

# Разработка приложений на платформе .NET

---

Лекция 22  
LINQ

# Сегодня

- Расширяющие методы
- Неявная типизация
- Анонимные типы
- Language Integrated Query (LINQ)
  - LINQ to Objects

# Расширяющие методы

---



# Демонстрация

---

Расширяющие методы



# Расширяющие методы

- Расширяющие методы (Extension Method) появились в NET 3.5
- Расширяют общедоступный интерфейс типа
- Может расширять уже существующие, откомпилированные типы
- Добавляет новую функциональность
- Не могут нарушить инкапсуляцию
- Не являются методами экземпляра класса

# Расширяющие методы

- Определение
  - Должен определяться в статическом классе (и сам быть статическим)
  - Ключевое слово **this** перед первым параметром
  - Первый параметр – тип для которого создается расширяющий метод

```
public static class StringExtension
{
    public static void Print(this string s, int count)
    {    Console.WriteLine(s);    }
}
```

- Использование

```
string s = "Привет";
```

- Вызов метода экземпляра

```
s.Print(5);
```

- Вызов статического метода статического класса

```
StringExtension.Print(s, 5);
```

# Расширяющие методы

- По сути Расширяющие методы – это “синтаксический сахар” вызовов статических методов
- В отличии от обычных методов не имеет прямого доступа к членам типа: `private` и `protected` члены недоступны
- Для использования вызова метода экземпляра необходимо импортировать пространство имен, в котором определен расширяющий метод
- Использовать вызов расширяющего метода как статический метод расширяющего класса можно и с указанием полного имени без импорта пространства имен.
- Использование `using` имеет побочный эффект. Добавление расширяющих методов.



# Поиск нужного метода

1. В текущем типе
  2. В родительских типах
  3. Поиск расширений в текущем пространстве имен
  4. Поиск расширений в импортированных пространствах имен (using)
- Ограничения
    - Если в импортированных пространствах имен найдено несколько расширяющих методов с одной сигнатурой, то – ошибка



# Неявная типизация

---

# Ключевое слово var

- Заменяет объявление типа переменной
- Обязательно должна использоваться инициализация переменной при объявлении
- Компилятор по правой части определяет тип переменной и заменяет var этим типом
- Если компилятору не удаётся по правой части однозначно определить тип выражения – ошибка

- Примеры:

var i = 5;

преобразуется в int i =5;

var n;

ошибка

var d = 5.5; d="март";

ошибка, d имеет тип double

var f = условие? 5 : "текст";

Ошибка. Невозможно однозначно определить тип правой части\*

var l = new Dictionary<Complex, List<IEnumerable<Vector>>>();

var s = Math.Sign(5.5);

\* гипотетически, предположив, что тернарный оператор допускает такой вариант (что реально не так)

# Анонимные типы

---

# Анонимные типы

- Временный класс для объединения некоторого связанного набора данных
- Без методов, событий и др. функциональности
- Используется только в текущем контекста и не предназначен для многократного использования
- Определение:
  - Использование `var`
  - Указание пар свойство - значение

```
var a = new { Name = "Петя", LastName = "Иванов", Age = 7 };  
var a = new { Real = complex.Re, complex.Im, Name="число" };
```

- Компилятор сам присвоит имя для типа. Самим получить и использовать это имя нельзя \*



# Использование анонимных типов

- Анонимные типы наследуются от типа object
- Автоматически переопределяются ToString(), GetHashCode(), GetType(), Equals()
- Свойства – транслируются в доступные только для чтения свойства анонимного класса
- Обращение с переменной анонимного типа:
  - Console.WriteLine(a.LastName);
  - Console.WriteLine(a.ToString());
  - a.Age = 10; - Ошибка. Свойство только для чтения
- Все свойства задаются только в момент создания экземпляра анонимного типа – как параметры сгенерированного конструктора

# Сравнение анонимных типов

```
var a = new { Name = "Петя", LastName = "Иванов", Age = 7 };  
var b = new { Name = "Петя", LastName = "Иванов", Age = 7 };  
Console.WriteLine(a.GetType() == b.GetType());      true  
Console.WriteLine(a.Equals(b));                      true  
Console.WriteLine(a == b);                           false
```

- Equals() – проверяет каждую пару имя-значения на эквивалентность
- Компилятор генерирует новый тип только тогда, когда анонимный тип имеет уникальные имена свойств

# Демонстрация

---

Анонимные типы

# ВСПОМНИМ

- Инициализация объектов и коллекций
  - `Complex c = new Complex { re = 5, im = 7 };`
  - `var l = new List<Complex> { new Complex(),  
new Complex { re = 5, im = 7 },  
c };`
- Лямбда-выражения
  - `button.Click += (sender, e) => e.ToString();`
  - `result = Integral(x => x*x, a, b, count);`
- Стандартные делегаты
  - `TResult Func<T, TResult>(T t)`
  - `TResult Func<T1, T2, TResult>(T1 t1, T2 t2)`
  - `bool Predicate<T>(T t)`
  - `void Action<T> (T t)`



# Language Integrated Query (LINQ)

---

Язык интегрированных запросов

# LINQ

- Появился в NET 3.5
- Единый язык доступа к данным различной природы (objects, БД, XML, DataSet, Entity)
- SQL подобный язык
- Строго типизированные запросы
- В NET 4 появился параллельный вариант выполнения запросов (PLINQ)

# Виды LINQ

- LINQ to Object
- LINQ to XML
- LINQ to DataSet
- LINQ to SQL
- LINQ to Entities

# LINQ

- Основные сборки:
- `System.Core.dll` – общий для LINQ
- `System.Data.DataSetExtension.dll` – LINQ to DataSet
- `System.Xml.Linq.dll` – LINQ to XML
- Необходим импорт пространства имен `System.Linq`
- Реализован в виде Расширяющих методов



# LINQ

- `IEnumerable<T>` расширяет интерфейс `IEnumerable`
- Итерация по типам перечисления `IEnumerable<T>`
  - При импортировании пространства имен `System.Linq` многие типы получают “реализацию” интерфейса `IEnumerable<T>` и др. благодаря расширяющим методам вида:
    - `IEnumerable<T> переменная.AsEnumerable()`
- Итерация может быть по перечислениям любого типа
- Возвращаемые значения
  - Неизвестен возвращаемый тип, но он почти всегда реализует интерфейс `IEnumerable<T>`
  - Запросы могут возвращать анонимные типы

# Расширение IEnumerable<T>

- Массивы
- Обобщенные коллекции (System.Collections.Generic): List<T>, Dictionary<K,V> и др.
- Необобщенные коллекции (System.Collections): ArrayList, Hashtable и др. не реализуют IEnumerable<T>, но реализуют IEnumerable
- Если тип реализует IEnumerable, но не реализует IEnumerable<T> можно использовать механизм приведения к обобщенному интерфейсу OfType<T>() или Cast<T>()
  - IEnumerable<int> arrayList.OfType<int>()

# Выражения запросов

---



# Основные операции

- Любое LINQ выражение начинается с **from ... in ...** и заканчивается инструкцией **select**
- **from ... in ...** – позволяет извлечь данные из последовательности. Перечисляет значения из исходной последовательности
- **select** – выбирает новую последовательность из контейнера. Определяет возвращаемые данные  
var результат = **from** элемент **in** контейнер  
**select** возвращаемые данные
- Элемент – любое имя переменной, которое можно потом использовать в запросе
- Контейнер – последовательность данных  
IEnumerable<double> result = **from** complex **in** complexList  
**select** complex.Re;



# Получение подмножества данных

Фильтрация последовательности:

```
var результат = from элемент in контейнер  
                where условие  
                select возвращаемые данные
```

Условие – bool выражение (над каждым элементом)

Условие - может быть сложным bool выражением

- Пример:

```
IEnumerable<double> result = from complex in comlexList  
                             where complex.Abs > 5  
                             select complex.Re;
```

# Возвращение анонимных типов

- Возвращаемые набор данных может быть перечислением анонимного типа

```
var result = from complex in listOfComplex
              select new { complex.Re,
                           Modul = complex.Abs,
                           Sign = complex.re > 0 }

foreach (var item in result)
{
    Console.WriteLine("Re = {0}, Module = {1}, Sign = {2}",
                      item.Re,
                      item.Modul,
                      item.Sign );
}
```

# Сортировка данных

var результат = from элемент in контейнер  
                  orderby поле [descending][ascending]  
                  select возвращаемые данные

- Примеры:

```
IEnumerable<double> result = from complex in comlexList  
                             orderby complex.Re  
                             select complex.Re;
```

```
IEnumerable<Complex> result = from complex in comlexList  
                             where complex.Re > 0  
                             orderby complex.Abs descending  
                             select complex;
```

# Демонстрации

---

LINQ to Object  
Выражения запросов



# Join

Объединение двух последовательностей

```
var результат = from элемент1 in контейнер1
                join элемент2 in контейнер2
                on что-то_от_элемент1 equals
                что-то_от_элемент2
                select возвращаемые данные
```

- Пример:

```
var result = from c1 in comlexList1
              join c2 in comlexList2 on c1.Re equals c2.Re
              select new { c1, c2 };
```

# Группировка

Группировка данных

var результат = **from** элемент1 **in** контейнер1  
                  **group** что\_группируем **by** по\_чему\_группируем  
                  **into** групповая\_переменная  
                  **select** возвращаемые\_данные

- Пример:

```
IEnumerable<IGrouping<double, Complex>> complexGroup =  
    from c in complexList  
    group c by c.Re into g  
    select g;  
  
foreach (IGrouping<double, Complex> item in complexGroup)  
{  
    Console.WriteLine("Key {0}, value: {1}", item.Key, string.Join(", ", item));  
}
```

# Задание временной переменной

var результат = from элемент1 in контейнер1  
let переменная = выражение  
select возвращаемые данные

Заведенную переменную можно использовать только внутри выражения

- Пример:

```
IEnumerable<Complex> cs = from c in complexList  
    let max = Math.Max(c.Re, c.Im) * 0.9  
    where c.Re > max  
    select c;
```



# Отложенное выполнение

- Запрос не выполняется до тех пор, пока не будет начата итерация по последовательности
- Это позволяет применять один и тот же запрос многократно к одному и тому же контейнеру с гарантией получения свежих результатов
- Позволяет использовать итераторы по бесконечным коллекциям
- Внимание! Ошибки в запросе не проявятся пока не начнется итерация по последовательности
- Некоторые методы вызывают немедленное полное выполнение LINQ запроса.
  - ToArray(), ToList() и др.
  - Count(), OrderBy()\* и т.п.
  - Т.е. методы для работы которых нужен сразу весь результат



# Демонстрации

---

Отложенные запросы

# Точечная нотация

---



# Цепочка вызовов

- Выражения запросов преобразуются в вызов расширяющих методов

- Например:

```
IEnumerable<double> cs = from c in complexList
                        where c.Re > 0
                        select c.Abs;
```

Преобразуется в цепочку вызовов

```
IEnumerable<double> cs = complexList
                        .Where(c => c.Re > 0)
                        .Select(c => c.Abs);
```

- Fluent interface (цепочка вызовов)
  - `var result = list.Where(..).Distinct().OrderBy(...).Select(...)`
- Выражения запросов очень ограничены
- Возможно совмещать выражения запросов с точечной нотацией
- LINQ плохо относится к null последовательностям

# Операции

- Ограничение:
- **Where()** – фильтрация последовательности
  - `IEnumerable<TSource> Where<TSource>( this IEnumerable<TSource> source, Func<TSource, bool> predicate)`
  - `var res = complexList.Where(c => c.Re > 0);`
- Проекция:
- **Select()** – возврат новых элементов на основе входной последовательности
  - `IEnumerable<TResult> Select<TSource, TResult>(this IEnumerable<TSource> source, Func<TSource, TResult> selector)`
  - `IEnumerable<double> res = complexList.Select(c => c.Re);`
- **SelectMany()** – создание выходной последовательности с проекцией один ко многим из входной последовательности
  - `IEnumerable<TResult> SelectMany<TSource, TResult>(this IEnumerable<TSource> source, Func<TSource, IEnumerable<TResult>> selector)`
  - `IEnumerable<City> res = countries.SelectMany(c => c.Cities);`



# Разбиение последовательности

- **Разбиение:**
- **Take()** – возвращает первые N элементов последовательности
  - `IEnumerable<TSource> Take<TSource>(this IEnumerable<TSource> source, int count)`
  - `var res = complexList.Take(5);`
- **TakeWhile()** – возвращает первые элементы последовательности пока выполняется условие
  - `IEnumerable<TSource> TakeWhile<TSource>( this IEnumerable<TSource> source, Func<TSource, bool> predicate )`
  - `var res = complexList.TakeWhile(c => c.Re > 0);`
- **Skip()** – возвращает входную последовательность пропустив N первых
  - `IEnumerable<TSource> Skip<TSource>(this IEnumerable<TSource> source, int count)`
  - `var res = complexList.Skip(5);`
- **SkipWhile()** – пропускает первые элементы последовательности пока выполняется условие, возвращая остальные
  - `IEnumerable<TSource> SkipWhile<TSource>( this IEnumerable<TSource> source, Func<TSource, bool> predicate )`
  - `var res = complexList.SkipWhile(c => c.Re > 0);`

# Упорядочивание

- Сортировка последовательности:
- **OrderBy(), OrderByDescending()** – возвращает отсортированную последовательность
  - `IOrderedEnumerable<TSource> OrderBy<TSource, TKey>( this IEnumerable<TSource> source, Func<TSource, TKey> keySelector )`
  - `var res = complexList. OrderBy(c => c.Re);`
- **ThenBy(), ThenByDescending()** – возвращает отсортированную последовательность после OrderBy()
  - `IOrderedEnumerable<TSource> ThenBy<TSource, TKey>( this IOrderedEnumerable<TSource> source, Func<TSource, TKey> keySelector )`
  - `var res = complexList.OrderBy(c => c.Re).ThenBy(c => c.Im);`
- Перечисление в обратном порядке:
- **Reverse()** – перечисление в обратном порядке
  - `IEnumerable<TSource> Reverse<TSource>(this IEnumerable<TSource> source)`
  - `var res = complexList.Reverse();`

# Объединение последовательностей

- **Объединение последовательностей:**
- **Concat()** – объединяет 2 последовательности. Возвращает сначала все элементы первой последовательности, а затем второй
  - `IEnumerable<TSource> Concat<TSource>( this IEnumerable<TSource> first, IEnumerable<TSource> second )`
  - `var result = complexList.Concat(otherComplexList);`
- **Соединение:**
- **Join()** – возвращает объединенную последовательность
  - `IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>( this IEnumerable<TOuter> outer, IEnumerable<TInner> inner, Func<TOuter, TKey> outerKeySelector, Func<TInner, TKey> innerKeySelector, Func<TOuter, TInner, TResult> resultSelector )`
  - `var result = complexList.Join(otherComplexList, c => c.Re, c => c.Re, (c1, c2) => new { c1, c2 });`
- **GroupJoin()** – возвращает объединенную последовательность. В отличие от Join возвращает каждый элемент первой последовательности не более раза. При этом каждому элементу первой последовательности соответствует коллекция элементов второй последовательности
  - `IEnumerable<TResult> GroupJoin<TOuter, TInner, TKey, TResult>( this IEnumerable<TOuter> outer, IEnumerable<TInner> inner, Func<TOuter, TKey> outerKeySelector, Func<TInner, TKey> innerKeySelector, Func<TOuter, IEnumerable<TInner>, TResult> resultSelector )`
  - `var result = complexList.GroupJoin(complexList, c => c.Re, c => c.Re, (c1, c2) => new { c1, c2 });`



# Группировка

- **GroupBy()** – группируют последовательность по параметру
  - `IEnumerable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>(this IEnumerable<TSource> source, Func<TSource, TKey> keySelector)`
- `IEnumerable<IGrouping<double, Complex>> result = complexList.GroupBy(c => c.Re);`  
`foreach (IGrouping<double, Complex> group in result)`  
`{`  
`Console.WriteLine("Группа: ", group.Key);`  
`foreach (Complex item in group)`  
`{`  
`Console.WriteLine(item);`  
`}`  
`}`



# Множественные операции

- **Distinct()** – удаляет повторяющиеся элементы
  - `IEnumerable<TSource> Distinct<TSource>(this IEnumerable<TSource> source)`
  - `IEnumerable<Complex> result = complexList.Distinct();`
- **Union()** – возвращает объединение двух последовательностей как объединение двух множеств (удаляются одинаковые записи)
  - `IEnumerable<TSource> Union<TSource>(this IEnumerable<TSource> first, IEnumerable<TSource> second)`
  - `IEnumerable<Complex> result = complexList.Union(otherComplexList);`
- **Intersect()** – возвращает пересечение двух последовательностей как двух множеств
  - `IEnumerable<TSource> Intersect<TSource>( this IEnumerable<TSource> first, IEnumerable<TSource> second )`
  - `IEnumerable<Complex> result = complexList.Intersect(otherComplexList);`
- **Except()** – возвращает все элементы первой последовательности, которых нет во второй последовательности (вычитание множеств)
  - `IEnumerable<TSource> Except<TSource>( this IEnumerable<TSource> first, IEnumerable<TSource> second )`
  - `IEnumerable<Complex> result = complexList.Except(otherComplexList);`

# Преобразование типов

- Применимы к более общему интерфейсу `IEnumerable`. Таким образом любой тип реализующий не обобщенный интерфейс `IEnumerable` может использоваться в LINQ
- **`Cast<T>()`** – Преобразует каждый элемент последовательности к указанному типу. Если какой-то элемент не удалось преобразовать, то сгенерируется исключение `InvalidCastException`.
  - `IEnumerable<TResult> Cast<TResult>( this IEnumerable source )`
  - `IEnumerable<Complex> result = arrayList.Cast<Complex>();`
- **`OfType<T>()`** – Преобразует каждый элемент последовательности к указанному типу. Если какой-то элемент не удалось преобразовать, он пропускается. Исключение не сгенерируется.
  - `IEnumerable<TResult> OfType<TResult>(this IEnumerable source)`
  - `IEnumerable<Complex> result = arrayList.OfType<Complex>();`

# Преобразование в коллекции

- Выполняются сразу (не отложенные операции)
- **ToArray()** – преобразует последовательность в массив
  - `Complex[] result = complexList.ToArray();`
- **ToList()** – преобразует последовательность в список.
  - `List<Complex> result = complexList.ToList();`
- **ToDictionary()** – преобразует последовательность в словарь
  - `Dictionary<TKey, TSource> ToDictionary<TSource, TKey>`  
`(this IEnumerable<TSource> source,`  
`Func<TSource, TKey> keySelector )`
  - `Dictionary<TKey, TElement> ToDictionary<TSource, TKey, TElement>`  
`(this IEnumerable<TSource> source,`  
`Func<TSource, TKey> keySelector,`  
`Func<TSource, TElement> elementSelector)`
  - `Dictionary<double, Complex> result = complexList.ToDictionary(c => c.Re);`
  - `Dictionary<double, double> result = complexList.ToDictionary(c => c.Re, c => c.Abs);`
- **ToLookup()** – преобразует последовательность в словарь (ключи могут повторяться)
  - `ILookup<TKey, TSource> ToLookup<TSource, TKey>`  
`(this IEnumerable<TSource> source,`  
`Func<TSource, TKey> keySelector)`
  - `ILookup<TKey, TElement> ToLookup<TSource, TKey, TElement>`  
`(this IEnumerable<TSource> source,`  
`Func<TSource, TKey> keySelector,`  
`Func<TSource, TElement> elementSelector )`
  - `ILookup<double, Complex> result = complexList.ToLookup(c => c.Re);`
  - `ILookup<double, double> result = complexList.ToLookup(c => c.Re, c => c.Abs);`



# Отдельные элементы

- Выполняются сразу (не отложенные операции)
- **First()** – первый элемент последовательности (или первый элемент последовательности удовлетворяющий предикату). Если последовательность пустая, то сгенерируется исключение
  - `Complex result = complexList.First();`
- **FirstOrDefault()** – первый элемент последовательности (или первый элемент последовательности удовлетворяющий предикату). Если последовательность пустая, то вернется элемент по умолчанию
  - `Complex result = complexList.FirstOrDefault();`
- **Last()** – последний элемент последовательности (или последний элемент последовательности удовлетворяющий предикату). Если последовательность пустая, то сгенерируется исключение
  - `Complex result = complexList.Last();`
- **LastOrDefault()** – последний элемент последовательности (или последний элемент последовательности удовлетворяющий предикату). Если последовательность пустая, то вернется элемент по умолчанию
  - `Complex result = complexList.LastOrDefault();`
- **Single()** – первый и единственный элемент последовательности (или первый элемент последовательности удовлетворяющий предикату). Если последовательность пустая или в последовательности более одного элемента сгенерируется исключение
  - `Complex result = complexList.Single();`
- **SingleOrDefault()** – первый и единственный элемент последовательности (или первый элемент последовательности удовлетворяющий предикату). Если последовательность пустая, то вернется элемент по умолчанию. Если в последовательности более одного элемента сгенерируется исключение
  - `Complex result = complexList.SingleOrDefault();`
- **ElementAt()** – Возвращает элемент из последовательности по указанному индексу. Если в последовательности нет такого элемента, то сгенерируется исключение
  - `Complex result = complexList.ElementAt(5);`
- **ElementAtOrDefault()** – Возвращает элемент из последовательности по указанному индексу. Если в последовательности нет такого элемента, то вернется значение по умолчанию
  - `Complex result = complexList.ElementAtOrDefault(5);`



# Квантификаторы

- Выполняются сразу (не отложенные операции)
- **Any()** – возвращает true, если последовательность имеет хотя бы один элемент или если любой из элементов удовлетворяет условию
  - `bool result = complexList.Any();`
  - `bool result = complexList.Any(c => c.Re > 0);`
- **All()** – возвращает true, если все элементы удовлетворяют условию
  - `bool result = complexList.All(c => c.Re > 0);`
- **Contains()** – возвращает true, если последовательность содержит указанный элемент
  - `bool result = complexList.Contains(complex);`

# Агрегация

- Выполняются сразу (не отложенные операции)
- **Count(), LongCount()** – возвращает количество элементов в последовательности (если последовательность реализует `IList`, то возьмется свойство `IList.Count`)
  - `int result = complexList.Count();`
- **Sum()** – возвращает сумму числовых значений последовательности
  - `int result = intList.Sum();`
  - `double result = complexList.Sum(c => c.Re);`
- **Min(), Max(), Average()** – возвращает минимум, максимум или среднее арифметическое значение числовых значений последовательности
  - `int result = intList.Min();`
  - `double result = complexList.Min(c => c.Re);`
- **Aggregate()** – агрегирует последовательность с использованием пользовательской функции
  - `TSource Aggregate<TSource>`  
`(this IEnumerable<TSource> source,`  
`Func<TSource, TSource,`  
`TSource> func)`
  - `TResult Aggregate<TSource, TAccumulate, TResult>`  
`(this IEnumerable<TSource> source,`  
`TAccumulate seed,`  
`Func<TAccumulate, TSource, TAccumulate> func,`  
`Func<TAccumulate, TResult> resultSelector)`
  - `double result = complexList.Aggregate(1.0, (accumulation, currntElement) => accumulation * currntElement.Re);`

# Демонстрации

---

LINQ