

**The .NET  
Standard Query Operators**

**May 2006**

**Notice**

© 2012 Microsoft Corporation. All rights reserved.

Microsoft, Windows, Visual Basic, Visual C#, and Visual C++ are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries/regions.

Other product and company names mentioned herein may be the trademarks of their respective owners.

**Table of Contents**

1. Technical Specification 1

1.1 The Func delegate types 2

1.2 The Sequence class 2

1.3 Restriction operators 2

1.3.1 Where 2

1.4 Projection operators 3

1.4.1 Select 3

1.4.2 SelectMany 4

1.5 Partitioning operators 5

1.5.1 Take 5

1.5.2 Skip 5

1.5.3 TakeWhile 6

1.5.4 SkipWhile 6

1.6 Join operators 6

1.6.1 Join 6

1.6.2 GroupJoin 7

1.7 Concatenation operator 9

1.7.1 Concat 9

1.8 Ordering operators 9

1.8.1 OrderBy / ThenBy 9

1.8.2 Reverse 11

1.9 Grouping operators 11

1.9.1 GroupBy 11

1.10 Set operators 12

1.10.1 Distinct 12

1.10.2 Union 13

1.10.3 Intersect 13

1.10.4 Except 13

1.11 Conversion operators 14

1.11.1 ToSequence 14

1.11.2 ToArray 14

1.11.3 ToList 14

1.11.4 ToDictionary 15

1.11.5 ToLookup 15

1.11.6 OfType 16

1.11.7 Cast 17

1.12 Equality operator 17

1.12.1 EqualAll 17

1.13 Element operators 17

1.13.1 First 17

1.13.2 FirstOrDefault 18

1.13.3 Last 18

1.13.4 LastOrDefault 18

1.13.5 Single 19

1.13.6 SingleOrDefault 19

1.13.7 ElementAt 20

1.13.8 ElementAtOrDefault 20

1.13.9 DefaultIfEmpty 20

1.14 Generation operators 21

1.14.1 Range 21

1.14.2 Repeat 21

1.14.3 Empty 21

1.15 Quantifiers 21

1.15.1 Any 21

1.15.2 All 22

1.15.3 Contains 22

1.16 Aggregate operators 22

1.16.1 Count 22

1.16.2 LongCount 23

1.16.3 Sum 23

1.16.4 Min 24

1.16.5 Max 24

1.16.6 Average 25

1.16.7 Aggregate 26

# Technical Specification

The Standard Query Operators is an API that enables querying of any .NET array or collection. The Standard Query Operators API consists of the methods declared in the System.Query.Sequence static class in the assembly named System.Query.dll.

The Standard Query Operators API complies with the .NET 2.0 Common Language Specification (CLS) and is usable with any .NET Language that supports generics. While not required, the experience of using the Standard Query Operators is significantly enhanced with languages that support extension methods, lambda expressions, and native query syntax. The future releases of C# 3.0 and VB 9.0 will include these features.

The Standard Query Operators operate on sequences. Any object that implements the interface IEnumerable<T> for some type T is considered a sequence of that type.

The examples shown in this specification are all written in C# 3.0 and assume that the Standard Query Operators have been imported with the using clause:

using System.Query;

The examples refer to the following classes:

public class Customer  
{  
 public int CustomerID;  
 public string Name;  
 public string Address;  
 public string City;  
 public string Region;  
 public string PostalCode;  
 public string Country;  
 public string Phone;  
 public List<Order> Orders;  
}

public class Order  
{  
 public int OrderID;   
 public int CustomerID;  
 public Customer Customer;  
 public DateTime OrderDate;  
 public decimal Total;  
}

public class Product  
{  
 public int ProductID;   
 public string Name;  
 public string Category;  
 public decimal UnitPrice;  
 public int UnitsInStock;  
}

The examples furthermore assume the existence of the following three variables:

List<Customer> customers = GetCustomerList();

List<Order> orders = GetOrderList();

List<Product> products = GetProductList();

## The Func delegate types

The System.Query.Func family of generic delegate types can be used to construct delegate types “on the fly”, thus eliminating the need for explicit delegate type declarations.

public delegate TR Func<TR>();

public delegate TR Func<T0, TR>(T0 a0);

public delegate TR Func<T0, T1, TR>(T0 a0, T1 a1);

public delegate TR Func<T0, T1, T2, TR>(T0 a0, T1 a1, T2 a2);

public delegate TR Func<T0, T1, T2, T3, TR>(T0 a0, T1 a1, T2 a2, T3 a3);

In each of the Func types, the T0, T1, T2, and T3 type parameters represent argument types and the TR type parameter represents the result type.

The example below declares a local variable predicate of a delegate type that takes a Customer and returns bool. The local variable is assigned an anonymous method that returns true if the given customer is located in London. The delegate referenced by predicate is subsequently used to find all the customers in London.

Func<Customer, bool> predicate = c => c.City == "London";

IEnumerable<Customer> customersInLondon = customers.Where(predicate);

## The Sequence class

The System.Query.Sequence static class declares a set of methods known as the Standard Query Operators. The remaining sections of this chapter discusses these methods.

The majority of the Standard Query Operators are extension methods that extend IEnumerable<T>. Taken together, the methods compose to form a complete query language for arrays and collections that implement IEnumerable<T>.

For further details on extension methods, please refer to the C# 3.0 and VB 9.0 Language Specifications.

## Restriction operators

### Where

The Where operator filters a sequence based on a predicate.

public static IEnumerable<T> Where<T>(  
 this IEnumerable<T> source,  
 Func<T, bool> predicate);

public static IEnumerable<T> Where<T>(  
 this IEnumerable<T> source,  
 Func<T, int, bool> predicate);

The Where operator allocates and returns an enumerable object that captures the arguments passed to the operator. An ArgumentNullException is thrown if either argument is null.

When the object returned by Where is enumerated, it enumerates the source sequence and yields those elements for which the predicate function returns true. The first argument of the predicate function represents the element to test. The second argument, if present, represents the zero based index of the element within the source sequence.

The following example creates a sequence of those products that have a price greater than or equal to 10:

IEnumerable<Product> x = products.Where(p => p.UnitPrice >= 10);

In a C# 3.0 query expression, a where clause translates to an invocation of Where. The example above is equivalent to the translation of

IEnumerable<Product> x =  
 from p in products  
 where p.UnitPrice >= 10  
 select p;

## Projection operators

### Select

The Select operator performs a projection over a sequence.

public static IEnumerable<S> Select<T, S>(  
 this IEnumerable<T> source,  
 Func<T, S> selector);

public static IEnumerable<S> Select<T, S>(  
 this IEnumerable<T> source,  
 Func<T, int, S> selector);

The Select operator allocates and returns an enumerable object that captures the arguments passed to the operator. An ArgumentNullException is thrown if either argument is null.

When the object returned by Select is enumerated, it enumerates the source sequence and yields the results of evaluating the selector function for each element. The first argument of the selector function represents the element to process. The second argument, if present, represents the zero based index of the element within the source sequence.

The following example creates a sequence of the names of all products:

IEnumerable<string> productNames = products.Select(p => p.Name);

In a C# 3.0 query expression, a select clause translates to an invocation of Select. The example above is equivalent to the translation of

IEnumerable<string> productNames = from p in products select p.Name;

The following example creates a list of objects containing the name and price of each product with a price greater than or equal to 10:

var namesAndPrices =  
 products.  
 Where(p => p.UnitPrice >= 10).  
 Select(p => new { p.Name, p.UnitPrice }).  
 ToList();

The following example creates a sequence of the indices of those products that have a price greater than or equal to 10:

IEnumerable<int> indices =  
 products.  
 Select((product, index) => new { product, index }).  
 Where(x => x.product.UnitPrice >= 10).  
 Select(x => x.index);

### SelectMany

The SelectMany operator performs a one to many element projection over a sequence.

public static IEnumerable<S> SelectMany<T, S>(  
 this IEnumerable<T> source,  
 Func<T, IEnumerable<S>> selector);

public static IEnumerable<S> SelectMany<T, S>(  
 this IEnumerable<T> source,  
 Func<T, int, IEnumerable<S>> selector);

The SelectMany operator allocates and returns an enumerable object that captures the arguments passed to the operator. An ArgumentNullException is thrown if either argument is null.

When the object returned by SelectMany is enumerated, it enumerates the source sequence, maps each element to an enumerable object using the selector function, and enumerates and yields the elements of each such enumerable object. The first argument of the selector function represents the element to process. The second argument, if present, represents the zero based index of the element within the source sequence.

The following example creates a sequence of the orders of the customers in Denmark:

IEnumerable<Order> orders =  
 customers.  
 Where(c => c.Country == "Denmark").  
 SelectMany(c => c.Orders);

If the query had used Select instead of SelectMany, the result would have been of type IEnumerable<List<Order>> instead of IEnumerable<Order>.

The following example creates a sequence of objects containing the customer name and order ID of the orders in 2005 of the customers in Denmark:

var namesAndOrderIDs =  
 customers.  
 Where(c => c.Country == "Denmark").  
 SelectMany(c => c.Orders).  
 Where(o => o.OrderDate.Year == 2005).  
 Select(o => new { o.Customer.Name, o.OrderID });

In the example above, the Customer property is used to “navigate back” to fetch the Name property of the order’s customer. If an order had no Customer property (i.e. if the relationship was unidirectional), an alternative solution is to rewrite the query, keeping the current customer, c, in scope such that it can be referenced in the final Select:

var namesAndOrderIDs =  
 customers.  
 Where(c => c.Country == "Denmark").  
 SelectMany(c =>  
 c.Orders.  
 Where(o => o.OrderDate.Year == 2005).  
 Select(o => new { c.Name, o.OrderID })  
 );

In a C# 3.0 query expression, all but the initial from clause translate to invocations of SelectMany. The example above is equivalent to the translation of

var namesAndOrderIDs =  
 from c in customers  
 where c.Country == "Denmark"  
 from o in c.Orders  
 where o.OrderDate.Year == 2005  
 select new { c.Name, o.OrderID };

## Partitioning operators

### Take

The Take operator yields a given number of elements from a sequence and then skips the remainder of the sequence.

public static IEnumerable<T> Take<T>(  
 this IEnumerable<T> source,  
 int count);

The Take operator allocates and returns an enumerable object that captures the arguments passed to the operator. An ArgumentNullException is thrown if the source argument is null.

When the object returned by Take is enumerated, it enumerates the source sequence and yields elements until the number of elements given by the count argument have been yielded or the end of the source is reached. If the count argument is less than or equal to zero, the source sequence is not enumerated and no elements are yielded.

The Take and Skip operators are functional complements: For a given sequence s, the concatenation of s.Take(n) and s.Skip(n) yields the same sequence as s.

The following example creates a sequence of the most expensive 10 products:

IEnumerable<Product> MostExpensive10 =  
 products.OrderByDescending(p => p.UnitPrice).Take(10);

### Skip

The Skip operator skips a given number of elements from a sequence and then yields the remainder of the sequence.

public static IEnumerable<T> Skip<T>(  
 this IEnumerable<T> source,  
 int count);

The Skip operator allocates and returns an enumerable object that captures the arguments passed to the operator. An ArgumentNullException is thrown if the source argument is null.

When the object returned by Skip is enumerated, it enumerates the source sequence, skipping the number of elements given by the count argument and yielding the rest. If the source sequence contains fewer elements than given by the count argument, nothing is yielded. If the count argument is less an or equal to zero, all elements of the source sequence are yielded.

The Take and Skip operators are functional complements: Given a sequence s, the concatenation of s.Take(n) and s.Skip(n) is the same sequence as s.

The following example creates a sequence of all but the most expensive 10 products:

IEnumerable<Product> AllButMostExpensive10 =  
 products.OrderByDescending(p => p.UnitPrice).Skip(10);

### TakeWhile

The TakeWhile operator yields elements from a sequence while a test is true and then skips the remainder of the sequence.

public static IEnumerable<T> TakeWhile<T>(  
 this IEnumerable<T> source,  
 Func<T, bool> predicate);

public static IEnumerable<T> TakeWhile<T>(  
 this IEnumerable<T> source,  
 Func<T, int, bool> predicate);

The TakeWhile operator allocates and returns an enumerable object that captures the arguments passed to the operator. An ArgumentNullException is thrown if either argument is null.

When the object returned by TakeWhile is enumerated, it enumerates the source sequence, testing each element using the predicate function and yielding the element if the result was true. The enumeration stops when the predicate function returns false or the end of the source sequence is reached. The first argument of the predicate function represents the element to test. The second argument, if present, represents the zero based index of the element within the source sequence.

The TakeWhile and SkipWhile operators are functional complements: Given a sequence s and a pure function p, the concatenation of s.TakeWhile(p) and s.SkipWhile(p) is the same sequence as s.

### SkipWhile

The SkipWhile operator skips elements from a sequence while a test is true and then yields the remainder of the sequence.

public static IEnumerable<T> SkipWhile<T>(  
 this IEnumerable<T> source,  
 Func<T, bool> predicate);

public static IEnumerable<T> SkipWhile<T>(  
 this IEnumerable<T> source,  
 Func<T, int, bool> predicate);

The SkipWhile operator allocates and returns an enumerable object that captures the arguments passed to the operator. An ArgumentNullException is thrown if either argument is null.

When the object returned by SkipWhile is enumerated, it enumerates the source sequence, testing each element using the predicate function and skipping the element if the result was true. Once the predicate function returns false for an element, that element and the remaining elements are yielded with no further invocations of the predicate function. If the predicate function returns true for all elements in the sequence, no elements are yielded. The first argument of the predicate function represents the element to test. The second argument, if present, represents the zero based index of the element within the source sequence.

The TakeWhile and SkipWhile operators are functional complements: Given a sequence s and a pure function p, the concatenation of s.TakeWhile(p) and s.SkipWhile(p) is the same sequence as s.

## Join operators

### Join

The Join operator performs an inner join of two sequences based on matching keys extracted from the elements.

public static IEnumerable<V> Join<T, U, K, V>(  
 this IEnumerable<T> outer,  
 IEnumerable<U> inner,  
 Func<T, K> outerKeySelector,  
 Func<U, K> innerKeySelector,  
 Func<T, U, V> resultSelector);

The Join operator allocates and returns an enumerable object that captures the arguments passed to the operator. An ArgumentNullException is thrown if any argument is null.

The outerKeySelector and innerKeySelector arguments specify functions that extract the join key values from elements of the outer and inner sequences, respectively. The resultSelector argument specifies a function that creates a result element from two matching outer and inner sequence elements.

When the object returned by Join is enumerated, it first enumerates the inner sequence and evaluates the innerKeySelector function once for each inner element, collecting the elements by their keys in a hash table. Once all inner elements and keys have been collected, the outer sequence is enumerated. For each outer element, the outerKeySelector function is evaluated and the resulting key is used to look up the corresponding inner elements in the hash table. For each matching inner element (if any), the resultSelector function is evaluated for the outer and inner element pair, and the resulting object is yielded.

The Join operator preserves the order of the outer sequence elements, and for each outer element, the order of the matching inner sequence elements.

In relational database terms, the Join operator implements an inner equijoin. Other join operations, such as left outer join and right outer join have no dedicated standard query operators, but are subsets of the capabilities of the GroupJoin operator.

The following example joins customers and orders on their customer ID property, producing a sequence of tuples with customer name, order date, and order total:

var custOrders =  
 customers.  
 Join(orders, c => c.CustomerID, o => o.CustomerID,  
 (c, o) => new { c.Name, o.OrderDate, o.Total }  
 );

In a C# 3.0 query expression, a join clause translates to an invocation of Join. The example above is equivalent to the translation of

var custOrders =  
 from c in customers  
 join o in orders on c.CustomerID equals o.CustomerID  
 select new { c.Name, o.OrderDate, o.Total };

### GroupJoin

The GroupJoin operator performs a grouped join of two sequences based on matching keys extracted from the elements.

public static IEnumerable<V> GroupJoin<T, U, K, V>(  
 this IEnumerable<T> outer,  
 IEnumerable<U> inner,  
 Func<T, K> outerKeySelector,  
 Func<U, K> innerKeySelector,  
 Func<T, IEnumerable<U>, V> resultSelector);

The GroupJoin operator allocates and returns an enumerable object that captures the arguments passed to the operator. An ArgumentNullException is thrown if any argument is null.

The outerKeySelector and innerKeySelector arguments specify functions that extract the join key values from elements of the outer and inner sequences, respectively. The resultSelector argument specifies a function that creates a result element from an outer sequence element and its matching inner sequence elements.

When the object returned by GroupJoin is enumerated, it first enumerates the inner sequence and evaluates the innerKeySelector function once for each inner element, collecting the elements by their keys in a hash table. Once all inner elements and keys have been collected, the outer sequence is enumerated. For each outer element, the outerKeySelector function is evaluated, the resulting key is used to look up the corresponding inner elements in the hash table, the resultSelector function is evaluated for the outer element and the (possibly empty) sequence of matching inner elements, and the resulting object is yielded.

The GroupJoin operator preserves the order of the outer sequence elements, and for each outer element, the order of the matching inner sequence elements.

The GroupJoin operator produces hierarchical results (outer elements paired with sequences of matching inner elements) and has no direct equivalent in traditional relational database terms.

The following example performs a grouped join of customers with their orders, producing a sequence of tuples with customer name and total of all orders:

var custTotalOrders =  
 customers.  
 Join(orders, c => c.CustomerID, o => o.CustomerID,  
 (c, co) => new { c.Name, TotalOrders = co.Sum(o => o.Total) }  
 );

In a C# 3.0 query expression, a join…into clause translates to an invocation of GroupJoin. The example above is equivalent to the translation of

var custTotalOrders =  
 from c in customers  
 join o in orders on c.CustomerID equals o.CustomerID into co  
 select new { c.Name, TotalOrders = co.Sum(o => o.Total) };

The GroupJoin operator implements a superset of inner joins and left outer joins—both can be written in terms of grouped joins. For example, the inner join

var custTotalOrders =  
 from c in customers  
 join o in orders on c.CustomerID equals o.CustomerID  
 select new { c.Name, o.OrderDate, o.Total };

can be written as a grouped join followed by an iteration of the grouped orders

var custTotalOrders =  
 from c in customers  
 join o in orders on c.CustomerID equals o.CustomerID into co  
 from o in co  
 select new { c.Name, o.OrderDate, o.Total };

The query can be turned into a left outer join by applying the DefaultIfEmpty operator to the grouped orders

var custTotalOrders =  
 from c in customers  
 join o in orders on c.CustomerID equals o.CustomerID into co  
 from o in co.DefaultIfEmpty(emptyOrder)  
 select new { c.Name, o.OrderDate, o.Total };

where emptyOrder is an Order instance used to represent a missing order.

## Concatenation operator

### Concat

The Concat operator concatenates two sequences.

public static IEnumerable<T> Concat<T>(  
 this IEnumerable<T> first,  
 IEnumerable<T> second);

The Concat operator allocates and returns an enumerable object that captures the arguments passed to the operator. An ArgumentNullException is thrown if either argument is null.

When the object returned by Concat is enumerated, it enumerates the first sequence, yielding each element, and then enumerates the second sequence, yielding each element.

The following example extracts all distinct locations from the addresses of all customers:

IEnumerable<string> locations =  
 customers.Select(c => c.City).  
 Concat(customers.Select(c => c.Region)).  
 Concat(customers.Select(c => c.Country)).  
 Distinct();

An alternate way of concatenating sequences is to construct a sequence of sequences (such as an array of sequences) and apply the SelectMany operator with an identity selector function. For example:

IEnumerable<string> locations =  
 new[] {  
 customers.Select(c => c.City),  
 customers.Select(c => c.Region),  
 customers.Select(c => c.Country),  
 }.  
 SelectMany(s => s).  
 Distinct();

## Ordering operators

### OrderBy / ThenBy

The OrderBy / ThenBy family of operators order a sequence according to one or more keys.

public static OrderedSequence<T> OrderBy<T, K>(  
 this IEnumerable<T> source,  
 Func<T, K> keySelector);

public static OrderedSequence<T> OrderBy<T, K>(  
 this IEnumerable<T> source,  
 Func<T, K> keySelector,  
 IComparer<K> comparer);

public static OrderedSequence<T> OrderByDescending<T, K>(  
 this IEnumerable<T> source,  
 Func<T, K> keySelector);

public static OrderedSequence<T> OrderByDescending<T, K>(  
 this IEnumerable<T> source,  
 Func<T, K> keySelector,  
 IComparer<K> comparer);

public static OrderedSequence<T> ThenBy<T, K>(  
 this OrderedSequence<T> source,  
 Func<T, K> keySelector);

public static OrderedSequence<T> ThenBy<T, K>(  
 this OrderedSequence<T> source,  
 Func<T, K> keySelector,  
 IComparer<K> comparer);

public static OrderedSequence<T> ThenByDescending<T, K>(  
 this OrderedSequence<T> source,  
 Func<T, K> keySelector);

public static OrderedSequence<T> ThenByDescending<T, K>(  
 this OrderedSequence<T> source,  
 Func<T, K> keySelector,  
 IComparer<K> comparer);

The OrderBy, OrderByDescending, ThenBy, and ThenByDescending operators make up a family of operators that can be composed to order a sequence by multiple keys. A composition of the operators has the form

source . OrderBy(...) . ThenBy(...) . ThenBy(...) ...

where OrderBy(...) is an invocation of OrderBy or OrderByDescending and each ThenBy(...), if any, is an invocation of ThenBy or ThenByDescending. The initial OrderBy or OrderByDescending establishes the primary ordering, the first ThenBy or ThenByDescending establishes the secondary ordering, the second ThenBy or ThenByDescending establishes the tertiary ordering, and so on. Each ordering is defined by:

* A keySelector function that extracts the key value, of type K, from an element, of type T.
* An optional comparer for comparing key values. If no comparer is specified of if the comparer argument is null, the default comparer, Comparer<K>.Default, is used.
* A sort direction. The OrderBy and ThenBy methods establish an ascending ordering, the OrderByDescending and ThenByDescending methods establish a descending ordering.

An invocation of OrderBy, OrderByDescending, ThenBy, or ThenByDescending allocates and returns an enumerable object of type OrderedSequence<T> that captures the arguments passed to the operator. An ArgumentNullException is thrown if the source or keySelector argument is null. The OrderedSequence<T> class implements IEnumerable<T>, but otherwise introduces no public members.

When the object returned by one of the operators is enumerated, it first enumerates source, collecting all elements; then evaluates the keySelector function(s) once for each element, collecting the key values to order by; then sorts the elements according to the collected key values and the characteristics of each ordering; and finally, yields the elements in the resulting order.

The OrderBy / ThenBy operators performs an unstable sort; that is, if the key values of two elements are equal, the order of the elements might not be preserved. In contrast, a stable sort preserves the order of elements that have equal key values.

The following example creates a sequence of all products ordered first by category, then by descending price, and then by name.

IEnumerable<Product> orderedProducts1 =  
 products.  
 OrderBy(p => p.Category).  
 ThenByDescending(p => p.UnitPrice).  
 ThenBy(p => p.Name);

In a C# 3.0 query expression, an orderby clause translates to invocations of OrderBy, OrderByDescending, ThenBy, and ThenByDescending. The example above is equivalent to the translation of

IEnumerable<Product> orderedProducts1 =  
 from p in products  
 orderby p.Category, p.UnitPrice descending, p.Name  
 select p;

The following example creates a sequence of all beverage products ordered by case insensitive name:

IEnumerable<Product> orderedProducts2 =  
 products.  
 Where(p => p.Category == "Beverages").  
 OrderBy(p => p.Name, StringComparer.CurrentCultureIgnoreCase);

To order a sequence by the values of the elements themselves, specify the identity key selector x => x. For example:

IEnumerable<string> orderedProductNames =  
 products.  
 Where(p => p.Category == "Beverages").  
 Select(p => p.Name).  
 OrderBy(x => x);

### Reverse

The Reverse operator reverses the elements of a sequence.

public static IEnumerable<T> Reverse<T>(  
 this IEnumerable<T> source);

The Reverse operator allocates and returns an enumerable object that captures the source argument. An ArgumentNullException is thrown if the source argument is null.

When the object returned by Reverse is enumerated, it enumerates the source sequence, collecting all elements, and then yields the elements of the source sequence in reverse order.

## Grouping operators

### GroupBy

The GroupBy operator groups the elements of a sequence.

public static IEnumerable<IGrouping<K, T>> GroupBy<T, K>(  
 this IEnumerable<T> source,  
 Func<T, K> keySelector);

public static IEnumerable<IGrouping<K, T>> GroupBy<T, K>(  
 this IEnumerable<T> source,  
 Func<T, K> keySelector,  
 IEqualityComparer<K> comparer);

public static IEnumerable<IGrouping<K, E>> GroupBy<T, K, E>(  
 this IEnumerable<T> source,  
 Func<T, K> keySelector,  
 Func<T, E> elementSelector);

public static IEnumerable<IGrouping<K, E>> GroupBy<T, K, E>(  
 this IEnumerable<T> source,  
 Func<T, K> keySelector,  
 Func<T, E> elementSelector,  
 IEqualityComparer<K> comparer);

public interface IGrouping<K, T> : IEnumerable<T>  
{  
 K Key { get; }  
}

The GroupBy operator allocates and returns an enumerable object that captures the arguments passed to the operator. The comparer argument, if present, may be null. An ArgumentNullException is thrown if any other argument is null.

The keySelector argument specifies a function that extracts the key value from a source element. The elementSelector argument, if present, specifies a function that maps a source element to a destination element. If no elementSelector is specified, the source elements become the destination elements.

When the object returned by GroupBy is enumerated, it enumerates source and evaluates the keySelector and elementSelector (if present) functions once for each source element. Once all key and destination element pairs have been collected, a sequence of IGrouping<K, E> instances are yielded. Each IGrouping<K, E> instance represents a sequence of destination elements with a particular key value. The groupings are yielded in the order that their key values first occurred in the source sequence, and destination elements within a grouping are yielded in the order their source elements occurred in the source sequence. When creating the groupings, key values are compared using the given comparer, or, if a null comparer was specified, using the default equality comparer, EqualityComparer<K>.Default.

The following example groups all products by category:

IEnumerable<IGrouping<string, Product>> productsByCategory =  
 products.GroupBy(p => p.Category);

The following example groups all product names by product category:

IEnumerable<IGrouping<string, string>> productNamesByCategory =  
 products.GroupBy(p => p.Category, p => p.Name);

In a C# 3.0 query expression, a group…by clause translates to an invocation of GroupBy. The example above is equivalent to the translation of

IEnumerable<IGrouping<string, string>> productNamesByCategory =  
 from p in products  
 group p.Name by p.Category;

Note that the element and key selection expressions occur in the opposite order of the GroupBy operator.

## Set operators

**Note**

The May 2006 Technology Preview implementations of Distinct, Union, Intersect, and Except do not support null elements, and an ArgumentNullException is thrown if any element of a source sequence is null. The final product will not have this limitation.

### Distinct

The Distinct operator eliminates duplicate elements from a sequence.

public static IEnumerable<T> Distinct<T>(  
 this IEnumerable<T> source);

The Distinct operator allocates and returns an enumerable object that captures the source argument. An ArgumentNullException is thrown if the source argument is null.

When the object returned by Distinct is enumerated, it enumerates the source sequence, yielding each element that hasn’t previously been yielded. Elements are compared using their GetHashCode and Equals methods.

The following example produces a sequence of all product categories:

IEnumerable<string> productCategories =  
 products.Select(p => p.Category).Distinct();

### Union

The Union operator produces the set union of two sequences.

public static IEnumerable<T> Union<T>(  
 this IEnumerable<T> first,  
 IEnumerable<T> second);

The Union operator allocates and returns an enumerable object that captures the arguments passed to the operator. An ArgumentNullException is thrown if any argument is null.

When the object returned by Union is enumerated, it enumerates the first and second sequences, in that order, yielding each element that hasn’t previously been yielded. Elements are compared using their GetHashCode and Equals methods.

### Intersect

The Intersect operator produces the set intersection of two sequences.

public static IEnumerable<T> Intersect<T>(  
 this IEnumerable<T> first,  
 IEnumerable<T> second);

The Intersect operator allocates and returns an enumerable object that captures the arguments passed to the operator. An ArgumentNullException is thrown if any argument is null.

When the object returned by Intersect is enumerated, it enumerates the first sequence, collecting all distinct elements of that sequence. It then enumerates the second sequence, marking those elements that occur in both sequences. It finally yields the marked elements in the order in which they were collected. Elements are compared using their GetHashCode and Equals methods.

### Except

The Except operator produces the set difference between two sequences.

public static IEnumerable<T> Except<T>(  
 this IEnumerable<T> first,  
 IEnumerable<T> second);

The Except operator allocates and returns an enumerable object that captures the arguments passed to the operator. An ArgumentNullException is thrown if any argument is null.

When the object returned by Except is enumerated, it enumerates the first sequence, collecting all distinct elements of that sequence. It then enumerates the second sequence, removing those elements that were also contained in the first sequence. It finally yields the remaining elements in the order in which they were collected. Elements are compared using their GetHashCode and Equals methods.

## Conversion operators

### ToSequence

The ToSequence operator returns its argument typed as IEnumerable<T>.

public static IEnumerable<T> ToSequence<T>(  
 this IEnumerable<T> source);

The ToSequence operator simply returns the source argument. The operator has no effect other than to change the compile-time type of the source argument to IEnumerable<T>.

The ToSequence operator can be used to choose between query operator implementations in cases where a collection implements IEnumerable<T> but also has a different set of public query operators. For example, given a class Table<T> that implements IEnumerable<T> as well as its own Where, Select, SelectMany, and so on, the query

Table<Customer> custTable = GetCustomersTable();  
var query = custTable.Where(c => IsGoodCustomer(c));

will invoke the public Where operator of Table<T>. A Table<T> type that represents a database table would likely have a Where operator that takes the predicate argument as an expression tree and converts the tree into SQL for remote execution. If remote execution is not desired, for example because the predicate invokes a local method, the ToSequence operator can be used to hide Table<T>’s operators and instead make the Standard Query Operators available:

Table<Customer> custTable = GetCustomersTable();  
var query = custTable.ToSequence().Where(c => IsGoodCustomer(c));

This would now cause the query to execute locally.

### ToArray

The ToArray operator creates an array from a sequence.

public static T[] ToArray<T>(  
 this IEnumerable<T> source);

The ToArray operator enumerates the source sequence and returns an array containing the elements of the sequence. An ArgumentNullException is thrown if the source argument is null.

The following example produces an array of the names of all countries in which there are customers:

string[] customerCountries =  
 customers.Select(c => c.Country).Distinct().ToArray();

### ToList

The ToList operator creates a List<T> from a sequence.

public static List<T> ToList<T>(  
 this IEnumerable<T> source);

The ToList operator enumerates the source sequence and returns a List<T> containing the elements of the sequence. An ArgumentNullException is thrown if the source argument is null.

The following example produces a List<Customer> containing those customers that placed orders in 2005:

List<Customer> customersWithOrdersIn2005 =  
 customers.  
 Where(c => c.Orders.Any(o => o.OrderDate.Year == 2005)).  
 ToList();

### ToDictionary

The ToDictionary operator creates a Dictionary<K,E> from a sequence.

public static Dictionary<K, T> ToDictionary<T, K>(  
 this IEnumerable<T> source,  
 Func<T, K> keySelector);

public static Dictionary<K, T> ToDictionary<T, K>(  
 this IEnumerable<T> source,  
 Func<T, K> keySelector,  
 IEqualityComparer<K> comparer);

public static Dictionary<K, E> ToDictionary<T, K, E>(  
 this IEnumerable<T> source,  
 Func<T, K> keySelector,  
 Func<T, E> elementSelector);

public static Dictionary<K, E> ToDictionary<T, K, E>(  
 this IEnumerable<T> source,  
 Func<T, K> keySelector,  
 Func<T, E> elementSelector,  
 IEqualityComparer<K> comparer);

The ToDictionary operator enumerates the source sequence and evaluates the keySelector and elementSelector functions for each element to produce that element’s key and value. The resulting key and value pairs are returned in a Dictionary<K,E>. If no elementSelector was specified, the value for each element is simply the element itself. An ArgumentNullException is thrown if the source, keySelector, or elementSelector argument is null or if a key value produced by keySelector is null. An ArgumentException is thrown if keySelector produces a duplicate key value for two elements. In the resulting dictionary, key values are compared using the given comparer, or, if a null comparer was specified, using the default equality comparer, EqualityComparer<K>.Default.

The following example creates a Dictionary<int,Order> that maps from order ID to order for all orders in 2005:

Dictionary<int,Order> orders =  
 customers.  
 SelectMany(c => c.Orders).  
 Where(o => o.OrderDate.Year == 2005).  
 ToDictionary(o => o.OrderID);

The following example creates a Dictionary<string,decimal> that maps from category name to the maximum product price in that category:

Dictionary<string,decimal> categoryMaxPrice =  
 products.  
 GroupBy(p => p.Category).  
 ToDictionary(g => g.Key, g => g.Group.Max(p => p.UnitPrice));

### ToLookup

The ToLookup operator creates a Lookup<K, T> from a sequence.

public static Lookup<K, T> ToLookup<T, K>(  
 this IEnumerable<T> source,  
 Func<T, K> keySelector);

public static Lookup<K, T> ToLookup<T, K>(  
 this IEnumerable<T> source,  
 Func<T, K> keySelector,  
 IEqualityComparer<K> comparer);

public static Lookup<K, E> ToLookup<T, K, E>(  
 this IEnumerable<T> source,  
 Func<T, K> keySelector,  
 Func<T, E> elementSelector);

public static Lookup<K, E> ToLookup<T, K, E>(  
 this IEnumerable<T> source,  
 Func<T, K> keySelector,  
 Func<T, E> elementSelector,  
 IEqualityComparer<K> comparer);

public class Lookup<K, T> : IEnumerable<IGrouping<K, T>>  
{  
 public int Count { get; }  
 public IEnumerable<T> this[K key] { get; }  
 public bool Contains(K key);  
 public IEnumerator<IGrouping<K, T>> GetEnumerator();  
}

Lookup<K, T> implements a one-to-many dictionary that maps keys to sequences of values. This contrasts with Dictionary<K, T> which implements a one-to-one dictionary that maps keys to single values. The functionality provided by Lookup<K, T> is used in the implementations of the Join, GroupJoin, and GroupBy operators.

The ToLookup operator enumerates the source sequence and evaluates the keySelector and elementSelector functions for each element to produce that element’s key and value. The resulting key and value pairs are returned in a Lookup<K,E>. If no elementSelector was specified, the value for each element is simply the element itself. An ArgumentNullException is thrown if the source, keySelector, or elementSelector argument is null. When creating the Lookup<K, E>, key values are compared using the given comparer, or, if a null comparer was specified, using the default equality comparer, EqualityComparer<K>.Default.

The following example creates a Lookup<string, Product> that maps from category name to the sequence of products in that category:

Lookup<string,Product> productsByCategory =  
 products.ToLookup(p => p.Category);

IEnumerable<Product> beverages = productsByCategory["Beverage"];

### OfType

The OfType operator filters the elements of a sequence based on a type.

public static IEnumerable<T> OfType<T>(  
 this IEnumerable source);

The OfType operator allocates and returns an enumerable object that captures the source argument. An ArgumentNullException is thrown if the source argument is null.

When the object returned by OfType is enumerated, it enumerates the source sequence and yields those elements that are of type T. Specifically, each element e for which e is T evaluates to true is yielded by evaluating (T)e.

Given a class Employee that inherits from a class Person, the following example returns all employees from a list of persons:

List<Person> persons = GetListOfPersons();  
IEnumerable<Employee> employees = persons.OfType<Employee>();

### Cast

The Cast operator casts the elements of a sequence to a given type.

public static IEnumerable<T> Cast<T>(  
 this IEnumerable source);

The Cast operator allocates and returns an enumerable object that captures the source argument. An ArgumentNullException is thrown if the source argument is null.

When the object returned by Cast is enumerated, it enumerates the source sequence and yields each element cast to type T. An InvalidCastException is thrown if an element in the sequence cannot be cast to type T.

The Cast operator can be used to bridge between non-generic collections and the Standard Query Operators. For example, the non-generic ArrayList doesn’t implement IEnumerable<T>, but the Cast operator can be used to supply the missing type information:

ArrayList objects = GetOrders();  
IEnumerable<Order> ordersIn2005 =  
 objects.  
 Cast<Order>().  
 Where(o => o.OrderDate.Year == 2005);

In a C# 3.0 query expression, an explicitly typed iteration variable translates to an invocation of Cast. The example above is equivalent to the translation of

ArrayList objects = GetOrders();  
IEnumerable<Order> ordersIn2005 =  
 from Order o in objects  
 where o.OrderDate.Year == 2005  
 select o;

## Equality operator

### EqualAll

The EqualAll operator checks whether two sequences are equal.

public static bool EqualAll<T>(  
 this IEnumerable<T> first,  
 IEnumerable<T> second);

The EqualAll operator enumerates the two source sequences in parallel and compares corresponding elements using the Equals static method in System.Object. The method returns true if all corresponding elements compare equal and the two sequences are of equal length. Otherwise, the method returns false. An ArgumentNullException is thrown if either argument is null.

## Element operators

### First

The First operator returns the first element of a sequence.

public static T First<T>(  
 this IEnumerable<T> source);

public static T First<T>(  
 this IEnumerable<T> source,  
 Func<T, bool> predicate);

The First operator enumerates the source sequence and returns the first element for which the predicate function returns true. If no predicate function is specified, the First operator simply returns the first element of the sequence.

An ArgumentNullException is thrown if any argument is null. An InvalidOperationException is thrown if no element matches the predicate or if the source sequence is empty.

The following example returns the first customer with a given phone number:

string phone = "206-555-1212";  
Customer c = customers.First(c => c.Phone == phone);

In the example above, an exception is thrown if no customer with the given phone number exists. To instead return null when no element is found, use the FirstOrDefault operator.

### FirstOrDefault

The FirstOrDefault operator returns the first element of a sequence, or a default value if no element is found.

public static T FirstOrDefault<T>(  
 this IEnumerable<T> source);

public static T FirstOrDefault<T>(  
 this IEnumerable<T> source,  
 Func<T, bool> predicate);

The FirstOrDefault operator enumerates the source sequence and returns the first element for which the predicate function returns true. If no predicate function is specified, the FirstOrDefault operator simply returns the first element of the sequence.

An ArgumentNullException is thrown if any argument is null. If no element matches the predicate or if the source sequence is empty, default(T) is returned. The default value for reference and nullable types is null.

### Last

The Last operator returns the last element of a sequence.

public static T Last<T>(  
 this IEnumerable<T> source);

public static T Last<T>(  
 this IEnumerable<T> source,  
 Func<T, bool> predicate);

The Last operator enumerates the source sequence and returns the last element for which the predicate function returned true. If no predicate function is specified, the Last operator simply returns the last element of the sequence.

An ArgumentNullException is thrown if any argument is null. An InvalidOperationException is thrown if no element matches the predicate or if the source sequence is empty.

### LastOrDefault

The LastOrDefault operator returns the last element of a sequence, or a default value if no element is found.

public static T LastOrDefault<T>(  
 this IEnumerable<T> source);

public static T LastOrDefault<T>(  
 this IEnumerable<T> source,  
 Func<T, bool> predicate);

The LastOrDefault operator enumerates the source sequence and returns the last element for which the predicate function returned true. If no predicate function is specified, the LastOrDefault operator simply returns the last element of the sequence.

An ArgumentNullException is thrown if any argument is null. If no element matches the predicate or if the source sequence is empty, default(T) is returned. The default value for reference and nullable types is null.

### Single

The Single operator returns the single element of a sequence.

public static T Single<T>(  
 this IEnumerable<T> source);

public static T Single<T>(  
 this IEnumerable<T> source,  
 Func<T, bool> predicate);

The Single operator enumerates the source sequence and returns the single element for which the predicate function returned true. If no predicate function is specified, the Single operator simply returns the single element of the sequence.

An ArgumentNullException is thrown if any argument is null. An InvalidOperationException is thrown if the source sequence contains no matching element or more than one matching element.

The following example returns the single customer with a given customer ID:

int id = 12345;  
Customer c = customers.Single(c => c.CustomerID == id);

In the example above, an exception is thrown if no customer or more than one customer with the given ID exists. To instead return null when no element is found, use the SingleOrDefault operator.

### SingleOrDefault

The SingleOrDefault operator returns the single element of a sequence, or a default value if no element is found.

public static T SingleOrDefault<T>(  
 this IEnumerable<T> source);

public static T SingleOrDefault<T>(  
 this IEnumerable<T> source,  
 Func<T, bool> predicate);

The SingleOrDefault operator enumerates the source sequence and returns the single element for which the predicate function returned true. If no predicate function is specified, the SingleOrDefault operator simply returns the single element of the sequence.

An ArgumentNullException is thrown if any argument is null. An InvalidOperationException is thrown if the source sequence contains more than one matching element. If no element matches the predicate or if the source sequence is empty, default(T) is returned. The default value for reference and nullable types is null.

### ElementAt

The ElementAt operator returns the element at a given index in a sequence.

public static T ElementAt<T>(  
 this IEnumerable<T> source,  
 int index);

The ElementAt operator first checks whether the source sequence implements IList<T>. If so, the source sequence’s implementation of IList<T> is used to obtain the element at the given index. Otherwise, the source sequence is enumerated until index elements have been skipped, and the element found at that position in the sequence is returned. An ArgumentNullException is thrown if the source argument is null. An ArgumentOutOfRangeException is thrown if the index is less than zero or greater than or equal to the number of elements in the sequence.

The following example obtains the third most expensive product:

Product thirdMostExpensive =  
 products.OrderByDescending(p => p.UnitPrice).ElementAt(2);

### ElementAtOrDefault

The ElementAtOrDefault operator returns the element at a given index in a sequence, or a default value if the index is out of range.

public static T ElementAtOrDefault<T>(  
 this IEnumerable<T> source,  
 int index);

The ElementAtOrDefault operator first checks whether the source sequence implements IList<T>. If so, the source sequence’s implementation of IList<T> is used to obtain the element at the given index. Otherwise, the source sequence is enumerated until index elements have been skipped, and the element found at that position in the sequence is returned. An ArgumentNullException is thrown if the source argument is null. If the index is less than zero or greater than or equal to the number of elements in the sequence, default(T) is returned. The default value for reference and nullable types is null.

### DefaultIfEmpty

The DefaultIfEmpty operator supplies a default element for an empty sequence.

public static IEnumerable<T> DefaultIfEmpty<T>(  
 this IEnumerable<T> source);

public static IEnumerable<T> DefaultIfEmpty<T>(  
 this IEnumerable<T> source,  
 T defaultValue);

The DefaultIfEmpty operator allocates and returns an enumerable object that captures the arguments passed to the operator. An ArgumentNullException is thrown if the source argument is null.

When the object returned by DefaultIfEmpty is enumerated, it enumerates the source sequence and yields its elements. If the source sequence is empty, a single element with the given default value is yielded. If no default value argument is specified, default(T) is yielded in place of an empty sequence. The default value for reference and nullable types is null.

The DefaultIfEmpty operator can be combined with a grouping join to produce a left outer join. See §1.6.2 for an example.

## Generation operators

### Range

The Range operator generates a sequence of integral numbers.

public static IEnumerable<int> Range(  
 int start,  
 int count);

The Range operator allocates and returns an enumerable object that captures the arguments. An ArgumentOutOfRangeException is thrown if count is less than zero or if start + count – 1 is larger than int.MaxValue. When the object returned by Range is enumerated, it yields count sequential integers starting with the value start.

The following example produces an array of the squares of the numbers from 0 to 99:

int[] squares = Sequence.Range(0, 100).Select(x => x \* x).ToArray();

### Repeat

The Repeat operator generates a sequence by repeating a value a given number of times.

public static IEnumerable<T> Repeat<T>(  
 T element,  
 int count);

The Repeat operator allocates and returns an enumerable object that captures the arguments. An ArgumentOutOfRangeException is thrown if the specified count is less than zero. When the object returned by Repeat is enumerated, it yields count occurrences of element.

The following example produces a long[] with 256 elements containing the value -1.

long[] x = Sequence.Repeat(-1L, 256).ToArray();

### Empty

The Empty operator returns an empty sequence of a given type.

public static IEnumerable<T> Empty<T>();

The Empty operator caches a single empty sequence of the given type. When the object returned by Empty is enumerated, it yields nothing.

The following obtains an empty sequence of customers:

IEnumerable<Customer> noCustomers = Sequence.Empty<Customer>();

## Quantifiers

### Any

The Any operator checks whether any element of a sequence satisfies a condition.

public static bool Any<T>(  
 this IEnumerable<T> source);

public static bool Any<T>(  
 this IEnumerable<T> source,  
 Func<T, bool> predicate);

The Any operator enumerates the source sequence and returns true if any element satisfies the test given by the predicate. If no predicate function is specified the Any operator simply returns true if the source sequence contains any elements.

The enumeration of the source sequence is terminated as soon as the result is known.

An ArgumentNullException is thrown if any argument is null.

The following example checks whether any products with a price of 100 or more are out of stock.

bool b = products.Any(p => p.UnitPrice >= 100 && p.UnitsInStock == 0);

### All

The All operator checks whether all elements of a sequence satisfy a condition.

public static bool All<T>(  
 this IEnumerable<T> source,  
 Func<T, bool> predicate);

The All operator enumerates the source sequence and returns true if no element fails the test given by the predicate.

The enumeration of the source sequence is terminated as soon as the result is known.

An ArgumentNullException is thrown if any argument is null.

The All operator returns true for an empty sequence. This is consistent with established predicate logic and other query languages such as SQL.

The following example produces the names of the product categories for which all products are in stock:

IEnumerable<string> fullyStockedCategories =  
 products.  
 GroupBy(p => p.Category).  
 Where(g => g.Group.All(p => p.UnitsInStock > 0)).  
 Select(g => g.Key);

### Contains

The Contains operator checks whether a sequence contains a given element.

public static bool Contains<T>(  
 this IEnumerable<T> source,  
 T value);

The Contains operator first checks whether the source sequence implements ICollection<T>. If so, the Contains method in sequence’s implementation of ICollection<T> is invoked to obtain the result. Otherwise, the source sequence is enumerated to determine if it contains an element with the given value. If a matching element is found, the enumeration of the source sequence is terminated at that point. The elements and the given value are compared using the default equality comparer, EqualityComparer<K>.Default.

An ArgumentNullException is thrown if the source argument is null.

## Aggregate operators

### Count

The Count operator counts the number of elements in a sequence.

public static int Count<T>(  
 this IEnumerable<T> source);

public static int Count<T>(  
 this IEnumerable<T> source,  
 Func<T, bool> predicate);

The Count operator without a predicate first checks whether the source sequence implements ICollection<T>. If so, the sequence’s implementation of ICollection<T> is used to obtain the element count. Otherwise, the source sequence is enumerated to count the number of elements.

The Count operator with a predicate enumerates the source sequence and counts the number of elements for which the predicate function returns true.

For both Count operators, an ArgumentNullException is thrown if any argument is null, and an OverflowException is thrown if the count exceeds int.MaxValue.

The following example returns the number of customers in London:

int count = customers.Count(c => c.City == "London");

### LongCount

The LongCount operator counts the number of elements in a sequence.

public static long LongCount<T>(  
 this IEnumerable<T> source);

public static long LongCount<T>(  
 this IEnumerable<T> source,  
 Func<T, bool> predicate);

The LongCount operator enumerates the source sequence and counts the number of elements for which the predicate function returns true. If no predicate function is specified the LongCount operator simply counts all elements. The count of elements is returned as a value of type long.

For both Count operators, an ArgumentNullException is thrown if any argument is null.

### Sum

The Sum operator computes the sum of a sequence of numeric values.

public static *Numeric* Sum(  
 this IEnumerable<*Numeric*> source);

public static *Numeric* Sum<T>(  
 this IEnumerable<T> source,  
 Func<T, *Numeric*> selector);

The Numeric type is one of int, int?, long, long?, double, double?, decimal, and decimal?.

The Sum operator enumerates the source sequence, invokes the selector function for each element, and computes the sum of the resulting values. If no selector function is specified, the sum of the elements themselves is computed. An ArgumentNullException is thrown if any argument is null. If the sum is too large to represent using the Numeric type, an OverflowException is thrown or, for double and double?, a positive or negative infinity is returned.

The Sum operator returns zero for an empty sequence. Furthermore, the operator does not include null values in the result (null values can occur when the Numeric type is a nullable type).

The following example produces a sequence of customer names and order totals for a given year:

int year = 2005;  
var namesAndTotals =  
 customers.  
 Select(c => new {  
 c.Name,  
 TotalOrders =  
 c.Orders.  
 Where(o => o.OrderDate.Year == year).  
 Sum(o => o.Total)  
 });

### Min

The Min operator finds the minimum of a sequence of numeric values.

public static *Numeric* Min(  
 this IEnumerable<*Numeric*> source);

public static T Min<T>(  
 this IEnumerable<T> source);

public static *Numeric* Min<T>(  
 this IEnumerable<T> source,  
 Func<T, *Numeric*> selector);

public static S Min<T, S>(  
 this IEnumerable<T> source,  
 Func<T, S> selector);

The Numeric type is one of int, int?, long, long?, double, double?, decimal, and decimal?.

The Min operator enumerates the source sequence, invokes the selector function for each element, and finds the minimum of the resulting values. If no selector function is specified, the minimum of the elements themselves is computed. The values are compared using their implementation of the IComparable<T> interface, or, if the values do not implement that interface, the non-generic IComparable interface. An ArgumentNullException is thrown if any argument is null.

The Min implementations for the Numeric types are optimizations of the more general generic operators. Furthermore, the Min operators for int?, long?, double?, and decimal? return null if the source sequence is empty or contains only null values. The other Min operators throw an InvalidOperationException if the source sequence is empty.

The following example produces a sequence of name and lowest product price for each product category:

var minPriceByCategory =  
 products.  
 GroupBy(p => p.Category).  
 Select(g => new {  
 Category = g.Key,  
 MinPrice = g.Group.Min(p => p.UnitPrice)  
 });

### Max

The Max operator finds the maximum of a sequence of numeric values.

public static *Numeric* Max(  
 this IEnumerable<*Numeric*> source);

public static T Max<T>(  
 this IEnumerable<T> source);

public static *Numeric* Max<T>(  
 this IEnumerable<T> source,  
 Func<T, *Numeric*> selector);

public static S Max<T, S>(  
 this IEnumerable<T> source,  
 Func<T, S> selector);

The Numeric type is one of int, int?, long, long?, double, double?, decimal, and decimal?.

The Max operator enumerates the source sequence, invokes the selector function for each element, and finds the maximum of the resulting values. If no selector function is specified, the maximum of the elements themselves is computed. The values are compared using their implementation of the IComparable<T> interface, or, if the values do not implement that interface, the non-generic IComparable interface. An ArgumentNullException is thrown if any argument is null.

The Max implementations for the Numeric types are optimizations of the more general generic operators. Furthermore, the Max operators for int?, long?, double?, and decimal? return null if the source sequence is empty or contains only null values. The other Max operators throw an InvalidOperationException if the source sequence is empty.

The following example finds the total of the largest order in 2005:

decimal largestOrder =  
 customers.  
 SelectMany(c => c.Orders).  
 Where(o => o.OrderDate.Year == 2005).  
 Max(o => o.Total);

### Average

The Average operator computes the average of a sequence of numeric values.

public static *Result* Average(  
 this IEnumerable<*Numeric*> source);

public static *Result* Average<T>(  
 this IEnumerable<T> source,  
 Func<T, *Numeric*> selector);

The Numeric type is one of int, int?, long, long?, double, double?, decimal, and decimal?. When the Numeric type is int or long, the Result type is double. When the Numeric type is int? or long?, the Result type is double?. Otherwise, the Numeric and Result types are the same.

The Average operator enumerates the source sequence, invokes the selector function for each element, and computes the average of the resulting values. If no selector function is specified, the average of the elements themselves is computed. An ArgumentNullException is thrown if any argument is null.

For the Numeric types int, int?, long, or long?, if the sum of the elements is too large to represent in a long, an OverflowException is thrown. For the Numeric types decimal and decimal?, if the sum of the elements is too large to represent in a decimal, an OverflowException is thrown.

The following example computes the average order total for each customer:

var averageOrderTotals =  
 customers.  
 Select(c => new {  
 c.Name,  
 AverageOrderTotal = c.Orders.Average(o => o.Total)  
 });

### Aggregate

The Aggregate operator applies a function over a sequence.

public static T Aggregate<T>(  
 this IEnumerable<T> source,  
 Func<T, T, T> func);

public static U Aggregate<T, U>(  
 this IEnumerable<T> source,  
 U seed,  
 Func<U, T, U> func);

The Aggregate operator with a seed value starts by assigning the seed value to an internal accumulator. It then enumerates the source sequence, repeatedly computing the next accumulator value by invoking the specified function with the current accumulator value as the first argument and the current sequence element as the second argument. The final accumulator value is returned as the result. An ArgumentNullException is thrown if the source or func argument is null.

The Aggregate operator without a seed value uses the first element of the source sequence as the seed value, but otherwise functions as described above. If the source sequence is empty, the Aggregate operator without a seed value throws an InvalidOperationException.

The following example produces a sequence of category name and longest product name for each product category:

var longestNamesByCategory =  
 products.  
 GroupBy(p => p.Category).  
 Select(g => new {  
 Category = g.Key,  
 LongestName =  
 g.Group.  
 Select(p => p.Name).  
 Aggregate((s, t) => t.Length > s.Length ? t : s)  
 });