



Информатика

операторы над последовательностями

Мотивация

- Пусть имеется массив (источник данных):

```
int[] source = {7, 3, -10, 17, 0, 9, 5};
```

- Нужно найти положительные элементы

```
{7, 3, 17, 9, 5}
```

- И упорядочить их по убыванию

```
{17, 9, 7, 5, 3}
```

Обычное решение

```
List<int> results = new List<int>();  
foreach (var i in source)  
{  
    if (i > 0)  
    {  
        results.Add(i);  
    }  
}  
  
results.Sort((x1, x2) => x2 - x1);
```

LINQ запрос

Method syntax

```
var results = source.Where(i => i > 0)
                    .OrderByDescending(i=>i);
```

Query syntax

```
var results = from i in source
              where i > 0
              orderby i descending
              select i;
```

LINQ

Language Integrated Query

- **Интегрированный язык запросов**
 - язык внутри C#
- В основе лежат идеи:
 1. функционального программирования
 2. языка структурированных запросов SQL

LINQ и ФП

- Основан на старых добрых идеях Lisp:

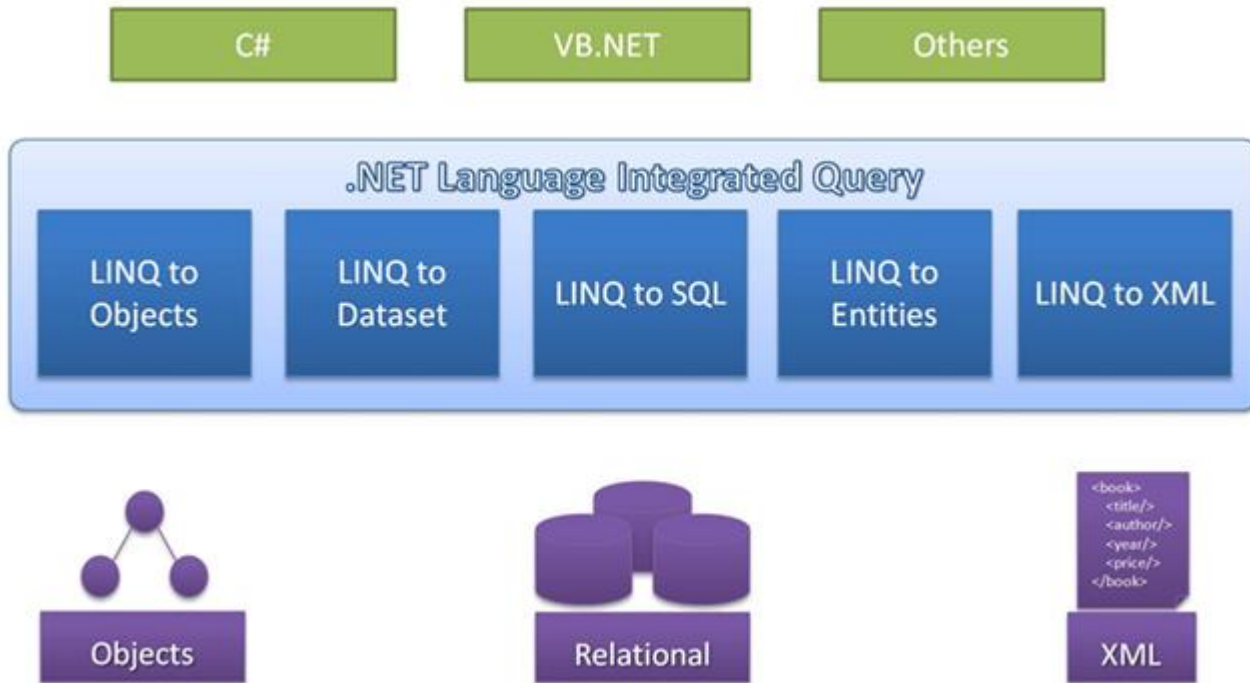
map, filter, reduce

- Программирование без побочных эффектов
- Ленивые вычисления

LINQ и SQL

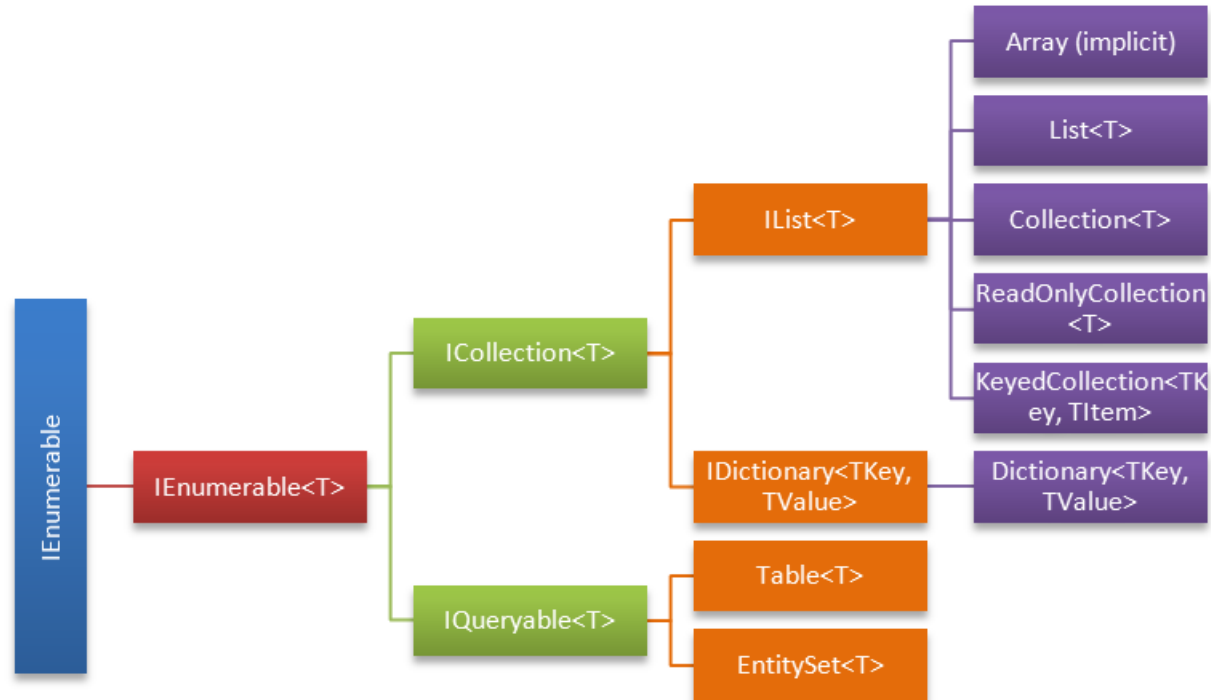
- LINQ – язык *запросов*
- LINQ запросы могут транслироваться в SQL для работы с БД
 - LINQ to SQL
 - LINQ to Entities

Инфраструктура LINQ



LINQ и коллекции

- LINQ может использоваться со всеми типами .NET коллекций



Концепции LINQ

- Последовательность
- Источник данных
- Запрос
- Оператор запроса
- Провайдер LINQ

Концепции LINQ

- Последовательность
 - коллекция с итератором
- Источник данных
 - Последовательность для запроса
- Запрос
 - Выражение, извлекающее информацию из последовательности (лениво)

Концепции LINQ

- Оператор запроса
 - Чистая функция, определённая на последовательностях, используемая в запросах
- Провайдер LINQ
 - Реализация операторов запросов для некоторого источника данных
 - LINQ to Objects, LINQ to SQL, LINQ to XML, LINQ to Twitter...

Списки в ФП: `map`, `filter`, `reduce`

- В функциональных языках списки играют особую роль
- С помощью фильтрации, проекции и свёртки списков можно решать широкий круг задач
- В С# наиболее близким аналогом списков из ФП являются последовательности

Проекция (Map)

- Применяет функцию преобразования к каждому элементу коллекции и возвращает получившуюся коллекцию
- Оператор Select в LINQ

```
var result = source.Select(x => x*x);
```

```
var uppercase = "Hello world!".Select(char.ToUpper);  
//что равносильно var uppercase = "Hello world!".ToUpper();
```

Фильтрация (Filter)

- Вычисляет предикат на каждом элементе коллекции и возвращает элементы, удовлетворяющие предикату
- Оператор Where в LINQ

```
var result = source.Where(i => i > 0);
```

Свёртка (Reduce | Fold)

- Применяет бинарную функцию на элементах коллекции и сворачивает коллекцию до одного значения
- ***Всё, что можно сделать в цикле, можно заменить свёрткой***
- Aggregate в LINQ

```
var result = source
    .Aggregate((acc, i) =>
        acc + (i > 0 ? i : 0));
```
- Частные случаи Count, Min, Max, Sum, Average

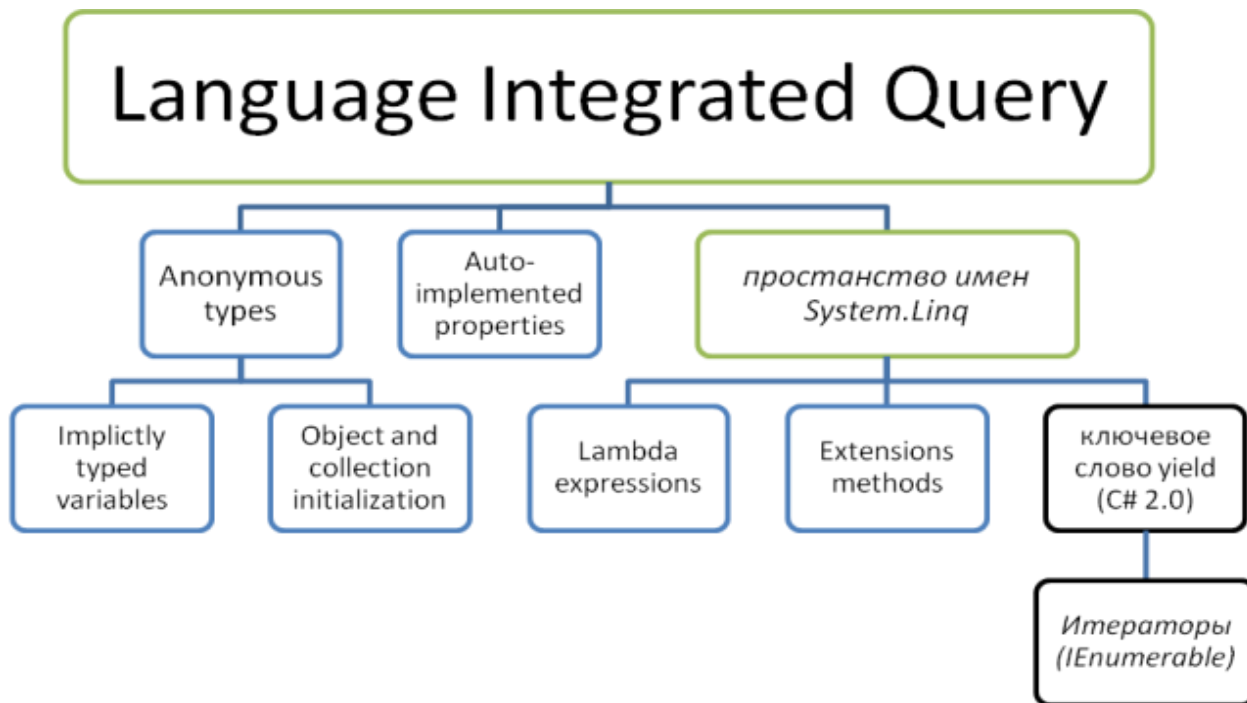
```
double result = source.Average();
```


Другие операторы LINQ

- Разбиение (Take/Skip)
- Объединение (Join, GroupJoin)
- Конкатенация
- Упорядочивание
- Группировка

Технологии в основе LINQ

Большинство элементов C# 3.0 создавались специально для LINQ



Последовательность IEnumerable

Коллекция элементов с итератором, доступным из метода GetEnumerator

```
public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

Последовательности и интервалы

```
IEnumerable<int> arraySeq =  
    new[] {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
foreach (var i in arraySeq)  
    Console.Write($"{i} ");  
Console.WriteLine();
```

```
IEnumerable<int> seq=Enumerable.Range(1, 10);  
foreach (var i in seq)  
    Console.Write($"{i} ");  
Console.WriteLine();
```

Генерация последовательностей

- Последовательности можно описывать как класс, реализующий `IEnumerable<T>`, с методом `GetEnumerator` создания экземпляров итератора для этого класса
- Класс итератора можно описать явно
- [Пример кода](#)

Генерация последовательностей

- Можно вместо явного описания итераторов использовать генераторы `yield return`
- [Пример кода](#)

IEnumerable - фабрика

- IEnumerable – *фабрика* создания итераторов (**паттерн фабричный метод**)
- Итератор – механизм получения текущего элемента и перехода к следующему
- *Существование коллекции необязательно!*

IEnumerable без коллекции

- Можно генерировать очень длинные последовательности (почти бесконечные)

```
static IEnumerable<long> Naturals()  
{  
    long i = 1;  
    while (i < long.MaxValue-1)  
        yield return i++;  
}
```


Что произойдёт?

```
static IEnumerable<long>
Naturals()
{
    long i = 1;
    while (i < long.MaxValue-1)
        yield return i++;
}
```

```
static void Main(string[] args)
{
    int i = 0;
    foreach(var n in
        Naturals())
    {
        if (i++ >= 10) break;
        Console.Write($"{n} ");
    }
    Console.WriteLine();
}
```

A tak?

```
static IEnumerable<long>
Naturals()
{
    long i = 1;
    while (i < long.MaxValue-1)
        yield return i++;
}
```

```
static void Main(string[] args)
{
    int i = 0;
    foreach(var n in
        Naturals().ToList())
    {
        if (i++ >= 10) break;
        Console.Write($"{n} ");
    }
    Console.WriteLine();
}
```

Ленивые вычисления

- При работе с IEnumerable последовательность не создаётся полностью
- Мы работаем с конечным автоматом
- Данные запрашиваются по одному
- *Это – ленивые вычисления*
- Во втором примере ToList заставил последовательность вычисляться

Ленивые вычисления

- Значение вычисляется не когда строится новая последовательность, а когда запускается перебор (MoveNext)
- Отложенное вычисление играет важное значение в LINQ
- Отделяет построение запроса от его исполнения

Лень и энергичность

- Все стандартные операторы запросов вычисляют значения лениво
- Есть исключения
 - Операторы, возвращающие скаляр: First, Count
 - Преобразования: ToArray, ToList, ToDictionary, ToLookup

Что плохого в коде?

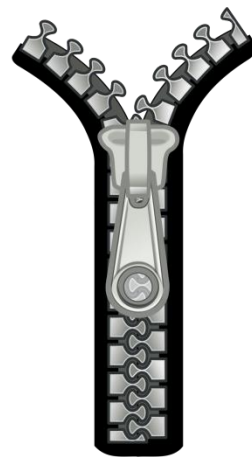
```
public static IEnumerable<PersonWithDish> GiveDishesBadWay
(IEnumerable<Person> people, IEnumerable<Dish> dishes)
{
    for (int i = 0; i < people.Count(); i++)
    {
        yield return new PersonWithDish()
        {
            PersonName = people.ElementAt(i).Name,
            DishName = dishes.ElementAt(i).Name
        };
    }
}
```

Как лучше

```
public static IEnumerable<PersonWithDish> GiveDishesBetterWay
(ReadOnlyList<Person> people, ReadOnlyList<Dish> dishes)
{
    for (int i = 0; i < people.Count; i++)
    {
        yield return new PersonWithDish()
        {
            PersonName = people[i].Name,
            DishName = dishes[i].Name
        };
    }
}
```

Ещё лучше: застёжка Zip

```
public static IEnumerable<PersonWithDish> GiveDishesBestWay
(IEnumerable<Person> people, IEnumerable<Dish> dishes)
{
    return people.Zip(dishes, (p, d) => new PersonWithDish
    {
        PersonName = p.Name,
        DishName = d.Name
    });
}
```



Использование var

- При работе с LINQ запросами рекомендуется использовать var
- Тип результата запроса может быть большим и страшным

```
IEnumerable<IGrouping<string, Person>> result  
    = people.GroupBy(n => n.Name);
```

VS

```
var result = people.GroupBy(n => n.Name);
```

Зачем еще использовать var

- Результатом запроса могут быть анонимные типы, их последовательности и группировки

```
var result = from person in people
              select new { person.FirstName,
                           person.LastName };
```

- Тип result – IEnumerable<'a>

где 'a = new { person.FirstName, person.LastName };

Анонимные типы

- Объявляются прямо в коде (inline)
 - наподобие лямбда выражений
- Экземпляры неизменяемые (immutable)
 - нельзя изменить значения свойств

```
var anonymous =  
    new {FirstName = "John", LastName = "Doe"};
```

- Созданы специально для LINQ

Методы расширения

Extension methods

- Публичные статические методы в публичных статических классах, принимающих первый аргумент как `this`
- Позволяют расширять функционал существующих типов (даже встроенных)

Без методов расширения плохо

Боль...

```
IEnumerable<int> result =  
    Enumerable.OrderByDescending  
    ( Enumerable.Where(source, x => x > 0),  
      x => x  
    );
```

Красота

```
var result = source.Where(x => x > 0)  
    .OrderByDescending(x => x);
```

Методы расширения

- Могут ещё сильнее украсить код

```
var period = 8.October(2015).To(DateTime.Today)
                                .Step(1.Days())
                                .Select(d => d.Date);
```

```
foreach (DateTime day in period)
{
    Console.WriteLine(day);
}
```

[Ссылка на код](#)

LINQ to Objects

- Подмножество LINQ, исполняемое в памяти
- Работает с любой .NET коллекцией – реализующей IEnumerable
- Не требует провайдера (такого, как LINQ to SQL, LINQ to XML)

Как реализован LINQ to Objects?

- Методами расширения
- Логика основана на преобразованиях последовательностей (итераторов)
- Хороший [цикл статей](#) по реализации LINQ to Objects от Jon Skeet

Реализация **фильтрации Where**

- Итерируем по последовательности
- Вычисляем значение предиката для каждого элемента
- Если элемент удовлетворяет предикату, возвращаем его в качестве следующего элемента результата (yield return)
 - Еще нужно проверить аргументы на null

Реализация Where

```
private static IEnumerable<TSource> WhereImpl<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    foreach (TSource item in source)
    {
        if (predicate(item))
        {
            yield return item;
        }
    }
}
```

Реализация проекции **Select**

- Итерируемся по последовательности
- Возвращаем в качестве следующего элемента результата результат вызова селектора (функции преобразования) на текущем элементе последовательности
 - Еще нужно проверить аргументы на null

Реализация **Select**

```
private static IEnumerable<TResult> SelectImpl
<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector)
{
    foreach (TSource item in source)
    {
        yield return selector(item);
    }
}
```

Реализация свёртки *Aggregate*

- Инициализируем аккумулятор
- Итерируемся по последовательности
- Вычисляем значение агрегатной функции на каждом элементе
- Обновляем аккумулятор
 - Еще нужно проверить аргументы на null

Реализация **Aggregate**

```
public static TAccumulate Aggregate<TSource, TAccumulate>(
    this IEnumerable<TSource> source,
    TAccumulate seed,
    Func<TAccumulate, TSource, TAccumulate> func)
{
    return source.Aggregate(seed, func, x => x);
}
```

Реализация **Aggregate**

```
public static TResult Aggregate<TSource, TAccumulate, TResult>(
    this IEnumerable<TSource> source,
    TAccumulate seed,
    Func<TAccumulate, TSource, TAccumulate> func,
    Func<TAccumulate, TResult> resultSelector)
{
    // Проверка аргументов на null, выбрасывание исключений
    TAccumulate current = seed;
    foreach (TSource item in source)
    {
        current = func(current, item);
    }
    return resultSelector(current);
}
```

Реализация Zip

- Итерируем по двум последовательностям одновременно, пока хотя бы одна не закончится
- Придётся работать с итераторами вручную (foreach не подойдёт)
- Вычисляем значение функции на каждой паре
- Возвращаем получившееся значение в качестве следующего элемента результата (yield return)
 - Ещё нужно проверить аргументы на null

LINQ с IList<T>

- В отличие от IEnumerable, LINQ при работе со списками IList может изменять данные – добавлять или удалять элементы из списка
- *Нарушается неизменяемость*

SQL внутри C# (почти)

- У LINQ есть два **равносильных** синтаксиса:
 - синтаксис запросов (Query Syntax)
 - синтаксис методов (Method Syntax)

```
var adults1 = people.Where(person => person.Age >= 18);
```

```
var adults2 = from person in people
               where person.Age >= 18
               select person;
```

Query Syntax

- Синтаксис запросов – синтаксических сахар для синтаксиса методов LINQ
– считайте, это препроцессинг
- SQL-подобное описание запросов к источникам данных

Query syntax

- Удобно использовать с объединениями и сортировками

```
from defect in SampleData.AllDefects
join subscription in SampleData.AllSubscriptions
  on defect.Project equals subscription.Project
select new { defect.Summary, subscription.EmailAddress };
```

```
SampleData.AllDefects.Join(SampleData.AllSubscriptions,
  defect => defect.Project, subscription => subscription.Project,
  (defect, subscription) => new
  {
    defect.Summary,
    subscription.EmailAddress
  });
```

Ключевые слова let и into

- В Query Syntax можно использовать let и into для хранения промежуточных результатов запроса
- Let создаёт новую переменную и инициализирует её результатом некоторого выражения
- Into создаёт новую переменную для хранения результатов применения группировки, объединения или проекции (select). Скрывает ранее использованную переменную

Ключевые слова let и into

```
var em = from e in emp
         group e by new { e.Salary, e.Id }
         into gEmp
         let avgSal = (gEmp.Sum(t => t.Salary) / gEmp.Count())
         where gEmp.Key.Salary == avgSal
         select new { gEmp.Key.Salary, gEmp.Key.Id };
```

Сортировка

- По умолчанию LINQ запросы не упорядочивают элементы
- Можно сортировать по возрастанию с помощью **OrderBy** (orderby в Query Syntax)
 - по убыванию с **OrderByDescending** (orderby ... descending в Query Syntax)

Примеры сортировки

```
var p= from s in people
        orderby s.FirstName descending
        select s;
```

```
var p2 = people.OrderByDescending(x => x.FirstName);
```

```
var p3= from s in people
        orderby s.FirstName , s.LastName descending
        select s;
```

```
var p4 = people.OrderByDescending(x => x.FirstName)
                .ThenBy(x => x.LastName);
```

Группировка

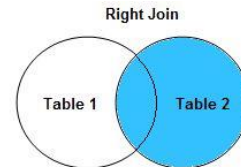
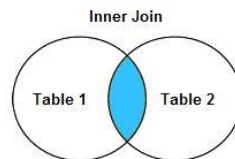
- Есть возможность группировать элементы по некоторому критерию
- К группам можно применять агрегатные функции

```
var p1 = from p in people  
        group p by p.LastName;
```

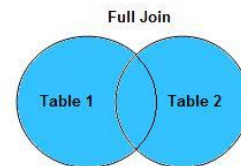
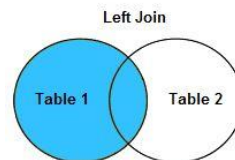
```
var p2 = people.GroupBy(x => x.LastName);
```

Объединения Join

- Есть возможность объединять источники данных для выполнения запросов
- Виды объединений
 - внутреннее
 - групповой
 - левое
 - декартово произведение



SQL Join



Внутренний Join

- Возвращает только элементы, содержащиеся в обоих источниках

```
from s in Students
join c in Cities on s.CityID equals c.CityID
select new { StudentName = s.FirstName, CityName = c.Name }
```

Внутренний Join

- Мы можем сравнивать элементы по сложным критериям используя анонимные типы
- Имена свойств анонимных типов должны совпадать

```
from s in Students
join a in Addresses on new { s.CityID, s.RegionID } equals
                        new { a.CityID, a.RegionID }
select new { StudentName = s.FirstName, CityName = a.CityName }
```

Group Join

- Мы можем генерировать последовательность объектов из одного источника вместе со связанными элементами второго источника
- Если связанных элементов нет, создастся пустой массив

Group Join

- Мы можем генерировать последовательность объектов из одного источника вместе со связанными элементами второго источника

```
from s in Students
join c in Courses on s.StudentID equals a.StudentID into t
select { StudentName = s.Name, Courses = t }
```

- Если связанных элементов нет, создастся пустой массив

Left Join

- Мы можем возвращать последовательность пар объектов из первого источника со связанными объектами из второго. Если связанного объекта во втором источнике нет, запишется null.

```
from s in Students
join a in Addresses on s.AddressID equals a.AddressID into t
from st in t.DefaultIfEmpty()
select { AddressID = (int?)st.AddressID,
        st.City, StudentName = s.Name }
```


Декартово произведение

- Мы можем генерировать все пары элементов из обоих источников

```
from s in Students  
from a in Addresses  
select { a.City, s.Name }
```

Parallel LINQ

- PLINQ – это расширение возможностей LINQ to Object возможностями параллельной обработки последовательности
- Для работы с PLINQ нужно преобразовать `IEnumerable<T>` в `ParallelQuery<T>` с помощью метода **AsParallel**

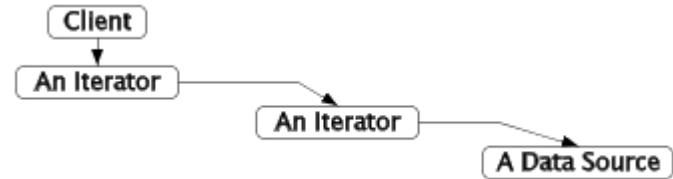
Порядок в Parallel LINQ

- При обработке запроса PLINQ данные разделяются для параллельной обработки
- Может нарушаться порядок относительно исходных данных
- Для сохранения порядка нужно использовать метод **AsOrdered**



LINQ и паттерн декоратор

- `IEnumerable<T>` не вычисляется пока не будет произведён перебор элементов последовательности (`foreach`, например)
- Операторы запросов LINQ возвращают список итераторов, декорирующих источник данных
- Паттерн **Декоратор**



IEnumerable + SelectMany = monad

- У монады есть две функции **return** и **bind** и нейтральный элемент
- **return** – втягивание в монаду: создание IEnumerable контейнера
- **bind** – связывание: реализовано стандартным методом SelectMany

IEnumerable + SelectMany = monad

Пример связывания:

```
IEnumerable<Shipper>
```

```
someWeirdListOfShippers =  
    customers
```

```
        .SelectMany(c => c.Addresses)
```

```
        .SelectMany(a => a.Orders)
```

```
        .SelectMany(o => o.Shippers);
```

Ковариация и контравариация

- Любопытные могли заметить что-то странное в интерфейсе IEnumerable
- `public interface IEnumerable<out T> ...`
- Вспомним принцип подстановки Барбары Лисков о правильном наследовании...

Предусловия и постусловия

- Принцип Барбары Лисков говорит о том, что
 - Предусловия не могут быть усилены в подклассе
 - Постусловия не могут быть ослаблены в подклассе
- Другими словами, *требовать меньше, а гарантировать больше*

Ковариация IEnumerable<out T>

- out – параметр типа говорит о ковариации
 - гарантируем больше
 - можем подставлять более узкий тип

```
// Covariance.
```

```
IEnumerable<string> strings = new List<string>();  
IEnumerable<object> objects = strings;
```

Контравариация Action<in T>

- in – параметр типа говорит о контравариации
 - требуем меньше
 - можем подставлять более широкий тип

// Предположим что метод SetObject объявлен в классе:

```
// static void SetObject(object o) { }
```

```
Action<object> actObject = SetObject;
```

```
Action<string> actString = actObject;
```

Ковариация и контравариация

```
static object GetObject() { return null; }  
static void SetObject(object obj) { }
```

```
static string GetString() { return ""; }  
static void SetString(string str) { }
```

```
static void Test()  
{  
    object[] array = new String[10];  
  
    Func<object> del = GetString;  
  
    Action<string> del2 = SetObject;  
}
```

[Статья на MSDN](#)

Пример использования LINQ

- Поиск прямоугольного треугольника
 - Задача с projecteuler.net

```
var triple=
    (from a in Enumerable.Range(1,333)
     from b in Enumerable.Range(a,500-a)
     let c = 1000 - a - b
     where a * a + b * b == c * c
     select new { a, b, c, product = a * b * c }
    ).Single();
Console.WriteLine(triple);
```



Вопросы?

e-mail: marchenko@it.kfu.ru