ОГБПОУ «ТОМСКИЙ ТЕХНИКУМ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ»

Утверждаю

Заместитель директора по УМР

\_\_\_\_\_\_\_\_\_\_\_\_/Е.А. Родзик

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Методические рекомендации по выполнению   
учебной-практической работы №7

учебной практики   
*УП 05. Программирование*

*Тема: «LINQ»*

г. Томск – 2020 г

РАССМОТРЕННО

на заседании ПЦК

«Информационные системы и программирование»

протокол №\_\_\_\_\_\_

от «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Председатель ПЦК

\_\_\_\_\_\_\_\_\_\_\_\_\_\_/ Фунтиков М.Н.Рекомендации разработаны на основе Федерального государственного образовательного стандарта (далее – ФГОС) по специальностям среднего профессионального образования (далее СПО) 09.02.07 – «Информационные системы и программирование» и в соответствии с примерной основной образовательной программой.

Организация-разработчик:

ОГБОУ СПО «Томский техникум информационных технологий»

Разработчики:

\_\_\_\_\_\_\_\_\_\_\_\_\_/ Сидиков И.Д.

\_\_\_\_\_\_\_\_\_\_\_\_\_/ Владимировна А.В.

**СОДЕРЖАНИЕ**

[1. Общая характеристика учебной-практической работы №7 4](#_Toc22557890)

[2. Краткие теоретические сведения 5](#_Toc22557891)

[3. Задания для выполнения 25](#_Toc22557892)

[4. Индивидуальные задания 26](#_Toc22557893)

[5. Контрольные вопросы 27](#_Toc22557894)

[6. Рекомендованная литература 28](#_Toc22557895)

[ПРИЛОЖЕНИЕ А 29](#_Toc22557896)

[ПРИЛОЖЕНИЕ Б 30](#_Toc22557897)

# ОБЩАЯ ХАРАКТЕРИСТИКА УЧЕБНОЙ-ПРАКТИЧЕСКОЙ РАБОТЫ №7

**Тема:** «LINQ».

**Цель работы:** получение первоначальных навыков по работе c сортировке, группировки и фильтрации данных на языке программирования C#

**Проверяемые компетенции:**

ОК 1, ОК 2, ОК 4, ОК 9, ПК11.2.

**Инструкция по выполнению:**

1. Ознакомьтесь с теоретическим материалом, необходимым для выполнения практической работы.
2. Выполните предложенное практическое задание.
3. Оформите отчет по учебной практике (шаблон представлен в приложении)
4. Отправьте отчет на проверку.

**Время выполнения заданий:** 6 часов.

**Критерии оценки:**

|  |  |  |
| --- | --- | --- |
| **№** | **Критерий** | **Баллы** |
| 1 | Отчет соответствует предложенному шаблону | 1 |
| 2 | Задание выполнено правильно | 2 |
| 3 | В коде присутствуют делегаты и события | 4 |
| 4 | Составлена блок схема к программе | 2 |
| 5 | Программа работает правильно без сбоев | 2 |
| 6 | Студент ответил на контрольные вопросы по лабораторной работе | 2 |
| 7 | Оформление отчета соответствует требованиям (шрифт, поля, отступы, интервалы, оформление рисунков, автоматическое оглавление) | 1 |
| 8 | Своевременность выполнения задания | 1 |
| Итого | | 15 |

**Перевод в пятибалльную систему оценивания:**

«отлично» - 13 - 15 баллов

«хорошо» - 10 -12 баллов

«удовлетворительно» - 7-9

«неудовлетворительно» <7 баллов

# КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

## Введение в LINQ

LINQ (Language-Integrated Query) представляет простой и удобный язык запросов к источнику данных. В качестве источника данных может выступать объект, реализующий интерфейс IEnumerable (например, стандартные коллекции, массивы), набор данных DataSet, документ XML. Но вне зависимости от типа источника LINQ позволяет применить ко всем один и тот же подход для выборки данных.

Существует несколько разновидностей LINQ:

* LINQ to Objects: применяется для работы с массивами и коллекциями
* LINQ to Entities: используется при обращении к базам данных через технологию Entity Framework
* LINQ to Sql: технология доступа к данным в MS SQL Server
* LINQ to XML: применяется при работе с файлами XML
* LINQ to DataSet: применяется при работе с объектом DataSet
* Parallel LINQ (PLINQ): используется для выполнения параллельных запросов

## Основы LINQ

В чем же удобство LINQ? Посмотрим на простейшем примере. Выберем из массива строки, начинающиеся на определенную букву, и отсортируем полученный список:

|  |
| --- |
| С# |
| string[] teams = {"Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона"};    var selectedTeams = new List<string>();  foreach(string s in teams)  {      if (s.ToUpper().StartsWith("Б"))          selectedTeams.Add(s);  }  selectedTeams.Sort();    foreach (string s in selectedTeams)      Console.WriteLine(s); |

Теперь проведем те же действия с помощью LINQ:

|  |
| --- |
| С# |
| string[] teams = {"Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона"};    var selectedTeams = from t in teams // определяем каждый объект из teams как t                      where t.ToUpper().StartsWith("Б") //фильтрация по критерию                      orderby t  // упорядочиваем по возрастанию                      select t; // выбираем объект    foreach (string s in selectedTeams)      Console.WriteLine(s); |

Чтобы использовать функциональность LINQ, убедимся, что в файле подключено пространство имен System.LINQ.

Итак, код стал меньше и проще. В принципе все выражение можно было бы записать в одну строку: var selectedTeams = from t in teams where t.ToUpper().StartsWith("Б") orderby t select t. Но для более понятной логической разбивки я поместил каждое отдельное подвыражение на отдельной строке.

Простейшее определение запроса LINQ выглядит следующим образом:

|  |
| --- |
| С# |
| from переменная in набор\_объектов  select переменная; |

Итак, что делает этот запрос LINQ? Выражение from t in teams проходит по всем элементам массива teams и определяет каждый элемент как t. Используя переменную t мы можем проводить над ней разные операции.

Несмотря на то, что мы не указываем тип переменной t, выражения LINQ являются строго типизированными. То есть среда автоматически распознает, что набор teams состоит из объектов string, поэтому переменная t будет рассматриваться в качестве строки.

Далее с помощью оператора where проводится фильтрация объектов, и если объект соответствует критерию (в данном случае начальная буква должна быть "Б"), то этот объект передается дальше.

Оператор orderby упорядочивает по возрастанию, то есть сортирует выбранные объекты.

Оператор select передает выбранные значения в результирующую выборку, которая возвращается LINQ-выражением.

В данном случае результатом выражения LINQ является объект IEnumerable<T>. Нередко результирующая выборка определяется с помощью ключевого слова var, тогда компилятор на этапе компиляции сам выводит тип.

Преимуществом подобных запросов также является и то, что они интуитивно похожи на запросы языка SQL, хотя и имеют некоторые отличия

### Методы расширения LINQ

Кроме стандартного синтаксиса from .. in .. select для создания запроса LINQ мы можем применять специальные методы расширения, которые определены для интерфейса IEnumerable. Как правило, эти методы реализуют ту же функциональность, что и операторы LINQ типа where или orderby.

Например:

|  |
| --- |
| С# |
| string[] teams = { "Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона" };    var selectedTeams = teams.Where(t=>t.ToUpper().StartsWith("Б")).OrderBy(t => t);    foreach (string s in selectedTeams)      Console.WriteLine(s); |

Запрос teams.Where(t=>t.ToUpper().StartsWith("Б")).OrderBy(t => t) будет аналогичен предыдущему. Он состоит из цепочки методов Where и OrderBy. В качестве аргумента эти методы принимают делегат или лямбда-выражение.

Не каждый метод расширения имеет аналог среди операторов LINQ, но в этом случае можно сочетать оба подхода. Например, используем стандартный синтаксис linq и метод расширения Count(), возвращающий количество элементов в выборке:

|  |
| --- |
| С# |
| int number = (from t in teams where t.ToUpper().StartsWith("Б") select t).Count(); |

### Список используемых методов расширения LINQ

* Select: определяет проекцию выбранных значений
* Where: определяет фильтр выборки
* OrderBy: упорядочивает элементы по возрастанию
* OrderByDescending: упорядочивает элементы по убыванию
* ThenBy: задает дополнительные критерии для упорядочивания элементов возрастанию
* ThenByDescending: задает дополнительные критерии для упорядочивания элементов по убыванию
* Join: соединяет две коллекции по определенному признаку
* GroupBy: группирует элементы по ключу
* ToLookup: группирует элементы по ключу, при этом все элементы добавляются в словарь
* GroupJoin: выполняет одновременно соединение коллекций и группировку элементов по ключу
* Reverse: располагает элементы в обратном порядке
* All: определяет, все ли элементы коллекции удовлетворяют определенному условию
* Any: определяет, удовлетворяет хотя бы один элемент коллекции определенному условию
* Contains: определяет, содержит ли коллекция определенный элемент
* Distinct: удаляет дублирующийся элементы из коллекции
* Except: возвращает разность двух коллекцию, то есть те элементы, которые создаются только в одной коллекции
* Union: объединяет две однородные коллекции
* Intersect: возвращает пересечение двух коллекций, то есть те элементы, которые встречаются в обоих коллекциях
* Count: подсчитывает количество элементов коллекции, которые удовлетворяют определенному условию
* Sum: подсчитывает сумму числовых значений в коллекции
* Average: подсчитывает cреднее значение числовых значений в коллекции
* Min: находит минимальное значение
* Max: находит максимальное значение
* Take: выбирает определенное количество элементов
* Skip: пропускает определенное количество элементов
* TakeWhile: возвращает цепочку элементов последовательности, до тех пор, пока условие истинно
* SkipWhile: пропускает элементы в последовательности, пока они удовлетворяют заданному условию, и затем возвращает оставшиеся элементы
* Concat: объединяет две коллекции
* Zip: объединяет две коллекции в соответствии с определенным условием
* First: выбирает первый элемент коллекции
* FirstOrDefault: выбирает первый элемент коллекции или возвращает значение по умолчанию
* Single: выбирает единственный элемент коллекции, если коллекция содержит больше или меньше одного элемента, то генерируется исключение
* SingleOrDefault: выбирает первый элемент коллекции или возвращает значение по умолчанию
* ElementAt: выбирает элемент последовательности по определенному индексу
* ElementAtOrDefault: выбирает элемент коллекции по определенному индексу или возвращает значение по умолчанию, если индекс вне допустимого диапазона
* Last: выбирает последний элемент коллекции
* LastOrDefault: выбирает последний элемент коллекции или возвращает значение по умолчанию

## Фильтрация выборки и проекция

### Фильтрация

Для выбора элементов из некоторого набора по условию используется метод Where. Например, выберем все четные элементы, которые больше 10.

Фильтрация с помощью операторов LINQ:

|  |
| --- |
| С# |
| int[] numbers = { 1, 2, 3, 4, 10, 34, 55, 66, 77, 88 };  IEnumerable<int> evens = from i in numbers                                 where i%2==0 && i>10                                 select i;  foreach (int i in evens)      Console.WriteLine(i); |

Здесь используется конструкция from: from i in numbers

Тот же запрос с помощью метода расширения:

|  |
| --- |
| С# |
| int[] numbers = { 1, 2, 3, 4, 10, 34, 55, 66, 77, 88 };  IEnumerable<int> evens = numbers.Where(i => i % 2 == 0 && i > 10); |

Если выражение в методе Where для определенного элемента будет равно true (в данном случае выражение i % 2 == 0 && i > 10), то данный элемент попадает в результирующую выборку.

### Выборка сложных объектов

Допустим, у нас есть класс пользователя:

|  |
| --- |
| С# |
| class User  {  public string Name { get;set; }  public int Age { get; set; }  public List<string> Languages { get; set; }  public User()  {  Languages = new List<string>();  }  } |

Создадим набор пользователей и выберем из них тех, которым больше 25 лет:

|  |
| --- |
| С# |
| List<User> users = new List<User>  {  new User {Name="Том", Age=23, Languages = new List<string> {"английский", "немецкий" }},  new User {Name="Боб", Age=27, Languages = new List<string> {"английский", "французский" }},  new User {Name="Джон", Age=29, Languages = new List<string> {"английский", "испанский" }},  new User {Name="Элис", Age=24, Languages = new List<string> {"испанский", "немецкий" }}  };    var selectedUsers = from user in users  where user.Age > 25  select user;  foreach (User user in selectedUsers)  Console.WriteLine($"{user.Name} - {user.Age}"); |

|  |
| --- |
| С# - консоль |
| Боб - 27  Джон - 29 |

Аналогичный запрос с помощью метода расширения Where:

|  |
| --- |
| С# |
| var selectedUsers = users.Where(u=> u.Age > 25); |

### Сложные фильтры

Теперь рассмотрим более сложные фильтры. Например, в классе пользователя есть список языков, которыми владеет пользователь. Что если нам надо отфильтровать пользователей по языку:

|  |
| --- |
| С# |
| var selectedUsers = from user in users                      from lang in user.Languages                      where user.Age < 28                      where lang == "английский"                      select user; |

Результат:

|  |
| --- |
| С# - консоль |
| Том - 23  Боб - 27 |

Для создания аналогичного запроса с помощью методов расширения применяется метод SelectMany:

|  |
| --- |
| С# |
| var selectedUsers = users.SelectMany(u => u.Languages,                              (u, l) => new { User = u, Lang = l })                            .Where(u => u.Lang == "английский" && u.User.Age < 28)                            .Select(u=>u.User); |

Метод SelectMany() в качестве первого параметра принимает последовательность, которую надо проецировать, а в качестве второго параметра - функцию преобразования, которая применяется к каждому элементу. На выходе она возвращает 8 пар "пользователь - язык" (new { User = u, Lang = l }), к которым потом применяетс фильтр с помощью Where.

### Проекция

Проекция позволяет спроектировать из текущего типа выборки какой-то другой тип. Для проекции используется оператор select. Допустим, у нас есть набор объектов следующего класса, представляющего пользователя:

|  |
| --- |
| С# |
| class User  {      public string Name { get;set; }      public int Age { get; set; }  } |

Но нам нужен не весь объект, а только его свойство Name:

|  |
| --- |
| С# |
| List<User> users = new List<User>();  users.Add(new User { Name = "Sam", Age = 43 });  users.Add(new User { Name = "Tom", Age = 33 });    var names = from u in users select u.Name;    foreach (string n in names)       Console.WriteLine(n); |

Результат выражения LINQ будет представлять набор строк, поскольку выражение select u.Name выбирают в результирующую выборку только значения свойства Name.

Аналогично можно создать объекты другого типа, в том числе анонимного:

|  |
| --- |
| С# |
| List<User> users = new List<User>();  users.Add(new User { Name = "Sam", Age = 43 });  users.Add(new User { Name = "Tom", Age = 33 });    var items = from u in users              select new              {                  FirstName = u.Name,                  DateOfBirth = DateTime.Now.Year - u.Age              };    foreach (var n in items)      Console.WriteLine($"{n.FirstName} - {n.DateOfBirth}"); |

Здесь оператор select создает объект анонимного типа, используя текущий объект User. И теперь результат будет содержать набор объектов данного анонимного типа, в котором определены два свойства: FirstName и DateOfBirth.

В качестве альтернативы мы могли бы использовать метод расширения Select():

|  |
| --- |
| С# |
| // выборка имен  var names = users.Select(u => u.Name);    // выборка объектов анонимного типа  var items = users.Select(u => new  {  FirstName = u.Name,  DateOfBirth = DateTime.Now.Year - u.Age  }); |

### Переменные в запросах и оператор let

Иногда возникает необходимость произвести в запросах LINQ какие-то дополнительные промежуточные вычисления. Для этих целей мы можем задать в запросах свои переменные с помощью оператора let:

|  |
| --- |
| С# |
| List<User> users = new List<User>()  {  new User { Name = "Sam", Age = 43 },  new User { Name = "Tom", Age = 33 }  };    var people = from u in users  let name = "Mr. " + u.Name  select new  {  Name = name,  Age = u.Age  }; |

В данном случае создается переменная name, значение которой равно "Mr. " + u.Name.

Возможность определения переменных, наверное, одно из главных преимуществ операторов LINQ по сравнению с методами расширения.

### Выборка из нескольких источников

В LINQ можно выбирать объекты не только из одного, но и из большего количества источников:

Например, возьмем классы:

|  |
| --- |
| С# |
| class Phone  {  public string Name { get; set; }  public string Company { get; set; }  }  class User  {  public string Name { get; set; }  public int Age { get; set; }  } |

Создадим два разных источника данных и произведем выборку:

|  |
| --- |
| С# |
| List<User> users = new List<User>()  {  new User { Name = "Sam", Age = 43 },  new User { Name = "Tom", Age = 33 }  };  List<Phone> phones = new List<Phone>()  {  new Phone {Name="Lumia 630", Company="Microsoft" },  new Phone {Name="iPhone 6", Company="Apple"},  };    var people = from user in users  from phone in phones  select new { Name = user.Name, Phone = phone.Name };    foreach (var p in people)  Console.WriteLine($"{p.Name} - {p.Phone}"); |

|  |
| --- |
| С# - консоль |
| Sam - Lumia 630  Sam - iPhone 6  Tom - Lumia 630  Tom - iPhone 6 |

Таким образом, при выборке из двух источников каждый элемент из первого источника будет сопоставляться с каждым элементом из второго источника. То есть получиться 4 пары.

## Сортировка

Для сортировки набора данных по возрастанию используется оператор orderby:

|  |
| --- |
| С# |
| int[] numbers = { 3, 12, 4, 10, 34, 20, 55, -66, 77, 88, 4 };  var orderedNumbers = from i in numbers                       orderby i                       select i;  foreach (int i in orderedNumbers)      Console.WriteLine(i); |

Оператор orderby принимает критерий сортировки. В данном случае в качестве критерия выступает само число.

Возьмем посложнее пример. Допустим, надо отсортировать выборку сложных объектов. Тогда в качестве критерия мы можем указать свойство класса объекта:

|  |
| --- |
| С# |
| List<User> users = new List<User>()  {      new User { Name = "Tom", Age = 33 },      new User { Name = "Bob", Age = 30 },      new User { Name = "Tom", Age = 21 },      new User { Name = "Sam", Age = 43 }  };    var sortedUsers = from u in users                    orderby u.Name                    select u;    foreach (User u in sortedUsers)      Console.WriteLine(u.Name); |

По умолчанию оператор orderby производит сортировку по возрастанию. Однако с помощью ключевых слов ascending (сортировка по возрастанию) и descending (сортировка по убыванию) можно явным образом указать направление сортировки:

|  |
| --- |
| С# |
| var sortedUsers = from u in users                    orderby u.Name descending                    select u; |

Вместо оператора orderby можно использовать методы расширения OrderBy:

|  |
| --- |
| С# |
| int[] numbers = { 3, 12, 4, 10, 34, 20, 55, -66, 77, 88, 4 };  IEnumerable<int> sortedNumbers = numbers.OrderBy(i=>i);    List<User> users = new List<User>()  {      new User { Name = "Tom", Age = 33 },      new User { Name = "Bob", Age = 30 },      new User { Name = "Tom", Age = 21 },      new User { Name = "Sam", Age = 43 }  };  var sortedUsers = users.OrderBy(u=>u.Name); |

Метод OrderBy()сортирует по возрастанию. Для сортировки по убыванию используется метод:

|  |
| --- |
| С# |
| var sortedUsers = users.OrderByDescending(u=>u.Name); |

### Множественные критерии сортировки

В наборах сложных объектов иногда встает ситуация, когда надо отсортировать не по одному, а сразу по нескольким полям. Для этого в запросе LINQ все критерии указываются в порядке приоритета через запятую:

|  |
| --- |
| С# |
| List<User> users = new List<User>()  {  new User { Name = "Tom", Age = 33 },  new User { Name = "Bob", Age = 30 },  new User { Name = "Tom", Age = 21 },  new User { Name = "Sam", Age = 43 }  };  var result = from user in users  orderby user.Name, user.Age  select user;  foreach (User u in result)  Console.WriteLine($"{u.Name} - {u.Age}"); |

|  |
| --- |
| С# - консоль |
| Alice - 28  Bob - 30  Sam - 43  Tom - 21  Tom - 33 |

С помощью методов расширения то же самое можно сделать через метод ThenBy()(для сортировки по возрастанию) и ThenByDescending() (для сортировки по убыванию):

|  |
| --- |
| С# |
| var result = users.OrderBy(u => u.Name).ThenBy(u => u.Age); |

Результат будет аналогичен предыдущему.

## Объединение, пересечение и разность коллекций

Кроме методов выборки LINQ имеет несколько методов, который позволяют из двух последовательностей объектов сгенерировать множество (то есть некий набор уникальных элементов): разность, объединение и пересечение.

### Разность последовательностей

С помощью метода Except можно получить разность двух последовательностей:

|  |
| --- |
| С# |
| string[] soft = { "Microsoft", "Google", "Apple"};  string[] hard = { "Apple", "IBM", "Samsung"};    // разность последовательностей  var result = soft.Except(hard);    foreach (string s in result)      Console.WriteLine(s); |

В данном случае из массива soft убираются все элементы, которые есть в массиве hard. Результатом операции будут два элемента:

|  |
| --- |
| С# - консоль |
| Microsoft  Google |

### Пересечение последовательностей

Для получения пересечения последовательностей, то есть общих для обоих наборов элементов, применяется метод Intersect:

|  |
| --- |
| С# |
| string[] soft = { "Microsoft", "Google", "Apple"};  string[] hard = { "Apple", "IBM", "Samsung"};    // пересечение последовательностей  var result = soft.Intersect(hard);    foreach (string s in result)      Console.WriteLine(s); |

Так как оба набора имеют только один общий элемент, то соответственно только он и попадет в результирующую выборку:

|  |
| --- |
| С# - консоль |
| Apple |

### Объединение последовательностей

Для объединения двух последовательностей используется метод Union. Его результатом является новый набор, в котором имеются элементы, как из первой, так и из второй последовательности. Повторяющиеся элементы добавляются в результат только один раз:

|  |
| --- |
| С# |
| string[] soft = { "Microsoft", "Google", "Apple"};  string[] hard = { "Apple", "IBM", "Samsung"};    // объединение последовательностей  var result = soft.Union(hard);    foreach (string s in result)  Console.WriteLine(s); |

|  |
| --- |
| С# - консоль |
| Microsoft  Google  Apple  IBM  Samsung |

Если же нам нужно простое объединение двух наборов, то мы можем использовать метод Concat:

|  |
| --- |
| С# |
| var result = soft.Concat(hard); |

Те элементы, которые встречаются в обоих наборах, дублируются.

### Удаление дубликатов

Для удаления дублей в наборе используется метод Distinct:

|  |
| --- |
| С# |
| var result = soft.Concat(hard).Distinct(); |

Последовательное применение методов Concat и Distinct будет подобно действию метода Union.

## Агрегатные операции

К агрегатным операциям относят различные операции над выборкой, например, получение числа элементов, получение минимального, максимального и среднего значения в выборке, а также суммирование значений.

### Метод Aggregate

Метод Aggregate выполняет общую агрегацию элементов коллекции в зависимости от указанного выражения. Например:

|  |
| --- |
| С# |
| int[] numbers = { 1, 2, 3, 4, 5};    int query = numbers.Aggregate((x,y)=> x - y); |

Переменная query будет представлять результат агрегации массива. В качестве условия агрегации используется выражение (x,y)=> x - y, то есть вначале из первого элемента вычитается второй, потом из получившегося значения вычитается третий и так далее. То есть будет эквивалентно выражению:

|  |
| --- |
| С# |
| int query = 1 - 2 - 3 - 4 - 5 |

В итоге мы получим число -13. Соответственно мы бы могли использовать любые другие операции, например, сложение:

|  |
| --- |
| С# |
| int query = numbers.Aggregate((x,y)=> x + y); // аналогично 1 + 2 + 3 + 4 + 5 |

### Получение размера выборки. Метод Count

Для получения числа элементов в выборке используется метод Count():

|  |
| --- |
| С# |
| int[] numbers = { 1, 2, 3, 4, 10, 34, 55, 66, 77, 88 };  int size = (from i in numbers where i % 2 == 0 && i > 10 select i).Count();  Console.WriteLine(size); |

Метод Count() в одной из версий также может принимать лямбда-выражение, которое устанавливает условие выборки. Поэтому мы можем в данном случае не использовать выражение Where:

|  |
| --- |
| С# |
| int size = numbers.Count(i => i % 2 == 0 && i > 10);  Console.WriteLine(size); |

### Получение суммы

Для получения суммы значений применяется метод Sum:

|  |
| --- |
| С# |
| int[] numbers = { 1, 2, 3, 4, 10, 34, 55, 66, 77, 88 };  List<User> users = new List<User>()  {      new User { Name = "Tom", Age = 23 },      new User { Name = "Sam", Age = 43 },      new User { Name = "Bill", Age = 35 }  };    int sum1 = numbers.Sum();  decimal sum2 = users.Sum(n => n.Age); |

Метод Sum() имеет ряд перегрузок. В частности, если у нас набор сложных объектов, как в примере выше, то мы можем указать свойство, значения которого будут суммироваться: users.Sum(n => n.Age)

### Максимальное, минимальное и среднее значения

Для нахождения минимального значения применяется метод Min(), для получения максимального - метод Max(), а для нахождения среднего значения - метод Average(). Их действие похоже на методы Sum и Count:

|  |
| --- |
| С# |
| int[] numbers = { 1, 2, 3, 4, 10, 34, 55, 66, 77, 88 };  List<User> users = new List<User>()  {      new User { Name = "Tom", Age = 23 },      new User { Name = "Sam", Age = 43 },      new User { Name = "Bill", Age = 35 }  };    int min1 = numbers.Min();  int min2 = users.Min(n => n.Age); // минимальный возраст    int max1 = numbers.Max();  int max2 = users.Max(n => n.Age); // максимальный возраст    double avr1 = numbers.Average();  double avr2 = users.Average(n => n.Age); //средний возраст |

## Методы Skip и Take

Метод Skip() пропускает определенное количество элементов, а метод Take() извлекает определенное число элементов. Нередко данные методы применяются вместе для создания постраничного вывода.

Извлечем три первых элемента:

|  |
| --- |
| С# |
| int[] numbers = { -3, -2, -1, 0, 1, 2, 3 };  var result = numbers.Take(3);    foreach (int i in result)      Console.WriteLine(i); |

Выберем все элементы, кроме первых трех:

|  |
| --- |
| С# |
| var result = numbers.Skip(3); |

Совмещая оба метода, мы можем выбрать определенное количество элементов начиная с определенного элемента. Например, выберем три элемента, начиная с пятого (то есть пропустив четыре элемента):

|  |
| --- |
| С# |
| int[] numbers = { -3, -2, -1, 0, 1, 2, 3 };  var result = numbers.Skip(4).Take(3);    foreach (int i in result)      Console.WriteLine(i); |

Похожим образом работают методы TakeWhile и SkipWhile.

Метод TakeWhile выбирает цепочку элементов, начиная с первого элемента, пока они удовлетворяют определенному условию. Например:

|  |
| --- |
| С# |
| string[] teams = { "Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона" };  foreach (var t in teams.TakeWhile(x=>x.StartsWith("Б")))      Console.WriteLine(t); |

Согласно условию, мы выбираем те команды, которые начинаются с буквы Б. В массиве есть три таких команды. Однако в цикле будут выведены только две первых:

|  |
| --- |
| С# - консоль |
| Бавария  Боруссия |

Потому что цепочка обрывается на третьей команде - "Реал Мадрид" - она не соответствует условию, и после этого выборка уже не идет.

Если бы первой командой в массиве стояла бы команда, начинающаяся не с буквы Б, например, "Реал Мадрид", то в этом случае метод возвратил бы нам 0 элементов.

В подобном русле действует метод SkipWhile. Он пропускает цепочку элементов, начиная с первого элемента, пока они удовлетворяют определенному условию. Например:

|  |
| --- |
| С# |
| string[] teams = { "Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона" };  foreach (var t in teams.SkipWhile(x=>x.StartsWith("Б")))      Console.WriteLine(t); |

Первые две команды, которые начинаются с буквы Б и соответствуют условию, будут пропущены. На третьей команде цепочка обрывается, поэтому последняя команда, начинающаяся с буквы Б, будет включена в выходной список:

|  |
| --- |
| С# - консоль |
| Реал Мадрид  Манчестер Сити  ПСЖ  Барселона |

И опять же если в массиве первый элемент не начинался бы с буквы Б, то цепочка пропускаемых элементов прервалась бы уже на первом элементе, и поэтому метод SkipWhile возвратил бы все элементы массива.

## Группировка

Для группировки данных по определенным параметрам применяется оператор group by или метод GroupBy(). Допустим, у нас есть набор из объектов следующего типа:

|  |
| --- |
| С# |
| class Phone  {  public string Name { get; set; }  public string Company { get; set; }  } |

Данный класс представляет модель телефона, в которой определены свойства для названия и компании-производителя. Сгруппируем набор телефонов по производителю:

|  |
| --- |
| С# |
| List<Phone> phones = new List<Phone>  {      new Phone {Name="Lumia 430", Company="Microsoft" },      new Phone {Name="Mi 5", Company="Xiaomi" },      new Phone {Name="LG G 3", Company="LG" },      new Phone {Name="iPhone 5", Company="Apple" },      new Phone {Name="Lumia 930", Company="Microsoft" },      new Phone {Name="iPhone 6", Company="Apple" },      new Phone {Name="Lumia 630", Company="Microsoft" },      new Phone {Name="LG G 4", Company="LG" }  };    var phoneGroups = from phone in phones                    group phone by phone.Company;    foreach (IGrouping<string, Phone> g in phoneGroups)  {      Console.WriteLine(g.Key);      foreach (var t in g)          Console.WriteLine(t.Name);      Console.WriteLine();  } |

Если в выражении LINQ последним оператором, выполняющим операции над выборкой, является group, то оператор select не применяется.

Оператор group принимает критерий по которому проводится группировка: group phone by phone.Company - в данном случае группировка по свойству Company.

Результатом оператора group является выборка, которая состоит из групп. Каждая группа представляет объект IGrouping<string, Phone>: параметр string указывает на тип ключа, а параметр Phone - на тип сгруппированных объектов.

Каждая группа имеет ключ, который мы можем получить через свойство Key: g.Key

Все элементы группы можно получить с помощью дополнительной итерации. Элементы группы имеют тот же тип, что и тип объектов, которые передавались оператору group, то есть в данном случае объекты типа Phone.

|  |
| --- |
| С# - консоль |
| Microsoft  Lumia 430  Lumia 930  Lumia 630  Xiaomi  Mi 5  LG  LG G 3  LG G4  Apple  iPhone 5  iPhone 6 |

Аналогичный запрос можно построить с помощью метода расширения GroupBy:

|  |
| --- |
| С# |
| var phoneGroups = phones.GroupBy(p => p.Company); |

Теперь изменим запрос и получим команду и создадим из группы новый объект:

|  |
| --- |
| С# |
| var phoneGroups2 = from phone in phones                     group phone by phone.Company into g                     select new { Name = g.Key, Count = g.Count() };  foreach (var group in phoneGroups2)      Console.WriteLine($"{group.Name} : {group.Count}"); |

Выражение group phone by phone.Company into g определяет переменную g, которая будет содержать группу. С помощью этой переменной мы можем затем создать новый объект анонимного типа: select new { Name = g.Key, Count = g.Count() } Теперь результат запроса LINQ будет представлять набор объектов таких анонимных типов, у которых два свойства Name и Count.

|  |
| --- |
| С# - консоль |
| Microsoft : 3  Xiaomi : 1  LG : 2  Apple : 2 |

Аналогичная операция с помощью метода GroupBy():

|  |
| --- |
| С# |
| var phoneGroups = phones.GroupBy(p => p.Company)                          .Select(g => new { Name = g.Key, Count = g.Count() }); |

Также мы можем осуществлять вложенные запросы:

|  |
| --- |
| С# |
| var phoneGroups2 = from phone in phones                     group phone by phone.Company into g                     select new                     {                          Name = g.Key,                          Count = g.Count(),                          Phones = from p in g select p                     };  foreach (var group in phoneGroups2)  {      Console.WriteLine($"{group.Name} : {group.Count}");      foreach(Phone phone in group.Phones)          Console.WriteLine(phone.Name);      Console.WriteLine();  } |

Здесь свойство Phones каждой группы формируется с помощью дополнительного запроса, выбирающего все телефоны в этой группе.

|  |
| --- |
| С# - консоль |
| Microsoft : 3  Lumia 430  Lumia 930  Lumia 630  Xiaomi : 1  Mi 5  LG : 2  LG G 3  LG G4  Apple : 2  iPhone 5  iPhone 6 |

Аналогичный запрос с помощью метода GroupBy:

|  |
| --- |
| С# |
| var phoneGroups = phones.GroupBy(p => p.Company)                          .Select(g => new                          {                              Name = g.Key,                              Count = g.Count(),                              Phones = g.Select(p =>p)                          }); |

## Соединение коллекций. Метод Join, GroupJoin и Zip

Соединение в LINQ используется для объединения двух разнотипных наборов в один. Для соединения используется оператор join или метод Join(). Как правило, данная операция применяется к двум наборам, которые имеют один общий критерий. Например, у нас есть два класса:

|  |
| --- |
| С# |
| class Player  {      public string Name { get; set; }      public string Team { get; set; }  }  class Team  {      public string Name { get; set; }      public string Country { get; set; }  } |

Объекты обоих классов будет иметь один общий критерий - название команды. Соединим по этому критерию два набора этих классов:

|  |
| --- |
| С# |
| List<Team> teams = new List<Team>()  {      new Team { Name = "Бавария", Country ="Германия" },      new Team { Name = "Барселона", Country ="Испания" }  };  List<Player> players = new List<Player>()  {      new Player {Name="Месси", Team="Барселона"},      new Player {Name="Неймар", Team="Барселона"},      new Player {Name="Роббен", Team="Бавария"}  };    var result = from pl in players               join t in teams on pl.Team equals t.Name               select new { Name = pl.Name, Team = pl.Team, Country = t.Country };    foreach (var item in result)      Console.WriteLine($"{item.Name} - {item.Team} ({item.Country})"); |

С помощью выражения join t in teams on pl.Team equals t.Name объект pl из списка players соединяется с объектом t из списка teams, если значение свойства pl.Team совпадает со значением свойства t.Name. Результатом соединения будет объект анонимного типа, который будет содержать три свойства. В итоге мы получим следующий вывод:

|  |
| --- |
| С# - консоль |
| Месси - Барселона (Испания)  Неймар - Барселона (Испания)  Роббен - Бавария (Германия) |

То же самое действие можно было бы выполнить с помощью метода Join():

|  |
| --- |
| С# |
| var result = players.Join(teams, // второй набор               p => p.Team, // свойство-селектор объекта из первого набора               t => t.Name, // свойство-селектор объекта из второго набора               (p, t) => new { Name = p.Name, Team = p.Team, Country = t.Country }); // результат |

Метод Join() принимает четыре параметра:

1. второй список, который соединяем с текущим
2. свойство объекта из текущего списка, по которому идет соединение
3. свойство объекта из второго списка, по которому идет соединение
4. новый объект, который получается в результате соединения

### GroupJoin

Метод GroupJoin кроме соединения последовательностей также выполняет и группировку. Например, возьмем вышеопределенные списки teams и players и сгуппируем всех игроков по командам:

|  |
| --- |
| С# |
| var result2 = teams.GroupJoin(                          players, // второй набор                          t => t.Name, // свойство-селектор объекта из первого набора                          pl => pl.Team, // свойство-селектор объекта из второго набора                          (team, pls) => new  // результирующий объект                          {                              Name = team.Name,                              Country = team.Country,                              Players = pls.Select(p=>p.Name)                          });    foreach (var team in result2)  {      Console.WriteLine(team.Name);      foreach (string player in team.Players)      {          Console.WriteLine(player);      }      Console.WriteLine();  } |

|  |
| --- |
| С# - консоль |
| Бавария  Роббен  Барселона  Месси  Неймар |

Метод GroupJoin, также как и метод Join, принимает все те же параметры. Только теперь во последний параметр - делегат передаются объект команды и набор игроков этой команды.

### Метод Zip

Метод Zip позволяет объединять две последовательности таким образом, что первый элемент из первой последовательности объединяется с первым элементом из второй последовательности, второй элемент из первой последовательности соединяется со вторым элементом из второй последовательности и так далее:

|  |
| --- |
| С# |
| List<Team> teams = new List<Team>()  {      new Team { Name = "Бавария", Country ="Германия" },      new Team { Name = "Барселона", Country ="Испания" },      new Team { Name = "Ювентус", Country ="Италия" }  };  List<Player> players = new List<Player>()  {      new Player {Name="Роббен", Team="Бавария"},      new Player {Name="Неймар", Team="Барселона"},      new Player {Name="Буффон", Team="Ювентус"}  };  var result2 = players.Zip(teams,                            (player, team) => new                             {                                  Name = player.Name,                                  Team = team.Name, Country = team.Country                             });  foreach (var player in result2)  {      Console.WriteLine($"{player.Name} - {player.Team} ({player.Country})");        Console.WriteLine();  } |

Метод Zip в качестве первого параметра принимает вторую последовательность, с которой надо соединяться, а в качестве второго параметра - делегат для создания нового объекта.

|  |
| --- |
| С# - консоль |
| Роббен - Бавария (Германия)  Неймар - Барселона (Испания)  Буффон - Ювентус (Италия) |

## Методы All и Any

Методы All, Any и Contains позволяют определить, соответствует ли коллекция определенному условию, и в зависимости от результата они возвращают true или false.

Метод All проверяет, соответствуют ли все элементы условию. Например, узнаем, у всех ли пользователей возраст превышает 20 и имя начинается с буквы T:

|  |
| --- |
| С# |
| List<Team> teams = new List<Team>()  {      new Team { Name = "Бавария", Country ="Германия" },      new Team { Name = "Барселона", Country ="Испания" },      new Team { Name = "Ювентус", Country ="Италия" }  };  List<Player> players = new List<Player>()  {      new Player {Name="Роббен", Team="Бавария"},      new Player {Name="Неймар", Team="Барселона"},      new Player {Name="Буффон", Team="Ювентус"}  };  var result2 = players.Zip(teams,                            (player, team) => new                             {                                  Name = player.Name,                                  Team = team.Name, Country = team.Country                             });  foreach (var player in result2)  {      Console.WriteLine($"{player.Name} - {player.Team} ({player.Country})");        Console.WriteLine();  } |

|  |
| --- |
| С# - консоль |
| У всех пользователей возраст больше 20  Не у всех пользователей имя начинается с T |

Поскольку у всех пользователей возвраст больше 20, то переменная result1 будет равна true. В то же время не у всех пользователей имя начинаяется с буквы T, поэтому вторая переменная result2 будет равна false.

Метод Any действует подобным образом, только позволяет узнать, соответствует ли хотя бы один элемент коллекции определенному условию:

|  |
| --- |
| С# |
| bool result1 = users.Any(u => u.Age < 20); //false  if (result1)      Console.WriteLine("Есть пользователи с возрастом меньше 20");  else      Console.WriteLine("У всех пользователей возраст больше 20");    bool result2 = users.Any(u => u.Name.StartsWith("T")); //true  if (result2)      Console.WriteLine("Есть пользователи, у которых имя начинается с T");  else      Console.WriteLine("Отсутствуют пользователи, у которых имя начинается с T"); |

|  |
| --- |
| С# - консоль |
| У всех пользователей возраст больше 20  Есть пользователи, у которых имя начинается с T |

Первое выражение вернет false, поскольку у всех пользователей возвраст больше 20. Второе выражение возвратит true, так как у нас есть в коллекции пользователь с именем Tom.

# ЗАДАНИЯ ДЛЯ ВЫПОЛНЕНИЯ

1. Решите задание используя LINQ
2. Спроектируйте блок схему к программе
3. Отобразить в отчете проделанные шаги
4. Ответить на контрольные вопросы
5. Написать вывод о проделанной работе

# ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ

|  |
| --- |
| Вариант 1 – Дано целочисленная последовательность. Найти количество ее положительных двузначных элементов, а также их среднее арифметическое (как вещественное число). Если требуемые элементы отсутствуют, то дважды вывести 0 (первый раз как целое, второй — как вещественное). |
| Вариант 2 – Дано целое число L (> 0) и строковая последовательность A. Строки последовательности A содержат только заглавные буквы латинского алфавита. Среди всех строк из A, имеющих длину L, найти наибольшую (в смысле лексикографического порядка). Вывести эту строку или пустую строку, если последовательность не содержит строк длины L. |
| Вариант 3 – Даны целые числа K1 и K2 и целочисленные последовательности A и B. Получить последовательность, содержащую все числа из A, большие K1, и все числа из B, меньшие K2. Отсортировать полученную последовательность по возрастанию. |
| Вариант 4 - Даны строковые последовательности A, B и С; все строки в каждой последовательности различны, имеют строковой последовательности в лексикографическом порядке по возрастанию. |
| Вариант 5 – Дано целочисленная последовательность. Среди всех элементов последовательности, оканчивающихся одной и той же цифрой, выбрать максимальный. Полученную последовательность максимальных элементов упорядочить по возрастанию их последних цифр. |
| Вариант 6 - Дано последовательность непустых строк. Среди всех строк, начинающихся с одного и того же символа, выбрать наиболее длинную. Если таких строк несколько, то выбрать первую по порядку их следования в исходной последовательности. Полученную последовательность строк упорядочить по возрастанию кодов их начальных символов |
| Вариант 7 - Дана последовательность непустых строк, содержащих только заглавные буквы латинского алфавита. Среди всех строк одинаковой длины выбрать первую в лексикографическом порядке (по возрастанию). Полученную последовательность строк упорядочить по убыванию их длин. |
| Вариант 8 - Дана последовательность непустых строк A, содержащих только заглавные буквы латинского алфавита. Для всех строк, начинающихся с одной и той же буквы, определить их суммарную длину и получить последовательность строк вида «S-C», где S — суммарная длина всех строк из A, которые начинаются с буквы С. Полученную последовательность упорядочить по убыванию числовых значений сумм, а при равных значениях сумм — по возрастанию кодов символов C. |
| Вариант 9 - Дано целое число K (> 0) и целочисленная последовательность A. Найти теоретико-множественную разность двух фрагментов A: первый содержит все четные числа, а второй — все числа с порядковыми номерами, большими K. В полученной последовательности (не содержащей одинаковых элементов) поменять порядок элементов на обратный. |
| Вариант 10 - Даны целые числа D и K (K > 0) и целочисленная последовательность A. Найти теоретико-множественное объединение двух фрагментов A: первый содержит все элементы до первого элемента, большего D (не включая его), а второй — все элементы, начиная с элемента с порядковым номером K. Полученную последовательность (не содержащую одинаковых элементов) отсортировать по убыванию. |

# КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Где применяются LINQ?
2. Что такое LINQ?
3. Как работает сортировка в LINQ?

# РЕКОМЕНДОВАННАЯ ЛИТЕРАТУРА

1. Официальная документация Microsoft C# — Текст: электронный // Microsoft [сайт]. — URL: https://docs.microsoft.com/ (дата обращения: 12.03.2020).
2. Сообщество IT-специалистов — Текст: электронный // Habr [сайт]. — URL: https://habr.com/ (дата обращения: 12.03.2020).
3. Сайт о программирование / — Текст: электронный // Metanit [сайт]. — https://metanit.com/ (дата обращения: 12.03.2020).
4. С# Tutotial — Текст: электронный // Tutotial [сайт]. — URL: https://csharp.net-tutorials.com/ (дата обращения: 13.03.2020).

# ПРИЛОЖЕНИЕ А

**Требования к отчету**

Общие требования:

1. Шрифт – Times New Roman, 14 пт.
2. Интервалы: междустрочный – 1,5 строки, интервал до и после абзаца – 0 пт.
3. Отступ первой строки – 1,25
4. Рисунки и подписи к ним выравниваются по центру.

Требования к структуре отчета:

1. Титульный лист
2. Содержание
3. Основная часть:
   1. Задачи
   2. Код
   3. Блок-схемы
   4. Результат работы
4. Выводы по работе
5. Ответы на контрольные вопросы

# ПРИЛОЖЕНИЕ Б

ОГБПОУ «ТОМСКИЙ ТЕХНИКУМ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ»

Отчет по учебной-практической работе №7

учебной практике   
*УП 05. Программирование*

*Тема: «LINQ»*

Выполнил:

студент \_\_\_\_\_\_\_ группы

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Проверил:  
преподаватель

Сидиков И.Д.

г. Томск – 2020 г