# Language Integrated Query (LINQ)

1

## OUTLINE

- Language-Integrated Query (LINQ)
- LINQ syntax
- LINQ projections
- Deferred execution
- Forcing immediate execution
- Joins
- Group join
- Aggregations
- The let clause
- The OfType method
- LINQ expression syntax and keyword reference

2

# LINQ

- Language-Integrated Query (**LINQ**) is a set of technologies based on the integration of query capabilities directly into the C# language
- You can use LINQ to consistently query data from Objects, collections that support IEnumerable, relational databases and XML, all using C# syntax
- LINQ queries are written using *query expression syntax* or *method syntax*
- Some queries can only use the method syntax (e.g., Count and Max)
- A query is **not** executed until you iterate over the query variable

3

3

# SCENARIO: NON-LINQ AND LINQ EQUIVALENT

Compare the two code snippets.

The left-hand snippet does **not** use LINQ.

The right-hand snippet uses LINQ.

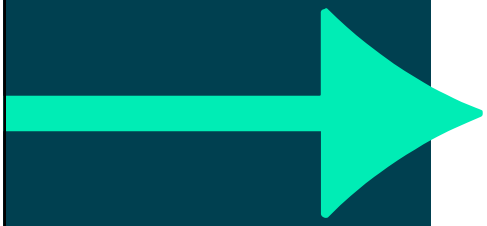
```
// Non-LINQ Example

// Specify the data source
int[] scores = { 97, 92, 81, 60 };

// Create a List to store the high scores
List<int> highScores = new();

// Iterate over the array to find the
// scores above 80 and add to the List
foreach (int score in scores)
{
    if(score > 80)
    {
        highScores.Add(score);
    }
}
// Iterate over the highScores List
foreach(int i in highScores)
{
    Console.Write(i + " ");
}
// Output: 97 92 81
```

```
// LINQ

// Specify the data source
int[] scores = { 97, 92, 81, 60 };

// Define the query expression
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query
foreach (int i in scoreQuery)
{
    Console.Write(i + " ");
}
// Output: 97 92 81
```

4

4

# LINQ SYNTAX



Query expression syntax:

```
// Specify the data source
int[] scores = { 97, 92, 81, 60 };

// Define the query expression
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query
foreach (int i in scoreQuery)
{
    Console.Write(i + " ");
}
// Output: 97 92 81
```

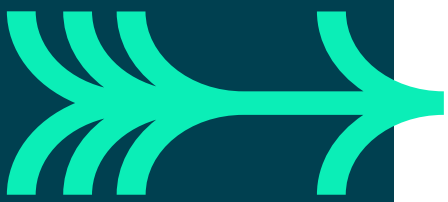
Query method syntax:

```
// Specify the data source
int[] scores = { 97, 92, 81, 60 };

// Define the query using method syntax
var query = scores.Where(score => score > 80)
                  .Select(score => score);

// Execute the query
foreach (var i in query)
{
    Console.Write(i + " ");
}
// Output: 97 92 81
```

5

5

# LINQ SYNTAX EXAMPLE



Query expression syntax:

```
// Specify the data source
List<Customer> customers = Customer.GetCustomers();

// Define the query using query expression syntax
var queryExpression = from c in customers
                      where c.City == "London"
                      orderby c.Balance
                      select c.CustomerName;

// Execute the query
foreach (var c in queryExpression)
{
    Console.WriteLine(c);
}
```

Query method syntax:

```
// Specify the data source
List<Customer> customers = Customer.GetCustomers();

// Define the query using method syntax
var queryMethod = customers.Where(c => c.City == "London")
                           .OrderBy(c => c.Balance)
                           .Select(c => c.CustomerName);

// Execute the query
foreach (var c in queryMethod)
{
    Console.WriteLine(c);
}
```

6

6

# LINQ PROJECTIONS

You can explicitly specify the query type if you project (select) a whole class e.g., Customer.

```
// Return type is IEnumerable<Customer>
IEnumerable<Customer> queryA = from c in customers
                               where c.City == "London"
                               orderby c.Balance
                               select c; // Customer is returned
```

If you project only some of the object's properties, the compiler will generate a new anonymous type.

Use the **var** keyword:

```
// Return type is IEnumerable<<anonymous type: string customerName, string City>>
IEnumerable<Customer> queryB = from c in customers
                               where c.City == "London"
                               orderby c.Balance
                               select new { c.CustomerName, c.City }; // CustomerName and City are returned

// use var
var queryC = from c in customers
                               where c.City == "London"
                               orderby c.Balance
                               select new { c.CustomerName, c.City }; // CustomerName and City are returned
```

7

7

# DEFERRED EXECUTION

A LINQ query is not executed until you iterate over the query variable in a foreach statement.

This allows the query to retrieve different data each time it is executed:

```
string[] names = { "Tommy", "Fiona", "Rashid", "Bobby" };

var query = from s in names
            where s.Length == 5
            select s;

foreach (string s in query)
{
    Console.Write(s + " ");
}
// Output: Tommy Fiona Bobby


names[0] = "Susie";

foreach (string s in query)
{
    Console.Write(s + " ");
}
// Output: Susie Fiona Bobby
```

8

8

# FORCING IMMEDIATE EXECUTION: AGGREGATE FUNCTIONS

- Queries that perform aggregation functions over a range of source elements must first iterate over those elements, therefore they execute **without** an explicit foreach statement
- These types of queries return a single value not an IEnumerable collection

```
int[] scores = { 97, 92, 81, 60, 40, 54, 80, 75 };

var failingScores =
    from score in scores
    where score < 80
    select score;

int countFailingScores = failingScores.Count();
Console.WriteLine("Number of failing scores is " + countFailingScores);

double avgFailingScores = failingScores.Average();
Console.WriteLine("Average of failing scores is " + avgFailingScores);
```

9

9

# FORCING IMMEDIATE EXECUTION: TOLIST OR TOARRAY

- To force execution of any LINQ query and cache its results, you can call the **ToList** or **ToArray** methods:

```
string[] names = { "Tommy", "Fiona", "Rashid", "Bobby" };

var query = (from s in names
             where s.Length == 5
             select s).ToList();//force execution

foreach (string s in query)
{
    Console.WriteLine(s);
}
// Output: Tommy Fiona Bobby

names[0] = "Susie";

foreach (string s in query)
{
    Console.WriteLine(s);
}
// Output: Tommy Fiona Bobby
```

10

10

# JOINS

The **join** clause is used to match elements in one collection with elements in another collection.

Find persons who have names that match the 'names' list:

```
string[] names = { "Vinita", "Pete" };

var people = new List<Person> {
    new Person("Pete",20),
    new Person("Rafael",50),
    new Person("Vinita",32),
    new Person("Pete",37),
    new Person("Tom",40),
};


var matchingPeople = from n in names
                     join p in people
                     on n equals p.Name
                     select p;

foreach (var p in matchingPeople)
{
    Console.WriteLine("{0}, {1}", p.Name, p.Age);
}
```

```
Vinita, 32
Pete, 20
Pete, 37
```

11

11

# GROUP JOIN

The **group join** method is useful for producing hierarchical data structures.

It pairs each element from the first collection with a set of correlated elements from the second collection

The query expression **join … into** translates to an invocation of **GroupJoin**

```
var groupJoined = from n in names
                  join p in people
                  on n equals p.Name into matchingNames
                  select new { Name = n, Persons = matchingNames };

foreach (var group in groupJoined)
{
    Console.WriteLine(group.Name);
    foreach (Person p in group.Persons)
    {
        Console.WriteLine("..." + p.Age);
    }
}
```

```
Vinita
...32
Pete
...20
...37
```

12

12

# AGGREGATIONS

- Aggregations execute without an explicit foreach statement
- The **baseQuery** uses lambda expressions to specify which property of Customer is to be aggregated
- **Qry2** projects the Balance property which is a decimal, so no lambdas are required

```
var baseQuery = from c in customers
                where c.City == "London"
                orderby c.Balance
                select c;// Customer

decimal avg = baseQuery.Average(c => c.Balance);
decimal max = baseQuery.Max(c => c.Balance);
decimal total = baseQuery.Sum(c => c.Balance);


IEnumerable<decimal> qry2 = from c in customers
                            where c.City == "London"
                            select c.Balance;// Decimal
avg = qry2.Average();
max = qry2.Max();
total = qry2.Sum();
```

13

13

# LET CLAUSE

- The **let** clause enables you to store the result of a sub-expression in order to use it in subsequent clauses

```
// Specify the data source
string[] strings =
{
    "Been there, done that.",
    "All Greek to me.",
    "A piece of cake."
};

// Split the sentence into an array of words
// and select those whose first letter is a vowel.
var query =
    from sentence in strings
    let words = sentence.Split(' ')
    from word in words
    let w = word.ToLower()
    where w[0] == 'a' || w[0] == 'e'
        || w[0] == 'i' || w[0] == 'o'
        || w[0] == 'u'
    select word;

// Execute the query.
foreach (var v in query)
{
    Console.WriteLine("\"{0}\" starts with a vowel", v);
}

/* Output:
"All" starts with a vowel
"A" starts with a vowel
"of" starts with a vowel
*/
```

14

14

# OFTYPE

The **OfType** method filters the elements of an IEnumerable based on a specified type:

```
public class Mammal
{
    0 references
    public string Name { get; set; }
}
```

```
public class Dog : Mammal
{
}
```

```
public class Cat : Mammal
{
}
```

```
List<Mammal> mammals = new() {
    new Dog() {Name="Rover"},
    new Cat() {Name="Tiddles"},
    new Dog() {Name="Snowy"}
};
var dogs = mammals.OfType<Dog>();

foreach (var dog in dogs)
{
    Console.WriteLine(dog.Name);
}

/* Output:
Rover
Snowy
*/
```

15

15

# LINQ QUERY KEYWORDS & METHODS

```
from
join .. on
 equals
where
group .. by
into
let
orderby
 descending
select
```

```
Join
Where
GroupBy
OrderBy
OrderByDescending
ThenBy
ThenByDescending
Select
```

```
Concat
Distinct
Except
Intersect
Union
GroupJoin
Reverse
SelectMany
SequenceEqual
Union
```

```
Contains
DefaultIfEmpty
ElementAt
Single
```

```
AsEnumerable
Cast
OfType
```

```
Aggregate
Average
Count
LongCount
Max
Min
Sum
```

```
First
Last
Skip
SkipWhile
Take
TakeWhile
```

```
All
Any
```

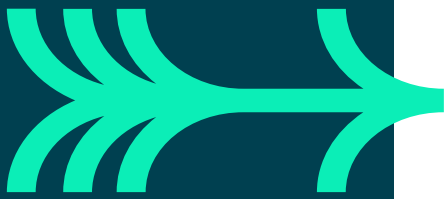
```
ToArray
ToDictionary
ToList
ToLookup
```

16

16

# SUMMARY

- Language-Integrated Query (LINQ)
- LINQ syntax
- LINQ projections
- Deferred execution
- Forcing immediate execution
- Joins
- Group join
- Aggregations
- The let clause
- The OfType method
- LINQ expression syntax and keyword reference



17

## Activity: Exercise 12

- Convert non-LINQ code into the LINQ equivalent

18

18