# Операция ToLookup



Операция ToLookup создает объект Lookup типа <K, T> или, возможно, <K, E> из входной последовательности типа T, где K — тип ключа, а T — тип хранимых значений. Либо же, если Lookup имеет тип <K, E>, то типом хранимых значений может быть E, который отличается от типа элементов входной последовательности T.

Хотя все прототипы операции ToLookup создают Lookup, возвращают они объект, реализующий **интерфейс ILookup**. В этой статье объект, реализующий интерфейс ILookup, обычно будет называться просто Lookup.

Если вы незнакомы с классом Lookup коллекций C#, то знайте, что он позволяет хранить элементы, которые могут быть извлечены по ключу. Каждый ключ должен быть уникальным, и под одним ключом может быть сохранено множество элементов. Обращение по индексу к Lookup с применением ключа извлекает последовательность сохраненных с этим ключом элементов.

Операция ToLookup имеет четыре прототипа, описанные ниже:

### Первый прототип операции ToLookup

C#

```
public static ILookup<K, T> ToLookup<T, K>(
       this IEnumerable<T> source,
       Func<T, K> keySelector);
```

В этом прототипе создается Lookup типа <K, T> и возвращается за счет перечисления входной последовательности source. Делегат метода keySelector вызывается для извлечений ключевого значения из каждого входного элемента, и этот ключ является ключом элемента в Lookup. Эта версия операции сохраняет в Lookup значения того же типа, что и элементы входной последовательности.

Поскольку прототип предотвращает указание объекта проверки эквивалентности IEqualityComparer<K>, данная версия Lookup по умолчанию использует в качестве такого объекта экземпляр класса EqualityComparer<K>.Default.



### Второй прототип операции ToLookup

Второй прототип ToLookup подобен первому, за исключением того, что предоставляет возможность указывать объект проверки эквивалентности IEqualityComparer<K>. Он рассматривается ниже:

C#

```
public static ILookup<K, T> ToLookup<T, K>(
     this IEnumerable<T> source,
     Func<T, K> keySelector,
     IEqualityComparer<K> comparer);
```

Этот прототип дает возможность указывать объект comparer типа IEquality Comparer<K> для проверки эквивалентности ключей. Поэтому при добавлении или обращении к элементу в Lookup он использует этот объект comparer для сравнения указанного ключа с ключами, уже находящимися в Lookup, определяя их соответствие.

Реализация по умолчанию интерфейса IEqualityComparer<K> предоставляется EqualityComparer.Default. Однако если вы собираетесь использовать класс проверки эквивалентности по умолчанию, то указывать объект, проверяющий эквивалентность, незачем, потому что предыдущий прототип применяет его в любом случае. Класс StringComparer реализует некоторые классы проверки эквивалентности, такие как, например, игнорирующий различия в регистре. Таким образом, ключи "Joe" и "joe" трактуются как один и тот же ключ.

### Третий прототип операции ToLookup

Третий прототип ToLookup подобен первому, за исключением того, что позволяет указывать селектор элемента, чтобы тип данных значения, сохраненного в Lookup, мог отличаться от типа элементов входной последовательности. Третий прототип представлен ниже:

C#

```
public static ILookup<K, E> ToLookup<T, K, E> (
        this IEnumerable<T> source,
        Func<T, K> keySelector,
        Func<T, E> elementSelector);
```

В аргументе elementSelector можно указывать делегат метода, возвращающий часть входного элемента, либо вновь созданный объект совершенно другого типа, который должен быть сохранен в Lookup.



### Четвертый прототип операции ToLookup

Четвертый прототип операции ToLookup концентрирует в себе лучшее из предыдущих. Он представляет собой комбинацию второго и третьего прототипов, а это означает, что можно указывать elementSelector и объект проверки эквивалентности comparer. Четвертый прототип описан ниже:

C#

```
public static ILookup<K, E> ToLookup<T, K, E>(
    this IEnumerable<T> source,
    Func<T, K> keySelector,
    Func<T, E> elementSelector,
    IEqualityComparer<K> comparer);
```

Этот прототип позволяет указывать и elementSelector, и comparer.

Если аргумент source, keySelector или elementSelector равен null, либо ключ, возвращенный keySelector, равен null, генерируется исключение ArgumentNullException.

В следующем примере прототипа ToLookup вместо типичного массива cars, который постоянно применялся, необходим класс с элементами, содержащими члены, которые могут быть использованы в качестве ключей, но не являются уникальными. Для этой цели задействован общий класс Actor (../level2/2_0.php).

Ниже содержится пример вызова операции ToLookup с использованием класса Actor:

C#

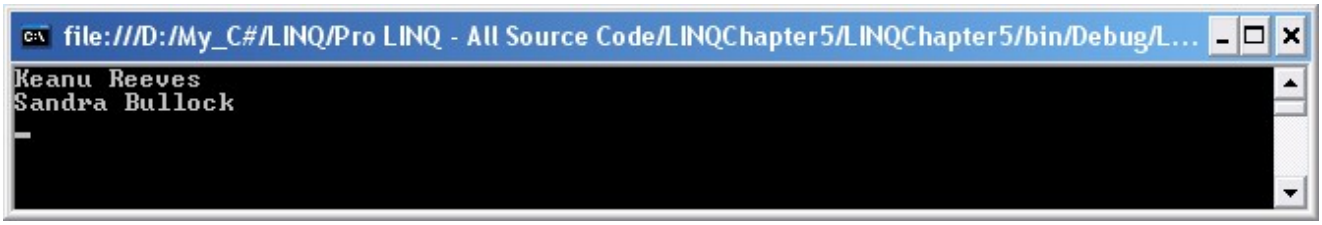
```
ILookup<int, Actor> lookup = Actor.GetActors().ToLookup(k => k.birthYear);

      // Посмотрим, можно ли найти кого-то, рожденного в 1964 г.
      IEnumerable<Actor> actors = lookup[1964];
      foreach (var actor in actors)
        Console.WriteLine("{0} {1}", actor.firstName, actor.lastName);
```

Сначала создается Lookup с указанием члена Actor.birthYear в качестве ключа к Lookup. Затем производится доступ к нему по индексу с ключом 1964. Затем возвращенные значения перечисляются. Ниже показан результат:

Пройди тесты

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

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




Неожиданно был получен множественный результат. Хорошо, что входная последовательность была преобразована в Lookup вместо Dictionary, т.к. имеется несколько элементов с одинаковым ключом.

Для построения примера, демонстрирующего работу второго прототипа ToLookup, в общий класс Actor вносится небольшая модификация. Будет создан класс Actor2, во всем идентичный Actor, за исключением того, что член birthYear теперь имеет тип string вместо int:

C#

```
public class Actor2
  {
    public string birthYear;
    public string firstName;
    public string lastName;

    public static Actor2[] GetActors()
    {
      Actor2[] actors = new Actor2[] {
        new Actor2 { birthYear = "1964", firstName = "Keanu", lastName = "Reeves" },
        new Actor2 { birthYear = "1968", firstName = "Owen", lastName = "Wilson" },
        new Actor2 { birthYear = "1960", firstName = "James", lastName = "Spader" },
        // Пример даты с ведущим нулем
        new Actor2 { birthYear = "01964", firstName = "Sandra",
          lastName = "Bullock" },
      };

      return (actors);
    }
  }
```

Обратите внимание, что в этом классе тип члена birthYear изменен на string. Теперь операция ToLookup будет вызываться, как показано ниже:

Пройди тесты

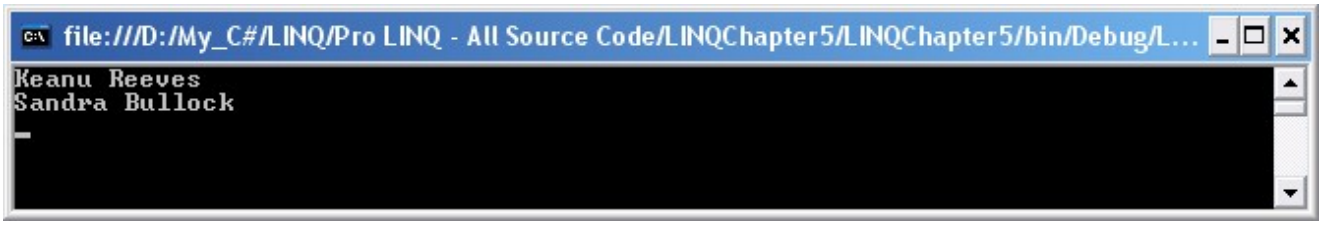
C#

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

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

```
ILookup<string, Actor2> lookup = Actor2.GetActors()
        .ToLookup(k => k.birthYear, new MyStringifiedNumberComparer());

IEnumerable<Actor2> actors = lookup["0001964"];
foreach (var actor in actors)
      Console.WriteLine("{0} {1}", actor.firstName, actor.lastName);

...

public class MyStringifiedNumberComparer : IEqualityComparer<string>
  {
    public bool Equals(string x, string y)
    {
      return (Int32.Parse(x) == Int32.Parse(y));
    }

    public int GetHashCode(string obj)
    {
      return Int32.Parse(obj).ToString().GetHashCode();
    }
  }
```

Здесь используется тот же самый объект проверки эквивалентности, что и в примере Dictionary. В этом случае входная последовательность преобразуется в Lookup, с предоставлением объекта проверки эквивалентности, поскольку известно, что ключ, хранимый в виде string, может иногда содержать ведущие нули. Этот объект знает, как с этим справиться. Результат получается таким же, как и в первом примере:




Обратите внимание на попытку извлечь все элементы со значением ключа "0001964". В результате получаются элементы с ключами "1964" и "01964". Значит, объект проверки эквивалентности работает.

В примере с третьим прототипом операции ToLookup применяется тот же самый класс Actor, что и в примере кода первого прототипа для ToLookup. Ниже показан необходимый код:

Пройди тесты C#

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

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

```
ILookup<int, string> lookup = Actor.GetActors()
        .ToLookup(k => k.birthYear,
                  a => string.Format("{0} {1}", a.firstName, a.lastName));

IEnumerable<string> actors = lookup[1964];
foreach (var actor in actors)
        Console.WriteLine("{0}", actor);
```

Использование варианта операции ToLookup с elementSelector позволяет сохранять в Lookup данные другого типа, отличного от типа данных элемента входной последовательности.

При написании примера для четвертого прототипа ToLookup используется класс Actor2 и общий класс MyStringifiedNumberComparer. Код примера показан ниже:

C#

```
ILookup<string, string> lookup = Actor2.GetActors()
        .ToLookup(k => k.birthYear,
                  a => string.Format("{0} {1}", a.firstName, a.lastName),
                  new MyStringifiedNumberComparer());

IEnumerable<string> actors = lookup["0001964"];
foreach (var actor in actors)
        Console.WriteLine("{0}", actor);
```

Как видите, здесь производится обращение к Lookup по индексу с указанием значения ключа, отличного от любого из значений ключей извлеченных значений, поэтому можно сделать вывод, что объект проверки эквивалентности работает. Вместо сохранения всего объекта Actor2 сохраняется просто интересующая строка.


| Назад (3_2.php) | 2 | 3 | 4 | Вперед (3_4.php) |
|---|---|---|---|---|