LINQ

Language Integrated Query

Canberra Institute of Technology

aie

SPECIALIST EDUCATORS IN
GAMES, ANIMATION & FILM VFX

Contents

- What is LINQ?
- Your first LINQ Query
- Using LINQ to analyse data
- Summary

Canberra Institute of Technology

aie

SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

What is LINQ?

- LINQ was introduced to .NET 3.5 to work with data in powerful ways that were not available before

- LINQ is a mammoth topic, so we will only be talking about the basics

What is LINQ?

- LINQ is a query based language included with .NET 3.5

- LINQ enables you to easily query a data source for entries that meet set conditions, in a similar way to how SQL allows you to query a database

This enables you to do some things really easily:

- Calculate how many even or odd numbers are in a collection
- Create new collections where elements meet a condition.
- Iterate through items of a collection in a sorted manner, without actually sorting the collection
- Compare how many items in one collection match items in another collection.
- Lots and lots of complicated analasys fairly easy.

Canberra Institute of Technology

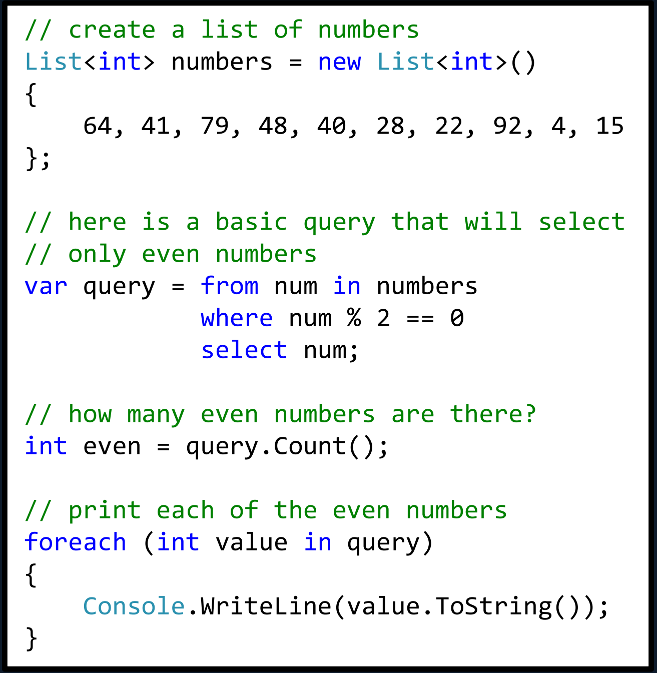
aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

What is a datasource?

- A data source is anything that stores a collection of data, such as:
  - Container types such as a List<T> or Dictionary<key,value>
  - Relational Database – often queried through SQL, but we can use LINQ as well
  - XML files
  - And more…
- We will only be covering basic LINQ queries over a collection

Your first LINQ Query

- The following code snippet creates a list of numbers
- A LINQ query has been created for this
  - We can get the number of items that matched the condition via the Count() function
  - We can iterate through each other items retrieved by our query

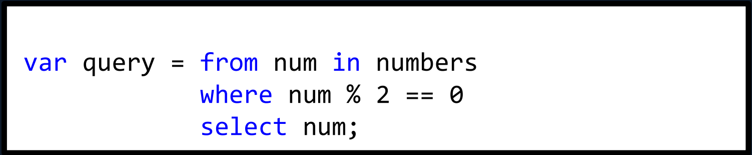


```
// create a list of numbers
List<int> numbers = new List<int>()
{
    64, 41, 79, 48, 40, 28, 22, 92, 4, 15
};

// here is a basic query that will select
// only even numbers
var query = from num in numbers
            where num % 2 == 0
            select num;

// how many even numbers are there?
int even = query.Count();

// print each of the even numbers
foreach (int value in query)
{
    Console.WriteLine(value.ToString());
}
```

Behind the scenes: Method Syntax

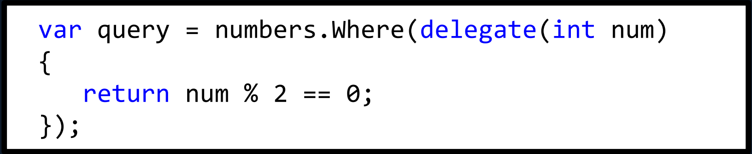
- When compiling your project, the query is converted to “method” syntax
  - Alternatively, you can skip LINQ and use method syntax instead. While method syntax looks easier, complex queries will quickly make method syntax impractical!

Original LINQ Query



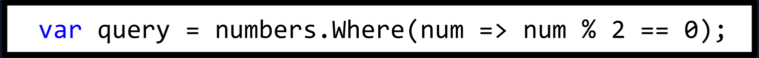
```
var query = from num in numbers
            where num % 2 == 0
            select num;
```

Using a lambda function (long hand)



```
var query = numbers.Where(delegate(int num)
{
   return num % 2 == 0;
});
```

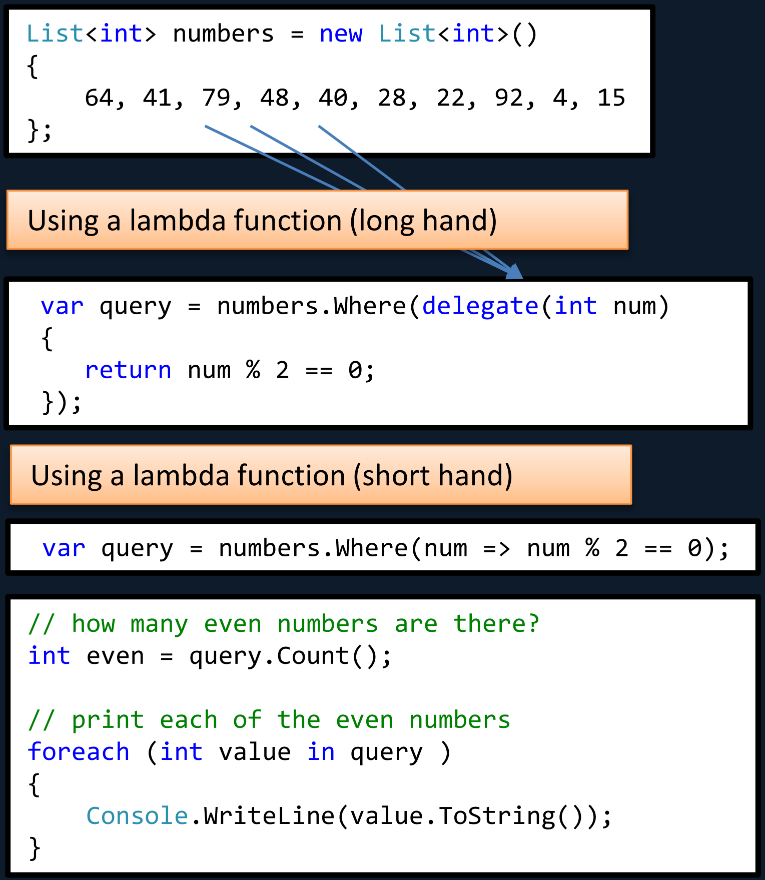
Using a lambda function (short hand)



```
var query = numbers.Where(num => num % 2 == 0);
```

Method Syntax

- This “Where” function takes a lambda as its parameter
- The lambda function is called once for each value in the collection “numbers”
- The value is passed in as the parameter int num to our lambda function
- The lambda must return true if the value should be selected



```
List<int> numbers = new List<int>()
{
    64, 41, 79, 48, 40, 28, 22, 92, 4, 15
};
```

Using a lambda function (long hand)

```
 var query = numbers.Where(delegate(int num)
 {
    return num % 2 == 0;
 });
```

Using a lambda function (short hand)

```
 var query = numbers.Where(num => num % 2 == 0);
```

```
// how many even numbers are there?
int even = query.Count();

// print each of the even numbers
foreach (int value in query )
{
    Console.WriteLine(value.ToString());
}
```

Canberra Institute of Technology

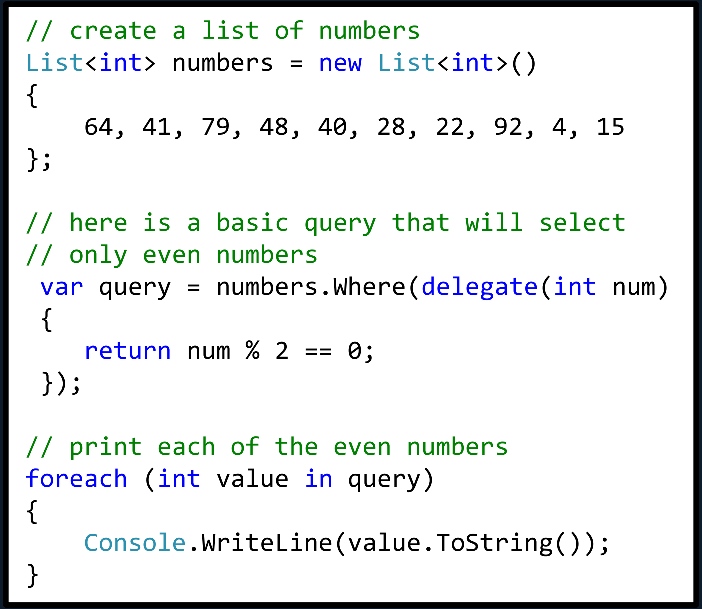
aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Please explain!

- Any object that inherits from IEnumerable has the option of providing functions for use with LINQ queries

- The “where” function that we’ve just seen is an extension method for IEnumerable<T> which is called for each item in the collection

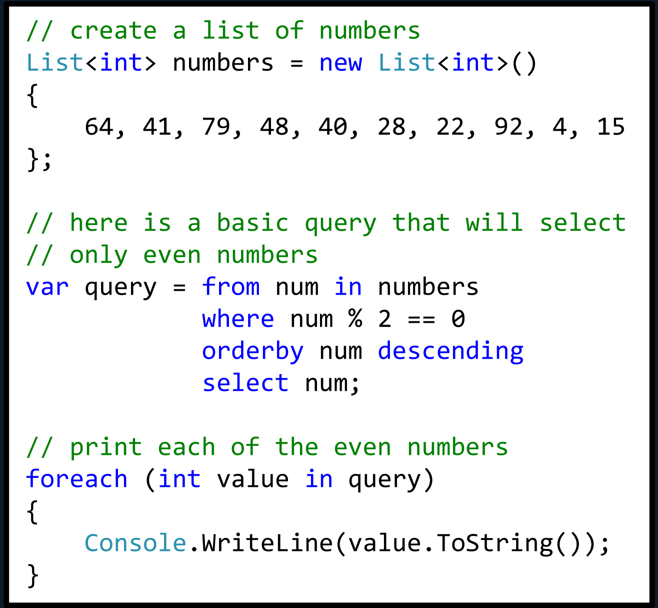
- The “where” function returns a new IEnumerable object, which will only enumerate of the items that returned true



```
// create a list of numbers
List<int> numbers = new List<int>()
{
    64, 41, 79, 48, 40, 28, 22, 92, 4, 15
};

// here is a basic query that will select
// only even numbers
 var query = numbers.Where(delegate(int num)
 {
    return num % 2 == 0;
 });

// print each of the even numbers
foreach (int value in query)
{
    Console.WriteLine(value.ToString());
}
```

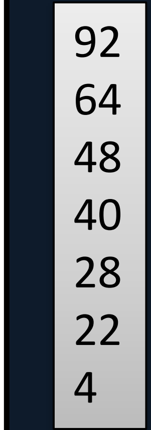
2nd LINQ Query

- We’ve now modified the query to order the items in descending order…



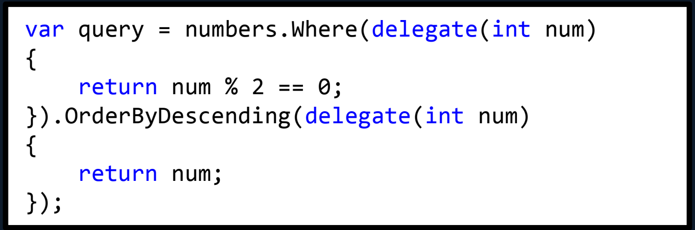
```
// create a list of numbers
List<int> numbers = new List<int>()
{
    64, 41, 79, 48, 40, 28, 22, 92, 4, 15
};

// here is a basic query that will select
// only even numbers
var query = from num in numbers
            where num % 2 == 0
            orderby num descending
            select num;

// print each of the even numbers
foreach (int value in query)
{
    Console.WriteLine(value.ToString());
}
```

outputs



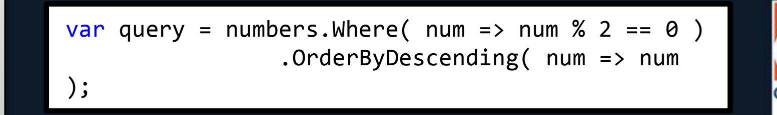
```
92
64
48
40
28
22
4
```

Same query – using method syntax (long hand)



```
var query = numbers.Where(delegate(int num)
{
    return num % 2 == 0;
}).OrderByDescending(delegate(int num)
{
    return num;
});
```

Same query – using method syntax (short hand)



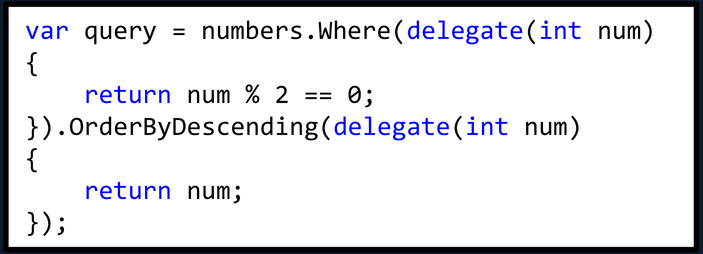
```
var query = numbers.Where( num => num % 2 == 0 )
                   .OrderByDescending( num => num
);
```

2nd LINQ Query Explained

- The "Where" function as we've seen returns an IEnumerable<T> type of object, which we can then call on another function

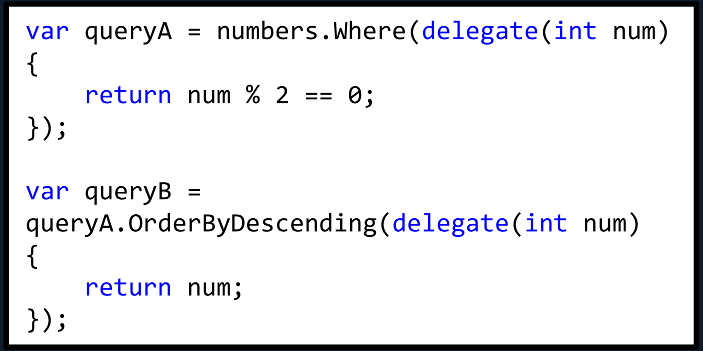
- In this case, we are calling OrderByDescending

Same query – using method syntax (long hand)



```
var query = numbers.Where(delegate(int num)
{
    return num % 2 == 0;
}).OrderByDescending(delegate(int num)
{
    return num;
});
```

Same as above, shows the return value of queryA can be further "filtered"

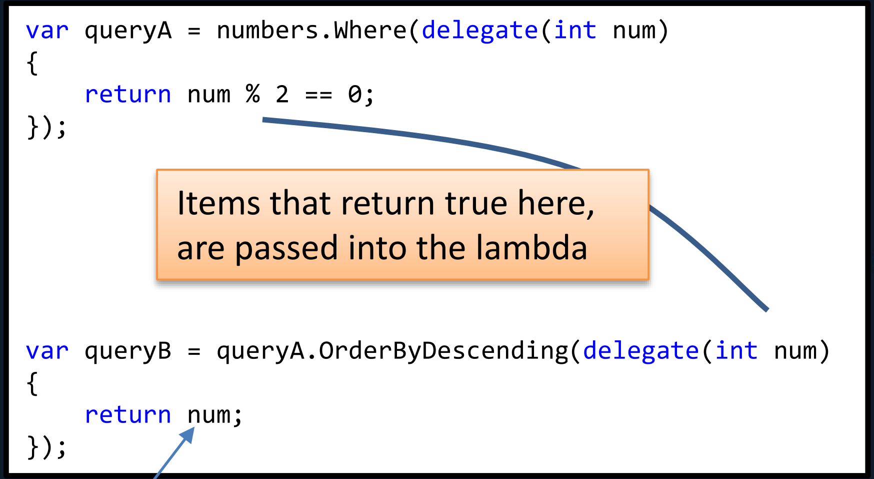


```
var queryA = numbers.Where(delegate(int num)
{
    return num % 2 == 0;
});

var queryB =
queryA.OrderByDescending(delegate(int num)
{
    return num;
});
```

2nd LINQ Query Explained

- OrderByDescending takes a lambda function where each item available to the “queryA” is passed in as a parameter (num)

- The value to be sorted is returned by the lambda



```
var queryA = numbers.Where(delegate(int num)
{
    return num % 2 == 0;
});


var queryB = queryA.OrderByDescending(delegate(int num)
{
    return num;
});
```

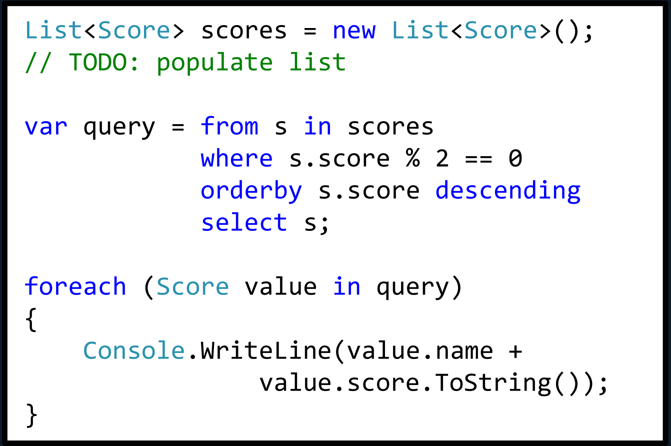
Items that return true here, are passed into the lambda

Why are we just returning the same value?

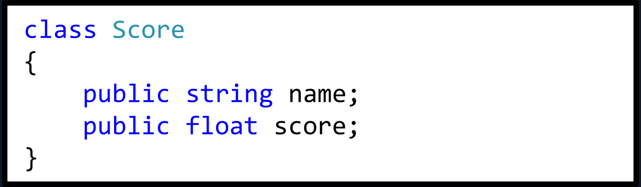
Order By – Return value explained

- Here, rather than a list of integers, we have a list of “Score” objects

- To sort the list of scores, we need to tell the query what value to use for sorting, where returning only even scores, sorted in descending order

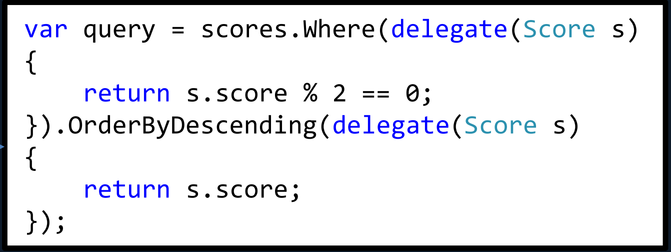


```
List<Score> scores = new List<Score>();
// TODO: populate list

var query = from s in scores
            where s.score % 2 == 0
            orderby s.score descending
            select s;

foreach (Score value in query)
{
    Console.WriteLine(value.name +
                value.score.ToString());
}
```



```
class Score
{
    public string name;
    public float score;
}
```

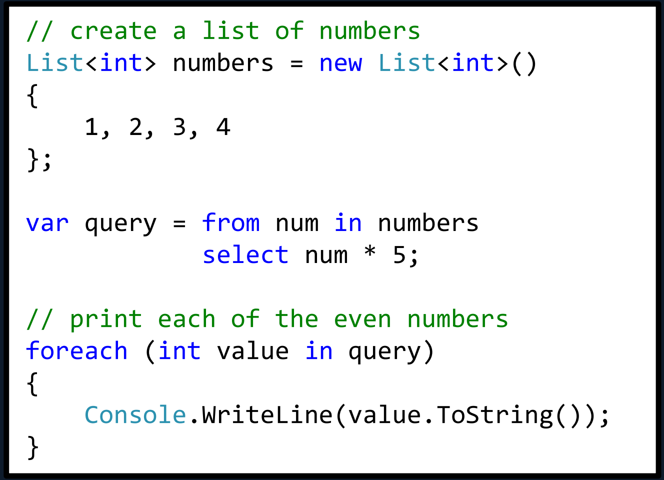
Method syntax



```
var query = scores.Where(delegate(Score s)
{
    return s.score % 2 == 0;
}).OrderByDescending(delegate(Score s)
{
    return s.score;
});
```

3rd LINQ Query

- You can modify the “Select” statement to return new values or data types.. For a simple example, we can just multiply the value by 5

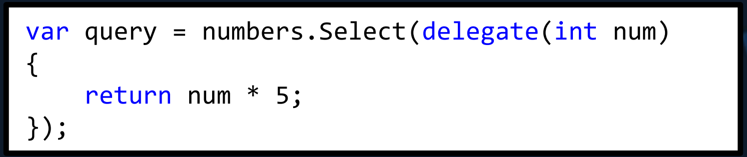


```
// create a list of numbers
List<int> numbers = new List<int>()
{
    1, 2, 3, 4
};

var query = from num in numbers
            select num * 5;

// print each of the even numbers
foreach (int value in query)
{
    Console.WriteLine(value.ToString());
}
```

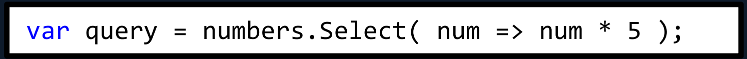
outputs

```
5
10
15
20
```

- With method syntax, there is a new Select function for us to use



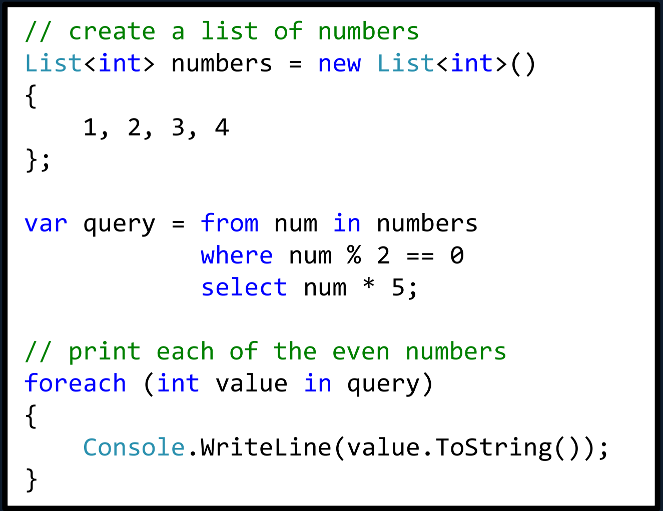
```
var query = numbers.Select(delegate(int num)
{
    return num * 5;
});
```



```
var query = numbers.Select( num => num * 5 );
```

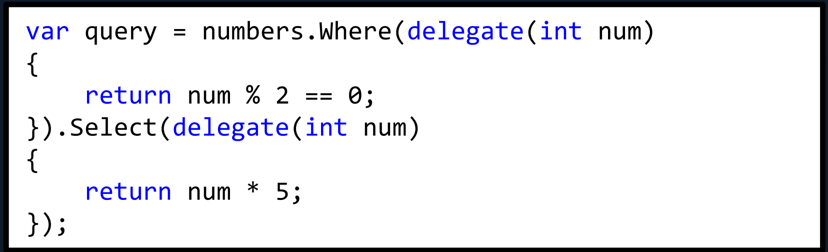
4th LINQ Query

- Lets put our "where" statement to get only even numbers back in…

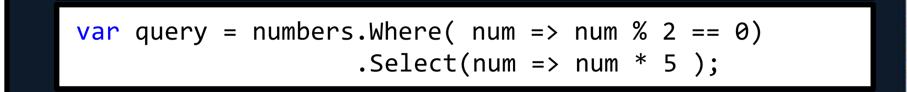


```
// create a list of numbers
List<int> numbers = new List<int>()
{
    1, 2, 3, 4
};

var query = from num in numbers
            where num % 2 == 0
            select num * 5;

// print each of the even numbers
foreach (int value in query)
{
    Console.WriteLine(value.ToString());
}
```

- With method syntax, there is a new Select function for use to use

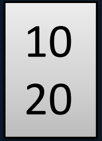


```
var query = numbers.Where(delegate(int num)
{
    return num % 2 == 0;
}).Select(delegate(int num)
{
    return num * 5;
});
```



```
var query = numbers.Where( num => num % 2 == 0)
                   .Select(num => num * 5 );
```

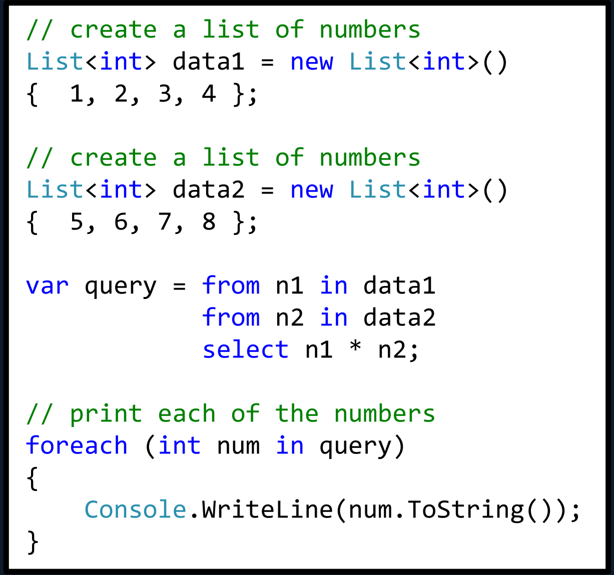
outputs



```
10
20
```

5th LINQ Query

- You can query data from multiple datasets…
  - This example multiplies each number from data 1 with each number in data2, because there are 4 numbers in each, 4 x 4 = 16 results



```
// create a list of numbers
List<int> data1 = new List<int>()
{  1, 2, 3, 4 };

// create a list of numbers
List<int> data2 = new List<int>()
{  5, 6, 7, 8 };

var query = from n1 in data1
            from n2 in data2
            select n1 * n2;

// print each of the numbers
foreach (int num in query)
{
    Console.WriteLine(num.ToString());
}
```

outputs

```
5
6
7
8
10
12
14
16
15
18
21
24
20
24
28
32
```

5th LINQ Query

- This gets a bit more complex with method syntax
  - The query compiles down to using SelectMany with takes 2 anonymous functions as arguments

```
var query = data1.SelectMany(delegate(int n1)
{
    return data2;

},delegate(int n1, int n2)
{
    return new { n1, n2 };

}).Select(delegate( dynamic value )
{
    return value.n1 * value.n2;
});
```

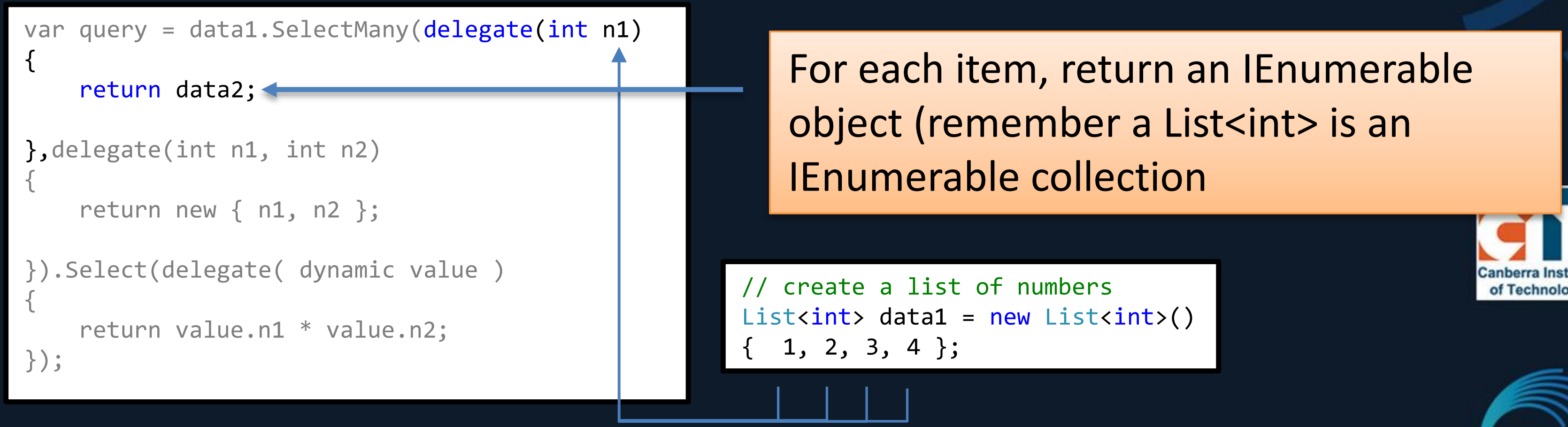
Yeah, ok, what is this mess!

Let us explain over the next few slides!

```
var query = data1.SelectMany(n1 => data2, (n1, n2) => new { n1, n2 })
                 .Select( pair => pair.n1 * pair.n2);
```

5th LINQ Query

- The first argument, our anonymous function that is highlighted, is the same for the select method
  - This function is called once for each item in data1



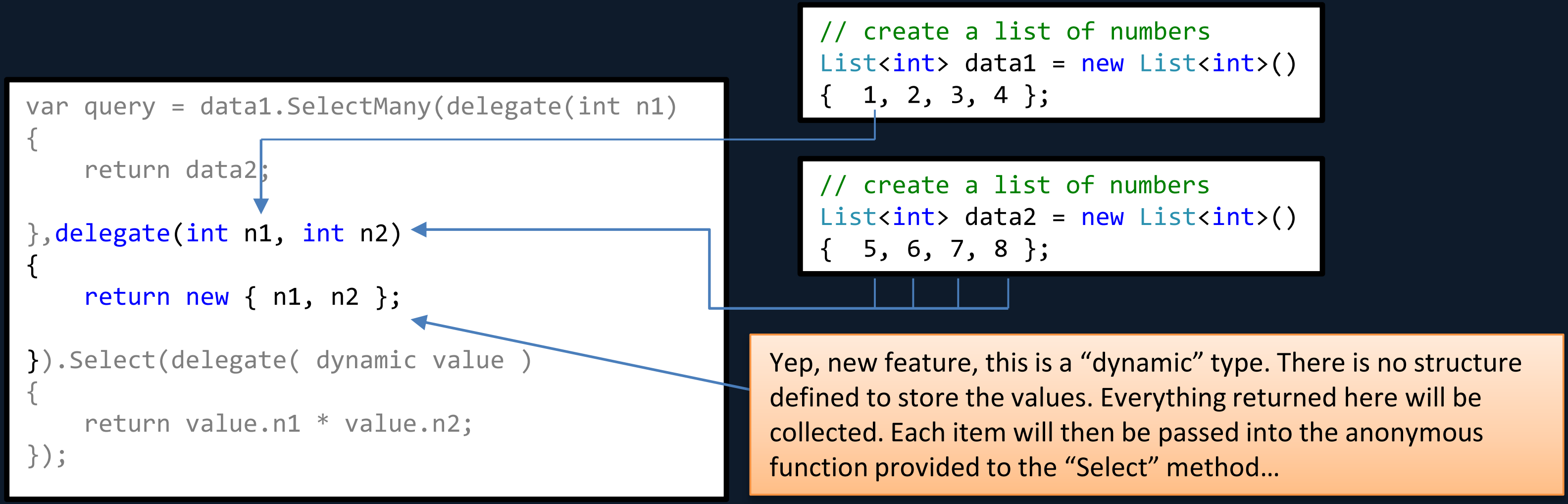
```
var query = data1.SelectMany(delegate(int n1)
{
    return data2;

},delegate(int n1, int n2)
{
    return new { n1, n2 };

}).Select(delegate( dynamic value )
{
    return value.n1 * value.n2;
});
```

For each item, return an IEnumerable object (remember a List<int> is an IEnumerable collection

```
// create a list of numbers
List<int> data1 = new List<int>()
{  1, 2, 3, 4 };
```

5th LINQ Query

- The second argument, our anonymous function that is highlighted is then called for all items in data2…



```
var query = data1.SelectMany(delegate(int n1)
{
    return data2;

},delegate(int n1, int n2)
{
    return new { n1, n2 };

}).Select(delegate( dynamic value )
{
    return value.n1 * value.n2;
});
```

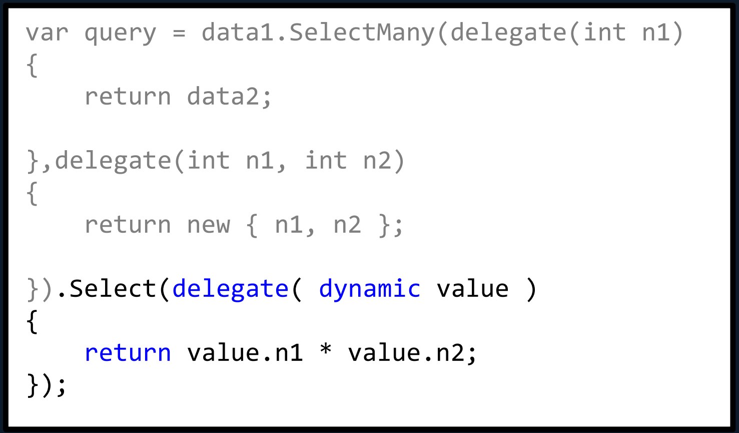
```
// create a list of numbers
List<int> data1 = new List<int>()
{  1, 2, 3, 4 };
```

```
// create a list of numbers
List<int> data2 = new List<int>()
{  5, 6, 7, 8 };
```

Yep, new feature, this is a “dynamic” type. There is no structure defined to store the values. Everything returned here will be collected. Each item will then be passed into the anonymous function provided to the “Select” method…

5th LINQ Query

- Finally, we can invoke the Select method. Unlike our 4th LINQ Query, this time we have access to both n1 and n2 from each dataset

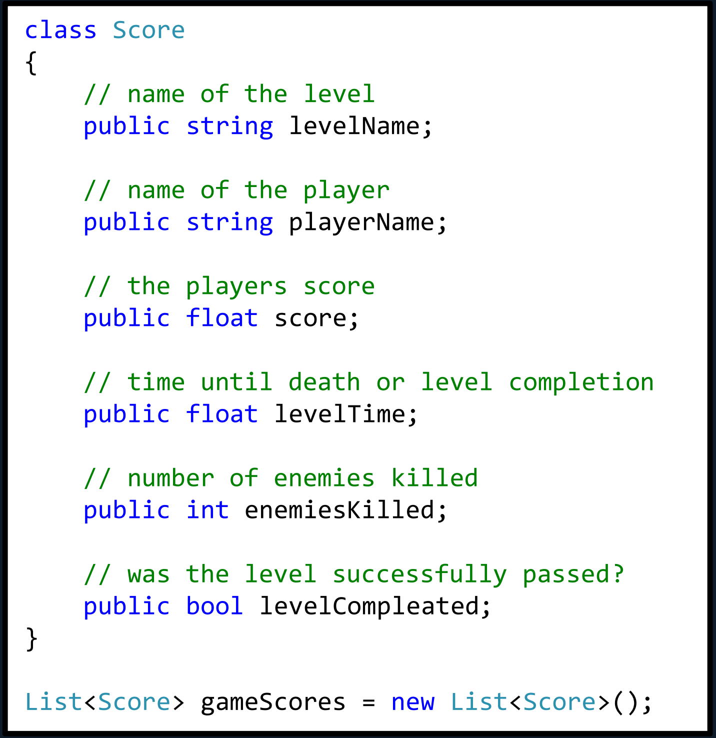


```
var query = data1.SelectMany(delegate(int n1)
{
    return data2;

},delegate(int n1, int n2)
{
    return new { n1, n2 };

}).Select(delegate( dynamic value )
{
    return value.n1 * value.n2;
});
```

Analysing Data

- Performing some basic queries on a list or 2 of integers only goes so far, and you can easily write alternative methods for achieving the same results

- Lets look at some more interesting information – lets assume that whenever the player dies, or completes the level, this information is recorded and added to our collection



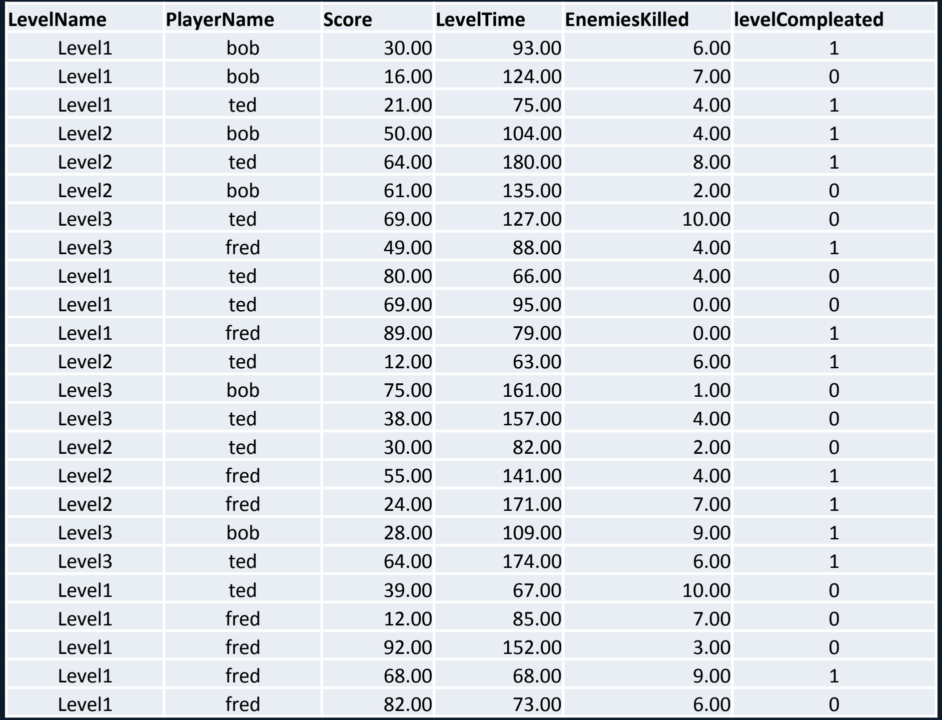
```
class Score
{
    // name of the level
    public string levelName;

    // name of the player
    public string playerName;

    // the players score
    public float score;

    // time until death or level completion
    public float levelTime;

    // number of enemies killed
    public int enemiesKilled;

    // was the level successfully passed?
    public bool levelCompleated;
}

List<Score> gameScores = new List<Score>();
```

Data Set

- Lets assume we have collected some data and recorded the results

- The values have been someone randomly generated

- For meaningful information, you would want to obtain a much large collection of non-random data!



LevelName	PlayerName	Score	LevelTime	EnemiesKilled	levelCompleated
Level1	bob	30.00	93.00	6.00	1
Level1	bob	16.00	124.00	7.00	0
Level1	ted	21.00	75.00	4.00	1
Level2	bob	50.00	104.00	4.00	1
Level2	ted	64.00	180.00	8.00	1
Level2	bob	61.00	135.00	2.00	0
Level3	ted	69.00	127.00	10.00	0
Level3	fred	49.00	88.00	4.00	1
Level1	ted	80.00	66.00	4.00	0
Level1	ted	69.00	95.00	0.00	0
Level1	fred	89.00	79.00	0.00	1
Level2	ted	12.00	63.00	6.00	1
Level3	bob	75.00	161.00	1.00	0
Level3	ted	38.00	157.00	4.00	0
Level2	ted	30.00	82.00	2.00	0
Level2	fred	55.00	141.00	4.00	1
Level2	fred	24.00	171.00	7.00	1
Level3	bob	28.00	109.00	9.00	1
Level3	ted	64.00	174.00	6.00	1
Level1	ted	39.00	67.00	10.00	0
Level1	fred	12.00	85.00	7.00	0
Level1	fred	92.00	152.00	3.00	0
Level1	fred	68.00	68.00	9.00	1
Level1	fred	82.00	73.00	6.00	0

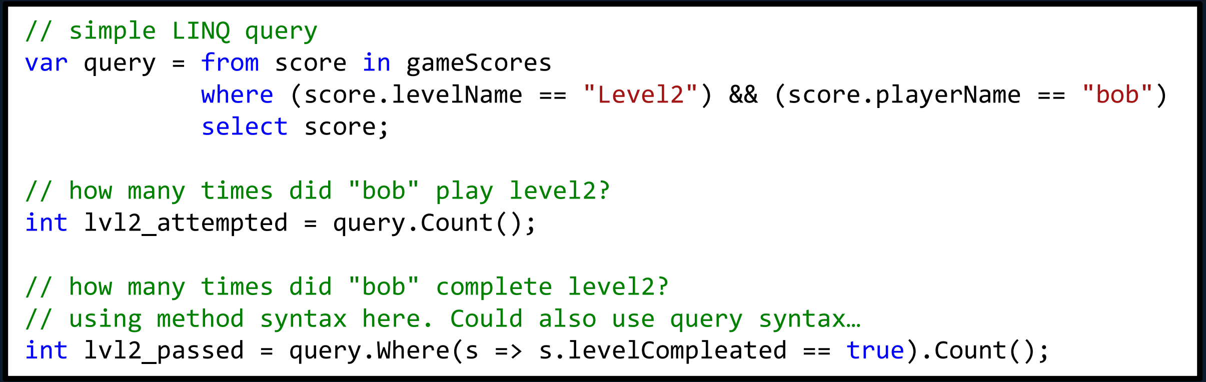
Analysing Data

- With this sort of information, we can analyse the data in interesting ways to collect information that couldn't be obtained by simply looking at a few records

- Simple stats can be obtained:
  - How many times a person attempted a level
  - Average number of kills on a particular level
  - Average time spent on a level
  - Average score obtained on a level overall
  - Average score obtained on a level for a particular player

Analysing Data

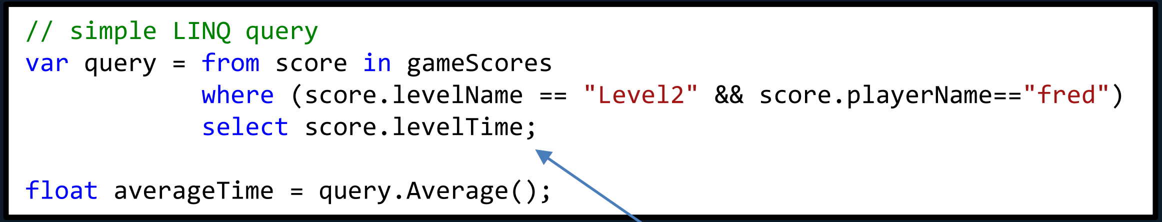
- How many times did someone play a level?



```
// simple LINQ query
var query = from score in gameScores
            where (score.levelName == "Level2") && (score.playerName == "bob")
            select score;

// how many times did "bob" play level2?
int lvl2_attempted = query.Count();

// how many times did "bob" complete level2?
// using method syntax here. Could also use query syntax…
int lvl2_passed = query.Where(s => s.levelCompleated == true).Count();
```

Analysing Data

- Average time “fred” spent on level2



```
// simple LINQ query
var query = from score in gameScores
            where (score.levelName == "Level2" && score.playerName=="fred")
            select score.levelTime;

float averageTime = query.Average();
```

Notice here, where selecting levelTime, which is a float. If we just selected “score”, the average could not be calculated…

Summary

- We’ve only scratched the surface regarding what LINQ can do, and have not covered many of the keywords available and its method syntax equivalent
- What we have done is covered the basics and provided enough foundation to learn more about this topic on your own