

# ТЕХНОЛОГІЯ LINQ ТА ОРГАНІЗАЦІЯ ДОСТУПУ ДО ДАНИХ

Лекція 08  
Об'єктно-орієнтоване програмування

# План лекції

---

- Огляд технології LINQ.
- Фільтрування, впорядкування та проєктування даних.
- З'єднання, групування та агрегування даних.

## LINQ Fundamentals with C# 6.0

---



**Scott Allen**

OdeToCode.com - @OdeToCode





# ОГЛЯД ТЕХНОЛОГІЇ LINQ

Питання 8.1.

# Робота з різними джерелами даних

---

- Класичні підходи до обробки даних з різних джерел у C#:
  - Якщо дані знаходяться в оперативній пам'яті як набір об'єктів, застосовувались API узагальнених колекцій для доступу до цих даних.
  - Якщо дані знаходяться в БД SQL Server, потрібно було застосовувати ADO.NET та писати SQL-команди для взаємодії з базою.
  - Для XML-даних маємо повний набір API для формування запитів до XML-документів.
- Дані підходи відрізняються та мають різну функціональну підтримку.



# Робота з різними джерелами даних

---



- Запити, вбудовані в мову C# (Language Integrated Query) пропонують єдиний API для доступу до різних джерел даних.



# Порівняємо реалізацію без LINQ та з її використанням

```
class Program
{
    static void Main(string[] args)
    {
        string path = @"C:\windows";
        ShowLargeFilesWithoutLinq(path);
        Console.WriteLine("\n***\n");
        ShowLargeFilesWithLinq(path);
    }

    private static void ShowLargeFilesWithLinq(string path)
    {
        var query = from file in new DirectoryInfo(path).GetFiles()
                    orderby file.Length descending
                    select file;

        foreach (var file in query.Take(5))
        {
            Console.WriteLine($"{file.Name,-20} : {file.Length,11:N0}");
        }
    }

    private static void ShowLargeFilesWithoutLinq(string path)
    {
        DirectoryInfo directory = new DirectoryInfo(path);
        FileInfo[] files = directory.GetFiles();
        Array.Sort(files, new FileInfoComparer());

        for (int i = 0; i < 5; i++)
        {
            FileInfo file = files[i];
            Console.WriteLine($"{file.Name, -20} : {file.Length, 11 :N0}");
        }
    }
}
```

24.11.2020

@Марченко С.В., ЧДБК, 2020

- Задача: вивести перелік 5 найбільших файлів у каталозі, відсортувавши їх за розміром

```
public class FileInfoComparer : IComparer<FileInfo>
{
    public int Compare(FileInfo x, FileInfo y)
    {
        return y.Length.CompareTo(x.Length);
    }
}
```

```
C:\Users\puasson\Source\Repos\LinqSamples\Intro...
explorer.exe      : 4 485 216
HelpPane.exe     : 1 075 200
pyw.exe          : 913 992
py.exe           : 913 480
regedit.exe      : 369 664

***

explorer.exe      : 4 485 216
HelpPane.exe     : 1 075 200
pyw.exe          : 913 992
py.exe           : 913 480
regedit.exe      : 369 664
```

# Еволюція ідей

---

- Базова ідея: зробити код зручним, читабельним та зрозумілим.
  - Щось на зразок такого: абстрактного (незалежного від джерела) контейнера для даних з використанням поширених операторів запитів

```
Sequence<Employee> scotts =  
    employees.Where(Name == "Scott");
```

- Перша подоба LINQ-запитів з'явилась у C# 2.0:

```
IEnumerable<Employee> scotts =  
    EnumerableExtensions.Where(employees,  
        delegate(Employee e)  
        {  
            return e.Name == "Scott";  
        }));
```

- Сучасний LINQ:

```
var scotts =  
    from e in employees  
    where e.Name == "Scott"  
    select e;
```

# Методи розширення (extension methods)

---

- Методи розширення дозволяють визначати статичні методи, які будуть членами будь-якого типу (класу, інтерфейсу, структури та ін., навіть запечатаного типу).
  - Перший параметр таких методів позначається модифікатором `this`:

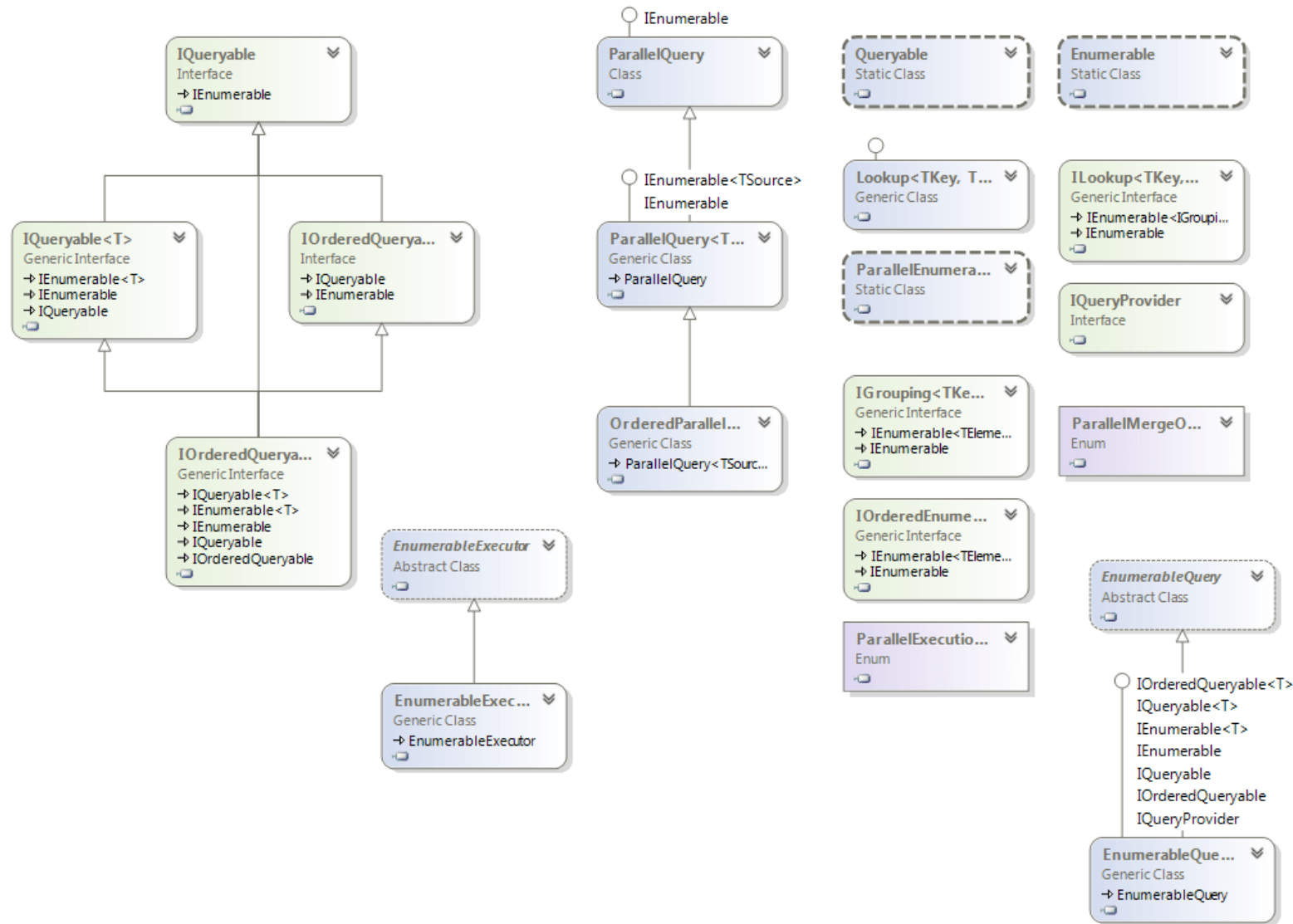
```
public static class StringExtensions
{
    static public double ToDouble(this string data)
    {
        double result = double.Parse(data);
        return result;
    }
}
```

- Тип цього параметру – це тип, який буде розширятись.
  - Тут даний метод буде поводити себе як метод об'єкта типу `string`.
  - Методи розширення не заміщують вже існуючі реалізації.
- Робота з методом розширення:
  - LINQ-оператори зазвичай визначаються як методи розширення.

```
string text = "43.35";
double data = text.ToDouble();
```



# Методи розширення LINQ

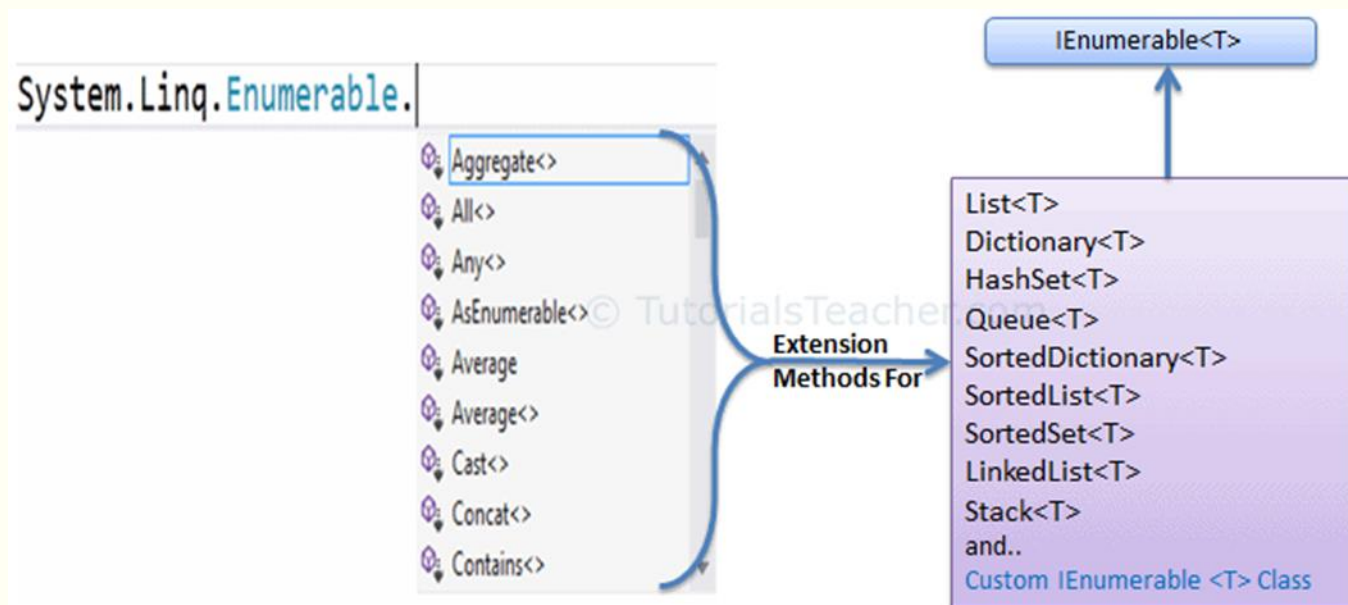


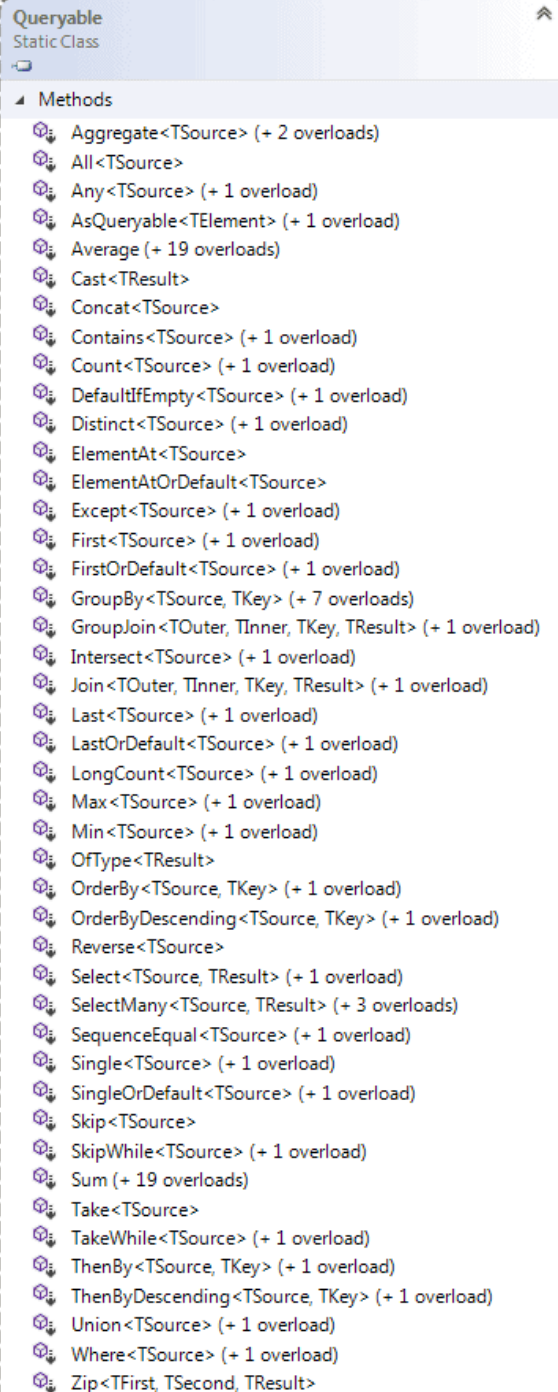
- Можемо записувати LINQ-запити для класів, що реалізують інтерфейс `IEnumerable<T>` або `IQueryable<T>`.
- Класи `Enumerable` та `Queryable` – це два статичних класи, які містять методи розширення для запису LINQ-запитів.

# Клас Enumerable

- Містить методи розширення для класів, які реалізують інтерфейс `IEnumerable<T>`, наприклад, всі вбудовані класи колекцій.

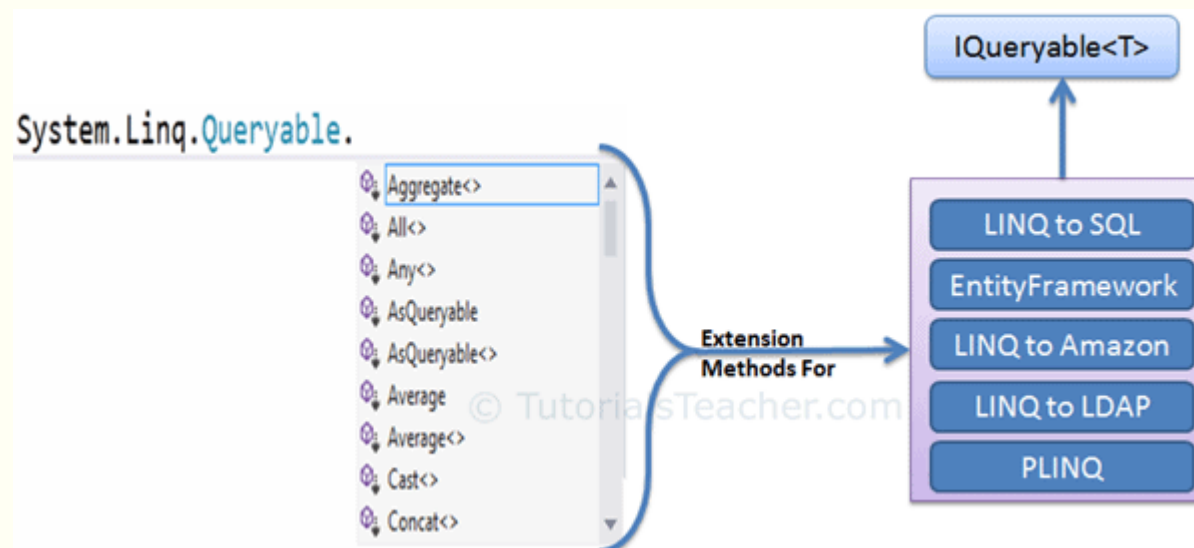
- Тому можемо застосовувати LINQ-запити для отримання (retrieve) даних з них.





# Клас Queryable

- Містить методи розширення для класів, які реалізують інтерфейс `IQueryable<T>`.
  - Інтерфейс `IQueryable<T>` застосовується для забезпечення можливостей формування запитів (querying) до спеціалізованих джерел даних, у яких тип даних відомий.
  - Наприклад, Entity Framework API реалізує інтерфейс `IQueryable<T>`, щоб підтримувати LINQ-запити до баз даних, на зразок MS SQL Server.
  - Також існують API для доступу до сторонніх даних; наприклад, LINQ to Amazon надає можливість використовувати LINQ з веб-сервісами [Amazon Web Services](#) для пошуку книжок та інших товарів.
  - Це досягається реалізацією інтерфейсу `IQueryable` для Amazon.



# Синтаксис запитів (query syntax)

---

- Синтаксис методів  
(Method syntax approach)

```
var query = developers.Where(e => e.Name.Length == 5)
                        .OrderByDescending(e => e.Name)
                        .Select(e => e);
```

- Синтаксис запитів:
  - Завжди починається з from
  - Закінчується словами select або group
  - Не всі LINQ-оператори доступні в синтаксисі запитів

```
var query2 = from developer in developers
              where developer.Name.Length == 5
              orderby developer.Name descending
              select developer;
```

# LINQ-запити (проєкт Queries)

---

```
public class Movie
{
    public string Title { get; set; }
    public float Rating { get; set; }

    int _year;
    public int Year {
        get
        {
            Console.WriteLine($"Returning {_year} for {Title}");
            return _year;
        }
        set
        {
            _year = value;
        }
    }
}
```

- Розглядаємо модель даних для представлення інформації про кінофільми.
  - Передбачаємо клас Movie, який описуватиме фільм за назвою, рейтингом та роком випуску.
  - В головному класі формуємо список кінофільмів.
  - Вони ніяк не впорядковані.

```
class Program
{
    static void Main(string[] args)
    {
        var movies = new List<Movie>
        {
            new Movie { Title = "The Dark Knight", Rating = 8.9f, Year = 2008 },
            new Movie { Title = "The King's Speech", Rating = 8.0f, Year = 2010 },
            new Movie { Title = "Casablanca", Rating = 8.5f, Year = 1942 },
            new Movie { Title = "Star Wars V", Rating = 8.7f, Year = 1980 }
        };
    }
}
```

@Марченко С.В., ЧДБК, 2020

# LINQ-запити (проект Queries)

---

```
public static class MyLinq
{
    public static IEnumerable<T> Filter<T>(this IEnumerable<T> source,
                                           Func<T, bool> predicate)
    {
        foreach (var item in source)
        {
            if (predicate(item))
            {
                yield return item;
            }
        }
    }
}
```

- Нехай потрібно вивести всі фільми, випущені після 2000 року.
  - Спробуємо створити власний метод розширення (замість Where()), який відбиратиме потрібні фільми – Filter().
  - Створимо статичний відкритий клас MyLinq, який міститиме в собі статичний узагальнений метод Filter<T>.
  - Func-делегат реалізує предикат, який вказує на відповідність умові елементів колекції source.
  - Оператор yield return здійснює відкладене виконання методу розширення.
  - Така реалізація функціонально подібна до методу розширення Where().

# Чи забезпечує метод розширення відкладене виконання?

```
class Program
{
    static void Main(string[] args)
    {
        var movies = new List<Movie>
        {
            new Movie { Title = "The Dark Knight",    Rating = 8.9f, Year = 2008 },
            new Movie { Title = "The King's Speech",  Rating = 8.0f, Year = 2010 },
            new Movie { Title = "Casablanca",         Rating = 8.5f, Year = 1942 },
            new Movie { Title = "Star Wars V",        Rating = 8.7f, Year = 1980 }
        };

        var query = movies.Filter(m => m.Year > 2000);

        var enumerator = query.GetEnumerator();
        while (enumerator.MoveNext())
        {
            Console.WriteLine(enumerator.Current.Title);
        }
    }
}
```

- Для демонстрації цього можна явно застосувати енумератор та за допомогою налагоджувальника (F10/F11) відстежити роботу методу.
- Також це явно зазначається у документації відповідного методу.

## Remarks

This method is implemented by using deferred execution. The immediate return value is an object that stores all the information that is required to perform the action. The query represented by this method is not executed until the object is enumerated either by calling its `GetEnumerator` method directly or by using `foreach` in Visual C# or `For Each` in Visual Basic.

In query expression syntax, a `where` (Visual C#) or `Where` (Visual Basic) clause translates to an invocation of `Where<TSource>(IEnumerable<TSource>, Func<TSource, Boolean>)`.



# Особливості відкладеного виконання

---

```
class Program
{
    static void Main(string[] args)
    {
        var movies = new List<Movie>
        {
            new Movie { Title = "The Dark Knight", Rating = 8.9f, Year = 2008 },
            new Movie { Title = "The King's Speech", Rating = 8.0f, Year = 2010 },
            new Movie { Title = "Casablanca", Rating = 8.5f, Year = 1942 },
            new Movie { Title = "Star Wars V", Rating = 8.7f, Year = 1980 }
        };

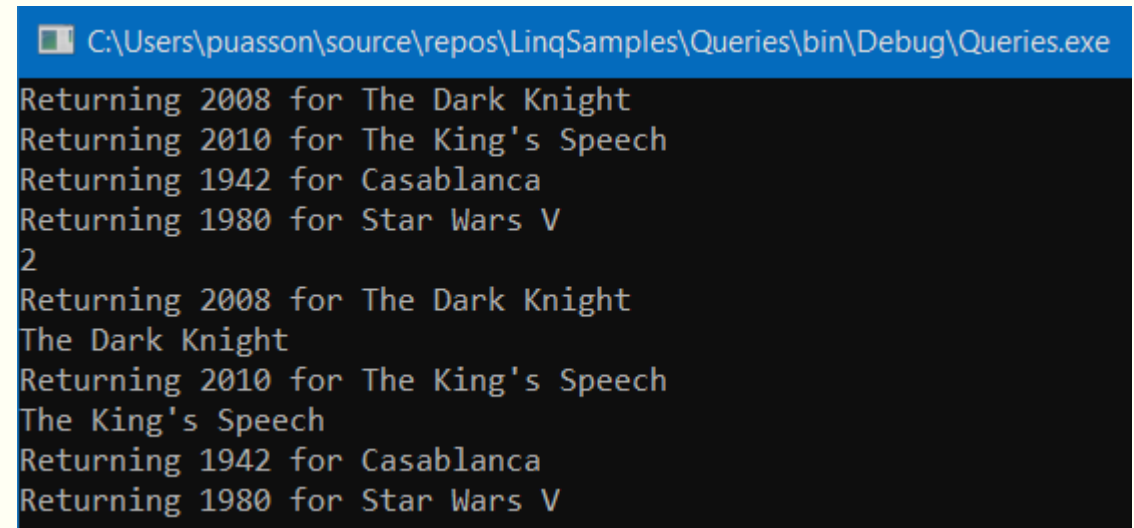
        var query = movies.Filter(m => m.Year > 2000);

        Console.WriteLine(query.Count());

        var enumerator = query.GetEnumerator();
        while (enumerator.MoveNext())
        {
            Console.WriteLine(enumerator.Current.Title);
        }

        Console.ReadKey();
    }
}
```

- Нехай крім переліку фільмів потрібно вивести також, скільки таких фільмів буде.
- Зауважте, що спочатку переглядалось всі 4 фільми для виводу результату – 2, а потім всі фільми заново переглядаються.
- Метод Count() не пропонує відкладеного виконання



```
C:\Users\puasson\source\repos\LinqSamples\Queries\bin\Debug\Queries.exe
Returning 2008 for The Dark Knight
Returning 2010 for The King's Speech
Returning 1942 for Casablanca
Returning 1980 for Star Wars V
2
Returning 2008 for The Dark Knight
The Dark Knight
Returning 2010 for The King's Speech
The King's Speech
Returning 1942 for Casablanca
Returning 1980 for Star Wars V
```



# Матеріалізація результату запиту в конкретну структуру даних

---

```
class Program
{
    static void Main(string[] args)
    {
        var movies = new List<Movie>
        {
            new Movie { Title = "The Dark Knight", Rating = 8.9f, Year = 2008 },
            new Movie { Title = "The King's Speech", Rating = 8.0f, Year = 2010 },
            new Movie { Title = "Casablanca", Rating = 8.5f, Year = 1942 },
            new Movie { Title = "Star Wars V", Rating = 8.7f, Year = 1980 }
        };

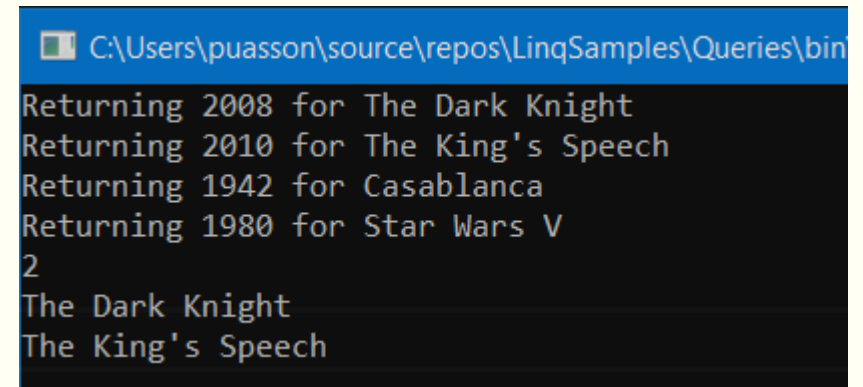
        var query = movies.Filter(m => m.Year > 2000).ToList();

        Console.WriteLine(query.Count());

        var enumerator = query.GetEnumerator();
        while (enumerator.MoveNext())
        {
            Console.WriteLine(enumerator.Current.Title);
        }

        Console.ReadKey();
    }
}
```

- Маючи масив чи список, ми сформуємо єдину версію набору даних:



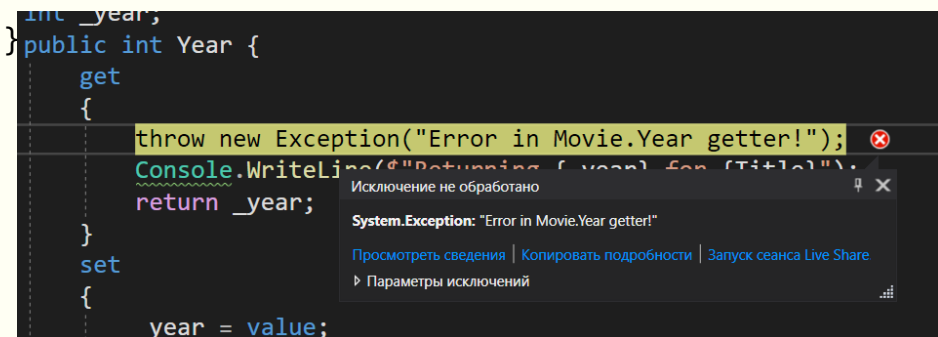
```
C:\Users\puasson\source\repos\LinqSamples\Queries\bin
Returning 2008 for The Dark Knight
Returning 2010 for The King's Speech
Returning 1942 for Casablanca
Returning 1980 for Star Wars V
2
The Dark Knight
The King's Speech
```

- Зазвичай, якщо метод розширення повертає абстрактний тип, на зразок `Ienumerable<T>`, тоді цей метод виконується відкладено.
- Інакше – ні. Наприклад, `ToList()` повертає список `List<T>`, а `Count()` – ціле число.

# Винятки та відкладене виконання

```
public class Movie
{
    public string Title { get; set; }
    public float Rating { get; set; }

    int _year;
    public int Year {
        get
        {
            throw new Exception("Error in Movie.Year getter!");
            Console.WriteLine($"Returning {_year} for {Title}");
            return _year;
        }
        set
        {
            _year = value;
        }
    }
}
```



- Моделюємо виняток в геттері для року.
  - Через відкладене виконання (звернення до геттера тільки при виклику Count()) виняток не потрапляє в блок try-catch та викидається додатком.
  - Проте при застосуванні ToList() спрацює обробник.

```
class Program
{
    static void Main(string[] args)
    {
        ...
        var query = Enumerable.Empty<Movie>();

        try
        {
            query = movies.Where(m => m.Year > 2000);
        }
        catch(Exception ex)
        {
            Console.WriteLine(ex.Message);
        }

        Console.WriteLine(query.Count());

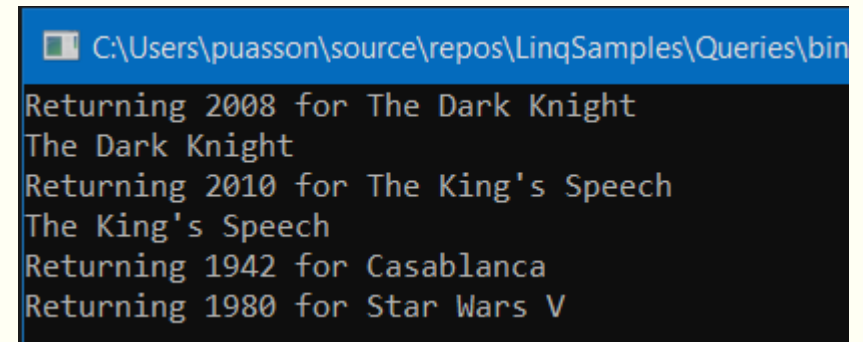
        ...
    }
}
```

# Потокові (streaming) оператори LINQ

---

- Where() – приклад потокового оператора.
  - Потокові оператори не зчитують все джерело даних перед генеруванням (yield) елементів.
  - У момент виконання потоковий оператор виконує операцію до кожного елемента джерела даних під час зчитування цього елемента та повертає (yields) елемент, якщо той доречний.
  - Потоковий оператор продовжує зчитувати елементи джерела даних, поки результуючий елемент може формуватись. Тобто більше одного елемента даних може зчитуватись, щоб сформувати результуючий елемент.
- Закоментуємо в прикладі виклик query.Count().
  - Ми проходимо один фільм за раз, перевіряємо відповідність критерію, а потім знову викликаємо Where() для наступного фільму.

```
var query = movies.Where(m => m.Year > 2000);  
//Console.WriteLine(query.Count());  
  
var enumerator = query.GetEnumerator();  
while (enumerator.MoveNext())  
{  
    Console.WriteLine(enumerator.Current.Title);  
}
```



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Users\puasson\source\repos\LinqSamples\Queries\bin'. The console output displays the results of a LINQ query filtering movies by year. It shows the year of the first movie that meets the criteria, followed by the movie title, and then continues for subsequent movies.

```
C:\Users\puasson\source\repos\LinqSamples\Queries\bin  
Returning 2008 for The Dark Knight  
The Dark Knight  
Returning 2010 for The King's Speech  
The King's Speech  
Returning 1942 for Casablanca  
Returning 1980 for Star Wars V
```

# Непотокові (non-streaming) оператори LINQ

---

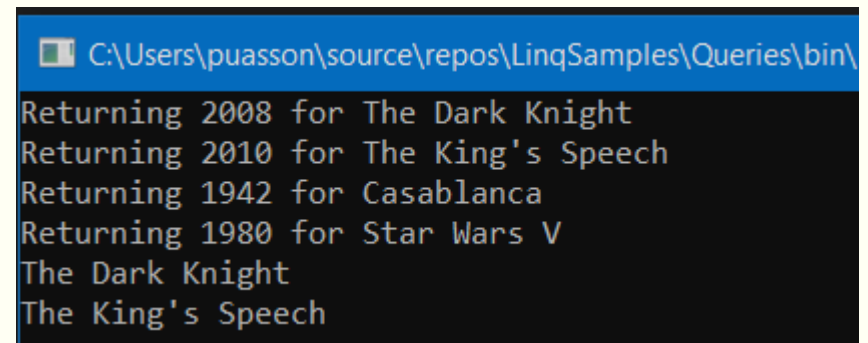
- **OrderByDescending()** – приклад непотокового оператора.
  - Непотокові оператори повинні зчитувати все джерело даних перед можливістю згенерувати результуючий елемент.
  - У цю категорію потрапляють операції сортування та групування.
  - У момент виконання непотокові оператори запиту зчитують все джерело даних, зберігають його в структуру даних, виконують операцію та генерують результуючі елементи.
- Застосуємо **OrderByDescending()** – вивід дещо зміниться.
  - Ми переглянемо ВСІ фільми до того, як запит почне формувати результат.
  - Робота подібна до негайного виконання (як **ToList()**), проте насправді є відкладеним виконанням.
  - Якщо закоментувати цикл **while**, нічого не буде виведено.
  - Непотокові оператори ефективно працюють після фільтрування даних (зменшення набору).

```
var query = movies.Where(m => m.Year > 2000)
                    .OrderByDescending(m => m.Rating);
```

```
var enumerator = query.GetEnumerator();
while (enumerator.MoveNext())
{
    Console.WriteLine(enumerator.Current.Title);
}
```

24.11.2020

@Марченко С.В., ЧДБК, 2020



```
C:\Users\puasson\source\repos\LinqSamples\Queries\bin\
Returning 2008 for The Dark Knight
Returning 2010 for The King's Speech
Returning 1942 for Casablanca
Returning 1980 for Star Wars V
The Dark Knight
The King's Speech
```

# Класифікація стандартних LINQ-операторів

Стандартний оператор	Вихідний тип	Негайне виконання	Відкладене потокове виконання	Відкладене непотокове виконання
<u>Aggregate</u>	TSource	X		
<u>All</u>	Boolean	X		
<u>Any</u>	Boolean	X		
<u>AsEnumerable</u>	IEnumerable<T>		X	
<u>Average</u>	Single numeric value	X		
<u>Cast</u>	IEnumerable<T>		X	
<u>Concat</u>	IEnumerable<T>		X	
<u>Contains</u>	Boolean	X		
<u>Count</u>	Int32	X		
<u>DefaultIfEmpty</u>	IEnumerable<T>		X	
<u>Distinct</u>	IEnumerable<T>		X	
<u>ElementAt</u>	TSource	X		
<u>ElementAtOrDefault</u>	TSource	X		
<u>Empty</u>	IEnumerable<T>	X		
<u>Except</u>	IEnumerable<T>		X	X
<u>First</u>	TSource	X		
<u>FirstOrDefault</u>	TSource	X		
<u>GroupBy</u>	IEnumerable<T>			X
<u>GroupJoin</u>	IEnumerable<T>		X	X
<u>Intersect</u>	IEnumerable<T>		X	X
<u>Join</u>	IEnumerable<T>		X	X
<u>Last</u>	TSource	X		
<u>LastOrDefault</u>	TSource	X		
<u>LongCount</u>	Int64	X		

# Ітерування по колекціях

Стандартний оператор	Вихідний тип	Негайне виконання	Відкладене потокове виконання	Відкладене непотокове виконання
<u>Max</u>	Числове значення, TSource чи TResult	X		
<u>Min</u>	Числове значення, TSource чи TResult	X		
<u>OfType</u>	<u>IEnumerable&lt;T&gt;</u>		X	
<u>OrderBy</u>	<u>IOrderedEnumerable&lt;TElement&gt;</u>			X
<u>OrderByDescending</u>	<u>IOrderedEnumerable&lt;TElement&gt;</u>			X
<u>Range</u>	<u>IEnumerable&lt;T&gt;</u>		X	
<u>Repeat</u>	<u>IEnumerable&lt;T&gt;</u>		X	
<u>Reverse</u>	<u>IEnumerable&lt;T&gt;</u>			X
<u>Select</u>	<u>IEnumerable&lt;T&gt;</u>		X	
<u>SelectMany</u>	<u>IEnumerable&lt;T&gt;</u>		X	
<u>SequenceEqual</u>	<u>Boolean</u>	X		
<u>Single</u>	TSource	X		
<u>SingleOrDefault</u>	TSource	X		
<u>Skip</u>	<u>IEnumerable&lt;T&gt;</u>		X	
<u>SkipWhile</u>	<u>IEnumerable&lt;T&gt;</u>		X	
<u>Sum</u>	Single numeric value	X		
<u>Take</u>	<u>IEnumerable&lt;T&gt;</u>		X	
<u>TakeWhile</u>	<u>IEnumerable&lt;T&gt;</u>		X	
<u>ThenBy</u>	<u>IOrderedEnumerable&lt;TElement&gt;</u>			X
<u>ThenByDescending</u>	<u>IOrderedEnumerable&lt;TElement&gt;</u>			X
<u>ToArray</u>	Масив TSource	X		
<u>ToDictionary</u>	<u>Dictionary&lt;TKey,TValue&gt;</u>	X		
<u>ToList</u>	<u>IList&lt;T&gt;</u>	X		
<u>ToLookup</u>	<u>ILookup&lt;TKey,TElement&gt;</u>	X		
<u>Union</u>	<u>IEnumerable&lt;T&gt;</u>		X	
<u>Where</u>	<u>IEnumerable&lt;T&gt;</u>		X	

# Запит до теоретично нескінченного джерела даних

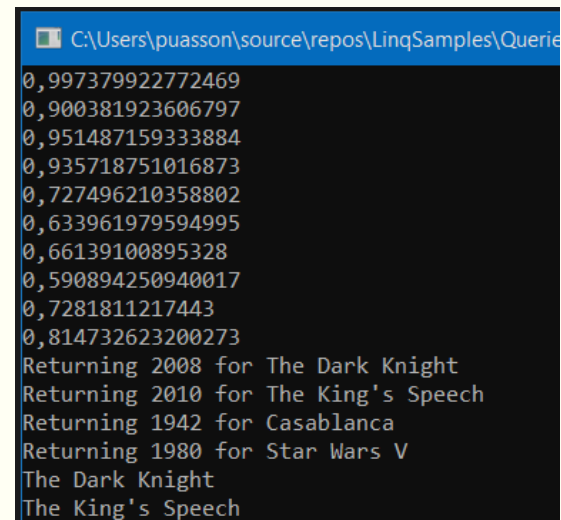
---

- Здійснюється з використанням тільки потокових операторів.
  - Додамо в клас MyLinq метод для генерування нескінченної послідовності випадкових чисел:

```
public static IEnumerable<double> Random()
{
    var random = new Random();
    while (true)
    {
        yield return random.NextDouble();
    }
}
```

- У методі Main() запишемо код, який вибиратиме 10 випадкових чисел (уникнення нескінченної генерації чисел), які більші за 0.5.

```
var numbers = MyLinq.Random().Where(n => n > 0.5).Take(10);
foreach (var number in numbers)
{
    Console.WriteLine(number);
}
```



```
C:\Users\puasson\source\repos\LinqSamples\Queries>
0,997379922772469
0,900381923606797
0,951487159333884
0,935718751016873
0,727496210358802
0,633961979594995
0,66139100895328
0,590894250940017
0,7281811217443
0,814732623200273
Returning 2008 for The Dark Knight
Returning 2010 for The King's Speech
Returning 1942 for Casablanca
Returning 1980 for Star Wars V
The Dark Knight
The King's Speech
```

# Про що потрібно пам'ятати

---

- Використовуйте простір імен `System.Linq` для застосування LINQ.
- LINQ API включає 2 основних статичних класи: `Enumerable` та `Queryable`.
- Статичний клас `Enumerable` містить методи розширення, які реалізують інтерфейс `IEnumerable<T>`.
- Тип `IEnumerable<T>` колекцій є in-memory колекцією на зразок `List`, `Dictionary`, `SortedList`, `Queue`, `HashSet`, `LinkedList`.
- Статичний клас `Queryable` включає методи розширення, які реалізують інтерфейс `IQueryable<T>`.
- Віддалений постачальник даних (Remote query provider) реалізує `IQueryable`, зокрема `Linq-to-SQL`, `LINQ-to-Azure` тощо.





# ДЯКУЮ ЗА УВАГУ!

Наступне запитання: Фільтрування, впорядкування та проектування даних