31927 32998: Application Development with .NET

Week-10 Lecture

Delegates, Anonymous methods,

Lambda and LINQ

Outline

- IEnumerable and IEnumerator
- Delegates
- Multicast Delegates
- Anonymous Methods
- Generic Delegates
- Action, Predicate and Func delegates
- LINQ introduction
- Lambda

IEnumerable<T> interface

Background:

There is a need to loop through collections of classes or list, IEnumarable interface loops over the collection.

- IEnumerable is the base interface for all non-generic collections that can be enumerated.
- IEnumerable<T> is generic version of the IEnumerable interface.
- IEnumerable<T> is the base interface for collections in the System.Collections.Generic namespace.
- The collection (Generic/non-Generic) that implements IEnumerable<T> or IEnumerable can be enumerated (or iterated) by using the foreach statement.

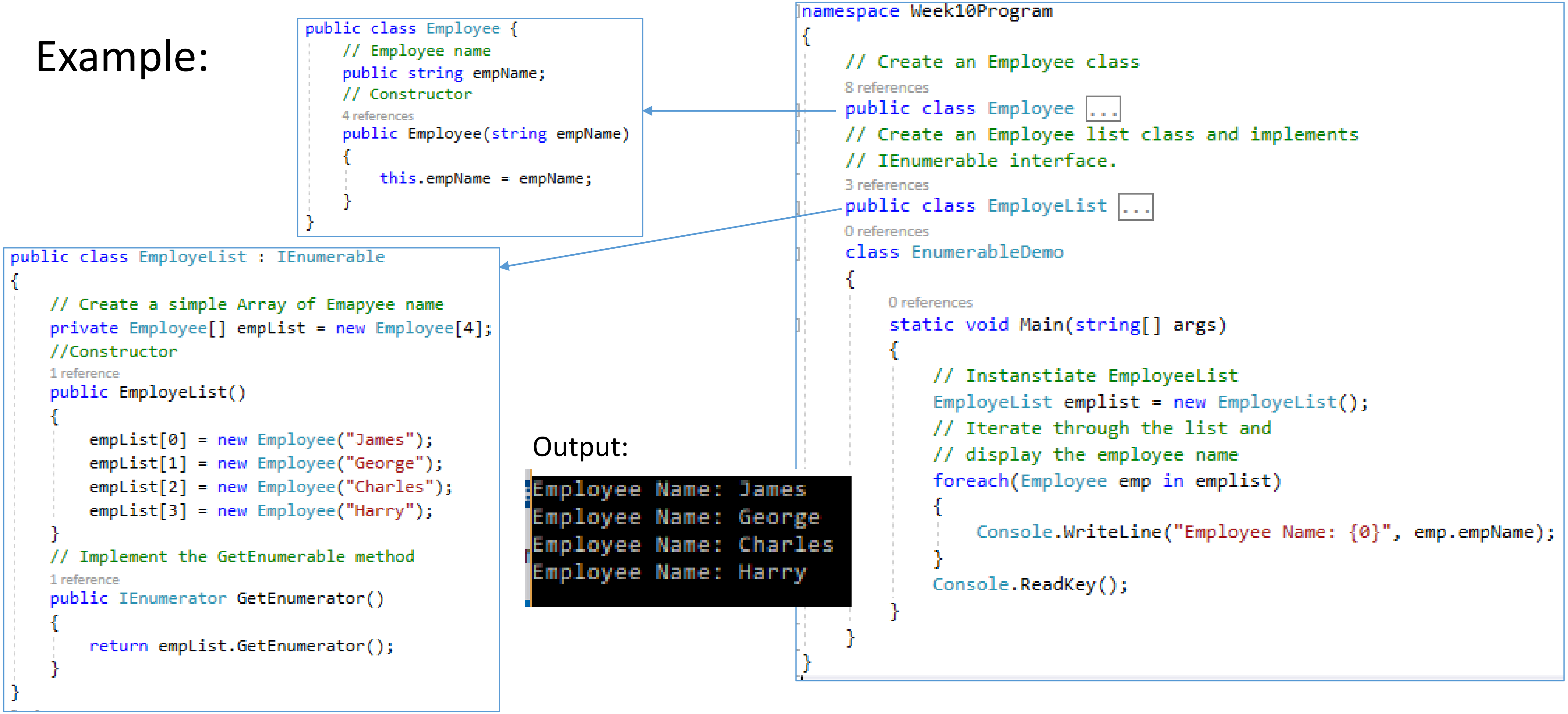
IEnumerable<T> interface

- In short, IEnumerable <T> exposes the enumerator, which supports a simple iteration over a collection of a specified type.
- IEnumerable <T> contains the following method:

Method	Description
IEnumerator<T> GetEnumerator()	Returns an enumerator for the collection

IEnumerable<T> interface

Example:



```
public class Employee {
    // Employee name
    public string empName;
    // Constructor
    4 references
    public Employee(string empName)
    {
        this.empName = empName;
    }
}
```

```
public class EmployeList : IEnumerable
{
    // Create a simple Array of Emapyee name
    private Employee[] empList = new Employee[4];
    //Constructor
    1 reference
    public EmployeList()
    {
        empList[0] = new Employee("James");
        empList[1] = new Employee("George");
        empList[2] = new Employee("Charles");
        empList[3] = new Employee("Harry");
    }
    // Implement the GetEnumerable method
    1 reference
    public IEnumerator GetEnumerator()
    {
        return empList.GetEnumerator();
    }
}
```

```
namespace Week10Program
{
    // Create an Employee class
    8 references
    public class Employee ...
    // Create an Employee list class and implements
    // IEnumerable interface.
    3 references
    public class EmployeList ...
    0 references
    class EnumerableDemo
    {
        0 references
        static void Main(string[] args)
        {
            // Instanstiate EmployeeList
            EmployeList emplist = new EmployeList();
            // Iterate through the list and
            // display the employee name
            foreach(Employee emp in emplist)
            {
                Console.WriteLine("Employee Name: {0}", emp.empName);
            }
            Console.ReadKey();
        }
    }
}
```

Output:

```
Employee Name: James
Employee Name: George
Employee Name: Charles
Employee Name: Harry
```

IEnumerator<T> interface

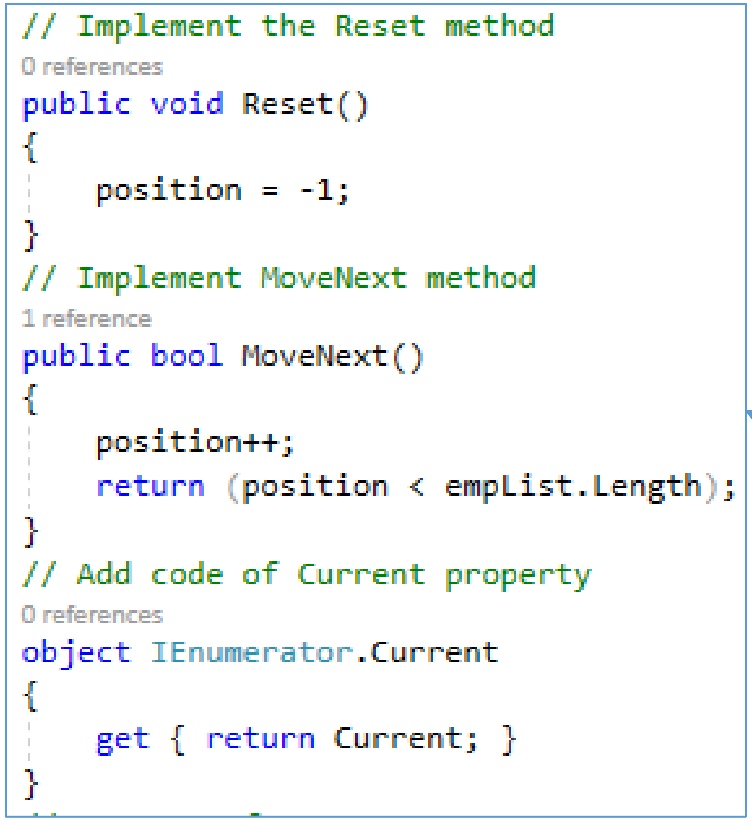
- IEnumerator is an object that points to a particular element in a collection.

- Has the following methods/properties

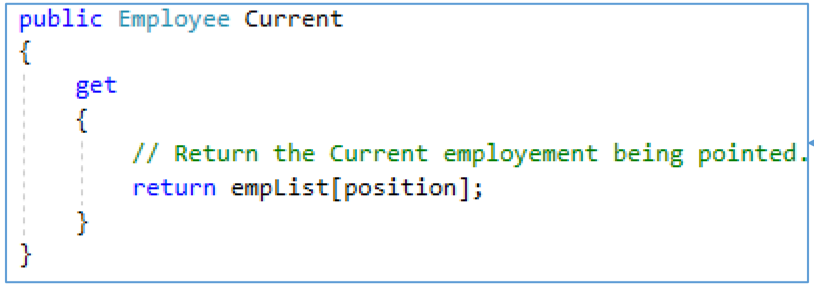
Method/Property	Description
T Current { get; }	Return the element the enumerator is pointing to.
void MoveNext()	Move enumerator to next element
void Reset()	Returns enumerator to beginning of collection, which is before the first element

IEnumerator<T> interface

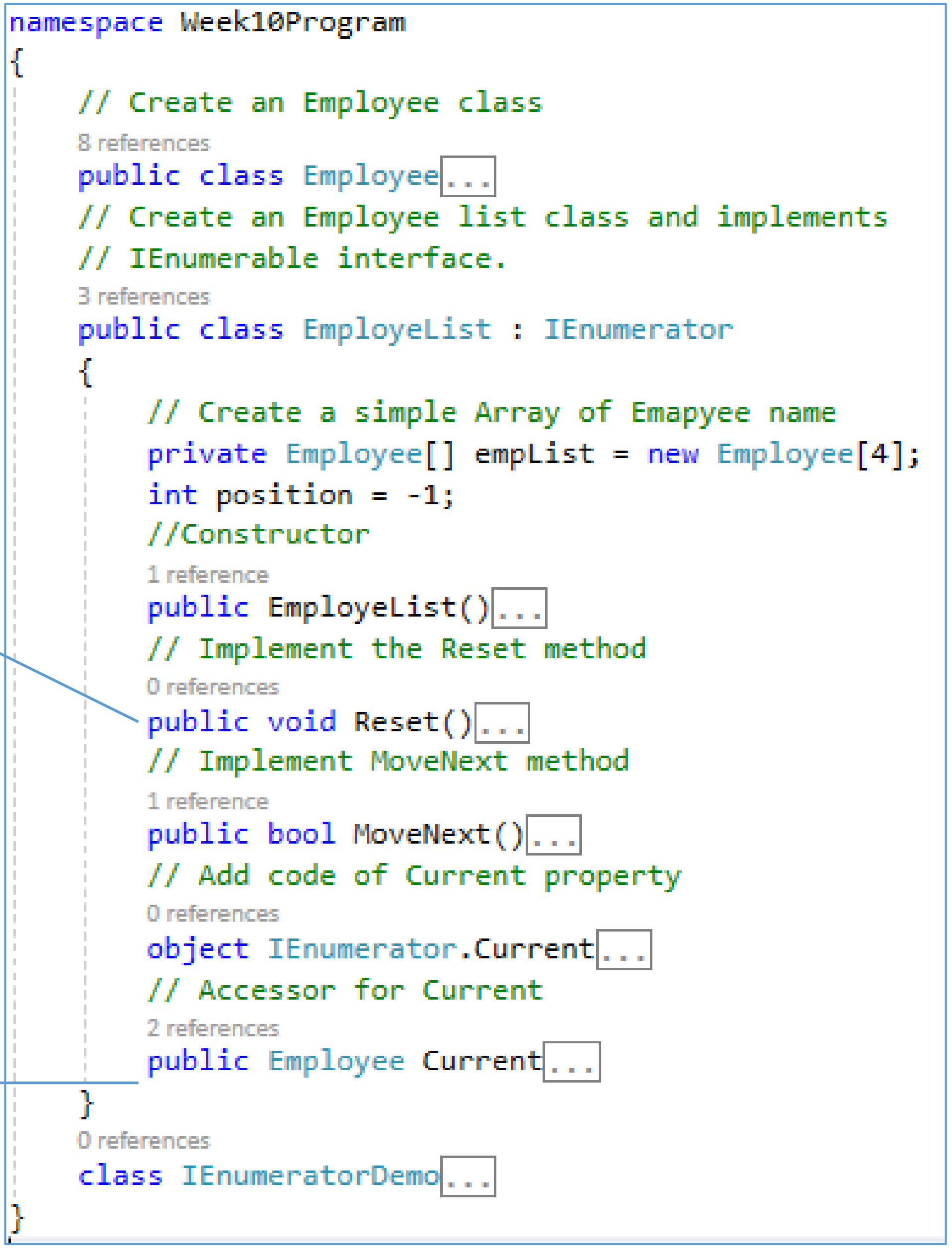
Example:



```
// Implement the Reset method
0 references
public void Reset()
{
    position = -1;
}
// Implement MoveNext method
1 reference
public bool MoveNext()
{
    position++;
    return (position < empList.Length);
}
// Add code of Current property
0 references
object IEnumerator.Current
{
    get { return Current; }
}
```



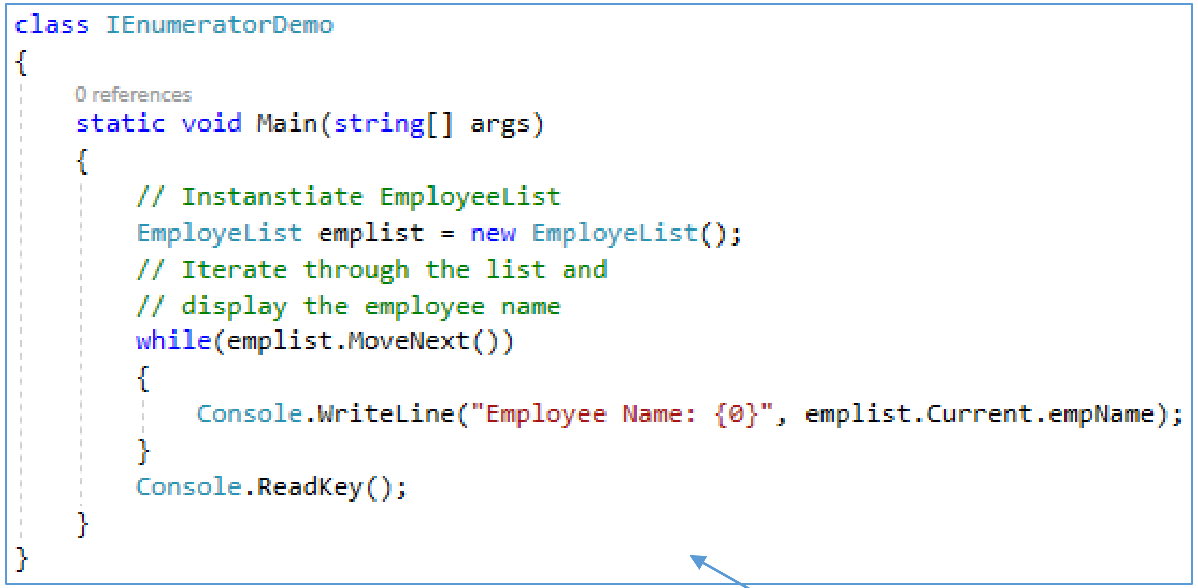
```
public Employee Current
{
    get
    {
        // Return the Current employement being pointed.
        return empList[position];
    }
}
```



```
namespace Week10Program
{
    // Create an Employee class
    8 references
    public class Employee...
    // Create an Employee list class and implements
    // IEnumerable interface.
    3 references
    public class EmployeList : IEnumerator
    {
        // Create a simple Array of Emapyee name
        private Employee[] empList = new Employee[4];
        int position = -1;
        //Constructor
        1 reference
        public EmployeList()...
        // Implement the Reset method
        0 references
        public void Reset()...
        // Implement MoveNext method
        1 reference
        public bool MoveNext()...
        // Add code of Current property
        0 references
        object IEnumerator.Current...
        // Accessor for Current
        2 references
        public Employee Current...
    }
    0 references
    class IEnumeratorDemo...
}
```

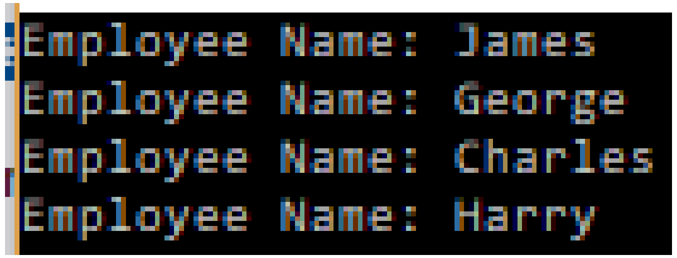
IEnumerator<T> interface

Example:

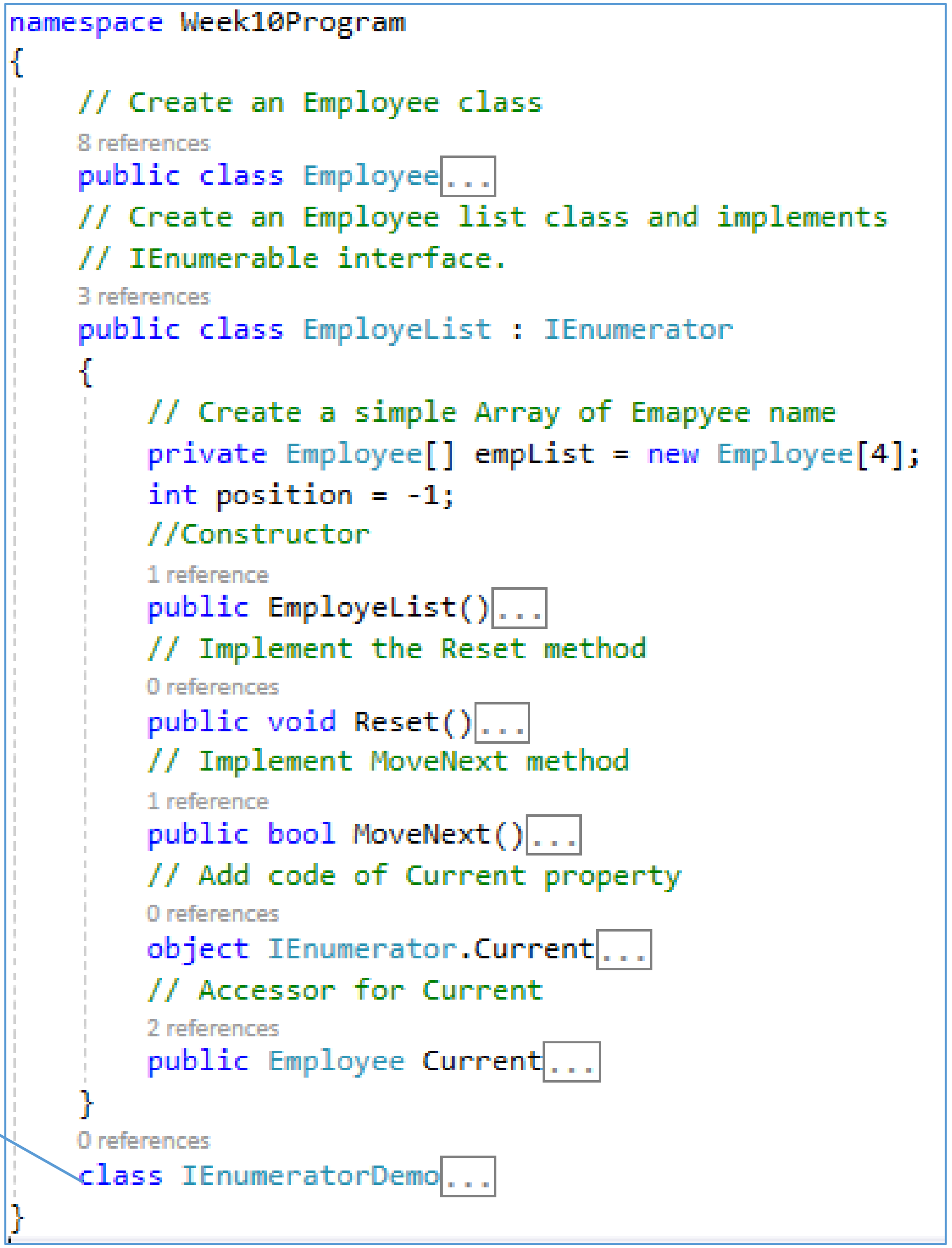


```
class IEnumeratorDemo
{
    0 references
    static void Main(string[] args)
    {
        // Instanstiate EmployeeList
        EmployeList emplist = new EmployeList();
        // Iterate through the list and
        // display the employee name
        while(emplist.MoveNext())
        {
            Console.WriteLine("Employee Name: {0}", emplist.Current.empName);
        }
        Console.ReadKey();
    }
}
```

Output:



```
Employee Name: James
Employee Name: George
Employee Name: Charles
Employee Name: Harry
```



```
namespace Week10Program
{
    // Create an Employee class
    8 references
    public class Employee...
    // Create an Employee list class and implements
    // IEnumerable interface.
    3 references
    public class EmployeList : IEnumerator
    {
        // Create a simple Array of Emapyee name
        private Employee[] empList = new Employee[4];
        int position = -1;
        //Constructor
        1 reference
        public EmployeList()...
        // Implement the Reset method
        0 references
        public void Reset()...
        // Implement MoveNext method
        1 reference
        public bool MoveNext()...
        // Add code of Current property
        0 references
        object IEnumerator.Current...
        // Accessor for Current
        2 references
        public Employee Current...
    }
    0 references
    class IEnumeratorDemo...
}
```

Delegates

- A Delegate is a type safe pointer to method/function
- They allow you to treat methods as if they are data and pass methods to other methods
- Also useful for creating “plug-in” code
- Delegates are reference types
- Delegates are especially used for implementing events and the call-back methods. All delegates are implicitly derived from the System.Delegate class

Delegates

- delegate keyword is used to create a delegate similar to creating a class!

Syntax:

<access_modifier> delegate <return_type> <delegate-name>(<parameter-list>)

- The signature of the delegate must match the signature of the function, the delegate points to, else → compiler error!
- This is the reason delegates are type safe!

Delegates

- We can create instance of a Delegate, by passing the function/method name as a parameter to the delegate constructor,

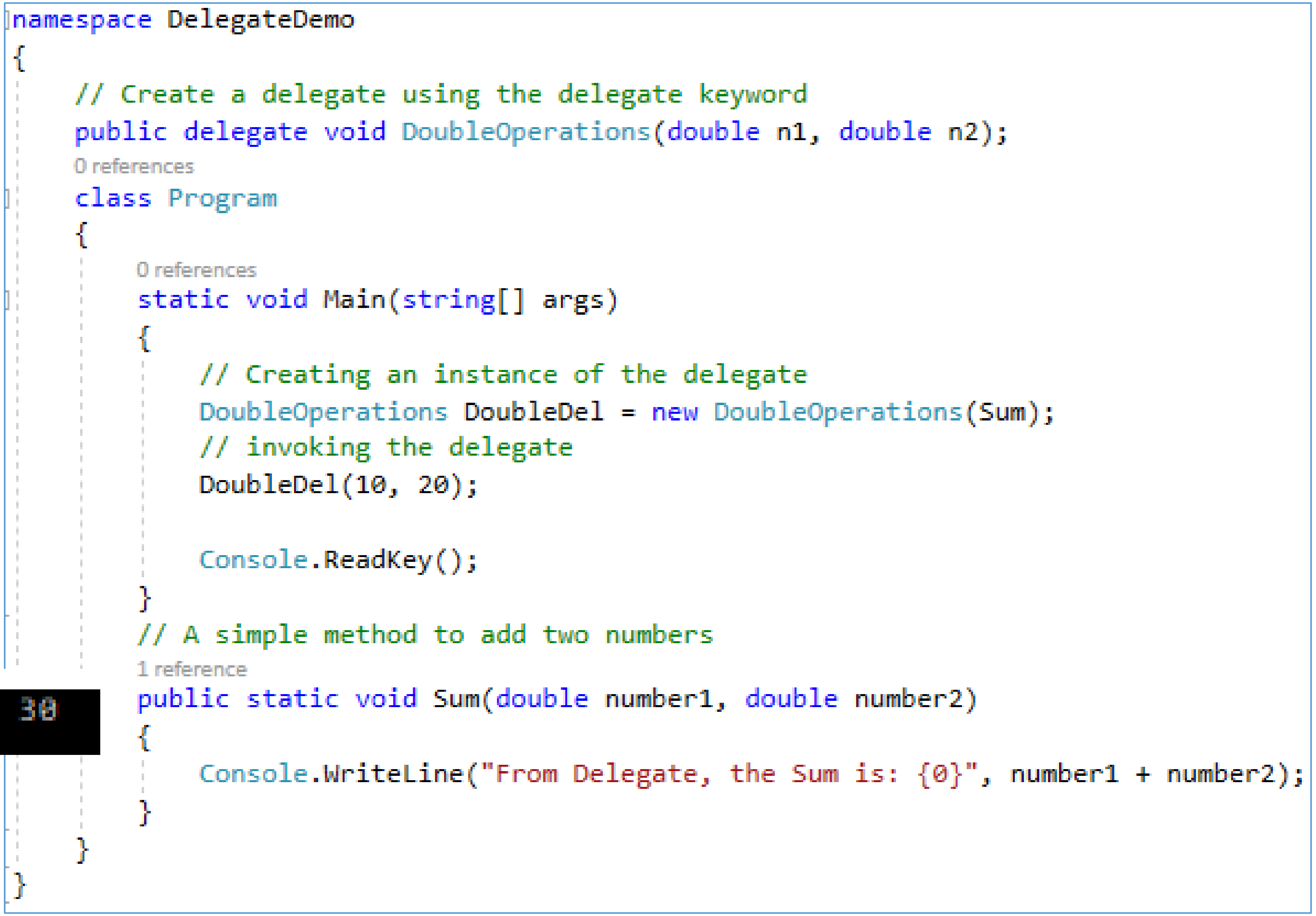
- Example:

```
public static void add (int number1, int number2){} // Function/Method

//Create a delegate

public delegate void OperationDelegate(int num1, int num2);

// Create an instance of the delegate and point to a function by passing the function name

OperationDelegate opDel = new OperationDelegate(add);

opDel(10, 20); // calling the delegate will internally invoke the add function
```

- The delegate will now point to the add function/method.

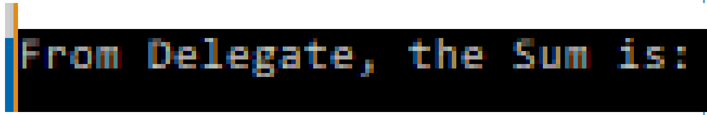
Delegates

- Example:



```
namespace DelegateDemo
{
    // Create a delegate using the delegate keyword
    public delegate void DoubleOperations(double n1, double n2);
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            // Creating an instance of the delegate
            DoubleOperations DoubleDel = new DoubleOperations(Sum);
            // invoking the delegate
            DoubleDel(10, 20);

            Console.ReadKey();
        }
        // A simple method to add two numbers
        1 reference
        public static void Sum(double number1, double number2)
        {
            Console.WriteLine("From Delegate, the Sum is: {0}", number1 + number2);
        }
    }
}
```

Output:



```
From Delegate, the Sum is: 30
```

Multi-Cast Delegates

- A Multi-cast Delegate has reference to more than one functions
- When invoking multi-cast delegate, all the functions the delegate is pointing to will be invoked.
- Can create an *invocation list* that may point to many methods
- If not a void, will return the result of the last method in the invocation list
- When called will successively call each method in order

Multi-Cast Delegates

- There are two approaches to create multi-cast delegate:

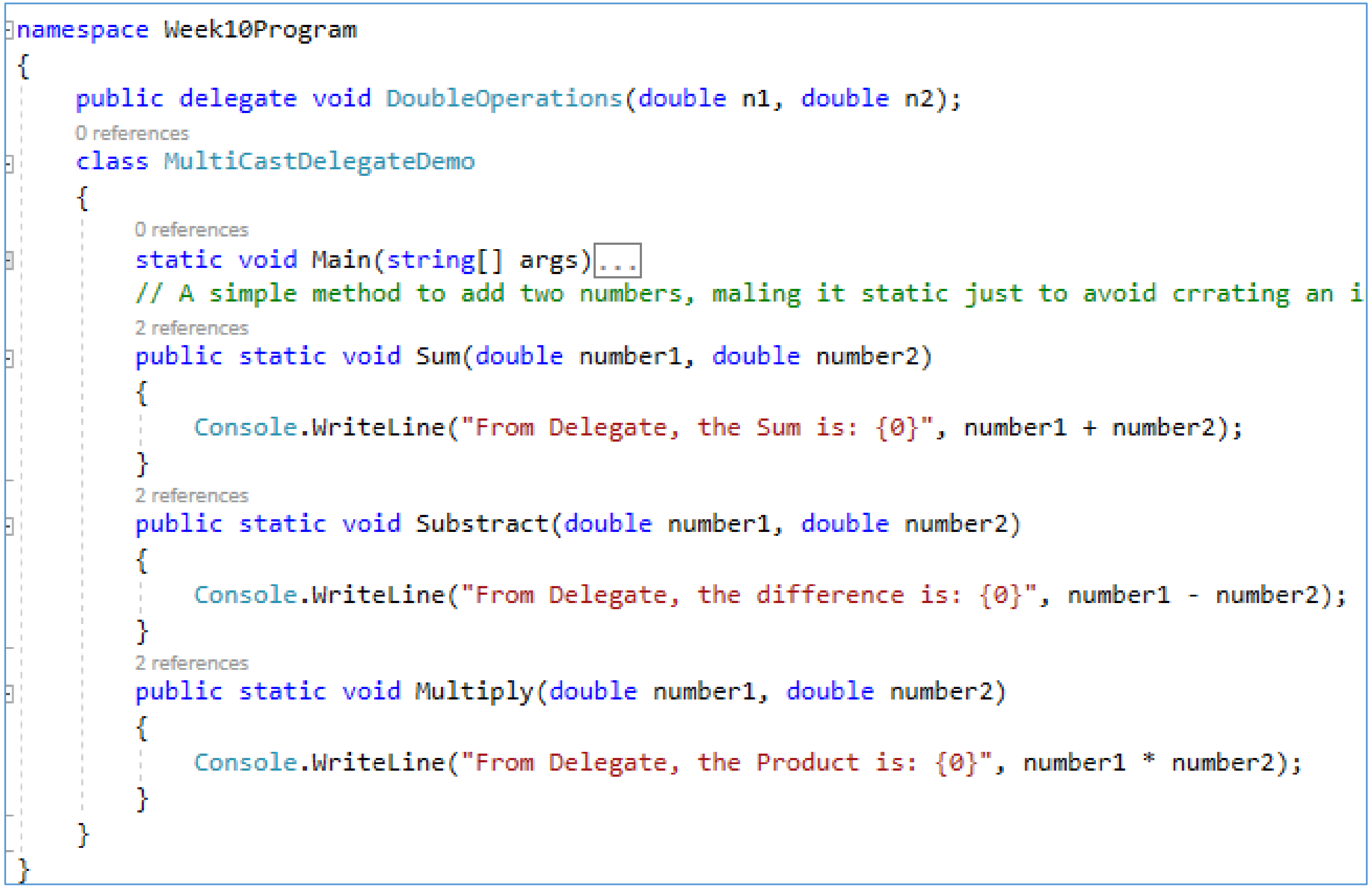
Option 1: using + to register a method with a delegate

Option 2: += to register a method with a delegate

- and -= are used to un-register a method with a delegate

Multi-Cast Delegates

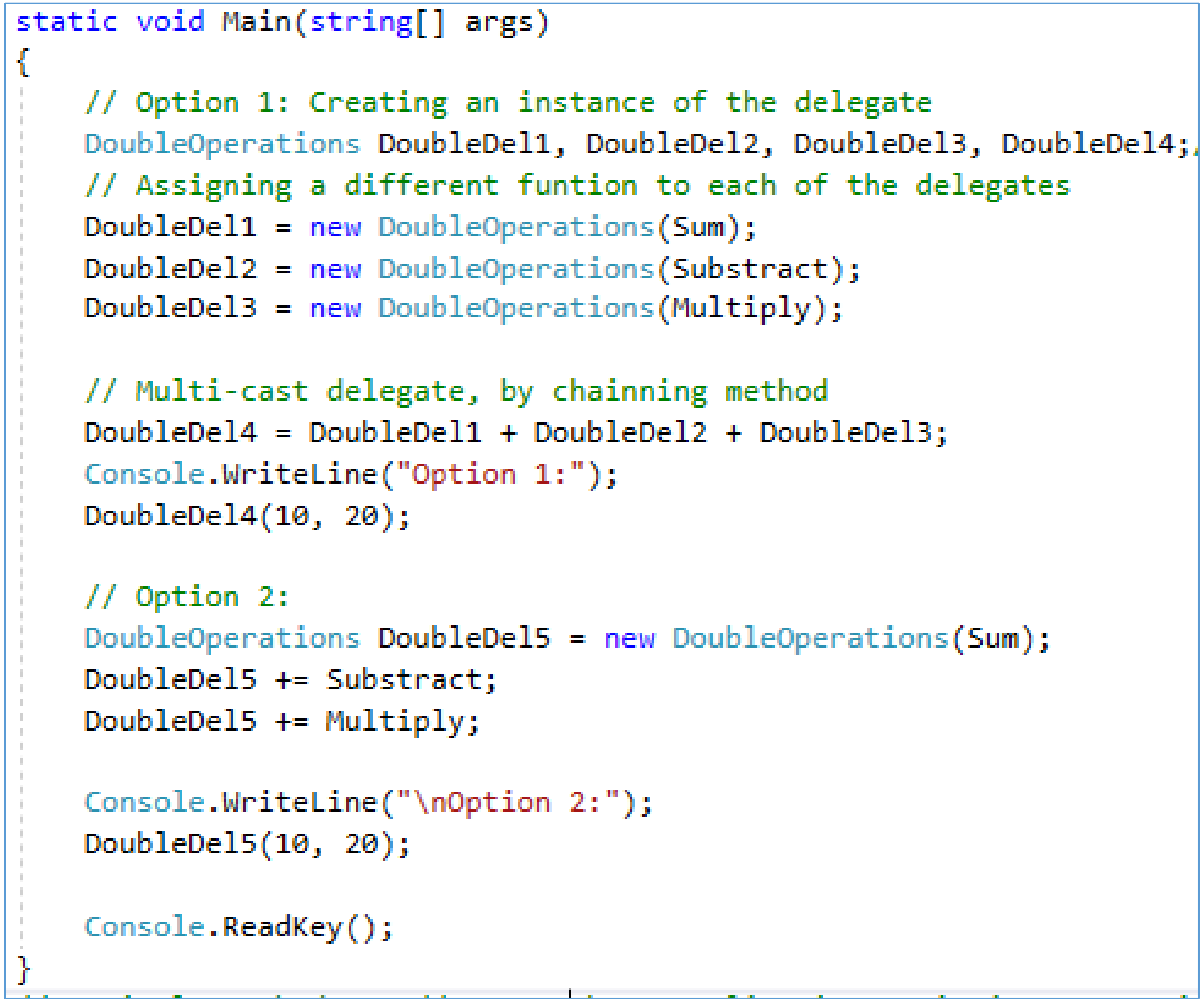
- Example:



```
namespace Week10Program
{
    public delegate void DoubleOperations(double n1, double n2);
    0 references
    class MultiCastDelegateDemo
    {
        0 references
        static void Main(string[] args)...
        // A simple method to add two numbers, maling it static just to avoid crrating an i
        2 references
        public static void Sum(double number1, double number2)
        {
            Console.WriteLine("From Delegate, the Sum is: {0}", number1 + number2);
        }
        2 references
        public static void Substract(double number1, double number2)
        {
            Console.WriteLine("From Delegate, the difference is: {0}", number1 - number2);
        }
        2 references
        public static void Multiply(double number1, double number2)
        {
            Console.WriteLine("From Delegate, the Product is: {0}", number1 * number2);
        }
    }
}
```

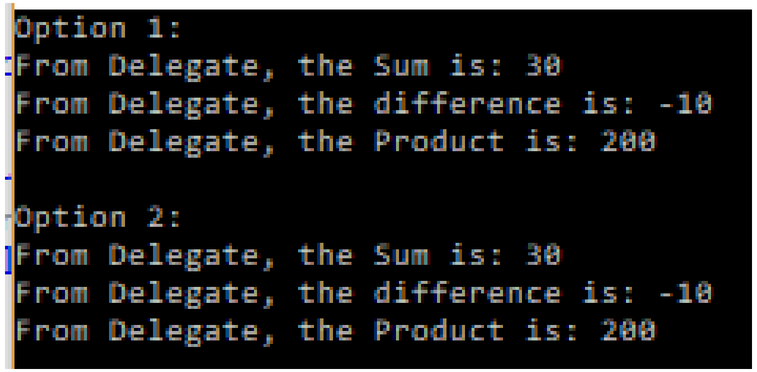
Multi-Cast Delegates

- Example:



```
static void Main(string[] args)
{
    // Option 1: Creating an instance of the delegate
    DoubleOperations DoubleDel1, DoubleDel2, DoubleDel3, DoubleDel4;
    // Assigning a different funtion to each of the delegates
    DoubleDel1 = new DoubleOperations(Sum);
    DoubleDel2 = new DoubleOperations(Substract);
    DoubleDel3 = new DoubleOperations(Multiply);

    // Multi-cast delegate, by chainning method
    DoubleDel4 = DoubleDel1 + DoubleDel2 + DoubleDel3;
    Console.WriteLine("Option 1:");
    DoubleDel4(10, 20);

    // Option 2:
    DoubleOperations DoubleDel5 = new DoubleOperations(Sum);
    DoubleDel5 += Substract;
    DoubleDel5 += Multiply;

    Console.WriteLine("\nOption 2:");
    DoubleDel5(10, 20);

    Console.ReadKey();
}
```

Output:



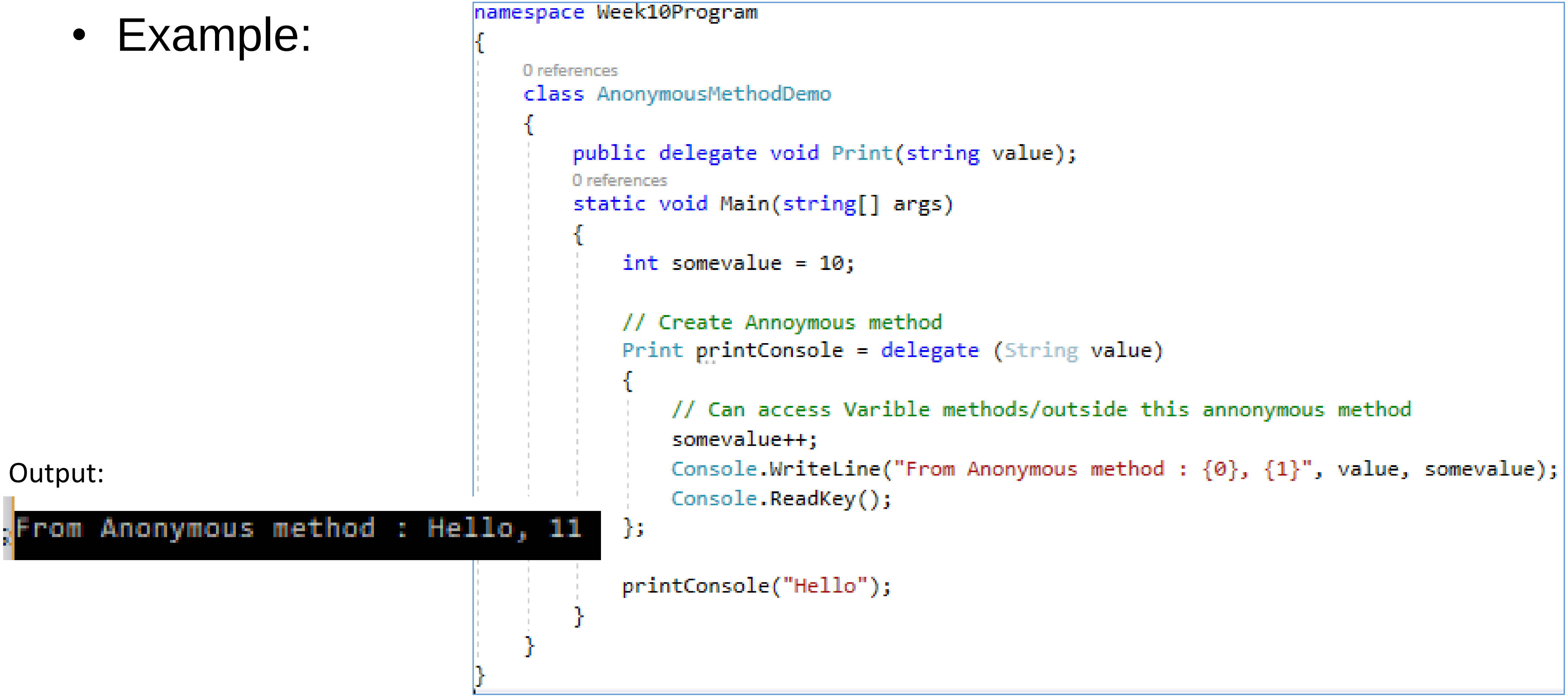
```
Option 1:
From Delegate, the Sum is: 30
From Delegate, the difference is: -10
From Delegate, the Product is: 200

Option 2:
From Delegate, the Sum is: 30
From Delegate, the difference is: -10
From Delegate, the Product is: 200
```

Anonymous Methods

- As the name suggests, it is method without a name!
- Can be defined using the delegate keyword
- Can be assigned a variable of delegate type
- Anonymous methods can access other variable/functions outside it
- Can be passed as a parameter and can be used as event handlers
- Can create an *invocation list* that may point to many methods
- If not a void, will return the result of the last method in the invocation list
- When called will successively call each method in order

Anonymous Methods

- Example:



```
namespace Week10Program
{
    0 references
    class AnonymousMethodDemo
    {
        public delegate void Print(string value);
        0 references
        static void Main(string[] args)
        {
            int somevalue = 10;

            // Create Annoymous method
            Print printConsole = delegate (String value)
            {
                // Can access Varible methods/outside this annonymous method
                somevalue++;
                Console.WriteLine("From Anonymous method : {0}, {1}", value, somevalue);
                Console.ReadKey();
            };

            printConsole("Hello");
        }
    }
}
```

Output:

```
From Anonymous method : Hello, 11
```

Generic Delegates

- Creating a large number of different delegates can be tedious.
- C# allows you to combine generics and delegates to create generic delegates

```
delegate void DelName<T>(T item);
delegate int DelName<T>(Titem);

delegate T  DelName<T>(Titem);
```

Generic Delegates

```
delegate void MyDelegate<T>(T value);

class MyClass
{
   static public void PrintLower(string s)
   {
      Console.WriteLine(s.ToLower());
   }
   static public void PrintUpper(string s)
   {
      Console.WriteLine(s.ToUpper());
   }
}

MyDelegate<string> strDel =
MyClass.PrintLower;
strDel += MyClass.PrintUpper;
strDel(“Hello”);   // call delegate
```

Action, Predicate and Func Delegates

- Besides creating you own generic delegates, C# has three predefined ones.

- `delegate void Action<T>(T arg);`
  which is designed to perform an action on some data of type T. It returns a void.

- `Delegate bool Predicate<T> (T arg);`
  which is designed to return a boolean on receipt of arg

-
  ```
  delegate T func<T>();
  delegate T func<T1, T>(T1 arg);
  delegate T func<T1, T2, T>(T1 arg1, T2 arg2);
  ```

LINQ (Language-Integrated Query)

Evolution:

- .Net 3.5 ,Visual Studio 2008
- LINQ (Language Integrated Query) is uniform query syntax in C# and VB.NET used to save and retrieve data from different sources.
- Eliminating the mismatch between programming languages and databases, as well as providing a single querying interface for different types of data sources.
- LINQ is a structured query syntax built in C# used to save and retrieve data from different types of data sources like an Object Collection, SQL server database, XML, web service etc. similar to SQL

Why LINQ ?

- LINQ always works with objects so you can use the same basic coding patterns to query and transform data in XML documents, SQL databases, ADO.NET Datasets, .NET collections, and any other format for which a LINQ provider is available.
- Use of 'foreach' or a 'for' loop to traverse the collection to find a particular object before .Net 2.0
- Then Delegates were used for such traversal.
- Need to more readability , compactness was required.
- Idea of LINQ was introduced to serve the purpose

Advantages of LINQ

- Familiar language
- Less coding
- Readable code
- Standard way to query multiple data sources
- Data shaping
- Lesser errors

LINQ Types

- LINQ to Objects
- LINQ to XML(XLINQ)
- LINQ to DataSet
- LINQ to SQL (DLINQ)
- LINQ to Entities

LINQ API

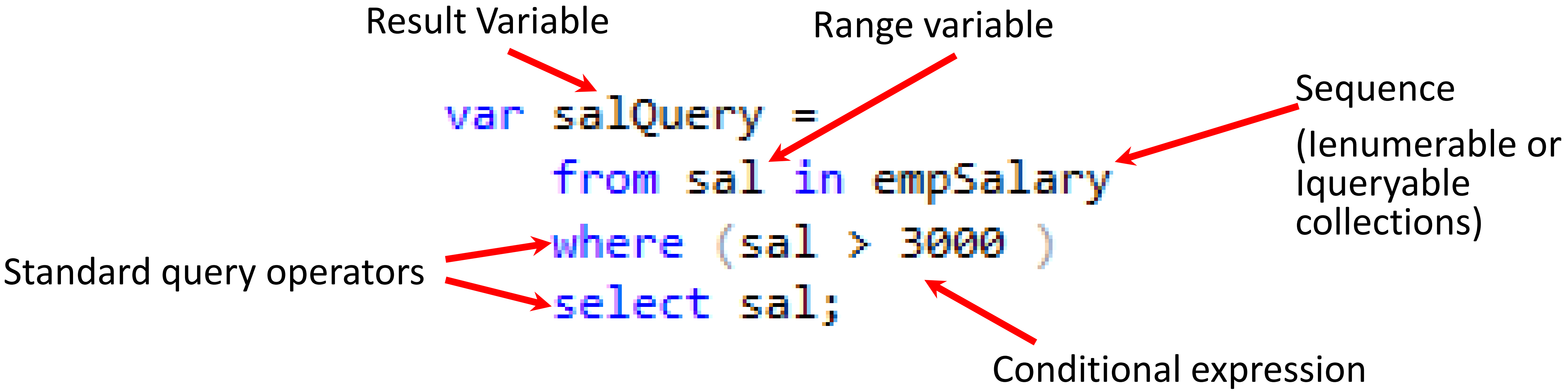
- System. Linq
  - includes the necessary classes & interfaces for LINQ. Enumerable and Queryable are two main static classes of LINQ API that contain extension methods.
  - Added automatically when you create a project.

LINQ Queries

- **Query**: It is an expression which retrieves data from a data source.
- Queries are expressed in a specialized query language, such as SQL for relational databases, XQuery for XML etc., which requires developers to learn each query language!
- LINQ simplifies the situation and uses a common model to retrieve data from different data sources!
- LINQ queries have three different parts:
    1. Obtain data source
    2. Create the query
    3. Execute the query

LINQ Queries

Query Syntax:



Result Variable

Range variable

Sequence

(Ienumerable or Iqueryable collections)

```
var salQuery =
    from sal in empSalary
    where (sal > 3000 )
    select sal;
```

Standard query operators

Conditional expression

LINQ Queries

- A query starts with `from`
- Next will follow a number of *filtering* statements, starts with `where`
- The query finishes with either `select` or `group`
- All queries are constructed using the following keywords:

`ascending`	`descending`	`equals`	`from`
`group`	`in`	`into`	`join`
`let`	`on`	`orderby`	`select`
`where`			

LINQ Queries: `from`

- A LINQ query will always start with the following

```
from someValues in  dataSource
```

- The first variable is the *range variable*. This receives the elements selected from the dataSource.

- The second variable is the data source where elements are selected from. The data source can be any LINQ compatible source.

LINQ Queries: `where`

- Next will follow a number of *filtering* statements. They have the following form.

```
where blah blah
```

- Each where clause must return a boolean.

- You can have multiple where clauses in a Linq Statement
  - `where n > 0`
  - `where n < 10`

LINQ Queries: `select`

- Specifies precisely what is obtained by the query

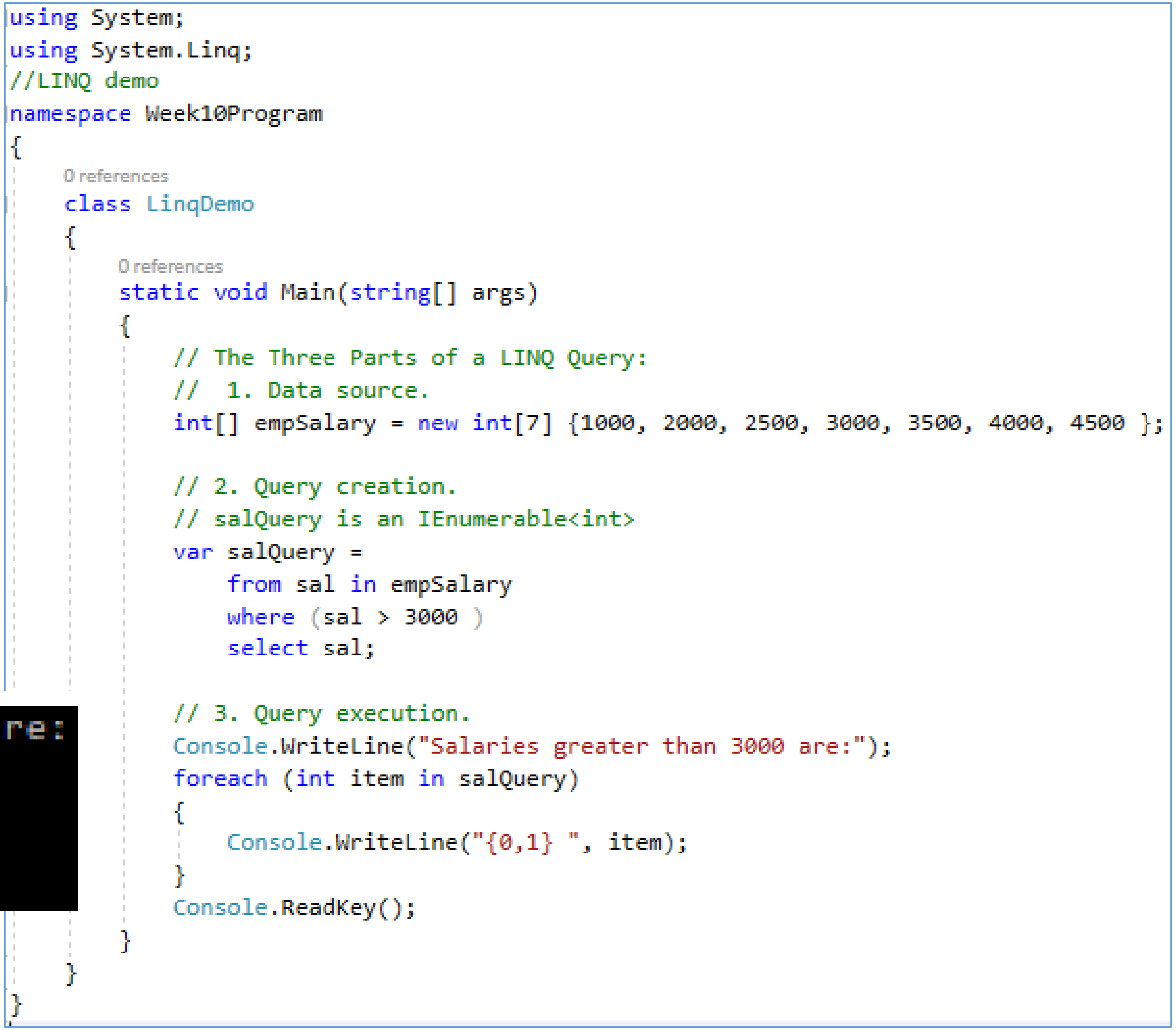
- Ends with a semi-colon since it ends a statement.

- At this point all that has been done is define the query. Only when the query is used will it be executed.

- The type of object returned by a query is an instance of
  - IEnumerable<T>.

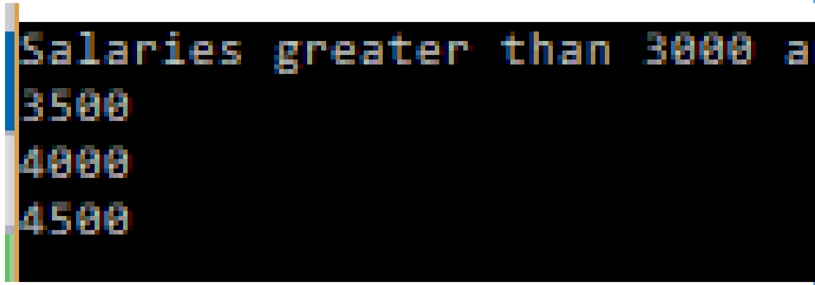
LINQ Queries

Example:



```
using System;
using System.Linq;
//LINQ demo
namespace Week10Program
{
    0 references
    class LinqDemo
    {
        0 references
        static void Main(string[] args)
        {
            // The Three Parts of a LINQ Query:
            //  1. Data source.
            int[] empSalary = new int[7] {1000, 2000, 2500, 3000, 3500, 4000, 4500 };

            // 2. Query creation.
            // salQuery is an IEnumerable<int>
            var salQuery =
                from sal in empSalary
                where (sal > 3000 )
                select sal;

            // 3. Query execution.
            Console.WriteLine("Salaries greater than 3000 are:");
            foreach (int item in salQuery)
            {
                Console.WriteLine("{0,1} ", item);
            }
            Console.ReadKey();
        }
    }
}
```

Output:



```
Salaries greater than 3000 are:
3500
4000
4500
```

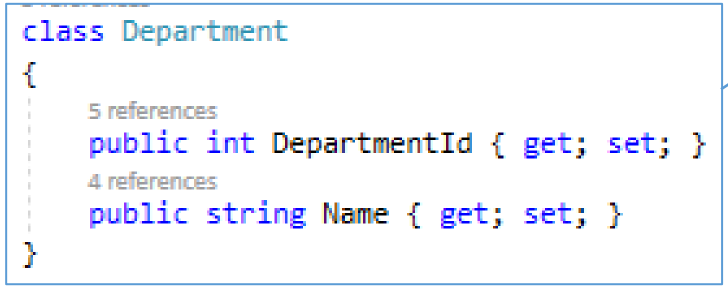
LINQ Operators

LINQ standard query operators can be categorized into the following ones on the basis of their functionality.

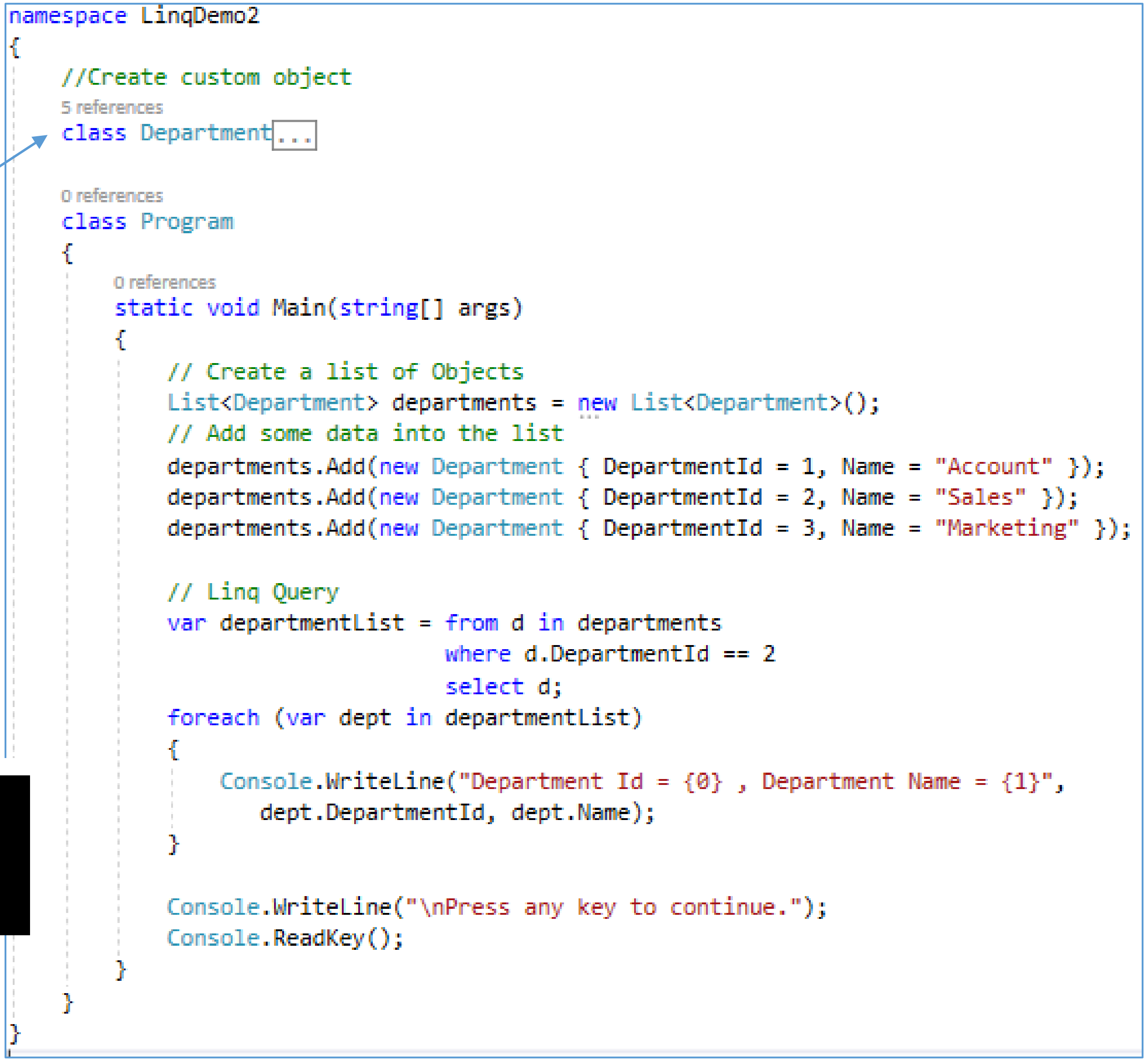
- Filtering Operators
- Join Operators
- Projection Operations
- Sorting Operators
- Grouping Operators
- Conversions
- Concatenation
- Aggregation
- Quantifier Operations
- Partition Operations
- Generation Operations
- Set Operations
- Equality
- Element Operators

LINQ with custom objects

- Example:

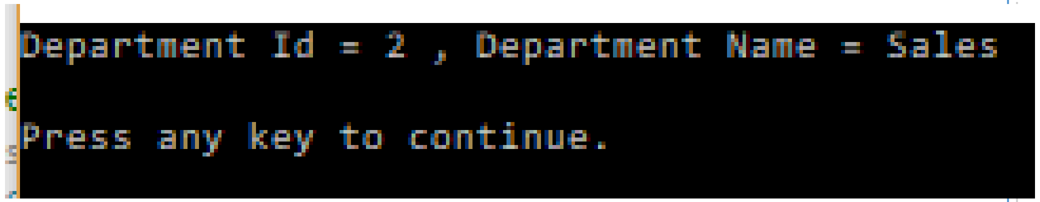


```
class Department
{
    5 references
    public int DepartmentId { get; set; }
    4 references
    public string Name { get; set; }
}
```



```
namespace LinqDemo2
{
    //Create custom object
    5 references
    class Department...

    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            // Create a list of Objects
            List<Department> departments = new List<Department>();
            // Add some data into the list
            departments.Add(new Department { DepartmentId = 1, Name = "Account" });
            departments.Add(new Department { DepartmentId = 2, Name = "Sales" });
            departments.Add(new Department { DepartmentId = 3, Name = "Marketing" });

            // Linq Query
            var departmentList = from d in departments
                                 where d.DepartmentId == 2
                                 select d;
            foreach (var dept in departmentList)
            {
                Console.WriteLine("Department Id = {0} , Department Name = {1}",
                    dept.DepartmentId, dept.Name);
            }

            Console.WriteLine("\nPress any key to continue.");
            Console.ReadKey();
        }
    }
}
```

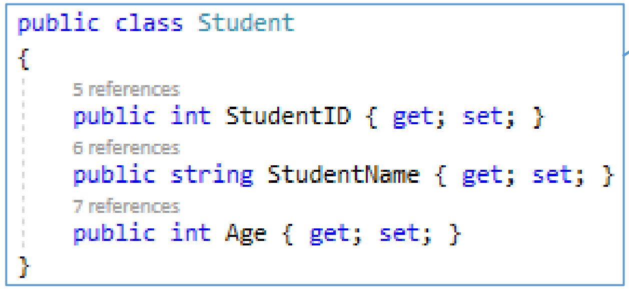
Output:



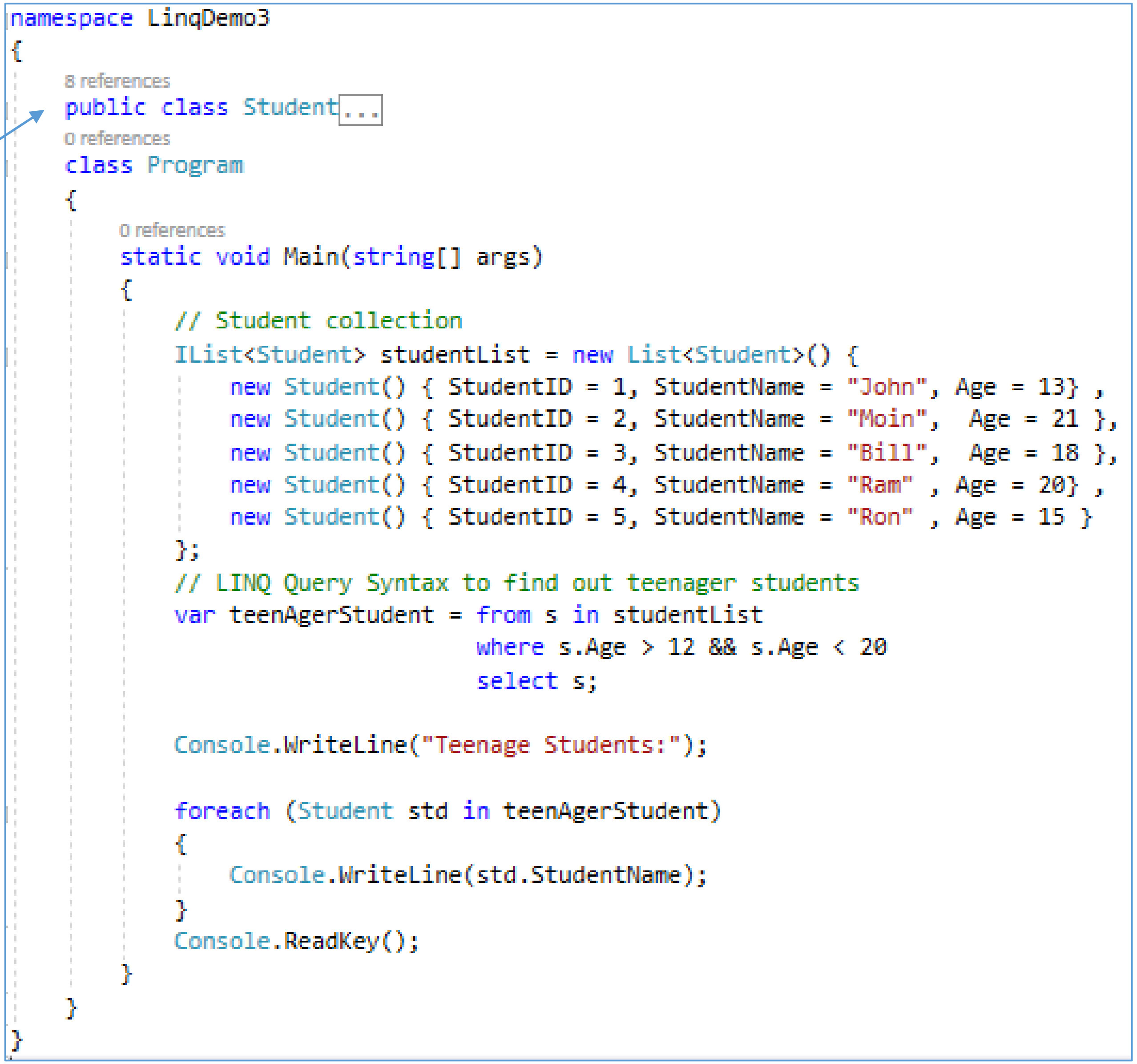
```
Department Id = 2 , Department Name = Sales

Press any key to continue.
```

LINQ with custom objects

- Example:

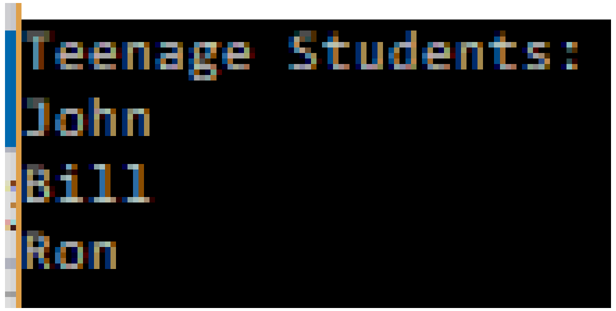


```
public class Student
{
    5 references
    public int StudentID { get; set; }
    6 references
    public string StudentName { get; set; }
    7 references
    public int Age { get; set; }
}
```



```
namespace LinqDemo3
{
    8 references
    public class Student...
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            // Student collection
            IList<Student> studentList = new List<Student>() {
                new Student() { StudentID = 1, StudentName = "John", Age = 13} ,
                new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 },
                new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 },
                new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,
                new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }
            };
            // LINQ Query Syntax to find out teenager students
            var teenAgerStudent = from s in studentList
                                  where s.Age > 12 && s.Age < 20
                                  select s;

            Console.WriteLine("Teenage Students:");

            foreach (Student std in teenAgerStudent)
            {
                Console.WriteLine(std.StudentName);
            }
            Console.ReadKey();
        }
    }
}
```

Output:



```
Teenage Students:
John
Bill
Ron
```

LINQ Queries: `orderby`

- Optional command you can add to your query to order the results of the query.

```
orderby n ascending
orderby n descending
```

- This assumes the data your are selecting is IComparable<T>

- Can order on multiple components.

LINQ Queries: `group`

- Allows you to create results that are grouped by keys.
- The query will return a sequence of IEnumerable containing elements of type IGrouping<TKey, TElement>

LINQ Queries: `join`

- We may need to integrate the results of two different queries.

- The join command allows us to do this.

```
from var_A    in   data_source_A
  join var_B  in   data_source_B
    on var_A.property  equals  var_B.property
```

LINQ: Query Methods

- Lambda expressions can be used in LINQ queries to form query methods

- It does this by extending the lambda expression with the where and select LINQ clauses.

LINQ: Lambda Expressions

- C# 3.0(.NET 3.5) introduced the lambda expression along with LINQ. The lambda expression is a shorter way of representing anonymous method using some special syntax.
- Before they can be used, a delegate must be declared.
- They introduce a new operator => and have the form

```
    (typed parameter_list) => expression
```

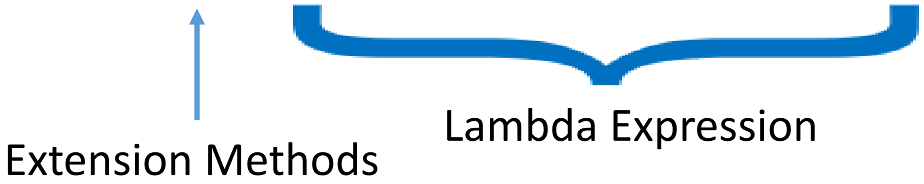
- *delegate(Student s) { return s.Age > 12 && s.Age < 20; };*
- *s => s.Age > 12 && s.Age < 20*

LINQ: Lambda Expressions

```
/ string collection
IList<string> stringList = new List<string>() {
    "C# Tutorials",
    "VB.NET Tutorials",
    "Learn C++",
    "MVC Tutorials" ,
    "Java"
};
```

- // LINQ Query Syntax

```
var result = stringList.Where(s => s.Contains("Tutorials"));
```



Lambda Expression

Extension Methods

LINQ: Lambda Expressions

- LINQ provides a number of other extension methods

Method	Description
All(condition)	Returns true if all elements in the sequence satisfy the condition
Any(condition)	Returns true if any element in the sequence satisfies the condition
Average()	Returns the average of the values in a numeric sequence
Contains(obj)	Returns true if the sequence contains the object
Count()	Returns a count of the number of elements in the sequence
First()	Returns the first element in the sequence
Last()	Returns the last element in the sequence
Max()	Returns the largest element in the sequence
Min()	Returns the smallest element in the sequence
Sum()	Returns the sum of the values in a numeric sequence