Операции OrderBy и OrderByDescending

Операции упорядочивания позволяют выстраивать входные последовательности в определенном порядке. Важно отметить, что и OrderBy, и OrderByDescending требуют входной последовательности типа IEnumerable<T> и возвращают последовательность типа IOrderedEnumerable<T>. Передавать операциям OrderBy и OrderByDescending в качестве входной последовательности IOrderedEnumerable<T> нельзя. Причина в том, что последующие вызовы операций OrderBy и OrderByDescending не принимают во внимание порядок, созданный предыдущими вызовами OrderBy и OrderByDescending. Это значит, что передавать последовательность, возвращенную из OrderBy либо OrderByDescending, в последующий вызов операции OrderBy или OrderByDescending не имеет смысла.

Если требуется большая степень упорядочивания, чем возможно достичь с помощью одиночного вызова операции OrderBy или OrderByDescending, необходимо последовательно вызывать операции ThenBy или ThenByDescending, речь о которых пойдет в следующей статье.

OrderBy

Операция OrderBy позволяет упорядочить входную последовательность на основе *метода keySelector*, который возвращает значение ключа для каждого входного элемента. Упорядоченная выходная последовательность IOrderedEnumerable<T> выдается в порядке возрастания на основе значений возвращенных ключей.

Сортировка, выполненная операцией OrderBy, определена как неустойчивая. Это значит, что она не сохраняет входной порядок элементов. Если два входных элемента поступают в операцию OrderBy в определенном порядке, и значения ключей этих двух элементов совпадают, их расположение в выходной последовательности может остаться прежним или поменяться, причем ни то, ни другое не гарантируется. Даже если все выглядит нормально, поскольку порядок определен как неустойчивый, всегда следует исходить из этого. Это значит, что никогда нельзя полагаться на порядок элементов, поступающих из операций OrderBy или ThenBy, для

любого поля кроме указанного в вызове метода. Сохранение любого порядка, который существует в последовательности, передаваемой любой из этих операций, не может гарантироваться.

Операция OrderBy имеет два прототипа, описанные ниже:

Первый прототип OrderBy

C#

```
public static IOrderedEnumerable<T> OrderBy<T, K>(
          this IEnumerable<T> source,
          Func<T, K> keySelector)
    where
          K : IComparable<K>;
```

В этом прототипе операции OrderBy передается входная последовательность source и делегат метода keySelector, а возвращается объект, который при перечислении проходит входную коллекцию source, собирая все элементы и передавая каждый из них методу keySelector, таким образом, извлекая каждый ключ и упорядочивая последовательность на основе этих ключей.

Методу keySelector получает входной элемент типа T и возвращает поле внутри элемента, которое используется в качестве значения ключа типа K для этого входного элемента. Типы K и T могут быть одинаковыми или разными. Тип значения, возвращенного методом keySelector, должен реализовывать интерфейс IComparable (../../../csharp/charp_theory/level12/12_16.php).

Второй прототип OrderBy

C#

```
public static IOrderedEnumerable<T> OrderBy<T, K>(
       this IEnumerable<T> source,
       Func<T, K> keySelector,
       IComparer<K> comparer);
```

Этот прототип такой же, как первый, за исключением того, что он позволяет передавать объект-компаратор. Если используется эта версия операции OrderBy, то нет необходимости в том, чтобы тип K реализовывал интерфейс IComparable.

Пройди тесты

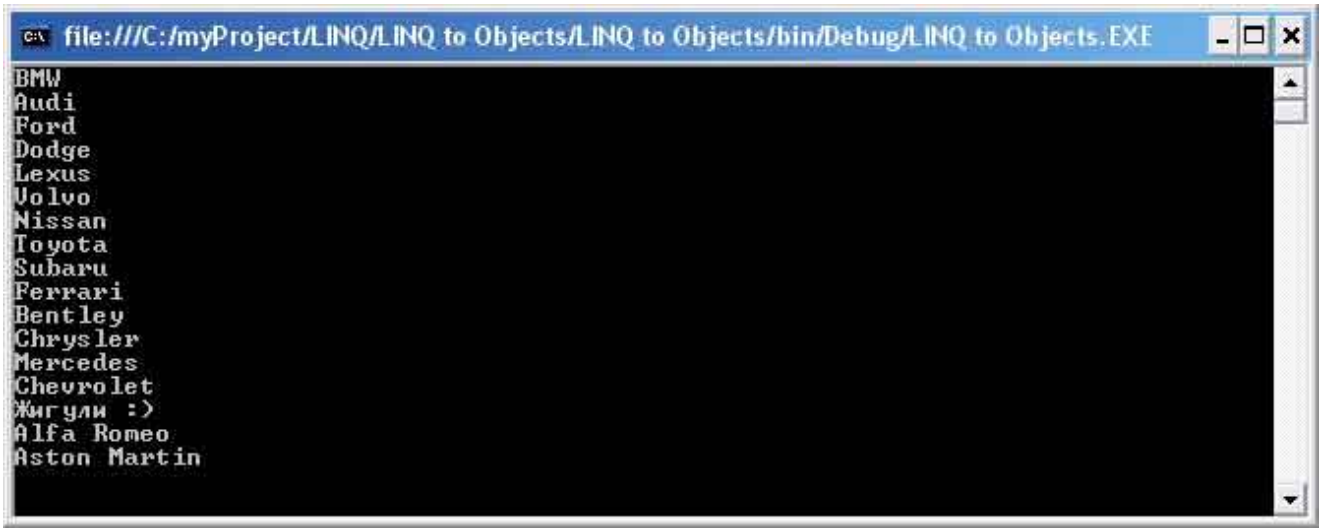
В следующем коде показан пример вызова первого прототипа:

C# тест (легкий) (https://professorweb.ru/test/c-sharp-test.html)

C# .NET тест (средний) (https://professorweb.ru/test/asp-test.html)

```
string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet",  "Chrysler",
"Dodge", "BMW",
                            "Ferrari",  "Bentley", "Ford", "Lexus", "Mercedes", "Toyota",
"Volvo", "Subaru", "Жигули :)"};

IEnumerable<string> auto = cars.OrderBy(s => s.Length);

foreach (string str in auto)
        Console.WriteLine(str);
```

Код в этом примере упорядочивает список машин по длине их названия. Результат выглядит следующим образом:



Теперь испытаем пример второго прототипа с использованием собственного компаратора. Прежде чем продемонстрировать код, стоит взглянуть на интерфейс IComparer (../../../csharp/charp_theory/level12/12_17.php):

C#

```
interface IComparer<T> {
       int Compare (T x, T y);
}
```

Интерфейс IComparer требует реализации единственного метода по имени Compare. Этот метод принимает два аргумента одного и того же типа T и возвращает значение int больше нуля, если первый аргумент больше второго, ноль — если аргументы эквивалентны, и значение меньше нуля — если первый аргумент меньше второго. Обратите внимание, насколько в этом интерфейсе и прототипе полезны обобщения C#.

В этом примере для ясности какой-либо компаратор по умолчанию не используется. Просто создается класс, реализующий интерфейс IComparer, который упорядочивает элементы на основе соотношения гласных и согласных:

C#

Пройди тесты

C# тест (легкий) (https://professorweb.ru/test/c-sharp-test.html)

.NET тест (средний) (https://professorweb.ru/test/asp-test.html)

```
public class MyVowelToConsonantRatioComparer : IComparer<string>
  {
    public int Compare(string s1, string s2)
    {
      int vCount1 = 0;
      int cCount1 = 0;
      int vCount2 = 0;
      int cCount2 = 0;

      GetVowelConsonantCount(s1, ref vCount1, ref cCount1);
      GetVowelConsonantCount(s2, ref vCount2, ref cCount2);

      double dRatio1 = (double)vCount1 / (double)cCount1;
      double dRatio2 = (double)vCount2 / (double)cCount2;

      if (dRatio1 < dRatio2)
        return (-1);
      else if (dRatio1 > dRatio2)
        return (1);
      else
        return (0);
    }

    //  Это общедоступный метод, так что код, использующий
    //  данный компаратор, может получить нужные значения
    public void GetVowelConsonantCount(string s,
                                       ref int vowelCount,
                                       ref int consonantCount)
    {
      // Гласные

      string vowels = "AEIOUYAEЁИОУЫЭЮЯ";

      // Инициализация счетчиков
      vowelCount = 0;
      consonantCount = 0;

      // Преобразуем в верхний регистр, чтобы избежать зависимости от регистра
      string sUpper = s.ToUpper();

      foreach (char ch in sUpper)
      {
        if (vowels.IndexOf(ch) < 0)
          consonantCount++;
        else
          vowelCount++;
      }

      return;
    }
  }
```

Пройди тесты

C# тест (легкий) (https://professorweb.ru/test/c-sharp-test.html)

.NET тест (средний) (https://professorweb.ru/test/asp-test.html)

Этот класс содержит два метода — Compare и GetVowelConstantCount. Метод Compare требуется интерфейсом IComparer. Метод GetVowelConstantCount необходим для внутреннего использования в Compare и позволяет получить количество гласных

и согласных во входной строке. Также нужна возможность вызывать ту же логику за пределами метода Compare, чтобы можно было получать значения для отображения при проходе циклом по упорядоченной последовательности.

Что конкретно делает компаратор — не столь важно. Весьма маловероятно, что когда-либо понадобится определять соотношение гласных и согласных в строке, и еще менее вероятно, что придется сравнивать две строки на основе этого соотношения. Более важно то, как был создан класс, реализующий интерфейс IComparer за счет реализации метода Compare. Здесь можно наблюдать обычную реализацию метода Compare в блоке if/else в конце метода. Как видите, в этом блоке кода возвращается -1, 1 или O, что требует контракт интерфейса IComparer.

Теперь можно использовать этот код:

C#

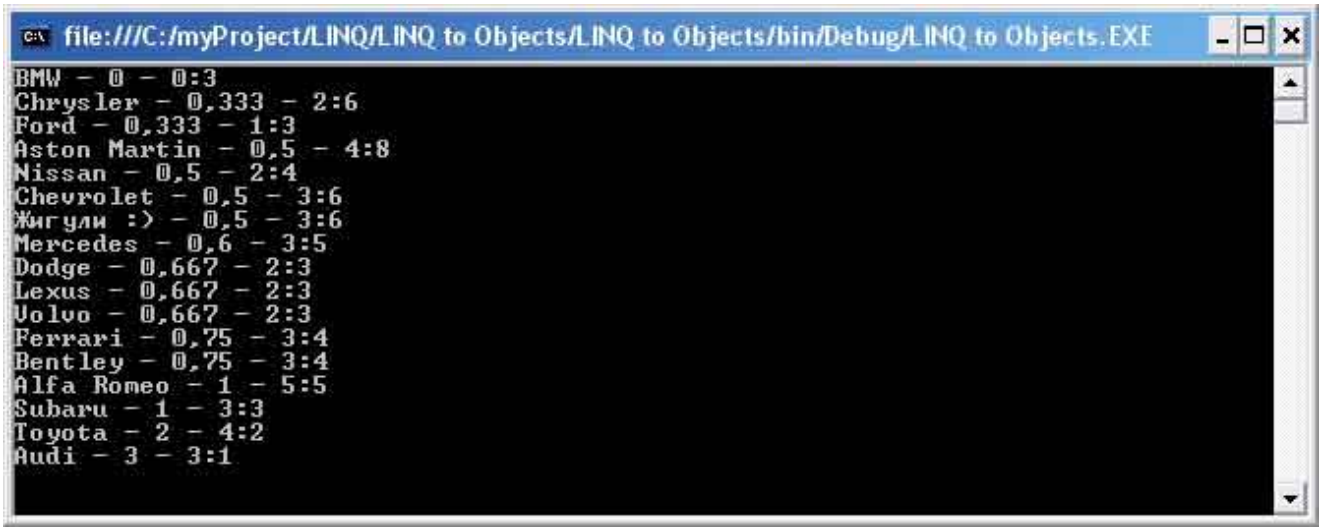
```
string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet",  "Chrysler",
"Dodge", "BMW",
                            "Ferrari",  "Bentley", "Ford", "Lexus", "Mercedes", "Toyota",
"Volvo", "Subaru", "Жигули :)"};

            MyVowelToConsonantRatioComparer myComp = new MyVowelToConsonantRatioComparer();
            IEnumerable<string> auto = cars.OrderBy((s => s), myComp);

            foreach (string str in auto)
            {
                int vCount = 0;
                int cCount = 0;
                myComp.GetVowelConsonantCount(str, ref vCount, ref cCount);
                double dRatio = Math.Round(((double)vCount / (double)cCount),3);
                Console.WriteLine(str + " - " + dRatio + " - " + vCount + ":" + cCount);
            }
```

Перед вызовом операции OrderBy создается экземпляр компаратора. Его экземпляр можно было бы создать в самом вызове метода OrderBy, но тогда не получилось бы ссылаться на него в цикле foreach. Ниже показаны результаты запуска примера:

Пройди тесты

C# тест (легкий) (https://professorweb.ru/test/c-sharp-test.html)

.NET тест (средний) (https://professorweb.ru/test/asp-test.html)



Как видите, элементы с меньшим соотношением гласных и согласных, выводятся первыми.

OrderByDescending

Эта операции прототипирована и ведет себя подобно операции OrderBy, но с тем отличием, что упорядочивает по убыванию. Ниже показан пример использования данной операции:

C#

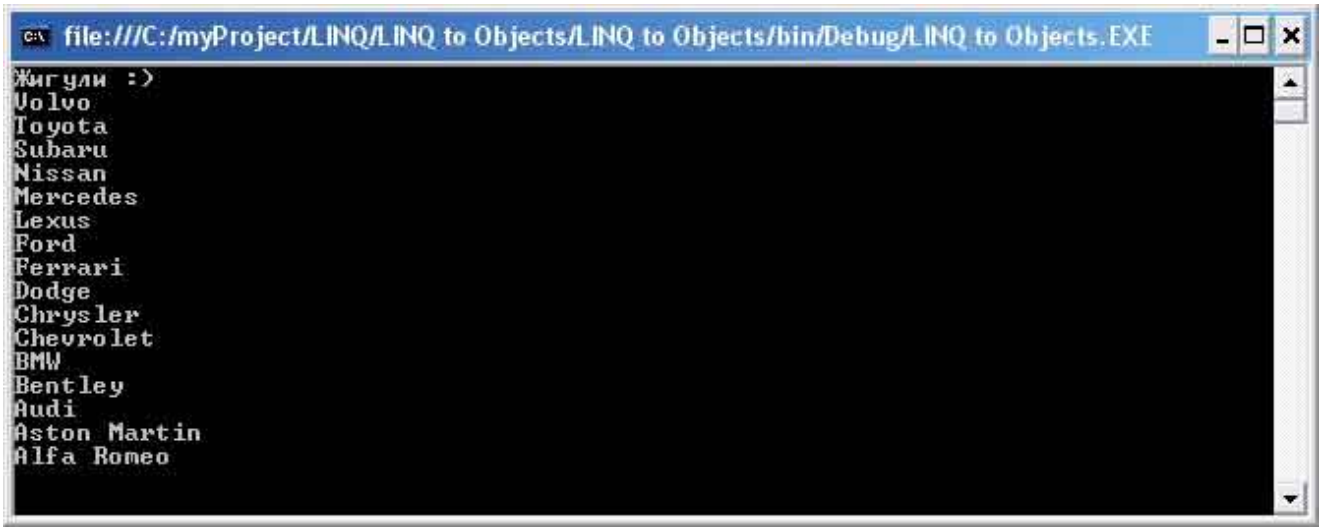
```
string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet",  "Chrysler",
"Dodge", "BMW",
                            "Ferrari",  "Bentley", "Ford", "Lexus", "Mercedes", "Toyota",
"Volvo", "Subaru", "Жигули :)"};

IEnumerable<string> auto = cars.OrderByDescending(s => s);

foreach (string str in auto)
        Console.WriteLine(str);
```

Как видите, названия машин располагаются в порядке по убыванию:

Пройди тесты

C# тест (легкий) (https://professorweb.ru/test/c-sharp-test.html)

.NET тест (средний) (https://professorweb.ru/test/asp-test.html)



Назад (2_4.php)	4	5	6	Вперед (2_6.php)

Пройди тесты

C# тест (легкий) (https://professorweb.ru/test/c-sharp-test.html)

.NET тест (средний) (https://professorweb.ru/test/asp-test.html)