");
Console.WriteLine("Encoded document:");
Console.WriteLine(File.ReadAllText("EncodedUtf16.xml"));
Console.WriteLine();
Console.WriteLine("Encoding of loaded document is:{0}", newDoc16.Declaration.Encoding);
```

This example produces the following output:

```
Creating a document with utf-8 encoding  
Encoding is:utf-8
```

```
Creating a document with utf-16 encoding  
Encoding is:utf-16
```

```
Encoded document:  
<?xml version="1.0" encoding="utf-8" standalone="yes"?>  
<Root>Content</Root>
```

```
Encoding of loaded document is:utf-8
```

```
Encoded document:  
<?xml version="1.0" encoding="utf-16" standalone="yes"?>  
<Root>Content</Root>
```

```
Encoding of loaded document is:utf-16
```

## See also

- [XDeclaration.Encoding](#)

# Using XSLT to Transform an XML Tree (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

You can create an XML tree, create an [XmlReader](#) from the XML tree, create a new document, and create an [XmlWriter](#) that will write into the new document. Then, you can invoke the XSLT transformation, passing the [XmlReader](#) and [XmlWriter](#) to the transformation. After the transformation successfully completes, the new XML tree is populated with the results of the transform.

## Example

```
string xslMarkup = @"<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl='http://www.w3.org/1999/XSL/Transform' version='1.0'>
  <xsl:template match='/Parent'>
    <Root>
      <C1>
        <xsl:value-of select='Child1' />
      </C1>
      <C2>
        <xsl:value-of select='Child2' />
      </C2>
    </Root>
  </xsl:template>
</xsl:stylesheet>";

XDocument xmlTree = new XDocument(
    new XElement("Parent",
        new XElement("Child1", "Child1 data"),
        new XElement("Child2", "Child2 data")
    )
);

XDocument newTree = new XDocument();
using (XmlWriter writer = newTree.CreateWriter()) {
    // Load the style sheet.
    XslCompiledTransform xslt = new XslCompiledTransform();
    xslt.Load(XmlReader.Create(new StringReader(xslMarkup)));

    // Execute the transform and output the results to a writer.
    xslt.Transform(xmlTree.CreateReader(), writer);
}

Console.WriteLine(newTree);
```

This example produces the following output:

```
<Root>
  <C1>Child1 data</C1>
  <C2>Child2 data</C2>
</Root>
```

## See also

- [XContainer.CreateWriter](#)
- [XmlNode.CreateReader](#)

# How to: Use Annotations to Transform LINQ to XML Trees in an XSLT Style (C#)

6/4/2019 • 7 minutes to read • [Edit Online](#)

Annotations can be used to facilitate transforms of an XML tree.

Some XML documents are "document centric with mixed content." With such documents, you don't necessarily know the shape of child nodes of an element. For instance, a node that contains text may look like this:

```
<text>A phrase with <b>bold</b> and <i>italic</i> text.</text>
```

For any given text node, there may be any number of child `<b>` and `<i>` elements. This approach extends to a number of other situations, such as pages that can contain a variety of child elements, such as regular paragraphs, bulleted paragraphs, and bitmaps. Cells in a table may contain text, drop down lists, or bitmaps. One of the primary characteristics of document centric XML is that you do not know which child element any particular element will have.

If you want to transform elements in a tree where you don't necessarily know much about the children of the elements that you want to transform, then this approach that uses annotations is an effective approach.

The summary of the approach is:

- First, annotate elements in the tree with a replacement element.
- Second, iterate through the entire tree, creating a new tree where you replace each element with its annotation. This example implements the iteration and creation of the new tree in a function named `XForm`.

In detail, the approach consists of:

- Execute one or more LINQ to XML queries that return the set of elements that you want to transform from one shape to another. For each element in the query, add a new `XElement` object as an annotation to the element. This new element will replace the annotated element in the new, transformed tree. This is simple code to write, as demonstrated by the example.
- The new element that is added as an annotation can contain new child nodes; it can form a sub-tree with any desired shape.
- There is a special rule: If a child node of the new element is in a different namespace, a namespace that is made up for this purpose (in this example, the namespace is `http://www.microsoft.com/LinqToXmlTransform/2007`), then that child element is not copied to the new tree. Instead, if the namespace is the above mentioned special namespace, and the local name of the element is `ApplyTransforms`, then the child nodes of the element in the source tree are iterated, and copied to the new tree (with the exception that annotated child elements are themselves transformed according to these rules).
- This is somewhat analogous to the specification of transforms in XSL. The query that selects a set of nodes is analogous to the XPath expression for a template. The code to create the new `XElement` that is saved as an annotation is analogous to the sequence constructor in XSL, and the `ApplyTransforms` element is analogous in function to the `xml:apply-templates` element in XSL.
- One advantage to taking this approach - as you formulate queries, you are always writing queries on the unmodified source tree. You need not worry about how modifications to the tree affect the queries that you are writing.

# Transforming a Tree

This first example renames all `Paragraph` nodes to `para`.

```
XNamespace xf = "http://www.microsoft.com/LinqToXmlTransform/2007";
XName at = xf + "ApplyTransforms";

XElement root = XElement.Parse(@"
<Root>
  <Paragraph>This is a sentence with <b>bold</b> and <i>italic</i> text.</Paragraph>
  <Paragraph>More text.</Paragraph>
</Root>");

// replace Paragraph with para
foreach (var el in root.Descendants("Paragraph"))
    el.AddAnnotation(
        new XElement("para",
            // same idea as xsl:apply-templates
            new XElement(xf + "ApplyTransforms")
        )
    );

// The XForm method, shown later in this topic, accomplishes the transform
XElement newRoot = XForm(root);

Console.WriteLine(newRoot);
```

This example produces the following output:

```
<Root>
  <para>This is a sentence with <b>bold</b> and <i>italic</i> text.</para>
  <para>More text.</para>
</Root>
```

## A More Complicated Transform

The following example queries the tree and calculates the average and sum of the `Data` elements, and adds them as new elements to the tree.

```

XNamespace xf = "http://www.microsoft.com/LinqToXmlTransform/2007";
XName at = xf + "ApplyTransforms";

XElement data = new XElement("Root",
    new XElement("Data", 20),
    new XElement("Data", 10),
    new XElement("Data", 3)
);

// while adding annotations, you can query the source tree all you want,
// as the tree is not mutated while annotating.
var avg = data.Elements("Data").Select(z => (Decimal)z).Average();
data.AddAnnotation(
    new XElement("Root",
        new XElement(xf + "ApplyTransforms"),
        new XElement("Average", $"{avg:F4}"),
        new XElement("Sum",
            data
                .Elements("Data")
                .Select(z => (int)z)
                .Sum()
        )
    )
);

Console.WriteLine("Before Transform");
Console.WriteLine("-----");
Console.WriteLine(data);
Console.WriteLine();
Console.WriteLine();

// The XForm method, shown later in this topic, accomplishes the transform
XElement newData = XForm(data);

Console.WriteLine("After Transform");
Console.WriteLine("-----");
Console.WriteLine(newData);

```

This example produces the following output:

```

Before Transform
-----
<Root>
  <Data>20</Data>
  <Data>10</Data>
  <Data>3</Data>
</Root>

After Transform
-----
<Root>
  <Data>20</Data>
  <Data>10</Data>
  <Data>3</Data>
  <Average>11.0000</Average>
  <Sum>33</Sum>
</Root>

```

## Effecting the Transform

A small function, `XForm`, creates a new transformed tree from the original, annotated tree.

- The pseudo code for the function is quite simple:

The function takes an XElement as an argument and returns an XElement.

If an element has an XElement annotation, then

Return a new XElement

The name of the new XElement is the annotation element's name.

All attributes are copied from the annotation to the new node.

All child nodes are copied from the annotation, with the exception that the special node xf:ApplyTransforms is recognized, and the source element's child nodes are iterated. If the source child node is not an XElement, it is copied to the new tree. If the source child is an XElement, then it is transformed by calling this function recursively.

If an element is not annotated

Return a new XElement

The name of the new XElement is the source element's name

All attributes are copied from the source element to the destination's element.

All child nodes are copied from the source element.

If the source child node is not an XElement, it is copied to the new tree. If the source child is an XElement, then it is transformed by calling this function recursively.

Following is the implementation of this function:

```

// Build a transformed XML tree per the annotations
static XElement XForm(XElement source)
{
    XNamespace xf = "http://www.microsoft.com/LinqToXmlTransform/2007";
    XName at = xf + "ApplyTransforms";

    if (source.Annotation<XElement>() != null)
    {
        XElement anno = source.Annotation<XElement>();
        return new XElement(anno.Name,
            anno.Attributes(),
            anno
                .Nodes()
                .Select(
                    (XNode n) =>
                    {
                        XElement annoEl = n as XElement;
                        if (annoEl != null)
                        {
                            if (annoEl.Name == at)
                                return (object)(
                                    source.Nodes()
                                        .Select(
                                            (XNode n2) =>
                                            {
                                                XElement e2 = n2 as XElement;
                                                if (e2 == null)
                                                    return n2;
                                                else
                                                    return XForm(e2);
                                            }
                                        )
                                    );
                            else
                                return n;
                        }
                        else
                            return n;
                    }
                )
            );
    }
    else
    {
        return new XElement(source.Name,
            source.Attributes(),
            source
                .Nodes()
                .Select(n =>
                {
                    XElement el = n as XElement;
                    if (el == null)
                        return n;
                    else
                        return XForm(el);
                }
                )
            );
    }
}

```

## Complete Example

The following code is a complete example that includes the `XForm` function. It includes a few of the typical uses of this type of transform:



```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml;
using System.Xml.Linq;

class Program
{
    static XNamespace xf = "http://www.microsoft.com/LinqToXmlTransform/2007";
    static XName at = xf + "ApplyTransforms";

    // Build a transformed XML tree per the annotations
    static XElement XForm(XElement source)
    {
        if (source.Annotation<XElement>() != null)
        {
            XElement anno = source.Annotation<XElement>();
            return new XElement(anno.Name,
                anno.Attributes(),
                anno
                    .Nodes()
                    .Select(
                        (XNode n) =>
                        {
                            XElement annoEl = n as XElement;
                            if (annoEl != null)
                            {
                                if (annoEl.Name == at)
                                    return (object)(
                                        source.Nodes()
                                            .Select(
                                                (XNode n2) =>
                                                {
                                                    XElement e2 = n2 as XElement;
                                                    if (e2 == null)
                                                        return n2;
                                                    else
                                                        return XForm(e2);
                                                }
                                            )
                                    );
                                else
                                    return n;
                            }
                            else
                                return n;
                        }
                    )
                );
        }
        else
        {
            return new XElement(source.Name,
                source.Attributes(),
                source
                    .Nodes()
                    .Select(n =>
                        {
                            XElement el = n as XElement;
                            if (el == null)
                                return n;
                            else
                                return XForm(el);
                        }
                    )
                );
        }
    }
}

```

```

}

static void Main(string[] args)
{
    XElement root = new XElement("Root",
        new XComment("A comment"),
        new XAttribute("Att1", 123),
        new XElement("Child", 1),
        new XElement("Child", 2),
        new XElement("Other",
            new XElement("GC", 3),
            new XElement("GC", 4)
        ),
        XElement.Parse(
            "<SomeMixedContent>This is <i>an</i> element that " +
            "<b>has</b> some mixed content</SomeMixedContent>"),
        new XElement("AnUnchangedElement", 42)
    );

    // each of the following serves the same semantic purpose as
    // XSLT templates and sequence constructors

    // replace Child with NewChild
    foreach (var el in root.Elements("Child"))
        el.AddAnnotation(new XElement("NewChild", (string)el));

    // replace first GC with GrandChild, add an attribute
    foreach (var el in root.Descendants("GC").Take(1))
        el.AddAnnotation(
            new XElement("GrandChild",
                new XAttribute("ANewAttribute", 999),
                (string)el
            )
        );

    // replace Other with NewOther, add new child elements around original content
    foreach (var el in root.Elements("Other"))
        el.AddAnnotation(
            new XElement("NewOther",
                new XElement("MyNewChild", 1),
                // same idea as xsl:apply-templates
                new XElement(xf + "ApplyTransforms"),
                new XElement("ChildThatComesAfter")
            )
        );

    // change name of element that has mixed content
    root.Descendants("SomeMixedContent").First().AddAnnotation(
        new XElement("MixedContent",
            new XElement(xf + "ApplyTransforms")
        )
    );

    // replace <b> with <Bold>
    foreach (var el in root.Descendants("b"))
        el.AddAnnotation(
            new XElement("Bold",
                new XElement(xf + "ApplyTransforms")
            )
        );

    // replace <i> with <Italic>
    foreach (var el in root.Descendants("i"))
        el.AddAnnotation(
            new XElement("Italic",
                new XElement(xf + "ApplyTransforms")
            )
        );
}

```

```

        Console.WriteLine("Before Transform");
        Console.WriteLine("-----");
        Console.WriteLine(root);
        Console.WriteLine();
        Console.WriteLine();
        XElement newRoot = XForm(root);

        Console.WriteLine("After Transform");
        Console.WriteLine("-----");
        Console.WriteLine(newRoot);
    }
}

```

This example produces the following output:

```

Before Transform
-----
<Root Att1="123">
  <!--A comment-->
  <Child>1</Child>
  <Child>2</Child>
  <Other>
    <GC>3</GC>
    <GC>4</GC>
  </Other>
  <SomeMixedContent>This is <i>an</i> element that <b>has</b> some mixed content</SomeMixedContent>
  <AnUnchangedElement>42</AnUnchangedElement>
</Root>

After Transform
-----
<Root Att1="123">
  <!--A comment-->
  <NewChild>1</NewChild>
  <NewChild>2</NewChild>
  <NewOther>
    <MyNewChild>1</MyNewChild>
    <GrandChild ANewAttribute="999">3</GrandChild>
    <GC>4</GC>
    <ChildThatComesAfter />
  </NewOther>
  <MixedContent>This is <Italic>an</Italic> element that <Bold>has</Bold> some mixed content</MixedContent>
  <AnUnchangedElement>42</AnUnchangedElement>
</Root>

```

# How to: Serialize Using XmlSerializer (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic shows an example that serializes and deserializes using [XmlSerializer](#).

## Example

The following example creates a number of objects that contain [XElement](#) objects. It then serializes them to a memory stream, and then deserializes them from the memory stream.

```

using System;
using System.IO;
using System.Linq;
using System.Xml;
using System.Xml.Serialization;
using System.Xml.Linq;

public class XElementContainer
{
    public XElement member;

    public XElementContainer()
    {
        member = XLinqTest.CreateXElement();
    }

    public override string ToString()
    {
        return member.ToString();
    }
}

public class XElementNullContainer
{
    public XElement member;

    public XElementNullContainer()
    {
    }
}

class XLinqTest
{
    static void Main(string[] args)
    {
        Test<XElementNullContainer>(new XElementNullContainer());
        Test<XElement>(CreateXElement());
        Test<XElementContainer>(new XElementContainer());
    }

    public static XElement CreateXElement()
    {
        XNamespace ns = "http://www.adventure-works.com";
        return new XElement(ns + "aw");
    }

    static void Test<T>(T obj)
    {
        using (MemoryStream stream = new MemoryStream())
        {
            XmlSerializer s = new XmlSerializer(typeof(T));
            Console.WriteLine("Testing for type: {0}", typeof(T));
            s.Serialize(XmlWriter.Create(stream), obj);
            stream.Flush();
            stream.Seek(0, SeekOrigin.Begin);
            object o = s.Deserialize(XmlReader.Create(stream));
            Console.WriteLine(" Deserialized type: {0}", o.GetType());
        }
    }
}

```

This example produces the following output:

```
Testing for type: XElementNullContainer
  Deserialized type: XElementNullContainer
Testing for type: System.Xml.Linq.XElement
  Deserialized type: System.Xml.Linq.XElement
Testing for type: XElementContainer
  Deserialized type: XElementContainer
```

# How to: Serialize Using DataContractSerializer (C#)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic shows an example that serializes and deserializes using [DataContractSerializer](#).

## Example

The following example creates a number of objects that contain [XElement](#) objects. It then serializes them to text files, and then deserializes them from the text files.

```

using System;
using System.Xml;
using System.Xml.Linq;
using System.IO;
using System.Runtime.Serialization;

public class XLinqTest
{
    public static void Main()
    {
        Test<XElement>(CreateXElement());
        Test<XElementContainer>(new XElementContainer());
        Test<XElementNullContainer>(new XElementNullContainer());
    }

    public static void Test<T>(T obj)
    {
        DataContractSerializer s = new DataContractSerializer(typeof(T));
        using (FileStream fs = File.Open("test" + typeof(T).Name + ".xml", FileMode.Create))
        {
            Console.WriteLine("Testing for type: {0}", typeof(T));
            s.WriteObject(fs, obj);
        }
        using (FileStream fs = File.Open("test" + typeof(T).Name + ".xml", FileMode.Open))
        {
            object s2 = s.ReadObject(fs);
            if (s2 == null)
                Console.WriteLine(" Deserialized object is null (Nothing in VB)");
            else
                Console.WriteLine(" Deserialized type: {0}", s2.GetType());
        }
    }

    public static XElement CreateXElement()
    {
        return new XElement(XName.Get("NameInNamespace", "http://www.adventure-works.org"));
    }
}

[DataContract]
public class XElementContainer
{
    [DataMember]
    public XElement member;

    public XElementContainer()
    {
        member = XLinqTest.CreateXElement();
    }
}

[DataContract]
public class XElementNullContainer
{
    [DataMember]
    public XElement member;

    public XElementNullContainer()
    {
        member = null;
    }
}

```

This example produces the following output:



```
Testing for type: System.Xml.Linq.XElement
  Deserialized type: System.Xml.Linq.XElement
Testing for type: XElementContainer
  Deserialized type: XElementContainer
Testing for type: XElementNullContainer
  Deserialized type: XElementNullContainer
```

# LINQ to XML Security (C#)

4/28/2019 • 7 minutes to read • [Edit Online](#)

This topic describes security issues associated with LINQ to XML. In addition, it provides some guidance for mitigating security exposure.

## LINQ to XML Security Overview

LINQ to XML is designed more for programming convenience than for server-side applications with stringent security requirements. Most XML scenarios consist of processing trusted XML documents, rather than processing untrusted XML documents that are uploaded to a server. LINQ to XML is optimized for these scenarios.

If you must process untrusted data from unknown sources, Microsoft recommends that you use an instance of the [XmlReader](#) class that has been configured to filter out known XML denial of service (DoS) attacks.

If you have configured an [XmlReader](#) to mitigate denial of service attacks, you can use that reader to populate a LINQ to XML tree and still benefit from the programmer productivity enhancements of LINQ to XML. Many mitigation techniques involve creating readers that are configured to mitigate the security issue, and then instantiating an XML tree through the configured reader.

XML is intrinsically vulnerable to denial of service attacks because documents are unbounded in size, depth, element name size, and more. Regardless of the component that you use to process XML, you should always be prepared to recycle the application domain if it uses excessive resources.

## Mitigation of XML, XSD, XPath, and XSLT Attacks

LINQ to XML is built upon [XmlReader](#) and [XmlWriter](#). LINQ to XML supports XSD and XPath through extension methods in the [System.Xml.Schema](#) and [System.Xml.XPath](#) namespaces. Using the [XmlReader](#), [XPathNavigator](#), and [XmlWriter](#) classes in conjunction with LINQ to XML, you can invoke XSLT to transform XML trees.

If you are operating in a less secure environment, there are a number of security issues that are associated with XML and the use of the classes in [System.Xml](#), [System.Xml.Schema](#), [System.Xml.XPath](#), and [System.Xml.Xsl](#). These issues include, but are not limited to, the following:

- XSD, XPath, and XSLT are string-based languages in which you can specify operations that consume a lot of time or memory. It is the responsibility of application programmers who take XSD, XPath, or XSLT strings from untrusted sources to validate that the strings are not malicious, or to monitor and mitigate the possibility that evaluating these strings will lead to excessive system resource consumption.
- XSD schemas (including inline schemas) are inherently vulnerable to denial of service attacks; you should not accept schemas from untrusted sources.
- XSD and XSLT can include references to other files, and such references can result in cross-zone and cross-domain attacks.
- External entities in DTDs can result in cross-zone and cross-domain attacks.
- DTDs are vulnerable to denial of service attacks.
- Exceptionally deep XML documents can pose denial of service issues; you might want to limit the depth of XML documents.
- Do not accept supporting components, such as [NameTable](#), [XmlNamespaceManager](#), and [XmlResolver](#) objects, from untrusted assemblies.

- Read data in chunks to mitigate large document attacks.
- Script blocks in XSLT style sheets can expose a number of attacks.
- Validate carefully before constructing dynamic XPath expressions.

## LINQ to XML Security Issues

The security issues in this topic are not presented in any particular order. All issues are important and should be addressed as appropriate.

A successful elevation of privilege attack gives a malicious assembly more control over its environment. A successful elevation of privilege attack can result in disclosure of data, denial of service, and more.

Applications should not disclose data to users who are not authorized to see that data.

Denial of service attacks cause the XML parser or LINQ to XML to consume excessive amounts of memory or CPU time. Denial of service attacks are considered to be less severe than elevation of privilege attacks or disclosure of data attacks. However, they are important in a scenario where a server needs to process XML documents from untrusted sources.

### Exceptions and Error Messages Might Reveal Data

The description of an error might reveal data, such as the data being transformed, file names, or implementation details. Error messages should not be exposed to callers that are not trusted. You should catch all errors and report errors with your own custom error messages.

### Do Not Call `CodeAccessPermissions.Assert` in an Event Handler

An assembly can have lesser or greater permissions. An assembly that has greater permissions has greater control over the computer and its environments.

If code in an assembly with greater permissions calls [CodeAccessPermission.Assert](#) in an event handler, and then the XML tree is passed to a malicious assembly that has restricted permissions, the malicious assembly can cause an event to be raised. Because the event runs code that is in the assembly with greater permissions, the malicious assembly would then be operating with elevated privileges.

Microsoft recommends that you never call [CodeAccessPermission.Assert](#) in an event handler.

### DTDs are Not Secure

Entities in DTDs are inherently not secure. It is possible for a malicious XML document that contains a DTD to cause the parser to use all memory and CPU time, causing a denial of service attack. Therefore, in LINQ to XML, DTD processing is turned off by default. You should not accept DTDs from untrusted sources.

One example of accepting DTDs from untrusted sources is a Web application that allows Web users to upload an XML file that references a DTD and a DTD file. Upon validation of the file, a malicious DTD could execute a denial of service attack on your server. Another example of accepting DTDs from untrusted sources is to reference a DTD on a network share that also allows anonymous FTP access.

### Avoid Excessive Buffer Allocation

Application developers should be aware that extremely large data sources can lead to resource exhaustion and denial of service attacks.

If a malicious user submits or uploads a very large XML document, it could cause LINQ to XML to consume excessive system resources. This can constitute a denial of service attack. To prevent this, you can set the [XmlReaderSettings.MaxCharactersInDocument](#) property, and create a reader that is then limited in the size of document that it can load. You then use the reader to create the XML tree.

For example, if you know that the maximum expected size of your XML documents coming from an untrusted

source will be less than 50K bytes, set [XmlReaderSettings.MaxCharactersInDocument](#) to 100,000. This will not encumber your processing of XML documents, and at the same time it will mitigate denial of service threats where documents might be uploaded that would consume large amounts of memory.

### **Avoid Excess Entity Expansion**

One of the known denial of service attacks when using a DTD is a document that causes excessive entity expansion. To prevent this, you can set the [XmlReaderSettings.MaxCharactersFromEntities](#) property, and create a reader that is then limited in the number of characters that result from entity expansion. You then use the reader to create the XML tree.

### **Limit the Depth of the XML Hierarchy**

One possible denial of service attack is when a document is submitted that has excessive depth of hierarchy. To prevent this, you can wrap a [XmlReader](#) in your own class that counts the depth of elements. If the depth exceeds a predetermined reasonable level, you can terminate the processing of the malicious document.

### **Protect Against Untrusted XmlReader or XmlWriter Implementations**

Administrators should verify that any externally supplied [XmlReader](#) or [XmlWriter](#) implementations have strong names and have been registered in the machine configuration. This prevents malicious code masquerading as a reader or writer from being loaded.

### **Periodically Free Objects that Reference XName**

To protect against certain kinds of attacks, application programmers should free all objects that reference an [XName](#) object in the application domain on a regular basis.

### **Protect Against Random XML Names**

Applications that take data from untrusted sources should consider using an [XmlReader](#) that is wrapped in custom code to inspect for the possibility of random XML names and namespaces. If such random XML names and namespaces are detected, the application can then terminate the processing of the malicious document.

You might want to limit the number of names in any given namespace (including names in no namespace) to a reasonable limit.

### **Annotations Are Accessible by Software Components that Share a LINQ to XML Tree**

LINQ to XML could be used to build processing pipelines in which different application components load, validate, query, transform, update, and save XML data that is passed between components as XML trees. This can help optimize performance, because the overhead of loading and serializing objects to XML text is done only at the ends of the pipeline. Developers must be aware, however, that all annotations and event handlers created by one component are accessible to other components. This can create a number of vulnerabilities if the components have different levels of trust. To build secure pipelines across less trusted components, you must serialize LINQ to XML objects to XML text before passing the data to an untrusted component.

Some security is provided by the common language runtime (CLR). For example, a component that does not include a private class cannot access annotations keyed by that class. However, annotations can be deleted by components that cannot read them. This could be used as a tampering attack.

## **See also**

- [Programming Guide \(LINQ to XML\) \(C#\)](#)

# Sample XML File: Typical Purchase Order (LINQ to XML)

6/4/2019 • 2 minutes to read • [Edit Online](#)

The following XML file is used in various examples in the LINQ to XML documentation. This file is a typical purchase order.

## PurchaseOrder.xml

```
<?xml version="1.0"?>
<PurchaseOrder PurchaseOrderNumber="99503" OrderDate="1999-10-20">
  <Address Type="Shipping">
    <Name>Ellen Adams</Name>
    <Street>123 Maple Street</Street>
    <City>Mill Valley</City>
    <State>CA</State>
    <Zip>10999</Zip>
    <Country>USA</Country>
  </Address>
  <Address Type="Billing">
    <Name>Tai Yee</Name>
    <Street>8 Oak Avenue</Street>
    <City>Old Town</City>
    <State>PA</State>
    <Zip>95819</Zip>
    <Country>USA</Country>
  </Address>
  <DeliveryNotes>Please leave packages in shed by driveway.</DeliveryNotes>
  <Items>
    <Item PartNumber="872-AA">
      <ProductName>Lawnmower</ProductName>
      <Quantity>1</Quantity>
      <USPrice>148.95</USPrice>
      <Comment>Confirm this is electric</Comment>
    </Item>
    <Item PartNumber="926-AA">
      <ProductName>Baby Monitor</ProductName>
      <Quantity>2</Quantity>
      <USPrice>39.98</USPrice>
      <ShipDate>1999-05-21</ShipDate>
    </Item>
  </Items>
</PurchaseOrder>
```

# Sample XML File: Typical Purchase Order in a Namespace

6/4/2019 • 2 minutes to read • [Edit Online](#)

The following XML file is used in various examples in the LINQ to XML documentation. This file is a typical purchase order. The XML is in a namespace.

## PurchaseOrderInNamespace.xml

```
<?xml version="1.0"?>
<aw:PurchaseOrder
  aw:PurchaseOrderNumber="99503"
  aw:OrderDate="1999-10-20"
  xmlns:aw="http://www.adventure-works.com">
  <aw:Address aw:Type="Shipping">
    <aw:Name>Ellen Adams</aw:Name>
    <aw:Street>123 Maple Street</aw:Street>
    <aw:City>Mill Valley</aw:City>
    <aw:State>CA</aw:State>
    <aw:Zip>10999</aw:Zip>
    <aw:Country>USA</aw:Country>
  </aw:Address>
  <aw:Address aw:Type="Billing">
    <aw:Name>Tai Yee</aw:Name>
    <aw:Street>8 Oak Avenue</aw:Street>
    <aw:City>Old Town</aw:City>
    <aw:State>PA</aw:State>
    <aw:Zip>95819</aw:Zip>
    <aw:Country>USA</aw:Country>
  </aw:Address>
  <aw:DeliveryNotes>Please leave packages in shed by driveway.</aw:DeliveryNotes>
  <aw:Items>
    <aw:Item aw:PartNumber="872-AA">
      <aw:ProductName>Lawnmower</aw:ProductName>
      <aw:Quantity>1</aw:Quantity>
      <aw:USPrice>148.95</aw:USPrice>
      <aw:Comment>Confirm this is electric</aw:Comment>
    </aw:Item>
    <aw:Item aw:PartNumber="926-AA">
      <aw:ProductName>Baby Monitor</aw:ProductName>
      <aw:Quantity>2</aw:Quantity>
      <aw:USPrice>39.98</aw:USPrice>
      <aw:ShipDate>1999-05-21</aw:ShipDate>
    </aw:Item>
  </aw:Items>
</aw:PurchaseOrder>
```

# Sample XML File: Multiple Purchase Orders (LINQ to XML)

6/4/2019 • 2 minutes to read • [Edit Online](#)

The following XML file is used in various examples in the LINQ to XML documentation. This file contains several purchase orders.

## PurchaseOrders.xml

```
<?xml version="1.0"?>
<PurchaseOrders>
  <PurchaseOrder PurchaseOrderNumber="99503" OrderDate="1999-10-20">
    <Address Type="Shipping">
      <Name>Ellen Adams</Name>
      <Street>123 Maple Street</Street>
      <City>Mill Valley</City>
      <State>CA</State>
      <Zip>10999</Zip>
      <Country>USA</Country>
    </Address>
    <Address Type="Billing">
      <Name>Tai Yee</Name>
      <Street>8 Oak Avenue</Street>
      <City>Old Town</City>
      <State>PA</State>
      <Zip>95819</Zip>
      <Country>USA</Country>
    </Address>
    <DeliveryNotes>Please leave packages in shed by driveway.</DeliveryNotes>
    <Items>
      <Item PartNumber="872-AA">
        <ProductName>Lawnmower</ProductName>
        <Quantity>1</Quantity>
        <USPrice>148.95</USPrice>
        <Comment>Confirm this is electric</Comment>
      </Item>
      <Item PartNumber="926-AA">
        <ProductName>Baby Monitor</ProductName>
        <Quantity>2</Quantity>
        <USPrice>39.98</USPrice>
        <ShipDate>1999-05-21</ShipDate>
      </Item>
    </Items>
  </PurchaseOrder>
  <PurchaseOrder PurchaseOrderNumber="99505" OrderDate="1999-10-22">
    <Address Type="Shipping">
      <Name>Cristian Osorio</Name>
      <Street>456 Main Street</Street>
      <City>Buffalo</City>
      <State>NY</State>
      <Zip>98112</Zip>
      <Country>USA</Country>
    </Address>
    <Address Type="Billing">
      <Name>Cristian Osorio</Name>
      <Street>456 Main Street</Street>
      <City>Buffalo</City>
      <State>NY</State>
      <Zip>98112</Zip>
      <Country>USA</Country>
    </Address>
```

```
<Country>USA</Country>
</Address>
<DeliveryNotes>Please notify me before shipping.</DeliveryNotes>
<Items>
  <Item PartNumber="456-NM">
    <ProductName>Power Supply</ProductName>
    <Quantity>1</Quantity>
    <USPrice>45.99</USPrice>
  </Item>
</Items>
</PurchaseOrder>
<PurchaseOrder PurchaseOrderNumber="99504" OrderDate="1999-10-22">
  <Address Type="Shipping">
    <Name>Jessica Arnold</Name>
    <Street>4055 Madison Ave</Street>
    <City>Seattle</City>
    <State>WA</State>
    <Zip>98112</Zip>
    <Country>USA</Country>
  </Address>
  <Address Type="Billing">
    <Name>Jessica Arnold</Name>
    <Street>4055 Madison Ave</Street>
    <City>Buffalo</City>
    <State>NY</State>
    <Zip>98112</Zip>
    <Country>USA</Country>
  </Address>
  <Items>
    <Item PartNumber="898-AZ">
      <ProductName>Computer Keyboard</ProductName>
      <Quantity>1</Quantity>
      <USPrice>29.99</USPrice>
    </Item>
    <Item PartNumber="898-AM">
      <ProductName>Wireless Mouse</ProductName>
      <Quantity>1</Quantity>
      <USPrice>14.99</USPrice>
    </Item>
  </Items>
</PurchaseOrder>
</PurchaseOrders>
```



# Sample XML File: Multiple Purchase Orders in a Namespace

6/4/2019 • 2 minutes to read • [Edit Online](#)

The following XML file is used in various examples in the LINQ to XML documentation. This file contains several purchase orders. The XML is in a namespace.

## PurchaseOrdersInNamespace.xml

```
<?xml version="1.0" encoding="utf-8"?>
<aw:PurchaseOrders xmlns:aw="http://www.adventure-works.com">
  <aw:PurchaseOrder aw:PurchaseOrderNumber="99503" aw:OrderDate="1999-10-20">
    <aw:Address aw:Type="Shipping">
      <aw:Name>Ellen Adams</aw:Name>
      <aw:Street>123 Maple Street</aw:Street>
      <aw:City>Mill Valley</aw:City>
      <aw:State>CA</aw:State>
      <aw:Zip>10999</aw:Zip>
      <aw:Country>USA</aw:Country>
    </aw:Address>
    <aw:Address aw:Type="Billing">
      <aw:Name>Tai Yee</aw:Name>
      <aw:Street>8 Oak Avenue</aw:Street>
      <aw:City>Old Town</aw:City>
      <aw:State>PA</aw:State>
      <aw:Zip>95819</aw:Zip>
      <aw:Country>USA</aw:Country>
    </aw:Address>
    <aw:DeliveryNotes>Please leave packages in shed by driveway.</aw:DeliveryNotes>
    <aw:Items>
      <aw:Item aw:PartNumber="872-AA">
        <aw:ProductName>Lawnmower</aw:ProductName>
        <aw:Quantity>1</aw:Quantity>
        <aw:USPrice>148.95</aw:USPrice>
        <aw:Comment>Confirm this is electric</aw:Comment>
      </aw:Item>
      <aw:Item aw:PartNumber="926-AA">
        <aw:ProductName>Baby Monitor</aw:ProductName>
        <aw:Quantity>2</aw:Quantity>
        <aw:USPrice>39.98</aw:USPrice>
        <aw:ShipDate>1999-05-21</aw:ShipDate>
      </aw:Item>
    </aw:Items>
  </aw:PurchaseOrder>
  <aw:PurchaseOrder aw:PurchaseOrderNumber="99505" aw:OrderDate="1999-10-22">
    <aw:Address aw:Type="Shipping">
      <aw:Name>Cristian Osorio</aw:Name>
      <aw:Street>456 Main Street</aw:Street>
      <aw:City>Buffalo</aw:City>
      <aw:State>NY</aw:State>
      <aw:Zip>98112</aw:Zip>
      <aw:Country>USA</aw:Country>
    </aw:Address>
    <aw:Address aw:Type="Billing">
      <aw:Name>Cristian Osorio</aw:Name>
      <aw:Street>456 Main Street</aw:Street>
      <aw:City>Buffalo</aw:City>
      <aw:State>NY</aw:State>
      <aw:Zip>98112</aw:Zip>
      <aw:Country>USA</aw:Country>
    </aw:Address>
```

```
<aw:Country>USA</aw:Country>
</aw:Address>
<aw:DeliveryNotes>Please notify me before shipping.</aw:DeliveryNotes>
<aw:Items>
  <aw:Item aw:PartNumber="456-NM">
    <aw:ProductName>Power Supply</aw:ProductName>
    <aw:Quantity>1</aw:Quantity>
    <aw:USPrice>45.99</aw:USPrice>
  </aw:Item>
</aw:Items>
</aw:PurchaseOrder>
<aw:PurchaseOrder aw:PurchaseOrderNumber="99504" aw:OrderDate="1999-10-22">
  <aw:Address aw:Type="Shipping">
    <aw:Name>Jessica Arnold</aw:Name>
    <aw:Street>4055 Madison Ave</aw:Street>
    <aw:City>Seattle</aw:City>
    <aw:State>WA</aw:State>
    <aw:Zip>98112</aw:Zip>
    <aw:Country>USA</aw:Country>
  </aw:Address>
  <aw:Address aw:Type="Billing">
    <aw:Name>Jessica Arnold</aw:Name>
    <aw:Street>4055 Madison Ave</aw:Street>
    <aw:City>Buffalo</aw:City>
    <aw:State>NY</aw:State>
    <aw:Zip>98112</aw:Zip>
    <aw:Country>USA</aw:Country>
  </aw:Address>
  <aw:Items>
    <aw:Item aw:PartNumber="898-AZ">
      <aw:ProductName>Computer Keyboard</aw:ProductName>
      <aw:Quantity>1</aw:Quantity>
      <aw:USPrice>29.99</aw:USPrice>
    </aw:Item>
    <aw:Item aw:PartNumber="898-AM">
      <aw:ProductName>Wireless Mouse</aw:ProductName>
      <aw:Quantity>1</aw:Quantity>
      <aw:USPrice>14.99</aw:USPrice>
    </aw:Item>
  </aw:Items>
</aw:PurchaseOrder>
</aw:PurchaseOrders>
```

# Sample XML File: Test Configuration (LINQ to XML)

6/4/2019 • 2 minutes to read • [Edit Online](#)

The following XML file is used in various examples in the LINQ to XML documentation. This is a test configuration file.

## TestConfig.xml

```
<?xml version="1.0"?>
<Tests>
  <Test TestId="0001" TestType="CMD">
    <Name>Convert number to string</Name>
    <CommandLine>Examp1.EXE</CommandLine>
    <Input>1</Input>
    <Output>One</Output>
  </Test>
  <Test TestId="0002" TestType="CMD">
    <Name>Find succeeding characters</Name>
    <CommandLine>Examp2.EXE</CommandLine>
    <Input>abc</Input>
    <Output>def</Output>
  </Test>
  <Test TestId="0003" TestType="GUI">
    <Name>Convert multiple numbers to strings</Name>
    <CommandLine>Examp2.EXE /Verbose</CommandLine>
    <Input>123</Input>
    <Output>One Two Three</Output>
  </Test>
  <Test TestId="0004" TestType="GUI">
    <Name>Find correlated key</Name>
    <CommandLine>Examp3.EXE</CommandLine>
    <Input>a1</Input>
    <Output>b1</Output>
  </Test>
  <Test TestId="0005" TestType="GUI">
    <Name>Count characters</Name>
    <CommandLine>FinalExamp.EXE</CommandLine>
    <Input>This is a test</Input>
    <Output>14</Output>
  </Test>
  <Test TestId="0006" TestType="GUI">
    <Name>Another Test</Name>
    <CommandLine>Examp2.EXE</CommandLine>
    <Input>Test Input</Input>
    <Output>10</Output>
  </Test>
</Tests>
```

# Sample XML File: Test Configuration in a Namespace

6/4/2019 • 2 minutes to read • [Edit Online](#)

The following XML file is used in various examples in the LINQ to XML documentation. This is a test configuration file. The XML is in a namespace.

## TestConfigInNamespace.xml

```
<?xml version="1.0"?>
<Tests xmlns="http://www.adatum.com">
  <Test TestId="0001" TestType="CMD">
    <Name>Convert number to string</Name>
    <CommandLine>Examp1.EXE</CommandLine>
    <Input>1</Input>
    <Output>One</Output>
  </Test>
  <Test TestId="0002" TestType="CMD">
    <Name>Find succeeding characters</Name>
    <CommandLine>Examp2.EXE</CommandLine>
    <Input>abc</Input>
    <Output>def</Output>
  </Test>
  <Test TestId="0003" TestType="GUI">
    <Name>Convert multiple numbers to strings</Name>
    <CommandLine>Examp2.EXE /Verbose</CommandLine>
    <Input>123</Input>
    <Output>One Two Three</Output>
  </Test>
  <Test TestId="0004" TestType="GUI">
    <Name>Find correlated key</Name>
    <CommandLine>Examp3.EXE</CommandLine>
    <Input>a1</Input>
    <Output>b1</Output>
  </Test>
  <Test TestId="0005" TestType="GUI">
    <Name>Count characters</Name>
    <CommandLine>FinalExamp.EXE</CommandLine>
    <Input>This is a test</Input>
    <Output>14</Output>
  </Test>
  <Test TestId="0006" TestType="GUI">
    <Name>Another Test</Name>
    <CommandLine>Examp2.EXE</CommandLine>
    <Input>Test Input</Input>
    <Output>10</Output>
  </Test>
</Tests>
```

# Sample XML File: Customers and Orders (LINQ to XML)

6/4/2019 • 3 minutes to read • [Edit Online](#)

The following XML file is used in various examples in the LINQ to XML documentation. This file contains customers and orders.

The topic [Sample XSD File: Customers and Orders](#) contains an XSD that can be used to validate this document. It uses the `xs:key` and `xs:keyref` features of XSD to establish that the `CustomerID` attribute of the `Customer` element is a key, and to establish a relationship between the `CustomerID` element in each `Order` element and the `CustomerID` attribute in each `Customer` element.

For an example of writing LINQ queries that take advantage of this relationship using the `Join` clause, see [How to: Join Two Collections \(LINQ to XML\) \(C#\)](#).

## CustomersOrders.xml

```
<?xml version="1.0" encoding="utf-8"?>
<Root>
  <Customers>
    <Customer CustomerID="GREAL">
      <CompanyName>Great Lakes Food Market</CompanyName>
      <ContactName>Howard Snyder</ContactName>
      <ContactTitle>Marketing Manager</ContactTitle>
      <Phone>(503) 555-7555</Phone>
      <FullAddress>
        <Address>2732 Baker Blvd.</Address>
        <City>Eugene</City>
        <Region>OR</Region>
        <PostalCode>97403</PostalCode>
        <Country>USA</Country>
      </FullAddress>
    </Customer>
    <Customer CustomerID="HUNGC">
      <CompanyName>Hungry Coyote Import Store</CompanyName>
      <ContactName>Yoshi Latimer</ContactName>
      <ContactTitle>Sales Representative</ContactTitle>
      <Phone>(503) 555-6874</Phone>
      <Fax>(503) 555-2376</Fax>
      <FullAddress>
        <Address>City Center Plaza 516 Main St.</Address>
        <City>Elgin</City>
        <Region>OR</Region>
        <PostalCode>97827</PostalCode>
        <Country>USA</Country>
      </FullAddress>
    </Customer>
    <Customer CustomerID="LAZYK">
      <CompanyName>Lazy K Kountry Store</CompanyName>
      <ContactName>John Steel</ContactName>
      <ContactTitle>Marketing Manager</ContactTitle>
      <Phone>(509) 555-7969</Phone>
      <Fax>(509) 555-6221</Fax>
      <FullAddress>
        <Address>12 Orchestra Terrace</Address>
        <City>Walla Walla</City>
        <Region>WA</Region>
        <PostalCode>99362</PostalCode>
      </FullAddress>
    </Customer>
  </Customers>

```

```
<Country>USA</Country>
</FullAddress>
</Customer>
<Customer CustomerID="LETSS">
  <CompanyName>Let's Stop N Shop</CompanyName>
  <ContactName>Jaime Yorres</ContactName>
  <ContactTitle>Owner</ContactTitle>
  <Phone>(415) 555-5938</Phone>
  <FullAddress>
    <Address>87 Polk St. Suite 5</Address>
    <City>San Francisco</City>
    <Region>CA</Region>
    <PostalCode>94117</PostalCode>
    <Country>USA</Country>
  </FullAddress>
</Customer>
</Customers>
<Orders>
  <Order>
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>6</EmployeeID>
    <OrderDate>1997-05-06T00:00:00</OrderDate>
    <RequiredDate>1997-05-20T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-05-09T00:00:00">
      <ShipVia>2</ShipVia>
      <Freight>3.35</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>8</EmployeeID>
    <OrderDate>1997-07-04T00:00:00</OrderDate>
    <RequiredDate>1997-08-01T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-07-14T00:00:00">
      <ShipVia>2</ShipVia>
      <Freight>4.42</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>1</EmployeeID>
    <OrderDate>1997-07-31T00:00:00</OrderDate>
    <RequiredDate>1997-08-28T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-08-05T00:00:00">
      <ShipVia>2</ShipVia>
      <Freight>116.53</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>4</EmployeeID>
```

```
<OrderDate>1997-07-31T00:00:00</OrderDate>
<RequiredDate>1997-08-28T00:00:00</RequiredDate>
<ShipInfo ShippedDate="1997-08-04T00:00:00">
  <ShipVia>2</ShipVia>
  <Freight>18.53</Freight>
  <ShipName>Great Lakes Food Market</ShipName>
  <ShipAddress>2732 Baker Blvd.</ShipAddress>
  <ShipCity>Eugene</ShipCity>
  <ShipRegion>OR</ShipRegion>
  <ShipPostalCode>97403</ShipPostalCode>
  <ShipCountry>USA</ShipCountry>
</ShipInfo>
</Order>
<Order>
  <CustomerID>GREAL</CustomerID>
  <EmployeeID>6</EmployeeID>
  <OrderDate>1997-09-04T00:00:00</OrderDate>
  <RequiredDate>1997-10-02T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1997-09-10T00:00:00">
    <ShipVia>1</ShipVia>
    <Freight>57.15</Freight>
    <ShipName>Great Lakes Food Market</ShipName>
    <ShipAddress>2732 Baker Blvd.</ShipAddress>
    <ShipCity>Eugene</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97403</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>GREAL</CustomerID>
  <EmployeeID>3</EmployeeID>
  <OrderDate>1997-09-25T00:00:00</OrderDate>
  <RequiredDate>1997-10-23T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1997-09-30T00:00:00">
    <ShipVia>3</ShipVia>
    <Freight>76.13</Freight>
    <ShipName>Great Lakes Food Market</ShipName>
    <ShipAddress>2732 Baker Blvd.</ShipAddress>
    <ShipCity>Eugene</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97403</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>GREAL</CustomerID>
  <EmployeeID>4</EmployeeID>
  <OrderDate>1998-01-06T00:00:00</OrderDate>
  <RequiredDate>1998-02-03T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1998-02-04T00:00:00">
    <ShipVia>2</ShipVia>
    <Freight>719.78</Freight>
    <ShipName>Great Lakes Food Market</ShipName>
    <ShipAddress>2732 Baker Blvd.</ShipAddress>
    <ShipCity>Eugene</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97403</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>GREAL</CustomerID>
  <EmployeeID>3</EmployeeID>
  <OrderDate>1998-03-09T00:00:00</OrderDate>
  <RequiredDate>1998-04-06T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1998-03-18T00:00:00">
    <ShipVia>2</ShipVia>
    <Freight>33.68</Freight>
```

```

    <ShipName>Great Lakes Food Market</ShipName>
    <ShipAddress>2732 Baker Blvd.</ShipAddress>
    <ShipCity>Eugene</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97403</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>GREAL</CustomerID>
  <EmployeeID>3</EmployeeID>
  <OrderDate>1998-04-07T00:00:00</OrderDate>
  <RequiredDate>1998-05-05T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1998-04-15T00:00:00">
    <ShipVia>2</ShipVia>
    <Freight>25.19</Freight>
    <ShipName>Great Lakes Food Market</ShipName>
    <ShipAddress>2732 Baker Blvd.</ShipAddress>
    <ShipCity>Eugene</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97403</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>GREAL</CustomerID>
  <EmployeeID>4</EmployeeID>
  <OrderDate>1998-04-22T00:00:00</OrderDate>
  <RequiredDate>1998-05-20T00:00:00</RequiredDate>
  <ShipInfo>
    <ShipVia>3</ShipVia>
    <Freight>18.84</Freight>
    <ShipName>Great Lakes Food Market</ShipName>
    <ShipAddress>2732 Baker Blvd.</ShipAddress>
    <ShipCity>Eugene</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97403</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>GREAL</CustomerID>
  <EmployeeID>4</EmployeeID>
  <OrderDate>1998-04-30T00:00:00</OrderDate>
  <RequiredDate>1998-06-11T00:00:00</RequiredDate>
  <ShipInfo>
    <ShipVia>3</ShipVia>
    <Freight>14.01</Freight>
    <ShipName>Great Lakes Food Market</ShipName>
    <ShipAddress>2732 Baker Blvd.</ShipAddress>
    <ShipCity>Eugene</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97403</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>HUNGC</CustomerID>
  <EmployeeID>3</EmployeeID>
  <OrderDate>1996-12-06T00:00:00</OrderDate>
  <RequiredDate>1997-01-03T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1996-12-09T00:00:00">
    <ShipVia>2</ShipVia>
    <Freight>20.12</Freight>
    <ShipName>Hungry Coyote Import Store</ShipName>
    <ShipAddress>City Center Plaza 516 Main St.</ShipAddress>
    <ShipCity>Elgin</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97827</ShipPostalCode>

```



```
<ShipPostalCode>97827</ShipPostalCode>
<ShipCountry>USA</ShipCountry>
</ShipInfo>
</Order>
<Order>
  <CustomerID>HUNGC</CustomerID>
  <EmployeeID>1</EmployeeID>
  <OrderDate>1996-12-25T00:00:00</OrderDate>
  <RequiredDate>1997-01-22T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1997-01-03T00:00:00">
    <ShipVia>3</ShipVia>
    <Freight>30.34</Freight>
    <ShipName>Hungry Coyote Import Store</ShipName>
    <ShipAddress>City Center Plaza 516 Main St.</ShipAddress>
    <ShipCity>Elgin</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97827</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>HUNGC</CustomerID>
  <EmployeeID>3</EmployeeID>
  <OrderDate>1997-01-15T00:00:00</OrderDate>
  <RequiredDate>1997-02-12T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1997-01-24T00:00:00">
    <ShipVia>1</ShipVia>
    <Freight>0.2</Freight>
    <ShipName>Hungry Coyote Import Store</ShipName>
    <ShipAddress>City Center Plaza 516 Main St.</ShipAddress>
    <ShipCity>Elgin</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97827</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>HUNGC</CustomerID>
  <EmployeeID>4</EmployeeID>
  <OrderDate>1997-07-16T00:00:00</OrderDate>
  <RequiredDate>1997-08-13T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1997-07-21T00:00:00">
    <ShipVia>1</ShipVia>
    <Freight>45.13</Freight>
    <ShipName>Hungry Coyote Import Store</ShipName>
    <ShipAddress>City Center Plaza 516 Main St.</ShipAddress>
    <ShipCity>Elgin</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97827</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>HUNGC</CustomerID>
  <EmployeeID>8</EmployeeID>
  <OrderDate>1997-09-08T00:00:00</OrderDate>
  <RequiredDate>1997-10-06T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1997-10-15T00:00:00">
    <ShipVia>1</ShipVia>
    <Freight>111.29</Freight>
    <ShipName>Hungry Coyote Import Store</ShipName>
    <ShipAddress>City Center Plaza 516 Main St.</ShipAddress>
    <ShipCity>Elgin</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97827</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
```

```
<CustomerID>LAZYK</CustomerID>
<EmployeeID>1</EmployeeID>
<OrderDate>1997-03-21T00:00:00</OrderDate>
<RequiredDate>1997-04-18T00:00:00</RequiredDate>
<ShipInfo ShippedDate="1997-04-10T00:00:00">
  <ShipVia>3</ShipVia>
  <Freight>7.48</Freight>
  <ShipName>Lazy K Kountry Store</ShipName>
  <ShipAddress>12 Orchestra Terrace</ShipAddress>
  <ShipCity>Walla Walla</ShipCity>
  <ShipRegion>WA</ShipRegion>
  <ShipPostalCode>99362</ShipPostalCode>
  <ShipCountry>USA</ShipCountry>
</ShipInfo>
</Order>
<Order>
  <CustomerID>LAZYK</CustomerID>
  <EmployeeID>8</EmployeeID>
  <OrderDate>1997-05-22T00:00:00</OrderDate>
  <RequiredDate>1997-06-19T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1997-06-26T00:00:00">
    <ShipVia>2</ShipVia>
    <Freight>11.92</Freight>
    <ShipName>Lazy K Kountry Store</ShipName>
    <ShipAddress>12 Orchestra Terrace</ShipAddress>
    <ShipCity>Walla Walla</ShipCity>
    <ShipRegion>WA</ShipRegion>
    <ShipPostalCode>99362</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>LETSS</CustomerID>
  <EmployeeID>1</EmployeeID>
  <OrderDate>1997-06-25T00:00:00</OrderDate>
  <RequiredDate>1997-07-23T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1997-07-04T00:00:00">
    <ShipVia>2</ShipVia>
    <Freight>13.73</Freight>
    <ShipName>Let's Stop N Shop</ShipName>
    <ShipAddress>87 Polk St. Suite 5</ShipAddress>
    <ShipCity>San Francisco</ShipCity>
    <ShipRegion>CA</ShipRegion>
    <ShipPostalCode>94117</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>LETSS</CustomerID>
  <EmployeeID>8</EmployeeID>
  <OrderDate>1997-10-27T00:00:00</OrderDate>
  <RequiredDate>1997-11-24T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1997-11-05T00:00:00">
    <ShipVia>2</ShipVia>
    <Freight>51.44</Freight>
    <ShipName>Let's Stop N Shop</ShipName>
    <ShipAddress>87 Polk St. Suite 5</ShipAddress>
    <ShipCity>San Francisco</ShipCity>
    <ShipRegion>CA</ShipRegion>
    <ShipPostalCode>94117</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>LETSS</CustomerID>
  <EmployeeID>6</EmployeeID>
  <OrderDate>1997-11-10T00:00:00</OrderDate>
  <RequiredDate>1997-12-08T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1997-11-21T00:00:00">
```

```
<ShipVia>2</ShipVia>
<Freight>45.97</Freight>
<ShipName>Let's Stop N Shop</ShipName>
<ShipAddress>87 Polk St. Suite 5</ShipAddress>
<ShipCity>San Francisco</ShipCity>
<ShipRegion>CA</ShipRegion>
<ShipPostalCode>94117</ShipPostalCode>
<ShipCountry>USA</ShipCountry>
</ShipInfo>
</Order>
<Order>
  <CustomerID>LETSS</CustomerID>
  <EmployeeID>4</EmployeeID>
  <OrderDate>1998-02-12T00:00:00</OrderDate>
  <RequiredDate>1998-03-12T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1998-02-13T00:00:00">
    <ShipVia>2</ShipVia>
    <Freight>90.97</Freight>
    <ShipName>Let's Stop N Shop</ShipName>
    <ShipAddress>87 Polk St. Suite 5</ShipAddress>
    <ShipCity>San Francisco</ShipCity>
    <ShipRegion>CA</ShipRegion>
    <ShipPostalCode>94117</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
</Orders>
</Root>
```

# Sample XSD File: Customers and Orders

6/4/2019 • 2 minutes to read • [Edit Online](#)

The following XSD file is used in various examples in the LINQ to XML documentation. This file contains a schema definition for the [Sample XML File: Customers and Orders \(LINQ to XML\)](#). The schema uses the `xs:key` and `xs:keyref` features of XSD to establish that the `CustomerID` attribute of the `Customer` element is a key, and to establish a relationship between the `CustomerID` element in each `Order` element and the `CustomerID` attribute in each `Customer` element.

For an example of writing LINQ queries that take advantage of this relationship using the `Join` clause, see [How to: Join Two Collections \(LINQ to XML\) \(C#\)](#).

## CustomersOrders.xsd

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name='Root'>
    <xs:complexType>
      <xs:sequence>
        <xs:element name='Customers'>
          <xs:complexType>
            <xs:sequence>
              <xs:element name='Customer' type='CustomerType' minOccurs='0' maxOccurs='unbounded' />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name='Orders'>
          <xs:complexType>
            <xs:sequence>
              <xs:element name='Order' type='OrderType' minOccurs='0' maxOccurs='unbounded' />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:key name='CustomerIDKey'>
    <xs:selector xpath='Customers/Customer' />
    <xs:field xpath='@CustomerID' />
  </xs:key>
  <xs:keyref name='CustomerIDKeyRef' refer='CustomerIDKey'>
    <xs:selector xpath='Orders/Order' />
    <xs:field xpath='CustomerID' />
  </xs:keyref>
</xs:schema>

<xs:complexType name='CustomerType'>
  <xs:sequence>
    <xs:element name='CompanyName' type='xs:string' />
    <xs:element name='ContactName' type='xs:string' />
    <xs:element name='ContactTitle' type='xs:string' />
    <xs:element name='Phone' type='xs:string' />
    <xs:element name='Fax' minOccurs='0' type='xs:string' />
    <xs:element name='FullAddress' type='AddressType' />
  </xs:sequence>
  <xs:attribute name='CustomerID' type='xs:token' />
</xs:complexType>

<xs:complexType name='AddressType'>
  <xs:sequence>
    <xs:element name='Address' type='xs:string' />
    <xs:element name='City' type='xs:string' />
```

```
<xs:element name='Region' type='xs:string' />
<xs:element name='PostalCode' type='xs:string' />
<xs:element name='Country' type='xs:string' />
</xs:sequence>
<xs:attribute name='CustomerID' type='xs:token' />
</xs:complexType>
<xs:complexType name='OrderType'>
  <xs:sequence>
    <xs:element name='CustomerID' type='xs:token' />
    <xs:element name='EmployeeID' type='xs:token' />
    <xs:element name='OrderDate' type='xs:dateTime' />
    <xs:element name='RequiredDate' type='xs:dateTime' />
    <xs:element name='ShipInfo' type='ShipInfoType' />
  </xs:sequence>
</xs:complexType>
<xs:complexType name='ShipInfoType'>
  <xs:sequence>
    <xs:element name='ShipVia' type='xs:integer' />
    <xs:element name='Freight' type='xs:decimal' />
    <xs:element name='ShipName' type='xs:string' />
    <xs:element name='ShipAddress' type='xs:string' />
    <xs:element name='ShipCity' type='xs:string' />
    <xs:element name='ShipRegion' type='xs:string' />
    <xs:element name='ShipPostalCode' type='xs:string' />
    <xs:element name='ShipCountry' type='xs:string' />
  </xs:sequence>
  <xs:attribute name='ShippedDate' type='xs:dateTime' />
</xs:complexType>
</xs:schema>
```

# Sample XML File: Customers and Orders in a Namespace

6/4/2019 • 3 minutes to read • [Edit Online](#)

The following XML file is used in various examples in the LINQ to XML documentation. This file contains customers and orders. The XML is in a namespace.

## CustomersOrdersInNamespace.xml

```
<?xml version="1.0" encoding="utf-8"?>
<Root xmlns="http://www.adventure-works.com">
  <Customers>
    <Customer CustomerID="GREAL">
      <CompanyName>Great Lakes Food Market</CompanyName>
      <ContactName>Howard Snyder</ContactName>
      <ContactTitle>Marketing Manager</ContactTitle>
      <Phone>(503) 555-7555</Phone>
      <FullAddress>
        <Address>2732 Baker Blvd.</Address>
        <City>Eugene</City>
        <Region>OR</Region>
        <PostalCode>97403</PostalCode>
        <Country>USA</Country>
      </FullAddress>
    </Customer>
    <Customer CustomerID="HUNGC">
      <CompanyName>Hungry Coyote Import Store</CompanyName>
      <ContactName>Yoshi Latimer</ContactName>
      <ContactTitle>Sales Representative</ContactTitle>
      <Phone>(503) 555-6874</Phone>
      <Fax>(503) 555-2376</Fax>
      <FullAddress>
        <Address>City Center Plaza 516 Main St.</Address>
        <City>Elgin</City>
        <Region>OR</Region>
        <PostalCode>97827</PostalCode>
        <Country>USA</Country>
      </FullAddress>
    </Customer>
    <Customer CustomerID="LAZYK">
      <CompanyName>Lazy K Kountry Store</CompanyName>
      <ContactName>John Steel</ContactName>
      <ContactTitle>Marketing Manager</ContactTitle>
      <Phone>(509) 555-7969</Phone>
      <Fax>(509) 555-6221</Fax>
      <FullAddress>
        <Address>12 Orchestra Terrace</Address>
        <City>Walla Walla</City>
        <Region>WA</Region>
        <PostalCode>99362</PostalCode>
        <Country>USA</Country>
      </FullAddress>
    </Customer>
    <Customer CustomerID="LETSS">
      <CompanyName>Let's Stop N Shop</CompanyName>
      <ContactName>Jaime Yorres</ContactName>
      <ContactTitle>Owner</ContactTitle>
      <Phone>(415) 555-5938</Phone>
      <FullAddress>
        <Address>87 Park St. Suite 5</Address>
```

```
<Address><FullAddress>
  <City>San Francisco</City>
  <Region>CA</Region>
  <PostalCode>94117</PostalCode>
  <Country>USA</Country>
</FullAddress>
</Customer>
</Customers>
<Orders>
  <Order>
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>6</EmployeeID>
    <OrderDate>1997-05-06T00:00:00</OrderDate>
    <RequiredDate>1997-05-20T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-05-09T00:00:00">
      <ShipVia>2</ShipVia>
      <Freight>3.35</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>8</EmployeeID>
    <OrderDate>1997-07-04T00:00:00</OrderDate>
    <RequiredDate>1997-08-01T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-07-14T00:00:00">
      <ShipVia>2</ShipVia>
      <Freight>4.42</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>1</EmployeeID>
    <OrderDate>1997-07-31T00:00:00</OrderDate>
    <RequiredDate>1997-08-28T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-08-05T00:00:00">
      <ShipVia>2</ShipVia>
      <Freight>116.53</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>4</EmployeeID>
    <OrderDate>1997-07-31T00:00:00</OrderDate>
    <RequiredDate>1997-08-28T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-08-04T00:00:00">
      <ShipVia>2</ShipVia>
      <Freight>18.53</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
```

```
<ShipPostalCode>97403</ShipPostalCode>
<ShipCountry>USA</ShipCountry>
</ShipInfo>
</Order>
<Order>
  <CustomerID>GREAL</CustomerID>
  <EmployeeID>6</EmployeeID>
  <OrderDate>1997-09-04T00:00:00</OrderDate>
  <RequiredDate>1997-10-02T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1997-09-10T00:00:00">
    <ShipVia>1</ShipVia>
    <Freight>57.15</Freight>
    <ShipName>Great Lakes Food Market</ShipName>
    <ShipAddress>2732 Baker Blvd.</ShipAddress>
    <ShipCity>Eugene</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97403</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>GREAL</CustomerID>
  <EmployeeID>3</EmployeeID>
  <OrderDate>1997-09-25T00:00:00</OrderDate>
  <RequiredDate>1997-10-23T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1997-09-30T00:00:00">
    <ShipVia>3</ShipVia>
    <Freight>76.13</Freight>
    <ShipName>Great Lakes Food Market</ShipName>
    <ShipAddress>2732 Baker Blvd.</ShipAddress>
    <ShipCity>Eugene</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97403</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>GREAL</CustomerID>
  <EmployeeID>4</EmployeeID>
  <OrderDate>1998-01-06T00:00:00</OrderDate>
  <RequiredDate>1998-02-03T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1998-02-04T00:00:00">
    <ShipVia>2</ShipVia>
    <Freight>719.78</Freight>
    <ShipName>Great Lakes Food Market</ShipName>
    <ShipAddress>2732 Baker Blvd.</ShipAddress>
    <ShipCity>Eugene</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97403</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>GREAL</CustomerID>
  <EmployeeID>3</EmployeeID>
  <OrderDate>1998-03-09T00:00:00</OrderDate>
  <RequiredDate>1998-04-06T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1998-03-18T00:00:00">
    <ShipVia>2</ShipVia>
    <Freight>33.68</Freight>
    <ShipName>Great Lakes Food Market</ShipName>
    <ShipAddress>2732 Baker Blvd.</ShipAddress>
    <ShipCity>Eugene</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97403</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
```



```
<CustomerID>GREAL</CustomerID>
<EmployeeID>3</EmployeeID>
<OrderDate>1998-04-07T00:00:00</OrderDate>
<RequiredDate>1998-05-05T00:00:00</RequiredDate>
<ShipInfo ShippedDate="1998-04-15T00:00:00">
  <ShipVia>2</ShipVia>
  <Freight>25.19</Freight>
  <ShipName>Great Lakes Food Market</ShipName>
  <ShipAddress>2732 Baker Blvd.</ShipAddress>
  <ShipCity>Eugene</ShipCity>
  <ShipRegion>OR</ShipRegion>
  <ShipPostalCode>97403</ShipPostalCode>
  <ShipCountry>USA</ShipCountry>
</ShipInfo>
</Order>
<Order>
  <CustomerID>GREAL</CustomerID>
  <EmployeeID>4</EmployeeID>
  <OrderDate>1998-04-22T00:00:00</OrderDate>
  <RequiredDate>1998-05-20T00:00:00</RequiredDate>
  <ShipInfo>
    <ShipVia>3</ShipVia>
    <Freight>18.84</Freight>
    <ShipName>Great Lakes Food Market</ShipName>
    <ShipAddress>2732 Baker Blvd.</ShipAddress>
    <ShipCity>Eugene</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97403</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>GREAL</CustomerID>
  <EmployeeID>4</EmployeeID>
  <OrderDate>1998-04-30T00:00:00</OrderDate>
  <RequiredDate>1998-06-11T00:00:00</RequiredDate>
  <ShipInfo>
    <ShipVia>3</ShipVia>
    <Freight>14.01</Freight>
    <ShipName>Great Lakes Food Market</ShipName>
    <ShipAddress>2732 Baker Blvd.</ShipAddress>
    <ShipCity>Eugene</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97403</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>HUNGC</CustomerID>
  <EmployeeID>3</EmployeeID>
  <OrderDate>1996-12-06T00:00:00</OrderDate>
  <RequiredDate>1997-01-03T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1996-12-09T00:00:00">
    <ShipVia>2</ShipVia>
    <Freight>20.12</Freight>
    <ShipName>Hungry Coyote Import Store</ShipName>
    <ShipAddress>City Center Plaza 516 Main St.</ShipAddress>
    <ShipCity>Elgin</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97827</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>HUNGC</CustomerID>
  <EmployeeID>1</EmployeeID>
  <OrderDate>1996-12-25T00:00:00</OrderDate>
  <RequiredDate>1997-01-22T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1997-01-03T00:00:00">
```

```
<ShipVia>3</ShipVia>
<Freight>30.34</Freight>
<ShipName>Hungry Coyote Import Store</ShipName>
<ShipAddress>City Center Plaza 516 Main St.</ShipAddress>
<ShipCity>Elgin</ShipCity>
<ShipRegion>OR</ShipRegion>
<ShipPostalCode>97827</ShipPostalCode>
<ShipCountry>USA</ShipCountry>
</ShipInfo>
</Order>
<Order>
  <CustomerID>HUNGC</CustomerID>
  <EmployeeID>3</EmployeeID>
  <OrderDate>1997-01-15T00:00:00</OrderDate>
  <RequiredDate>1997-02-12T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1997-01-24T00:00:00">
    <ShipVia>1</ShipVia>
    <Freight>0.2</Freight>
    <ShipName>Hungry Coyote Import Store</ShipName>
    <ShipAddress>City Center Plaza 516 Main St.</ShipAddress>
    <ShipCity>Elgin</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97827</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>HUNGC</CustomerID>
  <EmployeeID>4</EmployeeID>
  <OrderDate>1997-07-16T00:00:00</OrderDate>
  <RequiredDate>1997-08-13T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1997-07-21T00:00:00">
    <ShipVia>1</ShipVia>
    <Freight>45.13</Freight>
    <ShipName>Hungry Coyote Import Store</ShipName>
    <ShipAddress>City Center Plaza 516 Main St.</ShipAddress>
    <ShipCity>Elgin</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97827</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>HUNGC</CustomerID>
  <EmployeeID>8</EmployeeID>
  <OrderDate>1997-09-08T00:00:00</OrderDate>
  <RequiredDate>1997-10-06T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1997-10-15T00:00:00">
    <ShipVia>1</ShipVia>
    <Freight>111.29</Freight>
    <ShipName>Hungry Coyote Import Store</ShipName>
    <ShipAddress>City Center Plaza 516 Main St.</ShipAddress>
    <ShipCity>Elgin</ShipCity>
    <ShipRegion>OR</ShipRegion>
    <ShipPostalCode>97827</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>LAZYK</CustomerID>
  <EmployeeID>1</EmployeeID>
  <OrderDate>1997-03-21T00:00:00</OrderDate>
  <RequiredDate>1997-04-18T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1997-04-10T00:00:00">
    <ShipVia>3</ShipVia>
    <Freight>7.48</Freight>
    <ShipName>Lazy K Kountry Store</ShipName>
    <ShipAddress>12 Orchestra Terrace</ShipAddress>
    <ShipCity>Walla Walla</ShipCity>
```

```
<ShipRegion>WA</ShipRegion>
<ShipPostalCode>99362</ShipPostalCode>
<ShipCountry>USA</ShipCountry>
</ShipInfo>
</Order>
<Order>
  <CustomerID>LAZYK</CustomerID>
  <EmployeeID>8</EmployeeID>
  <OrderDate>1997-05-22T00:00:00</OrderDate>
  <RequiredDate>1997-06-19T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1997-06-26T00:00:00">
    <ShipVia>2</ShipVia>
    <Freight>11.92</Freight>
    <ShipName>Lazy K Kountry Store</ShipName>
    <ShipAddress>12 Orchestra Terrace</ShipAddress>
    <ShipCity>Walla Walla</ShipCity>
    <ShipRegion>WA</ShipRegion>
    <ShipPostalCode>99362</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>LETSS</CustomerID>
  <EmployeeID>1</EmployeeID>
  <OrderDate>1997-06-25T00:00:00</OrderDate>
  <RequiredDate>1997-07-23T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1997-07-04T00:00:00">
    <ShipVia>2</ShipVia>
    <Freight>13.73</Freight>
    <ShipName>Let's Stop N Shop</ShipName>
    <ShipAddress>87 Polk St. Suite 5</ShipAddress>
    <ShipCity>San Francisco</ShipCity>
    <ShipRegion>CA</ShipRegion>
    <ShipPostalCode>94117</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>LETSS</CustomerID>
  <EmployeeID>8</EmployeeID>
  <OrderDate>1997-10-27T00:00:00</OrderDate>
  <RequiredDate>1997-11-24T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1997-11-05T00:00:00">
    <ShipVia>2</ShipVia>
    <Freight>51.44</Freight>
    <ShipName>Let's Stop N Shop</ShipName>
    <ShipAddress>87 Polk St. Suite 5</ShipAddress>
    <ShipCity>San Francisco</ShipCity>
    <ShipRegion>CA</ShipRegion>
    <ShipPostalCode>94117</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
<Order>
  <CustomerID>LETSS</CustomerID>
  <EmployeeID>6</EmployeeID>
  <OrderDate>1997-11-10T00:00:00</OrderDate>
  <RequiredDate>1997-12-08T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1997-11-21T00:00:00">
    <ShipVia>2</ShipVia>
    <Freight>45.97</Freight>
    <ShipName>Let's Stop N Shop</ShipName>
    <ShipAddress>87 Polk St. Suite 5</ShipAddress>
    <ShipCity>San Francisco</ShipCity>
    <ShipRegion>CA</ShipRegion>
    <ShipPostalCode>94117</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
```

```
<Order>
  <CustomerID>LETSS</CustomerID>
  <EmployeeID>4</EmployeeID>
  <OrderDate>1998-02-12T00:00:00</OrderDate>
  <RequiredDate>1998-03-12T00:00:00</RequiredDate>
  <ShipInfo ShippedDate="1998-02-13T00:00:00">
    <ShipVia>2</ShipVia>
    <Freight>90.97</Freight>
    <ShipName>Let's Stop N Shop</ShipName>
    <ShipAddress>87 Polk St. Suite 5</ShipAddress>
    <ShipCity>San Francisco</ShipCity>
    <ShipRegion>CA</ShipRegion>
    <ShipPostalCode>94117</ShipPostalCode>
    <ShipCountry>USA</ShipCountry>
  </ShipInfo>
</Order>
</Orders>
</Root>
```

# Sample XML File: Numerical Data (LINQ to XML)

6/4/2019 • 2 minutes to read • [Edit Online](#)

The following XML file is used in various examples in the LINQ to XML documentation. This file contains numerical data for summing, averaging, and grouping.

## Data.xml

```
<Root>
  <TaxRate>7.25</TaxRate>
  <Data>
    <Category>A</Category>
    <Quantity>3</Quantity>
    <Price>24.50</Price>
  </Data>
  <Data>
    <Category>B</Category>
    <Quantity>1</Quantity>
    <Price>89.99</Price>
  </Data>
  <Data>
    <Category>A</Category>
    <Quantity>5</Quantity>
    <Price>4.95</Price>
  </Data>
  <Data>
    <Category>A</Category>
    <Quantity>3</Quantity>
    <Price>66.00</Price>
  </Data>
  <Data>
    <Category>B</Category>
    <Quantity>10</Quantity>
    <Price>.99</Price>
  </Data>
  <Data>
    <Category>A</Category>
    <Quantity>15</Quantity>
    <Price>29.00</Price>
  </Data>
  <Data>
    <Category>B</Category>
    <Quantity>8</Quantity>
    <Price>6.99</Price>
  </Data>
</Root>
```

# Sample XML File: Numerical Data in a Namespace

6/4/2019 • 2 minutes to read • [Edit Online](#)

The following XML file is used in various examples in the LINQ to XML documentation. This file contains numerical data for summing, averaging, and grouping. The XML is in a namespace.

## Data

```
<Root xmlns='http://www.adatum.com'>
  <TaxRate>7.25</TaxRate>
  <Data>
    <Category>A</Category>
    <Quantity>3</Quantity>
    <Price>24.50</Price>
  </Data>
  <Data>
    <Category>B</Category>
    <Quantity>1</Quantity>
    <Price>89.99</Price>
  </Data>
  <Data>
    <Category>A</Category>
    <Quantity>5</Quantity>
    <Price>4.95</Price>
  </Data>
  <Data>
    <Category>A</Category>
    <Quantity>3</Quantity>
    <Price>66.00</Price>
  </Data>
  <Data>
    <Category>B</Category>
    <Quantity>10</Quantity>
    <Price>.99</Price>
  </Data>
  <Data>
    <Category>A</Category>
    <Quantity>15</Quantity>
    <Price>29.00</Price>
  </Data>
  <Data>
    <Category>B</Category>
    <Quantity>8</Quantity>
    <Price>6.99</Price>
  </Data>
</Root>
```

# Sample XML File: Books (LINQ to XML)

6/4/2019 • 2 minutes to read • [Edit Online](#)

The following XML file is used in various examples in the LINQ to XML documentation. The file contains information about books.

## books.xml

```
<?xml version="1.0"?>
<Catalog>
  <Book id="bk101">
    <Author>Garghentini, Davide</Author>
    <Title>XML Developer's Guide</Title>
    <Genre>Computer</Genre>
    <Price>44.95</Price>
    <PublishDate>2000-10-01</PublishDate>
    <Description>An in-depth look at creating applications
    with XML.</Description>
  </Book>
  <Book id="bk102">
    <Author>Garcia, Debra</Author>
    <Title>Midnight Rain</Title>
    <Genre>Fantasy</Genre>
    <Price>5.95</Price>
    <PublishDate>2000-12-16</PublishDate>
    <Description>A former architect battles corporate zombies,
    an evil sorceress, and her own childhood to become queen
    of the world.</Description>
  </Book>
</Catalog>
```

# Sample XML File: Consolidated Purchase Orders

6/4/2019 • 2 minutes to read • [Edit Online](#)

The following XML file is used in various examples in the LINQ to XML documentation. This file is a set of purchase orders with different shapes from multiple companies. Purchase orders from each company are in separate namespaces.

## ConsolidatedPurchaseOrders.xml

```
<?xml version="1.0"?>
<PurchaseOrders xmlns="www.contoso.com">
  <PurchaseOrder
    PurchaseOrderNumber="99503"
    OrderDate="1999-10-20">
    <Address Type="Shipping">
      <Name>Ellen Adams</Name>
      <Street>123 Maple Street</Street>
      <City>Mill Valley</City>
      <State>CA</State>
      <Zip>10999</Zip>
      <Country>USA</Country>
    </Address>
    <Address Type="Billing">
      <Name>Tai Yee</Name>
      <Street>8 Oak Avenue</Street>
      <City>Old Town</City>
      <State>PA</State>
      <Zip>95819</Zip>
      <Country>USA</Country>
    </Address>
    <DeliveryNotes>Please leave packages in shed by driveway.</DeliveryNotes>
    <Items>
      <Item PartNumber="872-AA">
        <ProductName>Lawnmower</ProductName>
        <Quantity>1</Quantity>
        <USPrice>148.95</USPrice>
        <Comment>Confirm this is electric</Comment>
      </Item>
      <Item PartNumber="926-AA">
        <ProductName>Baby Monitor</ProductName>
        <Quantity>2</Quantity>
        <USPrice>39.98</USPrice>
        <ShipDate>1999-05-21</ShipDate>
      </Item>
    </Items>
  </PurchaseOrder>
  <PurchaseOrder PurchaseOrderNumber="99505" OrderDate="1999-10-22">
    <Address Type="Shipping">
      <Name>Cristian Osorio</Name>
      <Street>456 Main Street</Street>
      <City>Buffalo</City>
      <State>NY</State>
      <Zip>98112</Zip>
      <Country>USA</Country>
    </Address>
    <Address Type="Billing">
      <Name>Cristian Osorio</Name>
      <Street>456 Main Street</Street>
      <City>Buffalo</City>
      <State>NY</State>
      <Zip>98112</Zip>
```



```
<!--PurchaseOrder-->  
    <Country>USA</Country>  
  </Address>  
  <DeliveryNotes>Please notify by email before shipping.</DeliveryNotes>  
  <Items>  
    <Item PartNumber="456-NM">  
      <ProductName>Power Supply</ProductName>  
      <Quantity>1</Quantity>  
      <USPrice>45.99</USPrice>  
    </Item>  
  </Items>  
</PurchaseOrder>  
  
<PurchaseOrder PurchaseOrderNumber="99504" OrderDate="1999-10-22">  
  <Address Type="Shipping">  
    <Name>Jessica Arnold</Name>  
    <Street>4055 Madison Ave</Street>  
    <City>Seattle</City>  
    <State>WA</State>  
    <Zip>98112</Zip>  
    <Country>USA</Country>  
  </Address>  
  <Address Type="Billing">  
    <Name>Jessica Arnold</Name>  
    <Street>4055 Madison Ave</Street>  
    <City>Buffalo</City>  
    <State>NY</State>  
    <Zip>98112</Zip>  
    <Country>USA</Country>  
  </Address>  
  <DeliveryNotes>Please do not deliver on Saturday.</DeliveryNotes>  
  <Items>  
    <Item PartNumber="898-AZ">  
      <ProductName>Computer Keyboard</ProductName>  
      <Quantity>1</Quantity>  
      <USPrice>29.99</USPrice>  
    </Item>  
    <Item PartNumber="898-AM">  
      <ProductName>Wireless Mouse</ProductName>  
      <Quantity>1</Quantity>  
      <USPrice>14.99</USPrice>  
    </Item>  
  </Items>  
</PurchaseOrder>  
  
<aw:PurchaseOrder  
  PONumber="11223"  
  Date="2000-01-15"  
  xmlns:aw="http://www.adventure-works.com">  
  <aw:ShippingAddress>  
    <aw:Name>Chris Preston</aw:Name>  
    <aw:Street>123 Main St.</aw:Street>  
    <aw:City>Seattle</aw:City>  
    <aw:State>WA</aw:State>  
    <aw:Zip>98113</aw:Zip>  
    <aw:Country>USA</aw:Country>  
  </aw:ShippingAddress>  
  <aw:BillingAddress>  
    <aw:Name>Chris Preston</aw:Name>  
    <aw:Street>123 Main St.</aw:Street>  
    <aw:City>Seattle</aw:City>  
    <aw:State>WA</aw:State>  
    <aw:Zip>98113</aw:Zip>  
    <aw:Country>USA</aw:Country>  
  </aw:BillingAddress>  
  <aw:DeliveryInstructions>Ship only complete order.</aw:DeliveryInstructions>  
  <aw:Item PartNum="LIT-01">  
    <aw:ProductID>Litware Networking Card</aw:ProductID>  
    <aw:Qty>1</aw:Qty>  
    <aw:Price>20.99</aw:Price>  
  </aw:Item>
```

```
<aw:Item PartNum="L11-25">
  <aw:ProductID>Litware 17in LCD Monitor</aw:ProductID>
  <aw:Qty>1</aw:Qty>
  <aw:Price>199.99</aw:Price>
</aw:Item>
</aw:PurchaseOrder>
</PurchaseOrders>
```

# Reference (LINQ to XML)

6/4/2019 • 2 minutes to read • [Edit Online](#)

This topic contains links to the LINQ to XML reference topics.

## In This Section

For reference documentation for the LINQ to XML classes, see [System.Xml.Linq](#).

For reference documentation for the extension methods that help you validate XML trees against an XSD file, see [System.Xml.Schema.Extensions](#).

For reference documentation for the extension methods that enable you to evaluate XPath queries on an XML tree, see [System.Xml.XPath.Extensions](#).

## See also

- [LINQ to XML \(C#\)](#)

# LINQ to ADO.NET (Portal Page)

5/18/2019 • 2 minutes to read • [Edit Online](#)

LINQ to ADO.NET enables you to query over any enumerable object in ADO.NET by using the Language-Integrated Query (LINQ) programming model.

## NOTE

The LINQ to ADO.NET documentation is located in the ADO.NET section of the .NET Framework SDK: [LINQ and ADO.NET](#).

There are three separate ADO.NET Language-Integrated Query (LINQ) technologies: LINQ to DataSet, LINQ to SQL, and LINQ to Entities. LINQ to DataSet provides richer, optimized querying over the [DataSet](#), LINQ to SQL enables you to directly query SQL Server database schemas, and LINQ to Entities allows you to query an Entity Data Model.

## LINQ to DataSet

The [DataSet](#) is one of the most widely used components in ADO.NET, and is a key element of the disconnected programming model that ADO.NET is built on. Despite this prominence, however, the [DataSet](#) has limited query capabilities.

LINQ to DataSet enables you to build richer query capabilities into [DataSet](#) by using the same query functionality that is available for many other data sources.

For more information, see [LINQ to DataSet](#).

## LINQ to SQL

LINQ to SQL provides a run-time infrastructure for managing relational data as objects. In LINQ to SQL, the data model of a relational database is mapped to an object model expressed in the programming language of the developer. When you execute the application, LINQ to SQL translates language-integrated queries in the object model into SQL and sends them to the database for execution. When the database returns the results, LINQ to SQL translates them back into objects that you can manipulate.

LINQ to SQL includes support for stored procedures and user-defined functions in the database, and for inheritance in the object model.

For more information, see [LINQ to SQL](#).

## LINQ to Entities

Through the Entity Data Model, relational data is exposed as objects in the .NET environment. This makes the object layer an ideal target for LINQ support, allowing developers to formulate queries against the database from the language used to build the business logic. This capability is known as LINQ to Entities. See [LINQ to Entities](#) for more information.

## See also

- [LINQ and ADO.NET](#)
- [Language-Integrated Query \(LINQ\) \(C#\)](#)

# Enabling a Data Source for LINQ Querying

4/28/2019 • 3 minutes to read • [Edit Online](#)

There are various ways to extend LINQ to enable any data source to be queried in the LINQ pattern. The data source might be a data structure, a Web service, a file system, or a database, to name some. The LINQ pattern makes it easy for clients to query a data source for which LINQ querying is enabled, because the syntax and pattern of the query does not change. The ways in which LINQ can be extended to these data sources include the following:

- Implementing the [IEnumerable<T>](#) interface in a type to enable LINQ to Objects querying of that type.
- Creating standard query operator methods such as [Where](#) and [Select](#) that extend a type, to enable custom LINQ querying of that type.
- Creating a provider for your data source that implements the [IQueryable<T>](#) interface. A provider that implements this interface receives LINQ queries in the form of expression trees, which it can execute in a custom way, for example remotely.
- Creating a provider for your data source that takes advantage of an existing LINQ technology. Such a provider would enable not only querying, but also insert, update, and delete operations and mapping for user-defined types.

This topic discusses these options.

## How to Enable LINQ Querying of Your Data Source

### In-Memory Data

There are two ways you can enable LINQ querying of in-memory data. If the data is of a type that implements [IEnumerable<T>](#), you can query the data by using LINQ to Objects. If it does not make sense to enable enumeration of your type by implementing the [IEnumerable<T>](#) interface, you can define LINQ standard query operator methods in that type or create LINQ standard query operator methods that extend the type. Custom implementations of the standard query operators should use deferred execution to return the results.

### Remote Data

The best option for enabling LINQ querying of a remote data source is to implement the [IQueryable<T>](#) interface. However, this differs from extending a provider such as LINQ to SQL for a data source. No provider models for extending existing LINQ technologies, such as LINQ to SQL, to other types of data source are available in Visual Studio 2008.

## IQueryable LINQ Providers

LINQ providers that implement [IQueryable<T>](#) can vary widely in their complexity. This section discusses the different levels of complexity.

A less complex `IQueryable` provider might interface with a single method of a Web service. This type of provider is very specific because it expects specific information in the queries that it handles. It has a closed type system, perhaps exposing a single result type. Most of the execution of the query occurs locally, for example by using the [Enumerable](#) implementations of the standard query operators. A less complex provider might examine only one method call expression in the expression tree that represents the query, and let the remaining logic of the query be handled elsewhere.

An `IQueryable` provider of medium complexity might target a data source that has a partially expressive query

language. If it targets a Web service, it might interface with more than one method of the Web service and select the method to call based on the question that the query poses. A provider of medium complexity would have a richer type system than a simple provider, but it would still be a fixed type system. For example, the provider might expose types that have one-to-many relationships that can be traversed, but it would not provide mapping technology for user-defined types.

A complex `IQueryable` provider, such as the LINQ to SQL provider, might translate complete LINQ queries to an expressive query language, such as SQL. A complex provider is more general than a less complex provider, because it can handle a wider variety of questions in the query. It also has an open type system and therefore must contain extensive infrastructure to map user-defined types. Developing a complex provider requires a significant amount of effort.

## See also

- [IQueryable<T>](#)
- [IEnumerable<T>](#)
- [Enumerable](#)
- [Standard Query Operators Overview \(C#\)](#)
- [LINQ to Objects \(C#\)](#)

# Visual Studio IDE and Tools Support for LINQ (C#)

1/23/2019 • 2 minutes to read • [Edit Online](#)

The Visual Studio integrated development environment (IDE) provides the following features that support LINQ application development:

## Object Relational Designer

The Object Relational Designer is a visual design tool that you can use in [LINQ to SQL](#) applications to generate classes in C# that represent the relational data in an underlying database. For more information, see [LINQ to SQL Tools in Visual Studio](#).

## SQLMetal Command Line Tool

SQLMetal is a command-line tool that can be used in build processes to generate classes from existing databases for use in LINQ to SQL applications. For more information, see [SqlMetal.exe \(Code Generation Tool\)](#).

## LINQ-Aware Code Editor

The C# code editor supports LINQ extensively with IntelliSense and formatting capabilities.

## Visual Studio Debugger Support

The Visual Studio debugger supports debugging of query expressions. For more information, see [Debugging LINQ](#).

## See also

- [Language-Integrated Query \(LINQ\) \(C#\)](#)