5a

Introduction to LINQ and Generic Collections

OBJECTIVES

In this lecture you will learn:

- Basic LINQ concepts.
- How to query an array using LINQ.
- Basic .NET collections concepts.
- How to create and use a generic `List` collection.
- How to write a generic method.
- How to query a generic `List` collection using LINQ
- Use Lambda expressions.

(read **Chapter 20**, **Visual C# 2012 Step By Step**

and **Chapter 19**, **Illustrated C# 2012**)

Outline

9.1 Introduction

9.2 Querying an Array Using LINQ

9.3 Introduction to Collections

9.4 Querying a Generic Collection Using LINQ

9.5 Lambda expressions and Lambda statements

9.1 Introduction

- Although commonly used, arrays have limited capabilities.
- `Lists` are similar to arrays but provide additional functionality, such as **dynamic resizing**.
- Traditionally, programs used **SQL queries** to access a database.
- C#'s new **LINQ** (**Language-Integrated Query**) capabilities allow you to write **query expressions** that retrieve information from many data sources, not just databases.
- **LINQ to Objects** can be used to **filter** arrays and `Lists`, selecting elements that satisfy a set of conditions
- A **LINQ provider** is a set of classes that implement LINQ operations and enable programs to interact with data sources to perform tasks such as projecting, sorting, grouping and filtering elements.

- Figure 9.2 demonstrates querying an array of integers using LINQ.

LINQWithSimpleTypeArray.cs

(1 of 5)

```
// Fig. 9.2: LINQWithSimpleTypeArray.cs
// LINQ to Objects using an Integer array.
using System;
using System.Linq;
using System.Collections.Generic;

class LINQWithSimpleTypeArray
{
   public static void Main( string[] args )
   {
      // create an integer array
      int[] values = { 2, 9, 5, 0, 3, 7, 1, 4, 8, 5 };

      Display( values, "Original array:" ); // display original values

```

Fig. 9.2 | LINQ to Objects using an `int` array. (Part 1 of 5.)

Outline

LINQWithSimpleTypeArray.cs

(2 of 5)

```
      // LINQ query that obtains values greater than 4 from the array
      var filtered =
         from value in values
         where value > 4
         select value;

      // display filtered results
      Display( filtered, "Array values greater than 4:" );

      // use orderby clause to sort original array in ascending order
      var sorted =
         from value in values
         orderby value
         select value;

      // display sorted results
      Display( sorted, "Original array, sorted:" );

```

A LINQ query begins with a `from` **clause**, which specifies a **range variable** (`value`) and the data source to query (`values`).

If the condition in the `where` **clause** evaluates to `true`, the element is selected.

The `select` **clause** determines what value appears in the results.

The `orderby` **clause** sorts the query results in ascending order.

Fig. 9.2 | LINQ to Objects using an `int` array. (Part 2 of 5.)

LINQWithSimpleTypeArray.cs

(3 of 5)

```
      // sort the filtered results into descending order
      var sortFilteredResults =
         from value in filtered
         orderby value descending
         select value;

      // display the sorted results
      Display( sortFilteredResults,
         "Values greater than 4, descending order (separately):" );

      // filter original array and sort in descending order
      var sortAndFilter =
         from value in values
         where value > 4
         orderby value descending
         select value;

```

The `descending` modifier in the `orderby` clause sorts the results in descending order.

Fig. 9.2 | LINQ to Objects using an `int` array. (Part 3 of 5.)

LINQWithSimpleTypeArray.cs

(4 of 5)

```
         // display the filtered and sorted results
         Display( sortAndFilter,
            "Values greater than 4, descending order (one query):" );
      } // end Main

      // display a sequence of integers with the specified header
      public static void Display(
         IEnumerable< int > results, string header )
      {
         Console.Write( "{0}", header ); // display header

         // display each element, separated by spaces
         foreach ( var element in results )
            Console.Write( " {0}", element );
```

Fig. 9.2 | LINQ to Objects using an `int` array. (Part 4 of 5.)

LINQWithSimpleTypeArray.cs

(5 of 5)

```

      Console.WriteLine(); // add end of line
   } // end method Display
} // end class LINQWithSimpleTypeArray
```

```
Original array: 2 9 5 0 3 7 1 4 8 5
Array values greater than 4: 9 5 7 8 5
Original array, sorted: 0 1 2 3 4 5 5 7 8 9
Values greater than 4, descending order (separately): 9 8 7 5 5
Values greater than 4, descending order (one query): 9 8 7 5 5
```

Fig. 9.2 | LINQ to Objects using an `int` array. (Part 5 of 5.)

9.2 Querying an Array Using LINQ (Cont.)

- Repetition statements that filter arrays focus on the steps required to get the results. This is called **imperative programming**.
- LINQ queries, however, specify the conditions that selected elements must satisfy. This is known as **declarative programming**.
- The `System.Linq` namespace contains the LINQ to Objects provider.

9.2 Querying an Array Using LINQ (Cont.)

- A LINQ query begins with a **from clause**, which specifies a **range variable** (`value`) and the data source to query (`values`).
  - The range variable represents each item in the data source, much like the control variable in a `foreach` statement.
- If the condition in the **`where clause`** evaluates to `true`, the element is selected.
- A **predicate** is an expression that takes an element of a collection and returns `true` or `false` by testing a condition on that element.
- The **select clause** determines what value appears in the results.

9.2 Querying an Array Using LINQ (Cont.)

- The `Display` method takes an `IEnumerable<int>` object as an argument.
  - The type `int` enclosed in angle brackets after the type name indicates that this `IEnumerable` may only hold integers.
  - Any type may be used as a **type argument** in this manner—types can be passed as arguments to **generic types** just as objects are passed as arguments to methods.

9.2 Querying an Array Using LINQ (Cont.)

- **Interfaces** define and standardize the ways in which people and systems can interact with one another.
- A C# **interface describes a set of methods that can be called on an object**.
- A class that **implements an interface** must define each method in the interface with a signature identical to the one in the interface definition.

9.2 Querying an Array Using LINQ (Cont.)

- The `IEnumerable<T>` interface describes the functionality of any object that can be **iterated over** and thus **offers methods to access each element**.
- **Arrays and collections** already implement the `IEnumerable<T>` interface.
- A LINQ query **returns** an object that implements the `IEnumerable<T>` interface.
- With LINQ, the code that **selects elements** and the **code that displays them** **are kept separate**, making the code easier to understand and maintain.

9.2 Querying an Array Using LINQ (Cont.)

- The **orderby** **clause** sorts the query results in ascending order.
- The **descending** modifier in the `orderby` clause sorts the results in descending order.
- Any value that can be **compared** with other values of the same type may be used with the `orderby` clause.

9.2 Querying an Array Using LINQ (Cont.)

- LINQ is not limited to querying arrays of primitive types such as `integers`.
- **Comparable** types in .NET are those that implement the `IComparable<T>`.
- All **built-in types**, such as `string`, `int` and double implement `IComparable<T>`.

- Figure 9.3 presents the `Employee` class.

`Employee.cs`

(1 of 3)

```
// Fig. 9.3: Employee.cs
// Employee class with FirstName, LastName and MonthlySalary properties.
public class Employee
{
   private decimal monthlySalaryValue; // monthly salary of employee

   // auto-implemented property FirstName
   public string FirstName { get; set; }

   // auto-implemented property LastName
   public string LastName { get; set; }

   // constructor initializes first name, last name and monthly salary
   public Employee( string first, string last, decimal salary )
   {
```

Fig. 9.3 | `Employee` class with `FirstName`, `LastName` and `MonthlySalary` properties. (Part 1 of 3.)

Employee.cs

(2 of 3)

```
      FirstName = first;
      LastName = last;
      MonthlySalary = salary;
   } // end constructor

   // property that gets and sets the employee's monthly salary
   public decimal MonthlySalary
   {
      get
      {
         return monthlySalaryValue;
      } // end get
      set
      {
         if ( value >= 0M ) // if salary is nonnegative
         {
            monthlySalaryValue = value;
         } // end if
      } // end set
```

Fig. 9.3 | `Employee` class with `FirstName`, `LastName` and `MonthlySalary` properties. (Part 2 of 3.)

Outline

Employee.cs

(3 of 3)

```
   } // end property MonthlySalary

   // return a String containing the employee's information
   public override string ToString()
   {
      return string.Format( "{0,-10} {1,-10} {2,10:C}",
         FirstName, LastName, MonthlySalary );
   } // end method ToString
} // end class Employee
```

Fig. 9.3 | `Employee` class with `FirstName`, `LastName` and `MonthlySalary` properties. (Part 3 of 3.)

Outline

- Figure 9.4 uses LINQ to query an array of `Employee` objects.

`LINQWithArrayOfObjects.cs`

(1 of 5)

```
// Fig. 9.4: LINQWithArrayOfObjects.cs
// LINQ to Objects using an array of Employee objects.
using System;
using System.Linq;
using System.Collections.Generic;

public class LINQWithArrayOfObjects
{
   public static void Main( string[] args )
   {
```

Fig. 9.4 | LINQ to Objects using an array of Employee objects. (Part 1 of 5.)

```
      // initialize array of employees
      Employee[] employees = {
         new Employee( "Jason", "Red", 5000M ),
         new Employee( "Ashley", "Green", 7600M ),
         new Employee( "Matthew", "Indigo", 3587.5M ),
         new Employee( "James", "Indigo", 4700.77M ),
         new Employee( "Luke", "Indigo", 6200M ),
         new Employee( "Jason", "Blue", 3200M ),
         new Employee( "Wendy", "Brown", 4236.4M ) }; // end init list

      Display( employees, "Original array" ); // display all employees

      // filter a range of salaries using && in a LINQ query
      var between4K6K =
         from e in employees
         where e.MonthlySalary >= 4000M && e.MonthlySalary <= 6000M
         select e;

      // display employees making between 4000 and 6000 per month
      Display( between4K6K, string.Format(
         "Employees earning in the range {0:C}-{1:C} per month",
         4000, 6000 ) );
```

LINQWithArrayOfObjects.cs

(2 of 5)

A `where` clause can access the properties of the range variable.

Fig. 9.4 | LINQ to Objects using an array of Employee objects. (Part 2 of 5.)

LINQWithArrayOfObjects.cs

(3 of 5)

```
33
34       // order the employees by last name, then first name with LINQ
35       var nameSorted =
36          from e in employees
37          orderby e.LastName, e.FirstName
38          select e;
39
40       // header
41       Console.WriteLine( "First employee when sorted by name:" );
42
43       // attempt to display the first result of the above LINQ query
44       if ( nameSorted.Any() )
45          Console.WriteLine( nameSorted.First().ToString() + "\n" );
46       else
47          Console.WriteLine( "not found\n" );
48
49       // use LINQ to select employee last names
50       var lastNames =
51          from e in employees
52          select e.LastName;
53
```

An `orderby` clause can sort the results according to multiple properties, specified in a comma-separated list.

The query result's `Any` method returns `true` if there is at least one element, and `false` if there are no elements.

The query result's `First` method (line 45) returns the first element in the result

The `select` clause can be used to select a member of the range variable rather than the range variable itself.

Fig. 9.4 | LINQ to Objects using an array of Employee objects. (Part 3 of 5.)

```
      // use method Distinct to select unique last names
      Display( lastNames.Distinct(), "Unique employee last names" );

      // use LINQ to select first and last names
      var names =
         from e in employees
         select new { e.FirstName, Last = e.LastName };

      Display( names, "Names only" ); // display full names
   } // end Main

   // display a sequence of any type, each on a separate line
   public static void Display< T >(
      IEnumerable< T > results, string header )
   {
      Console.WriteLine( "{0}:", header ); // display header

      // display each element, separated by spaces
      foreach ( T element in results )
         Console.WriteLine( element );

      Console.WriteLine(); // add a blank line
   } // end method Display
} // end class LINQWithArrayOfObjects
```

LINQWithArrayOfObjects.cs

(4 of 5)

The **Distinct method** removes duplicate elements, causing all elements in the result to be unique.

The select clause can create a new object of **anonymous type** (a type with no name), which the compiler generates for you based on the properties listed in the curly braces ({}).

To define a generic method, you must specify a **type parameter list** which contains one or more type parameters separated by commas.

Fig. 9.4 | LINQ to Objects using an array of Employee objects. (Part 4 of 5.)

```
Original array:
Jason      Red        $5,000.00
Ashley     Green      $7,600.00
Matthew    Indigo     $3,587.50
James      Indigo     $4,700.77
Luke       Indigo     $6,200.00
Jason      Blue       $3,200.00
Wendy      Brown      $4,236.40

Employees earning in the range $4,000.00-$6,000.00 per month
Jason      Red        $5,000.00
James      Indigo     $4,700.77
Wendy      Brown      $4,236.40

First employee when sorted by name:
Jason      Blue       $3,200.00

Unique employee last names:
Red
Green
Indigo
Blue
Brown
```

```
Names only:
{ FirstName = Jason, Last = Red }
{ FirstName = Ashley, Last = Green }
{ FirstName = Matthew, Last = Indigo }
{ FirstName = James, Last = Indigo }
{ FirstName = Luke, Last = Indigo }
{ FirstName = Jason, Last = Blue }
{ FirstName = Wendy, Last = Brown }
```

**LINQWithArrayOf
Objects.cs**

(5 of 5)

Fig. 9.4 | LINQ to Objects using an array of Employee objects. (Part 5 of 5.)

9.2 Querying an Array Using LINQ (Cont.)

- A `where` clause can access the properties of the range variable.
- The conditional AND (&&) operator can be used to combine conditions.
- An `orderby` clause can sort the results according to multiple properties, specified in a comma-separated list.

9.2 Querying an Array Using LINQ (Cont.)

- The query result’s `Any` method returns `true` if there is at least one element, and `false` if there are no elements.
- The query result’s `First` method (line 45) returns **the first element** in the result.
- The `Count` method of the query result returns the number of elements in the results.
- The `select` clause can be used to **select a member of the range variable** rather than the range variable itself.
- The `Distinct method` removes duplicate elements, causing all elements in the result to be unique.

9.2 Querying an Array Using LINQ (Cont.)

- The select clause can create a new object of **anonymous type** (a type with no name), which the compiler generates for you based on the properties listed in the curly braces (**{}**).
- By **default**, the **name of the property** being selected is used as the property's name in the result.
- You can **specify a different name for the property** inside the anonymous type definition.

9.2 Querying an Array Using LINQ (Cont.)

- Implicitly typed local variables allow you to use anonymous types because you do not have to explicitly state the type when declaring such variables.
- When the compiler creates an anonymous type, it automatically generates a `ToString` method that returns a `string` representation of the object.

9.2 Querying an Array Using LINQ (Cont.)

- **Generic methods** enable you to create a single method definition that can be called with arguments of many types.
- To define a generic method, you must specify a **type parameter list** which contains one or more type parameters separated by commas.
- A **type parameter** is a placeholder for a type argument. They can be used to declare return types, parameter types and local variable types in generic method declarations.

9.2 Querying an Array Using LINQ (Cont.)

  - Can only appear once in the type-parameter list.
  - Can appear more than once in the method’s parameter list and body
  - Can be the method’s return type

- Type-parameter names must match throughout a method, but need not be unique among different generic methods.

Common Programming Error 9.1

If you forget to include the type-parameter list when declaring a generic method, the compiler will not recognize the type-parameter names when they’re encountered in the method, causing compilation errors.

9.3 Introduction to Collections

- The .NET Framework Class Library provides **collections**, which are used to store groups of related objects.
- Collections provide efficient methods that organize, store and retrieve your data without requiring knowledge of how the data is being stored.
- The collection class **`List<T>`** (from namespace `System.Collections.Generic`) does not need to be reallocated to change its size.

9.3 Introduction to Collections (Cont.)

- `List<T>` is called a **generic class** because it can be used with any type of object.
- `T` is a placeholder for the type of the objects stored in the list.
- Figure 9.5 shows some common methods and properties of class `List<T>`.

Method or property	Description
`Add`	Adds an element to the end of the `List`.
`Capacity`	Property that gets or sets the number of elements a `List` can store.
`Clear`	Removes all the elements from the `List`.
`Contains`	Returns `true` if the `List` contains the specified element; otherwise, returns `false`.
`Count`	Property that returns the number of elements stored in the `List`.

Fig. 9.5 | Some methods and properties of class `List<T>`. (Part 1 of 2.)

◀ ▶

9.3 Introduction to Collections (Cont.)

Method or property	Description
`IndexOf`	Returns the index of the first occurrence of the specified value in the `List`.
`Insert`	Inserts an element at the specified index.
`Remove`	Removes the first occurrence of the specified value.
`RemoveAt`	Removes the element at the specified index.
`RemoveRange`	Removes a specified number of elements starting at a specified index.
`Sort`	Sorts the `List`.
`TrimExcess`	Sets the Capacity of the List to the number of elements the List currently contains (`Count`).

Fig. 9.5 | Some methods and properties of class `List<T>`. (Part 2 of 2.)

- Figure 9.6 demonstrates dynamically resizing a `List` object.

`ListCollection.cs`

(1 of 4)

```
// Fig. 9.6: ListCollection.cs
// Generic List collection demonstration.
using System;
using System.Collections.Generic;

public class ListCollection
{
   public static void Main( string[] args )
   {
      // create a new List of strings
      List< string > items = new List< string >();

      items.Add( "red" ); // append an item to the List
      items.Insert( 0, "yellow" ); // insert the value at index 0

```

The `Add` method appends its argument to the end of the `List`.

The `Insert` method inserts a new element at the specified position.

Fig. 9.6 | Generic `List<T>` collection demonstration. (Part 1 of 4.)

`ListCollection.cs`

(2 of 4)

```
      // header
      Console.Write(
         "Display list contents with counter-controlled loop:" );

      // display the colors in the list
      for ( int i = 0; i < items.Count; i++ )
         Console.Write( " {0}", items[ i ] );

      // display colors using foreach in the Display method
      Display( items,
         "\nDisplay list contents with foreach statement:" );

      items.Add( "green" ); // add "green" to the end of the List
      items.Add( "yellow" ); // add "yellow" to the end of the List
      // display the List
      Display( items, "List with two new elements:" );

      items.Remove( "yellow" ); // remove the first "yellow"
      // display List
      Display( items, "Remove first instance of yellow:" );

      items.RemoveAt( 1 ); // remove item at index 1
      // display List
      Display( items, "Remove second list element (green):" );

```

`List`s can be indexed like arrays by placing the index in square brackets after the `List` variable's name.

The `Remove` method is used to remove the first instance of an element with a specific value.

`RemoveAt` removes the element at the specified index; all elements above that index are shifted down by one.

Fig. 9.6 | Generic `List<T>` collection demonstration. (Part 2 of 4.)

Outline

`ListCollection.cs`

(3 of 4)

```
      // check if a value is in the List
      Console.WriteLine( "\"red\" is {0}in the list",
         items.Contains( "red" ) ? string.Empty : "not " );

      // display number of elements in the List
      Console.WriteLine( "Count: {0}", items.Count );

      // display the capacity of the List
      Console.WriteLine( "Capacity: {0}", items.Capacity );
   } // end Main

   // display the List's elements on the console
   public static void Display( List< string > items, string header )
   {
      Console.Write( header ); // display header

      // display each element in items
      foreach ( var item in items )
         Console.Write( " {0}", item );

      Console.WriteLine(); // display end of line
   } // end method Display
} // end class ListCollection
```

The `Contains` method returns true if the element is found in the List, and false otherwise.

The `Capacity` property indicates how many items the `List` can hold without growing.

Fig. 9.6 | Generic `List<T>` collection demonstration. (Part 3 of 4.)

Outline

ListCollection.cs

(4 of 4)

```
        Console.WriteLine(); // display end of line
    } // end method Display
} // end class ListCollection
```

```
Display list contents with counter-controlled loop: yellow red
Display list contents with foreach statement: yellow red
List with two new elements: yellow red green yellow
Remove first instance of yellow: red green yellow
Remove second list element (green): red yellow
"red" is in the list
Count: 2
Capacity: 4
```

Fig. 9.6 | Generic `List<T>` collection demonstration. (Part 4 of 4.)

9.3 Introduction to Collections (Cont.)

- The `Add` method appends its argument to the end of the `List`.
- The `Insert` method inserts a new element at the specified position.
  - The first argument is an index—as with arrays, collection indices start at zero.
  - The second argument is the value that is to be inserted at the specified index.
  - All elements at the specified index and above are shifted up by one position.

9.3 Introduction to Collections (Cont.)

- The `Count` property returns the number of elements currently in the `List`.
- `Lists` can be indexed like arrays by placing the index in square brackets after the `List` variable's name.
- The `Remove` method is used to remove the first instance of an element with a specific value.
  - If no such element is in the List, `Remove` does nothing.
- `RemoveAt` removes the element at the specified index; all elements above that index are shifted down by one.

9.3 Introduction to Collections (Cont.)

- The `Contains` method returns `true` if the element is found in the `List`, and `false` otherwise.
- `Contains` compares its argument to each element of the `List` in order, so using `Contains` on a large `List` is inefficient.
- The `Capacity` property indicates how many items the `List` can hold without growing.
- `List` is implemented using an array behind the scenes. When the `List` grows, it must create a larger internal array and copy each element to the new array.
- A `List` grows only when an element is added and there is no space for the new element.
- The List **doubles** its `Capacity` each time it grows. By **default** it is 0.

◀ ▶

- You can use LINQ to Objects to query `Lists` just as arrays.
- In Fig. 9.7, a `List` of `strings` is converted to uppercase and searched for those that begin with "`R`".

LINQWithList Collection.cs

(1 of 2)

```
// Fig. 9.7: LINQWithListCollection.cs
// LINQ to Objects using a List< string >.
using System;
using System.Linq;
using System.Collections.Generic;

public class LINQWithListCollection
{
   public static void Main( string[] args )
   {
      // populate a List of strings with random case
      List< string > items = new List< string >();
      items.Add( "aQua" ); // add "aQua" to the end of the List
      items.Add( "RusT" ); // add "RusT" to the end of the List
      items.Add( "yElLow" ); // add "yElLow" to the end of the List
      items.Add( "rEd" ); // add "rEd" to the end of the List

      // convert all strings to uppercase; select those starting with "R"
      var startsWithR =
         from item in items
```

Fig. 9.7 | LINQ to Objects using a `List<string>`. (Part 1 of 2.)

```
         let uppercasedString = item.ToUpper()
         where uppercasedString.StartsWith( "R" )
         orderby uppercasedString
         select uppercasedString;

      // display query results
      foreach ( var item in startsWithR )
         Console.Write( "{0} ", item );

      Console.WriteLine(); // output end of line

      items.Add( "rUbY" ); // add "rUbY" to the end of the List
      items.Add( "SaFfRon" ); // add "SaFfRon" to the end of the List

      // display updated query results
      foreach ( var item in startsWithR )
         Console.Write( "{0} ", item );

      Console.WriteLine(); // output end of line
   } // end Main
} // end class LINQWithListCollection
```

```
RED RUST
RED RUBY RUST
```

LINQWithList Collection.cs

(2 of 2)

Fig. 9.7 | LINQ to Objects using a `List<string>`. (Part 2 of 2.)

9.4 Querying a Generic Collection Using LINQ (Cont.)

- LINQ’s `let` `clause` can be used to create a new range variable to store a temporary result for use later in the LINQ query.
- The `string` method `ToUpper` to converts a string to uppercase.
- The `string` method `StartsWith` performs a case sensitive comparison to determine whether a `string` starts with the `string` received as an argument.

9.4 Querying a Generic Collection Using LINQ (Cont.)

- LINQ uses **deferred execution**—the query executes only when you access the results, not when you define the query.
- LINQ extension methods `ToArray` and `ToList` immediately execute the query on which they are called.
  - These methods execute the query only once, improving efficiency.

◀ ▶

9.5 Lambda Expressions

The syntax for anonymous methods requires information that the compiler itself already knows.

C# 3.0 introduced *lambda expressions*, which pare down the syntax of anonymous methods.

If lambda expressions had been introduced first, there never would have been anonymous methods

Lambda Expressions

A lambda expression is an anonymous method that you can use to create delegates. By using lambda expressions, you can write local methods that can be passed as arguments or returned as the value of function calls. Lambda expressions are particularly helpful for writing LINQ query expressions.

To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator =>, and you put the expression or statement block on the other side.

Expression Lambdas

An **anonymous method** can be converted into a ***lambda expression*** by doing the following:

- Deleting the **`delegate`** keyword.
- Placing the **lambda operator`, =>`**, between the parameter list and the body of the anonymous method. (The **lambda operator** is read as "*goes to*.")

Expression Lambdas

A lambda expression with an expression on the right side of the => operator is called an *expression lambda*. **Expression lambdas** are used extensively for writing LINQ query expressions. An expression lambda returns the result of the expression and takes the following basic form:

```
(input parameters) => expression
```

The **parentheses** are optional only if the lambda has one input parameter; otherwise they **are required**. Two or more input parameters are **separated by commas** enclosed in parentheses

Expression Lambdas

Examples

```
(x, y) => x == y
```

Sometimes it is **difficult or impossible for the compiler to infer the input types**. When this occurs, you can **specify the types explicitly** as shown in the following example:

```
(int x, string s) => s.Length > x
```

Specify **zero input parameters** with **empty parentheses**:

```
() => SomeMethod()
```

Note in the previous example that **the body of an expression lambda can consist of a method call**.

Statement Lambdas

A **statement lambda** resembles an expression lambda except that **the statement(s) is enclosed in braces**:

```
(input parameters) => {statement;}
```

The body of a statement lambda can consist of any number of statements, however, **in prac**tice there are typically **no more than two or three**

Statement Lambdas

```
class Program
{
  delegate double MyDel( int par );
  delegate double MyDummy();
  static void Main()
  {   // Anonymous method
    MyDel del = delegate( int x ) {   return x + 1;    };
    MyDel le1 = ( int x ) => { return x + 1; }; // Lambda expression
    MyDel le2 =  ( x ) => { return x + 1; };    // Lambda expression
    MyDel le3 =   x   => { return x + 1; };     // Lambda expression
    MyDel le4 =   x   =>          x + 1;        // Lambda expression
    MyDummy dummy = () =>       13;      // Lambda with no arguments
    Console.WriteLine( "{0}", del( 12 ) );
    Console.WriteLine( "{0}", le1( 12 ) );
    Console.WriteLine( "{0}", le2( 12 ) );
    Console.WriteLine( "{0}", le3( 12 ) );
    Console.WriteLine( "{0}", dummy () );
  }
}
```

Statement Lambdas

If there’s only a **single, implicitly typed** parameter, you can **leave off the parentheses surrounding it,** as shown in the assignment to **`le3`**.

- **Lambda expressions** allow the **body of the expression** to be either a *statement block* or **an expression**.

If the statement block c**ontains a single `return` statement**, you can **replace the statement block with just the expression** that **follows** the **`return`** keyword, as shown in the assignment to **`le4`**.

◀ ▶

Lambdas with the Standard Query Operators

You can also **supply a lambda expression** when the **argument type** is an **`Expression<Func>`**, for example in the **standard query operators** that are defined in **`System.Linq.Queryable`**. When you specify an **`Expression<Func>`** argument, **the lambda will be compiled to an expression tree**.

A standard query operator, the **`Count`** method, is shown here:

```
int[] numbers={5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n%2 == 1);
```

Lambdas with the Standard Query Operators

The **compiler can infer the type of the input parameter**, or **you can also specify it explicitly**. This particular lambda expression counts those integers (`n`) which when divided by two have a remainder of `1`.

```
int oddNumbers = numbers.Count(n => n%2 == 1);
```

The following line of c**ode produces a sequence that contains all elements in the `numbers` array** that are to the **left side** of the `9` because that's the first number in the sequence that doesn't meet the condition:

```
var firstNumbersLessThan6 =
                     numbers.TakeWhile(n => n < 6);
```

Lambdas with the Standard Query Operators

The following example shows how to **specify multiple input parameters** by enclosing them in parentheses.

The method **returns all the elements** in the **`numbers` array** until a number `n` is encountered **whose value is less than its position `index`**.

```
var firstSmallNumbers =
 numbers.TakeWhile((n, index) => n >= index);
```

Note: Do not confuse the lambda operator (=>) with the greater than or equal operator (>=).

Variable Scope in Lambda Expressions

Lambdas can refer to ***outer variables*** (*the* ***same way*** *as* ***Anonymous methods***) that are **in scope in the method** that **defines the lambda function**, or **in scope in the type that contains the lambda expression**. Variables that are captured in this manner are **stored for use in the lambda expression even if the variables would otherwise go out of scope and be garbage collected**. An <u>**outer variable must be definitely assigned before it can be consumed in a lambda expression**</u>.

The following example demonstrates these rules.

Assume

```
 delegate bool D();
delegate bool D2(int i);
```

Variable Scope in Lambda Expressions

```
class Test
{
    D del;
    D2 del2;
    public void TestMethod(int input)
    {   // Assume input  = 5
        int j = 0;
        // Initialize the delegates with lambda expressions.
        // Note access to 2 outer variables.
        // del will be invoked within this method.
        del = () => { j = 10;  return j > input; };
        // del2 will be invoked after TestMethod goes out of scope.
        del2 = (x) => {return x == j; };
        // Demonstrate value of j:
        // Output: j = 0
        // The delegate has not been invoked yet.
        Console.WriteLine("j = {0}", j); // Invoke the delegate.
        bool boolResult = del();
        // Output: j = 10 b = True
        Console.WriteLine("j = {0}. b = {1}", j, boolResult);
    }
```

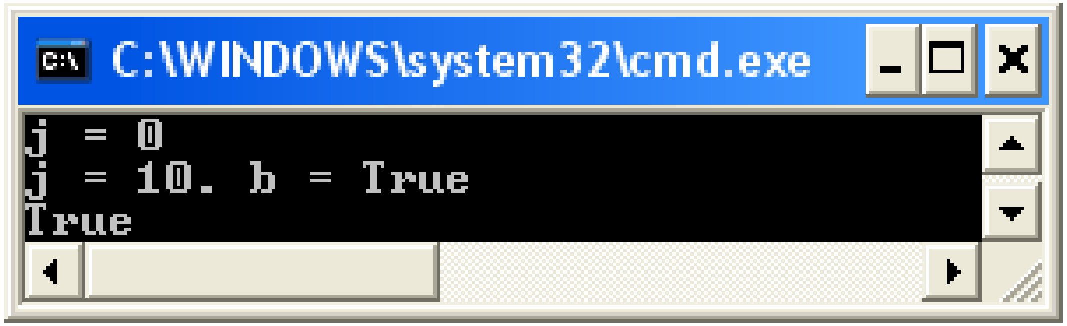
Variable Scope in Lambda Expressions

```
    static void Main()
    {
        Test test = new Test();
        test.TestMethod(5);
        // Updates to 10 the local var j in TestMethod returns b = 10 > 5
        // Prove that del2 still has a copy of
        // local variable j from TestMethod.
        bool result = test.del2(10);

        // Output: True (j == 10)
        Console.WriteLine(result);

        Console.ReadKey();
    }
}

delegate bool D();
delegate bool D2(int i);
```



```
j = 0
j = 10. b = True
True
```

Variable Scope in Lambda Expressions

The following rules apply to variable scope in lambda expressions:

- A **variable** that is captured will **not be garbage-collected** until the **delegate** that references it becomes **eligible for garbage collection**.
- **Variables** introduced **within a lambda expression** are **not visible in the outer method**.
- A lambda expression **cannot directly capture** a **ref** or **out** parameter **from an enclosing method**.
- A **return** statement in a lambda expression **does not cause the enclosing method to return**.
- A lambda expression cannot contain a **goto** statement, **break** statement, or **continue** statement that is inside the lambda function if the jump statement's target is outside the block. It is also an error to have a jump statement outside the lambda function block if the target is inside the block.