LINQ

C# Programming

In this chapter we will learn ...

- How to use Language Integrated Query

LINQ

What is LINQ

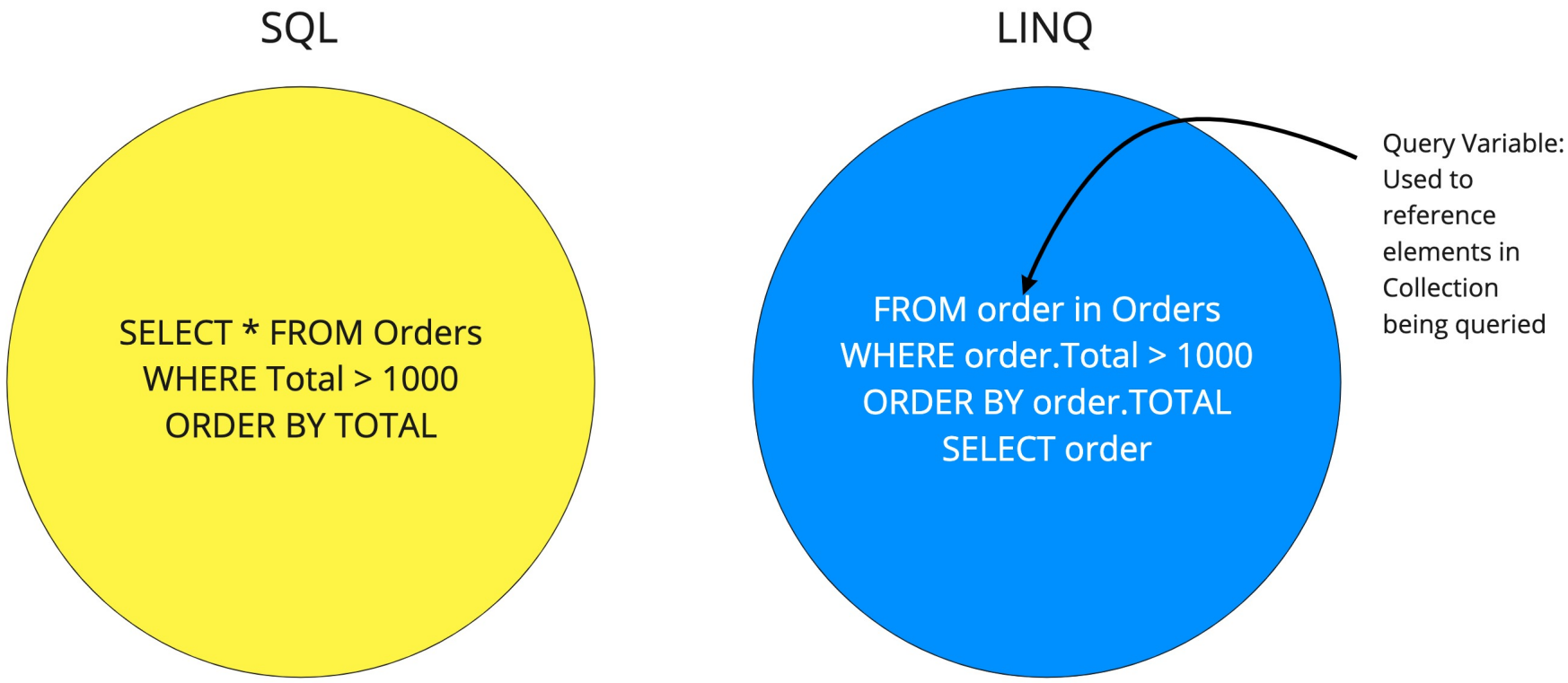
Language Integrated Query

- Release in .NET 3.5

Declarative querying of .NET collection/enumerable types

- SQL Like Syntax (read backwards!)
- If you can *foreach* through it you can LINQ query it!
- Specify your intent and LINQ does the rest

Comparing Query Languages



Using LINQ

To use LINQ you need to include using System.Linq

```
using System.Linq;
```

Now you can use a LINQ extension method against an IEnumerable type

LINQ Syntax

- Define Data Source and a enumeration variable to reference in query

```
from <variable> in <enumberable object>
```

- Optional - Define one or more *where* predicates and/or *order by* statements

```
where <predicate>, ...
order by <variable.property>, ...
```

- Define query return element/variable/anonymous type

```
select <variable>
```

LINQ Query Example

Simple LINQ Query

- Assuming Data

```
var people = new Person[] {new Person("Fred", 20, 30000), new Person("Mina", 30, 50000)};
```

- A simple query returning all elements would be

```
var results =   from p in people
                select p;

foreach(var person in results){
    Console.WriteLine($"{person.Name} is {person.Age}");
}
```

Filtering using where

- Assuming Data

```
var people = new Person[] {new Person("Fred", 20, 30000), new Person("Mina", 30, 50000)};
```

- A query returning people whose age > 20 would be

```
var results =   from p in people
                where p.age > 20
                select p;

foreach(var person in results){
    Console.WriteLine($"{person.Name} is {person.Age}");
}
```

Ordering using OrderBy

OrderBy and OrderByDescending

- Returns an *IOrderedEnumerable<source type>*
- Allows you to order based on a sort key you define derived from each element in turn

```
var people = new Person[] {new Person("Fred", 20, 30000), new Person("Mina", 30, 50000)};
// Order by the Salary
IOrderedEnumerable<Person> results = people.OrderBy(p => p.Salary);
```

Group By

Groups together Elements in a collection based on a property

Syntax

group <element variable> by <property> into <group variable>

- Identify unique groups for chosen property and create group object for each
- Assign each Group its Key value from the chosen property
- Collect all elements with the same Key into their respective Group

Group By - Example

```
var employees = new List<Employee> {
            new Employee { FirstName="Fred", LastName="Bloggs", Age=21 },
            new Employee { FirstName="Suzy", LastName="Simm", Age=27 },
            new Employee { FirstName="Toby", LastName="Thomas", Age=27 },
            new Employee { FirstName="Greg", LastName="Gregory", Age=21 }
        };

var groups =    from employee in employees
                group employee by  employee.Age into newGroup
                select newGroup;

foreach(var group in groups) {
    Console.WriteLine(group.Key); // Prints distinct age groups 21 and 27
}
```

Getting Started with LINQ - Group Functions

Group Functions allow group calculations across a simple collection:

- Count()
- Sum()
- Max(), Min()
- Average()

```
> var numbers = new int[] {5, 10 , 20, 35, 40};
> numbers.Sum();
110
> numbers.Average();
22
```

Group Functions - Complex Types

How do we Sum a List of Person objects == More complex

- By Salary ?
- By Age ?

```
// Assuming a Person class with Name, Age and Salary properties
> var people = new Person[] {new Person("Fred", 20, 30000), new Person("Mina", 30, 50000)};
```

- Use a Lambda expression as a property picker
    - Picks which property to Sum for each element in array

```
> people.Average(p => p.Age);
25
```

Using var with LINQ

It can get difficult to predict what a LINQ query will return

Instead of

```
IOrderedEnumerable<Person> results = people.OrderBy(p => p.Salary);
```

We would more usually write

```
var results = people.OrderBy(p => p.Salary);
```

- **var** allows type inference - Let the compiler do the work!

Lambda Syntax - Filtering using Where

```
var people = new Person[] {new Person("Fred", 20, 30000), new Person("Mina", 30, 50000)};
var results = people.Where(p => p.Age >= 18);
```

- Where method allows collection filtering based on a predicate
  - the result is IEnumerable<source type>
  - Lambda expression is used as a predicate method to filter
  - Lambda invoked for each element in collection
  - Element is included in result if Lambda returns **true**

LINQ Methods can be combined

Most LINQ calls return the result of the action ... So ...

- LINQ calls can be chained together

```
var results = people.Where(p.Age >= 18).OrderBy(p => p.Salary)
```

You can span multiple lines to aid readability

```
var results = people.Where(p.Age >= 18)
                    .OrderBy(p => p.Salary)
```

Controlling the returned result - Select

Usually LINQ returns an IEnumerable of the type you are querying

- Querying a List will return IEnumberable<int>

Select allows you to choose what is returned for each result row

```
var people = new Person[] {new Person("Fred", 20, 30000), new Person("Mina", 30, 50000)};
IEnumerable<string> names = people.Where(p => p.Age>=18)
                                  .Select(p=>p.Name);
```

Returning other results from a Select

Anonymous Types can be generated without a class or a struct as a template

```
// Creates a brand new anonymous type
var person = new {name="Fred", age =21};
```

We can use anonymous types in Select statements

```
var people = new Person[] {new Person("Fred", 20, 30000), new Person("Mina", 30, 50000)};
// Must use var!
var names = people.Where(p => p.Age>=18 )
                                  .Select(p=> new {p.Name, p.Age});
```

Returning other types from a Select

Select can be used to map the source element to a new type

- Select returns a new Employee for each Person

```
var people = new Person[] {new Person("Fred", 20, 30000), new Person("Mina", 30, 50000)};
// Must use var!
var names = people.Where(p => p.Salary>0)
                  .Select(p=> new Employee{FullName=p.Name, Salary=p.Salary});
```

How does LINQ work

LINQ is built on top of IEnumerable interface

- IEnumerable is implemented by almost all collections and arrays
- Implemented as a set of Extension methods to IEnumerable
- Any type implementing IEnumerable will have extension methods available

Extension methods include

- Select, Where, OrderBy
- Sum, Max, Min plus many more

What is IEnumerable

IEnumerable is an interface that

- Facilitates iteration through a collection/stream of items

It defines a single function

- GetEnumerator() - returns IEnumerator

IEnumerator defines methods to allow scrolling through collections

- MoveNext()
- Current
- Reset

Writing a Simple Enumerator

- This enumerator is designed to step through an array of Cars

```
class CarEnumerator : IEnumerator<Car> {
    Car[] cars; int position = -1;
    public CarEnumerator(Car[] cars) {
        this.cars = cars;
    }
    public object Current => cars[position];
    Car IEnumerator<Car>.Current => (Car)Current;
    public void Dispose() { }
    public bool MoveNext() {
        return ++position >= 0 && position < cars.Length;
    }
    public void Reset(){
        position=-1;
    }
}
```

Creating an IEnumerable Type

- All the type has to do is
  - Implement IEnumerable and GetEnumerator()
  - return a type implementing IEnumerator

```
class Garage : IEnumerable {
    public IEnumerator GetEnumerator() {
        return new CarEnumerator(   new Car[] {
                                    new Car { Make = "Ford" },
                                    new Car { Make ="Mini" } ,
                                    new Car { Make ="Porsche" } ,
                                    new Car { Make ="BMW" }
                            });
    }
}
```

What's the benefit of implementing IEnumerable?

Simple - You can user foreach on IEnumerable implementing types

foreach will

- Call GetEnumerator()
- Use the CarEnumerator to iterate through list of cars

```
var garage = new Garage();
foreach(Car car in garage) {
    Console.WriteLine(car.Make);
}
```

A even simpler enumerator

Call yield return from GetEnumerator()->IEnumerator

- Enables a calculated/generated enumerator to be created

```
class Garage : IEnumerable {
    public IEnumerator GetEnumerator() {
        yield return new Car { Make = "Ford" };
        yield return new Car { Make ="Mini" };
        yield return new Car { Make ="Porsche" };
        yield return new Car { Make ="BMW" };
    }
}
```

In this chapter we learned ...

- How to use Language Integrated Query