Технология LINQ

- **Language Integrated Query = Язык интегрированных запросов** к локальным и удаленным данным
- **Универсальность.** Единый язык (подход, операторы, ключевые слова) для выполнения запросов к разным источникам данных: локальные структуры (массивы, списки и др. последовательности), XML-документы, SQL-таблицы.
- **Безопасность.** Проверка типов в запросах осуществляется на этапе компиляции.
- **Удобство.** Составление запросов LINQ достаточно удобно за счет применения лямбда-выражений, методов-расширений. Язык LINQ включает более 40 операторов для решения типовых задач работы с данными

Технология LINQ

Objects	XML	Dataset	Entity
Локальные структуры данных (Array, List, ..)	Работа с документами в XML-формате	Удаленные источники данных (ADO.NET)	ADO.NET Entity

Средства C# 3.5 для LINQ

Неявно типизируемые переменные

Методы-расширения

Лямбда-выражения

Анонимные типы

Инициализация коллекций

Анонимные типы

- Объекты анонимных типов конструируются в контексте метода или лямбда-выражения

```
public void DoSomething()  {
        var person  = new {
                Name = “Jeffry”,
                LastName = “Richter”
        };
        ..
}
```

- При конструировании объекта указываем **названия свойств** и **значения** (константы, переменные, поля др.объекта)

Анонимные типы

- Заполняем константами:

```
var book1 = new { Title = “CLR via C#”, Year = 2014 };
```

- Заполняем локальными переменными:

```
string name = “John”; int age = 30;
var person = new { name, age };
```

- Заполняем полями другого объекта

```
Person p = new Person(“Jeffry”, “Richter”);
var book = new { title = “CLR”,
        author = p.Name[0] + “. “ + p.LastName };
```

Методы-расширения

- В C# существует возможность расширять функциональность типов, не изменяя описание этих типов, через методы-расширения
- Метод-расширение может быть объявлен в любом статическом классе

```
public static class Extenter {
    // Расширяем функциональность типа Object
    public static void HelloWorld(this object o) {
        Console.WriteLine("Hello, world! I'm " +
                          o.GetType().Name);
    }
    // Расширяем функциональность типов, реализующих
    //        интерфейс IList<Double>
    public static int CountIf(this IList<Double> o, double oIf) {
        ..
    }
}
```

```
// Полная форма записи
bool MyFunc(string s) { return s.Length >= 4; }
public void Main()
{
    ..
    string[] names = {“Tom”, “Dick”, “Harry”};
    IEnumerable<string> filtered =
      System.Linq.Enumerable.Where<string>(names,
            new Func<string, bool>(MyFunc));
    foreach (string s in filtered)
            Console.Write(n + “ “);

    //    Упрощенная форма записи
    var filtered = names.Where( s=> s.Length >= 4);

}
```

Две формы записи запросов

- Лямбда-синтаксис (*lambda syntax*)

```
var filtered = names.Where(s => s.Contains(“a”));
```

- Синтаксис запросов (*query syntax*)

```
var  filtered =
            from s in names
            where s.Contains(“a”)
            select s;
```

Лямбда-запросы

- Операторы запросов используют лямбда-выражения в качестве аргумента
- Лямбда-выражение описывает обработчик каждого элемента

  n => n + 1
- Тип входного элемента определяется по типу обрабатываемой последовательности
- Лямбда-выражение может состоять из фрагмента кода

```
int[] numbers = ..
var q = numbers.Where(n =>
                {
                        bool b;
                        ...
                        return b;
                });
```

Синтаксис запросов

- Запрос формируется с помощью ключевых слов (**from, select, where, orderby, descending, group by, let, into**)
- Расположение некоторых ключевых слов строго фиксировано: всегда начинается с конструкции **from,** всегда завершается либо конструкцией **select**, либо конструкцией **group**
- Транслируется компилятором в лямбда-синтаксис
- Не для всех методов имеются соответствующие ключевые слова в синтаксисе запросов

```
var q = from s in words
        where s.Contains("a")
        orderby s.Length descending
        select s;
```

Две формы записи для одинаковых задач

```
// Синтаксис запросов
var adults  =  from person in people
               where person.Age >= 18
               select person;
// Лямдба-синтаксис
var adults = people.Where(person => person.Age >= 18);


var adultNames    =    (from person in people
                        where person.Age >= 18
                        select person.Name).ToList();

var adultNames = people.Where(p=>p.Age>=18).ToList();
```

Основные операторы

- *Проекционные операторы*
  Select, SelectMany
- *Фильтрующие операторы*
  Where, Take, TakeWhile, Skip, SkipWhile, Distinct
- *Агрегирующие операторы:*
  Min, Max, Count, Sum, Aggregate
- *Поэлементные операторы*
  Single, First, Last, ElementAt, DefaultIfEmpty
- *Квантификаторы*
  Any, All, Contains, SequenceEqual
- *Группировка*
  GroupBy

Основные стандартные операторы

- *Методы упорядочивания*
  OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse
- *Методы объединения*
  Join, GroupJoin
- *Методы преобразования:*
  ToArray, ToList, ToDictionary, ToLookup, OfType, Cast
- *Методы генерации коллекций*
  Empty, Range, Repeat
- *Операции над множествами*
  Concat, Union, Intersect, Except

Select

- Проекционный оператор, выполняющий преобразование каждого элемента последовательности

```
// от массива String к массиву Int32
string[] data = {“23”, “34”, “78”, “21” };
int[] numbers = data.Select(s => int.Parse(s)).ToArray();

// от типа MethodInfo к анонимному типу
Type t = anyObject.GetType();
MethodInfo[] methods = t.GetMethods();
var selMethodInfo = methods.Select(m =>
                    new {
                            Name = m.Name,
                            Params = m.GetParameters().Length,
                            ReturnType = m.ReturnType
                    });
```

Select, SelectMany

- Оператор **SelectMany** осуществляет построение плоской последовательности однотипных элементов в случае иерархической организации исходной последовательности

```
var bag1 = files.Select(f => File.ReadAllText(f).Split(‘ ‘)).ToArray();

var bag2 = files.SelectMany(f => File.ReadAllText(f).Split(‘ ‘)).ToArray();
```

- Синтаксис запросов реализует оператор **SelectMany** следующим образом:

```
var words = from f in files
            from  w in File.ReadAllText(f).Split(‘ ‘)
            select w;
```

Select*

- Оператор Select содержит индексную перегрузку, позволяющую использовать индекс элемента при обработке

```
string[] names = {“Maria”, “Oleg”, “Ivan”, “Anna” };
var s = names.Select((s, idx) => “#” + idx + “. “ + s);
```

Where, Take, Skip, TakeWhile, SkipWhile

- Оператор **Where** выполняет отбор элементов, удовлетворяющих заданному условию

```
var longWords = allWords.Where(w => w.Length > 10);
// в форме запросов
var longWords = from w in allWords where w.Length > 10 select w;
```

- Оператор **Where** содержит индексную перегрузку, позволяющую использовать индекс элемента при проверке условия

- Оператор Take отбирает N –первых элементов

```
var firstWords = longWords.Take(10);
```

- Оператор Skip игнорирует N–первых элементов и отбирает все остальные

```
var anotherWords = longWords.Skip(10);
```

TakeWhile, SkipWhile

- Оператор **TakeWhile** отбирает элементы с начала последовательности до тех пор, пока выполняется указанное условие

  // Упорядоченный набор слов

  var sortedWords = ..

  var startWithA = sortedWords.**TakeWhile**(w => w.StartWith(“A”))

- Оператор **SkipWhile** игнорирует элементы с начала последовательности до тех пор, пока выполняется указанное условие

  var startNotWithA = sortedWords.**SkipWhile**(w => w.StartWith(“A”))

Distinct

- Оператор **Distinct** отбирает уникальные элементы последовательности

  var uniqueWords = manyWords.**Distinct()**.ToArray();

- При отборе используется метод GetHashCode, который должен возвращать одинаковый хэш-код для одинаковых элементов. При обработке пользовательских типов необходимо переопределить метод.

- Существует возможность настроить фильтрацию элементов с помощью внешнего объекта, реализующего интерфейс IEqualityComparer<T>

Агрегирующие операторы

- Операторы **Max, Min, Count, Sum** вычисляют для последовательности одно значение арифметического типа;

```
int maxValue = numbers.Max();
```

- Операторы содержат возможность дополнительной настройки:

```
// Вычисляем максимум по значениям, вычисленным для каждого
// элемента исходной последовательности
int maxParams = methods.Max(m =>
                            m.GetParameters().Length);

// Подсчитываем число элементов, удовлетворяющих условию
int longestWordsCount = allWords.Count(w =>
                w.Length >= 15);
```

Aggregate

```
// Реализуем попарное соединение элементов
string[] words = File.ReadAllText("file.txt").Split(' ');
string firstWords = words.Take(10)
                    .Aggregate((w1, w2) => w1 + ", " + w2);
Console.WriteLine(firstWords);

//Агрегируем с помощью накопителя
var totalDic = words.Aggregate(
                    // Исходный накопитель = новый словарь
                     new Dictionary<string, int>(),
                    // Обновляем накопитель dic, пробегая по словам w
                    (dic, w) => {
                                dic[w] = dic.ContainsKey(w) ?  dic[w]+1 : 1;
                                return dic;
                    }
        );
```

Single,First

- Поэлементные операторы возвращают только один элемент

```
// Первый элемент с начала последовательности
string firstWord = allWords.First();
// Первый элемент, удовлетворяющий условию
string firstCapsLockWord = allWords.First(w =>
                    w.All(c => char.IsUpper(c)));
// Единственный элемент, удовлетворяющий условию
string onlyOneCapsLockWord = allWords.Single(w =>
                    w.ToUpper().Equals(w));

// Первый элемент или значение по умолчанию, если элемент не найден
string s = allWords.FirstOrDefault(w => w.ToUpper().Equals(w));
```

Квантификаторы

- Операторы **Any, All, Contains, SequanceEquals** возвращают значение логичиского типа (true/false)

  bool bAnyLongWord = allWords.**Any**(w => w.Length > 15);

  bool bAllShortWords = allWords.**All**(w => w.Length < 5);

- Метод **Contains** ожидает в качестве аргумента объект того же типа, что и тип элементов последовательности

  Person p = new Person() { Name = “Igor”, Age = 23 };
  bool bContains = manyPersons.**Contains**(p);

- Существует возможность передачи функции сравнения элементов во внешнем объекте

  PersonEquality pEq = new PersonEquality();

  bool b = manyPersons.Contains(p, pEq);

GroupBy

- Оператор **GroupBy** выполняет группировку элементов по указанному ключу

```
string[] files = Directory.GetFiles(“C:\\”);
// Переходим к типам FileInfo для доступа к информации о файлах
var fileInfoData = files.Select(path => new FileInfo(path));
// Группируем файлы по расширениям
var grouppedFiles = fileInfoData.GroupBy(fi => fi.Extension);
```

- Существует несколько перегрузок метода GroupBy, позволяющих выполнить дополнительное преобразование групп элементов

```
var filesStat = fileInfoData.GroupBy(fi => fi.Extension,
        (key, group) => new {
                Расширение = key, Число = group.Count(),
                Макс_Размер = group.Max(f => f.Length) });
```

GroupBy

- Каждая группа содержит значение ключа группировки в встроенном свойстве **Key**:

```
var onlyExtensions = fileInfoData.GroupBy(fi=>fi.Extension)
                .Select(group => group.Key);
```

- Группа является также последовательностью элементов, к которой можно применять LINQ-операторы

```
var results = fileInfoData.GroupBy( fi => {
                return fi.Length < 1024 * 1024 ? “LESS_1MB” :
                        (fi.Length > 100 * 1024 * 1024 ? “MORE_100MB”
                                : “LESS_100MB”); })
                .Select(gr => gr.Key + “: “ + gr.Count());
```

Операции над множествами

```
int[] seq1 = { 1, 2, 3 }, seq2 = { 3, 4, 5 };
// Пересечение
var  commonality = seq1.Intersect (seq2);    //  { 3 }
// Исключение
var  difference1 = seq1.Except(seq2);    //  { 1, 2 }
var  difference2 = seq2.Except(seq1);    //  { 4, 5 }
// Объединение с повторами
var concat = seq1.Concat (seq2),    //  { 1, 2, 3, 3, 4, 5 }
// Объединение без повторов
var union  = seq1.Union(seq2);    //  { 1, 2, 3, 4, 5 }
```

```
// Объединение с ковариацией
MethodInfo[] methods = typeof (string).GetMethods();
PropertyInfo[] props = typeof (string).GetProperties();

// Объединяем два разнотипных массива
// с преобразованием к общему базовому классу
IEnumerable<MemberInfo> both =
    methods.Concat<MemberInfo> (props);

string[] MemberNames = (from mem in both
                        where mem.Name.ToUpper().StartsWith("S")
                        select mem.Name).ToArray();
```

Сортировка с помощью LINQ-запросов

Сортировка по нескольким ключам:

```
// в синтаксисе запросов
var orderedItems = from item in items
    orderby item.Rating descending, item.Price, item.Name
    select item;

// в лямбда-синтаксисе
var orderedItems = items
                    .OrderByDescending(item => item.Rating)
                    .ThenBy(item => item.Price)
                    .ThenBy(item => item.Name);
```

Объединение связанных списков

```
// Синтаксис запросов
var q1 = from defect in SampleData.AllDefects
         join subscription in SampleData.AllSubscriptions
         on defect.Project equals subscription.Project
         select new { defect.Summary, subscription.EmailAddress };

// Lambda-style
var q2 =  SampleData.AllDefects.Join(SampleData.AllSubscriptions,
              defect => defect.Project,
              subscription => subscription.Project,
              (defect, subscription) =>
                   new { defect.Summary, subscription.EmailAddress });
```

Отложенное выполнение

- Запрос описывает «правила» вычисления элементов
- Фактическое выполнение запроса осуществляется при обращении к элементам (например, в foreach-цикле)

```
// Объявление запроса
var q = numbers.Where(n => n % 2 == 0);
// Перебор элементов запроса
foreach(int element in q)
        Console.WriteLine(element);
```

- Немедленно выполняются запросы, которые завершаются:
  - операторами, возвращающими один элемент или скалярное значение (First, Count, All, Any и др.);
  - операторами преобразования типа: ToArray, ToList, ToDictionary, ToLookup

Отложенное выполнение

- Изменение входной последовательности

```
var numbers = new List<int>();
numbers.Add(1);
var q = numbers.Select(n=>n*10);
numbers.Add(2);
foreach(int n in q)
        Console.Write(n + " ");
// Получаем: ? ?

numbers.Clear();
foreach(int n in q)
        Console.Write(n + " ");
//  Получаем: ? ?
```

Захват внешних переменных

- При использовании внешних переменных в лямбда-выражении важно значение в момент выполнения, а не в момент определения запроса (при захвате реализуется связь по ссылке)

```
int[] numbers = { 1, 2 };
int factor = 10;
var q = numbers.Select(n => n * factor);

// Изменяем значение захваченной переменной и влияем на
// последующий результат запроса
factor = 20;

// Запускаем расчет элементов
foreach(int n in q)
        Console.Write(n + " ");
```

- Пример скрытой ошибки, связанной с отложенным выполнением:

```
// Замеряем время выполнения запроса
Stopwatch sw = new Stopwatch();
sw.Start();
var results = data.Select(el => SomeLongComputations(el));
sw.Stop();

// Удивляемся скорости LINQ: 0 мс
double time = sw.ElapsedMilleseconds;

// Исправляем эксперимент
sw.Start();
var results = data.Select(el => SomeLongComputations(el)).ToArray();
sw.Stop();
```

```
// Нахождение простых чисел без циклов и вспомогательных
    переменных

IEnumerable<int> numbers = Enumerable.Range(3, 1000000);

var q =
    from n in numbers
    where
        Enumerable.Range(2, (int)Math.Sqrt(n)).All(i => n % i > 0)
    select n;

int[] firstTen = q.Take(10).ToArray();
Console.WriteLine(string.Join(“, “, firstTen));
```

// Коррелированные подзапросы

```
DirectoryInfo[] dirs =
            new DirectoryInfo(@"d:\source").GetDirectories();
var query =
  from d in dirs
  where (d.Attributes & FileAttributes.System) == 0
  select new
  {
    DirectoryName = d.FullName,
    Created = d.CreationTime,
    Files = from f in d.GetFiles()
          where (f.Attributes & FileAttributes.Hidden) == 0
          select new { FileName = f.Name, f.Length, }
  };
foreach (var dirFiles in query)
{
  Console.WriteLine ("Directory: " + dirFiles.DirectoryName);
  foreach (var file in dirFiles.Files)
            Console.WriteLine ("  " + file.FileName + "Len: " + file.Length);
}
```

// Ключевое слов **let**

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<string> query =
    from n in names
    let vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                     .Replace ("o", "").Replace ("u", "")
    where vowelless.Length > 2
    orderby vowelless
    select n;
```

Встречаемость слов по файлам. Версия 1

```
var wordCounts = files
// читаем содержимое файлов/разбиваем на слова
        .SelectMany(path => File.ReadLines(path)
                                .SelectMany(line => line.Split(delimiters)))
        .GroupBy(word => word)
// для каждой группы формируем пару: «слово»-частота
        .Select(g => new { g.Key, Value = g.Count() })
// фильтруем по длине слов
        .Where(pair => pair.Key.Length > 4)
// упорядочиваем по убыванию частоты встречаемости
        .OrderBy(pair => -pair.Value)
        .ToList();
```

Встречаемость слов по файлам. Версия 2

```
var wordCounts = files
        .SelectMany(path => File.ReadLines(path)
                    .SelectMany(line =>
                                            line.Split(delimiters)))
        .Where(word => word.Length > 4)
        .GroupBy(word => word)
        .OrderBy(pair => -pair.Count())
        .ToDictionary(group => group.Key,
                                            group => group.Count());
```

Встречаемость слов по файлам. Версия 3

```
var counts3 = files
            .SelectMany(path => File.ReadLines(path)
                  .SelectMany(line => line.Split(delimiters)))
            .Where(word => word.Length > 4)
            .Aggregate(new Dictionary<string, int>(), (dic, w) => {
               if(dic.ContainsKey(w))
                  dic[w]++;
               else
                  dic.Add(w, 1);
               return dic;
            })
            .OrderBy(pair => -pair.Value)
            .ToDictionary(group => group.Key, group => group.Value);
```