ASP.NET MVC Entity Framework

SummitWorks™
GLOBAL SOLUTION ARCHITECTS

Authorized & published by Summitworks Technologies Inc

Agenda

- Working with Models
  - What are Filters?
  - Types of Filters
    - Authorization
    - Action
    - Result
    - Exception

- Understanding LINQ
  - LINQ Syntax with Examples
    - Usage of Lambda and Query Syntax
  - Database Initializers, and Data Annotations
  - Implementing Data Sorting & Data Filtering in a Controller

What are Filters?

Filters

In ASP.NET MVC, controllers define action methods that usually have a one-to-one relationship with possible user interactions, but sometimes you want to perform logic either before an action method is called or after an action method runs.

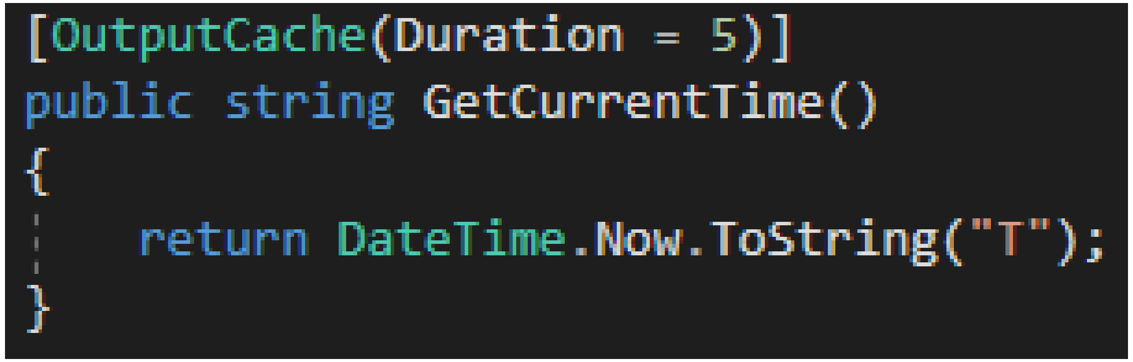
To support this, ASP.NET MVC provides filters. **Filters** are custom classes that provide both a declarative and programmatic means to add pre-action and post-action behavior to controller action methods.

An action filter is an attribute that you can apply to a controller action or an entire controller that modifies the way in which the action is executed. The ASP.NET MVC framework includes several action filters:

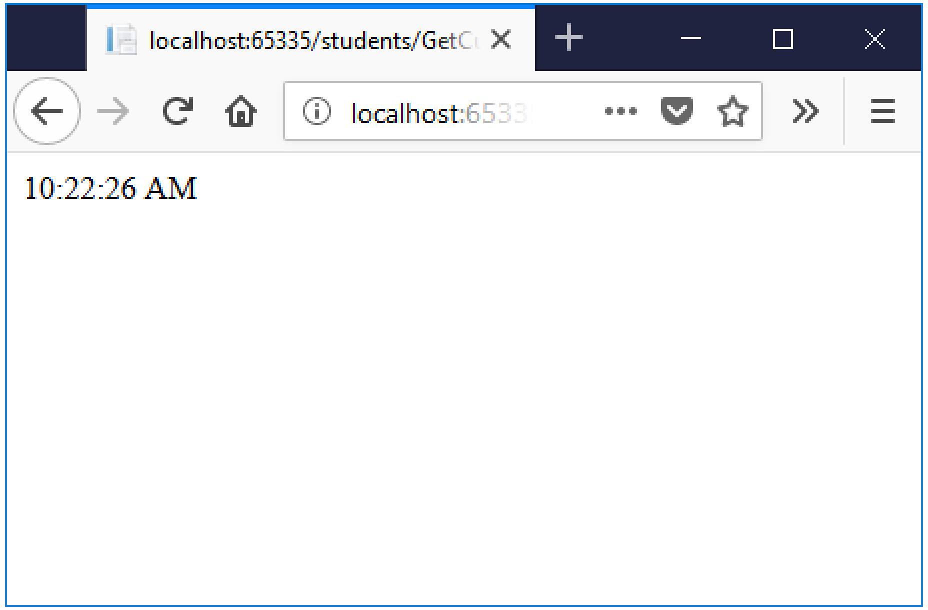
- **OutputCache**: Caches the output of a controller action for a specified amount of time.
- **HandleError**: Handles errors raised when a controller action is executed.
- **Authorize**: Enables you to restrict access to a particular user or role.

OutputCache

An action filter **OutputCache** is applied to an action named GetCurrentTime() that returns the string. This filter causes the value returned by the action to be cached for 5 seconds.

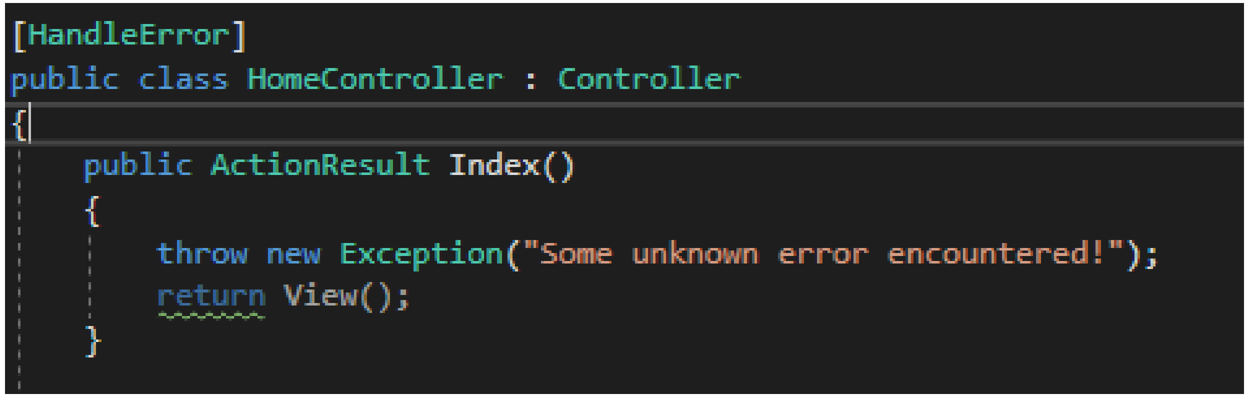


```
[OutputCache(Duration = 5)]
public string GetCurrentTime()
{
    return DateTime.Now.ToString("T");
}
```

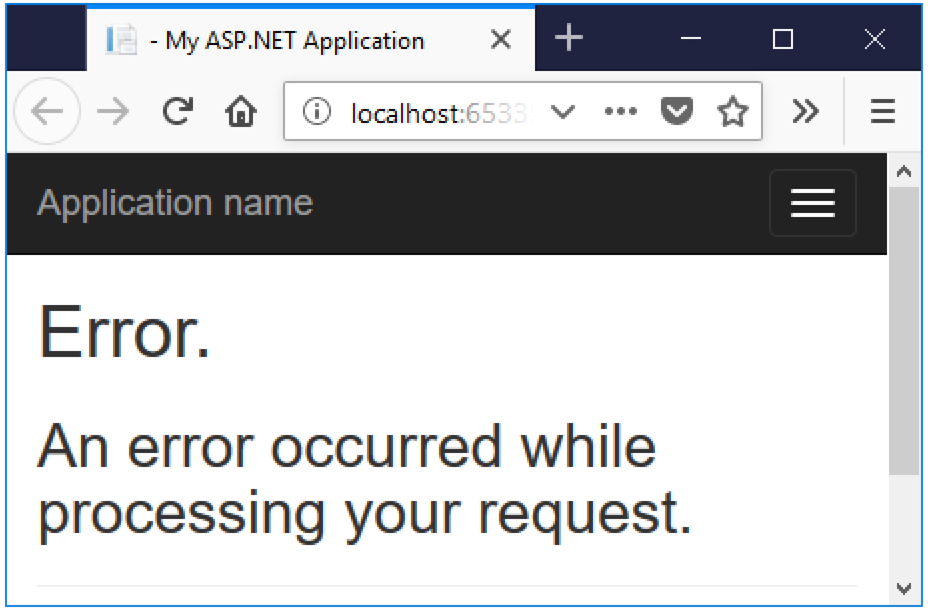


```
10:22:26 AM
```

HandleError



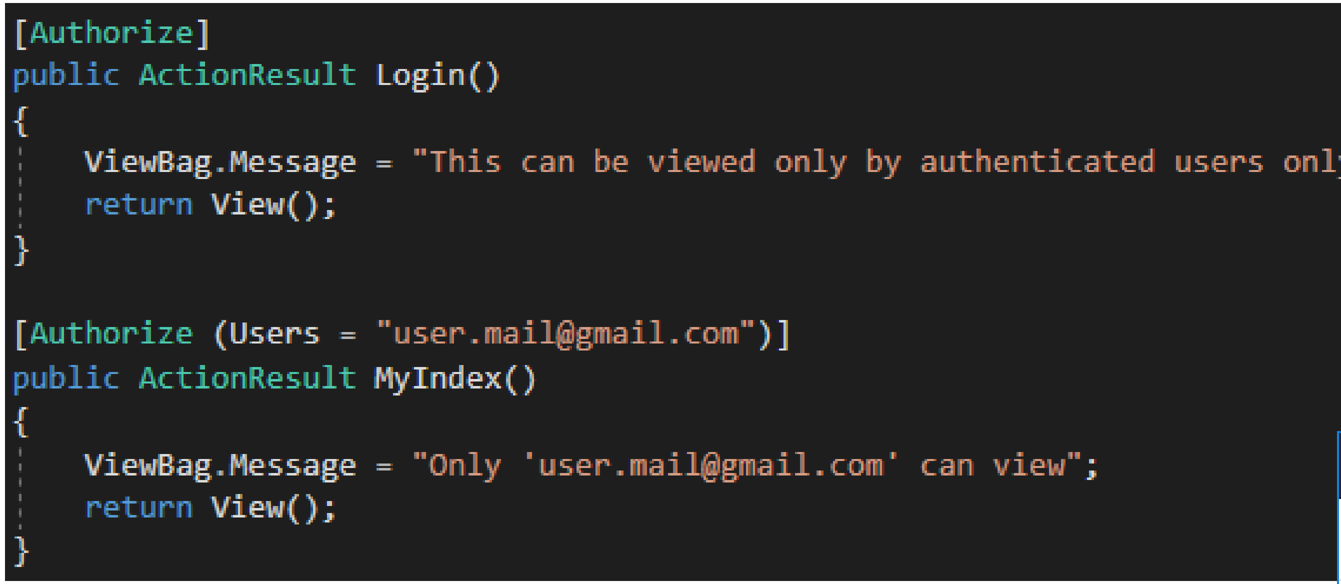
```
[HandleError]
public class HomeController : Controller
{
    public ActionResult Index()
    {
        throw new Exception("Some unknown error encountered!");
        return View();
    }
```



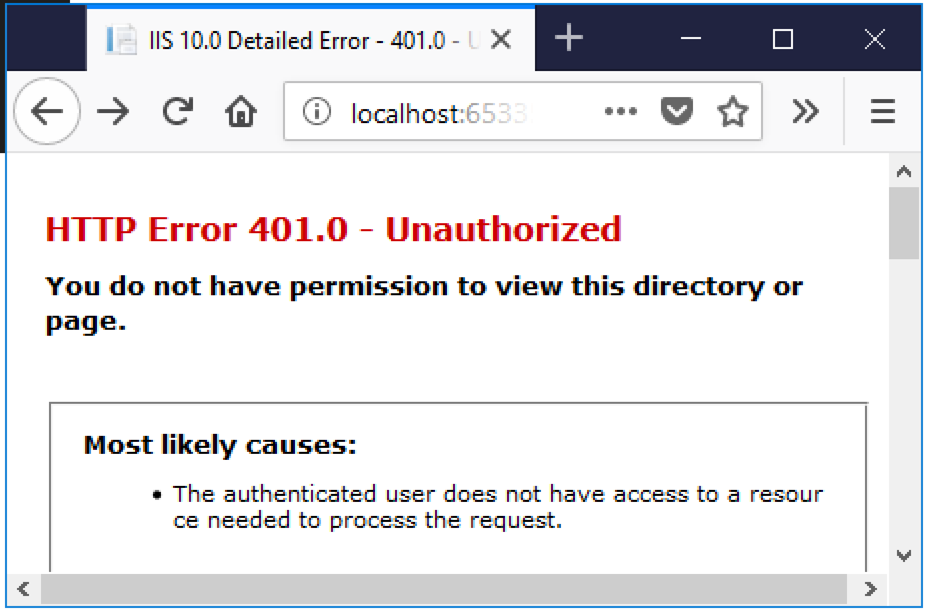
Error.

An error occurred while processing your request.

Authorize



```
[Authorize]
public ActionResult Login()
{
    ViewBag.Message = "This can be viewed only by authenticated users only";
    return View();
}

[Authorize (Users = "user.mail@gmail.com")]
public ActionResult MyIndex()
{
    ViewBag.Message = "Only 'user.mail@gmail.com' can view";
    return View();
}
```



HTTP Error 401.0 - Unauthorized

You do not have permission to view this directory or page.

Most likely causes:

- The authenticated user does not have access to a resour ce needed to process the request.

Types of Filters

The ASP.NET MVC framework supports four different types of filters:

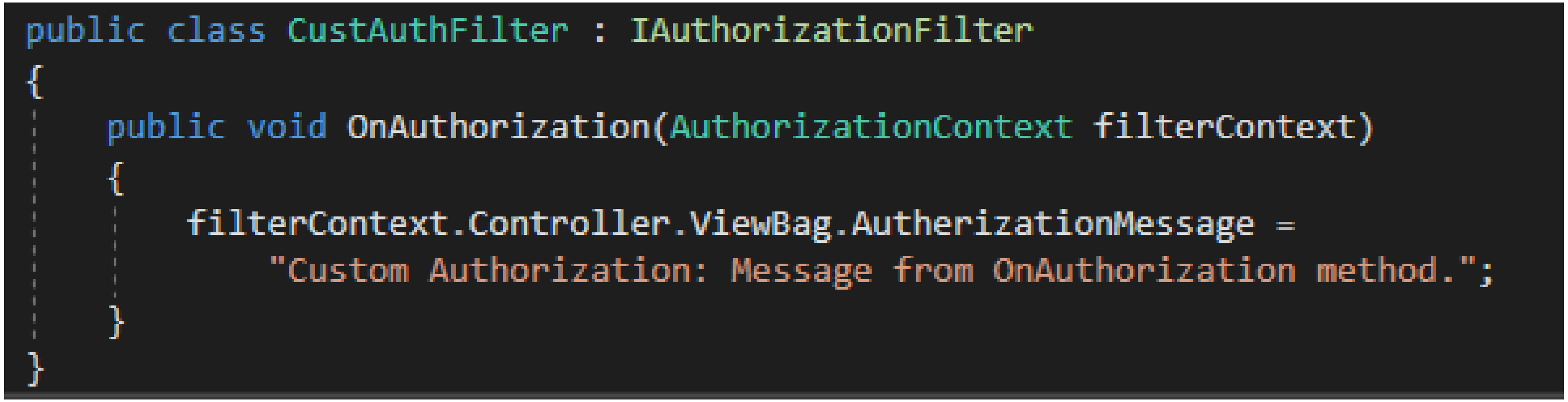
Filter Type	Description
Authorization filters	Performs authentication and authorizes before executing action method.
Action filters	Performs some operation before and after an action method executes.
Result filters	Performs some operation before or after the execution of view result.
Exception filters	Performs some operation if there is an unhandled exception thrown during the execution of the ASP.NET MVC pipeline.

Filters are executed in the order listed above. For example, authorization filters are always executed before action filters and exception filters are always executed after every other type of filter.

Authorization filters are used to implement authentication and authorization for controller actions. For example, the Authorize filter is an example of an Authorization filter.

Authorization Filters

- Used to authorize a request.
- This filter will be executed once after user is authenticated.
- The interface that needs to be implemented for this filter is *IAuthorizationFilter.*

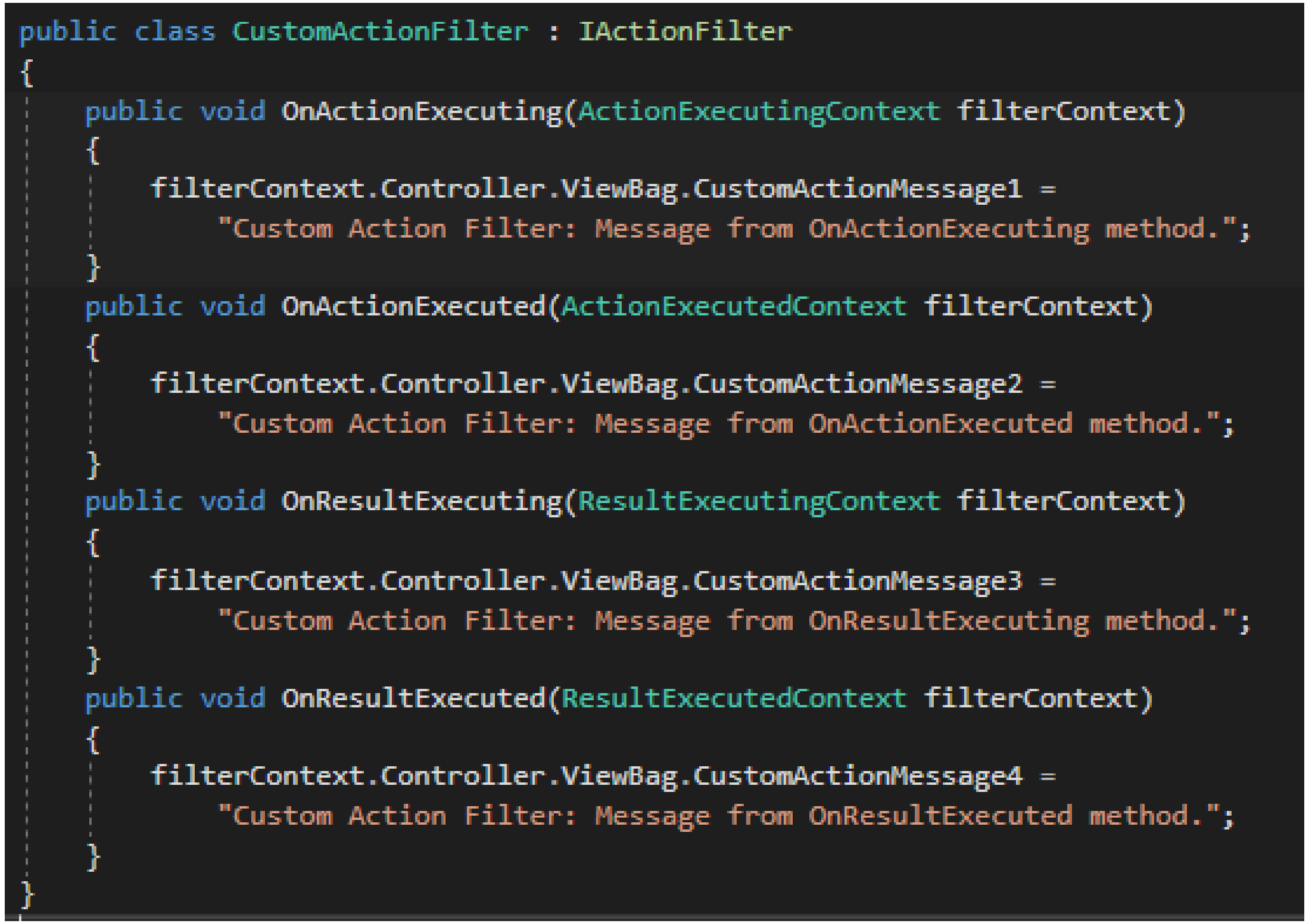


```
public class CustAuthFilter : IAuthorizationFilter
{
    public void OnAuthorization(AuthorizationContext filterContext)
    {
        filterContext.Controller.ViewBag.AutherizationMessage =
            "Custom Authorization: Message from OnAuthorization method.";
    }
}
```

Action Filters

- This filter will be called before and after the action starts executing and after the action has executed.
- The interface that needs to be implemented for this filter is *IActionFilter.*
- There are 4 events available in an action filter:

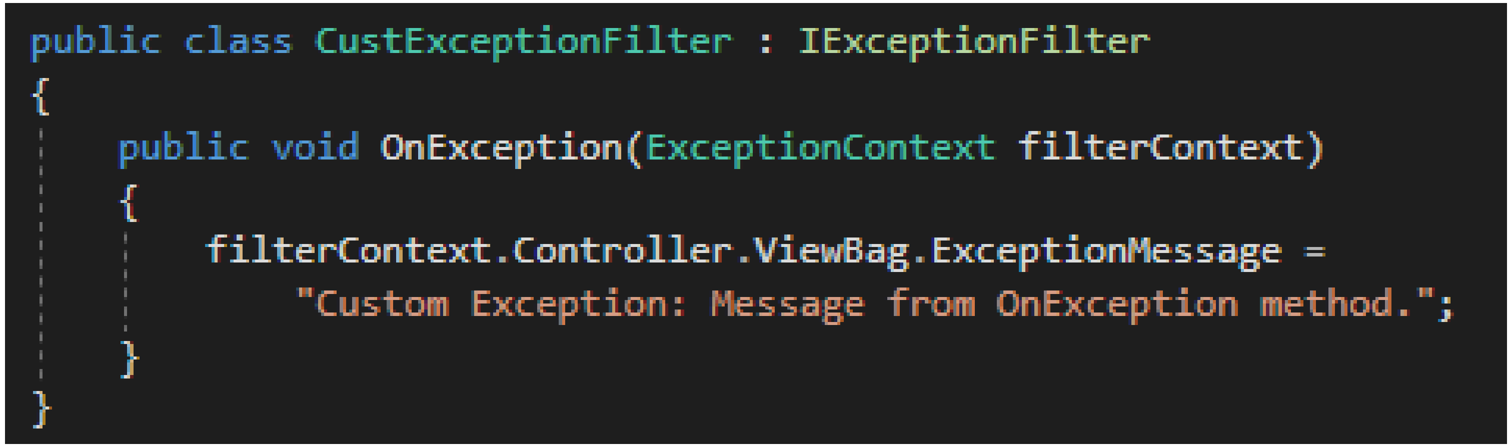
  1. **OnActionExecuting:** Runs before execution of Action method.
  2. **OnActionExecuted:** Runs after execution of Action method.
  3. **OnResultExecuting**: Runs before content is rendered to View.
  4. **OnResultExecuted:** Runs after content is rendered to view.



```
public class CustomActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext filterContext)
    {
        filterContext.Controller.ViewBag.CustomActionMessage1 =
            "Custom Action Filter: Message from OnActionExecuting method.";
    }
    public void OnActionExecuted(ActionExecutedContext filterContext)
    {
        filterContext.Controller.ViewBag.CustomActionMessage2 =
            "Custom Action Filter: Message from OnActionExecuted method.";
    }
    public void OnResultExecuting(ResultExecutingContext filterContext)
    {
        filterContext.Controller.ViewBag.CustomActionMessage3 =
            "Custom Action Filter: Message from OnResultExecuting method.";
    }
    public void OnResultExecuted(ResultExecutedContext filterContext)
    {
        filterContext.Controller.ViewBag.CustomActionMessage4 =
            "Custom Action Filter: Message from OnResultExecuted method.";
    }
}
```

Exception Filters

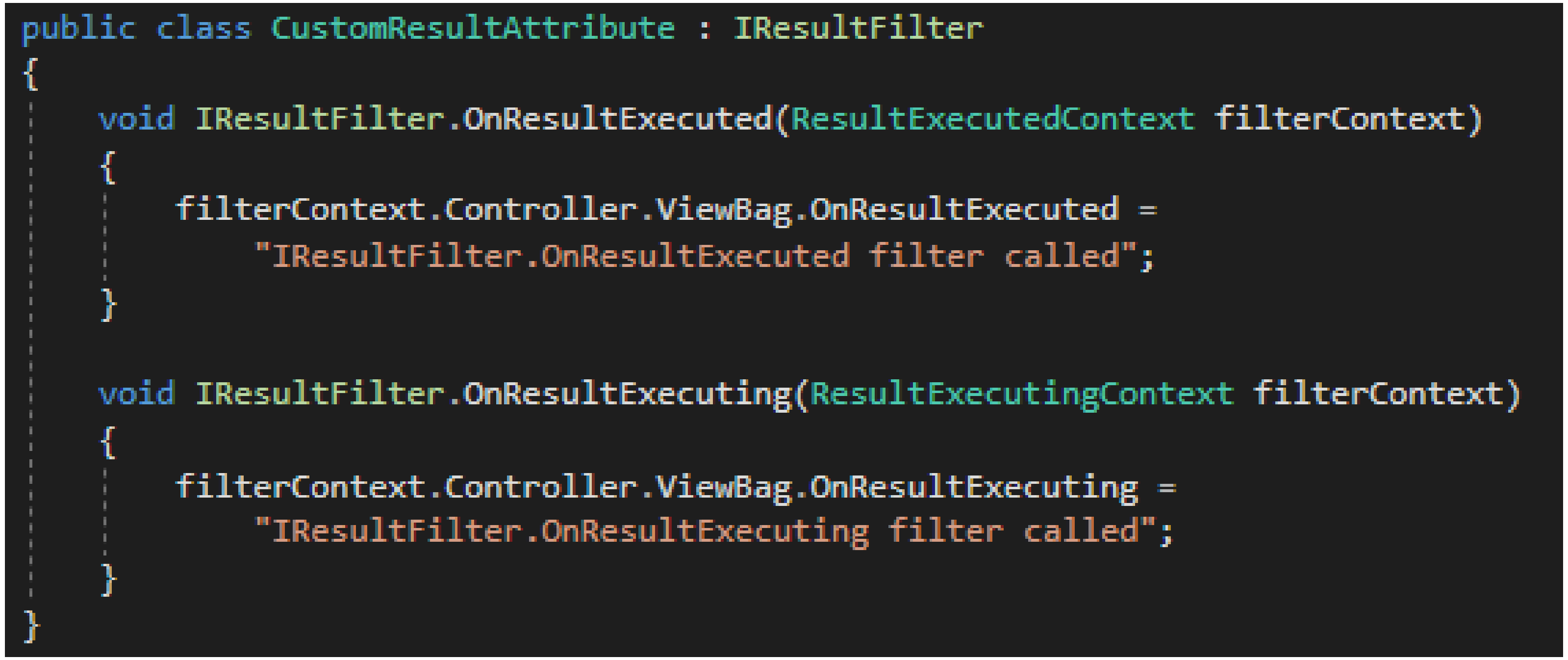
- This filter is used to capture any exceptions if raised by controller or an action method.
- The interface that needs to be implemented for this filter is *IExceptionFilter.*



```
public class CustExceptionFilter : IExceptionFilter
{
    public void OnException(ExceptionContext filterContext)
    {
        filterContext.Controller.ViewBag.ExceptionMessage =
            "Custom Exception: Message from OnException method.";
    }
}
```

Result Filter

- Result filters can run code immediately before and after the execution of individual action results. They run only when the action method has executed successfully.
- Implement either the *IResultFilter*.



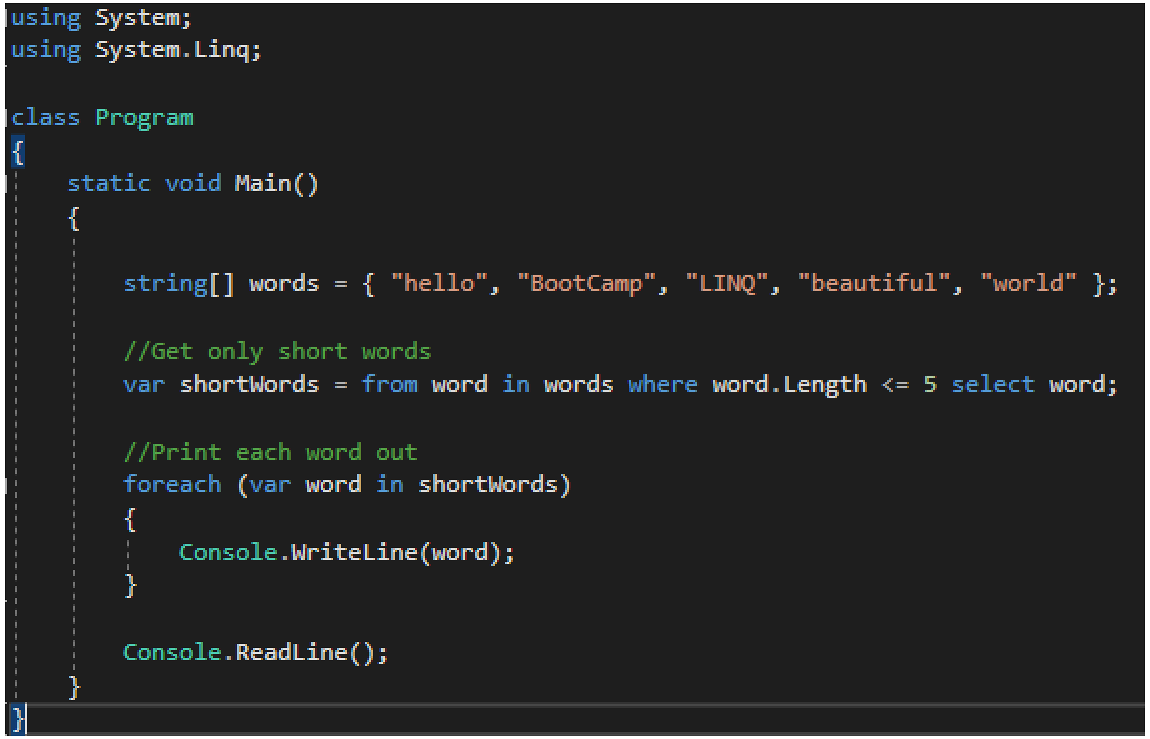
```
public class CustomResultAttribute : IResultFilter
{
    void IResultFilter.OnResultExecuted(ResultExecutedContext filterContext)
    {
        filterContext.Controller.ViewBag.OnResultExecuted =
            "IResultFilter.OnResultExecuted filter called";
    }

    void IResultFilter.OnResultExecuting(ResultExecutingContext filterContext)
    {
        filterContext.Controller.ViewBag.OnResultExecuting =
            "IResultFilter.OnResultExecuting filter called";
    }
}
```

Understanding LINQ

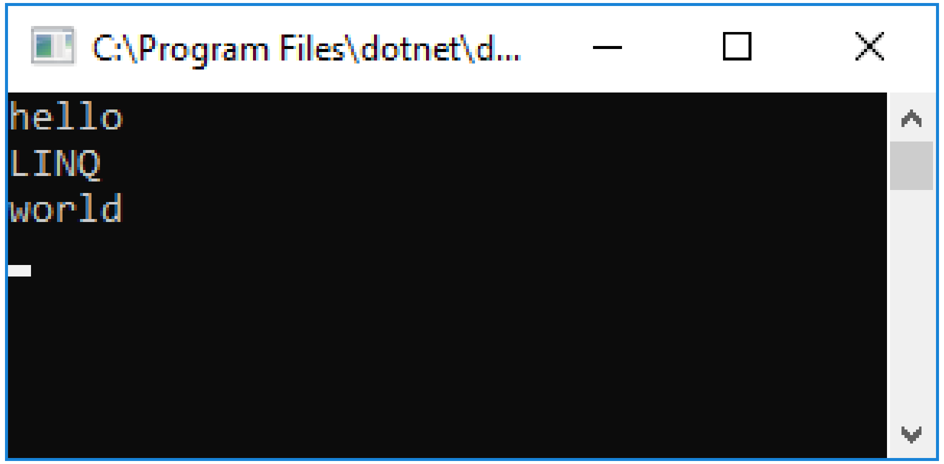
Understanding LINQ

Developers across the world have always encountered problems in querying data because of the lack of a defined path and need to master a multiple of technologies like SQL, Web Services, XQuery, etc.

Introduced in Visual Studio 2008 and designed by Anders Hejlsberg, LINQ (Language Integrated Query) allows writing queries even without the knowledge of query languages like SQL, XML etc. LINQ queries can be written for diverse data types.



```
using System;
using System.Linq;

class Program
{
    static void Main()
    {

        string[] words = { "hello", "BootCamp", "LINQ", "beautiful", "world" };

        //Get only short words
        var shortWords = from word in words where word.Length <= 5 select word;

        //Print each word out
        foreach (var word in shortWords)
        {
            Console.WriteLine(word);
        }

        Console.ReadLine();
    }
}
```

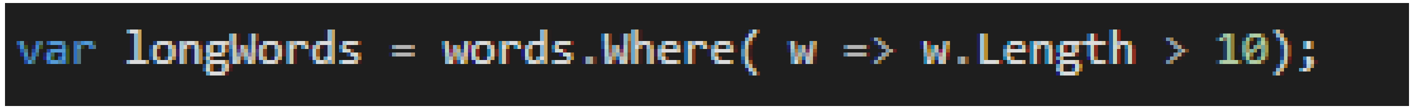


```
hello
LINQ
world
```

LINQ Syntax

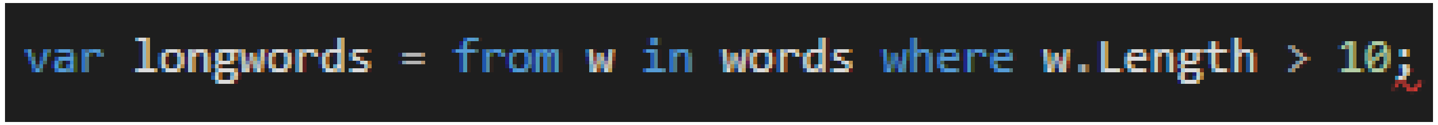
There are two syntaxes of LINQ. These are the following ones.

- **Lambda (Method) Syntax**



```
var longWords = words.Where( w => w.Length > 10);
```

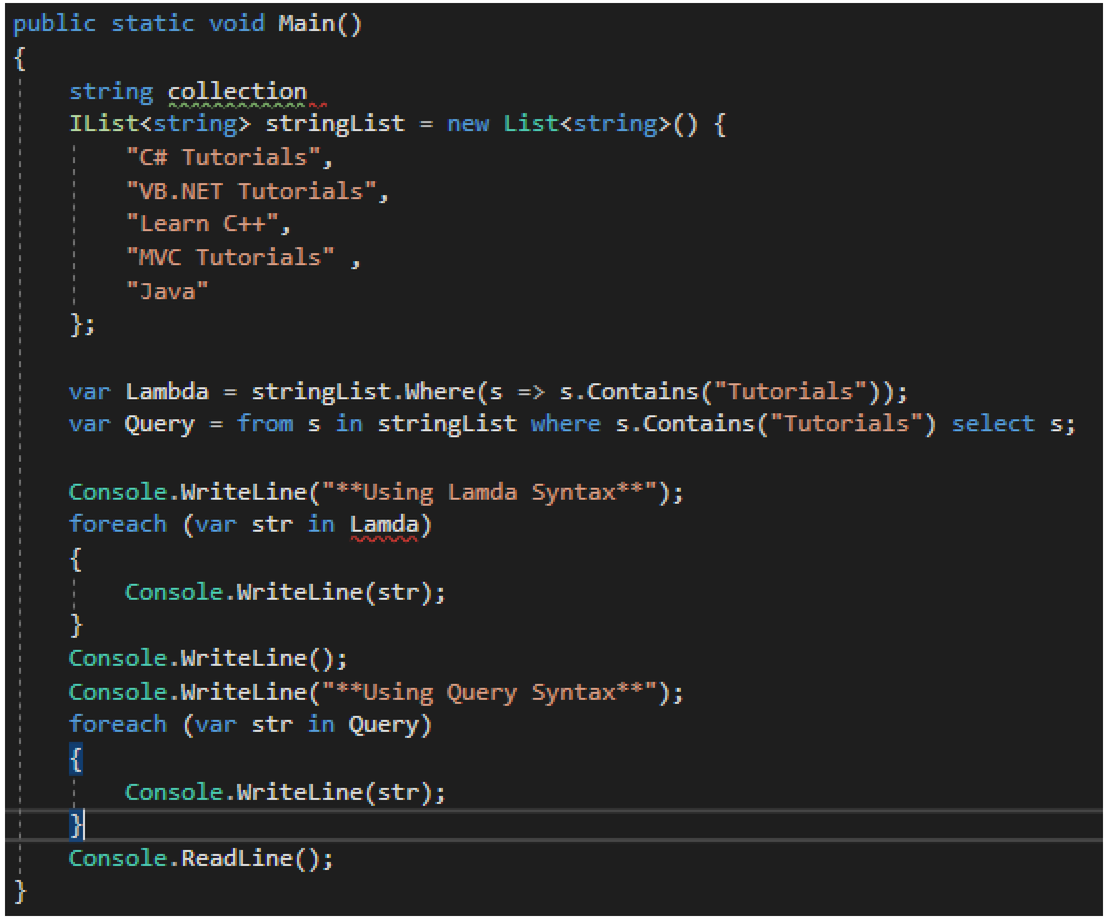
- **Query (Comprehension) Syntax**



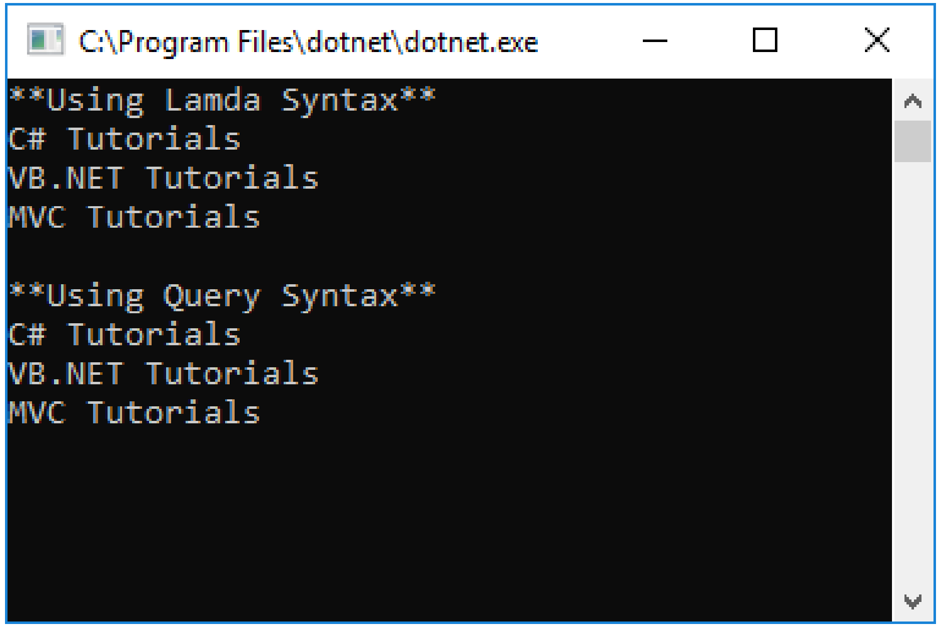
```
var longwords = from w in words where w.Length > 10;
```

LINQ Syntax

- **Example using where and contains**



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

    var Lambda = stringList.Where(s => s.Contains("Tutorials"));
    var Query = from s in stringList where s.Contains("Tutorials") select s;

    Console.WriteLine("**Using Lamda Syntax**");
    foreach (var str in Lamda)
    {
        Console.WriteLine(str);
    }
    Console.WriteLine();
    Console.WriteLine("**Using Query Syntax**");
    foreach (var str in Query)
    {
        Console.WriteLine(str);
    }
    Console.ReadLine();
}
```

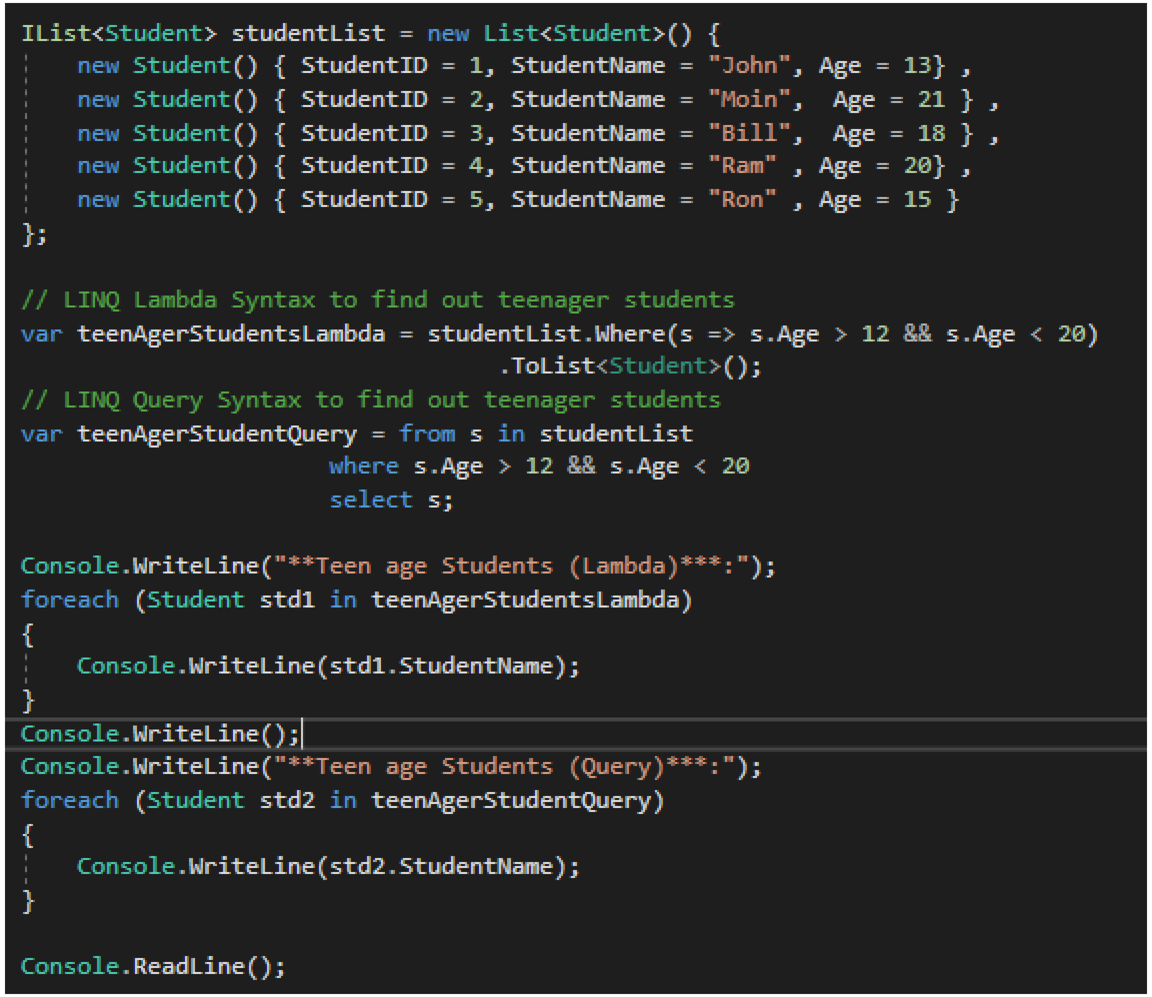


C:\Program Files\dotnet\dotnet.exe

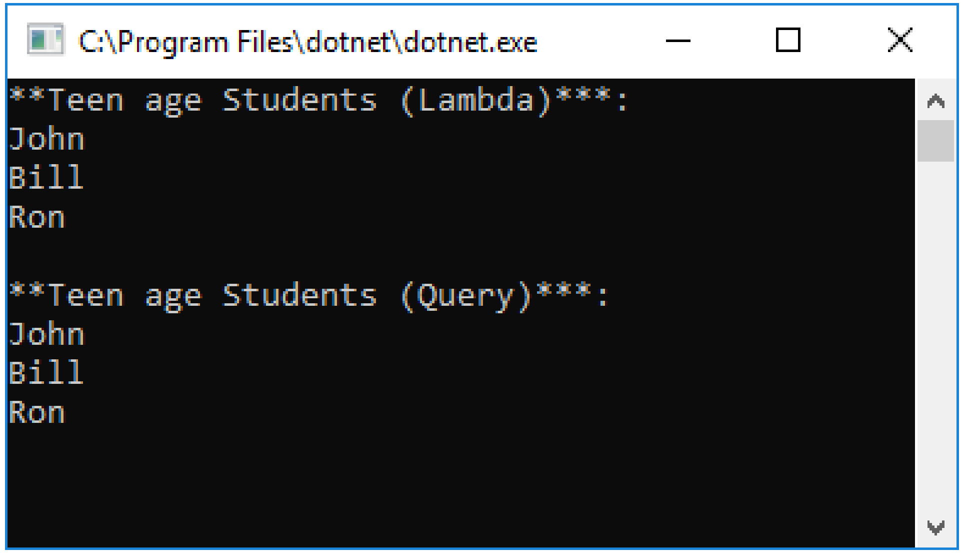
```
**Using Lamda Syntax**
C# Tutorials
VB.NET Tutorials
MVC Tutorials

**Using Query Syntax**
C# Tutorials
VB.NET Tutorials
MVC Tutorials
```

LINQ Syntax

- **Example using where**



```
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 Lambda Syntax to find out teenager students
var teenAgerStudentsLambda = studentList.Where(s => s.Age > 12 && s.Age < 20)
                                  .ToList<Student>();
// LINQ Query Syntax to find out teenager students
var teenAgerStudentQuery = from s in studentList
                      where s.Age > 12 && s.Age < 20
                      select s;

Console.WriteLine("**Teen age Students (Lambda)***:");
foreach (Student std1 in teenAgerStudentsLambda)
{
    Console.WriteLine(std1.StudentName);
}
Console.WriteLine();
Console.WriteLine("**Teen age Students (Query)***:");
foreach (Student std2 in teenAgerStudentQuery)
{
    Console.WriteLine(std2.StudentName);
}

Console.ReadLine();
```



```
**Teen age Students (Lambda)***:
John
Bill
Ron

**Teen age Students (Query)***:
John
Bill
Ron
```

Standard Query Operators

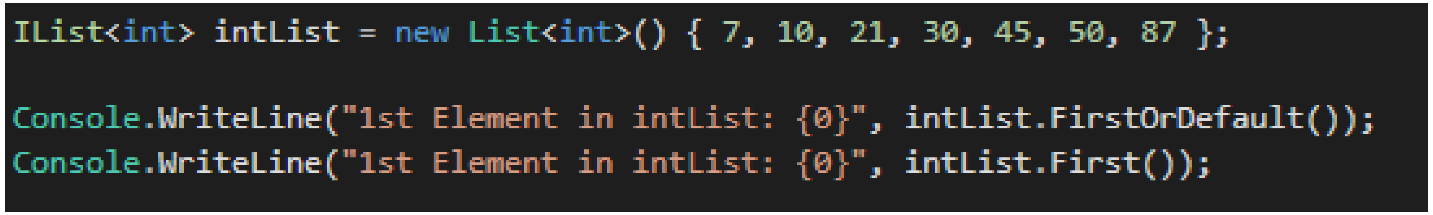
There are over 50 standard query operators available in LINQ that provide different functionalities like filtering, sorting, grouping, aggregation, concatenation, etc.

Standard Query Operators can be classified based on the functionality they provide.

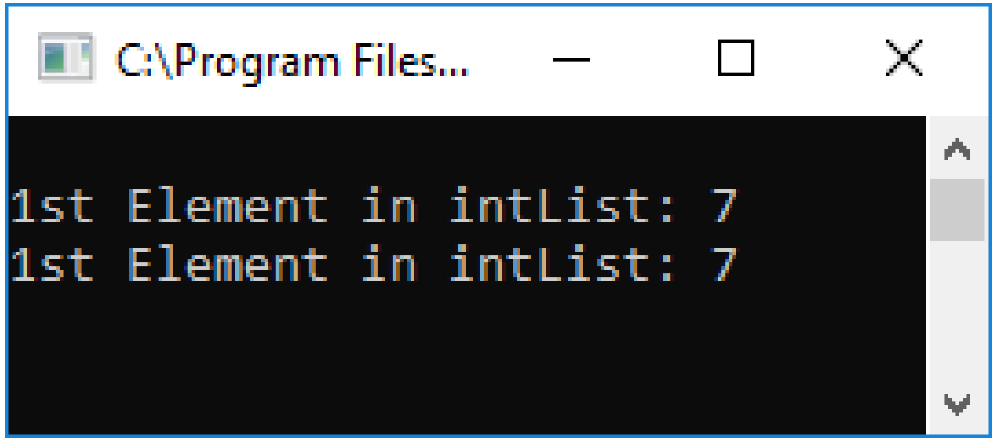
Classification	Standard Query Operators
Filtering	Where, OfType
Sorting	OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse
Grouping	GroupBy, ToLookup
Join	GroupJoin, Join
Projection	Select, SelectMany
Aggregation	Aggregate, Average, Count, LongCount, Max, Min, Sum
Quantifiers	All, Any, Contains
Elements	ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault
Set	Distinct, Except, Intersect, Union
Partitioning	Skip, SkipWhile, Take, TakeWhile
Concatenation	Concat
Equality	SequenceEqual
Generation	DefaultEmpty, Empty, Range, Repeat
Conversion	AsEnumerable, AsQueryable, Cast, ToArray, ToDictionary, ToList

LINQ Syntax

- **Example using First and FirstOrDefault**



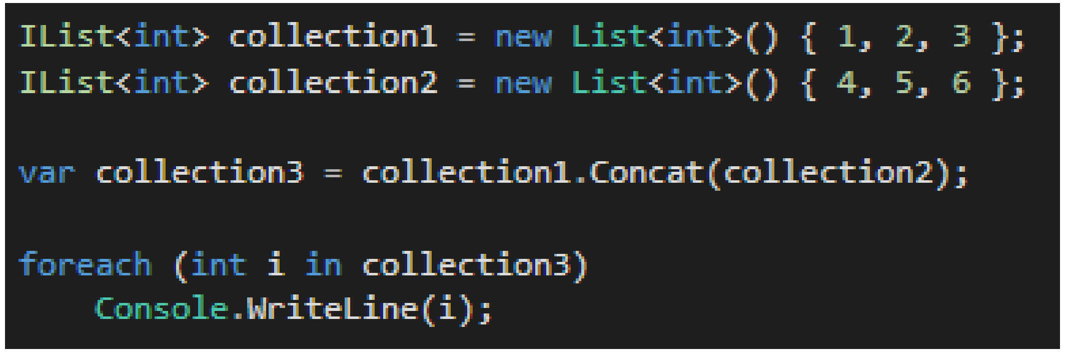
```
IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };

Console.WriteLine("1st Element in intList: {0}", intList.FirstOrDefault());
Console.WriteLine("1st Element in intList: {0}", intList.First());
```



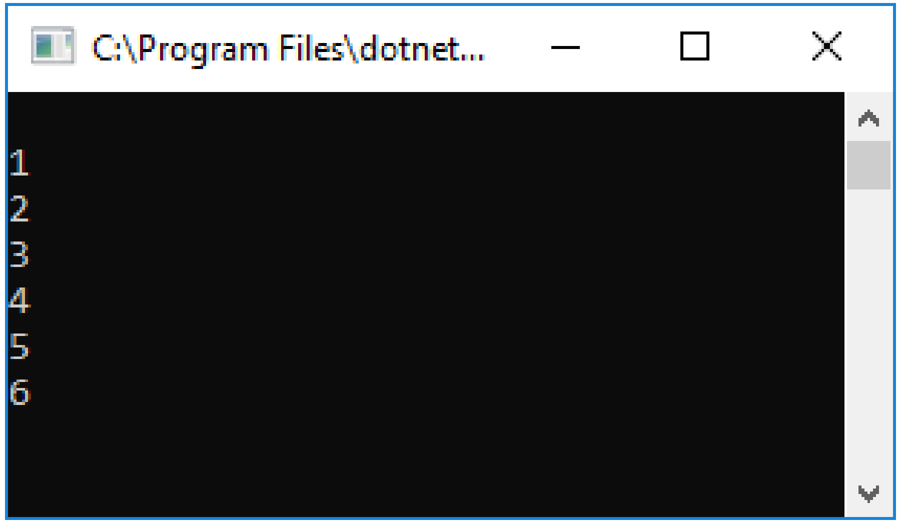
```
1st Element in intList: 7
1st Element in intList: 7
```

LINQ Syntax

- **Example using Concat**



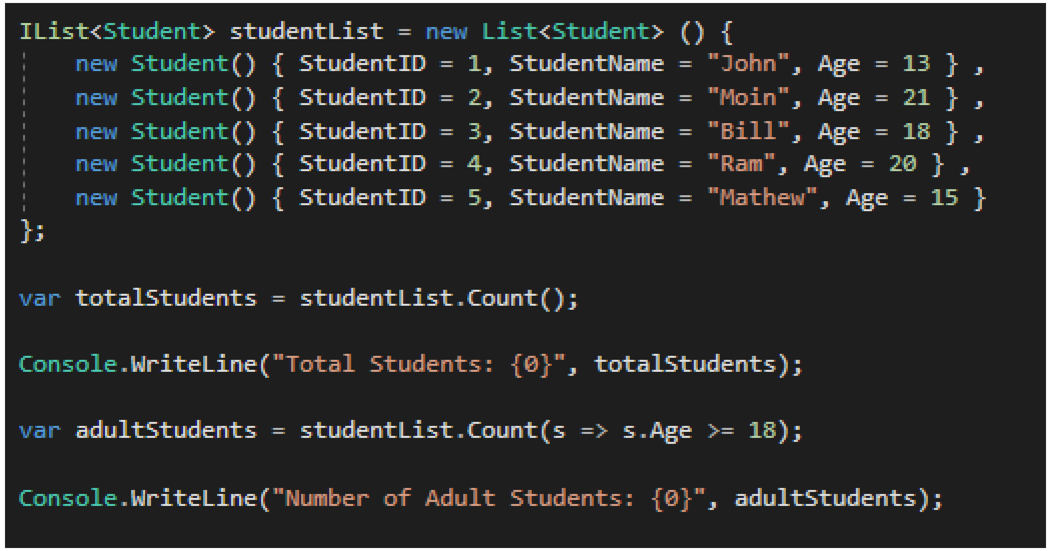
```
IList<int> collection1 = new List<int>() { 1, 2, 3 };
IList<int> collection2 = new List<int>() { 4, 5, 6 };

var collection3 = collection1.Concat(collection2);

foreach (int i in collection3)
    Console.WriteLine(i);
```



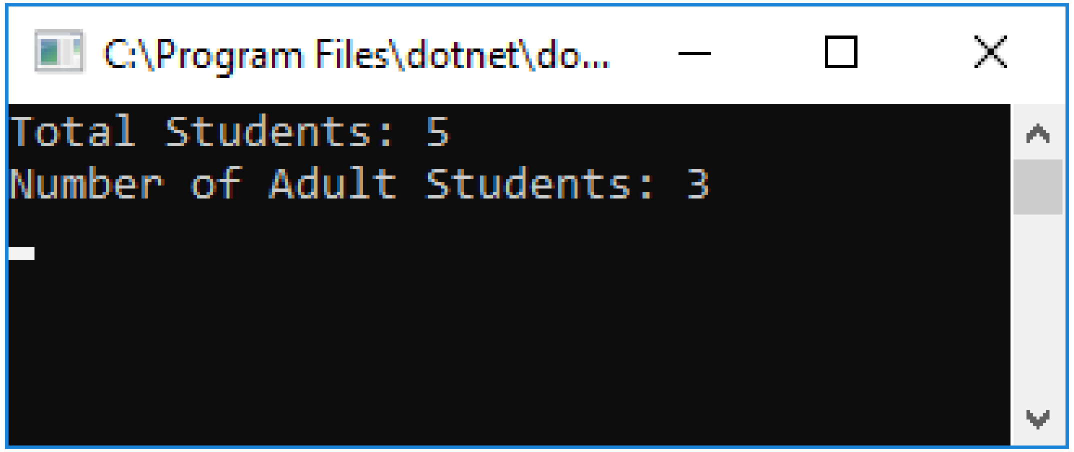
```
1
2
3
4
5
6
```

LINQ Syntax

- **Example using Count**



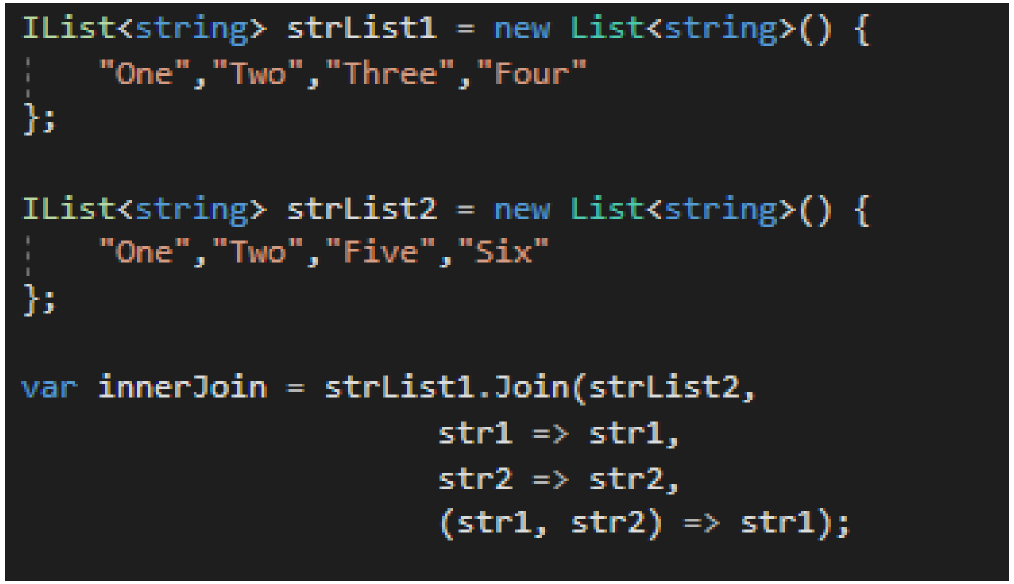
```
IList<Student> studentList = new List<Student> () {
    new Student() { StudentID = 1, StudentName = "John", Age = 13 } ,
    new Student() { StudentID = 2, StudentName = "Moin", Age = 21 } ,
    new Student() { StudentID = 3, StudentName = "Bill", Age = 18 } ,
    new Student() { StudentID = 4, StudentName = "Ram", Age = 20 } ,
    new Student() { StudentID = 5, StudentName = "Mathew", Age = 15 }
};

var totalStudents = studentList.Count();

Console.WriteLine("Total Students: {0}", totalStudents);

var adultStudents = studentList.Count(s => s.Age >= 18);

Console.WriteLine("Number of Adult Students: {0}", adultStudents);
```



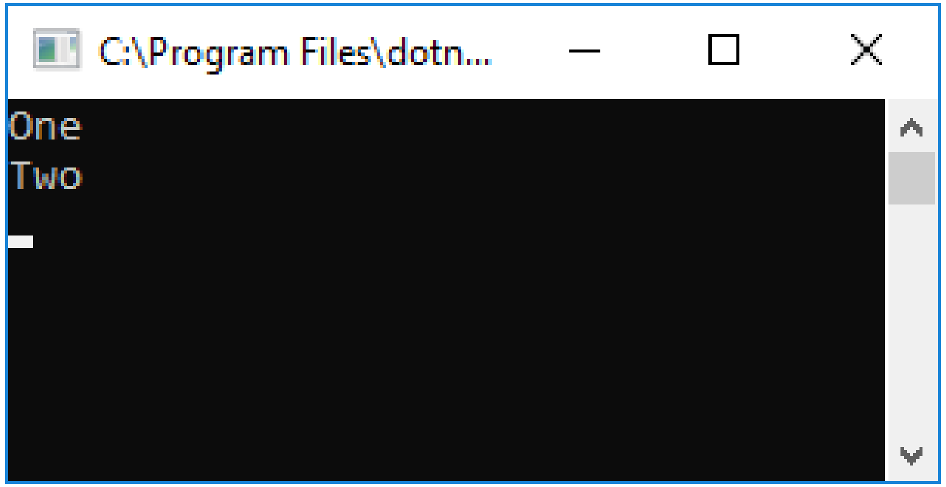
```
Total Students: 5
Number of Adult Students: 3
```

LINQ Syntax

- **Example using Join**



```
IList<string> strList1 = new List<string>() {
    "One","Two","Three","Four"
};

IList<string> strList2 = new List<string>() {
    "One","Two","Five","Six"
};

var innerJoin = strList1.Join(strList2,
                      str1 => str1,
                      str2 => str2,
                      (str1, str2) => str1);
```



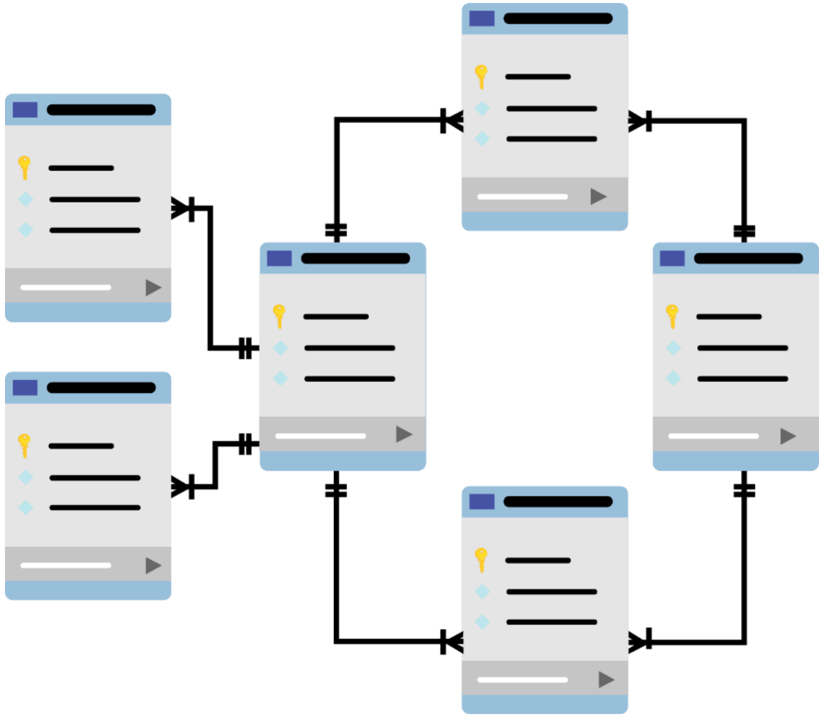
```
One
Two
```

Database Initializers

In code-first approach the user who is working will only concentrating on creating classes, models and writing code, rest of work like creating database, creating tables, assigning keys, etc, will be look over by the Entity framework.

The Code-First approach has there own principle or strategy. These strategies are the main backbone of code-first approach. These strategies are also called as database Initializers.

- **CreateDatabaseIfNotExists**
- **DropCreateDatabaseWhenModelChanges**
- **DropCreateDatabaseAlways**



Database Initializers

CreateDatabaseIfNotExists

- This is the **default** initializer. As the name suggests, it will create the database if none exists as per the configuration. However, if you change the model class and then run the application with this initializer, then it will throw an exception.

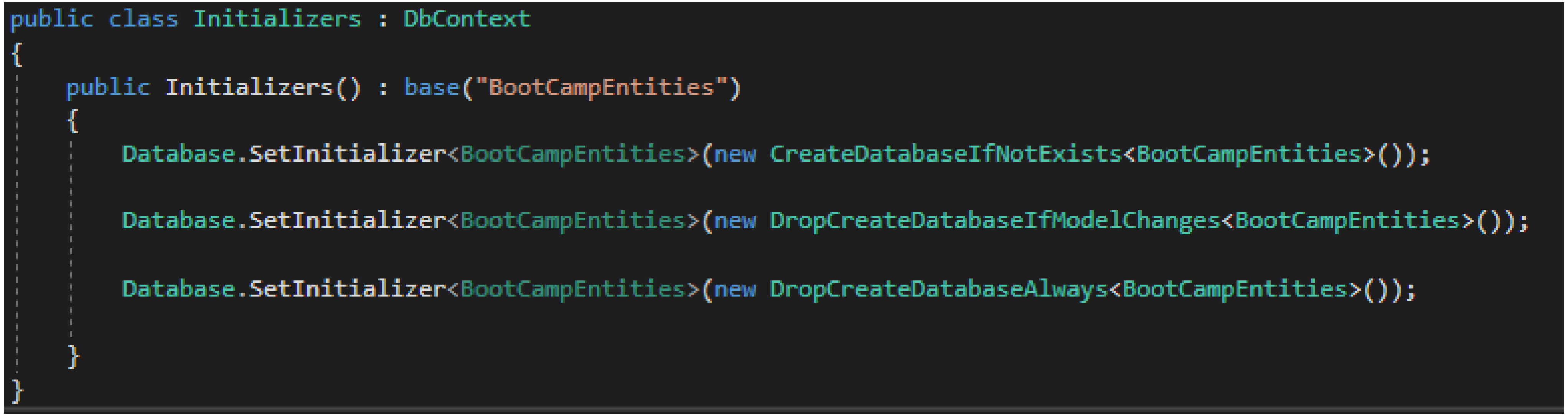
DropCreateDatabaseIfModelChanges

- This initializer drops an existing database and creates a new database, if your model classes (entity classes) have been changed. So, you don't have to worry about maintaining your database schema, when your model classes change.

DropCreateDatabaseAlways

- As the name suggests, this initializer drops an existing database every time you run the application, irrespective of whether your model classes have changed or not. This will be useful when you want a fresh database every time you run the application, for example when you are developing the application.

Database Initializers



```
public class Initializers : DbContext
{
    public Initializers() : base("BootCampEntities")
    {
        Database.SetInitializer<BootCampEntities>(new CreateDatabaseIfNotExists<BootCampEntities>());

        Database.SetInitializer<BootCampEntities>(new DropCreateDatabaseIfModelChanges<BootCampEntities>());

        Database.SetInitializer<BootCampEntities>(new DropCreateDatabaseAlways<BootCampEntities>());

    }
}
```

Data Annotations

- Data Annotations is used to configure your model classes, which will highlight the most commonly needed configurations.

- Data Annotations are also understood by a number of .NET applications, such as ASP.NET MVC, which allows these applications to leverage the same annotations for client-side validations.

- Data Annotation attributes override default Code-First conventions.

System.ComponentModel.DataAnnotations includes the following attributes that impacts the nullability or size of the column:

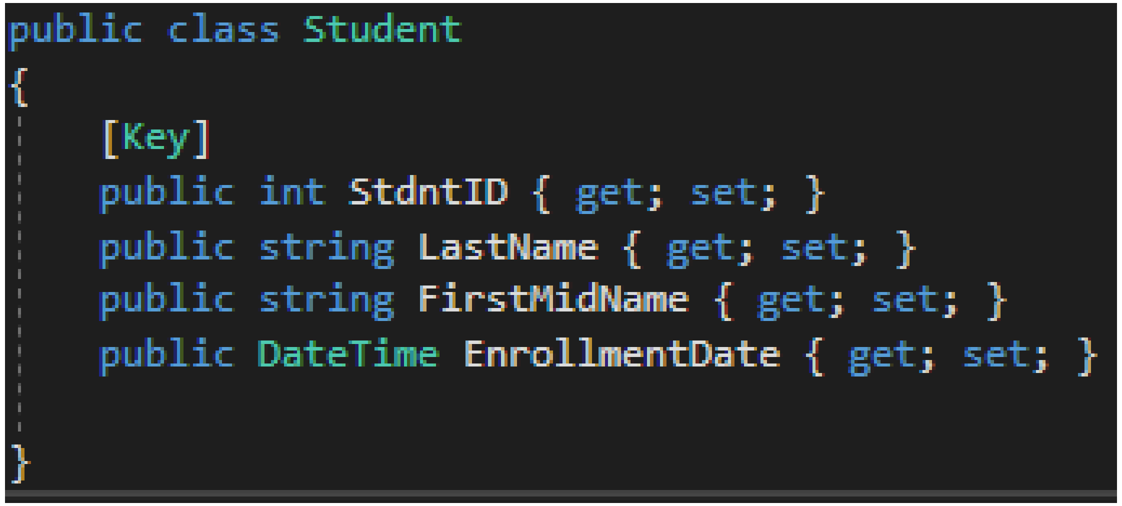
- Key
- Timestamp
- ConcurrencyCheck
- Required
- MinLength
- MaxLength
- StringLength

Data Annotations

Key

Entity Framework relies on every entity having a key value that it uses for tracking entities. One of the conventions that Code First depends on is how it implies which property is the key in each of the Code First classes.

The convention is to look for a property named “Id” or one that combines the class name and “Id”, such as “StudentId”. The property will map to a primary key column in the database. The Student, Course and Enrollment classes follow this convention.



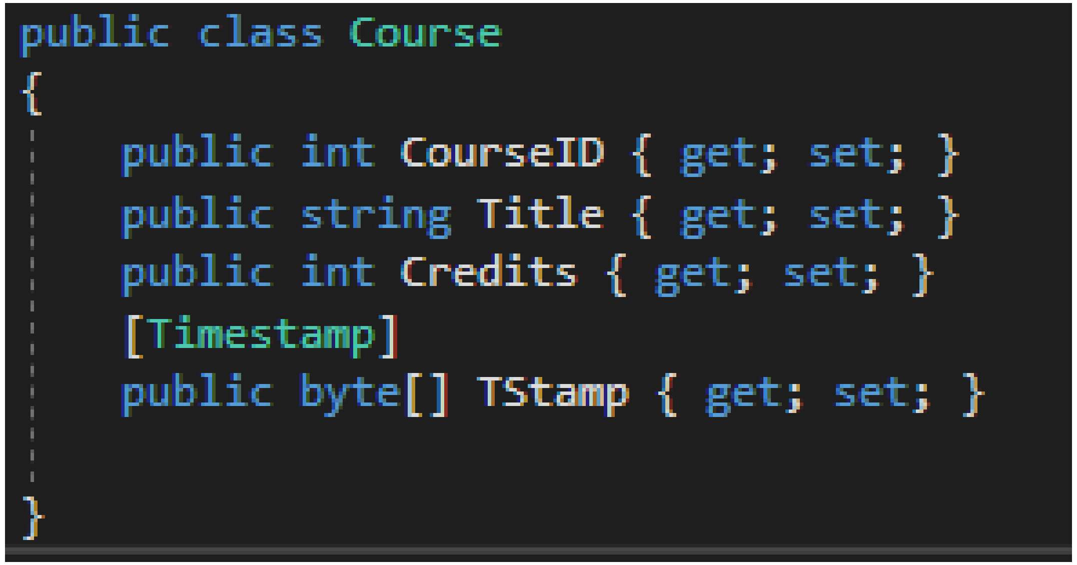
```
public class Student
{
    [Key]
    public int StdntID { get; set; }
    public string LastName { get; set; }
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }

}
```

Data Annotations

Timestamp

Code First will treat Timestamp properties the same as ConcurrencyCheck properties, but it will also ensure that the database field generated by Code First is non-nullable.

It's more common to use rowversion or timestamp fields for concurrency checking. But rather than using the ConcurrencyCheck annotation, you can use the more specific TimeStamp annotation as long as the type of the property is byte array. You can only have one timestamp property in a given class.

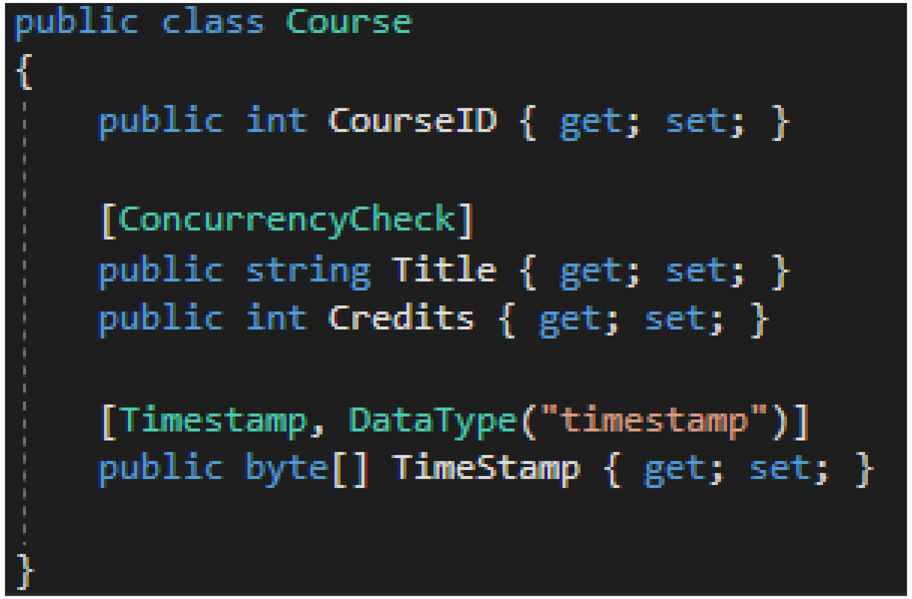


```
public class Course
{
    public int CourseID { get; set; }
    public string Title { get; set; }
    public int Credits { get; set; }
    [Timestamp]
    public byte[] TStamp { get; set; }

}
```

Data Annotations

ConcurrencyCheck

The ConcurrencyCheck annotation allows you to flag one or more properties to be used for concurrency checking in the database, when a user edits or deletes an entity. If you've been working with the EF Designer, this aligns with setting a property's ConcurrencyMode to Fixed.

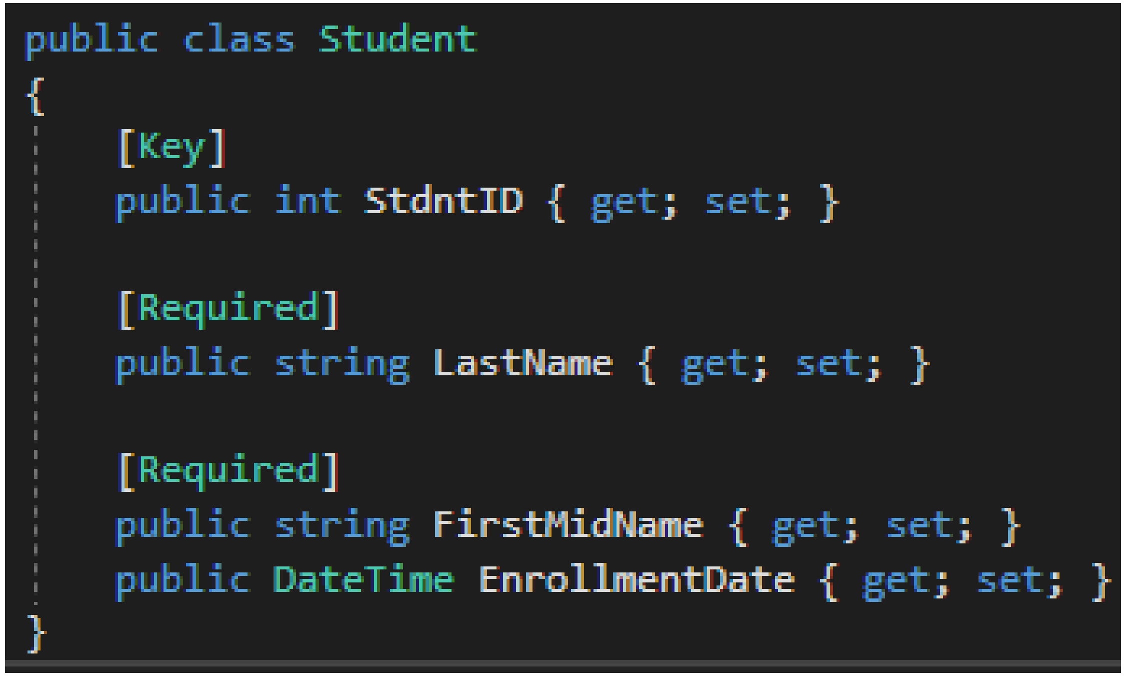


```
public class Course
{
    public int CourseID { get; set; }

    [ConcurrencyCheck]
    public string Title { get; set; }
    public int Credits { get; set; }

    [Timestamp, DataType("timestamp")]
    public byte[] TimeStamp { get; set; }

}
```

Data Annotations

Required

The Required annotation tells EF that a particular property is required. Let’s have a look at the following Student class in which Required id is added to the FirstMidName property. Required attribute will force EF to ensure that the property has data in it.



```
public class Student
{
    [Key]
    public int StdntID { get; set; }

    [Required]
    public string LastName { get; set; }

    [Required]
    public string FirstMidName { get; set; }
    public DateTime EnrollmentDate { get; set; }
}
```

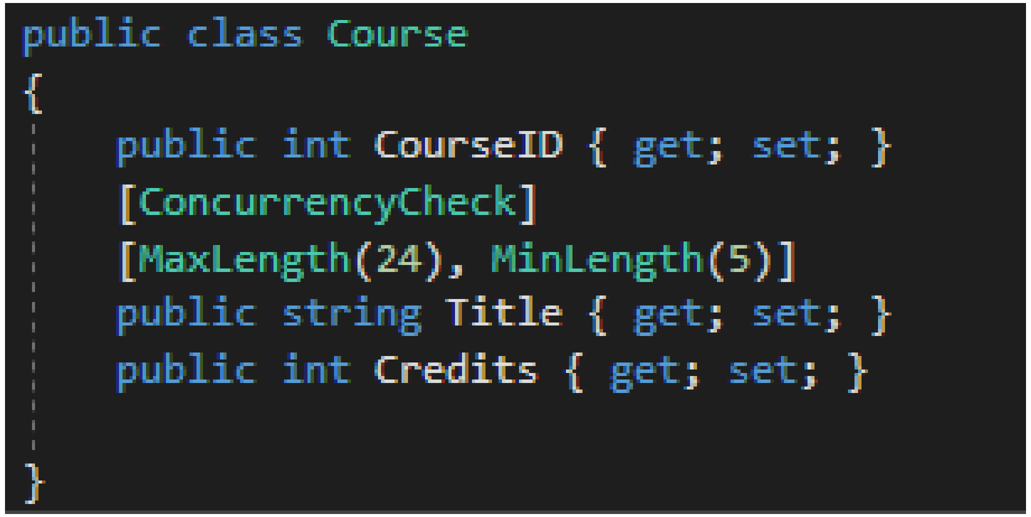
Data Annotations

MinLength

The MinLength attribute allows you to specify additional property validations, just as you did with MaxLength. MinLength attribute can also be used with MaxLength attribute as shown in the following code.

MaxLength

The MaxLength attribute allows you to specify additional property validations. It can be applied to a string or array type property of a domain class. EF Code First will set the size of a column as specified in MaxLength attribute.

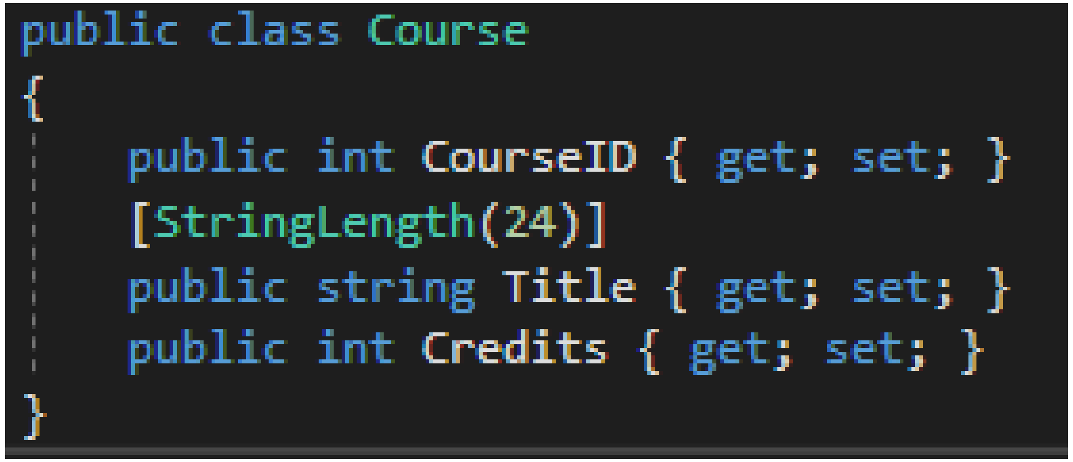


```
public class Course
{
    public int CourseID { get; set; }
    [ConcurrencyCheck]
    [MaxLength(24), MinLength(5)]
    public string Title { get; set; }
    public int Credits { get; set; }

}
```

Data Annotations

StringLength

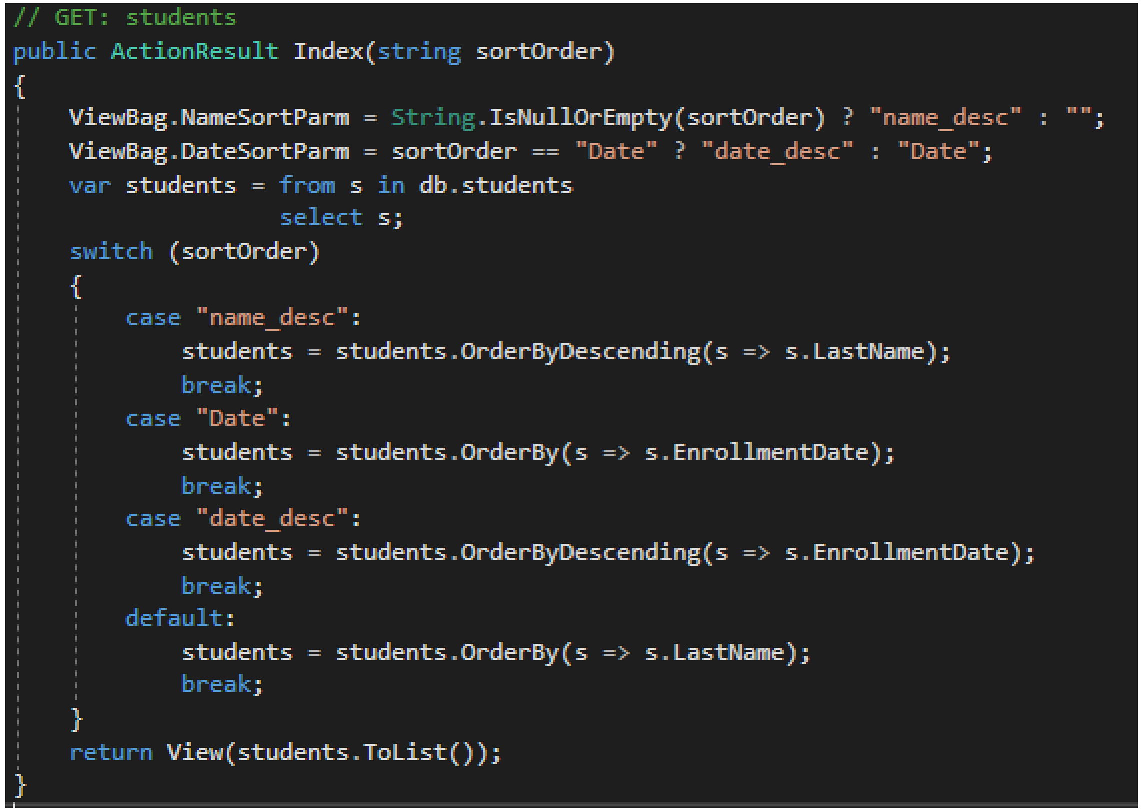
StringLength also allows you to specify additional property validations like MaxLength. The difference being StringLength attribute can only be applied to a string type property of Domain classes.



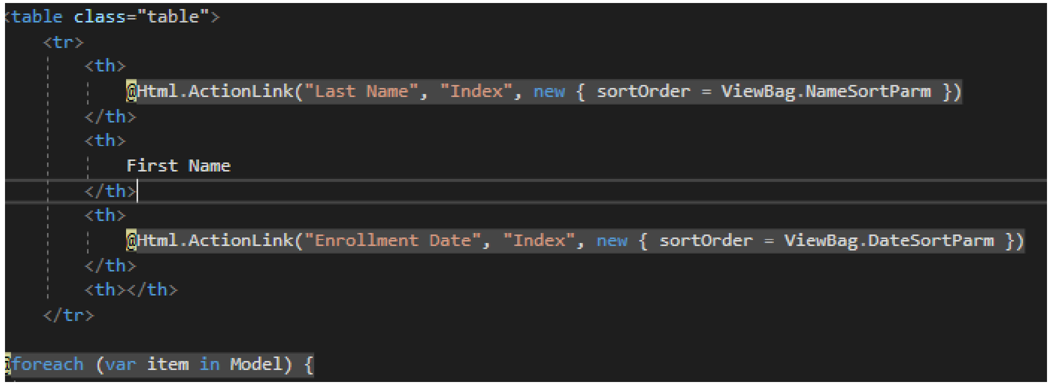
```
public class Course
{
    public int CourseID { get; set; }
    [StringLength(24)]
    public string Title { get; set; }
    public int Credits { get; set; }
}
```

Data Filtering

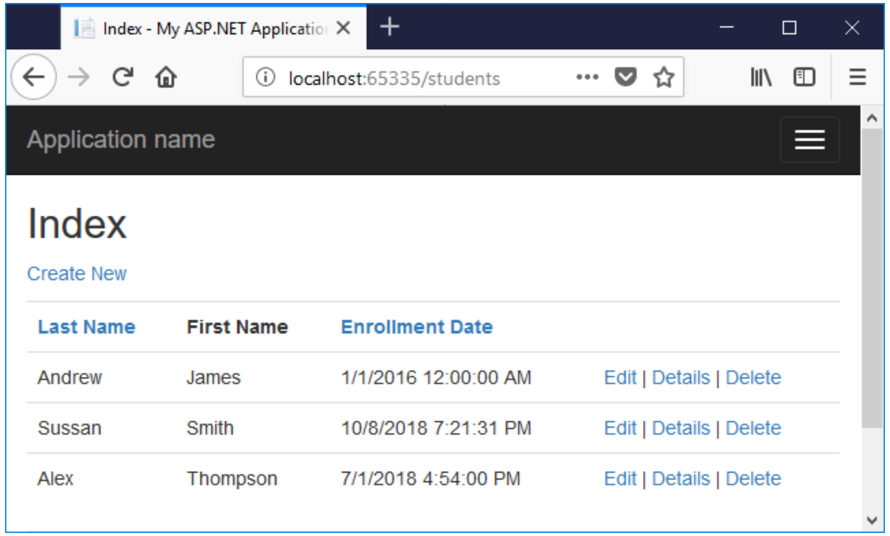
Now lets understand how to add sorting to the **Students** Index page.



```
// GET: students
public ActionResult Index(string sortOrder)
{
    ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewBag.DateSortParm = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in db.students
                   select s;
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }
    return View(students.ToList());
}
```



```
<table class="table">
    <tr>
        <th>
            @Html.ActionLink("Last Name", "Index", new { sortOrder = ViewBag.NameSortParm })
        </th>
        <th>
            First Name
        </th>
        <th>
            @Html.ActionLink("Enrollment Date", "Index", new { sortOrder = ViewBag.DateSortParm })
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
```



Index

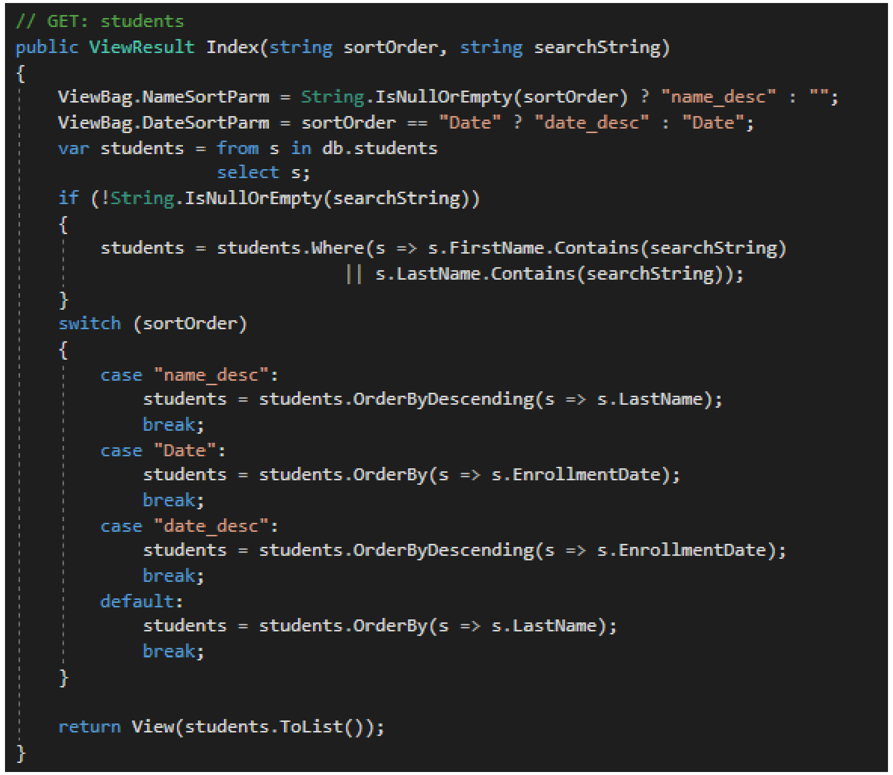
Create New

Last Name	First Name	Enrollment Date	
Andrew	James	1/1/2016 12:00:00 AM	Edit | Details | Delete
Sussan	Smith	10/8/2018 7:21:31 PM	Edit | Details | Delete
Alex	Thompson	7/1/2018 4:54:00 PM	Edit | Details | Delete

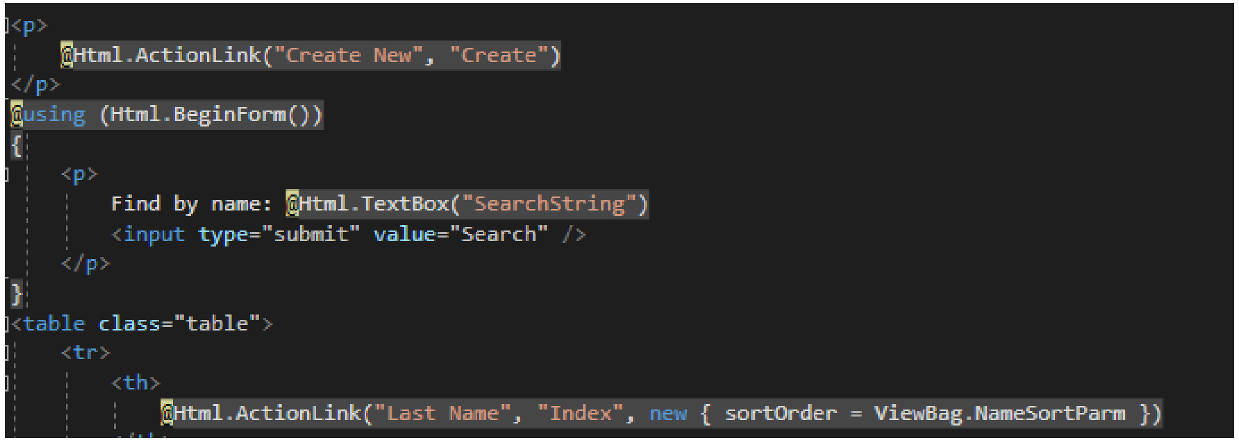
@ Copyright 2019, Summitworks Technologies Inc.

Data Filtering

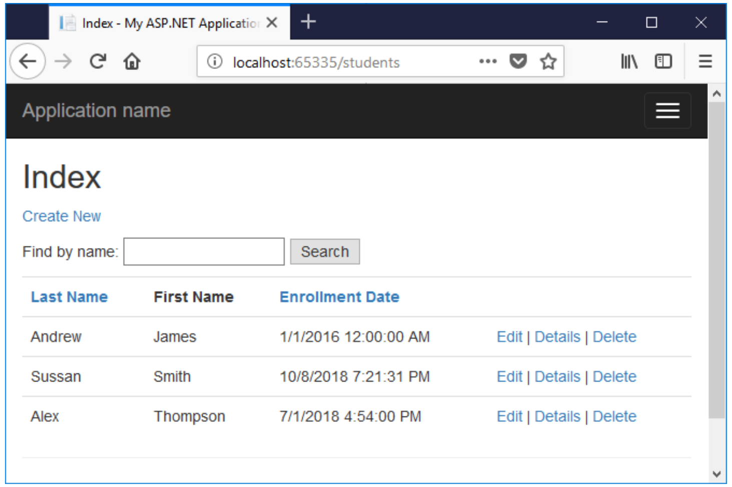
Now lets understand how to add filtering to the **Students** Index page.



```
// GET: students
public ViewResult Index(string sortOrder, string searchString)
{
    ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "name_desc" : "";
    ViewBag.DateSortParm = sortOrder == "Date" ? "date_desc" : "Date";
    var students = from s in db.students
                   select s;
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s => s.FirstName.Contains(searchString)
                              || s.LastName.Contains(searchString));
    }
    switch (sortOrder)
    {
        case "name_desc":
            students = students.OrderByDescending(s => s.LastName);
            break;
        case "Date":
            students = students.OrderBy(s => s.EnrollmentDate);
            break;
        case "date_desc":
            students = students.OrderByDescending(s => s.EnrollmentDate);
            break;
        default:
            students = students.OrderBy(s => s.LastName);
            break;
    }

    return View(students.ToList());
}
```



```
<p>
    @Html.ActionLink("Create New", "Create")
</p>
@using (Html.BeginForm())
{
    <p>
        Find by name: @Html.TextBox("SearchString")
        <input type="submit" value="Search" />
    </p>
}
<table class="table">
    <tr>
        <th>
            @Html.ActionLink("Last Name", "Index", new { sortOrder = ViewBag.NameSortParm })
```



Index

Create New

Find by name: [ ] Search

Last Name	First Name	Enrollment Date	
Andrew	James	1/1/2016 12:00:00 AM	Edit | Details | Delete
Sussan	Smith	10/8/2018 7:21:31 PM	Edit | Details | Delete
Alex	Thompson	7/1/2018 4:54:00 PM	Edit | Details | Delete