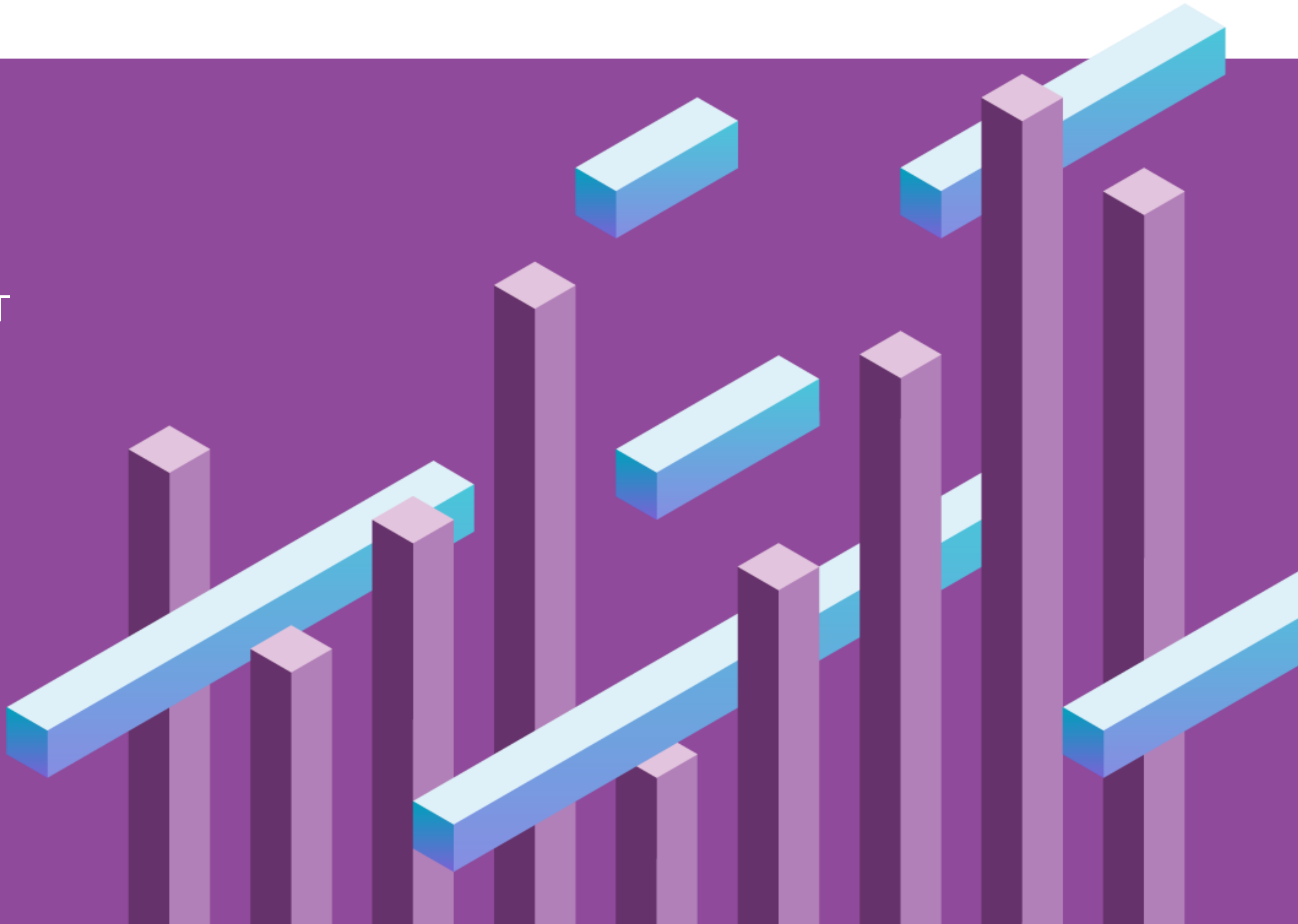
C#: Using LINQ Queries & Operators

Eric Fisher
AUTHOR, PLURALSIGHT

@EricJamesFisher



LINQ Queries

LINQ allows you to query data from different types of data sources using one syntax rather than needing to learn a variety of different syntaxes.

Datasource

LINQ can use any type of IEnumerable or IEnumerable<T> as a datasource.

Over the next few slides we'll work through a variety of ways to query a datasource using LINQ

Datasource

Our example datasource will be a simple array of strings

Example.cs

```
public void GetUserRecords()
{
    var users = new string[] { "Emily", "Jacob", "Thomas" };
}
```

Select Query

Select queries contain instructions for retrieving information from our datasource

Example.cs

```
public void GetUserRecords()
{
    var users = new string[] { "Emily", "Jacob", "Thomas" };

    var userQuery =
        from user in users
        select user;
}
```

Note: userQuery will contain our query information NOT the results of that query.

Select Query Structure

The from clause specifies our datasource

Example.cs

```
public void GetUserRecords()
{
    var users = new string[] { "Emily", "Jacob", "Thomas" };

    var userQuery =
        from user in users
        select user;
}
```

from is followed by what variable name we'll use for each object in our datasource

Users is our datasource. (in this example an array of strings)

Select Query Structure

The select clause specifies the “shape” or type of each element to be returned

Example.cs

```
public void GetUserRecords()
{
    var users = new string[] { "Emily", "Jacob", "Thomas" };

    var userQuery =
        from user in users
        select user;
}
```

Since we’re selecting user, and user is a string, we’ll be returning a collection of strings

When Do Queries Execute?

Queries contain only instructions, data isn't grabbed until the query is executed

Example.cs

```
public void GetUserRecords()
{
    var users = new string[] { "Emily", "Jacob", "Thomas" };

    var userQuery =
        from user in users
        select user;

    userQuery.Count();
}
```

The query has not executed at this point

The query will execute when Count is called

Using userQuery in a foreach loop, calling Count, ToList, ToArray, etc will all trigger the query to execute.

Where Clause

The where clause allows us to grab only a subset of the data conditionally

Example.cs

```
public void GetUserRecords()
{
    var users = new string[] { "Emily", "Jacob", "Thomas" };

    var userQuery =
        from user in users
        where user.Contains("m")
        select user;

    userQuery.Count();
}
```

Only records that match conditions following the where operator will be returned

Count will now return 2 instead of three, because "Jacob" doesn't pass our where condition

Orderby Clause

The orderby clause allows us to arrange our results in a given order

Example.cs

```
…
var users = new string[] { "Emily", "Jacob", "Thomas" };

var userQuery =
    from user in users
    where user.Contains("m")
    orderby user.Length ascending
    select user;

foreach (var user in userQuery)
{
    Console.WriteLine(user);
}
…
```

We can arrange our result in ascending or descending order

Our orderby will arrange our results based on the length of each string

Orderby Output Example

Example.cs

```
…
var users = new string[] { "Emily", "Jacob", "Thomas" };

var userQuery =
    from user in users
    where user.Contains("m")
    orderby user.Length ascending
    select user;

foreach (var user in userQuery)
{
    Console.WriteLine(user);
}
…
```

Example Output

```
> dotnet run
Emily
Thomas
>
```

Groupby Clause

The groupby clause allows us to group our results into sets of Key Value groups

Example.cs

```
…
var userQuery =
    from user in users
    group user by user.Length into userGroup
    select userGroup;

foreach (var userGroup in userQuery)
{
    Console.WriteLine("{0} characters long", userGroup.Key);
    foreach (var user in userGroup)
    {
        Console.WriteLine(user);
…
```

We are grouping our users by their string.Length into a new object userGroup

Each key value group will have a Key property which is the value that set was grouped by

Iterating Through Group Query Results

Example.cs

```
…
var userQuery =
    from user in users
    group user by user.Length into userGroup
    select userGroup;

foreach (var userGroup in userQuery)
{
    Console.WriteLine("{0} characters long", userGroup.Key);
    foreach (var user in userGroup)
    {
        Console.WriteLine(user);
…
```

Our userGroup object contains one or more sets of key value groups so we need to foreach through our sets of groups, as well as the groups themselves

Groupby Output Example

Example.cs

```
…
var userQuery =
    from user in users
    group user by user.Length into userGroup
    select userGroup;

foreach (var userGroup in userQuery)
{
    Console.WriteLine("{0} characters long", userGroup.Key);
    foreach (var user in userGroup)
    {
        Console.WriteLine(user);
…
```

Example Output

```
> dotnet run
5 characters long
Emily
Jacob
6 characters long
Thomas
>
```

Summary

LINQ allows you to query data from different types of datasources using a common syntax

LINQ can use any `IEnumberable` or `IEnumberable<T>` as a datasource

Queries don't execute at the time you create the query, they will execute when the results are needed

Where clauses filter your results conditionally

Orderby clauses arrange your results

Groupby clauses put your results into a collection of groups