

М. Э. Абрамян

Технология LINQ на примерах

Практикум с использованием
электронного задачника
Programming Taskbook for LINQ



Москва, 2014

УДК 004.438.NET
ББК 32.973.202
A13

Абрамян М. Э.

A13 Технология LINQ на примерах. Практикум с использованием электронного задачника Programming Taskbook for LINQ. – М.: ДМК Пресс, 2014. – 326 с.: ил.

ISBN 978-5-94074-981-3

Книга является практическим введением в технологию LINQ платформы .NET. Она содержит формулировки 250 учебных заданий, связанных с интерфейсами LINQ to Objects и LINQ to XML и включенных в электронный задачник Programming Taskbook for LINQ. В книге также приводятся примеры решений большого числа типовых задач, позволяющие изучить все категории запросов LINQ to Objects и компоненты объектной модели XML DOM, входящей в LINQ to XML. Дополнительный раздел книги посвящен особенностям интерфейсов LINQ to SQL и LINQ to Entities.

Издание предназначено как для начинающих, так и для опытных программистов, желающих получить практические навыки применения технологии LINQ. Оно также может использоваться в качестве задачника-практикума для студентов вузов, посвященных программированию на платформе .NET.

УДК 004.438.NET
ББК 32.973

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

© Абрамян М. Э., 2014

ISBN 978-5-94074-981-3

© Оформление, ДМК Пресс, 2014



Содержание

| | |
|--------------------------|---|
| Предисловие | 6 |
|--------------------------|---|

| | |
|---|---|
| Глава 1. Технология LINQ и ее изучение с применением задачника Programming Taskbook for LINQ | 9 |
|---|---|

| | |
|--|----|
| 1.1. Технология LINQ и связанные с ней программные интерфейсы | 9 |
| 1.2. Общее описание групп заданий LinqBegin, LinqObj, LinqXml | 12 |
| 1.3. Особенности выполнения заданий с использованием задачника Programming Taskbook for LINQ | 15 |

| | |
|---|----|
| Глава 2. Знакомство с запросами LINQ: группа LinqBegin | 17 |
|---|----|

| | |
|--|----|
| 2.1. Поэлементные операции, агрегирование и генерирование последовательностей..... | 18 |
| 2.2. Фильтрация, сортировка, теоретико-множественные операции..... | 20 |
| 2.3. Проецирование | 23 |
| 2.4. Объединение и группировка | 25 |

| | |
|--|----|
| Глава 3. Технология LINQ to Objects: группа LinqObj | 30 |
|--|----|

| | |
|--|----|
| 3.1. Обработка отдельных последовательностей..... | 30 |
| 3.2. Обработка нескольких взаимосвязанных последовательностей..... | 62 |

Глава 4. Технология LINQ to XML: группа LinqXml 78

| | |
|---|----|
| 4.1. Создание XML-документа | 79 |
| 4.2. Анализ содержимого XML-документа | 82 |
| 4.3. Преобразование XML-документа | 84 |
| 4.4. Преобразование типов при обработке XML-документа | 87 |
| 4.5. Работа с пространствами имен XML-документа | 91 |
| 4.6. Дополнительные задания на обработку XML-документов | 92 |

Глава 5. Примеры решения задач из группы**LinqBegin 113**

| | |
|---|-----|
| 5.1. Поэлементные операции: LinqBegin4 | 113 |
| 5.1.1. Создание проекта-заготовки и знакомство с заданием | 113 |
| 5.1.2. Выполнение задания | 120 |
| 5.2. Операция агрегирования и генерирование последовательностей: LinqBegin15 | 129 |
| 5.3. Фильтрация, сортировка, теоретико-множественные операции: LinqBegin31 | 135 |
| 5.4. Проецирование: LinqBegin43 | 146 |
| 5.5. Объединение: LinqBegin52, LinqBegin54 | 153 |
| 5.5.1. Объединение последовательностей и его виды | 153 |
| 5.5.2. Построение перекрестного объединения: LinqBegin52 | 156 |
| 5.5.3. Выражения запросов | 160 |
| 5.5.4. Построение плоского левого внешнего объединения: LinqBegin54 | 165 |
| 5.6. Группировка: LinqBegin60 | 172 |

Глава 6. Примеры решения задач из группы**LinqObj 180**

| | |
|---|-----|
| 6.1. Простое задание на обработку отдельной последовательности: LinqObj4 | 180 |
| 6.1.1. Создание проекта-заготовки и знакомство с заданием. Дополнительные средства окна задачника, связанные с просмотром файловых данных | 180 |
| 6.1.2. Выполнение задания | 186 |
| 6.2. Более сложные задания на обработку отдельных последовательностей: LinqObj41, LinqObj61 | 194 |
| 6.3. Обработка взаимосвязанных последовательностей: LinqObj98 | 203 |

Глава 7. Примеры решения задач из группы

| | |
|---|-----|
| LinqXml | 218 |
| 7.1. Создание XML-документа: LinqXml10 | 218 |
| 7.2. Анализ содержимого XML-документа: LinqXml20 | 227 |
| 7.3. Преобразование XML-документа: LinqXml28, LinqXml32, LinqXml37 | 236 |
| 7.4. Преобразование типов при обработке XML-документа: LinqXml50 | 248 |
| 7.5. Работа с пространствами имен XML-документа: LinqXml57 | 254 |
| 7.6. Дополнительные задания на обработку XML-документов: LinqXml61, LinqXml82..... | 264 |

**Глава 8. Новые средства языка C# 3.0,
связанные с технологией LINQ**

| | |
|---|-----|
| 8.1. Лямбда-выражения | 278 |
| 8.2. Анонимные типы и описатель var | 283 |
| 8.3. Методы расширения..... | 285 |

**Глава 9. Технологии LINQ для обработки
удаленных источников данных**

| | |
|--|-----|
| 9.1. Интерфейсы LINQ to SQL и LINQ to Entities | 288 |
| 9.2. Интерфейс IQueryable<T> и интерпретируемые запросы | 289 |
| 9.3. Основные ограничения на запросы LINQ для удаленных источников данных | 293 |
| 9.4. Пример применения интерфейса LINQ to SQL: LinqObj71 | 294 |
| 9.4.1. Создание и настройка локальной базы данных..... | 294 |
| 9.4.2. Создание и использование простейшей объектной модели базы данных | 299 |
| 9.4.3. Создание и настройка базы данных, основанной на службах..... | 305 |
| 9.4.4. Автоматическая генерация объектной модели базы данных и особенности ее использования | 311 |

| | |
|-------------------------|-----|
| Литература | 317 |
|-------------------------|-----|

| | |
|------------------------|-----|
| Указатель | 318 |
|------------------------|-----|



Предисловие

Книга, предлагаемая вашему вниманию, представляет собой практическое введение в технологию LINQ платформы .NET. Эта технология входит в состав платформы .NET Framework, начиная с версии 3.5; она предоставляет программисту, использующему языки C# или Visual Basic .NET, средства высокого уровня для обработки различных наборов данных. С каждой категорией данных связываются особые интерфейсы (LINQ API), в частности интерфейс LINQ to Objects предназначен для работы с локальными коллекциями, LINQ to XML – для обработки данных в формате XML, интерфейсы LINQ to SQL и LINQ to Entities (Entity Framework) – для работы с удаленными хранилищами данных. При этом базовый набор средств обработки (*запросов LINQ*) остается неизменным для любого интерфейса, обеспечивая универсальный характер технологии LINQ.

К преимуществам технологии LINQ, помимо ее универсального характера, можно отнести краткость и наглядность программ, полученных с ее применением, а также простоту реализации достаточно сложных алгоритмов обработки данных. В то же время, для того чтобы получить эффективные реализации алгоритмов, основанных на применении технологии LINQ, необходимо владеть всем арсеналом доступных запросов и других компонентов LINQ API, выбирая и комбинируя их способом, наиболее подходящим для каждого конкретного случая.

Один из подходов к практическому освоению технологии LINQ (как и любой технологии программирования) состоит в решении набора учебных задач, связанных с различными возможностями этой технологии. Между тем среди книг, посвященных технологии LINQ и включающих фундаментальные руководства [4–5] и справочники [3] (следует также отметить подробные разделы о технологии LINQ в книгах [1–2]), подобные задачки отсутствуют. Предлагаемая книга призвана восполнить этот пробел.

В книге содержатся 250 учебных задач, посвященных интерфейсам LINQ to Object и LINQ to XML. Выбор указанных интерфейсов обосновывается в гл. 1 и обусловлен тем, что первый из них позволяет изучить базовый набор запросов LINQ, а второй дает возможность закрепить навыки в использовании технологии LINQ, применяя ее к задачам обработки данных в формате XML – одном из наиболее распространенных в настоящее время универсальных форматов хранения информации.

Задачи разбиты на три группы:

- ❑ LinqBegin (60 задач, посвященных конкретным запросам LINQ, – см. гл. 2);
- ❑ LinqObj (100 задач на обработку локальных коллекций – см. гл. 3);
- ❑ LinqXml (90 задач, связанных с обработкой документов XML, – см. гл. 4).

Поскольку при выполнении заданий важно ориентироваться на эффективные приемы применения технологии LINQ, в книге приводится большое число примеров решения типовых задач из указанных групп (см. гл. 5–7). Формулировки решенных задач, приведенные в гл. 2–4, снабжаются примечанием, в котором указывается номер пункта с решением данной задачи.

Книга также включает описание новых возможностей языка C# версии 3.0, непосредственно связанных с технологией LINQ (гл. 8), и краткий обзор особенностей интерфейсов LINQ to SQL и LINQ to Entities (гл. 9).

Все задачи, приведенные в книге, входят в состав *электронного задачника Programming Taskbook for LINQ*, являющегося одним из дополнений универсального задачника по программированию Programming Taskbook. Электронный задачник предоставляет программам наборы исходных данных, проверяет правильность полученных результатов, диагностирует различные виды ошибок в программах и отображает на экране все данные, связанные с заданием.

Автоматическая генерация вариантов исходных данных, предоставляемых задачником программе, обеспечивает ее надежное тестирование. Эта возможность оказывается особенно полезной в ситуациях, когда исходные данные являются достаточно сложными и действия по их подготовке и вводу в программу требуют больших усилий. В качестве примеров можно привести задания на обработку массивов, файлов, линейных динамических структур и деревьев. К заданиям со сложными исходными данными можно отнести и за-

дания, связанные с применением технологии LINQ, поскольку все они посвящены обработке последовательностей (в заданиях группы LinqBegin это числовые или строковые последовательности, в заданиях LinqObj это последовательности записей, хранящиеся в текстовых файлах, в заданиях LinqXml – XML-документы).

Задачник включает средства ввода-вывода, позволяющие легко организовать чтение исходных данных и запись результатов. Кроме того, в нем предусмотрены средства отладки, упрощающие поиск и исправление ошибок, и средства визуализации всех данных, связанных с выполняемым заданием. Все эти средства подробно описываются в гл. 5–7, содержащих описания решений типовых задач.

Задания могут выполняться как на языке C#, так и на Visual Basic .NET в любой из версий среды программирования Microsoft Visual Studio, поддерживающих технологию LINQ (версии 2008, 2010, 2012).

Электронный задачник Programming Taskbook и его дополнение Programming Taskbook for LINQ доступны на сайте <http://ptaskbook.com/>.

Приведенные к книге примеры кода доступны на сайте издательства «ДМК Пресс» www.dmk.ru.



Глава 1. Технология LINQ и ее изучение с применением задачника Programming Taskbook for LINQ

1.1. Технология LINQ и связанные с ней программные интерфейсы

Технология LINQ (Language Integrated Query – «запрос, интегрированный в язык»; произносится «линк») расширяет возможности языка программирования путем включения в него дополнительных средств высокого уровня (*запросов LINQ*), предназначенных для преобразования различных наборов данных. «LINQ – это технология Microsoft, предназначенная для обеспечения механизма поддержки уровня языка для опроса данных всех типов» ([4], с. 21). «LINQ – это набор функциональных возможностей языка C# 3.0 и платформы .NET Framework 3.5, обеспечивающих написание безопасных в смысле типизации структурированных запросов к локальным коллекциям объектов и удаленным источникам данных» ([1], с. 315). В каждом из приведенных определений подчеркивается *универсальность* технологии LINQ. Входящие в нее средства можно применять для обработки наборов данных самых разных видов: массивов и других *локальных коллекций* данных, удаленных баз данных, XML-документов, а также любых других источников данных, для которых реализован программный интерфейс LINQ (LINQ API).

В версии .NET Framework 3.5 – первой из версий .NET, поддерживающих технологию LINQ, – были представлены следующие компоненты LINQ API:

- *LINQ to Objects* – базовый интерфейс для стандартных запросов к локальным коллекциям; данный интерфейс основан на

методах класса `System.Linq.Enumerable` (включает около 40 видов запросов) и позволяет обрабатывать любые объекты, реализующие интерфейс `IEnumerable<T>`;

- ❑ *LINQ to XML* – интерфейс, предназначенный для обработки XML-документов; он включает не только набор запросов, дополняющих стандартные запросы *LINQ to Objects* и реализованных в методах класса `System.Xml.Linq.Extensions` (около 10 видов запросов), но и новую *объектную модель* XML-документов (XML DOM), реализованную в виде иерархии классов из пространства имен `System.Xml.Linq`;
- ❑ *LINQ to SQL* и *LINQ to Entities (Entity Framework)* – интерфейсы, предназначенные для взаимодействия с удаленными базами данных в качестве источников наборов данных; эти интерфейсы основаны на методах класса `System.Linq.Queryable` и предназначены для обработки объектов, реализующих интерфейс `IQueryable<T>`.

Одновременно с включением в .NET интерфейсов LINQ API в языки C# и Visual Basic .NET были добавлены новые возможности, позволяющие максимально упростить программный код, связанный с запросами LINQ. Основными из этих возможностей являются *лямбда-выражения*, *методы расширения* и *анонимные типы* (а также *деревья выражений*, используемые в интерфейсах *LINQ to SQL* и *LINQ to Entities*). Кроме того, в языки были включены новые конструкции (так называемые *выражения запросов*), позволяющие представить наиболее сложные запросы LINQ не только в виде цепочек методов расширения, содержащих лямбда-выражения, но и в более наглядном виде, подобном выражениям языка структурированных запросов SQL (такой вариант синтаксиса запросов LINQ получил в русскоязычной литературе название *синтаксиса, облегчающего восприятие*).

Технология LINQ обладает расширяемой архитектурой, которая позволяет разрабатывать дополнительные интерфейсы LINQ API, обеспечивающие обработку специализированных наборов данных. Подобные интерфейсы могут быть реализованы сторонними разработчиками для обеспечения доступа к их хранилищам данных, представленным в специальном формате. В Интернете можно найти ряд дополнительных интерфейсов LINQ, реализованных с той или иной степенью полноты (в качестве примера можно указать *LINQ to Google* и *LINQ to Wiki*), хотя в настоящее время широкое

распространение получили лишь перечисленные выше интерфейсы, реализованные самой компанией Microsoft.

В версии 4.0 .NET Framework набор интерфейсов LINQ был дополнен новым интерфейсом *PLINQ (Parallel LINQ)*. Данный интерфейс содержит дополнительный набор запросов, связанных с «распараллеливанием» обработки локальных коллекций, и новые реализации всех стандартных запросов LINQ, предусматривающие их выполнение с использованием нескольких потоков (threads). Интерфейс PLINQ основан на методах класса System.Linq.Parallel-Enumerable (около 10 методов) и обеспечивает обработку локальных коллекций типа `ParallelQuery<T>`.

При использовании любых интерфейсов LINQ необходимо прежде всего владеть базовым набором методов LINQ, чтобы наиболее эффективным образом формировать из них последовательность запросов, которая приводит к требуемому преобразованию исходного набора данных. Для изучения методов LINQ проще всего обратиться к интерфейсу LINQ to Objects, поскольку он ориентирован на наиболее простой вид последовательностей – локальные коллекции объектов (например, массивы или объекты типа `List<T>`).

Среди трех специализированных интерфейсов (LINQ to XML, LINQ to SQL, LINQ to Entities) особый интерес представляет LINQ to XML. Это связано с тем обстоятельством, что с помощью формата XML можно обеспечить представление любых структур данных, причем полученное представление будет платформенно-, аппаратно- и программно-независимым. Кроме того, во многих предметных областях уже имеются XML-спецификации для представления данных. Таким образом, при разработке современных программ достаточно часто будет требоваться включение в них средств, связанных с обработкой данных в формате XML. Интерфейс LINQ to XML предоставляет все необходимые средства обработки XML-данных, причем по удобству использования и наглядности получаемого кода они превосходят средства стандартной модели W3C DOM, предложенной консорциумом W3C – официальным разработчиком стандарта XML. Поэтому в предлагаемой книге наряду с интерфейсом LINQ to Objects подробно рассматривается и интерфейс LINQ to XML.

При использовании интерфейсов LINQ to SQL и LINQ to Entities необходимо предварительно установить связь с удаленными наборами данных, однако после установки такой связи программный код для обработки удаленных наборов методами LINQ не будет отличаться (или будет иметь незначительные отличия) от кода,

обеспечивающего аналогичную обработку локальных коллекций. Поэтому при условии предварительного изучения LINQ to Objects применение технологии LINQ для обработки удаленных наборов данных не должно вызывать особых трудностей; необходимо лишь дополнительно ознакомиться с действиями, требуемыми для установления связи с удаленной базой данных и получения от нее наборов данных для обработки. Однако указанные вопросы относятся, скорее, к технологиям работы с базами данных, поэтому в книге они обсуждаются очень кратко (см. гл. 9). Дополнительные сведения можно почерпнуть из более подробных руководств по использованию технологии LINQ, например [4–5].

Особенности применения интерфейса PLINQ в книге не рассматриваются, поскольку они относятся скорее к особенностям параллельного многопоточного программирования, чем к собственно технологии LINQ. Не случайно в фундаментальном руководстве [2] интерфейс PLINQ описывается в главе 23 «Параллельное программирование», а не в главах 8–10, специально посвященных технологии LINQ и ее интерфейсам.

1.2. Общее описание групп заданий LinqBegin, LinqObj, LinqXml

Как было отмечено выше, книга посвящена базовым средствам технологии LINQ, реализованным в интерфейсе LINQ to Objects, а также дополнительным возможностям, входящим в интерфейс LINQ to XML и связанным с применением технологии LINQ для обработки данных в формате XML. Для изучения указанных средств применяется «практический» подход, основанный на решении большого числа учебных задач – как относящихся к отдельным категориям запросов LINQ, так и требующих применения всей совокупности средств, входящих в тот или иной интерфейс. Изучив описания решений типовых задач, приведенных в гл. 5–7, читатель должен попытаться применить полученные знания для самостоятельного решения хотя бы части задач, формулировки которых содержатся в гл. 2–4.

Книга содержит 250 задач, разбитых на три группы: LinqBegin, LinqObj и LinqXml.

Группа LinqBegin (60 заданий) предназначена для ознакомления с базовыми запросами LINQ. Каждой категории запросов в ней посвящена отдельная подгруппа; описание подгрупп и связанных с ними заданий приводится в табл. 1.1.

Таблица 1.1. Подгруппы группы LinqBegin

| Название подгруппы | Изучаемые запросы LINQ | Число задач | Номера задач | Решенные задачи |
|--|--|-------------|--------------|---------------------------------|
| Поэлементные операции, агрегирование и генерирование последовательностей | First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault; Count, Sum, Average, Max, Min, Aggregate; Range | 15 | 1–15 | 4 (п. 5.1), 15 (п. 5.2) |
| Фильтрация, сортировка, теоретико-множественные операции | Where, TakeWhile, SkipWhile, Take, Skip; OrderBy, OrderByDescending, ThenBy, ThenByDescending; Distinct, Reverse; Union, Intersect, Except | 16 | 16–31 | 31 (п. 5.3) |
| Проецирование | Select, SelectMany | 12 | 32–43 | 43 (п. 5.4) |
| Объединение и группировка | Concat; Join, GroupJoin; DefaultIfEmpty; GroupBy | 17 | 44–60 | 52, 54 (п. 5.5), 60 (п. 5.6) |

Группа LinqObj предназначена для закрепления навыков применения запросов LINQ и включает 100 заданий на использование интерфейса LINQ to Objects для обработки последовательностей и наборов взаимосвязанных последовательностей. При выполнении заданий группы LinqObj необходимо самостоятельно определить набор методов LINQ, обеспечивающих требуемое преобразование исходных последовательностей.

Задания группы LinqObj разбиты на *серии*, каждая из которых связана с определенной предметной областью и включает задачи различного уровня сложности. Описание подгрупп и серий, входящих в группу LinqObj, приводится в табл. 1.2.

Таблица 1.2. Подгруппы и серии задач группы LinqObj

| Название подгруппы | Серии задач | Число задач | Номера задач | Решенные задачи |
|--|-------------------------------------|-------------|--------------|-----------------------------|
| Обработка отдельных последовательностей | Клиенты фитнес-центра | 12 | 1–12 | 4 (п. 6.1) |
| | Абитуриенты | 12 | 13–24 | |
| | Задолжники по коммунальным платежам | 12 | 25–36 | |
| | Автозаправочные станции | 12 | 37–48 | 41 (п. 6.2) |
| | Баллы ЕГЭ | 12 | 49–60 | |
| | Оценки по предметам | 10 | 61–70 | 61 (п. 6.2) |
| Обработка нескольких взаимосвязанных последовательностей | Продажи товаров | 30 | 71–100 | 98 (п. 6.3), 71 (п. 9.4) |

Формулировки заданий группы LinqObj, относящихся к первым четырем сериям, содержат достаточно подробные указания или ссылки на похожие задания; в двух следующих сериях указаниями

снабжаются лишь отдельные задания, а в последней серии (гл. 3.2) указания отсутствуют.

Группа LinqXml (90 заданий) посвящена интерфейсу LINQ to XML и содержит как подгруппы, посвященные отдельным видам преобразований XML-документа, так и завершающую подгруппу с дополнительными заданиями на обработку XML-документа, в которых требуется совместно использовать различные средства интерфейса LINQ to XML. Описание подгрупп, входящих в группу LinqXml, приводится в табл. 1.3.

Таблица 1.3. Подгруппы группы LinqXml

| Название подгруппы | Число задач | Номера задач | Решенные задачи |
|--|-------------|--------------|---------------------|
| Создание XML-документа | 10 | 1–10 | 10 (п. 7.1) |
| Анализ содержимого XML-документа | 10 | 11–20 | 20 (п. 7.2) |
| Преобразование XML-документа | 20 | 21–40 | 28, 32, 37 (п. 7.3) |
| Преобразование типов при обработке XML-документа | 12 | 41–52 | 50 (п. 7.4) |
| Работа с пространствами имен XML-документа | 8 | 53–60 | 57 (п. 7.5) |
| Дополнительные задания на обработку XML-документов | 30 | 61–90 | 61, 82 (п. 7.6) |

Завершающая подгруппа группы LinqXml состоит из *серий* задач, связанных с теми же предметными областями, что и задачи группы LinqObj. Каждая серия содержит задачи трех видов: (1) простые задачи, требующие лишь изменения способа представления данных в XML-документе (без изменения его структуры); (2) задачи средней сложности, требующие выполнения дополнительной группировки исходных данных; (3) сложные задачи, требующие группировки или объединения исходных данных, а также включения в документ новых данных, полученных из исходных путем их некоторого преобразования (в частности, агрегирования). Описание серий задач из завершающей подгруппы приводится в табл. 1.4.

Таблица 1.4. Серии задач завершающей подгруппы группы LinqXml

| Серия задач | Задачи на изменение структуры | Задачи на группировку | Задачи на определение новых данных |
|-------------------------------------|-------------------------------|-----------------------|------------------------------------|
| Клиенты фитнес-центра | 61–62 | 63–64 | 65–67 |
| Автозаправочные станции | 68–69 | 70–71 | 72–75 |
| Задолжники по коммунальным платежам | 76–77 | 78–79 | 80–82 |
| Оценки по предметам | 83–84 | 85–86 | 87–90 |

1.3. Особенности выполнения заданий с использованием задачника *Programming Taskbook for LINQ*

Все описанные группы задач входят в состав электронного задачника *Programming Taskbook for LINQ*, являющегося дополнением к универсальному задачнику по программированию *Programming Taskbook*. Использование задачника предоставляет учащемуся следующие дополнительные возможности:

- ❑ при выполнении заданий программа получает от задачника набор исходных тестовых данных для обработки (данные генерируются с использованием датчика случайных чисел);
- ❑ задачник автоматически контролирует правильность действий по вводу-выводу данных, информируя учащегося о характере обнаруженных ошибок;
- ❑ результаты, полученные программой, передаются задачнику для проверки (сравнения с образцом правильных данных);
- ❑ полученные результаты наряду с исходными данными и примером правильного решения отображаются в наглядном виде в окне задачника;
- ❑ задание считается выполненным (и информация об этом заносится в специальный файл результатов), если программа учащегося успешно обработает подряд несколько наборов исходных данных, предложенных задачником.

Отмеченные возможности позволяют существенно ускорить процесс выполнения задач, упрощают поиск и исправление ошибок и обеспечивают надежную проверку правильности предложенных решений. Заметим, что все эти возможности доступны при выполнении заданий из любых групп, входящих в задачник *Programming Taskbook*.

Процесс выполнения задания с использованием задачника *Programming Taskbook for LINQ*, начиная с создания проекта-заготовки и заканчивая просмотром содержимого файла результатов с информацией обо всех тестовых запусках программы, подробно описан в п. 5.1.

В задачнике предусмотрены специальные средства ввода-вывода, позволяющие программе учащегося получить исходные данные, сгенерированные задачником, и передать ему найденные результаты. При выполнении заданий группы *LinqBegin* можно использовать

функции, вызов которых обеспечивает ввод или вывод не отдельных элементов данных, а последовательности в целом (см. п. 5.1.2 и 5.3).

В заданиях групп `LinqObj` и `LinqXml` исходные наборы данных должны быть считаны из текстовых файлов (предварительно сформированных задачиком), а результаты – записаны в новый текстовый файл. Действия по чтению данных из текстовых файлов и записи в них результатов легко реализуются с помощью средств стандартной библиотеки платформы .NET (см. п. 6.1 и 7.1). Содержимое всех файлов, связанных с выполняемым заданием, автоматически отображается в окне задачника; это позволяет ознакомиться с вариантами исходных данных и примером правильного решения, просмотреть файл, созданный программой учащегося, а также сравнить его с «эталонным» файлом с правильными результатами. Возможности задачника, связанные с просмотром файловых данных, подробно описываются в п. 6.1.1.

Задачник позволяет выполнять отладочную печать в специальный раздел окна (раздел отладки). В частности, предусмотрен метод расширения `Show`, осуществляющий отладочную печать всех элементов последовательности; данный метод может включаться непосредственно в цепочку методов LINQ, используемых для решения поставленной задачи. Отладочные средства задачника подробно описываются в п. 5.3.



Глава 2. Знакомство с запросами LINQ: группа LinqBegin

При вводе (выводе) последовательности вначале следует ввести (соответственно, вывести) ее размер, а затем ее элементы. Все входные последовательности являются непустыми. Выходные последовательности могут быть пустыми; в этом случае требуется вывести единственное число 0 – размер данной последовательности.

Если в задании идет речь о *порядковых номерах* элементов последовательности, то предполагается, что нумерация ведется от 1 (таким образом, порядковый номер элемента равен *индексу* этого элемента, *увеличенному на 1*).

Для обработки входной последовательности в большинстве заданий достаточно указать *единственный* оператор, содержащий вызовы нужных запросов LINQ to Objects и другие необходимые конструкции, в частности операцию ?? языка C#.

При выполнении заданий с использованием задачника Programming Taskbook можно использовать дополнительные методы, определенные в задачнике:

- ❑ методы GetEnumeratorInt и GetEnumeratorString обеспечивают ввод исходных последовательностей с элементами целого и строкового типа соответственно (выполняется ввод размера последовательности и всех ее элементов, возвращается введенная последовательность);
- ❑ метод Put является методом расширения для последовательности и обеспечивает вывод этой последовательности (выводятся размер последовательности и все ее элементы);
- ❑ метод Show также является методом расширения для последовательности; он обеспечивает печать последовательности в разделе отладки окна задачника и возвращает эту же последовательность (отладочная печать может сопровождаться ком-

ментарием, который указывается в качестве необязательного строкового параметра метода Show).

Использование вспомогательных методов иллюстрируется приведенным ниже фрагментом программы, решающей следующую задачу: извлечь из исходной целочисленной последовательности четные отрицательные числа и заменить порядок их следования на обратный.

```
// Ввод исходных данных
var a = GetEnumerableInt();
// Обработка
var r = a.Where(e => e % 2 == 0 && e < 0).Reverse();
// Вывод результатов
r.Put();
```

Все этапы решения можно объединить в одном операторе, состоящем из цепочки последовательно вызываемых методов:

```
GetEnumerableInt().Where(e => e % 2 == 0 && e < 0).Reverse().Put();
```

Возможен вариант решения, в котором дополнительно выполняется отладочная печать (в данном случае полученная последовательность четных отрицательных чисел печатается перед изменением порядка следования ее элементов и после этого изменения):

```
GetEnumerableInt().Where(e => e % 2 == 0 && e < 0)
    .Show().Reverse().Show().Put();
```

Отладочная печать позволяет увидеть состояние последовательности на различных этапах ее преобразования и тем самым облегчает поиск ошибок.

2.1. Поэлементные операции, агрегирование и генерирование последовательностей

Изучаемые запросы LINQ:

- ❑ First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault (поэлементные операции);
- ❑ Count, Sum, Average, Max, Min, Aggregate (агрегирование);
- ❑ Range (генерирование последовательностей).

Перед выполнением заданий из данного пункта следует ознакомиться с примером решения задачи LinqBegin4, приведенным в п. 5.1.

Описание особенностей использования методов `Range` и `Aggregate` приводится в примере решения задачи `LinqBegin15` (см. п. 5.2).

LinqBegin1. Дана целочисленная последовательность, содержащая как положительные, так и отрицательные числа. Вывести ее первый положительный элемент и последний отрицательный элемент.

LinqBegin2. Даны цифра D (однозначное целое число) и целочисленная последовательность A . Вывести первый положительный элемент последовательности A , оканчивающийся цифрой D . Если требуемых элементов в последовательности A нет, то вывести 0.

LinqBegin3. Даны целое число L (> 0) и строковая последовательность A . Вывести последнюю строку из A , начинающуюся с цифры и имеющую длину L . Если требуемых строк в последовательности A нет, то вывести строку «Not found».

Указание. Для обработки ситуации, связанной с отсутствием требуемых строк, использовать операцию `??`.

LinqBegin4. Даны символ C и строковая последовательность A . Если A содержит единственный элемент, оканчивающийся символом C , то вывести этот элемент; если требуемых строк в A нет, то вывести пустую строку; если требуемых строк больше одной, то вывести строку «Егггг».

Указание. Использовать `try`-блок для перехвата возможного исключения.

Примечание. Решение данной задачи приведено в п. 5.1.

LinqBegin5. Даны символ C и строковая последовательность A . Найти количество элементов A , которые содержат более одного символа и при этом начинаются и оканчиваются символом C .

LinqBegin6. Дана строковая последовательность. Найти сумму длин всех строк, входящих в данную последовательность.

LinqBegin7. Дана целочисленная последовательность. Найти количество ее отрицательных элементов, а также их сумму. Если отрицательные элементы отсутствуют, то дважды вывести 0.

LinqBegin8. Дана 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).

LinqBegin9. Дана целочисленная последовательность. Вывести ее минимальный положительный элемент или число 0, если последовательность не содержит положительных элементов.

LinqBegin10. Даны 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 .

LinqBegin11. Дана последовательность непустых строк. Используя метод `Aggregate`, получить строку, состоящую из начальных символов всех строк исходной последовательности.

LinqBegin12. Дана целочисленная последовательность. Используя метод `Aggregate`, найти произведение последних цифр всех элементов последовательности. Чтобы избежать целочисленного переполнения, при вычислении произведения использовать вещественный числовой тип.

LinqBegin13. Дано целое число $N (> 0)$. Используя методы `Range` и `Sum`, найти сумму $1 + (1/2) + \dots + (1/N)$ (как вещественное число).

LinqBegin14. Даны целые числа A и B ($A < B$). Используя методы `Range` и `Average`, найти среднее арифметическое квадратов всех целых чисел от A до B включительно: $(A^2 + (A + 1)^2 + \dots + B^2) / (B - A + 1)$ (как вещественное число).

LinqBegin15. Дано целое число N ($0 \leq N \leq 15$). Используя методы `Range` и `Aggregate`, найти факториал числа N : $N! = 1 \cdot 2 \cdot \dots \cdot N$ при $N \geq 1$; $0! = 1$. Чтобы избежать целочисленного переполнения, при вычислении факториала использовать вещественный числовой тип.

Примечание. Решение данной задачи приведено в п. 5.2.

2.2. Фильтрация, сортировка, теоретико-множественные операции

Изучаемые запросы LINQ:

- ❑ `Where`, `TakeWhile`, `SkipWhile`, `Take`, `Skip` (фильтрация);
- ❑ `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending` (сортировка);
- ❑ `Distinct`, `Reverse` (удаление повторяющихся элементов и инвертирование);
- ❑ `Union`, `Intersect`, `Except` (теоретико-множественные операции).

Перед выполнением заданий из данного пункта следует ознакомиться с примером решения задачи LinqBegin31, приведенным в п. 5.3.

LinqBegin16. Дана целочисленная последовательность. Извлечь из нее все положительные числа, сохранив их исходный порядок следования.

LinqBegin17. Дана целочисленная последовательность. Извлечь из нее все нечетные числа, сохранив их исходный порядок следования и удалив все вхождения повторяющихся элементов, кроме первых.

LinqBegin18. Дана целочисленная последовательность. Извлечь из нее все четные отрицательные числа, поменяв порядок извлеченных чисел на обратный.

LinqBegin19. Даны цифра D (целое однозначное число) и целочисленная последовательность A . Извлечь из A все различные положительные числа, оканчивающиеся цифрой D (в исходном порядке). При наличии повторяющихся элементов удалять все их вхождения, кроме последних.

Указание. Последовательно применить методы Reverse, Distinct, Reverse.

LinqBegin20. Дана целочисленная последовательность. Извлечь из нее все положительные двузначные числа, отсортировав их по возрастанию.

LinqBegin21. Дана строковая последовательность. Строки последовательности содержат только заглавные буквы латинского алфавита. Отсортировать последовательность по возрастанию длин строк, а строки одинаковой длины – в лексикографическом порядке по убыванию.

LinqBegin22. Даны целое число $K (> 0)$ и строковая последовательность A . Строки последовательности содержат только цифры и заглавные буквы латинского алфавита. Извлечь из A все строки длины K , оканчивающиеся цифрой, отсортировав их в лексикографическом порядке по возрастанию.

LinqBegin23. Даны целое число $K (> 0)$ и целочисленная последовательность A . Начиная с элемента A с порядковым номером K , извлечь из A все нечетные двузначные числа, отсортировав их по убыванию.

LinqBegin24. Даны целое число $K (> 0)$ и строковая последовательность A . Из элементов A , предшествующих элементу с порядко-

вым номером K , извлечь те строки, которые имеют нечетную длину и начинаются с заглавной латинской буквы, изменив порядок следования извлеченных строк на обратный.

LinqBegin25. Даны целые числа K_1 и K_2 и целочисленная последовательность A ; $1 \leq K_1 < K_2 \leq N$, где N – размер последовательности A . Найти сумму положительных элементов последовательности с порядковыми номерами от K_1 до K_2 включительно.

LinqBegin26. Даны целые числа K_1 и K_2 и последовательность непустых строк A ; $1 < K_1 < K_2 \leq N$, где N – размер последовательности A . Найти среднее арифметическое длин всех элементов последовательности, кроме элементов с порядковыми номерами от K_1 до K_2 включительно, и вывести его как вещественное число.

LinqBegin27. Даны целое число D и целочисленная последовательность A . Начиная с первого элемента A , большего D , извлечь из A все нечетные положительные числа, поменяв порядок извлеченных чисел на обратный.

LinqBegin28. Даны целое число L (> 0) и последовательность непустых строк A . Строки последовательности содержат только цифры и заглавные буквы латинского алфавита. Из элементов A , предшествующих первому элементу, длина которого превышает L , извлечь строки, оканчивающиеся буквой. Полученную последовательность отсортировать по убыванию длин строк, а строки одинаковой длины – в лексикографическом порядке по возрастанию.

LinqBegin29. 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.

LinqBegin30. Даны 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.

LinqBegin31. Даны целое число K (> 0) и последовательность непустых строк A . Строки последовательности содержат только цифры и заглавные буквы латинского алфавита. Найти теоретико-множественное пересечение двух фрагментов A : первый содержит

К начальным элементам, а второй – все элементы, расположенные после последнего элемента, оканчивающегося цифрой. Полученную последовательность (не содержащую одинаковых элементов) отсортировать по возрастанию длин строк, а строки одинаковой длины – в лексикографическом порядке по возрастанию.

Примечание. Решение данной задачи приведено в п. 5.3.

2.3. Проецирование

Изучаемые запросы LINQ:

□ Select, SelectMany (проецирование).

Перед выполнением заданий из данного пункта следует ознакомиться с примером решения задачи `LinqBegin43`, приведенным в п. 5.4.

LinqBegin32. Дана последовательность непустых строк *A*. Получить последовательность символов, каждый элемент которой является начальным символом соответствующей строки из *A*. Порядок символов должен быть обратным по отношению к порядку элементов исходной последовательности.

LinqBegin33. Дана целочисленная последовательность. Обрабатывая только положительные числа, получить последовательность их последних цифр и удалить в полученной последовательности все вхождения одинаковых цифр, кроме первого. Порядок полученных цифр должен соответствовать порядку исходных чисел.

LinqBegin34. Дана последовательность положительных целых чисел. Обрабатывая только нечетные числа, получить последовательность их строковых представлений и отсортировать ее в лексикографическом порядке по возрастанию.

LinqBegin35. Дана целочисленная последовательность. Получить последовательность чисел, каждый элемент которой равен произведению соответствующего элемента исходной последовательности на его порядковый номер (1, 2, ...). В полученной последовательности удалить все элементы, не являющиеся двузначными, и поменять порядок оставшихся элементов на обратный.

LinqBegin36. Дана последовательность непустых строк. Получить последовательность символов, которая определяется следующим образом: если соответствующая строка исходной последовательности имеет нечетную длину, то в качестве символа берется первый символ

этой строки; в противном случае берется последний символ строки. Отсортировать полученные символы по убыванию их кодов.

LinqBegin37. Дана строковая последовательность A . Строки последовательности содержат только заглавные буквы латинского алфавита. Получить новую последовательность строк, элементы которой определяются по соответствующим элементам A следующим образом: пустые строки в новую последовательность не включаются, а к непустым приписывается порядковый номер данной строки в исходной последовательности (например, если пятый элемент A имеет вид «ABC», то в полученной последовательности он будет иметь вид «ABC5»). При нумерации должны учитываться и пустые строки последовательности A . Отсортировать полученную последовательность в лексикографическом порядке по возрастанию.

LinqBegin38. Дана целочисленная последовательность A . Получить новую последовательность чисел, элементы которой определяются по соответствующим элементам последовательности A следующим образом: если порядковый номер элемента A делится на 3 (3, 6, ...), то этот элемент в новую последовательность не включается; если остаток от деления порядкового номера на 3 равен 1 (1, 4, ...), то в новую последовательность добавляется удвоенное значение этого элемента; в противном случае (для элементов A с номерами 2, 5, ...) элемент добавляется в новую последовательность без изменений. В полученной последовательности сохранить исходный порядок следования элементов.

LinqBegin39. Дана строковая последовательность A . Получить последовательность цифровых символов, входящих в строки последовательности A (символы могут повторяться). Порядок символов должен соответствовать порядку строк A и порядку следования символов в каждой строке.

Указание. Использовать метод `SelectMany` с учетом того, что строка может интерпретироваться как последовательность символов.

LinqBegin40. Даны число K (> 0) и строковая последовательность A . Получить последовательность символов, содержащую символы всех строк из A , имеющих длину, большую или равную K (символы могут повторяться). В полученной последовательности поменять порядок элементов на обратный.

LinqBegin41. Даны целое число K (> 0) и строковая последовательность A . Каждый элемент последовательности представляет

сбой несколько слов из заглавных латинских букв, разделенных символами «.» (точка). Получить последовательность строк, содержащую все слова длины K из элементов A в лексикографическом порядке по возрастанию (слова могут повторяться).

LinqBegin42. Дана последовательность непустых строк. Получить последовательность символов, которая определяется следующим образом: для строк с нечетными порядковыми номерами (1, 3, ...) в последовательность символов включаются все прописные латинские буквы, содержащиеся в этих строках, а для строк с четными номерами (2, 4, ...) – все их строчные латинские буквы. В полученной последовательности символов сохранить их исходный порядок следования.

LinqBegin43. Даны целое число $K (> 0)$ и последовательность непустых строк A . Получить последовательность символов, которая определяется следующим образом: для первых K элементов последовательности A в новую последовательность заносятся символы, стоящие на нечетных позициях данной строки (1, 3, ...), а для остальных элементов A – символы на четных позициях (2, 4, ...). В полученной последовательности поменять порядок элементов на обратный.

Примечание. Решение данной задачи приведено в п. 5.4.

2.4. Объединение и группировка

Изучаемые запросы LINQ:

- ☐ Concat (сцепление);
- ☐ Join, GroupJoin (объединение);
- ☐ DefaultIfEmpty (замена пустой последовательности на одноэлементную);
- ☐ GroupBy (группировка).

Перед выполнением заданий, связанных с *объединением* последовательностей (LinqBegin44–LinqBegin55), следует ознакомиться с примерами решения задач LinqBegin52 и LinqBegin54, приведенными в п. 5.5.

Перед выполнением заданий, связанных с *группировкой* последовательностей (LinqBegin56–LinqBegin60), следует ознакомиться с примером решения задачи LinqBegin60, приведенным в п. 5.6.

LinqBegin44. Даны целые числа K_1 и K_2 и целочисленные последовательности A и B . Получить последовательность, содержащую

все числа из A , большие K_1 , и все числа из B , меньшие K_2 . Отсортировать полученную последовательность по возрастанию.

LinqBegin45. Даны целые положительные числа L_1 и L_2 и строковые последовательности A и B . Строки последовательностей содержат только цифры и заглавные буквы латинского алфавита. Получить последовательность, содержащую все строки из A длины L_1 и все строки из B длины L_2 . Отсортировать полученную последовательность в лексикографическом порядке по убыванию.

LinqBegin46. Даны последовательности положительных целых чисел A и B ; все числа в каждой последовательности различны. Найти последовательность всех пар чисел, удовлетворяющих следующим условиям: первый элемент пары принадлежит последовательности A , второй принадлежит B , и оба элемента оканчиваются одной и той же цифрой. Результирующая последовательность называется *внутренним объединением* последовательностей A и B по *ключу*, определяемому последними цифрами исходных чисел. Представить найденное объединение в виде последовательности строк, содержащих первый и второй элементы пары, разделенные дефисом, например «49-129». Порядок следования пар должен определяться исходным порядком элементов последовательности A , а для равных первых элементов – порядком элементов последовательности B .

LinqBegin47. Даны последовательности положительных целых чисел A и B ; все числа в каждой последовательности различны. Найти внутреннее объединение A и B (см. LinqBegin46), пары в котором должны удовлетворять следующему условию: последняя цифра первого элемента пары (из A) должна совпадать с первой цифрой второго элемента пары (из B). Представить найденное объединение в виде последовательности строк, содержащих первый и второй элементы пары, разделенные двоеточием, например «49:921». Порядок следования пар должен определяться исходным порядком элементов последовательности A , а для равных первых элементов пар – лексикографическим порядком строковых представлений вторых элементов (по возрастанию).

LinqBegin48. Даны строковые последовательности A и B ; все строки в каждой последовательности различны, имеют ненулевую длину и содержат только цифры и заглавные буквы латинского алфавита. Найти внутреннее объединение A и B (см. LinqBegin46), каждая пара которого должна содержать строки одинаковой длины. Представить найденное объединение в виде последовательности строк, содержащих первый и второй элементы пары, разделенные

двоеточием, например «AB:CD». Порядок следования пар должен определяться лексикографическим порядком первых элементов пар (по возрастанию), а для равных первых элементов – лексикографическим порядком вторых элементов пар (по убыванию).

LinqBegin49. Даны строковые последовательности A , B и C ; все строки в каждой последовательности различны, имеют ненулевую длину и содержат только цифры и заглавные буквы латинского алфавита. Найти внутреннее объединение A , B и C (см. LinqBegin46), каждая тройка которого должна содержать строки, начинающиеся с одного и того же символа. Представить найденное объединение в виде последовательности строк вида « $E_A=E_B=E_C$ », где E_A , E_B , E_C – элементы из A , B , C соответственно. Для различных элементов E_A сохраняется исходный порядок их следования, для равных элементов E_A порядок троек определяется лексикографическим порядком элементов E_B (по возрастанию), а для равных элементов E_A и E_B – лексикографическим порядком элементов E_C (по убыванию).

LinqBegin50. Даны строковые последовательности A и B ; все строки в каждой последовательности различны и имеют ненулевую длину. Получить последовательность строк вида « $E:N$ », где E обозначает один из элементов последовательности A , а N – количество элементов из B , начинающихся с того же символа, что и элемент E (например, «abc:4»); количество N может быть равно 0. Порядок элементов полученной последовательности должен определяться исходным порядком элементов последовательности A .

Указание. Использовать метод GroupJoin.

LinqBegin51. Даны последовательности положительных целых чисел A и B ; все числа в последовательности A различны. Получить последовательность строк вида « $S:E$ », где S обозначает сумму тех чисел из B , которые оканчиваются на ту же цифру, что и число E – один из элементов последовательности A (например, «74:23»); если для числа E не найдено ни одного подходящего числа из последовательности B , то в качестве S указать 0. Расположить элементы полученной последовательности по возрастанию значений найденных сумм, а при равных суммах – по убыванию значений элементов A .

LinqBegin52. Даны строковые последовательности A и B ; все строки в каждой последовательности различны, имеют ненулевую длину и содержат только цифры и заглавные буквы латинского алфавита. Получить последовательность всевозможных комбинаций вида « $E_A=E_B$ », где E_A – некоторый элемент из A , E_B – некоторый

элемент из B , причем оба элемента оканчиваются цифрой (например, «AF3=D78»). Упорядочить полученную последовательность в лексикографическом порядке по возрастанию элементов E_A , а при одинаковых элементах E_A – в лексикографическом порядке по убыванию элементов E_B .

Указание. Для перебора комбинаций использовать методы `SelectMany` и `Select`.

Примечание. Решение данной задачи приведено в п. 5.5.2.

LinqBegin53. Даны целочисленные последовательности A и B . Получить последовательность всех *различных* сумм, в которых первое слагаемое берется из A , а второе – из B . Упорядочить полученную последовательность по возрастанию.

LinqBegin54. Даны строковые последовательности A и B ; все строки в каждой последовательности различны, имеют ненулевую длину и содержат только цифры и заглавные буквы латинского алфавита. Найти последовательность всех пар строк, удовлетворяющих следующим условиям: первый элемент пары принадлежит последовательности A , а второй либо является одним из элементов последовательности B , начинающихся с того же символа, что и первый элемент пары, либо является пустой строкой (если B не содержит ни одной подходящей строки). Результирующая последовательность называется *левым внешним объединением* последовательностей A и B по *ключу*, определяемому первыми символами исходных строк. Представить найденное объединение в виде последовательности строк вида « $E_A.E_B$ », где E_A – элемент из A , а E_B – либо один из соответствующих ему элементов из B , либо пустая строка. Расположить элементы полученной строковой последовательности в лексикографическом порядке по возрастанию.

Указание. Использовать методы `GroupJoin`, `DefaultIfEmpty`, `Select` и `SelectMany`.

Примечание. Решение данной задачи приведено в п. 5.5.4.

LinqBegin55. Даны последовательности положительных целых чисел A и B ; все числа в каждой последовательности различны. Найти левое внешнее объединение A и B (см. LinqBegin54), пары в котором должны удовлетворять следующему условию: оба элемента пары оканчиваются одной и той же цифрой. Представить найденное объединение в виде последовательности строк вида « $E_A:E_B$ », где E_A – число из A , а E_B – либо одно из соответствующих ему чисел из B ,

либо 0 (если в B не содержится чисел, соответствующих E_A). Расположить элементы полученной последовательности по убыванию чисел E_A , а при одинаковых числах E_A – по возрастанию чисел E_B .

LinqBegin56. Дана целочисленная последовательность A . *Сгруппировать* элементы последовательности A , оканчивающиеся одной и той же цифрой, и на основе этой группировки получить последовательность строк вида « $D:S$ », где D – *ключ группировки* (то есть некоторая цифра, которой оканчивается хотя бы одно из чисел последовательности A), а S – сумма всех чисел из A , которые оканчиваются цифрой D . Полученную последовательность упорядочить по возрастанию ключей.

Указание. Использовать метод GroupBy.

LinqBegin57. Дана целочисленная последовательность. Среди всех элементов последовательности, оканчивающихся одной и той же цифрой, выбрать максимальный. Полученную последовательность максимальных элементов упорядочить по возрастанию их последних цифр.

LinqBegin58. Дана 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.

LinqBegin59. 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.

LinqBegin60. 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 C . Полученную последовательность упорядочить по убыванию числовых значений сумм, а при равных значениях сумм – по возрастанию кодов символов C .

Примечание. Решение данной задачи приведено в п. 5.6.



Глава 3. Технология LINQ to Objects: группа LinqObj

В каждом задании даются имена одного или нескольких текстовых файлов, содержащих исходные последовательности, а также имя текстового файла, в который требуется записать результаты обработки исходных последовательностей (имя результирующего файла указывается последним). Каждая исходная последовательность содержится в отдельном файле. Все исходные файлы содержат текст в кодировке «windows-1251»; эта же кодировка должна использоваться при записи полученных данных в результирующий файл.

Каждый элемент последовательности размещается в отдельной строке файла, в начале и конце строки пробелы отсутствуют, поля элемента не содержат пробелов и разделяются ровно одним пробелом. Все исходные числовые данные являются положительными. В качестве десятичного разделителя используется *точка*.

Если в задание входят дополнительные числовые или строковые исходные данные, то они указываются в начале набора исходных данных (перед именами файлов).

3.1. Обработка отдельных последовательностей

Перед выполнением заданий из данного пункта следует ознакомиться с примерами решения задач LinqObj4, LinqObj41 и LinqObj61, приведенными в п. 6.1–6.2.

LinqObj1. Исходная последовательность содержит сведения о клиентах фитнес-центра. Каждый элемент последовательности включает следующие целочисленные поля:

<Код клиента> <Год> <Номер месяца> <Продолжительность занятий (в часах)>

Найти элемент последовательности с минимальной продолжительностью занятий. Вывести эту продолжительность, а также соответствующие ей год и номер месяца (в указанном порядке на той же строке). Если имеется несколько элементов с минимальной продолжительностью, то вывести данные того из них, который является последним в исходной последовательности.

Указание. Для нахождения требуемого элемента следует использовать методы `OrderByDescending` и `Last`. Вывод полученных результатов организовать с учетом последнего примечания в п. 6.1.2.

LinqObj2. Исходная последовательность содержит сведения о клиентах фитнес-центра. Каждый элемент последовательности включает следующие целочисленные поля:

<Номер месяца> <Год> <Код клиента> <Продолжительность занятий (в часах)>

Найти элемент последовательности с максимальной продолжительностью занятий. Вывести эту продолжительность, а также соответствующие ей год и номер месяца (в указанном порядке на той же строке). Если имеется несколько элементов с максимальной продолжительностью, то вывести данные, соответствующие самой поздней дате.

Указание. Ср. с LinqObj1. Для нахождения требуемого элемента следует использовать сортировку по набору ключей «продолжительность занятий, год, номер месяца».

LinqObj3. Исходная последовательность содержит сведения о клиентах фитнес-центра. Каждый элемент последовательности включает следующие целочисленные поля:

<Год> <Номер месяца> <Продолжительность занятий (в часах)> <Код клиента>

Определить год, в котором суммарная продолжительность занятий всех клиентов была наибольшей, и вывести этот год и наибольшую суммарную продолжительность. Если таких годов было несколько, то вывести наименьший из них.

Указание. Ср. с LinqObj1. Использовать группировку по полю «Год».

LinqObj4. Исходная последовательность содержит сведения о клиентах фитнес-центра. Каждый элемент последовательности включает следующие целочисленные поля:

<Год> <Номер месяца> <Продолжительность занятий (в часах)> <Код клиента>

Для каждого клиента, присутствующего в исходных данных, определить суммарную продолжительность занятий в течение всех лет (вначале выводить суммарную продолжительность, затем код клиента). Сведения о каждом клиенте выводить на новой строке и упорядочивать по убыванию суммарной продолжительности, а при их равенстве – по возрастанию кода клиента.

Примечание. Решение данной задачи приведено в п. 6.1.

LinqObj5. Исходная последовательность содержит сведения о клиентах фитнес-центра. Каждый элемент последовательности включает следующие целочисленные поля:

<Код клиента> <Продолжительность занятий (в часах)> <Год> <Номер месяца>

Для каждого клиента, присутствующего в исходных данных, определить общее количество месяцев, в течение которых он посещал занятия (вначале выводить количество месяцев, затем код клиента). Сведения о каждом клиенте выводить на новой строке и упорядочивать по возрастанию количества месяцев, а при их равенстве – по возрастанию кода клиента.

Указание. Ср. с LinqObj4.

LinqObj6. Исходная последовательность содержит сведения о клиентах фитнес-центра. Каждый элемент последовательности включает следующие целочисленные поля:

<Код клиента> <Продолжительность занятий (в часах)> <Год> <Номер месяца>

Для каждого месяца определить суммарную продолжительность занятий всех клиентов за все годы (вначале выводить суммарную продолжительность, затем номер месяца). Если данные о некотором месяце отсутствуют, то для этого месяца вывести 0. Сведения о каждом месяце выводить на новой строке и упорядочивать по убыванию суммарной продолжительности, а при равной продолжительности – по возрастанию номера месяца.

Указание. С помощью метода Range сформировать вспомогательную последовательность месяцев (1, 2, ..., 12) и выполнить ее левое внешнее объединение с исходной последовательностью, используя метод GroupJoin.

LinqObj7. Дано целое число K – код одного из клиентов фитнес-центра. Исходная последовательность содержит сведения о клиентах этого фитнес-центра. Каждый элемент последовательности включает следующие целочисленные поля:

<Продолжительность занятий (в часах)> <Год> <Номер месяца> <Код клиента>

Для каждого года, в котором клиент с кодом K посещал центр, определить месяц, в котором продолжительность занятий данного клиента была наибольшей для данного года (если таких месяцев несколько, то выбирать месяц с наименьшим номером). Сведения о каждом годе выводить на новой строке в следующем порядке: год, номер месяца, продолжительность занятий в этом месяце. Упорядочивать сведения по убыванию номера года. Если данные о клиенте с кодом K отсутствуют, то записать в результирующий файл строку «Нет данных».

Указание. Для отбора данных, связанных с клиентом K , использовать метод `Where`. Затем выполнить группировку по полю «год» и для каждой полученной последовательности выбрать требуемый месяц с помощью сортировки по набору ключей «продолжительность занятий, номер месяца». Обработку особой ситуации, связанной с отсутствием требуемых данных, выполнять с использованием метода `DefaultIfEmpty` с параметром «Нет данных».

LinqObj8. Дано целое число K – код одного из клиентов фитнес-центра. Исходная последовательность содержит сведения о клиентах этого фитнес-центра. Каждый элемент последовательности включает следующие целочисленные поля:

<Продолжительность занятий (в часах)> <Код клиента> <Год> <Номер месяца>

Для каждого года, в котором клиент с кодом K посещал центр, определить месяц, в котором продолжительность занятий данного клиента была наименьшей для данного года (если таких месяцев несколько, то выбирать первый из этих месяцев в исходном наборе; месяцы с нулевой продолжительностью занятий не учитывать). Сведения о каждом годе выводить на новой строке в следующем порядке: наименьшая продолжительность занятий, год, номер месяца. Упорядочивать сведения по возрастанию продолжительности занятий, а при равной продолжительности – по возрастанию номера года. Если данные о клиенте с кодом K отсутствуют, то записать в результирующий файл строку «Нет данных».

Указание. Ср. с LinqObj7.

LinqObj9. Дано целое число K – код одного из клиентов фитнес-центра. Исходная последовательность содержит сведения о клиентах этого фитнес-центра. Каждый элемент последовательности включает следующие целочисленные поля:

<Код клиента> <Продолжительность занятий (в часах)> <Номер месяца> <Год>

Для каждого года, в котором клиент с кодом K посещал центр, определить число месяцев, для которых продолжительность занятий данного клиента превосходила 15 часов (вначале выводить число месяцев, затем год). Если для некоторого года требуемые месяцы отсутствуют, то вывести для него 0. Сведения о каждом годе выводить на новой строке; данные упорядочивать по убыванию числа месяцев, а при равном числе месяцев – по возрастанию номера года. Если данные об указанном клиенте отсутствуют, то записать в результирующий файл строку «Нет данных».

Указание. Ср. с LinqObj7.

LinqObj10. Исходная последовательность содержит сведения о клиентах фитнес-центра. Каждый элемент последовательности включает следующие целочисленные поля:

<Год> <Номер месяца> <Код клиента> <Продолжительность занятий (в часах)>

Для каждой пары «год–месяц», присутствующей в исходных данных, определить количество клиентов, которые посещали центр в указанное время (вначале выводится год, затем месяц, потом количество клиентов). Сведения о каждой паре «год–месяц» выводить на новой строке и упорядочивать по убыванию номера года, а для одинакового номера года – по возрастанию номера месяца.

Указание. Использовать группировку по составному ключу «год, номер месяца», представив ключ в виде анонимного типа с полями `year` и `month`. В дальнейшем использовать поля ключа для сортировки полученной последовательности.

LinqObj11. Исходная последовательность содержит сведения о клиентах фитнес-центра. Каждый элемент последовательности включает следующие целочисленные поля:

<Код клиента> <Год> <Номер месяца> <Продолжительность занятий (в часах)>

Для каждой пары «год–месяц», присутствующей в исходных данных, определить общую продолжительность занятий всех клиентов в указанное время (вначале выводится общая продолжительность, затем год, потом месяц). Сведения о каждой паре «год–месяц» выводить на новой строке и упорядочивать по возрастанию общей продолжительности занятий, для одинаковой продолжительности – по убыванию номера года, а для одинакового номера года – по возрастанию номера месяца.

Указание. Ср. с LinqObj10.

LinqObj12. Дано целое число P ($10 < P < 50$). Исходная последовательность содержит сведения о клиентах фитнес-центра. Каждый элемент последовательности включает следующие целочисленные поля:

<Продолжительность занятий (в часах)> <Код клиента> <Номер месяца> <Год>

Для каждого года, присутствующего в исходных данных, определить количество месяцев, в которых суммарная длительность занятий всех клиентов составляла более P процентов от суммарной длительности за этот год (вначале выводить количество месяцев, затем год). Если в некотором году ни для одного месяца не выполнялось требуемое условие, то вывести для него 0. Сведения о каждом годе выводить на новой строке и упорядочивать по убыванию количества месяцев, а для одинакового количества – по возрастанию номера года.

Указание. Выполнить группировку по полю «год», включив в элементы сгруппированной последовательности (представленные в виде анонимного типа) вспомогательное поле *total*, равное суммарной длительности занятий за данный год, и последовательность *lengths* значений длительности для каждого имеющегося месяца этого года (для нахождения последовательности требуется выполнить дополнительную группировку по полю «номер месяца»). Использовать поле *total* при последующем анализе последовательности *lengths* для подсчета числа требуемых месяцев.

LinqObj13. Исходная последовательность содержит сведения об абитуриентах. Каждый элемент последовательности включает следующие поля:

<Номер школы> <Год поступления> <Фамилия>

Для каждого года, присутствующего в исходных данных, найти школу с наибольшим номером среди школ, которые окончили абитуриенты, поступившие в этом году, и вывести год и найденный номер школы. Сведения о каждом годе выводить на новой строке и упорядочивать по возрастанию номера года.

Указание. Использовать группировку по полю «год» и метод `Max`.

LinqObj14. Исходная последовательность содержит сведения об абитуриентах. Каждый элемент последовательности включает следующие поля:

<Год поступления> <Номер школы> <Фамилия>

Определить, в какие годы общее число абитуриентов для всех школ было наибольшим, и вывести это число, а также количество таких лет. Каждое число выводить на новой строке.

Указание. Выполнить группировку по полю «год», сохранив полученную последовательность во вспомогательной переменной. Использовать полученную последовательность для нахождения наибольшего числа абитуриентов (с помощью метода `Max`), также сохранив это число во вспомогательной переменной `max`. Вторично использовать полученную последовательность (совместно с переменной `max`) для подсчета количества лет, для которых число абитуриентов было максимальным. Вывод полученных результатов организовать с учетом последнего примечания в п. 6.1.2.

LinqObj15. Исходная последовательность содержит сведения об абитуриентах. Каждый элемент последовательности включает следующие поля:

<Номер школы> <Год поступления> <Фамилия>

Определить, в какие годы общее число абитуриентов для всех школ было наибольшим, и вывести это число, а также годы, в которые оно было достигнуто (годы упорядочивать по возрастанию, каждое число выводить на новой строке).

Указание. Ср. с LinqObj14.

LinqObj16. Исходная последовательность содержит сведения об абитуриентах. Каждый элемент последовательности включает следующие поля:

<Год поступления> <Номер школы> <Фамилия>

Для каждого года, присутствующего в исходных данных, вывести общее число абитуриентов, поступивших в этом году (вначале указывать число абитуриентов, затем год). Сведения о каждом годе выводить на новой строке и упорядочивать по убыванию числа поступивших, а для совпадающих чисел – по возрастанию номера года.

Указание. Использовать группировку по полю «год» и метод Count.

LinqObj17. Исходная последовательность содержит сведения об абитуриентах. Каждый элемент последовательности включает следующие поля:

<Номер школы> <Год поступления> <Фамилия>

Для каждого года, присутствующего в исходных данных, вывести число различных школ, которые окончили абитуриенты, поступившие в этом году (вначале указывать число школ, затем год). Сведения о каждом годе выводить на новой строке и упорядочивать по возрастанию числа школ, а для совпадающих чисел – по возрастанию номера года.

Указание. С помощью подходящего варианта метода GroupBy выполнить группировку по полю «год», сохранив в полученных последовательностях (соответствующих различным годам) только значения номеров школ и вызвав для этих последовательностей методы Distinct и Count.

LinqObj18. Исходная последовательность содержит сведения об абитуриентах. Каждый элемент последовательности включает следующие поля:

<Год поступления> <Номер школы> <Фамилия>

Найти годы, для которых число абитуриентов было не меньше среднего значения по всем годам (вначале указывать число абитуриентов для данного года, затем год). Сведения о каждом годе выводить на новой строке и упорядочивать по убыванию числа абитуриентов, а для совпадающих чисел – по возрастанию номера года.

Указание. Ср. с LinqObj15.

LinqObj19. Исходная последовательность содержит сведения об абитуриентах. Каждый элемент последовательности включает следующие поля:

<Фамилия> <Год поступления> <Номер школы>

Для каждой школы вывести общее число абитуриентов за все годы и фамилию первого из абитуриентов этой школы, содержащихся в исходном наборе данных (вначале указывать номер школы, затем число абитуриентов, потом фамилию). Сведения о каждой школе выводить на новой строке и упорядочивать по возрастанию номеров школ.

Указание. Ср. с LinqObj16. Для отбора первого из абитуриентов использовать метод First.

LinqObj20. Исходная последовательность содержит сведения об абитуриентах. Каждый элемент последовательности включает следующие поля:

<Фамилия> <Номер школы> <Год поступления>

Определить, для каких школ общее число абитуриентов за все годы было наибольшим, и вывести данные об абитуриентах из этих школ (вначале указывать номер школы, затем фамилию абитуриента). Сведения о каждом абитуриенте выводить на новой строке и упорядочивать по возрастанию номеров школ, а для одинаковых номеров – в порядке следования абитуриентов в исходном наборе данных.

Указание. Ср. с LinqObj15. После отбора требуемых школ отсортировать полученную иерархическую последовательность по номерам школ и преобразовать ее в плоскую последовательность методом SelectMany.

LinqObj21. Исходная последовательность содержит сведения об абитуриентах. Каждый элемент последовательности включает следующие поля:

<Фамилия> <Год поступления> <Номер школы>

Определить, для каких школ общее число абитуриентов за все годы было наибольшим, и вывести для каждой из этих школ данные о первом абитуриенте в алфавитном порядке (вначале указывать фамилию абитуриента, затем номер школы). Сведения о каждом абиту-

риенте выводить на новой строке и упорядочивать в алфавитном порядке, а для одинаковых фамилий – по возрастанию номеров школ.

Указание. Ср. с LinqObj20. В данном случае сортировку надо выполнять после преобразования иерархической последовательности в плоскую.

LinqObj22. Исходная последовательность содержит сведения об абитуриентах. Каждый элемент последовательности включает следующие поля:

<Фамилия> <Номер школы> <Год поступления>

Для каждой школы найти годы поступления абитуриентов из этой школы и вывести номер школы и найденные для нее годы (годы располагаются на той же строке, что и номер школы, и упорядочиваются по возрастанию). Сведения о каждой школе выводить на новой строке и упорядочивать по возрастанию номеров школ.

Указание. Ср. с LinqObj17. Для формирования строки со списком лет, соответствующих каждому номеру школы, использовать метод Aggregate.

LinqObj23. Исходная последовательность содержит сведения об абитуриентах. Каждый элемент последовательности включает следующие поля:

<Фамилия> <Год поступления> <Номер школы>

Для каждой пары «год–школа», присутствующей в исходных данных, найти число абитуриентов, относящихся к этому году и школе, и вывести год, номер школы и найденное число абитуриентов. Сведения о каждой паре «год–школа» выводить на новой строке и упорядочивать по убыванию года, а для совпадающих годов – по возрастанию номера школы.

Указание. Ср. с LinqObj10.

LinqObj24. Исходная последовательность содержит сведения об абитуриентах. Каждый элемент последовательности включает следующие поля:

<Фамилия> <Номер школы> <Год поступления>

Для каждой пары «школа–год», присутствующей в исходных данных, найти трех первых абитуриентов, относящихся к этой школе

и году, и вывести номер школы, год и найденные фамилии (в порядке их следования в исходном наборе данных). Если для некоторой пары «школа–год» имеется менее трех абитуриентов, то вывести информацию обо всех абитуриентах, относящихся к этой паре. Сведения о каждой паре «школа–год» выводить на новой строке и упорядочивать по возрастанию номера школы, а для совпадающих номеров – по убыванию года.

Указание. Ср. с LinqObj10. Для отбора трех абитуриентов использовать метод Take. Для формирования строки со списком фамилий использовать метод Aggregate.

LinqObj25. Исходная последовательность содержит сведения о задолжниках по оплате коммунальных услуг, живущих в 144-квартирном 9-этажном доме. Каждый элемент последовательности включает следующие поля:

<Фамилия> <Задолженность> <Номер квартиры>

Задолженность указывается в виде дробного числа (целая часть – рубли, дробная часть – копейки). В каждом подъезде на каждом этаже располагаются по 4 квартиры. Найти номер подъезда, жильцы которого имеют наибольшую суммарную задолженность, и вывести этот номер вместе с размером суммарной задолженности (выводится с двумя дробными знаками). Считать, что суммарные задолженности для всех подъездов имеют различные значения.

Указание. Выполнить группировку по номеру подъезда, используя следующую формулу нахождения номера подъезда $\text{entrance} = (\text{flat} - 1) / 36 + 1$ (операция «/» обозначает деление нацело). Учесть особенности обработки вещественных чисел, отмеченные при описании решения задачи LinqObj61 в п. 6.2. Вывод полученных результатов организовать с учетом последнего примечания в п. 6.1.2.

LinqObj26. Исходная последовательность содержит сведения о задолжниках по оплате коммунальных услуг, живущих в 144-квартирном 9-этажном доме. Каждый элемент последовательности включает следующие поля:

<Номер квартиры> <Фамилия> <Задолженность>

Задолженность указывается в виде дробного числа (целая часть – рубли, дробная часть – копейки). В каждом подъезде на каждом

этаже располагаются по 4 квартиры. Для каждого из 4 подъездов дома вывести сведения о задолжниках, живущих в этом подъезде: номер подъезда, число задолжников, средняя задолженность для жильцов этого подъезда (выводится с двумя дробными знаками). Жильцы, не имеющие долга, при вычислении средней задолженности не учитываются. Сведения о каждом подъезде выводить на отдельной строке и упорядочивать по возрастанию номера подъезда. Если в каком-либо подъезде задолжники отсутствуют, то данные об этом подъезде не выводить.

Указание. Ср. с LinqObj25.

LinqObj27. Исходная последовательность содержит сведения о задолжниках по оплате коммунальных услуг, живущих в 144-квартирном 9-этажном доме. Каждый элемент последовательности включает следующие поля:

<Фамилия> <Номер квартиры> <Задолженность>

Задолженность указывается в виде дробного числа (целая часть – рубли, дробная часть – копейки). В каждом подъезде на каждом этаже располагаются по 4 квартиры. Для каждого из 9 этажей дома вывести сведения о задолжниках, живущих на этом этаже: число задолжников, номер этажа, суммарная задолженность для жильцов этого этажа (выводится с двумя дробными знаками). Сведения о каждом этаже выводить на отдельной строке и упорядочивать по возрастанию числа задолжников, а для совпадающих чисел – по возрастанию этажа. Если на каком-либо этаже задолжники отсутствуют, то данные об этом этаже не выводить.

Указание. Ср. с LinqObj26. Использовать формулу нахождения номера этажа `floor` по номеру квартиры `flat`:
$$\text{floor} = (\text{flat} - 1) \% 36 / 4 + 1$$
 (операция «/» обозначает деление нацело, операция «%» – взятие остатка от деления нацело).

LinqObj28. Исходная последовательность содержит сведения о задолжниках по оплате коммунальных услуг, живущих в 144-квартирном 9-этажном доме. Каждый элемент последовательности включает следующие поля:

<Задолженность> <Фамилия> <Номер квартиры>

Задолженность указывается в виде дробного числа (целая часть – рубли, дробная часть – копейки). В каждом подъезде на каждом

этаже располагаются по 4 квартиры. Для каждого из 9 этажей дома вывести сведения о задолжниках, живущих на этом этаже: номер этажа, суммарная задолженность для жильцов этого этажа (выводится с двумя дробными знаками), число задолжников. Сведения о каждом этаже выводить на отдельной строке и упорядочивать по убыванию номера этажа. Если на каком-либо этаже задолжники отсутствуют, то вывести для этого этажа нулевые данные.

Указание. Ср. с LinqObj27. В данном случае вместо группировки по этажам следует выполнить левое внешнее объединение последовательности этажей (1, 2, ..., 9) и исходной последовательности (см. указание к LinqObj6).

LinqObj29. Исходная последовательность содержит сведения о задолжниках по оплате коммунальных услуг, живущих в 144-квартирном 9-этажном доме. Каждый элемент последовательности включает следующие поля:

<Номер квартиры> <Фамилия> <Задолженность>

Задолженность указывается в виде дробного числа (целая часть – рубли, дробная часть – копейки). В каждом подъезде на каждом этаже располагаются по 4 квартиры. Для каждого из 4 подъездов дома найти жильца с наибольшей задолженностью и вывести сведения о нем: номер подъезда, номер квартиры, фамилия жильца, задолженность (выводится с двумя дробными знаками). Считать, что в наборе исходных данных все задолженности имеют различные значения. Сведения о каждом задолжнике выводить на отдельной строке и упорядочивать по возрастанию номера подъезда. Если в каком-либо подъезде задолжники отсутствуют, то данные об этом подъезде не выводить.

Указание. Ср. с LinqObj25. Для нахождения жильца с наибольшей задолженностью отсортировать последовательность жильцов по убыванию задолженности и воспользоваться методом First.

LinqObj30. Исходная последовательность содержит сведения о задолжниках по оплате коммунальных услуг, живущих в 144-квартирном 9-этажном доме. Каждый элемент последовательности включает следующие поля:

<Фамилия> <Задолженность> <Номер квартиры>

Задолженность указывается в виде дробного числа (целая часть – рубли, дробная часть – копейки). В каждом подъезде на каждом этаже располагаются по 4 квартиры. Для каждого из 9 этажей дома найти жильца с наименьшей задолженностью и вывести сведения о нем: номер квартиры, номер этажа, фамилия жильца, задолженность (выводится с двумя дробными знаками). Считать, что в наборе исходных данных все задолженности имеют различные значения. Сведения о каждом задолжнике выводить на отдельной строке и упорядочивать по возрастанию номера квартиры. Если на каком-либо этаже задолжники отсутствуют, то данные об этом этаже не выводить.

Указание. Ср. с LinqObj27 и LinqObj29.

LinqObj31. Исходная последовательность содержит сведения о задолжниках по оплате коммунальных услуг, живущих в 144-квартирном 9-этажном доме. Каждый элемент последовательности включает следующие поля:

<Задолженность> <Фамилия> <Номер квартиры>

Задолженность указывается в виде дробного числа (целая часть – рубли, дробная часть – копейки). В каждом подъезде на каждом этаже располагаются по 4 квартиры. Для каждого из 4 подъездов дома найти трех жильцов с наибольшей задолженностью и вывести сведения о них: задолженность (выводится с двумя дробными знаками), номер подъезда, номер квартиры, фамилия жильца. Считать, что в наборе исходных данных все задолженности имеют различные значения. Сведения о каждом задолжнике выводить на отдельной строке и упорядочивать по убыванию размера задолженности (номер подъезда при сортировке не учитывать). Если в каком-либо подъезде число задолжников меньше трех, то включить в полученный набор всех задолжников этого подъезда.

Указание. Ср. с LinqObj25. Для отбора трех задолжников использовать метод Take. См. также указание к LinqObj21.

LinqObj32. Исходная последовательность содержит сведения о задолжниках по оплате коммунальных услуг, живущих в 144-квартирном 9-этажном доме. Каждый элемент последовательности включает следующие поля:

<Фамилия> <Номер квартиры> <Задолженность>

Задолженность указывается в виде дробного числа (целая часть – рубли, дробная часть – копейки). В каждом подъезде на каждом этаже располагаются по 4 квартиры. Для каждого из 9 этажей дома найти жильца с наименьшей задолженностью и вывести сведения о нем: номер этажа и задолженность (выводится с двумя дробными знаками). Считать, что в наборе исходных данных все задолженности имеют различные значения. Сведения о каждом этаже выводить на отдельной строке и упорядочивать по возрастанию номера этажа. Если на каком-либо этаже задолжники отсутствуют, то для этого этажа вывести задолженность, равную 0.00.

Указание. Ср. с LinqObj28; по поводу определения жильца с наименьшей задолженностью см. указание к LinqObj29. При обработке пустых последовательностей использовать метод `DefaultIfEmpty`.

LinqObj33. Исходная последовательность содержит сведения о задолжниках по оплате коммунальных услуг, живущих в 144-квартирном 9-этажном доме. Каждый элемент последовательности включает следующие поля:

<Номер квартиры> <Задолженность> <Фамилия>

Задолженность указывается в виде дробного числа (целая часть – рубли, дробная часть – копейки). В каждом подъезде на каждом этаже располагаются по 4 квартиры. Найти жильцов, долг которых не меньше величины средней задолженности по дому, и вывести сведения о них: номер квартиры, фамилия, задолженность (выводится с двумя дробными знаками). Жильцы, не имеющие долга, при вычислении средней задолженности не учитываются. Сведения о каждом задолжнике выводить на отдельной строке и упорядочивать по возрастанию номеров квартир.

Указание. Для того чтобы избежать многократного вычисления средней задолженности по дому, сохранить ее значение (найденное с помощью метода `Average`) во вспомогательной переменной.

LinqObj34. Исходная последовательность содержит сведения о задолжниках по оплате коммунальных услуг, живущих в 144-квартирном 9-этажном доме. Каждый элемент последовательности включает следующие поля:

<Задолженность> <Номер квартиры> <Фамилия>

Задолженность указывается в виде дробного числа (целая часть – рубли, дробная часть – копейки). В каждом подъезде на каждом этаже располагаются по 4 квартиры. Найти жильцов, долг которых не больше величины средней задолженности по дому, и вывести сведения о них: номер этажа, номер квартиры, фамилия, задолженность (выводится с двумя дробными знаками). Жильцы, не имеющие долга, при вычислении средней задолженности не учитываются. Сведения о каждом задолжнике выводить на отдельной строке и упорядочивать по убыванию номеров этажей, а для одинаковых этажей – по возрастанию номеров квартир.

Указание. Ср. с LinqObj33.

LinqObj35. Исходная последовательность содержит сведения о задолжниках по оплате коммунальных услуг, живущих в 144-квартирном 9-этажном доме. Каждый элемент последовательности включает следующие поля:

<Задолженность> <Фамилия> <Номер квартиры>

Задолженность указывается в виде дробного числа (целая часть – рубли, дробная часть – копейки). В каждом подъезде на каждом этаже располагаются по 4 квартиры. Для каждого из 4 подъездов дома найти задолжников, долг которых не меньше величины средней задолженности по данному подъезду, и вывести сведения о них: номер подъезда, задолженность (выводится с двумя дробными знаками), фамилия, номер квартиры. Жильцы, не имеющие долга, при вычислении средней задолженности не учитываются. Сведения о каждом задолжнике выводить на отдельной строке и упорядочивать по возрастанию номеров подъездов, а для одинаковых подъездов – по убыванию размера задолженности. Считать, что в наборе исходных данных все задолженности имеют различные значения.

Указание. См. указания к LinqObj25 и LinqObj12.

LinqObj36. Исходная последовательность содержит сведения о задолжниках по оплате коммунальных услуг, живущих в 144-квартирном 9-этажном доме. Каждый элемент последовательности включает следующие поля:

<Номер квартиры> <Фамилия> <Задолженность>

Задолженность указывается в виде дробного числа (целая часть – рубли, дробная часть – копейки). В каждом подъезде на каждом

этаже располагаются по 4 квартиры. Для каждого из 9 этажей дома найти должников, долг которых не больше величины средней задолженности по данному этажу, и вывести сведения о них: номер этажа, задолженность (выводится с двумя дробными знаками), фамилия, номер квартиры. Жильцы, не имеющие долга, при вычислении средней задолженности не учитываются. Сведения о каждом должнике выводить на отдельной строке и упорядочивать по возрастанию номеров этажей, а для одинаковых этажей – по возрастанию размера задолженности. Считать, что в наборе исходных данных все задолженности имеют различные значения.

Указание. Ср. с LinqObj35; см. также указание к LinqObj27.

LinqObj37. Исходная последовательность содержит сведения об автозаправочных станциях (АЗС). Каждый элемент последовательности включает следующие поля:

| | | | |
|------------|-----------------|-----------------------------|---------|
| <Компания> | <Марка бензина> | <Цена 1 литра (в копейках)> | <Улица> |
|------------|-----------------|-----------------------------|---------|

Названия компаний и улиц не содержат пробелов. В качестве марки бензина указываются числа 92, 95 или 98. Каждая компания имеет не более одной АЗС на каждой улице; цены на разных АЗС одной и той же компании могут различаться. Для каждой марки бензина, присутствующей в исходных данных, определить минимальную и максимальную цену литра бензина этой марки (вначале выводить марку, затем цены в указанном порядке). Сведения о каждой марке выводить на новой строке и упорядочивать по убыванию значения марки.

LinqObj38. Исходная последовательность содержит сведения об автозаправочных станциях (АЗС). Каждый элемент последовательности включает следующие поля:

| | | | |
|-----------------------------|-----------------|------------|---------|
| <Цена 1 литра (в копейках)> | <Марка бензина> | <Компания> | <Улица> |
|-----------------------------|-----------------|------------|---------|

Названия компаний и улиц не содержат пробелов. В качестве марки бензина указываются числа 92, 95 или 98. Каждая компания имеет не более одной АЗС на каждой улице; цены на разных АЗС одной и той же компании могут различаться. Для каждой марки бензина, присутствующей в исходных данных, определить количество станций, предлагавших эту марку (вначале выводить количество станций, затем номер марки). Сведения о каждой марке выводить на новой строке и упорядочивать по возрастанию количества станций, а для одинакового количества – по возрастанию значения марки.

LinqObj39. Дано целое число M – значение одной из марок бензина. Исходная последовательность содержит сведения об автозаправочных станциях (АЗС). Каждый элемент последовательности включает следующие поля:

<Улица> <Компания> <Марка бензина> <Цена 1 литра (в копейках)>

Названия компаний и улиц не содержат пробелов. В качестве марки бензина указываются числа 92, 95 или 98. Каждая компания имеет не более одной АЗС на каждой улице; цены на разных АЗС одной и той же компании могут различаться. Для каждой улицы определить количество АЗС, предлагавших марку бензина M (вначале выводить количество АЗС на данной улице, затем название улицы; количество АЗС может быть равно 0). Сведения о каждой улице выводить на новой строке и упорядочивать по возрастанию количества АЗС, а для одинакового количества – по названиям улиц в алфавитном порядке.

Указание. Ср. с LinqObj41.

LinqObj40. Исходная последовательность содержит сведения об автозаправочных станциях (АЗС). Каждый элемент последовательности включает следующие поля:

<Компания> <Улица> <Марка бензина> <Цена 1 литра (в копейках)>

Названия компаний и улиц не содержат пробелов. В качестве марки бензина указываются числа 92, 95 или 98. Каждая компания имеет не более одной АЗС на каждой улице; цены на разных АЗС одной и той же компании могут различаться. Для каждой улицы определить количество АЗС, предлагавших определенную марку бензина (вначале выводить название улицы, затем три числа – количество АЗС для бензина марки 92, 95 и 98; некоторые из этих чисел могут быть равны 0). Сведения о каждой улице выводить на новой строке и упорядочивать по названиям улиц в алфавитном порядке.

Указание. После выполнения группировки по полю «улица» построить левое внешнее объединение последовательности (92, 95, 98) и последовательностей, полученных в результате группировки. Для формирования строки со списком найденных чисел использовать метод `Aggregate`.

LinqObj41. Дано целое число M – значение одной из марок бензина. Исходная последовательность содержит сведения об автоза-

правочных станциях (АЗС). Каждый элемент последовательности включает следующие поля:

<Марка бензина> <Улица> <Компания> <Цена 1 литра (в копейках)>

Названия компаний и улиц не содержат пробелов. В качестве марки бензина указываются числа 92, 95 или 98. Каждая компания имеет не более одной АЗС на каждой улице; цены на разных АЗС одной и той же компании могут различаться. Для каждой улицы, на которой имеются АЗС с бензином марки *M*, определить максимальную цену бензина этой марки (вначале выводить максимальную цену, затем название улицы). Сведения о каждой улице выводить на новой строке и упорядочивать по возрастанию максимальной цены, а для одинаковой цены – по названиям улиц в алфавитном порядке. Если ни одной АЗС с бензином марки *M* не найдено, то записать в результирующий файл строку «Нет».

Примечание. Решение данной задачи приведено в п. 6.2.

LinqObj42. Исходная последовательность содержит сведения об автозаправочных станциях (АЗС). Каждый элемент последовательности включает следующие поля:

<Марка бензина> <Компания> <Улица> <Цена 1 литра (в копейках)>

Названия компаний и улиц не содержат пробелов. В качестве марки бензина указываются числа 92, 95 или 98. Каждая компания имеет не более одной АЗС на каждой улице; цены на разных АЗС одной и той же компании могут различаться. Для каждой улицы определить минимальную цену бензина каждой марки (вначале выводить название улицы, затем три числа – минимальную цену для бензина марки 92, 95 и 98). При отсутствии бензина нужной марки выводить число 0. Сведения о каждой улице выводить на новой строке и упорядочивать по названиям улиц в алфавитном порядке.

Указание. Ср. с LinqObj40.

LinqObj43. Дано целое число *M* – значение одной из марок бензина. Исходная последовательность содержит сведения об автозаправочных станциях (АЗС). Каждый элемент последовательности включает следующие поля:

<Цена 1 литра (в копейках)> <Марка бензина> <Улица> <Компания>

Названия компаний и улиц не содержат пробелов. В качестве марки бензина указываются числа 92, 95 или 98. Каждая компания имеет не более одной АЗС на каждой улице; цены на разных АЗС одной и той же компании могут различаться. Для каждой компании определить разброс цен на бензин указанной марки M (вначале выводить разность максимальной и минимальной цены бензина марки M для АЗС данной компании, затем – название компании). Если бензин марки M не предлагался данной компанией, то разброс положить равным -1 . Сведения о каждой компании выводить на новой строке, данные упорядочивать по убыванию значений разброса, а для равных значений разброса – по названиям компаний в алфавитном порядке.

Указание. В отличие от решения LinqObj41, перед группировкой (по полю «компания») нельзя выполнять отбор элементов, соответствующих марке M , поскольку в результате такого отбора сгруппированная последовательность не будет содержать информацию о компаниях, станции которых не предлагали бензин марки M . Таким образом, отбор следует проводить для уже сгруппированных данных, предусматривая особую обработку пустых последовательностей (с использованием метода `DefaultIfEmpty`).

LinqObj44. Исходная последовательность содержит сведения об автозаправочных станциях (АЗС). Каждый элемент последовательности включает следующие поля:

| | | | |
|-----------------|-----------------------------|------------|---------|
| <Марка бензина> | <Цена 1 литра (в копейках)> | <Компания> | <Улица> |
|-----------------|-----------------------------|------------|---------|

Названия компаний и улиц не содержат пробелов. В качестве марки бензина указываются числа 92, 95 или 98. Каждая компания имеет не более одной АЗС на каждой улице; цены на разных АЗС одной и той же компании могут различаться. Для каждой компании определить разброс цен для всех марок бензина (вначале выводить название компании, затем три числа – разброс цен для бензина марки 92, 95 и 98). При отсутствии бензина нужной марки выводить число -1 . Сведения о каждой компании выводить на новой строке и упорядочивать по названиям компаний в алфавитном порядке.

Указание. Ср. с LinqObj40.

LinqObj45. Исходная последовательность содержит сведения об автозаправочных станциях (АЗС). Каждый элемент последовательности включает следующие поля:

<Компания> <Цена 1 литра (в копейках)> <Марка бензина> <Улица>

Названия компаний и улиц не содержат пробелов. В качестве марки бензина указываются числа 92, 95 или 98. Каждая компания имеет не более одной АЗС на каждой улице; цены на разных АЗС одной и той же компании могут различаться. Для каждой улицы определить количество АЗС (вначале выводить название улицы, затем количество АЗС). Сведения о каждой улице выводить на новой строке и упорядочивать по названиям улиц в алфавитном порядке.

Указание. Выполнить группировку по полю «улица» и для каждой полученной последовательности выполнить группировку по полю «компания».

LinqObj46. Исходная последовательность содержит сведения об автозаправочных станциях (АЗС). Каждый элемент последовательности включает следующие поля:

<Улица> <Марка бензина> <Цена 1 литра (в копейках)> <Компания>

Названия компаний и улиц не содержат пробелов. В качестве марки бензина указываются числа 92, 95 или 98. Каждая компания имеет не более одной АЗС на каждой улице; цены на разных АЗС одной и той же компании могут различаться. Для каждой компании определить количество АЗС, предлагавших все три марки бензина (вначале выводить количество АЗС, затем – название компании; количество может быть равно 0). Сведения о каждой компании выводить на новой строке и упорядочивать по убыванию количества АЗС, а при равных количествах – по названиям компаний в алфавитном порядке.

Указание. Ср. с LinqObj45.

LinqObj47. Исходная последовательность содержит сведения об автозаправочных станциях (АЗС). Каждый элемент последовательности включает следующие поля:

<Цена 1 литра (в копейках)> <Компания> <Улица> <Марка бензина>

Названия компаний и улиц не содержат пробелов. В качестве марки бензина указываются числа 92, 95 или 98. Каждая компания имеет не более одной АЗС на каждой улице; цены на разных АЗС одной и той же компании могут различаться. Вывести данные обо всех АЗС, предлагавших не менее двух марок бензина (вначале вы-

водится название компании, затем название улицы, потом количество предлагавшихся марок бензина). Сведения о каждой АЗС выводить на новой строке и упорядочивать по названиям компаний в алфавитном порядке, а для одинаковых компаний – по названиям улиц (также в алфавитном порядке). Если ни одной требуемой АЗС не найдено, то записать в результирующий файл строку «Нет».

Указание. Выполнить группировку по ключу, полученному в результате сцепления полей «компания» и «улица» (с пробелом между ними). Для обработки ситуации, связанной с пустой последовательностью, использовать метод `DefaultIfEmpty` с параметром «Нет».

LinqObj48. Исходная последовательность содержит сведения об автозаправочных станциях (АЗС). Каждый элемент последовательности включает следующие поля:

| | | | |
|-----------------------------|---------|-----------------|------------|
| <Цена 1 литра (в копейках)> | <Улица> | <Марка бензина> | <Компания> |
|-----------------------------|---------|-----------------|------------|

Названия компаний и улиц не содержат пробелов. В качестве марки бензина указываются числа 92, 95 или 98. Каждая компания имеет не более одной АЗС на каждой улице; цены на разных АЗС одной и той же компании могут различаться. Перебрать все возможные комбинации улиц и компаний, содержащихся в исходном наборе данных, и для каждой пары «улица–компания» вывести название улицы, название компании и количество марок бензина, которое предлагает АЗС данной компании, расположенная на данной улице (если АЗС отсутствует, то количество полагается равным 0). Сведения о каждой паре выводить на новой строке и упорядочивать по названиям улиц в алфавитном порядке, а для одинаковых названий улиц – по названиям компаний (также в алфавитном порядке).

Указание. Построить перекрестное объединение последовательности всех улиц и последовательности всех компаний, получив в результате набор строк, каждая из которых содержит название улицы и название компании, разделенные пробелом. После этого построить левое внешнее объединение полученного набора строк и исходной последовательности.

LinqObj49. Исходная последовательность содержит сведения о результатах сдачи учащимися ЕГЭ по математике, русскому языку и информатике (в указанном порядке). Каждый элемент последовательности включает следующие поля:

<Фамилия> <Инициалы> <Номер школы> <Баллы ЕГЭ>

Баллы ЕГЭ представляют собой три целых числа в диапазоне от 0 до 100, которые отделяются друг от друга одним пробелом. Определить наименьший суммарный балл и вывести его. Вывести также сведения обо всех учащихся, получивших наименьший суммарный балл (для каждого учащегося указывать фамилию, инициалы и номер школы). Сведения о каждом учащемся выводить на отдельной строке и располагать в порядке их следования в исходном наборе.

Указание. Для того чтобы избежать многократного вычисления наименьшего суммарного балла, сохранить его значение (найденное с помощью метода `Min`) во вспомогательной переменной. Вывод полученных результатов организовать с учетом последнего примечания в п. 6.1.2.

LinqObj50. Исходная последовательность содержит сведения о результатах сдачи учащимися ЕГЭ по математике, русскому языку и информатике (в указанном порядке). Каждый элемент последовательности включает следующие поля:

<Фамилия> <Инициалы> <Баллы ЕГЭ> <Номер школы>

Баллы ЕГЭ представляют собой три целых числа в диапазоне от 0 до 100, которые отделяются друг от друга одним пробелом. Определить два наибольших суммарных балла и вывести эти баллы на одной строке в порядке убывания (считать, что в исходных данных всегда присутствуют учащиеся с различными суммарными баллами). Также вывести сведения обо всех учащихся, получивших один из двух наибольших суммарных баллов (для каждого учащегося указывать фамилию, инициалы и суммарный балл). Сведения о каждом учащемся выводить на отдельной строке и располагать в порядке их следования в исходном наборе.

Указание. Ср. с LinqObj49. Для нахождения двух наибольших суммарных баллов сформировать последовательность различных суммарных баллов и применить к ней методы `OrderByDescending` и `Take`. При отборе учащихся удобно использовать метод `Contains`.

LinqObj51. Исходная последовательность содержит сведения о результатах сдачи учащимися ЕГЭ по математике, русскому языку и информатике (в указанном порядке). Каждый элемент последовательности включает следующие поля:

<Баллы ЕГЭ> <Фамилия> <Инициалы> <Номер школы>

Баллы ЕГЭ представляют собой три целых числа в диапазоне от 0 до 100, которые отделяются друг от друга одним пробелом. Для каждой школы вывести сведения об учащемся, набравшем наибольший балл ЕГЭ по информатике среди учащихся этой школы. Если таких учащихся несколько, то вывести сведения о первом учащемся в порядке их следования в исходном наборе. Сведения о каждом учащемся выводить на отдельной строке, указывая номер школы, фамилию учащегося, его инициалы и балл ЕГЭ по информатике. Данные упорядочивать по возрастанию номера школы.

Указание. Ср. с LinqObj29.

LinqObj52. Исходная последовательность содержит сведения о результатах сдачи учащимися ЕГЭ по математике, русскому языку и информатике (в указанном порядке). Каждый элемент последовательности включает следующие поля:

<Номер школы> <Фамилия> <Инициалы> <Баллы ЕГЭ>

Баллы ЕГЭ представляют собой три целых числа в диапазоне от 0 до 100, которые отделяются друг от друга одним пробелом. Для каждой школы вывести сведения об учащемся, набравшем наименьший суммарный балл ЕГЭ среди учащихся этой школы. Если таких учащихся несколько, то вывести сведения о первом учащемся в алфавитном порядке их фамилий и инициалов. Сведения о каждом учащемся выводить на отдельной строке, указывая номер школы, суммарный балл ЕГЭ, фамилию учащегося и его инициалы. Данные упорядочивать по убыванию номера школы.

Указание. Ср. с LinqObj51.

LinqObj53. Исходная последовательность содержит сведения о результатах сдачи учащимися ЕГЭ по математике, русскому языку и информатике (в указанном порядке). Каждый элемент последовательности включает следующие поля:

<Фамилия> <Инициалы> <Номер школы> <Баллы ЕГЭ>

Баллы ЕГЭ представляют собой три целых числа в диапазоне от 0 до 100, которые отделяются друг от друга одним пробелом. Для каждой школы определить количество учащихся, суммарный балл которых превышает 150 баллов (вначале выводится количество

учащихся, набравших в сумме более 150 баллов, затем номер школы; количество учащихся может быть равно 0). Сведения о каждой школе выводить на новой строке и упорядочивать по убыванию количества учащихся, а для одинакового количества – по возрастанию номера школы.

Указание. Ср. с LinqObj41. В данном случае отбор по суммарному баллу следует проводить после группировки по номеру школы.

LinqObj54. Исходная последовательность содержит сведения о результатах сдачи учащимися ЕГЭ по математике, русскому языку и информатике (в указанном порядке). Каждый элемент последовательности включает следующие поля:

<Баллы ЕГЭ> <Номер школы> <Фамилия> <Инициалы>

Баллы ЕГЭ представляют собой три целых числа в диапазоне от 0 до 100, которые отделяются друг от друга одним пробелом. Для каждой школы найти среднее значение суммарного балла ЕГЭ, набранного учащимися этой школы (среднее значение является целым числом – результатом *деления нацело* суммы баллов всех учащихся на количество учащихся). Сведения о каждой школе выводить на отдельной строке, указывая средний суммарный балл ЕГЭ и номер школы. Данные упорядочивать по убыванию среднего балла, а при равных значениях среднего балла – по возрастанию номера школы.

Указание. Ср. с LinqObj41.

LinqObj55. Исходная последовательность содержит сведения о результатах сдачи учащимися ЕГЭ по математике, русскому языку и информатике (в указанном порядке). Каждый элемент последовательности включает следующие поля:

<Баллы ЕГЭ> <Фамилия> <Инициалы> <Номер школы>

Баллы ЕГЭ представляют собой три целых числа в диапазоне от 0 до 100, которые отделяются друг от друга одним пробелом. Вывести сведения об учащихся, набравших не менее 50 баллов по каждому предмету (вначале выводятся фамилия и инициалы, затем номер школы и суммарный балл ЕГЭ по всем предметам). Сведения о каждом учащемся выводить на отдельной строке в алфавитном порядке фамилий и инициалов, а при их совпадении – в порядке следования учащихся в наборе исходных данных. Если ни один из учащихся не

удовлетворяет указанным условиям, то записать в результирующий файл текст «Требуемые учащиеся не найдены».

LinqObj56. Исходная последовательность содержит сведения о результатах сдачи учащимися ЕГЭ по математике, русскому языку и информатике (в указанном порядке). Каждый элемент последовательности включает следующие поля:

<Фамилия> <Инициалы> <Баллы ЕГЭ> <Номер школы>

Баллы ЕГЭ представляют собой три целых числа в диапазоне от 0 до 100, которые отделяются друг от друга одним пробелом. Вывести сведения об учащихся, набравших более 90 баллов хотя бы по одному из предметов (вначале выводятся фамилия и инициалы, затем номер школы). Сведения о каждом учащемся выводить на отдельной строке и располагать в алфавитном порядке фамилий и инициалов, а при их совпадении – по возрастанию номера школы. Если ни один из учащихся не удовлетворяет указанным условиям, то записать в результирующий файл текст «Требуемые учащиеся не найдены».

LinqObj57. Исходная последовательность содержит сведения о результатах сдачи учащимися ЕГЭ по математике, русскому языку и информатике (в указанном порядке). Каждый элемент последовательности включает следующие поля:

<Номер школы> <Фамилия> <Инициалы> <Баллы ЕГЭ>

Баллы ЕГЭ представляют собой три целых числа в диапазоне от 0 до 100, которые отделяются друг от друга одним пробелом. Для каждой школы вывести фамилии и инициалы трех первых учащихся (в алфавитном порядке), набравших менее 50 баллов по каждому предмету (вначале выводится номер школы, затем фамилия и инициалы). Сведения о каждом учащемся выводить на отдельной строке и упорядочивать по возрастанию номера школы, а для совпадающих номеров – в алфавитном порядке фамилий и инициалов. Если для некоторой школы имеется менее трех учащихся, удовлетворяющих указанным условиям, то вывести сведения обо всех таких учащихся. Если в исходном наборе нет ни одного учащегося, удовлетворяющего указанным условиям, то записать в результирующий файл текст «Требуемые учащиеся не найдены».

LinqObj58. Исходная последовательность содержит сведения о результатах сдачи учащимися ЕГЭ по математике, русскому языку

и информатике (в указанном порядке). Каждый элемент последовательности включает следующие поля:

<Фамилия> <Инициалы> <Номер школы> <Баллы ЕГЭ>

Баллы ЕГЭ представляют собой три целых числа в диапазоне от 0 до 100, которые отделяются друг от друга одним пробелом. Для каждой школы найти трех первых учащихся (в алфавитном порядке), набравших менее 50 баллов хотя бы по одному из предметов, и вывести их фамилию, инициалы и номер школы. Сведения о каждом учащемся выводить на отдельной строке и упорядочивать в алфавитном порядке фамилий и инициалов, а при их совпадении – по возрастанию номера школы. Если для некоторой школы имеется менее трех учащихся, удовлетворяющих указанным условиям, то вывести сведения обо всех таких учащихся. Если в исходном наборе нет ни одного учащегося, удовлетворяющего указанным условиям, то записать в результирующий файл текст «Требуемые учащиеся не найдены».

LinqObj59. Исходная последовательность содержит сведения о результатах сдачи учащимися ЕГЭ по математике, русскому языку и информатике (в указанном порядке). Каждый элемент последовательности включает следующие поля:

<Номер школы> <Баллы ЕГЭ> <Фамилия> <Инициалы>

Баллы ЕГЭ представляют собой три целых числа в диапазоне от 0 до 100, которые отделяются друг от друга одним пробелом. Для каждой школы и каждого предмета определить количество учащихся, набравших не менее 50 баллов по этому предмету (вначале выводится номер школы, затем три числа – количество учащихся этой школы, набравших требуемое число баллов по математике, русскому языку и информатике; некоторые из чисел могут быть равны 0). Сведения о каждой школе выводить на новой строке и упорядочивать по возрастанию номера школы.

Указание. Сохранить значения баллов для каждого учащегося в целочисленном массиве *b*; для обработки всех элементов этого массива (при подсчете требуемого в задаче числа учащихся по каждому предмету) использовать вспомогательную последовательность (0, 1, 2), определив ее с помощью метода *Range* и применив к ней метод *Select* с лямбда-выражением *i => ee.Count(e => e.b[i] >= 50)*, где *ee* – последовательность всех учащихся определенной школы, полученная в результате группи-

ровки. Для формирования строки со списком найденных чисел использовать метод `Aggregate`.

LinqObj60. Исходная последовательность содержит сведения о результатах сдачи учащимися ЕГЭ по математике, русскому языку и информатике (в указанном порядке). Каждый элемент последовательности включает следующие поля:

<Фамилия> <Инициалы> <Баллы ЕГЭ> <Номер школы>

Баллы ЕГЭ представляют собой три целых числа в диапазоне от 0 до 100, которые отделяются друг от друга одним пробелом. Для каждой школы и каждого предмета найти среднее значение балла ЕГЭ, набранного учащимися этой школы (среднее значение является целым числом – результатом *деления нацело* суммы баллов всех учащихся на количество учащихся). Сведения о каждой школе выводить на отдельной строке, указывая номер школы и средние баллы по математике, русскому языку и информатике. Данные упорядочивать по убыванию номера школы.

Указание. Ср. с `LinqObj59`.

LinqObj61. Исходная последовательность содержит сведения об оценках учащихся по трем предметам: алгебре, геометрии и информатике. Каждый элемент последовательности содержит данные об одной оценке и включает следующие поля:

<Фамилия> <Инициалы> <Класс> <Название предмета> <Оценка>

Полных однофамильцев (с совпадающей фамилией и инициалами) среди учащихся нет. Класс задается целым числом, оценка – целое число в диапазоне 2–5. Название предмета указывается с заглавной буквы. Для каждого учащегося определить среднюю оценку по каждому предмету и вывести ее с двумя дробными знаками (если по какому-либо предмету учащийся не получил ни одной оценки, то вывести для этого предмета 0.00). Сведения о каждом учащемся выводить на отдельной строке, указывая фамилию, инициалы и средние оценки по алгебре, геометрии и информатике. Данные располагать в алфавитном порядке фамилий и инициалов.

Примечание. Решение данной задачи приведено в п. 6.2.

LinqObj62. Исходная последовательность содержит сведения об оценках учащихся по трем предметам: алгебре, геометрии и информ-

матике. Каждый элемент последовательности содержит данные об одной оценке и включает следующие поля:

<Класс> <Фамилия> <Инициалы> <Оценка> <Название предмета>

Полных однофамильцев (с совпадающей фамилией и инициалами) среди учащихся нет. Класс задается целым числом, оценка – целое число в диапазоне 2–5. Название предмета указывается с заглавной буквы. Для каждого учащегося определить количество оценок по каждому предмету (если по какому-либо предмету учащийся не получил ни одной оценки, то вывести для этого предмета число 0). Сведения о каждом учащемся выводить на отдельной строке, указывая класс, фамилию, инициалы и количество оценок по алгебре, геометрии и информатике. Данные располагать в порядке возрастания номера класса, а для одинаковых классов – в алфавитном порядке фамилий и инициалов.

Указание. Ср. с LinqObj61.

LinqObj63. Исходная последовательность содержит сведения об оценках учащихся по трем предметам: алгебре, геометрии и информатике. Каждый элемент последовательности содержит данные об одной оценке и включает следующие поля:

<Название предмета> <Фамилия> <Инициалы> <Класс> <Оценка>

Полных однофамильцев (с совпадающей фамилией и инициалами) среди учащихся нет. Класс задается целым числом, оценка – целое число в диапазоне 2–5. Название предмета указывается с заглавной буквы. Вывести сведения об учащихся, имеющих по алгебре среднюю оценку не более 4; фамилию, инициалы, номер класса и среднюю оценку по алгебре (выводится с двумя дробными знаками). Для учащихся, не имеющих ни одной оценки по алгебре, считать среднюю оценку равной 0.00. Сведения о каждом учащемся выводить на отдельной строке и располагать в алфавитном порядке их фамилий и инициалов. Если ни один из учащихся не удовлетворяет указанным условиям, то записать в результирующий файл текст «Требуемые учащиеся не найдены».

LinqObj64. Исходная последовательность содержит сведения об оценках учащихся по трем предметам: алгебре, геометрии и информатике. Каждый элемент последовательности содержит данные об одной оценке и включает следующие поля:

<Класс> <Фамилия> <Инициалы> <Название предмета> <Оценка>

Полных однофамильцев (с совпадающей фамилией и инициалами) среди учащихся нет. Класс задается целым числом, оценка – целое число в диапазоне 2–5. Название предмета указывается с заглавной буквы. Вывести сведения об учащихся, имеющих по информатике среднюю оценку не менее 4: номер класса, фамилию, инициалы и среднюю оценку по информатике (выводится с двумя дробными знаками). Сведения о каждом учащемся выводить на отдельной строке и располагать в порядке возрастания классов, а для одинаковых классов – в алфавитном порядке фамилий и инициалов. Если ни один из учащихся не удовлетворяет указанным условиям, то записать в результирующий файл текст «Требуемые учащиеся не найдены».

LinqObj65. Дана строка S – название одного из трех предметов: алгебры, геометрии или информатики. Исходная последовательность содержит сведения об оценках учащихся по этим трем предметам. Каждый элемент последовательности содержит данные об одной оценке и включает следующие поля:

<Фамилия> <Инициалы> <Название предмета> <Оценка> <Класс>

Полных однофамильцев (с совпадающей фамилией и инициалами) среди учащихся нет. Класс задается целым числом, оценка – целое число в диапазоне 2–5. Название предмета указывается с заглавной буквы. Для каждого класса, присутствующего в наборе исходных данных, определить число учащихся, имеющих по предмету S среднюю оценку не более 3.5 или не имеющих ни одной оценки по этому предмету. Сведения о каждом классе выводить на отдельной строке, указывая число найденных учащихся (число может быть равно 0) и номер класса. Данные упорядочивать по возрастанию числа учащихся, а для совпадающих чисел – по убыванию номера класса.

LinqObj66. Дана строка S – название одного из трех предметов: алгебры, геометрии или информатики. Исходная последовательность содержит сведения об оценках учащихся по этим трем предметам. Каждый элемент последовательности содержит данные об одной оценке и включает следующие поля:

<Название предмета> <Фамилия> <Инициалы> <Оценка> <Класс>

Полных однофамильцев (с совпадающей фамилией и инициалами) среди учащихся нет. Класс задается целым числом, оценка – целое число в диапазоне 2–5. Название предмета указывается с заглавной буквы. Для каждого класса, присутствующего в наборе исходных данных, определить число учащихся, имеющих по предмету S среднюю оценку не менее 3.5 и при этом не получивших ни одной двойки по этому предмету. Сведения о каждом классе вывести на отдельной строке, указывая номер класса и число найденных учащихся (число может быть равно 0). Данные упорядочивать по возрастанию номера класса.

LinqObj67. Исходная последовательность содержит сведения об оценках учащихся по трем предметам: алгебре, геометрии и информатике. Каждый элемент последовательности содержит данные об одной оценке и включает следующие поля:

| | | | | |
|---------|---------------------|-----------|------------|----------|
| <Класс> | <Название предмета> | <Фамилия> | <Инициалы> | <Оценка> |
|---------|---------------------|-----------|------------|----------|

Полных однофамильцев (с совпадающей фамилией и инициалами) среди учащихся нет. Класс задается целым числом, оценка – целое число в диапазоне 2–5. Название предмета указывается с заглавной буквы. Найти всех *двоечников* – учащихся, получивших хотя бы одну двойку по какому-либо предмету. Вывести сведения о каждом из двоечников: номер класса, фамилию, инициалы и полученное число двоек. Сведения о каждом двоечнике выводить на отдельной строке и располагать по убыванию классов, а для одинаковых классов – в алфавитном порядке фамилий и инициалов. Если в наборе исходных данных нет ни одной двойки, то записать в результирующий файл текст «Требуемые учащиеся не найдены».

LinqObj68. Исходная последовательность содержит сведения об оценках учащихся по трем предметам: алгебре, геометрии и информатике. Каждый элемент последовательности содержит данные об одной оценке и включает следующие поля:

| | | | | |
|---------|----------|-----------|------------|---------------------|
| <Класс> | <Оценка> | <Фамилия> | <Инициалы> | <Название предмета> |
|---------|----------|-----------|------------|---------------------|

Полных однофамильцев (с совпадающей фамилией и инициалами) среди учащихся нет. Класс задается целым числом, оценка – целое число в диапазоне 2–5. Название предмета указывается с заглавной буквы. Найти всех *хорошистов* – учащихся, не получивших ни одной двойки и тройки, но имеющих хотя бы одну четверку по какому-либо предмету. Вывести сведения о каждом хорошисте:

полученное число четверок, фамилию, инициалы и номер класса. Сведения о каждом учащемся выводить на отдельной строке и располагать по возрастанию количества четверок, а при их равенстве – в алфавитном порядке фамилий и инициалов. Если в наборе исходных данных нет ни одного учащегося, удовлетворяющего указанным условиям, то записать в результирующий файл текст «Требуемые учащиеся не найдены».

LinqObj69. Исходная последовательность содержит сведения об оценках учащихся по трем предметам: алгебре, геометрии и информатике. Каждый элемент последовательности содержит данные об одной оценке и включает следующие поля:

| | | | | |
|---------|-----------|------------|---------------------|----------|
| <Класс> | <Фамилия> | <Инициалы> | <Название предмета> | <Оценка> |
|---------|-----------|------------|---------------------|----------|

Полных однофамильцев (с совпадающей фамилией и инициалами) среди учащихся нет. Класс задается целым числом, оценка – целое число в диапазоне 2–5. Название предмета указывается с заглавной буквы. Для каждого класса найти *зlostных двоечников* – учащихся, получивших в данном классе максимальное суммарное число двоек по всем предметам (число не должно быть нулевым). Вывести сведения о каждом из зlostных двоечников: фамилию, инициалы, номер класса и полученное число двоек. Сведения о каждом двоечнике выводить на отдельной строке и располагать в алфавитном порядке их фамилий и инициалов (сортировку по классам не проводить). Если в наборе исходных данных нет ни одной двойки, то записать в результирующий файл текст «Требуемые учащиеся не найдены».

Указание. Ср. с LinqObj12. В данном случае в элементы сгруппированной по классам последовательности следует включить вспомогательное поле `maxcnt2`, равное максимальному количеству двоек для учащихся данного класса, и последовательность `pupils` учащихся этого класса с полями `fam` (фамилия и инициалы), `cls` (номер класса) и `cnt2` (количество двоек). Затем из сгруппированной последовательности надо исключить классы, для которых значение `maxcnt2` равно 0, а полученную последовательность классов преобразовать в плоскую последовательность учащихся (зlostных двоечников), используя метод `SelectMany` со следующим лямбда-выражением:

```
e => e.pupils.Where(e1 => e1.cnt2 == e.maxcnt2)
```

LinqObj70. Исходная последовательность содержит сведения об оценках учащихся по трем предметам: алгебре, геометрии и информ-

матике. Каждый элемент последовательности содержит данные об одной оценке и включает следующие поля:

<Оценка> <Класс> <Фамилия> <Инициалы> <Название предмета>

Полных однофамильцев (с совпадающей фамилией и инициалами) среди учащихся нет. Класс задается целым числом, оценка – целое число в диапазоне 2–5. Название предмета указывается с заглавной буквы. Для каждого класса найти *лучших учеников* – учащихся, получивших в данном классе максимальное суммарное число пятерок по всем предметам (число не должно быть нулевым). При поиске лучших учеников (в частности, при определении максимального суммарного числа пятерок) не следует учитывать учащихся, получивших хотя бы одну двойку или тройку. Вывести сведения о каждом из лучших учеников: номер класса, фамилию, инициалы и полученное число пятерок. Сведения о каждом учащемся выводить на отдельной строке и располагать по возрастанию классов, а для одинаковых классов – в алфавитном порядке фамилий и инициалов. Если в наборе исходных данных нет ни одного учащегося, удовлетворяющего указанным условиям, то записать в результирующий файл текст «Требуемые учащиеся не найдены».

Указание. Ср. с LinqObj69.

3.2. Обработка нескольких взаимосвязанных последовательностей

В каждом задании данной подгруппы требуется обработать несколько (от двух до четырех) последовательностей из следующего набора:

- A: сведения о потребителях, содержащие поля «Код потребителя», «Год рождения», «Улица проживания»;
- B: сведения о товарах, содержащие поля «Артикул товара», «Категория», «Страна-производитель»;
- C: скидки для потребителей в различных магазинах, содержащие поля «Код потребителя», «Название магазина», «Скидка (в процентах)»;
- D: цены товаров в различных магазинах, содержащие поля «Артикул товара», «Название магазина», «Цена (в рублях)»;
- E: сведения о покупках потребителей в различных магазинах, содержащие поля «Код потребителя», «Артикул товара», «Название магазина».

Порядок следования полей для элементов каждой последовательности определяется в формулировке задания.

В последовательности *A* все элементы имеют различные значения поля «Код потребителя». В последовательности *B* все элементы имеют различные значения поля «Артикул товара». В последовательности *C* все элементы имеют различные комбинации полей «Код потребителя» и «Название магазина». В последовательности *D* все элементы имеют различные комбинации полей «Артикул товара» и «Название магазина». Последовательность *E* может содержать одинаковые элементы (это соответствует ситуации, при которой один и тот же потребитель приобрел в одном и том же магазине несколько одинаковых товаров).

Все значения кодов потребителей и артикулов товаров, присутствующие в последовательностях *C*, *D* и *E*, обязательно содержатся в последовательностях *A* и *B*. Некоторые значения кодов потребителей и артикулов товаров, присутствующие в последовательностях *A* и *B*, могут отсутствовать в остальных последовательностях. Любая комбинация «магазин–товар», присутствующая в последовательности *E*, обязательно присутствует и в последовательности *D*. Комбинация «потребитель–магазин», присутствующая в последовательности *E*, может отсутствовать в последовательности *C*; это означает, что при покупке указанного товара в данном магазине потребитель не имел скидки (то есть скидка была равна 0).

Коды потребителей, годы рождения, скидки и цены задаются целыми числами; значения скидок лежат в диапазоне от 5 до 50. Прочие поля являются строковыми и не содержат пробелов. Артикулы товаров имеют формат «AAdddd-dddd», где на позициях, помеченных символом «A», располагается какая-либо заглавная латинская буква, а на позициях, помеченных символом «d», – какая-либо цифра.

Если потребитель приобрел товар, имеющий цену p , со скидкой d процентов, то размер скидки на данный товар должен вычисляться по формуле $p \cdot d / 100$, где символ «/» обозначает операцию целочисленного деления (иными словами, при вычислении размера скидки копейки отбрасываются).

Перед выполнением заданий из данного пункта следует ознакомиться с примером решения задачи LinqObj98, приведенным в п. 6.3.

LinqObj71. Даны последовательности *A* и *C*, включающие следующие поля:

A: <Код потребителя> <Год рождения> <Улица проживания>
C: <Код потребителя> <Скидка (в процентах)> <Название магазина>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждого магазина определить потребителей, имеющих максимальную скидку в этом магазине (вначале выводится название магазина, затем код потребителя, его год рождения и значение максимальной скидки). Если для некоторого магазина имеется несколько потребителей с максимальной скидкой, то вывести данные о потребителе с минимальным кодом. Сведения о каждом магазине выводить на новой строке и упорядочивать по названиям магазинов в алфавитном порядке.

Примечание. Решение данной задачи приведено в п. 9.4.

LinqObj72. Даны последовательности A и C, включающие следующие поля:

A: <Код потребителя> <Улица проживания> <Год рождения>
C: <Скидка (в процентах)> <Код потребителя> <Название магазина>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждого потребителя, указанного в A, определить количество магазинов, в которых ему предоставляется скидка (вначале выводится количество магазинов, затем код потребителя, потом его улица проживания). Если у некоторого потребителя нет скидки ни в одном магазине, то для него выводится количество магазинов, равное 0. Сведения о каждом потребителе выводить на новой строке и упорядочивать по возрастанию количества магазинов, а при равном количестве – по возрастанию кодов потребителей.

LinqObj73. Даны последовательности A и C, включающие следующие поля:

A: <Год рождения> <Код потребителя> <Улица проживания>
C: <Код потребителя> <Название магазина> <Скидка (в процентах)>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждого магазина и каждой улицы определить количество потребителей, живущих на этой улице и имеющих скидку в этом магазине (вначале выводится название магазина, затем название улицы, потом количество потребителей со скидкой). Если для некоторой пары «магазин–улица» потребители со скидкой не найде-

ны, то данные об этой паре не выводятся. Сведения о каждой паре «магазин–улица» выводить на новой строке и упорядочивать по названиям магазинов в алфавитном порядке, а для одинаковых названий магазинов – по названиям улиц (также в алфавитном порядке).

LinqObj74. Даны последовательности *A* и *C*, включающие следующие поля:

A: <Улица проживания> <Код потребителя> <Год рождения>
C: <Название магазина> <Скидка (в процентах)> <Код потребителя>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждого магазина и каждого года рождения из *A* определить среднюю скидку в данном магазине (в процентах) для потребителей с этим годом рождения (вначале выводится название магазина, затем год рождения, потом значение средней скидки в процентах). Дробная часть найденного среднего значения отбрасывается, скидка выводится как целое число. При вычислении средней скидки учитываются только потребители данного года рождения, имеющие скидку в данном магазине. Если таких потребителей для данного магазина нет, то для этой пары «магазин–год» в качестве средней скидки выводится 0. Сведения о каждой паре «магазин–год» выводить на новой строке и упорядочивать по названиям магазинов в алфавитном порядке, а для одинаковых названий магазинов – по возрастанию номеров года.

LinqObj75. Даны последовательности *B* и *D*, включающие следующие поля:

B: <Артикул товара> <Категория> <Страна-производитель>
D: <Название магазина> <Артикул товара> <Цена (в рублях)>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждого магазина и каждой категории товаров определить количество различных артикулов товаров данной категории, имеющих в данном магазине (вначале выводится название магазина, затем категория, потом количество различных артикулов товаров). Если для некоторого магазина товары данной категории отсутствуют, то информация о соответствующей паре «магазин–категория» не выводится. Сведения о каждой паре «магазин–категория» выводить на новой строке и упорядочивать по названиям магазинов в алфавитном порядке, а для одинаковых названий магазинов – по названиям категорий (также в алфавитном порядке).

LinqObj76. Даны последовательности B и D , включающие следующие поля:

B : <Страна-производитель> <Категория> <Артикул товара>
 D : <Артикул товара> <Название магазина> <Цена (в рублях)>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждой страны-производителя определить количество магазинов, предлагающих товары, произведенные в этой стране, а также минимальную цену для товаров из данной страны по всем магазинам (вначале выводится количество магазинов, затем страна, потом минимальная цена). Если для некоторой страны не найдено ни одного товара, представленного в каком-либо магазине, то количество магазинов и минимальная цена полагаются равными 0. Сведения о каждой стране выводить на новой строке и упорядочивать по возрастанию количества магазинов, а в случае одинакового количества – по названиям стран в алфавитном порядке.

LinqObj77. Даны последовательности B и D , включающие следующие поля:

B : <Категория> <Артикул товара> <Страна-производитель>
 D : <Артикул товара> <Цена (в рублях)> <Название магазина>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждой категории товаров определить количество магазинов, предлагающих товары данной категории, а также количество стран, в которых произведены товары данной категории, представленные в магазинах (вначале выводится количество магазинов, затем название категории, потом количество стран). Если для некоторой категории не найдено ни одного товара, представленного в каком-либо магазине, то информация о данной категории не выводится. Сведения о каждой категории выводить на новой строке и упорядочивать по убыванию количества магазинов, а в случае одинакового количества – по названиям категорий в алфавитном порядке.

LinqObj78. Даны последовательности D и E , включающие следующие поля:

D : <Цена (в рублях)> <Артикул товара> <Название магазина>
 E : <Код потребителя> <Название магазина> <Артикул товара>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждого товара определить количество покупок данного товара и максимальную цену покупки (вначале выво-

дится количество покупок, затем артикул товара, потом максимальная цена покупки). Если некоторый товар ни разу не был продан, то информация об этом товаре не выводится. Сведения о каждом товаре выводить на новой строке и упорядочивать по возрастанию количества покупок, а в случае одинакового количества – по артикулам товаров в алфавитном порядке.

LinqObj79. Даны последовательности D и E , включающие следующие поля:

D: <Название магазина> <Цена (в рублях)> <Артикул товара>
E: <Код потребителя> <Артикул товара> <Название магазина>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждого магазина и каждого потребителя определить суммарную стоимость покупок, сделанных этим потребителем в данном магазине (вначале выводится название магазина, затем код потребителя, потом суммарная стоимость покупок). Если потребитель не приобрел ни одного товара в некотором магазине, то информация о соответствующей паре «магазин–потребитель» не выводится. Сведения о каждой паре «магазин–потребитель» выводить на новой строке и упорядочивать по названиям магазинов в алфавитном порядке, а в случае одинаковых названий – по возрастанию кодов потребителей.

LinqObj80. Даны последовательности D и E , включающие следующие поля:

D: <Цена (в рублях)> <Название магазина> <Артикул товара>
E: <Артикул товара> <Название магазина> <Код потребителя>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждой пары «магазин–товар», указанной в D , определить суммарную стоимость продаж этого товара в данном магазине (вначале выводится название магазина, затем артикул товара, потом суммарная стоимость его продаж). Если товар ни разу не был продан в некотором магазине, то для соответствующей пары «магазин–товар» в качестве суммарной стоимости выводится 0. Сведения о каждой паре «магазин–товар» выводить на новой строке и упорядочивать по названиям магазинов в алфавитном порядке, а в случае одинаковых названий – по артикулам товаров (также в алфавитном порядке).

LinqObj81. Даны последовательности B , D и E , включающие следующие поля:

B: <Артикул товара> <Страна-производитель> <Категория>
D: <Название магазина> <Артикул товара> <Цена (в рублях)>
E: <Название магазина> <Код потребителя> <Артикул товара>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждой страны-производителя определить количество приобретенных товаров из данной страны и суммарную стоимость приобретенных товаров (вначале выводится название страны, затем количество товаров, потом суммарная стоимость). Если сведения о проданных товарах для некоторой страны-производителя отсутствуют, то информация об этой стране не выводится. Сведения о каждой стране выводить на новой строке и упорядочивать по названиям стран в алфавитном порядке.

LinqObj82. Даны последовательности *B*, *D* и *E*, включающие следующие поля:

B: <Категория> <Страна-производитель> <Артикул товара>
D: <Цена (в рублях)> <Артикул товара> <Название магазина>
E: <Артикул товара> <Код потребителя> <Название магазина>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждого потребителя определить количество категорий приобретенных им товаров и максимальную цену одной его покупки (вначале выводится количество категорий товаров, затем код потребителя, потом максимальная цена покупки). Сведения о каждом потребителе выводить на новой строке и упорядочивать по убыванию количества категорий, а при совпадении количества категорий – по возрастанию кодов потребителей.

LinqObj83. Даны последовательности *B*, *D* и *E*, включающие следующие поля:

B: <Страна-производитель> <Артикул товара> <Категория>
D: <Цена (в рублях)> <Название магазина> <Артикул товара>
E: <Название магазина> <Артикул товара> <Код потребителя>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждого магазина, указанного в *E*, и каждой страны-производителя определить суммарную стоимость товаров из данной страны, проданных в данном магазине (вначале выводится название магазина, затем название страны, потом суммарная стоимость). Если для некоторой пары «магазин–страна» отсутствует информация о проданных товарах, то в качестве суммарной стоимости выводится 0. Сведения о каждой паре «магазин–страна» вы-

водить на новой строке и упорядочивать по названиям магазинов в алфавитном порядке, а для одинаковых названий магазинов – по названиям стран (также в алфавитном порядке).

LinqObj84. Даны последовательности *C*, *D* и *E*, включающие следующие поля:

| | | | |
|----|------------------------|---------------------|-------------------|
| C: | <Скидка (в процентах)> | <Название магазина> | <Код потребителя> |
| D: | <Артикул товара> | <Название магазина> | <Цена (в рублях)> |
| E: | <Артикул товара> | <Название магазина> | <Код потребителя> |

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждого магазина и каждого товара определить количество покупок этого товара со скидкой в данном магазине и суммарную стоимость этих покупок с учетом скидки (вначале выводится название магазина, затем артикул товара, потом количество покупок со скидкой и их суммарная стоимость). При вычислении размера скидки на товар копейки отбрасываются. Если для некоторой пары «магазин–товар» не найдено ни одной покупки со скидкой, то информация о данной паре не выводится. Если не найдено ни одной подходящей пары «магазин–товар», то записать в результирующий файл текст «Требуемые данные не найдены». Сведения о каждой паре «магазин–товар» выводить на новой строке и упорядочивать по названиям магазинов в алфавитном порядке, а для одинаковых названий – по артикулам товаров (также в алфавитном порядке).

LinqObj85. Даны последовательности *C*, *D* и *E*, включающие следующие поля:

| | | | |
|----|---------------------|-------------------|------------------------|
| C: | <Название магазина> | <Код потребителя> | <Скидка (в процентах)> |
| D: | <Артикул товара> | <Цена (в рублях)> | <Название магазина> |
| E: | <Артикул товара> | <Код потребителя> | <Название магазина> |

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждой пары «потребитель–магазин», указанной в *E*, определить суммарный размер скидок на все товары, приобретенные этим потребителем в данном магазине (вначале выводится код потребителя, затем название магазина, потом суммарный размер скидки). При вычислении размера скидки на каждый приобретенный товар копейки отбрасываются. Если потребитель приобретал товары в некотором магазине без скидки, то информация о соответствующей паре «потребитель–магазин» не выводится. Если не найдено ни одной подходящей пары «потребитель–мага-

зин», то записать в результирующий файл текст «Требуемые данные не найдены». Сведения о каждой паре «потребитель–магазин» выводить на новой строке и упорядочивать по возрастанию кодов потребителей, а для одинаковых кодов – по названиям магазинов в алфавитном порядке.

LinqObj86. Даны последовательности C , D и E , включающие следующие поля:

| | | | |
|----|------------------------|---------------------|---------------------|
| C: | <Скидка (в процентах)> | <Код потребителя> | <Название магазина> |
| D: | <Название магазина> | <Цена (в рублях)> | <Артикул товара> |
| E: | <Код потребителя> | <Название магазина> | <Артикул товара> |

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждой пары «товар–магазин», указанной в E , определить максимальный размер скидки на этот товар при его приобретении в данном магазине (вначале выводится артикул товара, затем название магазина, потом максимальный размер скидки). При вычислении размера скидки на товар копейки отбрасываются. Если все продажи товара в некотором магазине проводились без скидки, то в качестве максимального размера скидки для данной пары выводится 0. Сведения о каждой паре «товар–магазин» выводить на новой строке и упорядочивать по артикулам товаров в алфавитном порядке, а для одинаковых артикулов – по названиям магазинов (также в алфавитном порядке).

LinqObj87. Даны последовательности A , D и E , включающие следующие поля:

| | | | |
|----|---------------------|---------------------|-------------------|
| A: | <Год рождения> | <Улица проживания> | <Код потребителя> |
| D: | <Артикул товара> | <Название магазина> | <Цена (в рублях)> |
| E: | <Название магазина> | <Код потребителя> | <Артикул товара> |

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждой улицы проживания и каждого магазина, указанного в E , определить суммарную стоимость покупок в данном магазине, сделанных всеми потребителями, живущими на данной улице (вначале выводится название улицы, затем название магазина, потом суммарная стоимость покупок). Если для некоторой пары «улица–магазин» отсутствует информация о проданных товарах, то в качестве суммарной стоимости выводится 0. Сведения о каждой паре «улица–магазин» выводить на новой строке и упорядочивать по названиям улиц в алфавитном порядке, а для одинаковых названий улиц – по названиям магазинов (также в алфавитном порядке).

LinqObj88. Даны последовательности A , D и E , включающие следующие поля:

A: <Улица проживания> <Год рождения> <Код потребителя>
D: <Артикул товара> <Цена (в рублях)> <Название магазина>
E: <Название магазина> <Артикул товара> <Код потребителя>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждого года рождения, указанного в A , и каждого товара, указанного в E , определить суммарную стоимость покупок данного товара, сделанных всеми потребителями с данным годом рождения (вначале выводится год рождения, затем артикул товара, потом суммарная стоимость покупок). Если для некоторой пары «год–товар» отсутствуют сведения о проданных товарах, то информация об этой паре не выводится. Сведения о каждой паре «год–товар» выводить на новой строке и упорядочивать по убыванию номеров года, а для одинаковых номеров – по артикулам товаров в алфавитном порядке.

LinqObj89. Даны последовательности A , D и E , включающие следующие поля:

A: <Код потребителя> <Год рождения> <Улица проживания>
D: <Название магазина> <Цена (в рублях)> <Артикул товара>
E: <Код потребителя> <Артикул товара> <Название магазина>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждого магазина, указанного в E , определить потребителя с наименьшим годом рождения, купившего один или более товаров в данном магазине (вначале выводится название магазина, затем код потребителя, год его рождения и суммарная стоимость товаров, приобретенных потребителем в данном магазине). Если для некоторого магазина имеется несколько покупателей с наименьшим годом рождения, то выводятся данные обо всех таких покупателях. Сведения о каждой паре «магазин–потребитель» выводить на новой строке и упорядочивать по названиям магазинов в алфавитном порядке, а для одинаковых названий магазинов – по возрастанию кодов потребителей.

LinqObj90. Даны последовательности A , B и E , включающие следующие поля:

A: <Улица проживания> <Код потребителя> <Год рождения>
B: <Страна-производитель> <Категория> <Артикул товара>
E: <Артикул товара> <Код потребителя> <Название магазина>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждого года рождения из *A* определить страну, в которой было произведено максимальное количество товаров, приобретенных потребителями этого года рождения (вначале выводится год, затем название страны, потом максимальное количество покупок). Если для некоторой пары «год–страна» отсутствует информация о проданных товарах, то эта пара не обрабатывается (в частности, если потребители некоторого года рождения не сделали ни одной покупки, то информация об этом годе не выводится). Если для какого-либо года рождения имеется несколько стран с наибольшим числом приобретенных товаров, то выводятся данные о первой из таких стран (в алфавитном порядке). Сведения о каждом годе выводить на новой строке и упорядочивать по убыванию номера года.

LinqObj91. Даны последовательности *A*, *B* и *E*, включающие следующие поля:

A: <Улица проживания> <Год рождения> <Код потребителя>
B: <Артикул товара> <Страна-производитель> <Категория>
E: <Название магазина> <Артикул товара> <Код потребителя>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждой улицы и каждой категории товаров определить количество стран, в которых были произведены товары данной категории, купленные потребителями, живущими на этой улице (вначале выводится название улицы, затем название категории, потом количество стран). Если для какой-либо категории отсутствует информация о товарах, проданных жителям некоторой улицы, то информация о соответствующей паре «улица–категория» не выводится. Сведения о каждой паре «улица–категория» выводить на новой строке и упорядочивать по названиям улиц в алфавитном порядке, а для одинаковых улиц – по названиям категорий (также в алфавитном порядке).

LinqObj92. Даны последовательности *A*, *B* и *E*, включающие следующие поля:

A: <Год рождения> <Улица проживания> <Код потребителя>
B: <Страна-производитель> <Артикул товара> <Категория>
E: <Артикул товара> <Название магазина> <Код потребителя>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждой категории товаров определить улицу с максимальным суммарным количеством товаров данной кате-

гории, приобретенных жителями этой улицы (вначале выводится название категории, затем название улицы, потом максимальное суммарное количество покупок). Если для какой-либо категории отсутствует информация о товарах, проданных жителям некоторой улицы, то суммарное количество покупок для соответствующей пары «категория–улица» считается равным 0 (при этом возможна ситуация, когда наибольшее количество покупок для какой-либо категории равно 0). Если для некоторой категории имеется несколько улиц с наибольшим количеством покупок, то выводятся данные обо всех таких улицах. Сведения о каждой паре «категория–улица» выводить на новой строке и упорядочивать по названиям категорий в алфавитном порядке, а для одинаковых категорий – по названиям улиц (также в алфавитном порядке).

LinqObj93. Даны последовательности *A*, *B*, *C* и *E*, включающие следующие поля:

A: <Код потребителя> <Улица проживания> <Год рождения>
B: <Категория> <Страна-производитель> <Артикул товара>
C: <Название магазина> <Код потребителя> <Скидка (в процентах)>
E: <Код потребителя> <Артикул товара> <Название магазина>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждой страны-производителя и улицы проживания определить максимальный размер скидки (в процентах) для изделий, произведенных в данной стране и приобретенных потребителями, живущими на данной улице (вначале выводится название страны, затем название улицы, потом максимальный размер скидки). Если для некоторой пары «страна–улица» все товары были приобретены без скидки, то в качестве максимального размера скидки выводится 0. Если для некоторой пары «страна–улица» отсутствует информация о приобретенных товарах, то информация о данной паре не выводится. Сведения о каждой паре «страна–улица» выводить на новой строке и упорядочивать по названиям стран в алфавитном порядке, а для одинаковых названий – по названиям улиц (также в алфавитном порядке).

LinqObj94. Даны последовательности *A*, *B*, *C* и *E*, включающие следующие поля:

A: <Год рождения> <Код потребителя> <Улица проживания>
B: <Артикул товара> <Категория> <Страна-производитель>
C: <Код потребителя> <Название магазина> <Скидка (в процентах)>
E: <Название магазина> <Код потребителя> <Артикул товара>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждого магазина, указанного в *E*, и каждой категории товаров определить минимальное значение года рождения среди тех потребителей, которые приобрели в данном магазине один или более товаров данной категории, и количество товаров данной категории, приобретенных со скидкой в данном магазине потребителями этого года рождения (вначале выводится название магазина, затем название категории, потом номер минимального года рождения и количество товаров, приобретенных со скидкой). Если для некоторой пары «магазин–категория» информация о проданных товарах отсутствует, то данные об этой паре не выводятся. Если для некоторой пары «магазин–категория» покупатели с минимальным годом рождения приобрели все товары без скидки, то в качестве значения количества товаров, приобретенных со скидкой, выводится 0. Сведения о каждой паре «магазин–категория» выводить на новой строке и упорядочивать по названиям магазинов в алфавитном порядке, а для одинаковых названий магазинов – по названиям категорий (также в алфавитном порядке).

LinqObj95. Даны последовательности *A*, *C*, *D* и *E*, включающие следующие поля:

A: <Код потребителя> <Улица проживания> <Год рождения>
C: <Название магазина> <Скидка (в процентах)> <Код потребителя>
D: <Название магазина> <Артикул товара> <Цена (в рублях)>
E: <Код потребителя> <Название магазина> <Артикул товара>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждого артикула товара, указанного в *E*, и каждой улицы проживания определить суммарный размер скидки на изделия данного артикула, приобретенные потребителями, живущими на данной улице (вначале выводится артикул товара, затем название улицы, потом суммарный размер скидки). При вычислении размера скидки на товар копейки отбрасываются. Если на проданный товар скидка отсутствует, то ее размер полагается равным 0. Если для некоторой пары «товар–улица» отсутствует информация о приобретенных товарах, то данные об этой паре не выводятся. Если для некоторой пары «товар–улица» все изделия были приобретены без скидок, то в качестве суммарной скидки для этой пары выводится 0. Сведения о каждой паре «товар–улица» выводить на новой строке и упорядочивать по артикулам товаров в алфавитном порядке, а для одинаковых артикулов – по названиям улиц (также в алфавитном порядке).

LinqObj96. Даны последовательности *A*, *C*, *D* и *E*, включающие следующие поля:

A: <Улица проживания> <Год рождения> <Код потребителя>
C: <Код потребителя> <Скидка (в процентах)> <Название магазина>
D: <Цена (в рублях)> <Артикул товара> <Название магазина>
E: <Название магазина> <Артикул товара> <Код потребителя>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждого года рождения, указанного в *A*, и каждого магазина, указанного в *E*, определить суммарную стоимость всех товаров (с учетом скидки), проданных в данном магазине потребителям данного года рождения (вначале выводится номер года, затем название магазина, потом суммарная стоимость проданных товаров с учетом скидки). При вычислении размера скидки на товар копейки отбрасываются. Если на проданный товар скидка отсутствует, то ее размер полагается равным 0. Если для некоторой пары «год–магазин» отсутствует информация о проданных товарах, то данные об этой паре не выводятся. Сведения о каждой паре «год–магазин» выводить на новой строке и упорядочивать по возрастанию номеров года, а для одинаковых номеров – по названиям магазинов в алфавитном порядке.

LinqObj97. Даны последовательности *B*, *C*, *D* и *E*, включающие следующие поля:

B: <Категория> <Артикул товара> <Страна-производитель>
C: <Скидка (в процентах)> <Название магазина> <Код потребителя>
D: <Цена (в рублях)> <Название магазина> <Артикул товара>
E: <Название магазина> <Код потребителя> <Артикул товара>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждой страны-производителя и каждого потребителя определить суммарную стоимость (с учетом скидки) всех товаров, произведенных в данной стране и приобретенных этим потребителем (вначале выводится название страны, затем код потребителя, потом суммарная стоимость с учетом скидки). При вычислении размера скидки на товар копейки отбрасываются. Если на проданный товар скидка отсутствует, то ее размер полагается равным 0. Если для некоторой пары «страна–потребитель» данные о покупках отсутствуют, то информация об этой паре не выводится. Сведения о каждой паре «страна–потребитель» выводить на новой строке и упорядочивать по названиям стран в алфавитном порядке, а для одинаковых названий стран – по возрастанию кодов потребителей.

LinqObj98. Даны последовательности B , C , D и E , включающие следующие поля:

B : <Страна-производитель> <Артикул товара> <Категория>
 C : <Скидка (в процентах)> <Код потребителя> <Название магазина>
 D : <Артикул товара> <Цена (в рублях)> <Название магазина>
 E : <Код потребителя> <Артикул товара> <Название магазина>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждой категории товаров и каждого магазина, указанного в E , определить суммарный размер скидки на все товары данной категории, проданные в данном магазине (вначале выводится название категории, затем название магазина, потом суммарная скидка). При вычислении размера скидки на товар копейки отбрасываются. Если на проданный товар скидка отсутствует, то ее размер полагается равным 0. Если для некоторой категории товаров в каком-либо магазине не было продаж, то суммарная скидка для этой пары «категория–магазин» полагается равной –1. Сведения о каждой паре «категория–магазин» выводить на новой строке и упорядочивать по названиям категорий в алфавитном порядке, а для одинаковых названий категорий – по названиям магазинов (также в алфавитном порядке).

Примечание. Решение данной задачи приведено в п. 6.3.

LinqObj99. Даны последовательности A , B , D и E , включающие следующие поля:

A : <Код потребителя> <Улица проживания> <Год рождения>
 B : <Категория> <Страна-производитель> <Артикул товара>
 D : <Артикул товара> <Название магазина> <Цена (в рублях)>
 E : <Артикул товара> <Название магазина> <Код потребителя>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждой категории товаров и каждой улицы проживания определить магазин, продавший товар данной категории по минимальной цене потребителю, живущему на данной улице (вначале выводится название категории, затем название улицы, потом название магазина и минимальная цена товара). Если для некоторой пары «категория–улица» отсутствует информация о проданных товарах, то данные об этой паре не выводятся. Если для некоторой пары «категория–улица» имеется несколько магазинов, продавших товар по минимальной цене, то выводятся данные обо всех таких магазинах. Сведения о каждой тройке «категория–ули-

ца–магазин» выводить на новой строке и упорядочивать по названиям категорий в алфавитном порядке, для одинаковых названий категорий – по названиям улиц, а для одинаковых названий улиц – по названиям магазинов (также в алфавитном порядке).

LinqObj100. Даны последовательности *A*, *B*, *D* и *E*, включающие следующие поля:

A: <Улица проживания> <Код потребителя> <Год рождения>
B: <Артикул товара> <Страна-производитель> <Категория>
D: <Название магазина> <Цена (в рублях)> <Артикул товара>
E: <Артикул товара> <Код потребителя> <Название магазина>

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждой страны-производителя и каждого магазина определить потребителя с наибольшим годом рождения, купившего в данном магазине один или более товаров, произведенных в данной стране (вначале выводится название страны, затем название магазина, потом год рождения потребителя, его код, а также суммарная стоимость товаров из данной страны, купленных в этом магазине). Если для некоторой пары «страна–магазин» отсутствует информация о проданных товарах, то данные об этой паре не выводятся. Если для некоторой пары «страна–магазин» имеется несколько потребителей с наибольшим годом рождения, то выводятся данные обо всех таких потребителях. Сведения о каждой тройке «страна–магазин–потребитель» выводить на новой строке и упорядочивать по названиям стран в алфавитном порядке, для одинаковых названий стран – по названиям магазинов (также в алфавитном порядке), а для одинаковых магазинов – по возрастанию кодов потребителей.



Глава 4. Технология LINQ to XML: группа LinqXml

Узлом (node) XML-документа называется любой его компонент, не являющийся *атрибутом* (attribute), в частности *комментарий* (comment), *инструкция обработки* (processing instruction), обычный текст. *Элементом* (element) XML-документа называется именованный компонент, который может содержать другие узлы, а также иметь атрибуты. В объектной модели X-DOM, входящей в состав интерфейса LINQ to XML, с каждым видом компонентов XML-документа связан соответствующий класс: XObject – общий предок всех компонентов, XmlNode – общий предок всех узлов, XText – текстовый узел, то есть узел, представляющий собой обычный текст, XComment – комментарий, XProcessingInstruction – инструкция обработки, XElement – элемент, XAttribute – атрибут. С XML-документом связан класс XDocument.

Если некоторый узел *B* XML-документа содержится внутри некоторого элемента *A*, то элемент *A* называется *предком* (ancestor) узла *B*, а узел *B* – *потомком* (descendant) элемента *A*. Если элемент *A* является ближайшим предком узла *B*, то *B* называется *дочерним узлом* (child node) элемента *A*, а элемент *A* – *родительским элементом* (parent) узла *B*; если при этом узел *B* является элементом, то он называется *дочерним элементом* (child element) элемента *A*. Первый элемент XML-документа называется *корневым элементом* (root); корневой элемент является предком для всех других элементов XML-документа, сам корневой элемент предков не имеет.

Корневой элемент считается элементом *нулевого уровня*, его дочерние узлы/элементы – узлами/элементами *первого уровня*, их дочерние узлы/элементы – узлами/элементами *второго уровня* и т. д.

В XML-документе имеется единственный элемент нулевого уровня (корневой элемент), однако могут присутствовать несколько узлов нулевого уровня (комментариев или инструкций обработки).

Если в задании говорится, что элемент содержит текстовую строку (или число), то это означает, что соответствующая строка (или строковое представление числа) является дочерним текстовым узлом данного элемента.

Во всех заданиях предполагается, что элемент имеет не больше одного дочернего текстового узла, а текстовый узел содержит хотя бы один значащий символ (то есть символ, отличный от пробела и управляющих символов). При сохранении XML-документа следует использовать метод `Save` класса `XDocument` с единственным параметром – именем файла; это обеспечит автоматическое форматирование сохраняемого документа и удаление всех текстовых узлов, не содержащих значащих символов.

Элементы, не содержащие дочерних узлов, могут представляться в двух вариантах: в виде *парных тегов*, между которыми отсутствует текст (`<a>`), и в виде одного *комбинированного тега* (`<a />`). Элементы, не содержащие узлов, могут иметь атрибуты.

Если в условии сказано, что дан XML-документ, то это означает, что дано имя файла, содержащего этот документ. Преобразование XML-документа всегда должно завершаться сохранением преобразованного документа в том же файле, из которого был считан исходный вариант этого документа.

Если в условии упоминаются порядковые номера элементов некоторой последовательности, то предполагается, что нумерация начинается от 1.

В заданиях подгрупп, предшествующих подгруппе «Работа с пространствами имен XML-документа», предполагается, что имена всех элементов и атрибутов XML-документа имеют пустое пространство имен.

Указания, приведенные к некоторым заданиям, следует учитывать и при выполнении последующих заданий текущей подгруппы.

4.1. Создание XML-документа

Во всех заданиях данной подгруппы предполагается, что исходные текстовые файлы содержат текст в кодировке «windows-1251», а все файловые строки являются непустыми. Создаваемые XML-документы также должны иметь кодировку «windows-1251».

Перед выполнением заданий из данного пункта следует ознакомиться с примером решения задачи LinqXml10, приведенным в п. 7.1.

LinqXml1. Даны имена существующего текстового файла и создаваемого XML-документа. Создать XML-документ с корневым элементом `root` и элементами первого уровня `line`, каждый из которых содержит одну строку из исходного файла.

Указание. В конструкторе корневого элемента использовать последовательность объектов `XElement`, полученную методом `Select` из исходного набора строк.

LinqXml2. Даны имена существующего текстового файла и создаваемого XML-документа. Создать XML-документ с корневым элементом `root` и элементами первого уровня, каждый из которых содержит одну строку из исходного файла и имеет имя `line` с приспанным к нему порядковым номером строки (`line1`, `line2` и т. д.).

LinqXml3. Даны имена существующего текстового файла и создаваемого XML-документа. Создать XML-документ с корневым элементом `root` и элементами первого уровня `line`, каждый из которых содержит одну строку из исходного файла. Элемент, содержащий строку с порядковым номером N (1, 2, ...), должен иметь атрибут `num` со значением, равным N .

LinqXml4. Даны имена существующего текстового файла и создаваемого XML-документа. Каждая строка текстового файла содержит несколько (одно или более) слов, разделенных ровно одним пробелом. Создать XML-документ с корневым элементом `root`, элементами первого уровня `line` и элементами второго уровня `word`. Элементы `line` соответствуют строкам исходного файла и не содержат дочерних текстовых узлов, элементы `word` каждого элемента `line` содержат по одному слову из соответствующей строки (слова располагаются в алфавитном порядке).

LinqXml5. Даны имена существующего текстового файла и создаваемого XML-документа. Каждая строка текстового файла содержит несколько (одно или более) слов, разделенных ровно одним пробелом. Создать XML-документ с корневым элементом `root`, элементами первого уровня `line` и элементами второго уровня `word`. Элементы `line` соответствуют строкам исходного файла и не содержат дочерних текстовых узлов, элементы `word` каждого элемента `line` содержат по одному слову из соответствующей строки (слова располагаются в порядке их следования в исходной строке). Элемент

`line` должен содержать атрибут `num`, равный порядковому номеру строки в исходном файле, элемент `word` должен содержать атрибут `num`, равный порядковому номеру слова в строке.

LinqXml6. Даны имена существующего текстового файла и создаваемого XML-документа. Каждая строка текстового файла содержит несколько (одно или более) целых чисел, разделенных ровно одним пробелом. Создать XML-документ с корневым элементом `root`, элементами первого уровня `line` и элементами второго уровня `number`. Элементы `line` соответствуют строкам исходного файла и не содержат дочерних текстовых узлов, элементы `number` каждого элемента `line` содержат по одному числу из соответствующей строки (числа располагаются в порядке убывания). Элемент `line` должен содержать атрибут `sum`, равный сумме всех чисел из соответствующей строки.

LinqXml7. Даны имена существующего текстового файла и создаваемого XML-документа. Каждая строка текстового файла содержит несколько (одно или более) целых чисел, разделенных ровно одним пробелом. Создать XML-документ с корневым элементом `root`, элементами первого уровня `line` и элементами второго уровня `sum-positive` и `number-negative`. Элементы `line` соответствуют строкам исходного файла и не содержат дочерних текстовых узлов, элемент `sum-positive` является первым дочерним элементом каждого элемента `line` и содержит сумму всех положительных чисел из соответствующей строки, элементы `number-negative` содержат по одному отрицательному числу из соответствующей строки (числа располагаются в порядке, обратном порядку их следования в исходной строке).

LinqXml8. Даны имена существующего текстового файла и создаваемого XML-документа. Каждая строка текстового файла содержит несколько (одно или более) слов, разделенных ровно одним пробелом. Создать XML-документ с корневым элементом `root`, элементами первого уровня `line`, элементами второго уровня `word` и элементами третьего уровня `char`. Элементы `line` и `word` не содержат дочерних текстовых узлов. Элементы `line` соответствуют строкам исходного файла, элементы `word` каждого элемента `line` соответствуют словам из этой строки (слова располагаются в алфавитном порядке), элементы `char` каждого элемента `word` содержат по одному символу из соответствующего слова (символы располагаются в порядке их следования в слове).

LinqXml9. Даны имена существующего текстового файла и создаваемого XML-документа. Создать XML-документ с корневым эле-

ментом `root`, элементами первого уровня `line` и комментариями (комментарии являются дочерними узлами корневого элемента). Если строка текстового файла начинается с текста «comment:», то она (без текста «comment:») добавляется в XML-документ в качестве очередного комментария, в противном случае строка добавляется в качестве дочернего текстового узла в очередной элемент `line`.

LinqXml10. Даны имена существующего текстового файла и создаваемого XML-документа. Создать XML-документ с корневым элементом `root`, элементами первого уровня `line` и инструкциями обработки (инструкции обработки являются дочерними узлами корневого элемента). Если строка текстового файла начинается с текста «data:», то она (без текста «data:») добавляется в XML-документ в качестве данных к очередной инструкции обработки с именем `instr`, в противном случае строка добавляется в качестве дочернего текстового узла в очередной элемент `line`.

Примечание. Решение данной задачи приведено в п. 7.1.

4.2. Анализ содержимого XML-документа

Перед выполнением заданий из данного пункта следует ознакомиться с примером решения задачи LinqXml20, приведенным в п. 7.2.

LinqXml11. Дан XML-документ. Найти все различные имена его элементов и вывести каждое найденное имя вместе с числом его вхождений в документ. Имена элементов выводить в порядке их первого вхождения.

Указание. Использовать метод `GroupBy`.

LinqXml12. Дан XML-документ, содержащий хотя бы один элемент первого уровня. Найти все различные имена элементов первого уровня и вывести каждое найденное имя вместе с числом его вхождений в документ в качестве имени элемента первого уровня. Имена элементов выводить в алфавитном порядке.

LinqXml13. Дан XML-документ, содержащий хотя бы один атрибут. Вывести все различные имена атрибутов, входящих в документ. Порядок имен атрибутов должен соответствовать порядку их первого вхождения в документ.

Указание. Использовать методы `SelectMany` и `Distinct`.

LinqXml14. Дан XML-документ. Найти элементы второго уровня, имеющие дочерний текстовый узел, и вывести количество найденных элементов, а также имя каждого найденного элемента и значение его дочернего текстового узла. Порядок вывода элементов должен соответствовать порядку их следования в документе.

LinqXml15. Дан XML-документ, содержащий хотя бы один элемент первого уровня. Для каждого элемента первого уровня найти количество его потомков, имеющих не менее двух атрибутов, и вывести имя элемента первого уровня и найденное количество его потомков. Элементы выводить в алфавитном порядке их имен, а при совпадении имен – в порядке возрастания найденного количества потомков.

LinqXml16. Дан XML-документ, содержащий хотя бы один элемент первого уровня. Для каждого элемента первого уровня найти суммарное количество атрибутов у его элементов-потомков второго уровня (то есть элементов, являющихся дочерними элементами его дочерних элементов) и вывести найденное количество атрибутов и имя элемента. Элементы выводить в порядке убывания найденного количества атрибутов, а при совпадении количества атрибутов – в алфавитном порядке имен.

LinqXml17. Дан XML-документ, содержащий хотя бы один текстовый узел. Найти все различные имена элементов, имеющих дочерний текстовый узел, и вывести эти имена, а также значения всех связанных с ними дочерних текстовых узлов. Имена выводить в алфавитном порядке; текстовые значения, связанные с каждым именем, выводить в порядке их появления в документе.

LinqXml18. Дан XML-документ, содержащий хотя бы один атрибут. Найти все различные имена атрибутов и вывести эти имена, а также все связанные с ними значения (все значения считаются текстовыми). Порядок имен должен соответствовать порядку их первого вхождения в документ; значения, связанные с каждым именем, выводить в алфавитном порядке.

LinqXml19. Дан XML-документ, содержащий хотя бы один элемент первого уровня. Для каждого элемента первого уровня найти его дочерние элементы, имеющие максимальное количество потомков (при подсчете числа потомков учитывать также потомки-узлы, не являющиеся элементами). Перебирая элементы первого уровня в порядке их появления в XML-документе, вывести имя элемента, число N – максимальное количество потомков, имеющихся у его дочерних элементов (значение N может быть равно 0), и имена всех

дочерних элементов, имеющих N потомков (имена дочерних элементов выводить в алфавитном порядке; среди этих имен могут быть совпадающие). Если элемент первого уровня не содержит дочерних элементов, то в качестве значения N выводить -1 , а в качестве имени дочернего элемента – текст «no child».

LinqXml20. Дан XML-документ, содержащий хотя бы один элемент первого уровня. Для каждого элемента первого уровня найти его элементы-потомки, имеющие максимальное количество атрибутов. Перебирая элементы первого уровня в порядке их появления в XML-документе, вывести имя элемента, число N – максимальное количество атрибутов у его потомков (значение N может быть равно 0), и имена потомков, имеющих N атрибутов (имена потомков выводить в алфавитном порядке; среди этих имен могут быть совпадающие). Если элемент первого уровня не содержит элементов-потомков, то в качестве значения N выводить -1 , а в качестве имени потомка – текст «no child».

Примечание. Решение данной задачи приведено в п. 7.2.

4.3. Преобразование XML-документа

Перед выполнением заданий из данного пункта следует ознакомиться с примерами решения задач LinqXml28, LinqXml32, LinqXml37, приведенными в п. 7.3.

LinqXml21. Даны XML-документ и строка S . В строке записано имя одного из некорневых элементов исходного документа. Удалить из документа все элементы первого уровня с именем S .

Указание. Применить метод `Remove` к последовательности `Elements(S)` корневого элемента.

LinqXml22. Даны XML-документ и строка S . В строке записано имя одного из некорневых элементов исходного документа. Удалить из документа все элементы с именем S .

LinqXml23. Дан XML-документ. Удалить из документа все инструкции обработки.

Указание. Для получения последовательности всех инструкций обработки воспользоваться методом `OfType<XProcessingInstruction>`.

LinqXml24. Дан XML-документ. Удалить из документа все комментарии, являющиеся узлами первого или второго уровня (то есть

имеющие своим родительским элементом корневой элемент или элемент первого уровня).

LinqXml25. Дан XML-документ. Для всех элементов первого и второго уровней, имеющих более одного атрибута, удалить все их атрибуты.

LinqXml26. Дан XML-документ. Для всех элементов документа удалить все их атрибуты, кроме первого.

Указание. В предикате метода `Where`, отбирающем все атрибуты элемента, кроме первого, использовать метод `PreviousAttribute` класса `XAttribute`.

LinqXml27. Дан XML-документ. Для всех элементов второго уровня удалить все их дочерние узлы, кроме последнего.

LinqXml28. Дан XML-документ. Удалить дочерние текстовые узлы для всех элементов третьего уровня. Если текстовый узел является единственным дочерним узлом элемента, то после его удаления элемент должен быть представлен в виде комбинированного тега.

Указание. Использовать метод `OfType<XText>`.

Примечание. Решение данной задачи приведено в п. 7.3.

LinqXml29. Дан XML-документ. Удалить из документа все элементы первого и второго уровней, не содержащие дочерних узлов.

LinqXml30. Дан XML-документ. Удалить из документа все элементы третьего уровня, представленные комбинированным тегом.

Указание. Использовать свойство `IsEmpty` класса `XElement`.

LinqXml31. Даны XML-документ и строки S_1 и S_2 . В строке S_1 записано имя одного из элементов исходного документа, строка S_2 содержит допустимое имя элемента XML. После каждого элемента первого уровня с именем S_1 добавить элемент с именем S_2 . Атрибуты и потомки добавленного элемента должны совпадать с атрибутами и потомками предшествующего элемента.

Указание. Для каждого элемента S_1 вызвать метод `AddAfterSelf` с тремя параметрами: строкой S_2 и последовательностями `Attributes` и `Nodes` элемента S_1 .

LinqXml32. Дан XML-документ и строки S_1 и S_2 . В строке S_1 записано имя одного из элементов исходного документа, строка S_2 содержит допустимое имя элемента XML. Перед каждым элементом второго уровня с именем S_1 добавить элемент с именем S_2 . Добавленный элемент должен содержать последний атрибут и первый дочерний

элемент последующего элемента (если они есть). Если элемент S_1 не имеет дочерних элементов, то добавленный перед ним элемент S_2 должен быть представлен в виде комбинированного тега.

Указание. Использовать метод `FirstOrDefault`.

Примечание. Решение данной задачи приведено в п. 7.3.

LinqXml33. Дан XML-документ. Для каждого элемента первого уровня, имеющего атрибуты, добавить в конец его дочерних узлов элемент с именем `attr` и атрибутами, совпадающими с атрибутами обрабатываемого элемента первого уровня, после чего удалить все атрибуты у обрабатываемого элемента. Добавленный элемент `attr` должен быть представлен в виде комбинированного тега.

Указание. Использовать метод `ReplaceAttributes`, указав в качестве параметра новый дочерний элемент.

LinqXml34. Дан XML-документ. Для каждого элемента первого уровня, имеющего атрибуты, добавить в конец его дочерних узлов элементы с именами, совпадающими с именами его атрибутов, и текстовыми значениями, совпадающими со значениями соответствующих атрибутов, после чего удалить все атрибуты обрабатываемого элемента первого уровня.

Указание. Использовать метод `ReplaceAttributes`, указав в качестве параметра последовательность элементов, полученную методом `Select` из последовательности атрибутов.

LinqXml35. Дан XML-документ. Для каждого элемента второго уровня добавить в конец списка его атрибутов атрибут `child-count` со значением, равным количеству всех дочерних узлов этого элемента. Если элемент не имеет дочерних узлов, то атрибут `child-count` должен иметь значение 0.

LinqXml36. Дан XML-документ. Для каждого элемента второго уровня, имеющего потомков, добавить в конец списка его атрибутов атрибут `node-count` со значением, равным количеству узлов-потомков этого элемента (всех уровней).

LinqXml37. Дан XML-документ. Для каждого элемента второго уровня, имеющего потомков, добавить к его текстовому содержимому текстовое содержимое всех элементов-потомков, после чего удалить все его узлы-потомки, кроме дочернего текстового узла.

Указание. Использовать свойство `Value` класса `XElement`.

Примечание. Решение данной задачи приведено в п. 7.3.

LinqXml38. Дан XML-документ. Для каждого элемента, кроме корня, изменить его имя, добавив к нему слева исходные имена всех его предков, разделенные символом «-» (дефис). Например, если корневой элемент имеет имя `root`, то элемент `bb` второго уровня, родительский элемент которого имеет имя `aa`, должен получить имя `root-aa-bb`.

Указание. Перебирая все элементы последовательности `Descendants` корневого элемента, использовать их свойства `Parent` и `Name`.

LinqXml39. Дан XML-документ. Для каждого элемента, кроме корня, изменить его имя, добавив к нему слева исходное имя его родительского элемента, дополненное символом «-» (дефис). Например, элемент `cc` третьего уровня, родительский элемент которого имеет имя `bb`, должен получить имя `bb-cc`.

Указание. Организовать перебор последовательности `Descendants` корневого элемента в обратном порядке (используя метод `Reverse`).

LinqXml40. Дан XML-документ. Изменить имена атрибутов всех элементов, добавив слева к исходному имени атрибута имя содержащего его элемента, дополненное символом «-» (дефис).

Указание. Так как свойство `Name` класса `XAttribute` доступно только для чтения, следует сформировать новую последовательность атрибутов с требуемыми именами и значениями (применяя метод `Select` к последовательности `Attributes`), после чего указать ее в качестве параметра метода `ReplaceAttributes`.

4.4. Преобразование типов при обработке XML-документа

Перед выполнением заданий из данного пункта следует ознакомиться с примером решения задачи `LinqXml50`, приведенным в п. 7.4.

LinqXml41. Дан XML-документ. Любой его элемент содержит либо набор дочерних элементов, либо текстовое представление вещественного числа. Добавить к каждому элементу, содержащему дочерние элементы, атрибут `sum`, равный сумме чисел, указанных в дочерних элементах. Сумма округляется до двух дробных знаков, незначащие нули не отображаются. Если ни один из дочерних эле-

ментов не содержит текстового представления числа, то атрибут `sum` должен иметь значение 0.

Указание. Для преобразования текстового представления вещественного числа в само число достаточно выполнить приведение к типу `double` элемента XML, содержащего это текстовое представление. Для указания числового значения атрибута `sum` достаточно передать в качестве второго параметра конструктора `XAttribute` вещественное число, округленное требуемым образом (с помощью функции `Math.Round` с двумя параметрами).

LinqXml42. Дан XML-документ, в котором значения всех атрибутов являются текстовыми представлениями вещественных чисел. Добавить к каждому элементу первого уровня, содержащему дочерние элементы, дочерний элемент `sum`, содержащий текстовое представление суммы атрибутов всех дочерних элементов данного элемента. Сумма округляется до двух дробных знаков, незначащие нули не отображаются. Если ни один из дочерних элементов не содержит атрибутов, то элемент `sum` должен иметь значение 0.

LinqXml43. Дан XML-документ, в котором значения всех атрибутов являются текстовыми представлениями вещественных чисел. Добавить к каждому элементу первого уровня, содержащему дочерние элементы, дочерний элемент `max`, содержащий текстовое представление максимального из значений атрибутов всех элементов-потомков данного элемента. Если ни один из элементов-потомков не содержит атрибутов, то элемент `max` не добавлять.

Указание. Для единообразной обработки двух ситуаций (наличие или отсутствие атрибутов у потомков) можно построить по последовательности атрибутов последовательность числовых значений `Nullable`-типа `double?`, применить к ней метод `Max` и добавить новый элемент `max` с помощью метода `SetElementValue`, указав в качестве второго параметра результат, возвращенный методом `Max`. При отсутствии атрибутов у потомков метод `Max` вернет значение `null`; в этом случае метод `SetElementValue` не будет создавать новый элемент.

LinqXml44. Даны XML-документ и строка *S*. В строке записано имя одного из атрибутов исходного документа; известно, что все атрибуты с указанным именем содержат текстовое представление вещественного числа. Для каждого элемента выполнить следующие действия: перебирая всех его потомков, содержащих атрибут *S*, найти минимальное значение данного атрибута и записать это значение

в новый атрибут `min` обрабатываемого элемента. Если ни один из потомков элемента не содержит атрибута `S`, то атрибут `min` к этому элементу не добавлять.

Указание. Использовать приведение атрибута `Attribute(S)` к `Nullable-типу double?`; если атрибут с указанным именем отсутствует, то будет возвращено значение `null`. Для создания нового атрибута с найденным минимальным значением использовать метод `SetAttributeValue`; в случае значения `null` атрибут создан не будет.

LinqXml45. Дан XML-документ. Для каждого элемента, имеющего атрибуты, добавить в начало его набора дочерних узлов элемент с именем `odd-attr-count` и логическим значением, равным `true`, если суммарное количество атрибутов данного элемента и всех его элементов-потомков является нечетным, и `false` в противном случае.

Указание. В качестве параметра конструктора `XElement`, определяющего значение элемента, следует использовать логическое выражение; это позволит отобразить значение логической константы в соответствии со стандартом XML.

LinqXml46. Дан XML-документ. Для каждого элемента, имеющего дочерние элементы, добавить в конец его набора атрибутов атрибут с именем `odd-node-count` и логическим значением, равным `true`, если суммарное количество дочерних узлов у всех его дочерних элементов является нечетным, и `false` в противном случае.

LinqXml47. Дан XML-документ. Для каждого элемента, имеющего хотя бы один дочерний элемент, добавить дочерний элемент с именем `has-comments` и логическим значением, равным `true`, если данный элемент содержит в числе своих узлов-потомков один или более комментариев, и `false` в противном случае. Новый элемент добавить после первого имеющегося дочернего элемента.

LinqXml48. Дан XML-документ. Для каждого элемента, имеющего не менее двух дочерних узлов, добавить дочерний элемент с именем `has-instructions` и логическим значением, равным `true`, если данный элемент содержит в числе своих дочерних узлов одну или более инструкций обработки, и `false` в противном случае. Новый элемент добавить перед последним имеющимся дочерним узлом.

LinqXml49. Даны XML-документ и строка `S`. В строке записано имя одного из атрибутов исходного документа; известно, что все атрибуты с указанным именем содержат представление некоторого промежутка времени (в днях, часах, минутах и секундах) в формате,

принятом в стандарте XML. Добавить в корневой элемент атрибут `total-time`, равный суммарному значению промежутков времени, указанных во всех атрибутах *S* исходного документа.

Указание. Использовать приведение объекта `XAttribute` к типу `TimeSpan`. Для суммирования полученных промежутков времени использовать метод `Aggregate` и операцию «+» для типа `TimeSpan`.

LinqXml50. Дан XML-документ. С каждым элементом документа связывается некоторый промежуток времени (в днях, часах, минутах и секундах). Этот промежуток либо явно указывается в атрибуте `time` данного элемента (в формате, принятом в стандарте XML), либо, если данный атрибут отсутствует, считается равным одному дню. Добавить в начало набора дочерних узлов корневого элемента элемент `total-time` со значением, равным суммарному значению промежутков времени, связанных со всеми элементами первого уровня.

Указание. Использовать приведение объекта `XAttribute` к `Nullable`-типу `TimeSpan?` и операцию `??`.

Примечание. Решение данной задачи приведено в п. 7.4.

LinqXml51. Дан XML-документ. Любой его элемент содержит либо набор дочерних элементов, либо текстовое представление некоторой даты, соответствующее стандарту XML. Изменить все элементы, содержащие дату, следующим образом: добавить атрибут `year`, содержащий значение года из исходной даты, и дочерний элемент `day` с текстовым значением, равным значению дня из исходной даты, после чего удалить из обрабатываемого элемента исходную дату.

Указание. Использовать приведение элемента XML к типу `DateTime`.

LinqXml52. Дан XML-документ. С каждым элементом документа связывается некоторая дата, определяемая номерами года, месяца и дня. Компоненты этой даты указываются в атрибутах `year`, `month`, `day`. Если некоторые из этих атрибутов отсутствуют, то соответствующие компоненты определяются по умолчанию: 2000 для года, 1 для месяца и 10 для дня. Для каждого элемента добавить в начало его набора дочерних узлов элемент `date` с текстовым представлением даты, связанной с обрабатываемым элементом (дата записывается в формате, принятом в стандарте XML), и удалить имеющиеся атрибуты, связанные с компонентами даты.

4.5. Работа с пространствами имен XML-документа

Перед выполнением заданий из данного пункта следует ознакомиться с примером решения задачи LinqXml57, приведенным в п. 7.5.

LinqXml53. Дан XML-документ. В каждом элементе первого уровня определено пространство имен, распространяющееся на все его элементы-потомки. Для каждого элемента первого уровня добавить в конец его набора дочерних узлов элемент с именем `namespace` и значением, равным пространству имен обрабатываемого элемента первого уровня (пространство имен добавленного элемента должно совпадать с пространством имен его родительского элемента).

LinqXml54. Дан XML-документ, у которого корневой элемент и, возможно, какие-либо другие элементы имеют непустое пространство имен. Связать с пространством имен корневого элемента все элементы первого и второго уровней; для элементов более высоких уровней оставить их прежние пространства имен.

LinqXml55. Дан XML-документ. Преобразовать имена всех элементов второго уровня, удалив из них пространства имен (для элементов других уровней пространства имен не изменять).

LinqXml56. Дан XML-документ. В корневом элементе документа определен единственный префикс пространства имен. Снабдить данным префиксом имена всех элементов первого уровня и удалить из этих элементов определения исходных пространств имен (если такие определения имеются).

LinqXml57. Даны XML-документ и строки S_1 и S_2 , содержащие различные пространства имен. Удалить в документе определения исходных пространств имен и определить в корневом элементе два префикса пространств имен: префикс x , связанный с S_1 , и префикс y , связанный с S_2 . Снабдить префиксом x элементы нулевого и первого уровней, а префиксом y – элементы последующих уровней.

Примечание. Решение данной задачи приведено в п. 7.5.

LinqXml58. Даны XML-документ и строка S , содержащая некоторое пространство имен. Определить в корневом элементе префикс `node`, связанный с пространством имен, заданным в строке S , и добавить в каждый элемент первого уровня два атрибута: атрибут `node:count` со значением, равным количеству потомков-узлов для данного элемента, и атрибут `xml:count` со значением, равным коли-

честву потомков-элементов для данного элемента (xml – префикс пространства имен XML).

Указание. Использовать свойство Xml класса XNamespace.

LinqXml59. Дан XML-документ. В каждом из элементов первого уровня определен единственный префикс пространства имен, причем известно, что все атрибуты с этим префиксом имеют целочисленные значения. Для каждого элемента первого уровня и его элементов-потомков удвоить значения всех атрибутов с префиксом пространства имен, определенным в этом элементе.

LinqXml60. Дан XML-документ, корневой элемент которого содержит определения двух префиксов пространств имен с именами x и y. Эти префиксы используются далее в именах некоторых элементов (у атрибутов префиксы отсутствуют). Удалить определение префикса y и для всех элементов, снабженных этим префиксом, заменить его на префикс x, а для всех элементов, снабженных префиксом x, заменить его на префикс xml пространства имен XML.

4.6. Дополнительные задания на обработку XML-документов

Во всех заданиях данной подгруппы предполагается, что в корневом элементе исходного XML-документа определено некоторое пространство имен, распространяющееся на все его элементы-потомки. Это же пространство имен необходимо использовать для имен всех элементов преобразованного документа. Префиксы пространств имен в заданиях данной подгруппы не используются.

Перед выполнением заданий из данного пункта следует ознакомиться с примерами решения задач LinqXml61 и LinqXml82, приведенными в п. 7.6.

LinqXml61. Дан XML-документ с информацией о клиентах фитнес-центра. Образец элемента первого уровня:

```
<record>
  <id>10</id>
  <date>2000-05-01T00:00:00</date>
  <time>PT5H13M</time>
</record>
```

Здесь id – код клиента (целое число), date – дата с информацией о годе и месяце, time – продолжительность занятий (в часах и ми-

нутах) данного клиента в течение указанного месяца. Преобразовать документ, изменив элементы первого уровня следующим образом:

```
<time id="10" year="2000" month="5">PT5H13M</time>
```

Порядок следования элементов первого уровня не изменять.

Примечание. Решение данной задачи приведено в п. 7.6.

LinqXml62. Дан XML-документ с информацией о клиентах фитнес-центра. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml61):

```
<time year="2000" month="5" id="10">PT5H13M</time>
```

Преобразовать документ, изменив элементы первого уровня следующим образом:

```
<id10 date="2000-05-01T00:00:00">313</id10>
```

Имя элемента должно иметь префикс `id`, после которого указывается код клиента; в значении атрибута `date` день должен быть равен 1, а время должно быть нулевым. Значение элемента равно продолжительности занятий клиента в данном месяце, переведенной в минуты. Элементы должны быть отсортированы по возрастанию кода клиента, а для одинаковых значений кода – по возрастанию даты.

LinqXml63. Дан XML-документ с информацией о клиентах фитнес-центра. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml61):

```
<record date="2000-05-01T00:00:00" id="10" time="PT5H13M" />
```

Преобразовать документ, выполнив группировку данных по кодам клиентов и изменив элементы первого уровня следующим образом:

```
<client id="10">
  <time year="2000" month="5">PT5H13M</time>
  ...
</client>
```

Элементы первого уровня должны быть отсортированы по возрастанию кода клиента, их дочерние элементы – по возрастанию номера года, а для одинаковых значений года – по возрастанию номера месяца.

LinqXml64. Дан XML-документ с информацией о клиентах фитнес-центра. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml61):

```
<client id="10">
  <date>2000-05-01T00:00:00</date>
  <time>PT5H13M</time>
</client>
```

Преобразовать документ, сгруппировав данные по годам, а в пределах каждого года – по месяцам. Изменить элементы первого уровня следующим образом:

```
<y2000>
  <m5>
    <client id="10" time="313" />
    ...
  </m5>
  ...
</y2000>
```

Имя элемента первого уровня должно иметь префикс *y*, после которого указывается номер года; имя элемента второго уровня должно иметь префикс *m*, после которого указывается номер месяца. Атрибут *time* должен содержать продолжительность занятий в минутах. Элементы первого уровня должны быть отсортированы по убыванию номера года, их дочерние элементы – по возрастанию номера месяца. Элементы третьего уровня, имеющие общего родителя, должны быть отсортированы по возрастанию кодов клиентов.

LinqXml65. Дан XML-документ с информацией о клиентах фитнес-центра. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml61, данные сгруппированы по кодам клиентов; коды клиентов, снабженные префиксом *id*, указываются в качестве имен элементов первого уровня):

```
<id10>
  <info>
    <date>2000-05-01T00:00:00</date>
    <time>PT5H13M</time>
  </info>
  ...
</id10>
```

Преобразовать документ, сгруппировав данные по годам и изменив элементы первого уровня следующим образом:

```
<year value="2000">
  <total-time id="10">860</total-time>
  ...
</year>
```

Значение элемента второго уровня должно быть равно общей продолжительности занятий (в минутах) клиента с указанным кодом в течение указанного года. Элементы первого уровня должны быть отсортированы по возрастанию номера года, их дочерние элементы – по возрастанию кодов клиентов.

LinqXml66. Дан XML-документ с информацией о клиентах фитнес-центра. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml61, данные сгруппированы по кодам клиентов):

```
<client id="10">
  <info date="2000-05-01T00:00:00" time="PT5H13M" />
  ...
</client>
```

Преобразовать документ, сгруппировав данные по годам и месяцам и оставив сведения только о тех месяцах, в которых посещали занятия не менее трех клиентов. Изменить элементы первого уровня следующим образом:

```
<d2000-5 total-time="956" client-count="3">
  <id10 time="313" />
  ...
</d2000-5>
```

Имя элемента первого уровня должно иметь префикс **d**, после которого указывается номер года и, через дефис, номер месяца (незначащие нули не отображаются). Имя элемента второго уровня должно иметь префикс **id**, после которого указывается код клиента. Атрибут **total-time** должен содержать суммарную продолжительность занятий (в минутах) всех клиентов в данном месяце, атрибут **client-count** – количество клиентов, занимавшихся в этом месяце. Атрибут **time** для элементов второго уровня должен содержать продолжительность занятий (в минутах) клиента с указанным кодом в данном месяце. Элементы первого уровня должны быть отсортированы по возрастанию номера года, а для одинаковых номеров года – по возрастанию номера месяца; их дочерние элементы должны быть отсортированы по возрастанию кодов клиентов.

LinqXml67. Дан XML-документ с информацией о клиентах фитнес-центра. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml61):

```
<client id="10" time="PT5H13M">
  <year>2000</year>
  <month>5</month>
</client>
```

Преобразовать документ, сгруппировав данные по годам и месяцам и изменив элементы первого уровня следующим образом:

```
<y2000>
  <m1 total-time="0" client-count="0" />
  ...
  <m5 total-time="956" client-count="3" />
  ...
</y2000>
```

Имя элемента первого уровня должно иметь префикс *y*, после которого указывается номер года; имя элемента второго уровня должно иметь префикс *m*, после которого указывается номер месяца. Атрибут *total-time* должен содержать суммарную продолжительность занятий (в минутах) всех клиентов в данном месяце, атрибут *client-count* – количество клиентов, занимавшихся в этом месяце. Каждый элемент первого уровня должен содержать элементы второго уровня, соответствующие всем месяцам года; если в каком-либо месяце занятия не проводились, то атрибуты для этого месяца должны иметь нулевые значения. Элементы первого уровня должны быть отсортированы по возрастанию номера года, их дочерние элементы – по возрастанию номера месяца.

Указание. Для эффективного формирования последовательностей, связанных со всеми месяцами, использовать вспомогательную последовательность `Enumerable.Range(1, 12)` и метод `GroupJoin`.

LinqXml68. Дан XML-документ с информацией о ценах автозаправочных станций на бензин. Образец элемента первого уровня:

```
<record>
  <company>Лидер</company>
  <street>ул.Чехова</street>
  <brand>92</brand>
  <price>2200</price>
</record>
```

Здесь **street** – название улицы, **company** – название компании (названия улиц и компаний не содержат пробелов и являются допустимыми именами XML), **brand** – марка бензина (числа 92, 95 или 98), **price** – цена 1 литра бензина в копейках (целое число). Каждая компания имеет не более одной АЗС на каждой улице, цены на разных АЗС одной и той же компании могут различаться. Преобразовать документ, изменив элементы первого уровня следующим образом:

```
<station company="Лидер" street="ул.Чехова">
  <info>
    <brand>92</brand>
    <price>2200</price>
  </info>
</station>
```

Порядок следования элементов первого уровня не изменять.

LinqXml69. Дан XML-документ с информацией о ценах автозаправочных станций на бензин. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml68):

```
<station company="Лидер">
  <info street="ул.Чехова">
    <brand>92</brand>
    <price>2200</price>
  </info>
</station>
```

Преобразовать документ, изменив элементы первого уровня следующим образом:

```
<b92 company="Лидер" street="ул.Чехова" price="2200" />
```

Имя элемента должно иметь префикс **b**, после которого указывается марка бензина. Элементы представляются комбинированными тегам и должны быть отсортированы по возрастанию марок бензина, для одинаковых марок – в алфавитном порядке названий компаний, а для одинаковых компаний – в алфавитном порядке названий улиц.

LinqXml70. Дан XML-документ с информацией о ценах автозаправочных станций на бензин. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml68):

```
<station brand="98" price="2850">
  <company>Лидер</company>
  <street>ул.Авиаторов</street>
</station>
```

Преобразовать документ, выполнив группировку данных по названиям компаний, а в пределах каждой компании – по маркам бензина. Изменить элементы первого уровня следующим образом:

```
<company name="Лидер">
  <brand value="98">
    <price street="ул.Авиаторов">2850</price>
    ...
  </brand>
  ...
</company>
```

Элементы первого уровня должны быть отсортированы в алфавитном порядке названий компаний, а их дочерние элементы – по убыванию марок бензина. Элементы третьего уровня, имеющие общего родителя, должны быть отсортированы в алфавитном порядке названий улиц.

LinqXml71. Дан XML-документ с информацией о ценах автозаправочных станций на бензин. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml68):

```
<station street="ул.Авиаторов" company="Лидер">
  <info brand="98" price="2850" />
</station>
```

Преобразовать документ, выполнив группировку данных по маркам бензина, а в пределах каждой марки – по ценам 1 литра бензина. Изменить элементы первого уровня следующим образом:

```
<b98>
  <p2850>
    <info street="ул.Авиаторов" company="Лидер" />
    ...
  </p2850>
  ...
</b98>
```

Имя элемента первого уровня должно иметь префикс **b**, после которого указывается марка бензина; имя элемента второго уровня должно иметь префикс **p**, после которого указывается цена 1 литра бензина. Элементы первого уровня должны быть отсортированы по убыванию марок бензина, а их дочерние элементы – по убыванию цен. Элементы третьего уровня, имеющие общего родителя, должны быть отсортированы в алфавитном порядке названий улиц, а для одинаковых улиц – в алфавитном порядке названий компаний.

LinqXml72. Дан XML-документ с информацией о ценах автозаправочных станций на бензин. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml68, данные сгруппированы по названиям компаний; названия компаний указываются в качестве имен элементов первого уровня):

```
<Лидер>
  <price street="ул.Чехова" brand="92">2200</price>
  ...
</Лидер>
```

Преобразовать документ, выполнив группировку данных по названиям улиц, а в пределах каждой улицы – по маркам бензина. Изменить элементы первого уровня следующим образом:

```
<ул.Чехова>
  <b92>
    <min-price company="Премьер-нефть">2050</min-price>
    ...
  </b92>
  ...
</ул.Чехова>
```

Имя элемента первого уровня совпадает с названием улицы, имя элемента второго уровня должно иметь префикс **b**, после которого указывается марка бензина. Значение элемента третьего уровня равно минимальной цене бензина данной марки на данной улице, его атрибут **company** содержит название компании, на АЗС которой предлагается минимальная цена. Элементы первого уровня должны быть отсортированы в алфавитном порядке названий улиц, а их дочерние элементы – по возрастанию марок бензина. Если имеется несколько элементов третьего уровня, имеющих общего родителя (это означает, что на одной улице имеется несколько АЗС, на которых бензин данной марки имеет минимальную цену), то эти элементы должны быть отсортированы в алфавитном порядке названий компаний.

LinqXml73. Дан XML-документ с информацией о ценах автозаправочных станций на бензин. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml68, данные сгруппированы по названиям улиц; названия улиц указываются в качестве имен элементов первого уровня):

```
<ул.Чехова>
  <company name="Лидер">
    <brand>92</brand>
```

```
<price>2200</price>
</company>
...
</ул.Чехова>
```

Преобразовать документ, сгруппировав данные по названиям компаний и названиям улиц и оставив сведения только о тех АЗС, на которых предлагаются не менее двух марок бензина. Изменить элементы первого уровня следующим образом:

```
<Лидер_ул.Чехова brand-count="2">
  <b92 price="2200" />
  <b95 price="2450" />
</Лидер_ул.Чехова>
```

Имя элемента первого уровня содержит название компании, после которого следует символ подчеркивания и название улицы; имя элемента второго уровня должно иметь префикс **b**, после которого указывается марка бензина. Атрибут **brand-count** должен содержать количество марок бензина, предлагаемых на данной АЗС. Элементы первого уровня должны быть отсортированы в алфавитном порядке названий компаний, а для одинаковых названий компаний – в алфавитном порядке названий улиц; их дочерние элементы должны быть отсортированы по возрастанию марок бензина.

LinqXml74. Дан XML-документ с информацией о ценах автозаправочных станций на бензин. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml68, марки бензина, снабженные префиксом **brand**, указываются в качестве имен элементов первого уровня; атрибут **station** содержит названия улицы и компании, разделенные символом подчеркивания):

```
<brand92 station="ул.Чехова_Лидер" price="2200" />
```

Преобразовать документ, сгруппировав данные по названиям компаний и изменив элементы первого уровня следующим образом:

```
<Лидер>
  <ул.Садовая brand92="0" brand95="0" brand98="0" />
  <ул.Чехова brand92="2200" brand95="2450" brand98="0" />
  ...
</Лидер>
```

Имя элемента первого уровня совпадает с названием компании, имя элемента второго уровня совпадает с названием улицы. Атрибуты элементов второго уровня имеют префикс **brand**, после которо-

го указывается марка бензина; их значением является цена 1 литра бензина указанной марки или число 0, если на данной АЗС бензин указанной марки не предлагается. Для каждой компании должна выводиться информация по каждой улице, имеющейся в исходном документе, даже если на этой улице отсутствует АЗС данной компании (в этом случае значения всех атрибутов **brand** должны быть равны 0). Элементы первого уровня должны быть отсортированы в алфавитном порядке названий компаний, а их дочерние элементы – в алфавитном порядке названий улиц.

LinqXml75. Дан XML-документ с информацией о ценах автозаправочных станций на бензин. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml68, названия компании и улицы, разделенные символом подчеркивания, указываются в качестве имен элементов первого уровня):

```
<Лидер_ул.Чехова>
  <brand>92</brand>
  <price>2200</price>
</Лидер_ул.Чехова>
```

Преобразовать документ, сгруппировав данные по названиям улиц и изменив элементы первого уровня следующим образом:

```
<ул.Чехова>
  <brand98 station-count="0">0</brand98>
  <brand95 station-count="0">0</brand95>
  <brand92 station-count="3">2255</brand92>
</ул.Чехова>
```

Имя элемента первого уровня совпадает с названием улицы, имя элемента второго уровня имеет префикс **brand**, после которого указывается марка бензина. Атрибут **station-count** равен количеству АЗС, расположенных на данной улице и предлагающих бензин данной марки; значением элемента второго уровня является средняя цена 1 литра бензина данной марки по всем АЗС, расположенным на данной улице. Средняя цена находится по следующей формуле: *«суммарная цена по всем станциям»/«число станций»*, где операция *«/»* обозначает целочисленное деление. Если на данной улице отсутствуют АЗС, предлагающие бензин данной марки, то значение соответствующего элемента второго уровня и значение его атрибута **station-count** должны быть равны 0. Элементы первого уровня должны быть отсортированы в алфавитном порядке названий улиц, а их дочерние элементы – по убыванию марок бензина.

LinqXml76. Дан XML-документ с информацией о задолженности по оплате коммунальных услуг. Образец элемента первого уровня:

```
<record>
  <house>12</house>
  <flat>129</flat>
  <name>Сепреев Т.М.</name>
  <debt>1833.32</debt>
</record>
```

Здесь `house` – номер дома (целое число), `flat` – номер квартиры (целое число), `name` – фамилия и инициалы жильца (инициалы не содержат пробелов и отделяются от фамилии одним пробелом), `debt` – размер задолженности в виде дробного числа: целая часть – рубли, дробная часть – копейки (незначащие нули не указываются). Все дома являются 144-квартирными, имеют 9 этажей и 4 подъезда; на каждом этаже в каждом подъезде располагаются по 4 квартиры. Преобразовать документ, изменив элементы первого уровня следующим образом:

```
<debt house="12" flat="129">
  <name>Сепреев Т.М.</name>
  <value>1833.32</value>
</debt>
```

Порядок следования элементов первого уровня не изменять.

LinqXml77. Дан XML-документ с информацией о задолженности по оплате коммунальных услуг. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml76):

```
<debt house="12" flat="129" name="Сепреев Т.М.">1833.32</debt>
```

Преобразовать документ, изменив элементы первого уровня следующим образом:

```
<address12-129 name="Сепреев Т.М." debt="1833.32" />
```

Имя элемента должно иметь префикс `address`, после которого указывается номер дома и, через дефис, номер квартиры. Элементы представляются комбинированными тегами и должны быть отсортированы по возрастанию номеров домов, а для одинаковых номеров домов – по возрастанию номеров квартир.

LinqXml78. Дан XML-документ с информацией о задолженности по оплате коммунальных услуг. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml76):

```
<debt house="12" flat="23">
  <name>Иванов А.В.</name>
  <value>1245.64</value>
</debt>
```

Преобразовать документ, выполнив группировку данных по номеру дома, а в пределах каждого дома – по номеру подъезда. Изменить элементы первого уровня следующим образом:

```
<house number="12">
  <entrance number="1">
    <debt name="Иванов А.В." flat="23">1245.64</debt>
    ...
  </entrance>
  ...
</house>
```

Элементы первого уровня должны быть отсортированы по возрастанию номеров домов, а их дочерние элементы – по возрастанию номеров подъездов. Элементы третьего уровня, имеющие общего родителя, должны быть отсортированы по убыванию размера задолженности (предполагается, что размеры всех задолженностей являются различными). Подъезды, в которых отсутствуют задолжники, не отображаются.

LinqXml79. Дан XML-документ с информацией о задолженности по оплате коммунальных услуг. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml76):

```
<house value="12">
  <flat value="129" />
  <name value="Сергеев Т.М." />
  <debt value="1833.32" />
</house>
```

Преобразовать документ, выполнив группировку данных по номеру дома, а в пределах каждого дома – по номеру этажа. Изменить элементы первого уровня следующим образом:

```
<house12>
  <floor6>
    <Сергеев_Т.М. flat="129" debt="1833.32" />
    ...
  </floor6>
  ...
</house12>
```

Имя элемента первого уровня должно иметь префикс `house`, после которого указывается номер дома; имя элемента второго уровня должно иметь префикс `floor`, после которого указывается номер этажа. Имя элемента третьего уровня совпадает с фамилией и инициалами жильца; фамилия отделяется от инициалов символом подчеркивания. Элементы первого уровня должны быть отсортированы по возрастанию номеров домов, а их дочерние элементы – по убыванию номеров этажей. Элементы третьего уровня, имеющие общего родителя, должны быть отсортированы в алфавитном порядке фамилий и инициалов жильцов. Этажи, в которых отсутствуют задолжники, не отображаются.

LinqXml80. Дан XML-документ с информацией о задолженности по оплате коммунальных услуг. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml76, данные сгруппированы по номерам домов; в качестве элементов второго уровня указываются фамилии и инициалы жильцов, фамилия отделяется от инициалов символом подчеркивания):

```
<house number="12">
  <Иванов_А.В.>
    <flat value="23" />
    <debt value="1245.64" />
  </Иванов_А.В.>
  ...
</house>
```

Преобразовать документ, сохранив группировку данных по номеру дома, выполнив в пределах каждого дома группировку по номеру подъезда и изменив элементы первого уровня следующим образом:

```
<house12>
  <entrance1 total-debt="2493.38" count="3">
    <flat23 name="Иванов А.В." />
    ...
  </entrance1>
  ...
</house12>
```

Имя элемента первого уровня должно иметь префикс `house`, после которого указывается номер дома, имя элемента второго уровня должно иметь префикс `entrance`, после которого указывается номер подъезда, имя элемента третьего уровня должно иметь префикс `flat`, после которого указывается номер квартиры. Атрибут `total-debt` равен суммарной задолженности жильцов данного подъезда (зна-

чение задолженности должно округляться до двух дробных знаков, незначащие нули не отображаются), атрибут `count` равен количеству задолжников в данном подъезде. Элементы первого уровня должны быть отсортированы по возрастанию номеров домов, а их дочерние элементы – по возрастанию номеров подъездов. Элементы третьего уровня, имеющие общего родителя, должны быть отсортированы по возрастанию номеров квартир. Подъезды, в которых отсутствуют задолжники, не отображаются.

LinqXml81. Дан XML-документ с информацией о задолженности по оплате коммунальных услуг. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml76, данные сгруппированы по номерам домов; в качестве имен элементов первого уровня указываются номера домов, снабженные префиксом `house`, а в качестве имен элементов второго уровня – номера квартир, снабженные префиксом `flat`):

```
<house12>
  <flat23 name="Иванов А.В." debt="1245.64" />
  ...
</house12>
```

Преобразовать документ, сохранив группировку данных по номеру дома, выполнив в пределах каждого дома группировку по номеру подъезда и оставив сведения только о тех жильцах, размер задолженности которых не меньше среднего размера задолженности по данному подъезду. Изменить элементы первого уровня следующим образом:

```
<house number="12">
  <entrance number="1" count="4" avr-debt="1136">
    <debt flat="23" name="Иванов А.В.">1245.64</debt>
    <debt flat="28" name="Сидоров П.К.">1383.27</debt>
  </entrance>
  ...
</house>
```

Атрибут `count` равен количеству задолжников в данном подъезде, атрибут `avr-debt` определяет среднюю задолженность по данному подъезду в рублях (целое число), вычисленную по следующей формуле: $\langle \text{суммарная задолженность в копейках} \rangle / (\langle \text{количество задолжников} \rangle * 100)$ (символ \langle / \rangle обозначает операцию целочисленного деления). Элементы третьего уровня содержат сведения о тех жильцах, размер задолженности которых не меньше величины

avr-debt для данного подъезда. Элементы первого уровня должны быть отсортированы по возрастанию номеров домов, а их дочерние элементы – по возрастанию номеров подъездов. Элементы третьего уровня, имеющие общего родителя, должны быть отсортированы по возрастанию номеров квартир. Подъезды, в которых отсутствуют задолжники, не отображаются.

LinqXml82. Дан XML-документ с информацией о задолженности по оплате коммунальных услуг. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml76, в качестве имени элемента первого уровня указываются номера дома и квартиры, разделенные символом «-» (дефис) и снабженные префиксом `addr`, а в качестве значения этого элемента указывается размер задолженности для данной квартиры):

```
<addr12-23>1245.64</addr12-23>
```

Преобразовать документ, выполнив группировку данных по номеру дома, а в пределах каждого дома – по номеру этажа. Изменить элементы первого уровня следующим образом:

```
<house12>
  <floor1 count="0" total-debt="0" />
  ...
  <floor6 count="1" total-debt="1245.64" />
  ...
  <floor9 count="3" total-debt="3142.7" />
</house12>
```

Имя элемента первого уровня должно иметь префикс `house`, после которого указывается номер дома, имя элемента второго уровня должно иметь префикс `floor`, после которого указывается номер этажа. Атрибут `count` равен числу задолжников на данном этаже, атрибут `total-debt` определяет суммарную задолженность по данному этажу, округленную до двух дробных знаков (незначащие нули не отображаются). Если на данном этаже отсутствуют задолжники, то для соответствующего элемента второго уровня значения атрибутов `count` и `total-debt` должны быть равны 0. Элементы первого уровня должны быть отсортированы по возрастанию номеров домов, а их дочерние элементы – по возрастанию номеров этажей.

Примечание. Решение данной задачи приведено в п. 7.6.

LinqXml83. Дан XML-документ с информацией об оценках учащихся по различным предметам. Образец элемента первого уровня:

```
<record>
  <class>9</class>
  <name>Степанова Д.Б.</name>
  <subject>Физика</subject>
  <mark>4</mark>
</record>
```

Здесь `class` – номер класса (целое число от 7 до 11), `name` – фамилия и инициалы учащегося (инициалы не содержат пробелов и отделяются от фамилии одним пробелом), `subject` – название предмета, не содержащее пробелов, `mark` – оценка (целое число в диапазоне от 2 до 5). Полных однофамильцев (с совпадающей фамилией и инициалами) среди учащихся нет. Преобразовать документ, изменив элементы первого уровня следующим образом:

```
<mark subject="Физика">
  <name class="9">Степанова Д.Б.</name>
  <value>4</value>
</mark>
```

Порядок следования элементов первого уровня не изменять.

LinqXml84. Дан XML-документ с информацией об оценках учащихся по различным предметам. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml83):

```
<pupil class="9" name="Степанова Д.Б.">
  <subject>Физика</subject>
  <mark>4</mark>
</pupil>
```

Преобразовать документ, изменив элементы первого уровня следующим образом:

```
<class9 name="Степанова Д.Б." subject="Физика">4</class9>
```

Имя элемента должно иметь префикс `class`, после которого указывается номер класса. Элементы должны быть отсортированы по возрастанию номеров классов, для одинаковых номеров классов – в алфавитном порядке фамилий и инициалов учащихся, для каждого учащегося – в алфавитном порядке названий предметов, а для одинаковых предметов – по возрастанию оценок.

LinqXml85. Дан XML-документ с информацией об оценках учащихся по различным предметам. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml83):

```
<info class="9" name="Степанова Д.Б." subject="Физика" mark="4" />
```

Преобразовать документ, выполнив группировку данных по номеру класса, в пределах каждого класса – по учащимся, а для каждого учащегося – по предметам. Изменить элементы первого уровня следующим образом:

```
<class number="9">
  <pupil name="Степанова Д.Б.">
    <subject name="Физика">
      <mark>4</mark>
      ...
    </subject>
    ...
  </pupil>
  ...
</class>
```

Элементы первого уровня должны быть отсортированы по возрастанию номеров классов, а их дочерние элементы – в алфавитном порядке фамилий и инициалов учащихся. Элементы третьего уровня, имеющие общего родителя, должны быть отсортированы в алфавитном порядке названий предметов, а элементы четвертого уровня, имеющие общего родителя, должны быть отсортированы по убыванию оценок.

LinqXml86. Дан XML-документ с информацией об оценках учащихся по различным предметам. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml83):

```
<pupil name="Степанова Д.Б." class="9">
  <info mark="4" subject="Физика" />
</pupil>
```

Преобразовать документ, выполнив группировку данных по учащимся и изменив элементы первого уровня следующим образом:

```
<Степанова_Д.Б. class="9">
  <mark4 subject="Физика" />
  ...
</Степанова_Д.Б.>
```

Имя элемента первого уровня совпадает с фамилией и инициалами учащегося (пробел между фамилией и инициалами заменяется символом подчеркивания), имя элемента второго уровня должно иметь префикс `mark`, после которого указывается оценка. Элементы

первого уровня должны быть отсортированы в алфавитном порядке фамилий и инициалов учащихся, их дочерние элементы – по убыванию оценок, а для одинаковых оценок – в алфавитном порядке названий предметов.

LinqXml87. Дан XML-документ с информацией об оценках учащихся по различным предметам. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml83, данные сгруппированы по учащимся):

```
<pupil name="Степанова Д.Б." class="9">
  <mark subject="Физика">4</mark>
  ...
</pupil>
```

Преобразовать документ, выполнив группировку данных по названиям предметов и изменив элементы первого уровня следующим образом:

```
<Физика>
  <class9>
    <mark-count>4</mark-count>
    <avr-mark>4.1</avr-mark>
  </class9>
  ...
</Физика>
```

Имя элемента первого уровня совпадает с названием предмета, имя элемента второго уровня должно иметь префикс `class`, после которого указывается номер класса. Значение элемента `mark-count` равно количеству оценок по данному предмету, выставленных в данном классе; значение элемента `avr-mark` равно среднему значению этих оценок, найденному по следующей формуле: $10 \cdot \frac{\text{«сумма оценок»}}{\text{«количество оценок»}} \cdot 0.1$ (символ `«/»` обозначает операцию целочисленного деления, полученное значение должно содержать не более одного дробного знака, незначащие нули не отображаются). Элементы первого уровня должны быть отсортированы в алфавитном порядке названий предметов, а их дочерние элементы – по возрастанию номеров классов. Для каждого предмета отображать только те классы, в которых выставлена хотя бы одна оценка по этому предмету.

LinqXml88. Дан XML-документ с информацией об оценках учащихся по различным предметам. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml83, данные сгруппированы по классам):

```
<class number="9">
  <pupil name="Степанова Д.Б." subject="Физика" mark="4" />
  ...
</class>
```

Преобразовать документ, выполнив группировку данных по предметам и оставив сведения только о тех учащихся, которые получили по данному предмету более двух оценок. Изменить элементы первого уровня следующим образом:

```
<subject name="Физика">
  <pupil class="9" name="Степанова Д.Б." m1="4" m2="3" m3="3" />
  ...
</subject>
```

Оценки каждого учащегося по данному предмету указываются в атрибутах, имеющих префикс m, после которого следует порядковый номер оценки. Элементы первого уровня должны быть отсортированы в алфавитном порядке названий предметов, их дочерние элементы – по возрастанию номеров классов, а для одинаковых классов – в алфавитном порядке фамилий и инициалов учащихся. Оценки для каждого учащегося должны располагаться в порядке убывания. Если для некоторого предмета не найдены учащиеся, имеющие по нему более двух оценок, то соответствующий элемент первого уровня должен быть представлен комбинированным тегом, например:

```
<subject name="Химия" />
```

LinqXml89. Дан XML-документ с информацией об оценках учащихся по различным предметам. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml83, в качестве имен элементов первого уровня указываются фамилии и инициалы учащихся; при этом пробел между фамилией и инициалами заменяется символом подчеркивания):

```
<Петров_С.Н. class="11" subject="Физика">4</Петров_С.Н.>
```

Преобразовать документ, выполнив группировку данных по предметам, а для каждого предмета – по классам. Изменить элементы первого уровня следующим образом:

```
<Физика>
  <class7 pupil-count="0" mark-count="0" />
```

```

...
<class11 pupil-count="3" mark-count="5" />
</Физика>

```

Имя элемента первого уровня совпадает с названием предмета, имя элемента второго уровня должно иметь префикс `class`, после которого указывается номер класса. Значение атрибута `pupil-count` равно количеству учащихся данного класса, имеющих хотя бы одну оценку по данному предмету, значение атрибута `mark-count` равно количеству оценок по данному предмету в данном классе. Для каждого предмета должна быть выведена информация по каждому классу (от 7 до 11); если в некотором классе по данному предмету не было опрошено ни одного учащегося, то атрибуты `pupil-count` и `mark-count` должны быть равны 0. Элементы первого уровня должны быть отсортированы в алфавитном порядке названий предметов, а их дочерние элементы – по возрастанию номеров классов.

LinqXml90. Дан XML-документ с информацией об оценках учащихся по различным предметам. Образец элемента первого уровня (смысл данных тот же, что и в `LinqXml83`; в качестве имен элементов первого уровня указываются фамилии с инициалами учащихся и номера классов; между фамилией и инициалами указывается символ подчеркивания, а между инициалами и номером класса – дефис):

```

<Степанова_Д.Б.-9 subject="Физика" mark="4" />

```

Преобразовать документ, сгруппировав данные по номерам классов, а для каждого класса – по учащимся. Изменить элементы первого уровня следующим образом:

```

<class9>
  <Степанова_Д.Б.>
    <История count="0">0</История>
    ...
    <Физика count="3">3.3</Физика>
  </Степанова_Д.Б.>
  ...
</class9>

```

Имя элемента первого уровня должно иметь префикс `class`, после которого указывается номер класса, имя элемента второго уровня совпадает с фамилией и инициалами учащегося, между которыми указывается символ подчеркивания. Имя элемента третьего уровня совпадает с названием предмета. Значение атрибута `count` равно количеству оценок по данному предмету, полученных данным уча-



щимся. Значение элемента третьего уровня равно средней оценке по данному предмету для данного учащегося; средняя оценка вычисляется по следующей формуле: $10 * \text{«сумма оценок»} / \text{«количество оценок»} * 0.1$ (символ «/» обозначает операцию целочисленного деления, полученное значение должно содержать не более одного дробного знака, незначащие нули не отображаются). Для каждого учащегося должна быть выведена информация по каждому предмету, входящему в исходный XML-документ; если по некоторому предмету учащийся не имеет оценок, то значение соответствующего элемента третьего уровня и значение его атрибута `count` должны быть равны 0. Элементы первого уровня должны быть отсортированы по возрастанию номеров классов, а их дочерние элементы – в алфавитном порядке фамилий учащихся. Элементы третьего уровня, имеющие общего родителя, должны быть отсортированы в алфавитном порядке названий предметов.



Глава 5. Примеры решения задач из группы LinqBegin

Задания группы LinqBegin предназначены для освоения различных методов преобразования последовательностей. Все эти методы являются *методами расширения* класса System.Linq.Enumerable и образуют интерфейс LINQ To Objects (методы расширения появились в версии 3.0 языка C#; они описываются в п. 8.3).

Методы, входящие в интерфейс LINQ to Object, называют *операторами запроса*, *лямбда-операторами* (чтобы подчеркнуть, что в качестве их параметров обычно указываются лямбда-выражения) или *q-операторами* (от английского слова query – «запрос»). Мы будем их называть *методами LINQ*.

Задания охватывают большинство методов LINQ и разбиты на 4 подгруппы, каждая из которых посвящена определенным видам преобразований: от простейших (поэлементные операции) до наиболее сложных (объединение и группировка).

5.1. Поэлементные операции: LinqBegin4

5.1.1. Создание проекта-заготовки и знакомство с заданием

Начнем с рассмотрения одной из задач, связанных с поэлементными операциями.

LinqBegin4. Даны символ *C* и строковая последовательность *A*. Если *A* содержит единственный элемент, оканчивающийся символом *C*, то вывести этот элемент; если требуемых строк в *A* нет, то вывести пустую строку; если требуемых строк больше одной, то вывести строку «Error».

Указание. Использовать try-блок для перехвата возможного исключения.

Выполнение задания с применением задачника Programming Taskbook начинается с создания проекта-заготовки. Для создания заготовки предназначен программный модуль PT4Load, входящий в состав задачника. Вызвать этот модуль можно с помощью ярлыка Load.lnk, который автоматически создается в рабочем каталоге задачника (по умолчанию рабочий каталог размещается на диске С и имеет имя PT4Work; с помощью программы PT4Setup, также входящей в состав задачника и доступной из его меню **Пуск** \Rightarrow **Программы** \Rightarrow **Programming Taskbook 4**, можно определять другие рабочие каталоги).

Заметим, что группы заданий, связанные с технологией LINQ, не входят в базовый набор электронного задачника Programming Taskbook. Для того чтобы эти группы были доступны, необходимо после установки задачника Programming Taskbook установить его дополнение Programming Taskbook for LINQ – задачник по технологии LINQ.

После запуска модуля PT4Load на экране появится его окно, в котором будут перечислены все доступные группы заданий (рис. 1).

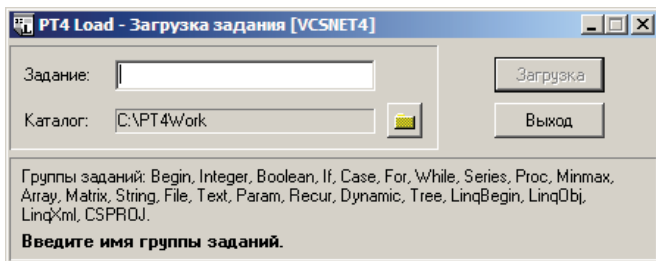


Рис. 1. Окно программного модуля PT4Load

В заголовке окна указываются имя текущего языка программирования и номер его версии или версии среды программирования. Приведенный рисунок соответствует настройке, при которой текущим языком является язык C# 4.0, а средой программирования – Microsoft Visual Studio 2010.

При выполнении заданий, посвященных технологии LINQ, необходимо использовать среду Microsoft Visual Studio, начиная с версии 2008, поскольку эта технология появилась в версии .NET 3.5,

вышедшей одновременно с Visual Studio 2008. В качестве языка программирования следует выбрать либо C#, либо Visual Basic .NET. Если указан язык или среда, отличные от требуемых, следует вызвать контекстное меню модуля PT4Load (щелкнув в его окне правой кнопкой мыши) и выбрать допустимый вариант из появившегося списка. Во всех описаниях примеров выполнения заданий предполагается, что выбраны язык C# и среда Visual Studio версии 2008, 2010 или 2012.

В списке групп должна содержаться группа LinqBegin. Ее отсутствие может объясняться двумя причинами: либо в качестве текущего языка выбран язык, отличный от C# и Visual Basic .NET, либо на компьютере не установлено дополнение Programming Taskbook for LINQ.

После ввода имени задания (в нашем случае LinqBegin4) кнопка **Загрузка** в окне **PT4Load** станет доступной и, нажав ее (или клавишу **Enter**), мы создадим проект-заготовку для указанного задания, которая будет немедленно загружена в среду Visual Studio. Созданный проект будет состоять из нескольких файлов, однако для решения задачи нам потребуется только файл с именем LinqBegin4.cs. Именно этот файл будет загружен в редактор среды Visual Studio. Приведем его начальную часть:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using PT4;

namespace PT4Tasks
{
    public class MyTask : PT
    {
        // При решении задач группы LinqBegin доступны следующие
        // дополнительные методы, определенные в задачнике:
        //
        //     GetEnumeratorInt() - ввод числовой последовательности;
        //
        //     GetEnumeratorString() - ввод строковой последовательности;
        //
        //     Put() (метод расширения) - вывод последовательности;
        //
        //     Show() и Show(cmt) (методы расширения) - отладочная печать
        //         последовательности, cmt - строковый комментарий;
```

```
//
// Show(e => r) и Show(cmt, e => r) (методы расширения) -
// отладочная печать значений г, полученных из элементов е
// последовательности, cmt - строковый комментарий.

public static void Solve()
{
    Task("LinqBegin4");
}

. . .
```

Файл LinqBegin4.cs начинается с директив using, подключающих основные пространства имен, в том числе System.Collections.Generic – для обобщенных коллекций, System.Linq – для класса Enumerable, содержащего методы расширения LINQ to Objects, System.Text – для классов, связанных с обработкой текста, и PT4 – для класса PT, обеспечивающего доступ к функциям ядра задачника Programming Taskbook. Затем следует описание класса MyTask – потомка класса PT. Метод Solve, в котором требуется запрограммировать решение задачи, содержит вызов метода Task, инициализирующего задачу с указанным именем.

Завершающая часть файла, не приведенная выше, содержит реализацию вспомогательных методов, которые могут оказаться полезными при выполнении заданий, связанных с технологией LINQ. Краткое описание этих методов приводится в комментариях, расположенных перед методом Solve.

Примечание. Если созданная заготовка содержит только метод Solve и в ней отсутствуют вспомогательные методы и связанные с ними комментарии, то это означает, что была выбрана одна из ранних сред Visual Studio для платформы .NET (2003 или 2005), в которой технология LINQ не поддерживается.

Для запуска созданной программы достаточно нажать клавишу **F5**. При этом на экране появится окно задачника (рис. 2). Приведенный на рисунке вид окна соответствует *режиму с динамической компоновкой разделов*, который появился в версии 4.11 задачника Programming Taskbook.

В дальнейшем мы всегда будем использовать режим с динамической компоновкой. Если окно отображается в «старом» режиме с фиксированной компоновкой (рис. 3), то для переключения на новый режим достаточно нажать клавишу **F4**.

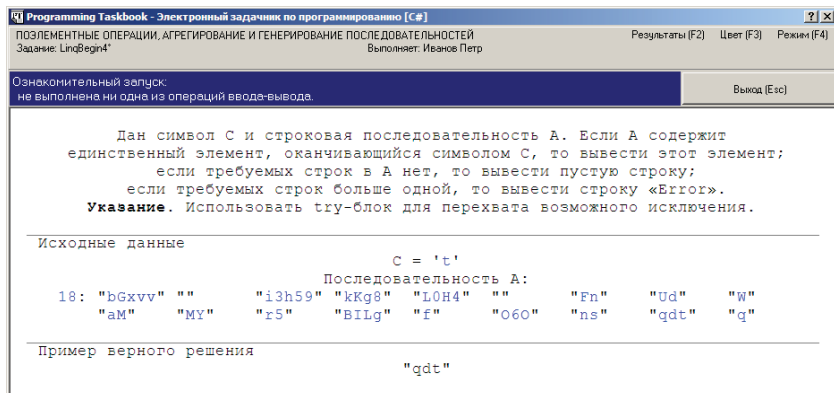


Рис. 2. Ознакомительный запуск задания LinqBegin4 (режим окна с динамической компоновкой)

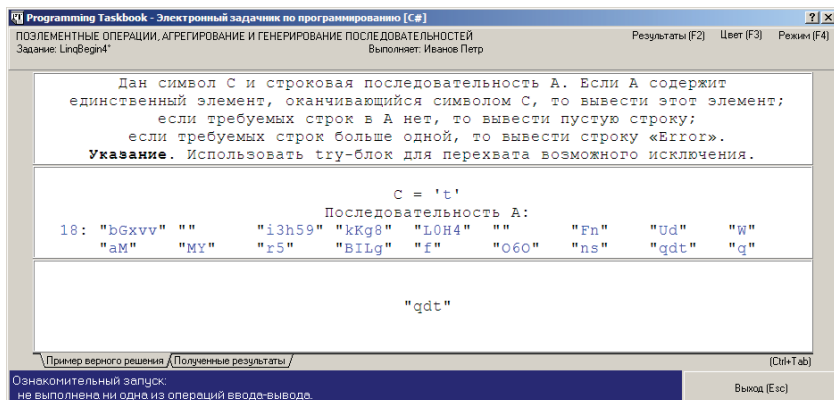


Рис. 3. Ознакомительный запуск задания LinqBegin4 (режим окна с фиксированной компоновкой)

Поскольку в программе не выполняются действия по вводу и выводу данных, запуск программы считается *ознакомительным*. При ознакомительном запуске в окне задачника отображаются три раздела: с формулировкой задания, исходными данными и примером верного решения.

Все исходные данные выделяются особым *цветом*, чтобы отличить их от комментариев: если для окна установлен цветовой режим

«на черном фоне», то данные выделяются желтым цветом; в режиме «на белом фоне» (как на рис. 2 и 3) данные выделяются синим цветом. Для переключения цветового режима достаточно нажать клавишу **F3**.

Запуск программы позволяет ознакомиться с образцом исходных данных и соответствующим этим исходным данным примером правильного решения. В нашем случае в набор исходных данных входят символ *C* и строковая последовательность *A*. Во всех заданиях группы LinqBegin при определении последовательности вначале указывается *количество* ее элементов, а затем – *значения* самих элементов (на экране количество элементов отделяется от списка значений двоеточием), причем список значений может занимать несколько экранных строк. Символьные данные заключаются в одинарные кавычки, а строковые – в двойные, что соответствует представлению символьных и строковых констант, принятому в языке C#. Использование кавычек позволяет, в частности, «увидеть» пустые строки, входящие в наборы исходных или результирующих данных (как в примере исходных данных, приведенном на рис. 2 и 3).

Для выхода из программы достаточно нажать кнопку **Выход**, клавишу **Esc** или клавишу **F5** (то есть ту же клавишу, которая обеспечивает запуск программы из среды Visual Studio).

Чтобы более подробно ознакомиться с заданием (и группой заданий в целом), можно использовать два специальных режима задачника: демонстрационный режим и режим отображения заданий в формате html.

Для запуска программы в *демонстрационном режиме* достаточно дополнить параметр метода Task символом «?» (в нашем случае вызов метода примет вид Task("LinqBegin4?");). Окно задачника в демонстрационном режиме имеет дополнительные кнопки, позволяющие переходить к предыдущему или последующему заданию, а также отображать на экране различные наборы исходных данных и связанные с ними образцы правильного решения (см. рис. 4). На рисунке приведена ситуация, в которой исходная последовательность содержит *несколько* элементов, оканчивающихся символом *C*, равным «g» (это элементы «2uo7g» и «pcg»), поэтому для правильного решения требуется вывести строку «Error».

Для отображения задания в *формате html* достаточно дополнить параметр метода Task символом «#» (в нашем случае вызов метода примет вид Task("LinqBegin4#");). При запуске такого варианта программы будет запущен html-браузер, используемый в системе

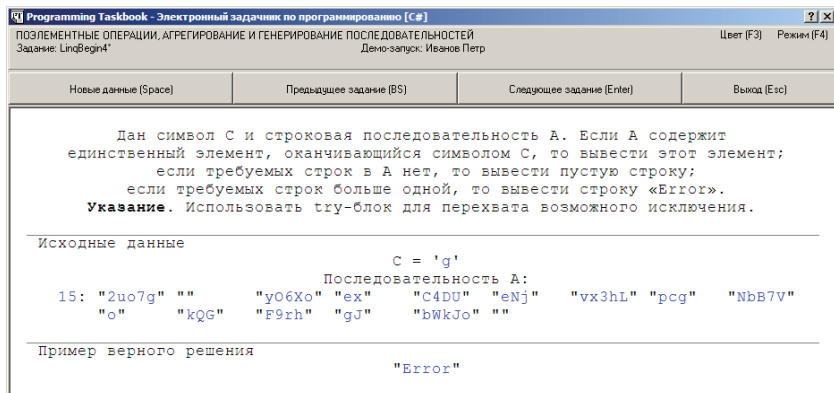


Рис. 4. Запуск задания LinqBegin4 в демонстрационном режиме

по умолчанию, и в него загрузится html-страница, содержащая формулировку задания и текст *преамбулы* к соответствующей группе заданий (см. рис. 5).

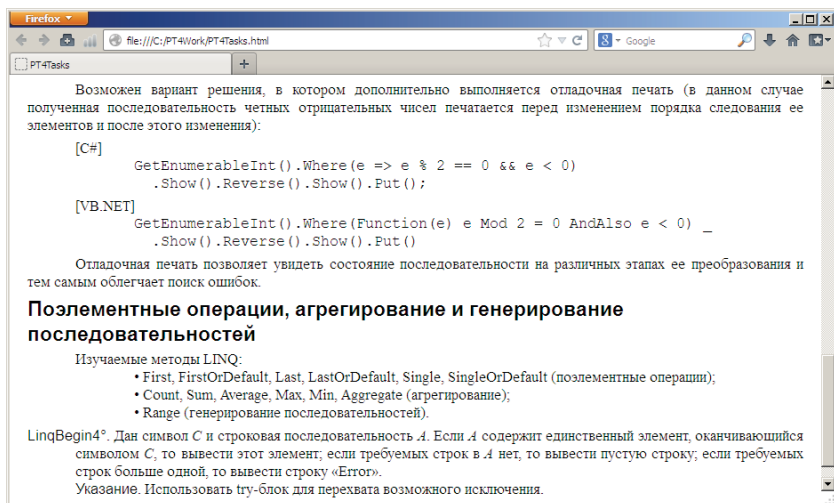


Рис. 5. Отображение задания LinqBegin4 в формате html

В данном случае на экране, кроме формулировки задания LinqBegin4, отображаются преамбула ко всей группе LinqBegin (текст этой преамбулы приведен в гл. 2) и преамбула к подгруппе «Поэле-

ментные операции, агрегирование и генерирование последовательностей», в которую входит задание LinqBegin4.

Режим html-страницы удобен в нескольких отношениях. Во-первых, только в этом режиме можно ознакомиться с преамбулами к группе заданий и ее подгруппам и тем самым получить дополнительные сведения, которые могут оказаться полезными при выполнении заданий. Во-вторых, окно браузера с html-страницей не требуется закрывать, для того чтобы продолжить работу над заданием; таким образом, с помощью этого окна можно в любой момент ознакомиться с формулировкой задания, даже если текущее состояние программы не позволяет откомпилировать ее и запустить на выполнение.

Имеется возможность отобразить в html-режиме *все* задания, входящие в некоторую группу. Для этого в параметре метода Task следует удалить номер задания, оставив только имя группы и символ «#» (например, `Task("LinqBegin#");`).

Закончив на этом обзор возможностей задачника, предназначенных для формирования проекта-заготовки и знакомства с выбранным заданием, перейдем к описанию самого процесса решения.

5.1.2. Выполнение задания

Первым этапом при выполнении любого задания является ввод исходных данных. Если задание выполняется с применением электронного задачника, то именно задачник формирует набор исходных данных и предоставляет его программе. Понятно, что для получения исходных данных, подготовленных задачиком, в программе необходимо использовать специальные *методы ввода*. Эти статические методы определены в классе PT, являющемся *предком* класса MyTask. Поэтому при вызове методов ввода в функции Solve можно не указывать имя класса, в котором они определены.

В соответствии с подходом к организации ввода, принятым в стандартной библиотеке .NET, для ввода используются *функции без параметров*, каждая из которых возвращает один элемент исходных данных определенного типа. В задачнике Programming Taskbook предусмотрены функции для ввода данных всех базовых типов:

- ❑ GetBool – для ввода логических данных (типа bool);
- ❑ GetInt – для ввода целочисленных данных (типа int);
- ❑ GetDouble – для ввода вещественных данных (типа double);
- ❑ GetChar – для ввода символьных данных (типа char);
- ❑ GetString – для ввода строковых данных (типа string).

Этих функций достаточно для того, чтобы организовать ввод любых данных, входящих в задания групп LinqBegin, LinqObj и LinqXml. Например, для ввода данных из задания LinqBegin4 можно использовать следующий фрагмент (здесь и далее будем приводить только описание функции Solve, поскольку остальная часть проекта не требует изменений):

```
public static void Solve()
{
    Task("LinqBegin4");
    char c = GetChar();
    string[] a = new string[GetInt()];
    for (int i = 0; i < a.Length; i++)
        a[i] = GetString();
}
```

В данном фрагменте вначале описывается и вводится символ *c*, затем описывается и создается строковый массив *a* (при этом считывается размер исходной последовательности, который указывается в конструкторе массива), после чего организуется цикл для ввода значений самих элементов массива.

При запуске программы окно задачника примет вид, подобный приведенному на рис. 6. Этот запуск уже не считается ознакомительным, поскольку в программе выполняются действия по вводу-выводу данных.

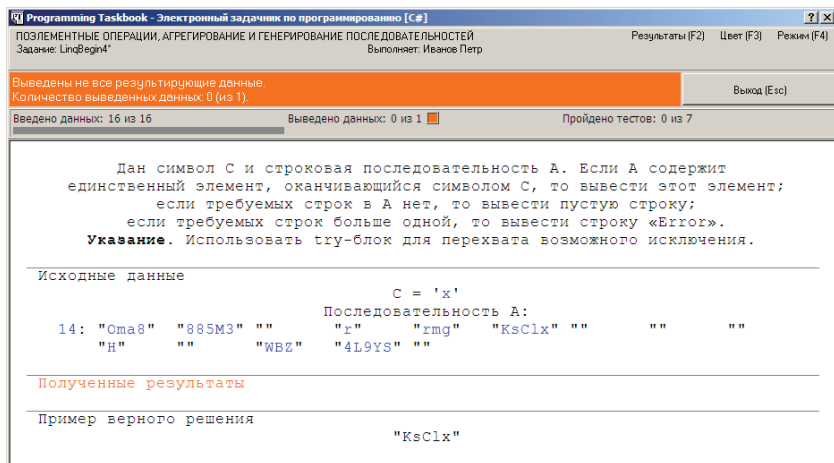


Рис. 6. Ошибочное выполнение задания LinqBegin4 (не выведены результаты)

Если запуск программы не является ни ознакомительным, ни демонстрационным, то в окне задачника появляется дополнительная панель (*панель индикаторов*), содержащая информацию о количестве введенных и выведенных данных, а также о количестве успешно пройденных тестовых испытаний программы.

В нашем случае в программе выполнен ввод всех исходных данных (о чем свидетельствует первый индикатор и связанный с ним текст), однако результирующие данные не выведены, поэтому тестовое испытание программы признано ошибочным. Описание ошибки приводится в информационной панели (расположенной рядом с кнопкой **Выход**), причем фон панели соответствует типу ошибки (для ошибок, связанных с вводом или выводом недостаточного числа исходных или результирующих данных, используется оранжевый цвет). Чтобы подчеркнуть, что ошибка связана с выводом результатов, рядом с текстом *второго* индикатора указывается прямоугольный маркер того же цвета, что и цвет фона информационной панели. Кроме того, этим же цветом выделяется заголовок раздела **Полученные результаты**. Все эти дополнительные элементы окна призваны упростить поиск и исправление ошибок, выявленных задачиком при выполнении программы.

Следует обратить внимание на то, что при ошибочном решении в окне задачника отображается не только раздел с полученными результатами (в нашем случае этот раздел является пустым), но и раздел с примером верного решения. Приведенная в нем информация часто оказывается полезной при поиске и исправлении допущенных ошибок.

Прежде чем перейти к обсуждению решения задачи, отметим одну дополнительную возможность, доступную при выполнении заданий группы LinqBegin, а именно наличие *вспомогательных функций ввода* `GetEnumeratorInt` и `GetEnumeratorString`, описанных в классе `MyTask` и обеспечивающих быстрый ввод целочисленных и строковых последовательностей. Благодаря этим функциям упрощаются действия по вводу исходных данных, а полученные решения становятся более краткими и наглядными.

С применением функции `GetEnumeratorString` фрагмент решения, отвечающий за ввод исходных данных, будет состоять всего из двух строк:

```
public static void Solve()
{
    Task("LinqBegin4");
```

```
char c = GetChar();  
var a = GetEnumerableString();  
}
```

В приведенном варианте программы использована возможность языка C#, появившаяся одновременно с технологией LINQ и тесно с ней связанная: *любая переменная может быть описана с помощью ключевого слова var, если данная переменная сразу инициализируется некоторым значением*. В подобной ситуации тип переменной автоматически определяется по типу инициализирующего выражения (говорят также, что тип переменной *выводится* из типа инициализирующего выражения). В нашем случае эта возможность позволяет использовать более краткую запись – слово var вместо типа `Enumerable<string>`, который имеет возвращаемое значение функции `GetEnumerableString`. Заметим, что в некоторых случаях (связанных с использованием так называемых *анонимных типов*, также появившихся в языке C# одновременно с технологией LINQ, – см. п. 8.2) без слова var при описании переменных обойтись невозможно.

При запуске нового варианта решения будет выведено прежнее сообщение об ошибке («Выведены не все результирующие данные»).

Приступим к обработке введенных данных, основанной на применении технологии LINQ. При выборе подходящего метода LINQ полезно обратиться к списку методов, указанному в преамбуле к той подгруппе, к которой относится задание (см. п. 2.1, а также рис. 5), поскольку именно эти методы должны применяться при решении задач данной подгруппы. Впрочем, при выполнении заданий из последующих подгрупп может потребоваться использовать не только те методы, которые в них изучаются, но и какие-либо из методов, рассмотренных в предшествующих подгруппах.

В нашем случае, очевидно, необходимо использовать один из методов, связанных с выбором *единственного* элемента: `Single` или `SingleOrDefault` (к поэлементным операциям относятся также методы `First` и `FirstOrDefault`, связанные с выбором *начального* элемента последовательности, `Last` и `LastOrDefault`, связанные с выбором ее *конечного* элемента, и `ElementAt` и `ElementAtOrDefault`, возвращающие элемент с требуемым индексом). Из всех методов, относящихся к поэлементным операциям, методы `Single` и `SingleOrDefault` характеризуются наиболее сложным поведением (именно по этой причине мы выбрали для разбора задание `LinqBegin4`).

Для любой поэлементной операции можно указать параметр-*предикат*, который позволяет отобрать те элементы, для которых требуется выполнить соответствующую операцию. Подобный параметр надо оформить в виде *лямбда-выражения*, принимающего элемент последовательности и возвращающего логическое значение (true, если элемент следует учитывать при выполнении операции, false в противном случае).

Примечание. Всюду в дальнейшем параметры-делегаты, используемые в методах интерфейсов LINQ to Objects и LINQ to XML, называются лямбда-выражениями. Разумеется, в качестве таких параметров можно указывать делегаты, определенные различными способами (например, оформленные в виде вспомогательного метода класса или описанные в виде анонимного метода по правилам C# версии 2.0 – см. п. 8.1), однако представление подобных параметров в виде лямбда-выражений является наиболее естественным и наглядным.

По условию задания LinqBegin4 требуется проанализировать элементы исходной последовательности, оканчивающиеся символом *C*. В качестве предиката в данном случае можно использовать следующее лямбда-выражение (подробное описание лямбда-выражений приводится в п. 8.1):

```
e => e.Length != 0 && e[e.Length - 1] == c
```

Отметим, что без первого условия указанное логическое выражение приводило бы к возбуждению исключительной ситуации *IndexOutOfRangeException* при обработке пустых строк (которые могут входить в исходную последовательность). Для связывания двух условий необходимо использовать операцию *&&* (логическое И с сокращенной схемой вычисления), при которой в случае ложного первого операнда второй операнд *не анализируется* (операцию *&* с полной схемой вычисления в данном случае применять нельзя, как нельзя и менять порядок следования данных условий). Вместо сравнения длин строк в первом условии можно использовать сравнение самих строк (*e != ""*), однако данное сравнение будет выполняться медленнее, чем сравнение длин.

Указанный вариант реализации предиката не является единственно возможным. Если воспользоваться методом *EndsWith* класса *string*, то условие можно представить в виде этого единственного метода:

```
e => e.EndsWith(c.ToString())
```

В данном случае, однако, требуется явным образом преобразовать символ *c* к строковому типу, вызвав для него метод `ToString`. Кроме того, следует учитывать, что метод `EndsWith` требует сравнения *строк*, которое выполняется дольше, чем сравнение *символов*.

Выберем один из описанных предикатов и используем его в методе `Single`, возвращающем *единственный* элемент последовательности, удовлетворяющий указанному предикату. Полученный элемент надо передать задачику для проверки правильности решения; для этого следует использовать еще один статический метод класса `PT` – метод `Put`, который может принимать произвольное количество параметров базовых типов (логического, целочисленного, вещественного, символьного, строкового). В результате очередной вариант функции `Solve` с решением задачи примет вид:

```
public static void Solve()
{
    Task("LinqBegin4");
    char c = GetChar();
    var a = GetEnumerableString();
    Put(a.Single(e => e.Length != 0 && e[e.Length - 1] == c));
}
```

Полученная программа будет правильно обрабатывать последовательности, содержащие единственный элемент, оканчивающийся символом *c*, однако если последовательность не содержит таких элементов или их имеется более одного, то будет возбуждаться исключительная ситуация `InvalidOperationException` (рис. 7).

С исключительной ситуацией будут связываться два вида сообщений: «Sequence contains more than one matching element» («Последовательность содержит более одного подходящего элемента» – см. рис. 7) и «Sequence contains no matching element» («Последовательность не содержит подходящих элементов»).

Чтобы перехватить и обработать исключительную ситуацию, следует использовать конструкцию `try-catch`. Для доступа к объекту-исключению его надо описать в заголовке блока `catch`. Имея этот объект, можно обратиться к его свойству `Message`, содержащему текстовое описание исключения, по этому свойству определить причину возбуждения исключения и вывести требуемую строку (напомним, что по условию задания в случае отсутствия подходящих элементов надо вывести пустую строку, а в случае двух и более подходящих элементов – строку «Error»).

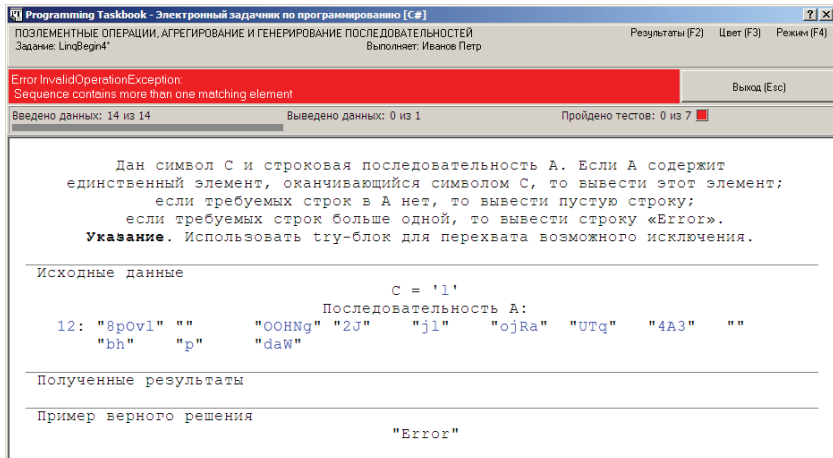


Рис. 7. Ошибочное выполнение задания LinqBegin4 (возбуждено исключение)

Получаем первый вариант правильного решения:

```
public static void Solve()
{
    Task("LinqBegin4");
    char c = GetChar();
    var a = GetEnumerableString();
    try
    {
        Put(a.Single(e => e.Length != 0 && e[e.Length - 1] == c));
    }
    catch (InvalidOperationException ex)
    {
        if (ex.Message.Contains("more"))
            Put("Error");
        else
            Put("");
    }
}
```

Решение можно упростить, если воспользоваться вариантом метода `Single` – `SingleOrDefault`, который *не приводит* к возбуждению исключения в случае, если в последовательности *отсутствуют* требуемые элементы (заметим, что аналогичным образом ведут себя и другие поэлементные операции, содержащие текст «`OrDefault`»: `FirstOrDefault`, `LastOrDefault` и `ElementAtOrDefault`). В подобной

ситуации метод `SingleOrDefault` возвращает *значение по умолчанию* (для числовых последовательностей это 0, для строковых последовательностей, как и для любых последовательностей со ссылочными данными, – значение `null`). Таким образом, при использовании метода `SingleOrDefault` исключительная ситуация будет возбуждаться только в случае, если исходная последовательность содержит *более одного* требуемого элемента.

Во втором варианте решения раздел `catch` стал более кратким, а раздел `try` увеличился:

```
public static void Solve()
{
    Task("LinqBegin4");
    char c = GetChar();
    var a = GetEnumerableString();
    try
    {
        string r = a.SingleOrDefault(e => e.Length != 0 &&
            e[e.Length - 1] == c);
        Put(r != null ? r : "");
    }
    catch
    {
        Put("Error");
    }
}
```

Вместо тернарной операции `?:` можно использовать операцию `??`, появившуюся в версии C# 2.0 и часто используемую в программах, применяющих технологию LINQ. Выражение `a ?? b` возвращает значение `a`, если оно отлично от `null`, и значение `b` в противном случае. Таким образом, оператор `Put` из раздела `try` можно переписать в виде:

```
Put(r ?? "");
```

В подобной ситуации отпадает необходимость в переменной `r`, и раздел `try` сокращается до единственного оператора:

```
public static void Solve()
{
    Task("LinqBegin4");
    char c = GetChar();
    var a = GetEnumerableString();
    try
```

```
{
    Put(a.SingleOrDefault(e => e.Length != 0 &&
        e[e.Length - 1] == c) ?? "");
}
catch
{
    Put("Error");
}
}
```

Поскольку переменная **a** используется в единственном месте программы, в ней тоже нет необходимости. Это позволяет еще более сократить полученное решение:

```
public static void Solve()
{
    Task("LinqBegin4");
    char c = GetChar();
    try
    {
        Put(GetEnumerableString().SingleOrDefault(e =>
            e.Length != 0 && e[e.Length - 1] == c) ?? "");
    }
    catch
    {
        Put("Error");
    }
}
```

Следует заметить, что при использовании технологии LINQ очень часто содержательная часть программы представляет собой *цепочку последовательных вызовов методов*; в нашем случае цепочка состоит всего из двух вызовов (`GetEnumerableString` и `SingleOrDefault`), однако дополнительно включает операцию `??`. Заметим также, что при использовании методов `FirstOrDefault`, `LastOrDefault` и `ElementAtOrDefault` необходимости в явной обработке исключений не возникает (хотя операция `??` для них также может оказаться полезной).

Выполнив семь успешных тестовых запусков программы, мы получим сообщение о том, что задание выполнено (рис. 8). При успешном тестовом запуске в окне отсутствует раздел с примером верного решения, поскольку этот раздел совпадал бы с разделом полученных результатов.

Если нажать на клавишу **F2**, можно отобразить окно с протоколом выполнения задания (этот протокол в зашифрованном виде хранит-

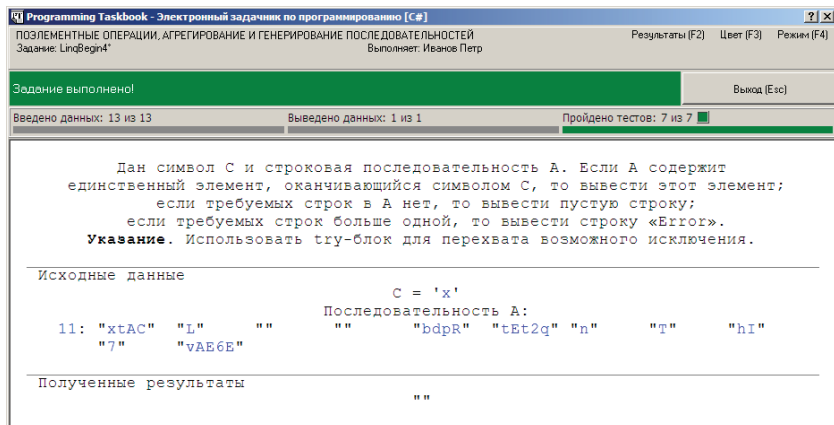


Рис. 8. Успешное выполнение задания LinqBegin4

ся в файле результатов results.dat). В нашем случае информация о ходе выполнения задания может иметь следующий вид:

| | | | |
|------------|--------|-------|--|
| LinqBegin4 | S16/05 | 11:43 | Ознакомительный запуск. |
| LinqBegin4 | S16/05 | 11:55 | Выведены не все результирующие данные. |
| LinqBegin4 | S16/05 | 12:03 | Error InvalidOperationException.--3 |
| LinqBegin4 | S16/05 | 12:08 | Задание выполнено! |

Символ, указываемый перед информацией о дате и времени тестового запуска, определяет язык программирования, на котором выполнялось задание (символ S соответствует языку C# (C Sharp); для языка VB.NET используется символ B). Числовое значение, указываемое в конце строки, означает количество тестовых запусков, выполненных подряд и завершившихся с одним и тем же результатом.

5.2. Операция агрегирования и генерирование последовательностей: LinqBegin15

В набор методов LINQ to Objects включены наиболее распространенные *операции агрегирования*, то есть операции, возвращающие некоторую общую характеристику последовательности:

- Count – количество элементов;

- Sum – сумма;
- Average – среднее арифметическое;
- Max – максимальное значение;
- Min – минимальное значение.

Все эти операции могут иметь параметр (лямбда-выражение). Для операции Count этот параметр является *предикатом* и позволяет определить *количество* элементов с требуемым свойством. Для остальных операций агрегирования с помощью параметра можно определить последовательность значений, к которой будет применена указанная операция (параметр задает правило, по которому для каждого элемента исходной последовательности будет определено соответствующее ему значение; такие параметры называются *селекторами*). Если параметр-селектор не указывается, то операция агрегирования применяется непосредственно к самим элементам последовательности. Для применимости операций Sum и Average необходимо, чтобы обрабатываемые элементы были *числами*, а для применимости операций Max и Min требуется, чтобы для элементов имели смысл отношения сравнения «меньше» и «больше».

Предусмотрена также «универсальная» операция Aggregate, позволяющая определять нестандартные «агрегирующие» характеристики последовательности. Поскольку эта операция является наиболее сложной из агрегирующих операций, рассмотрим особенности ее использования на примере задачи LinqBegin15.

LinqBegin15. Дано целое число N ($0 \leq N \leq 15$). Используя методы Range и Aggregate, найти *факториал* числа N : $N! = 1 \cdot 2 \cdot \dots \cdot N$ при $N \geq 1$; $0! = 1$. Чтобы избежать целочисленного переполнения, при вычислении факториала использовать вещественный числовой тип.

После создания проекта-заготовки и его запуска на экране появится окно с примером исходных данных и правильных результатов (рис. 9).

Данная задача является одной из трех задач группы LinqBegin (LinqBegin13–LinqBegin15), в которых отсутствуют исходные последовательности; во всех этих задачах требуется сгенерировать последовательность с использованием метода Range. Метод Range относится к числу немногих методов LINQ, не являющихся методами расширения; иными словами, его нельзя применять к *уже имеющейся* последовательности. Для вызова этого и подобных ему методов необходимо обращаться к классу Enumerable (напомним,

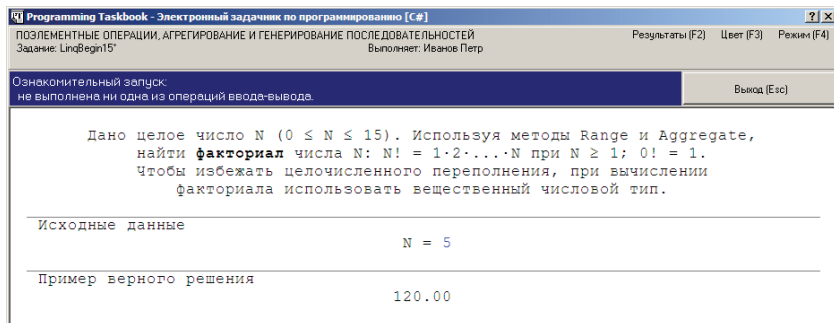


Рис. 9. Ознакомительный запуск задания LinqBegin15

что именно в этом классе определены все методы, входящие в интерфейс LINQ to Objects; в том числе и все методы расширения).

При решении данной задачи с использованием технологии LINQ мы должны вначале создать вспомогательную последовательность целых чисел (со значениями от 1 до N), после чего перемножить эти числа. Операция перемножения элементов последовательности является агрегирующей операцией; ее следует выполнять с помощью метода `Aggregate`, поскольку для перемножения элементов не предусмотрено специального метода агрегирования.

Примечание. Описанный выше алгоритм представляет собой перевод «на язык LINQ» обычного цикла с параметром (цикла `for`). Понятно, что для рассматриваемой ситуации, связанной с вычислением факториала, применение технологии LINQ не может считаться оправданным, поскольку с помощью цикла `for` требуемое значение можно вычислить и быстрее, и проще. Однако на примере подобной задачи удобно изучить особенности рассматриваемых методов LINQ, что, собственно, и является целью любого задания из группы LinqBegin.

Опишем параметры, предусмотренные для методов `Range` и `Aggregate`.

В методе `Range` следует указывать два параметра: начальное значение элемента целочисленной последовательности и количество ее элементов (каждый последующий элемент будет на 1 больше предыдущего). Подчеркнем, что второй параметр определяет не значение конечного элемента, а количество элементов, поэтому для генерации, например, последовательности чисел от 2 до 10 надо использовать набор параметров (2, 9).

Метод `Aggregate` может иметь три варианта набора параметров; опишем наиболее часто используемый вариант с двумя параметрами. Первый параметр представляет собой начальное значение вычисляемой характеристики *a* (такая характеристика называется *аккумулятором*, так как в ней «аккумулируется» информация обо всех элементах последовательности). Второй параметр является лямбда-выражением с параметрами (*a*, *e*), в котором определяется, каким образом к уже имеющемуся значению аккумулятора *a* будет добавляться значение очередного элемента последовательности *e*.

Примечание. Имеется вариант метода `Aggregate` с единственным параметром – лямбда-выражением. В этом варианте в качестве начального значения аккумулятора берется первый элемент обрабатываемой последовательности, а лямбда-выражение используется при обработке остальных ее элементов. Такой вариант метода является менее универсальным, поскольку, во-первых, он может применяться только к непустым последовательностям, и, во-вторых, в нем аккумулятор не может иметь тип, отличный от типа элементов последовательности.

Приведем первый вариант решения задачи:

```
public static void Solve()
{
    Task("LinqBegin15");
    int n = GetInt();
    double f = Enumerable.Range(1, n).Aggregate(1, (a, e) => a * e);
    Put(f);
}
```

В данном варианте вначале вводится исходное число целого типа, которое сохраняется в переменной *n*, затем методом `Range` создается последовательность целых чисел от 1 до *n*, к которой сразу применяется метод `Aggregate` со вторым параметром (лямбда-выражением), обеспечивающим накопление произведения. В соответствии с условием задачи полученное произведение записывается в переменную *вещественного* типа, которая затем выводится методом `Put`.

При тестировании данного решения мы обнаружим, что для исходных значений *n*, не превосходящих 12, факториал вычисляется правильно, а для значений, равных или больших 13, получается неверный результат (см. рис. 10).

Анализируя полученные результаты, нетрудно заметить, что ошибочное значение выводится в случае, когда значение факториала превышает максимальное значение типа `int`.

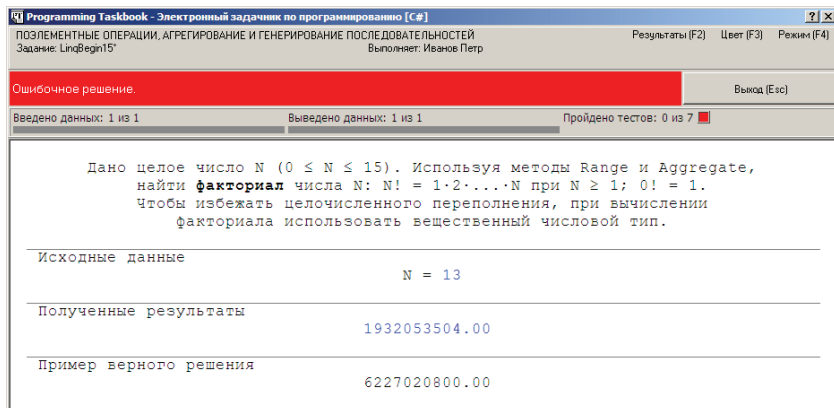


Рис. 10. Ошибочное выполнение задания LinqBegin15

Причина ошибки заключается в том, что в нашей программе при вычислении произведения *не используется* вещественный тип, поскольку тип аккумулятора *а* «выводится» из типа первого параметра, в качестве которого мы указали *целочисленную* константу 1. Таким образом, все операции умножения выполняются над целыми числами, а в этом случае при получении результата, превосходящего максимальное целое число, возникает ситуация *целочисленного переполнения*. В зависимости от настроек компилятора эта ситуация либо приводит к возбуждению исключения, либо (по умолчанию) результат просто урезается за счет отбрасывания старших двоичных разрядов. В нашем случае имеет место второй вариант: метод Aggregate завершается без возбуждения исключения, однако возвращает ошибочное значение. Последующее приведение этого значения (целого типа) к вещественному типу, которое выполняется автоматически при его присваивании вещественной переменной *f*, уже не может исправить ранее возникшую ошибку.

Итак, несмотря на использование вещественной переменной *f*, вычисление факториала выполняется с применением только целочисленного умножения, что и приводит (иногда) к ошибочному результату. Для правильного решения необходимо изменить метод Aggregate таким образом, чтобы в процессе самого *вычисления* произведения использовалось вещественное умножение. Один из возможных способов исправления состоит в явном указании типов параметров лямбда-выражения:

```
double f = Enumerable.Range(1, n)
    .Aggregate(1, (double a, int e) => a * e);
```

В данном случае тип параметра-аккумулятора *a* в лямбда-выражении не «выводится» из типа первого параметра метода `Aggregate`, а определяется явным образом как `double`, поэтому результат вычисляется как вещественное число, с применением операции вещественного умножения. Заметим, что для параметра *e* произвол в выборе типа невозможен: при явном указании типов мы должны описать параметр *e* как *целочисленный*, поскольку целочисленной является сама обрабатываемая последовательность.

После указанного исправления программа будет правильно обрабатывать любые значения *N* из диапазона 0–15 (см. рис. 11).

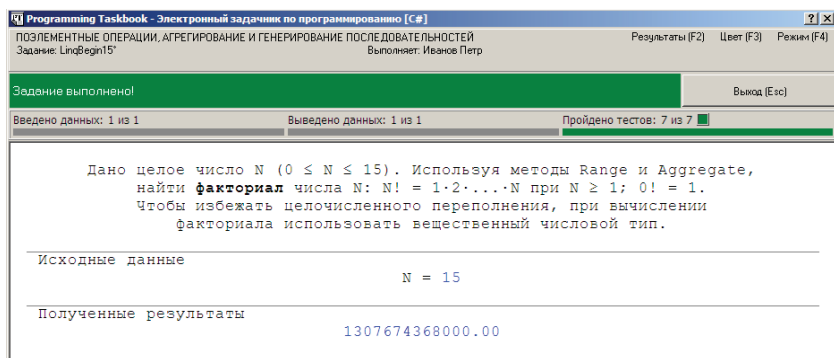


Рис. 11. Успешное выполнение задания LinqBegin15

Возможен правильный вариант решения, не требующий явного указания типов параметров лямбда-выражения, поскольку тип переменной-аккумулятора может быть выведен из типа первого параметра метода `Aggregate`, определяющего начальное значение аккумулятора. Таким образом, достаточно указать в качестве этого начального значения *вещественную* единицу (измененный параметр выделен полужирным шрифтом):

```
double f = Enumerable.Range(1, n).Aggregate(1.0, (a, e) => a * e);
```

Завершая обсуждение данного задания, представим его решение в более краткой форме. Поскольку и переменная *n*, и переменная *f*

имеют в последующем тексте программы по единственному вхождению, эти вхождения можно заменить на выражения, использованные для инициализации соответствующих переменных. В результате решение задачи может быть представлено в виде *единственного* оператора, включающего и ввод исходного числа, и его обработку, и вывод результата:

```
public static void Solve()
{
    Task("LinqBegin15");
    Put(Enumerable.Range(1, GetInt())
        .Aggregate(1.0, (a, e) => a * e));
}
```

Примечание. При анализе полученного алгоритма может возникнуть желание сократить размер исходной последовательности, генерируемой методом `Range`, удалив из нее первый элемент (со значением 1). Для этого достаточно изменить список параметров метода `Range` следующим образом: `(2, GetInt() - 1)`. Однако при тестировании данного варианта обнаружится, что он неправильно работает для случая $N = 0$, поскольку в этом случае второй параметр принимает недопустимое отрицательное значение. Описанный ранее алгоритм не требует специальной обработки особого случая: когда второй параметр метода `Range` равен 0, этот метод возвращает пустую последовательность, а использованный в программе вариант метода `Aggregate` при обработке пустой последовательности просто возвращает начальное значение аккумулятора, то есть вещественное число 1.

5.3. Фильтрация, сортировка, теоретико-множественные операции: LinqBegin31

Во второй подгруппе группы `LinqBegin` (см. п. 2.2) рассматривается большое количество методов LINQ, связанных со следующими категориями запросов:

- ❑ отбор части элементов последовательности, удовлетворяющих определенному признаку (*фильтрация*);
- ❑ изменение порядка следования элементов (*инвертирование и сортировка*);
- ❑ теоретико-множественные операции – *объединение* (Union), *пересечение* (Intersect) и *разность* (Except).

Все методы, связанные с указанными категориями, формируют по исходной последовательности (в случае теоретико-множественных операций – по *двум* однотипным исходным последовательностям) *новую последовательность*, в которой тип элементов не изменяется, однако могут изменяться их количество (вплоть до количества, равного 0) и порядок следования. Отметим, что последовательности, полученные при выполнении теоретико-множественных операций, не содержат одинаковых элементов.

Рассмотрим задачу LinqBegin31, при решении которой потребуется использовать методы LINQ всех описанных выше категорий.

LinqBegin31. Дано целое число $K (> 0)$ и последовательность непустых строк A . Строки последовательности содержат только цифры и заглавные буквы латинского алфавита. Найти теоретико-множественное пересечение двух фрагментов A : первый содержит K начальных элементов, а второй – все элементы, расположенные после последнего элемента, оканчивающегося цифрой. Полученную последовательность (не содержащую одинаковых элементов) отсортировать по возрастанию длин строк, а строки одинаковой длины – в лексикографическом порядке по возрастанию.

После создания проекта-заготовки и его запуска на экране появится окно задачника с примером исходных данных и правильных результатов (рис. 12).

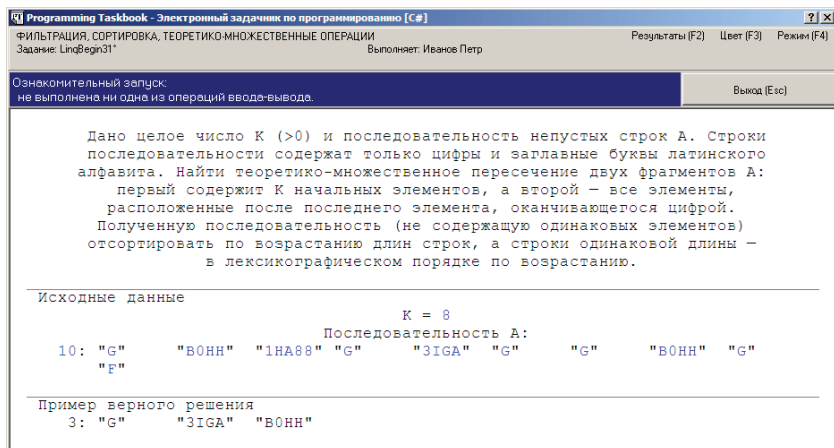


Рис. 12. Ознакомительный запуск задания LinqBegin31

Проанализируем приведенный на рисунке вариант исходных данных, чтобы понять, как был получен требуемый результат.

Исходная последовательность A содержит 10 элементов (будем их указывать, не заключая в кавычки и разделяя пробелами, поскольку среди них нет пустых строк и строк, содержащих пробелы):

G B0HH 1HA88 G ZIGA G G B0HH G F

Заметим, что среди элементов последовательности могут присутствовать элементы с одинаковыми значениями.

По исходной последовательности надо построить две новые. Первая из них – назовем ее B – содержит K начальных элементов (в данном случае K равно 8):

G B0HH 1HA88 G ZIGA G G B0HH

Вторая последовательность – назовем ее C – должна содержать все элементы, расположенные после последнего элемента, оканчивающегося цифрой. Последним элементом, оканчивающимся цифрой, в исходной последовательности является элемент 1HA88, поэтому последовательность C будет содержать семь элементов:

G ZIGA G G B0HH G F

К последовательностям B и C надо применить операцию *пересечения*; в результате будет сформирована последовательность, содержащая только те элементы, которые содержатся и в B , и в C , причем *без повторений*. При выполнении операции пересечения для последовательностей B и C вначале из последовательности B удаляются повторяющиеся элементы (остаются только их первые вхождения; таким образом, в нашем случае мы получим последовательность G B0HH 1HA88 ZIGA), а затем дополнительно удаляются все элементы, отсутствующие в последовательности C . В результате будет получена последовательность

G B0HH ZIGA

Заметим, что если изменить порядок операндов при вычислении пересечения, то есть найти пересечение последовательностей C и B , то результат будет содержать те же элементы, но *расположенные в другом порядке*. Действительно, при удалении из C повторяющихся элементов мы получим набор G ZIGA B0HH F, из которого затем будет удален элемент F, поскольку он отсутствует в последовательности B .

Нам осталось отсортировать полученную последовательность. В задании описывается вариант сортировки по *набору ключей*. Первым (главным) ключом является длина строкового элемента; таким образом прежде всего надо расположить элементы по возрастанию их длин. В нашем примере в результате такой сортировки последовательность не изменится:

G B0HH 3IGA

Вторым (подчиненным) ключом является само значение строкового элемента. Отсортировать по строковому ключу означает расположить данные в *лексикографическом порядке* этих ключей (иными словами, расположить ключи по возрастанию их первых символов, для одинаковых первых символов – по возрастанию вторых символов и т. д.; причем «пустой», то есть отсутствующий, символ считается меньшим любого существующего символа, цифра считается меньшей любой буквы, а буквы упорядочиваются по алфавиту). То обстоятельство, что строковые ключи являются подчиненными, означает, что лексикографическую сортировку необходимо проводить только среди строк *одинаковой длины*. У нас имеются две строки длины 4, причем цифра «3» считается меньшей, чем буква «B», поэтому после сортировки по подчиненному ключу последовательность примет вид

G 3IGA B0HH

Это и есть результирующая последовательность.

Приступим к реализации алгоритма решения задачи. Прежде всего необходимо организовать ввод исходных данных. Для ввода целого числа *K* воспользуемся методом `GetInt`, определенным в задачнике, а для ввода исходной последовательности используем вспомогательный метод `GetEnumerableString` (напомним, что этот метод описан в том же файле `LinqBegin31.cs`, в котором требуется запрограммировать решение):

```
int k = GetInt();  
var a = GetEnumerableString();
```

В данном случае нам необходимо сохранить исходную последовательность под определенным именем, поскольку на ее основе следует сформировать *две* вспомогательные последовательности (выше мы присвоили этим последовательностям имена *B* и *C*). Для большей

наглядности введем в нашу программу последовательности с этими же именами и сформируем их, используя подходящие методы фильтрации.

Имеются 5 методов фильтрации: `Take`, `TakeWhile`, `Skip`, `SkipWhile` и `Where`. Методы `Take` и `TakeWhile` возвращают *начальную* часть последовательности, методы `Skip` и `SkipWhile` – ее *конечную* часть, а «универсальный» метод `Where` позволяет получить выборку требуемых элементов независимо от их расположения в исходной последовательности (однако при этом порядок следования элементов в полученной выборке всегда будет соответствовать их порядку в исходной последовательности). Методы `Take` и `Skip` имеют единственный целочисленный параметр – количество начальных элементов последовательности, которые следует возвратить (для `Take`) или отбросить (для `Skip`). Если данное количество превышает размер последовательности, то `Take` возвращает всю последовательность, а `Skip` – пустую последовательность. Методы `TakeWhile`, `SkipWhile` и `Where` имеют единственный параметр-предикат (лямбда-выражение), определяющий условие фильтрации. Первый параметр предиката соответствует элементу фильтруемой последовательности; можно также использовать вариант предиката с двумя параметрами, в этом случае второй параметр определяет *индекс* обрабатываемого элемента (элементы индексируются от 0).

Для получения последовательности *B*, содержащей *K* начальных элементов последовательности *A*, достаточно использовать метод `Take`:

```
var b = a.Take(k);
```

Последовательность *C* должна содержать все элементы последовательности *A*, расположенные после последнего элемента, оканчивающегося цифрой. Хотя требуемая последовательность представляет собой *конечную* часть последовательности *A*, применить для ее получения метод `SkipWhile` не удастся, поскольку предикат, который распознает строки, оканчивающиеся цифрой, «сработает» уже на *первом* из таких элементов, в то время как нам требуется найти *последний* из них.

В подобной ситуации оказывается полезным метод `Reverse`, изменяющий порядок элементов последовательности на обратный (такая операция называется *инвертированием* последовательности). После инвертирования последовательности *A* те элементы, которые тре-

буется включить в последовательность *C*, будут расположены *перед* первым элементом, оканчивающимся цифрой, поэтому для их отбора достаточно использовать метод `TakeWhile`. Методы `Reverse` и `TakeWhile` можно вызвать в одном выражении, расположив их в цепочку:

```
var c = a.Reverse()  
        .TakeWhile(e => !char.IsDigit(e[e.Length - 1]));
```

Заметим, что в предикате можно не обрабатывать особую ситуацию, связанную с пустыми строками, так как *по условию задачи* все строки, входящие в исходную последовательность, являются непустыми.

На данном этапе выполнения задания было бы желательно проверить правильность полученных вспомогательных последовательностей *B* и *C*. Использовать для этого метод `Put` нельзя, так как этот метод предназначен для передачи задачику *результатов* выполнения задания, а в данном случае результатом должна быть *единственная* последовательность. Кроме того, при обработке данных, переданных методом `Put`, задачник контролирует их тип и количество и не отображает на экране «лишние» данные и данные неверных типов (то есть типов, отличающихся от типа очередного «правильного» элемента результирующих данных).

Для вывода отладочной информации в задачнике предусмотрены специальные методы `Show` и `ShowLine`. В качестве единственного параметра этих методов можно указать элемент данных любого типа; этот элемент будет автоматически преобразован к своему строковому представлению (методом `ToString`), которое будет выведено в специальном *разделе отладки* окна задачника. Раздел отладки располагается в нижней части окна и отображается на экране только в случае, если в нем содержится какая-либо информация. Метод `ShowLine` отличается от метода `Show` тем, что после вывода элемента данных выполняет автоматический переход на новую строку в разделе отладки.

Методы `Show` и `ShowLine` удобны для вывода скалярных данных, однако если в качестве их параметра указать последовательность, то на экране будет выведено не содержимое этой последовательности, а полное имя ее типа (поскольку так работает метод `ToString` для последовательностей). При этом, в силу особенностей реализации последовательностей `LINQ`, в имена их типов будут включаться и методы `LINQ`, использованные при их формировании. Например,

если к имеющемуся у нас фрагменту решения добавить оператор `Show(b)`, то в разделе отладки будет выведен следующий текст:

```
System.Linq.Enumerable+<TakeIterator>d__3a`1[System.String]
```

Разумеется, для вывода отладочной информации, связанной с элементами последовательности, можно перебрать эти элементы в цикле `foreach` и вызвать метод `Show` для каждого элемента, однако включение в программу подобного фрагмента сделает текст решения менее наглядным. В то же время возможность отладочной печати получаемых последовательностей может оказать существенную помощь при выполнении заданий, поэтому в программы-заготовки для заданий групп `LinqBegin`, `LinqObj` и `LinqXml` включен специальный *метод расширения* `Show`, обеспечивающий простой способ отладочной печати последовательностей.


Данный метод выводит в раздел отладки размер последовательности и все ее элементы; при этом размер отделяется от первого элемента двоеточием, а соседние элементы отделяются друг от друга пробелом.

В качестве необязательного параметра метода расширения `Show` можно указать *комментарий*, который будет добавлен перед выведенной последовательностью, а также *лямбда-выражение*, позволяющее настроить вид выводимых элементов (например, выражение `e => "\"" + e + "\""` будет заключать выводимые данные в кавычки, что может оказаться полезным, если в последовательность входят пустые строки).

После вывода последовательности метод `Show` обеспечивает автоматический переход на новую строку в разделе отладки.

Продemonстрируем применение метода расширения `Show`, добавив в наше решение два оператора:

```
b.Show("B");  
c.Show("C");
```

При запуске полученной программы на экране появится окно с разделом отладки, содержащим информацию о последовательностях *B* и *C* (рис. 13). На приведенном рисунке скрыт раздел с формулировкой заданий; для этого действия достаточно нажать клавишу **Del** (повторное нажатие восстанавливает раздел с формулировкой; кроме того, для восстановления раздела можно щелкнуть мышью на маркере , расположенном в правой части скрытого раздела).

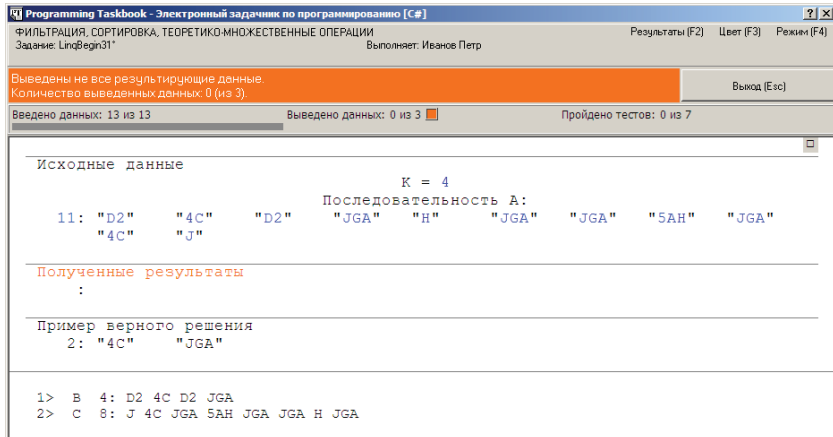


Рис. 13. Окно задачника с разделом отладки (задание LinqBegin31)

Анализируя данные в окне отладки, можно убедиться в том, что элементы последовательностей *B* и *C* найдены правильно. Порядок следования элементов последовательности *C* противоположен их исходному порядку, однако это обстоятельство не следует принимать во внимание, поскольку при последующем выполнении операции пересечения *B* и *C* порядок следования элементов в *C* учитываться не будет (учитывается лишь порядок следования элементов в *первом* операнде – последовательности *B*). Заметим, что для восстановления в последовательности *C* исходного порядка следования элементов было бы достаточно еще раз применить к этой последовательности операцию инвертирования *Reverse*.

Метод расширения *Show* возвращает последовательность, к которой он применен, поэтому его можно включать в цепочку других методов LINQ. Таким образом, для отладочной печати последовательностей *B* и *C* достаточно дополнить цепочки методов, использованные при построении этих последовательностей:

```
var b = a.Take(k).Show("B");
var c = a.Reverse()
    .TakeWhile(e => !char.IsDigit(e[e.Length - 1])).Show("C");
```

Для нахождения пересечения последовательностей *B* и *C* следует применить к первой из этих последовательностей (*B*) метод *Intersect*, указав вторую последовательность (*C*) в качестве параметра

этого метода. Полученную последовательность выведем не только в разделе отладки, но и в разделе результатов, используя *метод расширения* Put (этот вариант метода Put не имеет параметров; он должен указываться в конце цепочки методов LINQ):

```
b.Intersect(c).Show("D").Put();
```

Результат запуска программы приведен на рис. 14. Количество выведенных данных в разделе результатов совпадает с требуемым, однако их значения отличаются от правильных (элементы последовательности расположены не в том порядке), поэтому информационная панель содержит текст «Ошибочное решение» на красном фоне.

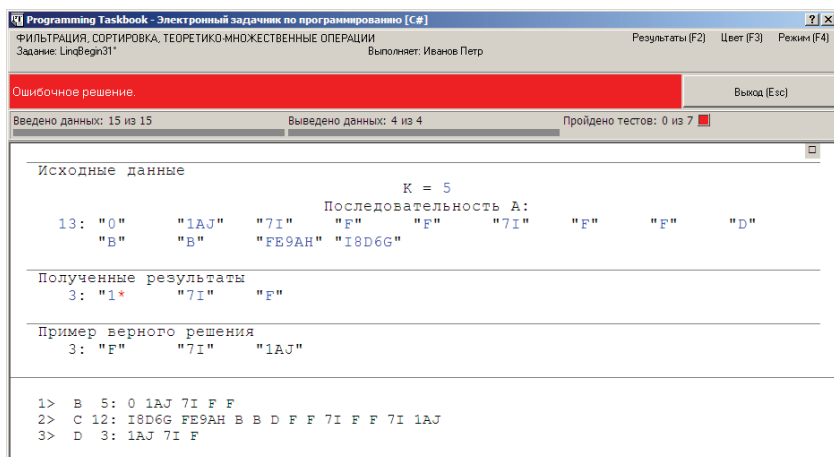


Рис. 14. Ошибочное выполнение задания LinqBegin31

Следует обратить внимание на одну особенность выведенных данных: для первого элемента полученной последовательности (1AJ) в разделе результатов выведен только один символ, после которого указана красная звездочка. Подобным образом выводятся строковые данные, размер которых превышает размер соответствующих им «правильных» данных. В нашем случае «правильный» элемент последовательности (F) имеет длину 1, поэтому при выводе в качестве этого элемента более длинных строк выполняется их «урезание» до 1 символа. Благодаря такому урезанию можно гарантировать, что неверные данные большого размера не испортят внешнего вида

раздела результатов (в частности, не скроют других отображенных в нем элементов и не выйдут за границы раздела). Для того чтобы увидеть элементы, отображенные в разделе результатов лишь частично, достаточно навести указатель мыши на данный раздел; при этом около указателя мыши появится всплывающая подсказка, содержащая полный текст всех «урезанных» данных (рис. 15).

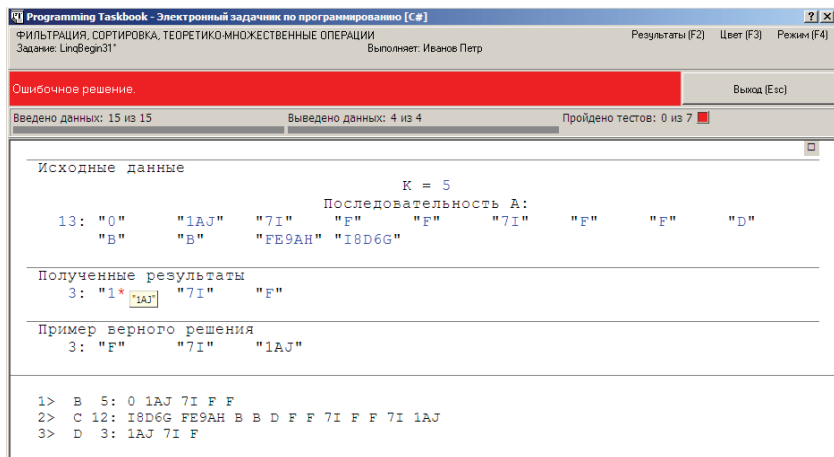


Рис. 15. Окно задачника с всплывающей подсказкой (задание LinqBegin31)

Заметим, что в разделе отладки элементы пересечения выведены полностью (с пересечением мы связали комментарий «D»).

Осталось выполнить сортировку полученной последовательности.

Для сортировки последовательностей предусмотрены четыре метода LINQ: `OrderBy`, `OrderByDescending`, `ThenBy` и `ThenByDescending`. Методы, имена которых оканчиваются словом `Descending`, обеспечивают сортировку по убыванию ключа; при отсутствии этого слова выполняется сортировка по возрастанию ключа. Методы `OrderBy` и `OrderByDescending` должны использоваться при сортировке по главному ключу; сортировка по подчиненным ключам (если они имеются) должна выполняться методами `ThenBy` и `ThenByDescending`. Ключ сортировки в каждом методе определяется с помощью лямбда-выражения; параметром лямбда-выражения является элемент последовательности. Если ключ сортировки имеет тип, не реализующий интерфейс `IComparable` (это означает, что для значений

этого типа не определены отношения «меньше» и «больше»), или при сортировке надо использовать способ упорядочивания, отличный от стандартного, то в методе сортировки необходимо указать второй параметр `comparer` типа `Comparer<Key>`, где `Key` обозначает тип ключа. Поскольку все базовые типы .NET, в том числе типы `int` и `string`, реализуют интерфейс `IComparable`, при выполнении заданий группы `LinqBegin` не требуется использовать варианты методов сортировки с двумя параметрами.

Примечание. Все методы LINQ, связанные с сортировкой, являются устойчивыми. Это означает, что относительное расположение элементов с одинаковыми ключами в результате сортировки не изменяется. Например, если выполняется сортировка по длине строк, а последовательность содержит несколько строк одинаковой длины, то в отсортированной последовательности эти строки будут располагаться в том же порядке, что и в исходной последовательности.

При предварительном обсуждении задания `LinqBegin31` мы отметили, что описанный в нем способ сортировки требует двух ключей: главного, определяемого длиной строки, и подчиненного, определяемого самой строкой, причем для обоих ключей требуется сортировка по возрастанию. Таким образом, к последовательности-пересечению необходимо применить два метода сортировки: `OrderBy` с лямбда-выражением, возвращающим длину строкового элемента, и `ThenBy` с лямбда-выражением, возвращающим сам строковый элемент. Укажем эти методы между имеющимися методами `Show` и `Put`, разместив их для наглядности на отдельной строке:

```
b.Intersect(c).Show("D")
    .OrderBy(e => e.Length).ThenBy(e => e)
    .Put();
```

Запустив полученную программу, мы можем убедиться в том, что она правильно обрабатывает любые наборы исходных данных; в том числе и такие, для которых полученная последовательность не содержит ни одного элемента (в этом случае в разделе результатов выводится единственное число 0 – размер этой последовательности).

После прохождения семи тестовых испытаний в окне задачника будет выведено сообщение о том, что задание выполнено (рис. 16).

Вывод отладочной информации не влияет на проверку правильности предложенного решения. В приведенном окне раздел отладки

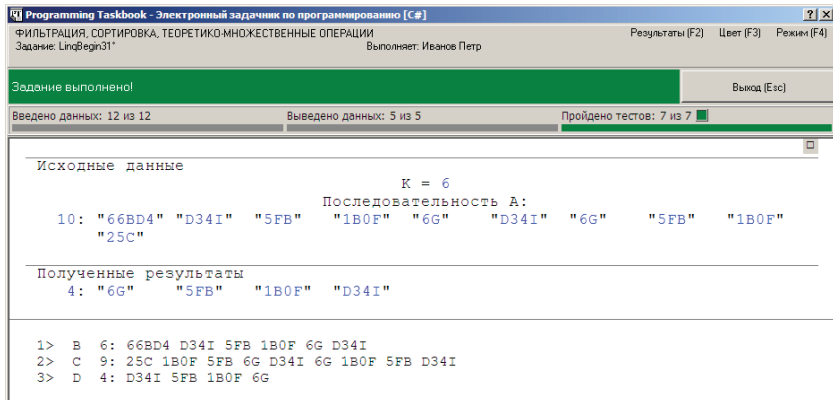


Рис. 16. Успешное выполнение задания LinqBegin31

отражает все промежуточные этапы решения: построение последовательностей *B* и *C*, а также нахождение их пересечения *D*.

Поскольку последовательности *B* и *C* используются в программе лишь при вызове метода `Intersect`, в решении можно обойтись без применения переменных *b* и *c*. Приведем полный текст функции `Solve` с подобным вариантом решения, сохранив в нем все вызовы отладочного метода `Show`:

```
public static void Solve()
{
    Task("LinqBegin31");
    int k = GetInt();
    var a = GetEnumerableString();
    a.Take(k).Show("B")
        .Intersect(a.Reverse()
            .TakeWhile(e => !char.IsDigit(e[e.Length - 1])).Show("C"))
        .Show("D")
        .OrderBy(e => e.Length).ThenBy(e => e)
        .Put();
}
```

5.4. Проецирование: LinqBegin43

В третьей подгруппе группы LinqBegin (см. п. 2.3) рассматриваются методы `Select` и `SelectMany`, действие которых обычно описывается термином «проецирование». В отличие от ранее рассмотренных методов LINQ, методы проецирования формируют по исходной

последовательности новую последовательность, элементы которой могут иметь тип, отличный от типа элементов исходной последовательности. При этом метод `Select` выполняет проецирование вида «один в один», то есть по каждому элементу исходной последовательности строит *единственный* элемент новой последовательности, а метод `SelectMany` выполняет проецирование вида «один во многие», при котором по одному элементу исходной последовательности строится *набор* новых элементов, и все полученные элементы включаются в результирующую последовательность.

Таким образом, число элементов в исходной последовательности и последовательности, полученной из исходной с помощью метода `Select`, *всегда совпадает*, в то время как для метода `SelectMany` этого утверждать нельзя. Обычно в результате использования метода `SelectMany` получается последовательность, количество элементов в которой *превосходит* количество элементов в исходной последовательности, но возможна ситуация, при которой после применения метода `SelectMany` число элементов *уменьшится*, по сравнению с исходной последовательностью. Это объясняется тем, что при проецировании «один во многие» некоторые элементы могут переводиться в *пустые* наборы, которые никак не влияют на состав полученной последовательности.

Методы проецирования принимают единственный параметр – лямбда-выражение, определяющее правило, согласно которому по элементу исходной последовательности будет построен элемент (для метода `Select`) или набор элементов (для метода `SelectMany`) новой последовательности. На тип возвращаемого значения лямбда-выражения для метода `Select` не накладывается никаких ограничений, в то время как возвращаемое значение лямбда-выражения для метода `SelectMany` обязательно должно быть *последовательностью*, то есть реализовывать интерфейс `IEnumerable<T>` для некоторого типа `T`. Заметим, что среди базовых типов .NET только тип `string` может интерпретироваться как последовательность, а именно как последовательность входящих в строку *символов*.

Лямбда-выражение, используемое в методах проецирования, как и лямбда-выражение для некоторых методов фильтрации (`Where`, `TakeWhile` и `SkipWhile` – см. п. 5.3), может содержать второй, необязательный параметр – *индекс* обрабатываемого элемента последовательности.

Одно и то же лямбда-выражение, указанное в методах `Select` и `SelectMany`, приведет к созданию различных последовательностей

(более того, последовательностей *разного типа*). Если, например, лямбда-выражение порождает по исходному элементу последовательность целых чисел, то после применения метода Select с этим лямбда-выражением будет получена *последовательность целочисленных последовательностей*, в то время как после применения метода SelectMany с этим же лямбда-выражением будет получена *последовательность целых чисел*, взятых из всех последовательностей, возвращенных лямбда-выражением.

Последовательности, элементы которых являются последовательностями, называются *иерархическими*. Метод SelectMany позволяет преобразовать иерархическую последовательность в «плоскую», элементами которой являются элементы последовательностей, входящих в иерархическую последовательность.

В качестве примера использования методов проецирования рассмотрим задачу LinqBegin43 – последнюю из задач, входящих в подгруппу, посвященную этим методам.

LinqBegin43. Дано целое число $K (> 0)$ и последовательность непустых строк A . Получить последовательность символов, которая определяется следующим образом: для первых K элементов последовательности A в новую последовательность заносятся символы, стоящие на нечетных позициях данной строки (1, 3, ...), а для остальных элементов A – символы на четных позициях (2, 4, ...). В полученной последовательности поменять порядок элементов на обратный.

При решении этой задачи, как и других задач данной подгруппы, могут потребоваться методы LINQ, описанные в предшествующих подгруппах, в частности методы фильтрации и сортировки.

После создания проекта-заготовки и его запуска на экране появится окно задачника с примером исходных данных и правильных результатов (рис. 17).

При выполнении задания необходимо обратить внимание на то, что в условии задачи символы в строках *нумеруются от 1*. Чтобы подчеркнуть этот факт, в формулировке приводятся номера позиций, которые считаются нечетными (1, 3, ...) и четными (2, 4, ...). Понятие *порядкового номера* и нумерация от 1 используется в формулировках всех задач, входящих в состав электронного задачника Programming Taskbook; это позволяет избежать неоднозначности, возникающей при употреблении понятия *индекса* и связанной с тем, что в различных языках программирования применяются

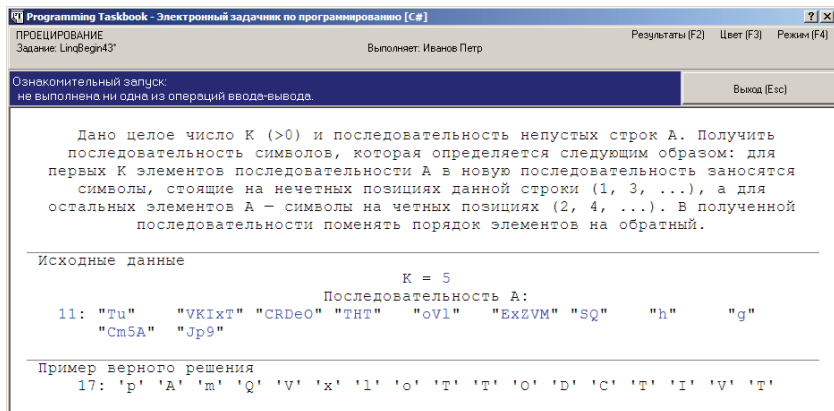


Рис. 17. Ознакомительный запуск задания LinqBegin43

разные варианты индексации элементов последовательностей (от 0 или от 1).

Если перейти от порядковых номеров к индексам символов в строке, то условие отбора требуемых символов будет выглядеть следующим образом: для первых K строк исходной последовательности в новую последовательность надо включать символы с *четными индексами* (0, 2, ...), а для оставшихся строк – символы с *нечетными индексами* (1, 3, ...). В качестве примера приведем набор строк, показанный на рис. 17, и подчеркнем те символы, которые должны быть включены в результирующую последовательность (строки будем отделять друг от друга пробелами; после первых 5 строк добавим несколько дополнительных пробелов):

Tu VKIXt CRDeO THT oVl ExZVM SQ h g Cm5A Jp9

Если перебрать подчеркнутые символы в порядке справа налево, то мы получим последовательность, указанную в примере верного решения.

Каждый элемент исходной строковой последовательности порождает несколько элементов результирующей символьной последовательности (в нашем примере – от одного до трех; имеются также две односимвольные строки, которые не вносят вклад в формируемую последовательность). Поэтому для решения задачи следует использовать метод `SelectMany`. Определим вид лямбда-выражения для данного метода.

Вначале предположим, что e – это одна из K начальных строк исходной последовательности (то есть элемент последовательности с индексом от 0 до $K - 1$). Поскольку строку e можно рассматривать как последовательность входящих в нее символов, для отбора из нее символов с четными индексами можно использовать метод `Where` с лямбда-выражением, содержащим два параметра:

```
e.Where((c, i) => i % 2 == 0)
```

В случае если e – одна из оставшихся строк (с индексом, большим или равным K), из нее надо отобрать символы с нечетными индексами, дающими при делении на 2 остаток, равный 1:

```
e.Where((c, i) => i % 2 == 1)
```

Отметим, что при проверке целого числа на нечетность эта формула будет верной только в случае *неотрицательных* целых чисел, поскольку для отрицательных чисел i результатом операции $i \% 2$ будет число -1 . В нашем случае приведенной формулой можно пользоваться, так как индексы элементов последовательности являются неотрицательными.

Если предположить, что элемент e имеет индекс n , то можно привести простую формулу, позволяющую определить для него требуемый остаток от деления (в формуле используется тернарная операция `?:`):

```
n < k ? 0 : 1
```

Таким образом, если элемент e исходной последовательности имеет индекс n , то набор символов, который этот элемент должен добавлять в результирующую последовательность, определяется следующим выражением (скобки, в которые заключено выражение с тернарной операцией, являются обязательными):

```
e.Where((c, i) => i % 2 == (n < k ? 0 : 1))
```

Указанное выражение следует использовать в лямбда-выражении для метода `SelectMany`:

```
.SelectMany((e, n) => e.Where((c, i) => i % 2 == (n < k ? 0 : 1)))
```

Осталось организовать ввод исходных данных и выполнить инвертирование и вывод полученной последовательности:

```
public static void Solve()
{
    Task("LinqBegin43");
    int k = GetInt();
    GetEnumerableString()
        .SelectMany((e, n) =>
            e.Where((c, i) => i % 2 == (n < k ? 0 : 1)))
        .Reverse().Put();
}
```

Выполнив семь тестовых запусков программы, мы получим сообщение «Задание выполнено!».

При выполнении заданий, связанных с созданием наборов вспомогательных последовательностей, бывает полезно отображать в окне задачника (в разделе отладки) информацию, связанную с этими последовательностями. Использование метода расширения Show, реализованного в задачнике (см. п. 5.3), позволяет очень просто организовать подобную отладочную печать. В качестве примера приведем вариант предыдущей программы, в котором дополнительно выводятся на экран все символьные последовательности, порожденные строками, входящими в исходную последовательность (добавленный в программу вызов метода Show выделен полужирным шрифтом):

```
public static void Solve()
{
    Task("LinqBegin43");
    int k = GetInt();
    GetEnumerableString()
        .SelectMany((e, n) =>
            e.Where((c, i) => i % 2 == (n < k ? 0 : 1)).Show())
        .Reverse().Put();
}
```

На рис. 18 приведен вид окна, полученного после запуска нового варианта программы (для уменьшения размеров окна в нем скрыт раздел с формулировкой задания).

Примечание. Если метод Show вызывается для последовательности, формируемой в каком-либо лямбда-выражении, то возможна ситуация, когда при выполнении программы окно отладки не появится на экране. Данная ситуация связана с особенностью обработки запросов LINQ, возвращающих последовательности. Подобные запросы являются отложенными, то есть выполняются

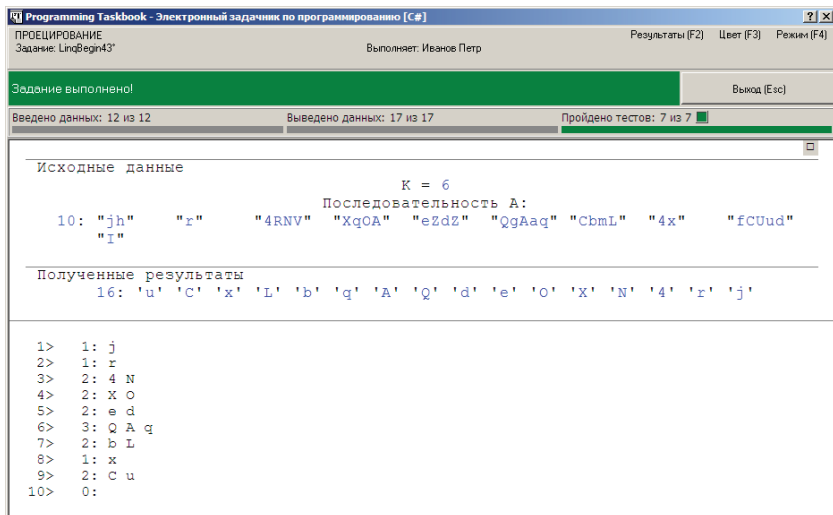


Рис. 18. Успешное выполнение задания LinqBegin43

не в момент их появления в программе, а лишь впоследствии, когда к полученной последовательности будет применена одна из операций, выполняющихся немедленно (например, поэлементная операция, операция агрегирования или операция экспортирования), или когда будет организован цикл по перебору элементов этой последовательности (подобные циклы организуются, в частности, в методах Show и Put). Если ни одно из указанных действий в программе не выполняется, то не будут выполнены и отложенные запросы, а значит, и входящие в их лямбда-выражения методы Show. Для моделирования подобной ситуации достаточно удалить в предыдущей программе вызов метода Put.

Завершая обсуждение задачи LinqBegin43, отметим, что для нее (как и для многих других задач группы LinqBegin) можно реализовать несколько вариантов решения, используя различные комбинации методов LINQ. Например, учитывая то обстоятельство, что в задании требуется по-разному обрабатывать первые K элементов исходной последовательности и ее оставшиеся элементы, можно использовать методы фильтрации Take и Skip, применив метод SelectMany к каждой из возвращенных этими методами последовательностей. При этом упростится запись лямбда-выражений для методов SelectMany, однако потребуются дополнительно организовать *сцен-*

ление двух полученных частей результирующей последовательности, используя метод `Concat`. Приведем программную реализацию данного варианта решения:

```
public static void Solve()
{
    Task("LinqBegin43");
    int k = GetInt();
    var a = GetEnumerableString();
    a.Take(k).SelectMany(e => e.Where((c, i) => i % 2 == 0))
        .Concat(a.Skip(k)
            .SelectMany(e => e.Where((c, i) => i % 2 == 1)))
        .Reverse().Put();
}
```

Применение методов `Take` и `Skip` делает алгоритм решения более наглядным, однако требует вызова большего количества методов LINQ, в том числе двух вызовов метода `SelectMany` с почти одинаковыми лямбда-выражениями. Предложенный ранее вариант решения, несколько проигрывая в наглядности, является более кратким и эффективным.

5.5. Объединение: LinqBegin52, LinqBegin54

Четвертая, завершающая подгруппа группы LinqBegin (см. п. 2.4) посвящена наиболее сложным действиям, выполняемым над последовательностями: *объединению* и *группировке*. В данном пункте рассматриваются различные варианты объединения последовательностей и связанные с ними методы LINQ. Кроме того, в нем описываются средства языка C#, связанные с построением *выражений запросов* (см. п. 5.5.3). Группировке последовательностей посвящен пункт 5.6.

5.5.1. Объединение последовательностей и его виды

Под *объединением* двух последовательностей понимается построение новой последовательности, элементы которой являются комбинациями элементов исходных последовательностей. Для построения различных вариантов объединений последовательностей по ключу предусмотрены два специальных метода LINQ – `Join` и `GroupJoin`;

кроме того, при построении некоторых вариантов объединений требуется использовать методы проецирования `Select` и `SelectMany` (см. п. 5.4), а также вспомогательный метод `DefaultIfEmpty`, обеспечивающий специальную обработку пустых последовательностей.

Одним из вариантов объединения является *перекрестное объединение* (*декартово произведение*), при котором элементы результирующей последовательности строятся по *всевозможным парам* элементов исходных последовательностей (первым компонентом пары является один из элементов первой последовательности, вторым – один из элементов второй последовательности). Часто требуется построить подмножество перекрестного объединения, отобрав только пары, удовлетворяющие дополнительному условию. Для построения перекрестного объединения (или его подмножества, если условие отбора пар *нельзя* свести к проверке *совпадения ключей*, вычисляемых по первому и второму компонентам пары) надо использовать комбинацию методов `SelectMany` и `Select`. Подобному варианту объединения посвящена задача LinqBegin52, которая рассматривается в п. 5.5.2.

В важном частном случае, когда в объединение надо включать только пары элементов, имеющих *одинаковые ключи*, следует использовать метод `Join`. Этот метод вызывается для первой последовательности (называемой *внешней* последовательностью); вторая (*внутренняя*) последовательность указывается в качестве первого параметра метода, а два следующих параметра являются лямбда-выражениями, задающими способ нахождения ключа для внешней и внутренней последовательности соответственно. Последний, четвертый параметр метода `Join` представляет собой лямбда-выражение с двумя параметрами – элементами внешней и внутренней последовательности, имеющими одинаковый ключ; это лямбда-выражение предназначено для конструирования элемента результирующего объединения по паре элементов исходных последовательностей, указанных в качестве его параметров.

Объединение, построенное с помощью метода `Join`, называется *внутренним объединением*. Во внутреннее объединение включаются только те элементы внешней последовательности, для которых имеется хотя бы один элемент внутренней последовательности с таким же ключом.

Метод `GroupJoin` предназначен для построения *левого внешнего объединения*, в котором каждому элементу внешней (левой) последовательности сопоставляется *набор* элементов внутренней

(правой) последовательности с таким же ключом. Основным отличием левого внешнего объединения от внутреннего является то, что в левое внешнее объединение гарантированно включаются все элементы внешней последовательности, даже если какому-либо из них не соответствует ни один элемент внутренней последовательности (подобным элементам внешней последовательности ставится в соответствие *пустой* набор элементов внутренней последовательности). Метод `GroupJoin` имеет такой же набор параметров, как и метод `Join`, за одним исключением: лямбда-выражение, являющееся четвертым параметром метода `GroupJoin` и возвращающее элемент левого внешнего объединения, должно принимать в качестве своих параметров некоторый элемент внешней последовательности и *последовательность* элементов внутренней последовательности, имеющих тот же ключ, что и элемент внешней последовательности.

Поскольку элементы левого внешнего объединения строятся по *набору* элементов внутренней последовательности, соответствующему некоторому элементу внешней последовательности, подобное объединение называется *иерархическим* (в отличие от «плоского» внутреннего объединения, в котором каждый элемент строится по одному элементу внешней и одному элементу внутренней последовательности).

Возможен «плоский» вариант левого внешнего объединения, элементами которого являются пары элементов внешней и внутренней последовательности с одинаковыми ключами. Важным отличием подобного «плоского» левого внешнего объединения от внутреннего объединения является то, что оно включает *все* элементы внешней последовательности, в том числе и те, которым не соответствует ни один элемент внутренней последовательности (подобные элементы внешней последовательности обычно образуют пару с некоторым особым элементом, не равным ни одному элементу внутренней последовательности). Плоский вариант левого внешнего объединения строится с использованием метода `GroupJoin`, однако для него требуется дополнительно использовать методы `SelectMany` и `DefaultIfEmpty`. Подобному варианту объединения посвящена задача `LinqBegin54`, рассматриваемая в п. 5.5.4.

В версии .NET 4.0 набор методов LINQ, связанных с объединением, был дополнен методом `Zip`, позволяющим комбинировать две последовательности, объединяя их элементы с *одинаковыми индексами* (способ объединения определяется вторым параметром – лямбда-

выражением; «лишние» элементы более длинной последовательности не обрабатываются).

5.5.2. Построение перекрестного объединения: LinqBegin52

В качестве первого примера построения объединения рассмотрим задачу LinqBegin52, посвященную нахождению перекрестного объединения двух последовательностей.

LinqBegin52. Даны строковые последовательности A и B ; все строки в каждой последовательности различны, имеют ненулевую длину и содержат только цифры и заглавные буквы латинского алфавита. Получить последовательность всевозможных комбинаций вида « $E_A=E_B$ », где E_A – некоторый элемент из A , E_B – некоторый элемент из B , причем оба элемента оканчиваются цифрой (например, «AF3=D78»). Упорядочить полученную последовательность в лексикографическом порядке по возрастанию элементов E_A , а при одинаковых элементах E_A – в лексикографическом порядке по убыванию элементов E_B .

Указание. Для перебора комбинаций использовать методы SelectMany и Select.

После создания проекта-заготовки и его запуска на экране появится окно задачника (рис. 19).

В задании требуется построить перекрестное объединение тех элементов исходных последовательностей, которые оканчиваются цифрой. Поэтому первым шагом в решении является *фильтрация* исходных последовательностей в соответствии с указанным условием (для проверки правильности фильтрации будем выводить полученные последовательности в раздел отладки, используя метод расширения Show, описанный в п. 5.3):

```
var a = GetEnumerableString()  
    .Where(e => char.IsDigit(e[e.Length - 1])).Show("A");  
var b = GetEnumerableString()  
    .Where(e => char.IsDigit(e[e.Length - 1])).Show("B");
```

Для доступа к последнему символу строки e вместо выражения $e[e.Length - 1]$ можно использовать запрос LINQ $e.Last()$ (при этом строка e будет интерпретироваться как *последовательность символов*).

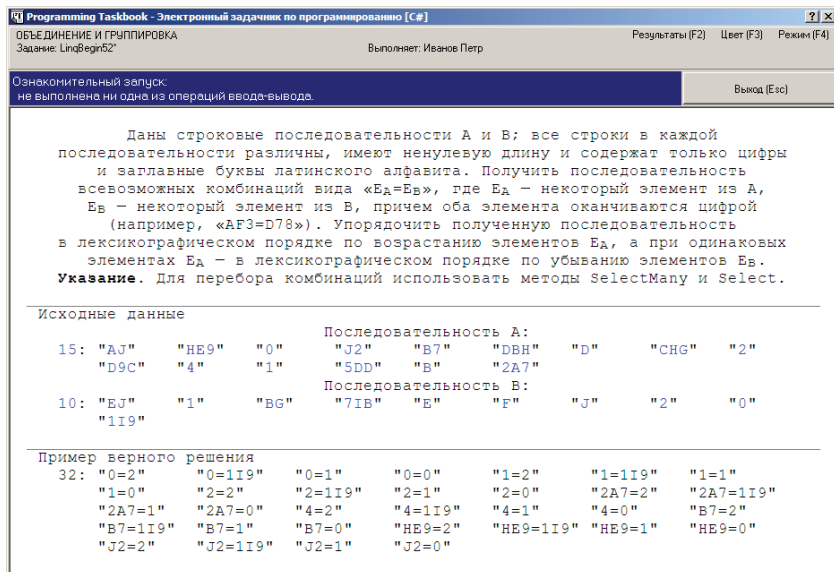


Рис. 19. Ознакомительный запуск задания LinqBegin52

Пусть e_1 обозначает некоторый элемент первой отфильтрованной последовательности. Тогда для получения всех пар перекрестного объединения, в которые входит элемент e_1 , достаточно выполнить следующее проецирование второй последовательности:

```
b.Select(e2 => e1 + "=" + e2)
```

Для различных элементов e_1 будут получены различные последовательности пар. В результирующее перекрестное объединение надо включить *все* элементы *всех* полученных последовательностей; подобное действие обеспечивается методом SelectMany:

```
a.SelectMany(e1 => b.Select(e2 => e1 + "=" + e2))
```

Итак, для построения перекрестного объединения следует использовать конструкцию, включающую вызов метода SelectMany для левой последовательности и вызов (в лямбда-выражении) метода Select для правой последовательности.

Примечание. Обращение к правой последовательности в приведенном выражении выполняется многократно, поэтому вмес-

то переменной *b* в данное выражение нельзя подставить выражение, использованное в качестве ее инициализатора (в противном случае будет выполняться многократный вызов метода `GetEnumeratorString`, что приведет к сообщению об ошибке «Попытка ввести лишние исходные данные»). Таким образом, в программе нельзя избавиться от вспомогательных переменных *a* и *b*.

Для завершения алгоритма осталось дополнить последнее выражение вызовом метода `Put`:

```
a.SelectMany(e1 => b.Select(e2 => e1 + "=" + e2)).Put();
```

Мы намеренно не учли условия задачи, связанного с сортировкой элементов полученной последовательности. При запуске приведенного варианта решения в окне задачника будут выведены все требуемые элементы объединения, однако порядок их следования будет отличаться от требуемого (см. рис. 20). Напомним, что красная звездочка отображается рядом с теми выведенными строковыми элементами, размер которых превышает размер «правильных» элементов; полный текст таких элементов можно отобразить во всплывающей подсказке, переместив указатель мыши в раздел результатов (на рисунке всплывающая подсказка изображена рядом с правой границей раздела результатов).

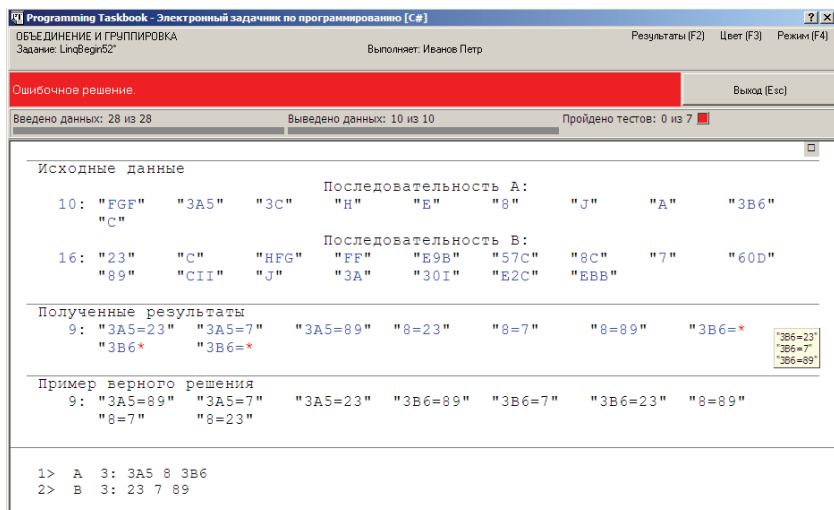


Рис. 20. Ошибочное выполнение задания LinqBegin52

Порядок элементов объединения определяется порядком следования элементов в левой последовательности, а для одинаковых элементов левой последовательности – порядком следования элементов в правой последовательности. Заметим, что указанное правило справедливо и для объединений, получаемых с помощью методов Join и GroupJoin.

Если бы в условии задачи использовался один и тот же вариант сортировки (по возрастанию или по убыванию) для левой и правой части полученных строк, то для упорядочивания этих строк требуемым образом было бы достаточно применить указанный вариант сортировки к результирующему объединению. Но в задании требуется упорядочить первые элементы пар в лексикографическом порядке по *возрастанию*, а вторые элементы пар (для одинаковых первых элементов) – по *убыванию*.

В подобной ситуации можно было бы «расщепить» методом Split полученные строки на две части, используя левую часть в качестве главного ключа в методе OrderBy, а правую часть – в качестве подчиненного ключа в методе ThenByDescending:

```
a.SelectMany(e1 => b.Select(e2 => e1 + "=" + e2))
  .OrderBy(e => e.Split('=')[0])
  .ThenByDescending(e => e.Split('=')[1])
  .Put();
```

Однако имеется более простой способ реализации требуемой сортировки: достаточно отсортировать нужным образом *исходные* отфильтрованные последовательности *a* и *b*. В результате получаем первый вариант правильного решения (в этом варианте сохранена отладочная печать последовательностей *a* и *b*, которая выполняется *после* их сортировки):

```
public static void Solve()
{
    Task("LinqBegin52");
    var a = GetEnumerableString()
        .Where(e => char.IsDigit(e[e.Length - 1]))
        .OrderBy(e => e).Show("A");
    var b = GetEnumerableString()
        .Where(e => char.IsDigit(e[e.Length - 1]))
        .OrderByDescending(e => e).Show("B");
    a.SelectMany(e1 => b.Select(e2 => e1 + "=" + e2)).Put();
}
```

После семи тестовых запусков программы в окне задачника будет выведено сообщение о том, что задание выполнено (рис. 21).

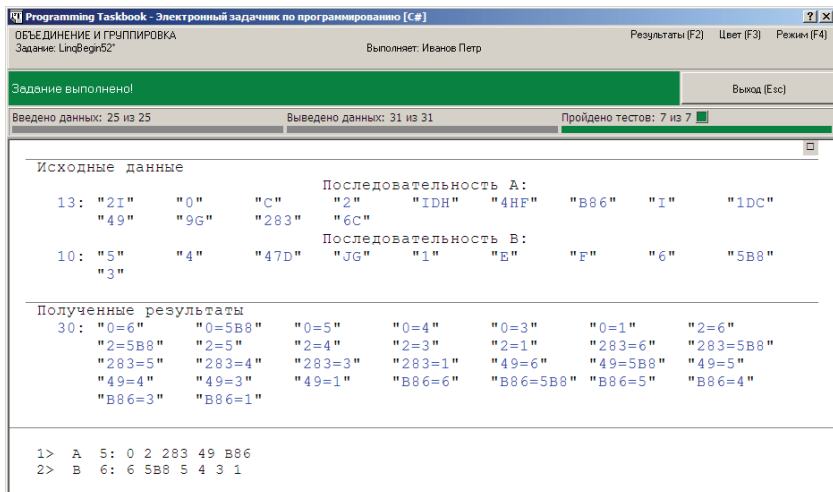


Рис. 21. Успешное выполнение задания LinqBegin52

5.5.3. Выражения запросов

В связи с появлением технологии LINQ в язык C# были добавлены новые конструкции, позволяющие оформлять запросы LINQ в виде, не требующем использования лямбда-выражений. Такой способ записи запросов LINQ получил название *выражений запросов* (query expressions); в некоторых книгах по технологии LINQ на русском языке он называется *синтаксисом, облегчающим восприятие запроса*. При построении сложных запросов, включающих ряд действий, связанных с фильтрацией, проецированием, сортировкой, объединением и группировкой, выражения запросов часто позволяют представить требуемые запросы в более наглядном виде, по сравнению с цепочками вызовов методов LINQ. Однако следует подчеркнуть, что выражения запросов предназначены лишь для *упрощения записи* сложных конструкций LINQ; любое выражение запроса в конечном итоге преобразуется компилятором к цепочке методов LINQ с лямбда-выражениями.

В данном пункте мы продемонстрируем использование выражений запросов при решении рассмотренной ранее задачи LinqBegin52.

Заметим, что применение выражений запросов наиболее оправдано при построении различного рода объединений, в особенности перекрестных. Именно поэтому в качестве первого примера применения выражений запросов мы выбрали задачу LinqBegin52.

Приведем фрагмент предыдущего решения этой задачи, непосредственно связанный с построением и выводом на экран перекрестного объединения (напомним, что перед этим выполнялись ввод, фильтрация и сортировка исходных последовательностей *a* и *b*):

```
a.SelectMany(e1 => b.Select(e2 => e1 + "=" + e2)).Put();
```

С применением выражения запроса этот же фрагмент можно записать более наглядным образом:

```
var c =  
    from e1 in a  
    from e2 in b  
    select e1 + "=" + e2;  
c.Put();
```

Выражение запроса начинается с конструкции *from e in a*, где переменная (или выражение) *a* определяет последовательность, а переменная *e* — связанный с этой последовательностью *перечислитель*, называемый также *переменной итерации* (range variable). Перечислитель обозначает *произвольный элемент последовательности*, который можно использовать в последующих конструкциях выражения запроса.

Одной из конструкций, которой можно завершать выражение запроса, является конструкция *select*; в ней указывается выражение, определяющее элемент формируемой последовательности. Обычно в этом выражении используется ранее введенный перечислитель исходной последовательности.

Важной особенностью выражений запросов является то, что в них допускается указывать *несколько* конструкций *from*; это позволяет использовать в последующей конструкции *select* перечислители всех последовательностей, указанных в конструкциях *from*, формируя тем самым перекрестное объединение этих последовательностей. В результате обработки подобных выражений запросов компилятор получает комбинации вызовов методов *SelectMany* и *Select*, подобных комбинации, приведенной выше.

Хотя обычно выражение запроса присваивается некоторой переменной, это не является обязательным. Допустимо использовать

выражение запроса в цепочке методов расширения; требуется лишь заключать это выражение в круглые скобки:

```
(from e1 in a
 from e2 in b
 select e1 + "=" + e2).Put();
```

Между конструкциями `from` и `select` можно указывать конструкцию `orderby`, позволяющую определить порядок перебора (то есть способ сортировки) элементов исходных последовательностей. В конструкции `orderby` ключи сортировки указываются через запятую, начиная с главного ключа; каждый ключ может снабжаться дополнительным модификатором `descending`, который обеспечивает сортировку по убыванию для данного ключа (по умолчанию выполняется сортировка по возрастанию ключа).

Продемонстрируем использование конструкции `orderby`, добавив ее в приведенное выше выражение запроса и удалив вызовы методов `OrderBy` из фрагмента, связанного с определением последовательностей `a` и `b`:

```
public static void Solve()
{
    Task("LinqBegin52");
    var a = GetEnumerableString()
        .Where(e => char.IsDigit(e[e.Length - 1]));
    var b = GetEnumerableString()
        .Where(e => char.IsDigit(e[e.Length - 1]));
    (from e1 in a
     from e2 in b
     orderby e1, e2 descending
     select e1 + "=" + e2).Put();
}
```

В выражениях запросов можно также использовать конструкцию `where`, обеспечивающую фильтрацию элементов. Данная конструкция указывается после конструкции `from`; в ней задается предикат, определяемый с помощью перечислителя из конструкции `from`.

Используя конструкцию `where`, мы можем перенести в выражение запроса и ту часть алгоритма, которая отвечает за отбор строк, оканчивающихся цифрой (для большей наглядности в данном варианте решения предикаты реализованы с использованием метода `Last`):

```
public static void Solve()
{
```

```
Task("LinqBegin52");  
var a = GetEnumerableString();  
var b = GetEnumerableString();  
(from e1 in a  
  where char.IsDigit(e1.Last())  
  from e2 in b  
  where char.IsDigit(e2.Last())  
  orderby e1, e2 descending  
  select e1 + "=" + e2).Put();  
}
```

Мы получили второй вариант решения, в котором все методы LINQ реализованы в виде фрагментов выражения запроса; при этом в полученном решении не осталось ни одного лямбда-выражения.

Заметим, что избавиться от переменных *a* и *b*, поместив вместо них в конструкции *from* явные вызовы функции *GetEnumerableString*, нельзя, поскольку компилятор преобразует эти конструкции *from* в комбинацию методов *SelectMany* и *Select*, в которой выполняется *многократное* обращение ко второй последовательности (см. примечание в п. 5.5.2).

С учетом последнего обстоятельства может возникнуть впечатление, что вариант решения, использующий выражение запроса, выполняет *больше* действий, чем вариант, основанный на вызовах методов LINQ. В самом деле, в первом варианте, приведенном в конце предыдущего пункта, отбор строк, оканчивающихся цифрой, выполняется при построении последовательностей *a* и *b*, то есть, *казалось бы*, один раз для каждой последовательности, а во втором варианте этот отбор выполняется в процессе построения перекрестного объединения, при котором обращение ко второй последовательности производится многократно (а значит, многократно проводится и отбор ее элементов).

На самом деле многократный отбор элементов для последовательности *b* будет проводиться и в первом варианте решения. Это связано с уже упоминавшимся выше (см. примечание в п. 5.4) *отложенным* характером выполнения запросов, возвращающих последовательности. При выполнении оператора

```
var b = GetEnumerableString()  
  .Where(e => char.IsDigit(e[e.Length - 1]))  
  .OrderByDescending(e => e).Show("B");
```

в переменную *b* записывается не результат выполнения методов *Where* и *OrderByDescending*, а *сами эти запросы* (в специальном

формате). Немедленно выполняются только вспомогательные методы `GetEnumerableString` и `Show`, реализованные в задачнике. При последующих обращениях к последовательности `b`, при которых требуется организовать перебор всех ее элементов, методы `Where` и `OrderByDescending` каждый раз *выполняются заново*.

Чтобы в этом убедиться, достаточно немного модифицировать приведенный выше фрагмент программы:

```
var b = GetEnumerableString()  
    .Where(e => char.IsDigit(e.Show("W").Last()))  
    .OrderByDescending(e => e.Show("O"));
```

В данной модификации при каждом вызове лямбда-выражения для методов `Where` и `OrderByDescending` в раздел отладки записывается аргумент этого лямбда-выражения (в виде последовательности символов), причем аргумент для лямбда-выражения метода `Where` снабжается комментарием «W», а аргумент для лямбда-выражения метода `OrderByDescending` – комментарием «O». При запуске программы в разделе отладки будут выведены все элементы исходной последовательности *B*, снабженные комментарием «W», а затем – все оканчивающиеся цифрой элементы этой же последовательности, снабженные комментарием «O», причем *описанный набор данных будет выведен столько раз, сколько имеется элементов исходной последовательности A, оканчивающихся цифрой*. Это показывает, что отбор элементов последовательности *B* и их сортировка выполняются при обработке *каждого* элемента последовательности *A*, используемого при построении перекрестного объединения.

Для того чтобы обеспечить *однократное* выполнение действий по отбору и сортировке элементов исходных последовательностей, достаточно добавить в конец цепочки методов LINQ метод `ToArgu`, относящийся к категории *методов экспортирования* и преобразующий полученную последовательность в массив (к этой категории относятся также методы `ToList` и `ToDictionary`, преобразующие последовательность в список `List<T>` и ассоциативный массив `Dictionary<TKey,TValue>` соответственно). Указанное действие выполняется *немедленно*; при этом в переменные `a` и `b` будут записаны не последовательности с «отложенными» командами отбора и сортировки, а обычные массивы, которые затем будут использоваться при формировании перекрестного объединения. В приведенном ни-

же варианте решения добавленные методы ToArray выделены полужирным шрифтом:

```
public static void Solve()
{
    Task("LinqBegin52");
    var a = GetEnumerableString()
        .Where(e => char.IsDigit(e[e.Length - 1]))
        .OrderBy(e => e).Show("A").ToArray();
    var b = GetEnumerableString()
        .Where(e => char.IsDigit(e[e.Length - 1]))
        .OrderByDescending(e => e).Show("B").ToArray();
    a.SelectMany(e1 => b.Select(e2 => e1 + "=" + e2)).Put();
}
```

Если модифицировать данную программу тем же способом, которым был модифицирован предыдущий вариант программы (добавив в лямбда-выражения методов Where и OrderByDescending для последовательности **b** вызовы Show), то элементы последовательности **b**, обрабатываемые методами Where и OrderByDescending, будут выведены на экран лишь по одному разу.

Примечание. Отложенный характер выполнения запросов LINQ является их важной особенностью, полезной во многих ситуациях и, как правило, не оказывающей существенного влияния на эффективность. Рассмотренная выше ситуация, связанная с построением перекрестного объединения, является одной из немногих ситуаций, в которых (при обработке последовательностей большого размера) может оказаться целесообразным прибегнуть к методам экспортирования.

5.5.4. Построение плоского левого внешнего объединения: **LinqBegin54**

Обычно для объединения последовательностей используются специальные методы Join и GroupJoin. Рассмотрим особенности их применения, решив задачу LinqBegin54. В этой задаче требуется построить наиболее сложный из описанных в п. 5.5.1 вариантов объединения: *плоское левое внешнее объединение*.

LinqBegin54. Даны строковые последовательности *A* и *B*; все строки в каждой последовательности различны, имеют ненулевую длину и содержат только цифры и заглавные буквы латинского алфавита. Найти последовательность всех пар

строк, удовлетворяющих следующим условиям: первый элемент пары принадлежит последовательности A , а второй либо является одним из элементов последовательности B , начинающихся с того же символа, что и первый элемент пары, либо является пустой строкой (если B не содержит ни одной подходящей строки). Результирующая последовательность называется *левым внешним объединением* последовательностей A и B по *ключу*, определяемому первыми символами исходных строк. Представить найденное объединение в виде последовательности строк вида « $E_A E_B$ », где E_A – элемент из A , а E_B – либо один из соответствующих ему элементов из B , либо пустая строка. Расположить элементы полученной строковой последовательности в лексикографическом порядке по возрастанию.

Указание. Использовать методы GroupJoin, DefaultIfEmpty, Select и SelectMany.

На рис. 22 приводится вид окна задачника при запуске программы-заготовки, созданной для данного задания.

Programming Taskbook - Электронный задачник по программированию [C#]

ОБЪЕДИНЕНИЕ И ГРУППИРОВКА
Задание: LinqBegin54

Выполняет: Иванов Петр

Результаты (F2) Цвет (F3) Режим (F4)

Ознакомительный запуск
не выполнена ни одна из операций ввода-вывода.

Выход (Esc)

Даны строковые последовательности A и B ; все строки в каждой последовательности различны, имеют ненулевую длину и содержат только цифры и заглавные буквы латинского алфавита. Найти последовательность всех пар строк, удовлетворяющих следующим условиям: первый элемент пары принадлежит последовательности A , а второй либо является одним из элементов последовательности B , начинающихся с того же символа, что и первый элемент пары, либо является пустой строкой (если B не содержит ни одной подходящей строки). Результирующая последовательность называется **левым внешним объединением** последовательностей A и B по **ключу**, определяемому первыми символами исходных строк. Представить найденное объединение в виде последовательности строк вида « $E_A E_B$ », где E_A – элемент из A , а E_B – либо один из соответствующих ему элементов из B , либо пустая строка. Расположить элементы полученной строковой последовательности в лексикографическом порядке по возрастанию.

Указание. Использовать методы GroupJoin, DefaultIfEmpty, Select и SelectMany.

| Исходные данные | | | | | | | |
|------------------------|---------|---------|-------|-----------------------|-------|-----------|-----------|
| Последовательность A: | | | | Последовательность B: | | | |
| 10: "7" | "7BI" | "FC" | "I" | "EA" | "AC" | "E4" | "G8I" "J" |
| | "I1J" | | | | | | |
| 11: "J" | "9" | "67" | "I" | "980" | "J64" | "7" | "C" "5EJ" |
| | "64H" | "GI5" | | | | | |
| Пример верного решения | | | | | | | |
| 11: "7.7" | "7BI.7" | "AC." | "E4." | "EA." | "FC." | "G8I.GI5" | |
| | "I.I" | "I1J.I" | "J.J" | "J.J64" | | | |

Рис. 22. Ознакомительный запуск задания LinqBegin54

Важной особенностью полученной последовательности является наличие элементов, оканчивающихся точкой (например, АС. и Е4.). Эти элементы содержат строки из последовательности *A*, для которых нет соответствий в последовательности *B* (в данном случае нет строк, начинающихся с того же символа, что и строка из *A*). В полученную последовательность могут входить элементы с одинаковыми строками из *A* (например, J.J и J.J64), если строке из *A* соответствует *несколько* строк из *B*.

При выполнении данного задания нельзя использовать метод Join, поскольку он включает в результирующую последовательность только те элементы из первой исходной последовательности, для которых найдены соответствия во второй последовательности (напомним, что подобное объединение называется *внутренним*). Если требуется, чтобы в результирующую последовательность включались *все* элементы первой последовательности, то необходимо использовать метод GroupJoin, в котором каждому элементу первой последовательности сопоставляется набор всех соответствующих ему элементов второй последовательности, причем этот набор может быть пустым.

Результат подобного сопоставления можно представить в виде пары (e1, ee2), где e1 обозначает некоторый элемент первой последовательности, а ee2 – *набор* соответствующих ему элементов второй последовательности. Имея пару (e1, ee2), мы можем, используя метод Select для последовательности ee2, сформировать набор строковых пар того вида, который требуется в задании:

```
ee2.Select(e => e1 + "." + e)
```

Приведенное выражение надо указать в качестве возвращаемого значения последнего лямбда-выражения метода GroupJoin (это лямбда-выражение определяет вид элементов полученного объединения):

```
a.GroupJoin(b, e => e[0], e => e[0],  
            (e1, ee2) => ee2.Select(e => e1 + "." + e))
```

Метод GroupJoin вызывается для первой из исходных последовательностей (обозначенной через *a*), его первый параметр определяет вторую последовательность (*b*), а два последующих параметра являются лямбда-выражениями, задающими ключ, по которым определяется соответствие между элементами исходных последователь-

ностей (в первом из этих лямбда-выражений вычисляется ключ для первой последовательности, во втором – для второй последовательности).

В результате будет получена последовательность, элементами которой будут также последовательности, то есть иерархическая последовательность строк. Для того чтобы преобразовать ее в «плоскую» последовательность строковых элементов, достаточно выполнить ее проецирование с применением метода `SelectMany` (см. п. 5.4):

```
a.GroupJoin(b, e => e[0], e => e[0],  
    (e1, ee2) => ee2.Select(e => e1 + "." + e))  
    .SelectMany(e => e)
```

В соответствии с условием задачи полученную последовательность следует отсортировать в лексикографическом порядке по возрастанию; для этого достаточно применить к ней метод `OrderBy` (см. п. 5.3):


```
a.GroupJoin(b, e => e[0], e => e[0],  
    (e1, ee2) => ee2.Select(e => e1 + "." + e))  
    .SelectMany(e => e).OrderBy(e => e)
```

Осталось организовать ввод исходных последовательностей и вывод результата. Функции `GetEnumerableString`, обеспечивающие ввод, достаточно указать вместо идентификаторов `a` и `b`; метод вывода `Put` надо указать в конце цепочки методов LINQ. Перед вызовом метода `Put` добавим вызов метода `Show`, обеспечивающий вывод полученной последовательности в раздел отладки. В результате получим первый вариант решения задачи:

```
public static void Solve()  
{  
    Task("LinqBegin54");  
    GetEnumerableString().GroupJoin(GetEnumerableString(),  
        e => e[0], e => e[0],  
        (e1, ee2) => ee2.Select(e => e1 + "." + e))  
        .SelectMany(e => e).OrderBy(e => e).Show().Put();  
}
```

При запуске полученной программы будет выведено сообщение об ошибке «Выведены не все результирующие данные» (см. рис. 23).

Поскольку в разделе результатов часть строк выведена не полностью, для анализа полученных данных удобнее обратиться к разделу


Programming Task - Электронный задачник по программированию [C#]

Результаты (F2) Цвет (F3) Режим (F4)

ОБЪЕДИНЕНИЕ И ГРУППИРОВКА
 Задача: LinkBegin4

Выполняет: Иванков Петр

Вывод [Esc]

Выведены не все результирующие данные.
 Количество выведенных данных: 8 (из 14)

Введено данных: 24 из 24 Выведено данных: 8 из 14 Пройдено тестов: 0 из 7

Исходные данные

| | | | | | | | | |
|-----------------------|-------|-------|-------|-----|-------|------|-------|------|
| 12: "В" | "I" | "G" | "DJC" | "2" | "ICF" | "E7" | "CAB" | "HE" |
| "8" | "AD2" | "FI2" | | | | | | |
| Последовательность B: | | | | | | | | |
| 10: "EA" | "8" | "74I" | "J8B" | "C" | "HGH" | "G" | "AA7" | "9" |
| "GF5" | | | | | | | | |

Полученные результаты

| | | | | | | | |
|----------|--------|---------|-------|-------|---------|----------|--|
| 7: "8.*" | "AD2*" | "CAB.C" | "E7*" | "G.G" | "G.GF*" | "HE.HG*" | |
|----------|--------|---------|-------|-------|---------|----------|--|

Пример верного решения

| | | | | | | | |
|----------|-------|-----------|----------|---------|--------|---------|--|
| 13: "2." | "8.8" | "AD2.AA7" | "B." | "CAB.C" | "DJC." | "E7.EA" | |
| "FI2." | "G.G" | "G.GF5" | "HE.HGH" | "I." | "ICF." | | |

1> 7: 8.8 AD2.AA7 CAB.C E7.EA G.G G.GF5 HE.HGH

Рис. 23. Ошибочное выполнение задания LinqBegin54

отладки. Сравнивая данные в этом разделе и в разделе с примером верного решения, мы можем убедиться в том, что все элементы, содержащие строки из обеих исходных последовательностей, найдены верно и выведены в правильном порядке. Однако в полученной последовательности отсутствуют элементы, связанные с теми строками из последовательности A , для которых не найдено соответствий в последовательности B . Фактически вместо требуемого левого внешнего объединения мы получили *внутреннее* объединение исходных последовательностей.

Ошибка связана с неправильной обработкой пустых последовательностей элементов B , возвращаемых для тех элементов A , для которых в B не найдено соответствий. Если последовательность `ee2` в последнем лямбда-выражении метода `GroupJoin` является пустой, то для нее, конечно, можно вызвать метод `Select` (подчеркнем, что «пустота» последовательности `ee2` не означает, что `ee2` равна `null`; последовательность *существует*, просто в ней нет элементов). Однако метод `Select` для пустой последовательности также вернет пустую последовательность. Значит, метод `GroupJoin` вернет иерархическую последовательность, некоторые элементы которой будут пустыми последовательностями. При переводе иерархической последовательности в плоскую, который выполняется методом `SelectMany`, пустые последовательности пропускаются, не влияя на результат.

Итак, для исправления ошибки необходимо изменить обработку пар (e1, ee2) в случае, если ee2 является пустой последовательностью. В соответствии с условием задачи в этой ситуации необходимо возвращать последовательность из *единственного* элемента, включающего строку e1 и точку после нее.

Для подобных действий в наборе методов LINQ предусмотрен специальный метод DefaultIfEmpty с необязательным параметром. Если данный метод применяется к непустой последовательности, то он возвращает эту последовательность в неизменном виде; если же он применяется к пустой последовательности, то он возвращает последовательность, содержащую *единственный элемент* (тип элемента соответствует типу элементов исходной последовательности). При вызове метода DefaultIfEmpty без параметра элемент созданной последовательности будет иметь значение по умолчанию (для ссылочных типов это null, для числовых типов – нулевое число); если DefaultIfEmpty имеет параметр, то значение этого параметра присваивается элементу созданной последовательности.

В нашем случае достаточно перед вызовом метода Select для последовательности ee2 применить к ней метод DefaultIfEmpty с параметром – пустой строкой (в приведенном ниже исправленном варианте решения добавленный фрагмент выделен полужирным шрифтом):

```
public static void Solve()
{
    Task("LinqBegin54");
    GetEnumerableString().GroupJoin(GetEnumerableString(),
        e => e[0], e => e[0],
        (e1, ee2) => ee2.DefaultIfEmpty("")
        .Select(e => e1 + "." + e))
        .SelectMany(e => e).OrderBy(e => e).Show().Put();
}
```

После семи тестовых запусков программы в окне задачника будет выведено сообщение о том, что задание выполнено.

Приведем еще один вариант правильного решения, в котором вызов операций LINQ оформляется в виде выражения запроса (см. п. 5.5.3):

```
public static void Solve()
{
    Task("LinqBegin54");
    (from e1 in GetEnumerableString()
     join e2 in GetEnumerableString()
```

```
on e1[0] equals e2[0]
into ee2
from e in ee2.DefaultIfEmpty("")
orderby e1, e
select e1 + "." + e).Put();
}
```

В пояснении нуждается та часть выражения запроса, которая связана с операцией объединения. Данная часть начинается с конструкции `from`, в которой указываются внешняя (левая) последовательность и связанный с ней перечислитель (в нашем случае `e1`). Затем следует конструкция `join`, в которой указываются внутренняя (правая) последовательность и связанный с ней перечислитель (`e2`). Следующая конструкция `on` (продолжение конструкции `join`) определяет ключи для внешней и внутренней последовательностей; в ней используются перечислители, связанные с этими последовательностями. Конструкция `into` (также являющаяся продолжением конструкции `join`) вводит перечислитель, с которым связывается *последовательность* элементов внутренней последовательности, соответствующая элементу внешней последовательности. В последующих конструкциях выражения запроса можно использовать все введенные ранее перечислители. В нашем случае выполняется преобразование последовательности `ee2` методом `DefaultIfEmpty`, для преобразованной последовательности вводится перечислитель `e`, который (совместно с перечислителем `e1` внешней последовательности) используется для сортировки данных и формирования результирующего объединения.

Обратите внимание на то, что в первых двух конструкциях полученного выражения запроса допустимо вызывать методы `GetEnumerator`, поскольку многократного обращения к этим конструкциям в ходе выполнения запроса не происходит. Вместо сортировки по набору ключей `e1` и `e` можно было бы организовать сортировку следующего вида:

```
orderby e1 + "." + e
```

Имеется возможность избежать дублирования в запросе строкового выражения `e1 + "." + e`. Для этого в синтаксисе выражений запросов предусмотрена особая конструкция `let`, не связанная с каким-либо методом LINQ и предназначенная для определения вспомогательных переменных-перечислителей. Приведем вариант выражения запроса с использованием конструкции `let`:

```
(from e1 in GetEnumerableString()
 join e2 in GetEnumerableString()
 on e1[0] equals e2[0]
 into ee2
 from e in ee2.DefaultIfEmpty("")
 let r = e1 + "." + e
 orderby r
 select r).Put();
```

Метод `Join` переводится на язык выражений запросов аналогично методу `GroupJoin`. Единственное отличие состоит в отсутствии конструкции `into`, поскольку в методе `Join` элементы результирующего объединения формируются непосредственно по перечислителям исходных последовательностей. В качестве примера приведем выражение запроса, формирующее *внутреннее* объединение для исходных последовательностей (в данном случае вспомогательный перечислитель `r` определяется через перечислители `e1` и `e2`):

```
(from e1 in GetEnumerableString()
 join e2 in GetEnumerableString()
 on e1[0] equals e2[0]
 let r = e1 + "." + e2
 orderby r
 select r).Put();
```

Завершая обсуждение методов `Join` и `GroupJoin`, отметим, что для них используется эффективная реализация, не сводящаяся к двойному циклу с попарными проверками ключей: при выполнении этих методов внутренняя последовательность предварительно *индексируется по требуемому ключу*, что позволяет избежать многократного перебора элементов внутренней последовательности при нахождении требуемых пар.

5.6. Группировка: LinqBegin60

В данном пункте мы обсудим средства LINQ, связанные с *группировкой* данных. Этим средствам посвящены задания LinqBegin56–LinqBegin60, завершающие группу LinqBegin (см. п. 2.4).

В результате группировки последовательности по некоторому ключу формируется новая последовательность, каждый элемент которой соответствует определенному значению ключа и содержит информацию обо всех элементах исходной последовательности, имеющих этот ключ. В качестве ключа может использоваться

любая характеристика элементов исходной последовательности. На элементы последовательности, являющейся результатом группировки, также не налагается никаких специальных ограничений; единственное условие состоит в том, что каждому значению ключа, обнаруженному в исходной последовательности, будет соответствовать *единственный* элемент результирующей последовательности.

Для группировки последовательностей предусмотрен метод `GroupBy`, имеющий четыре перегруженных варианта. Первый параметр любого варианта метода представляет собой лямбда-выражение, определяющее ключ группировки. Если этот параметр является единственным, то результатом выполнения метода `GroupBy` выступает последовательность с элементами интерфейсного типа `IGrouping<K, E>`, где `K` является типом ключа, а `E` – типом элементов исходной последовательности. Элемент типа `IGrouping<K, E>` является последовательностью элементов типа `E` и, кроме того, обеспечивает доступ (на чтение) к дополнительному свойству `Key` типа `K`. В последовательности, полученной в результате выполнения метода `GroupBy` с одним параметром, каждый элемент содержит набор элементов исходной последовательности с одинаковым ключом, а свойство `Key` элемента позволяет получить значение этого ключа.

Группировка может сопровождаться проецированием; в результате полученная последовательность будет содержать элементы типа `IGrouping<K, R>`, где `R` – тип элементов, полученных из элементов исходной последовательности путем некоторого преобразования. Требуемое преобразование задается в качестве второго параметра метода `GroupBy` и представляет собой лямбда-выражение, принимающее элемент исходной последовательности и возвращающее преобразованный элемент (типа `R`).

Имеется вариант метода `GroupBy`, позволяющий явным образом определить элементы сгруппированной последовательности. Для этого в качестве второго параметра метода используется лямбда-выражение, принимающее два параметра: значение ключа группировки и последовательность элементов исходной последовательности с этим ключом. Возвращаемое значение данного лямбда-выражения определяет элемент сгруппированной последовательности, соответствующий указанному ключу.

Четвертый вариант метода `GroupBy` является комбинацией второго и третьего вариантов. Этот вариант метода, помимо первого параметра, определяющего ключ группировки, содержит еще два параметра. Второй параметр представляет собой лямбда-выраже-

ние, принимающее элемент исходной последовательности и возвращающее преобразованный элемент. Третий параметр является лямбда-выражением, определяющим элементы сгруппированной последовательности; параметрами этого лямбда-выражения являются значение ключа группировки и последовательность *преобразованных* элементов исходной последовательности с этим ключом. Таким образом, основное отличие четвертого варианта от третьего состоит в том, что при определении элементов сгруппированной последовательности используются предварительно преобразованные элементы исходной последовательности.

Особенности применения различных вариантов метода GroupBy мы проиллюстрируем, выполняя задание LinqBegin60 – последнее задание данной группы.

LinqBegin60. 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 C . Полученную последовательность упорядочить по убыванию числовых значений сумм, а при равных значениях сумм – по возрастанию кодов символов C .

На рис. 24 приводится вид окна задачника при запуске программы-заготовки для данного задания.

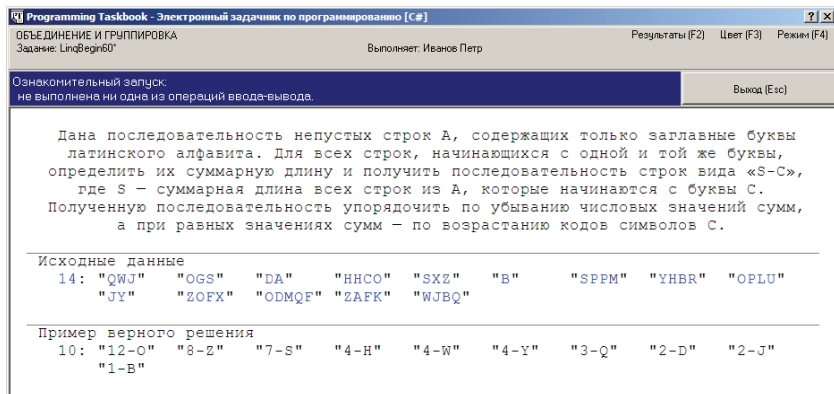


Рис. 24. Ознакомительный запуск задания LinqBegin60

Следует обратить внимание на сложный способ сортировки полученной последовательности: во-первых, главный ключ является числовым, и поэтому порядок элементов будет отличаться от лексикографического (в приведенном примере первая строка начинается с символа «1», вторая – с символа «8», третья – с символа «7»); во-вторых, для одинаковых значений главного ключа необходимо обеспечить сортировку символьной части строки в лексикографическом порядке по возрастанию.

Вначале используем вариант метода `GroupBy` с единственным параметром:

```
GetEnumerableString().GroupBy(k => k[0])
```

При определении ключа мы учли, что по условию задачи все строки в исходной последовательности являются непустыми.

Метод вернет последовательность с элементами типа `IGrouping<char, string>`. Каждый элемент последовательности будет содержать набор строк исходной последовательности, начинающихся с одного и того же символа, причем данный символ будет содержаться в свойстве `Key` этого элемента.

Прежде чем формировать на основе полученной последовательности требуемую последовательность строк, выполним сортировку:

```
GetEnumerableString().GroupBy(e => e[0])  
    .OrderByDescending(e => e.Sum(s => s.Length)).ThenBy(e => e.Key)
```

Для определения главного ключа сортировки мы воспользовались методом агрегирования `Sum` (см. п. 5.2); суммируемые числа (длины строк) определяются с помощью лямбда-выражения. В качестве подчиненного ключа сортировки используется ключ группировки.

Осталось выполнить проецирование полученной последовательности и вывести результат. Получаем первый вариант правильного решения:

```
GetEnumerableString().GroupBy(e => e[0])  
    .OrderByDescending(e => e.Sum(s => s.Length)).ThenBy(e => e.Key)  
    .Select(e => e.Sum(s => s.Length) + "-" + e.Key).Put();
```

Если воспользоваться вторым вариантом метода `GroupBy` и в процессе группировки перейти от исходных строк к их длинам, то можно упростить вызовы метода `Sum`, убрав из них лямбда-выражения:

```
GetEnumerableString().GroupBy(e => e[0], e => e.Length)
    .OrderByDescending(e => e.Sum()).ThenBy(e => e.Key)
    .Select(e => e.Sum() + "-" + e.Key).Put();
```

Заметим, что в данном случае элементы последовательности, возвращаемые методом `GroupBy`, имеют тип `IGrouping<char, int>`.

Используя третий вариант метода `GroupBy`, можно сформировать требуемую последовательность строк непосредственно при выполнении данного метода (указав вид ее элементов во втором лямбда-выражении). При этом не потребуется выполнять дополнительное проецирование (методом `Select`), однако усложнится определение ключей сортировки:

```
GetEnumerableString()
    .GroupBy(e => e[0], (k, ee) => ee.Sum(s => s.Length) + "-" + k)
    .OrderByDescending(e => int.Parse(e.Split('-')[0]))
    .ThenBy(e => e.Split('-')[1]).Put();
```

Напомним смысл параметров (`k`, `ee`) второго лямбда-выражения в методе `GroupBy`: `k` – значение ключа группировки, `ee` – последовательность элементов исходной последовательности, имеющих данный ключ.

В этом варианте решения для определения главного ключа сортировки вначале выполняется расщепление элемента последовательности на две части (по символу-разделителю «-»), после чего первый элемент полученного массива (с индексом 0) преобразуется в целое число методом `Parse` структуры `int`.

Если воспользоваться четвертым вариантом метода `GroupBy`, то можно упростить лямбда-выражение, определяющее элементы сгруппированной последовательности за счет предварительного преобразования элементов исходной последовательности. Действия по сортировке полученной последовательности не изменятся:

```
GetEnumerableString()
    .GroupBy(e => e[0], e => e.Length,
        (k, ee) => ee.Sum() + "-" + k)
    .OrderByDescending(e => int.Parse(e.Split('-')[0]))
    .ThenBy(e => e.Split('-')[1]).Put();
```

Мы видим, что формирование требуемых строк непосредственно в методе `GroupBy` приводит к усложнению действий по сортировке полученной последовательности. Это обусловлено преждевре-

менным объединением суммарной длины и ключа в строку. Чтобы упростить сортировку данных, можно представить элементы сгруппированной последовательности в виде пары (*s*, *k*) – «сумма, ключ», определив соответствующий *анонимный тип* непосредственно в лямбда-выражении:

```
GetEnumerableString().GroupBy(e => e[0], e => e.Length,  
    (k, ee) => new { k, s = ee.Sum() })  
    .OrderByDescending(e => e.s).ThenBy(e => e.k)  
    .Select(e => e.s + "-" + e.k).Put();
```

Анонимные типы активно используются при обработки последовательностей средствами LINQ. Определение анонимного типа начинается со слова `new`, после которого в фигурных скобках перечисляются имена свойств этого типа и их инициализирующие выражения. Если инициализирующее выражение представляет собой имя переменной, то имя свойства можно не указывать; в этом случае оно будет совпадать с именем инициализирующей переменной (в нашем случае вместо равенства `k = k` можно просто указать переменную `k`). Все свойства анонимного типа доступны *только для чтения*. Более подробно анонимные типы описываются в п. 8.2.

В последнем варианте решения, как и в первых двух вариантах, приходится дополнительно вызывать метод `Select`, в котором требуемая строка «собирается» из свойств анонимного типа. При этом введение анонимного типа позволяет представить лямбда-выражения для методов `OrderByDescending`, `ThenBy` и `Select` в более простом виде, по сравнению с любыми ранее рассмотренными вариантами.

Для операции группировки наряду с методом `GroupBy` предусмотрена также конструкция `group` выражения запроса. С помощью этой конструкции можно получать только последовательности с элементами типа `IGrouping<K, T>`, однако при этом можно выполнять дополнительное проецирование, как во втором варианте метода `GroupBy`. Конструкция `group` имеет вид

```
group e by k
```

В качестве *e* указывается либо переменная-перечислитель из предшествующей конструкции `from`, либо выражение, содержащее эту переменную (вариант с выражением соответствует второму варианту метода `GroupBy`, в котором применяется дополнительное проецирование). На месте идентификатора *k* указывается выражение, опреде-

ляющее ключ группировки (в этом выражении обычно используется переменная-перечислитель из той же конструкции `from`).

Отметим, что конструкция `group` является одной из двух конструкций, которой может оканчиваться выражение запроса (второй подобной конструкцией является конструкция `select`). Указанные конструкции могут также входить в выражение запроса в виде *вложенного запроса (подзапроса)*, если в дальнейшем предполагается использовать результаты, возвращенные этими конструкциями.

Проиллюстрируем использование конструкции `group`, приведя еще один вариант решения задачи LinqBegin60:

```
(from r in
    from e in GetEnumerableString()
    group e.Length by e[0]
let s = r.Sum()
let k = r.Key
orderby s descending, k
select s + "-" + k).Put();
```

Данный вариант содержит выражение запроса, включающее подзапрос, в котором выполняется группировка. В этом подзапросе выполняется дополнительное проецирование; таким образом, в полученной последовательности будут содержаться наборы *длин* исходных строк, сгруппированные по ключу – первому символу этих строк. Используя конструкцию `let`, мы вводим в запрос вспомогательные переменные `s` и `k`, соответствующие сумме длин и ключу, после чего применяем эти переменные в конструкции `orderby` для сортировки полученной последовательности и в конструкции `select` для формирования результирующей строковой последовательности.

В приведенном выражении запроса начальные действия (связанные с группировкой) пришлось оформить в виде подзапроса, расположенного между конструкциями внешнего запроса, в котором реализуются завершающие действия (сортировка и итоговое проецирование). Таким образом, порядок *записи* конструкций запроса не вполне соответствует порядку их *выполнения*, что несколько ухудшает восприятие запроса в целом (а также затрудняет конструирование запроса, которое приходится начинать «с середины»). Отмеченных недочетов в оформлении выражения запроса можно избежать, используя еще одну конструкцию – конструкцию *продолжения запроса* `into` (не следует смешивать эту конструкцию с элементом `into` конструкции `join`, рассмотренной в п. 5.5.4).

Конструкция продолжения запроса может указываться после конструкции `select` или `group`, завершающей предыдущий запрос. В ней указывается перечислитель для только что построенной последовательности, который можно в дальнейшем использовать при конструировании нового запроса (подобно перечислителям, вводимым с помощью конструкции `from`). Таким образом, используя конструкцию `into`, можно строить *цепочки* запросов, в которых не требуется прибегать к подзапросам.

С использованием конструкции `into` последний вариант решения можно представить в следующем виде (заметим, что при автоматическом форматировании выражения запроса, выполняемом в редакторе среды Visual Studio, конструкция `into` вместе с последующей частью запроса снабжается дополнительным отступом):

```
(from e in GetEnumerableString()
 group e.Length by e[0]
  into r
  let s = r.Sum()
  let k = r.Key
  orderby s descending, k
 select s + "-" + k).Put();
```

Данный вариант решения задачи LinqBegin60 является наиболее наглядным.

При построении запросов, содержащих конструкцию продолжения запроса, следует учитывать, что эта конструкция завершает область видимости всех введенных ранее перечислителей предыдущего запроса (остается доступным только тот перечислитель, который в ней указан).



Глава 6. Примеры решения задач из группы LinqObj

Задания группы LinqObj предназначены для закрепления навыков применения различных методов интерфейса LINQ to Objects. В отличие от заданий группы LinqBegin, эти задания не ориентированы на изучение какой-либо отдельной категории запросов; при их выполнении требуется самостоятельно выбрать методы LINQ, обеспечивающие получение требуемого результата. Еще одним отличием от заданий группы LinqBegin является более сложный вид исходных последовательностей: их элементы представляют собой *записи*, состоящие из нескольких полей. Указанные особенности приближают задания группы LinqObj к реальным задачам, возникающим при обработке сложных структур данных.

6.1. Простое задание на обработку отдельной последовательности: LinqObj4

6.1.1. Создание проекта-заготовки и знакомство с заданием. Дополнительные средства окна задачника, связанные с просмотром файловых данных

Вначале рассмотрим сравнительно простую задачу, связанную с обработкой отдельной последовательности.

LinqObj4. Исходная последовательность содержит сведения о клиентах фитнес-центра. Каждый элемент последовательности включает следующие целочисленные поля:

<Год> <Номер месяца> <Продолжительность занятий (в часах)> <Код клиента>

Для каждого клиента, присутствующего в исходных данных, определить суммарную продолжительность занятий в течение всех лет (вначале выводить суммарную продолжительность, затем код клиента). Сведения о каждом клиенте выводить на новой строке и упорядочивать по убыванию суммарной продолжительности, а при их равенстве – по возрастанию кода клиента.

Проект-заготовка для данного задания, как и для любых заданий, выполняемых с использованием электронного задачника Programming Taskbook, должен создаваться с помощью программного модуля PT4Load (см. п. 5.1.1). После создания этого проекта, автоматического запуска среды Visual Studio и загрузки в нее созданного проекта на экране будет отображен файл LinqObj4.cs, в который требуется ввести решение задачи. Приведем начальную часть этого файла:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using PT4;

namespace PT4Tasks
{
    public class MyTask : PT
    {
        // Для чтения набора строк из исходного текстового файла
        // в массив a типа string[] используйте оператор:
        //
        //     a = File.ReadAllLines(GetString(), Encoding.Default);
        //
        // Для записи последовательности r типа IEnumerable<string>
        // в результирующий текстовый файл используйте оператор:
        //
        //     File.WriteAllLines(GetString(), r.ToArray(), Encoding.Default);
        //
        // При решении задач группы LinqObj доступны следующие
        // дополнительные методы расширения, определенные в задачнике:
        //
        //     Show() и Show(cmt) - отладочная печать последовательности,
```

```
//      cmt - строковый комментарий;  
//  
//      Show(e => r) и Show(cmt, e => r) - отладочная печать  
//      значений r, полученных из элементов e последовательности,  
//      cmt - строковый комментарий.  
  
public static void Solve()  
{  
    Task("LinqObj4");  
  
}  
}  
. . .
```

Завершающая часть файла, не приведенная выше, содержит вспомогательный класс LinqObj, в котором реализованы варианты метода расширения Show, предназначенного для отладочной печати последовательностей (данный метод был подробно описан в п. 5.3).

По сравнению с файлами, создаваемыми для решения задач группы LinqBegin (см. п. 5.1.1), файлы-заготовки для задач группы LinqObj имеют следующие особенности:

- ❑ в список директив using добавлена директива подключения пространства имен System.IO, в котором определены классы, связанные с файловым вводом-выводом;
- ❑ класс MyTask содержит комментарий, описывающий действия, которые надо выполнить для ввода исходных последовательностей и вывода результатов;
- ❑ в файле отсутствуют описания дополнительных методов GetEnumeratorInt, GetEnumeratorString и метода расширения Put.

При выполнении заданий из группы LinqObj нет необходимости в использовании методов ввода-вывода GetEnumeratorInt, GetEnumeratorString и Put, поскольку как исходные, так и результирующие последовательности содержатся в *текстовых файлах*, для работы с которыми достаточно использовать средства стандартной библиотеки .NET. Из числа дополнительных методов ввода-вывода, предоставляемых задачиком, в заданиях группы LinqObj требуется использовать только методы GetInt и GetString, предназначенные для ввода целых чисел и строк соответственно.

Запустив созданную программу-заготовку, мы увидим на экране окно задачника в режиме ознакомительного запуска (рис. 25).

При ознакомительном запуске программы окно задачника содержит три раздела: с формулировкой задания, исходными данными

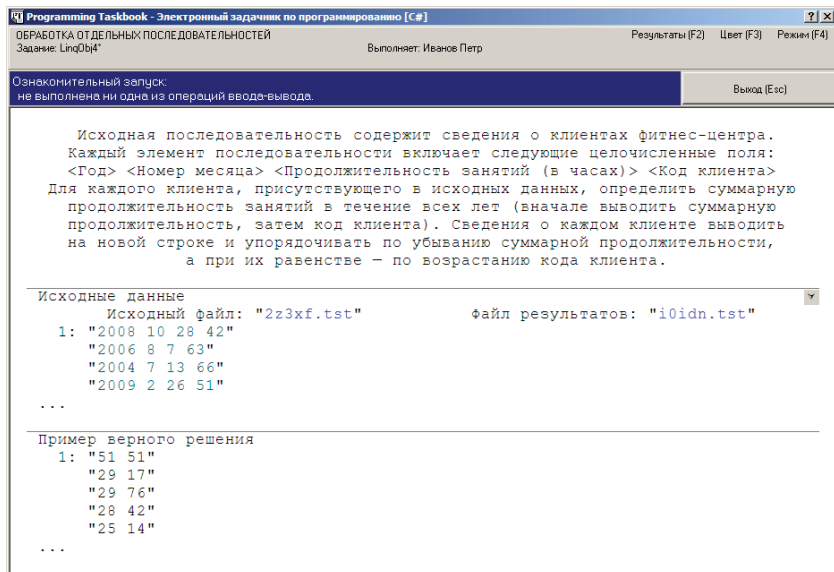


Рис. 25. Ознакомительный запуск задания LinqObj4



и примером верного решения. В начале раздела с исходными данными указаны две текстовые строки, снабженные комментариями «Исходный файл» и «Файл результатов». Первая строка определяет имя текстового файла, содержащего исходную последовательность. Этот файл автоматически создается задачиком при инициализации задания; получив его имя, программа сможет обратиться к нему и прочесть его данные. Вторая строка определяет имя текстового файла, в котором должна содержаться результирующая строковая последовательность. Программа должна заполнить указанный файл требуемыми данными, после чего задачник проанализирует содержимое этого файла, сравнив его с правильным вариантом решения. При каждом тестовом запуске программы имена файлов, как и их содержимое, изменяются.

В окне задачника отображаются не только имена, но и содержимое файлов, связанных с заданием. Каждая строка текстового файла заключается в кавычки и выводится на отдельной экранной строке; рядом с первой строкой файла указывается ее порядковый номер, равный 1. В разделах исходных данных и результатов содержимое файлов выделяется бирюзовым цветом (цветовое выделение позволяет отличить файловые строки от других данных, а также коммен-

тариев). В разделе с правильным решением все данные выводятся серым цветом, чтобы отличить их от «настоящих» данных, найденных программой.

Приведенный на рис. 25 вариант окна соответствует режиму *сокращенного* отображения файловых данных, в котором для каждого файла отображается лишь начальная часть его содержимого (от одной до пяти строк). Признаком того, что часть данных отсутствует, является *многоточие*, размещенное в нижней части тех разделов, в которых отображаются сокращенные данные. Этот режим удобен при первоначальном знакомстве с заданием, поскольку позволяет отобразить в окне сравнительно небольшого размера содержимое всех разделов. При более детальном анализе данных, а также при сравнении полученных ошибочных результатов с примером правильного решения следует использовать режим *полного* отображения содержимого текстовых файлов.

Завершающая часть данного пункта будет посвящена описанию различных возможностей, связанных с режимом полного отображения.

Для переключения между сокращенным и полным режимом отображения файловых данных достаточно нажать клавишу **Ins** или выполнить двойной щелчок мышью в одном из разделов с файловыми данными. Можно также щелкнуть на квадратном маркере, который появляется в правом верхнем углу раздела исходных данных, если окно содержит файловые данные. Изображение на этом маркере служит индикатором режима: вариант  со стрелкой, направленной вниз, обозначает режим сокращенного отображения (при наведении мыши на маркер в этом случае выводится подсказка «Развернуть содержимое текстовых файлов (Ins)»); вариант  со стрелкой, направленной вверх, обозначает режим полного отображения (с ним связана подсказка «Свернуть содержимое текстовых файлов (Ins)»).

На рис. 26 приведен вид окна в режиме полного отображения текстовых файлов. В этом режиме порядковый номер указывается перед каждой файловой строкой. При закрытии окна текущий режим отображения запоминается и при последующих запусках программы восстанавливается.

В случае, когда размеров окна недостаточно для отображения всех данных, окно снабжается полосой прокрутки, и, кроме того, на нем отображаются дополнительные маркеры (см. рис. 27). Прокрутку содержимого окна проще всего выполнять с помощью клавиш **Home**, **End**, **[↑]**, **[↓]**, **PgUp**, **PgDn**, а также используя колесико мыши.

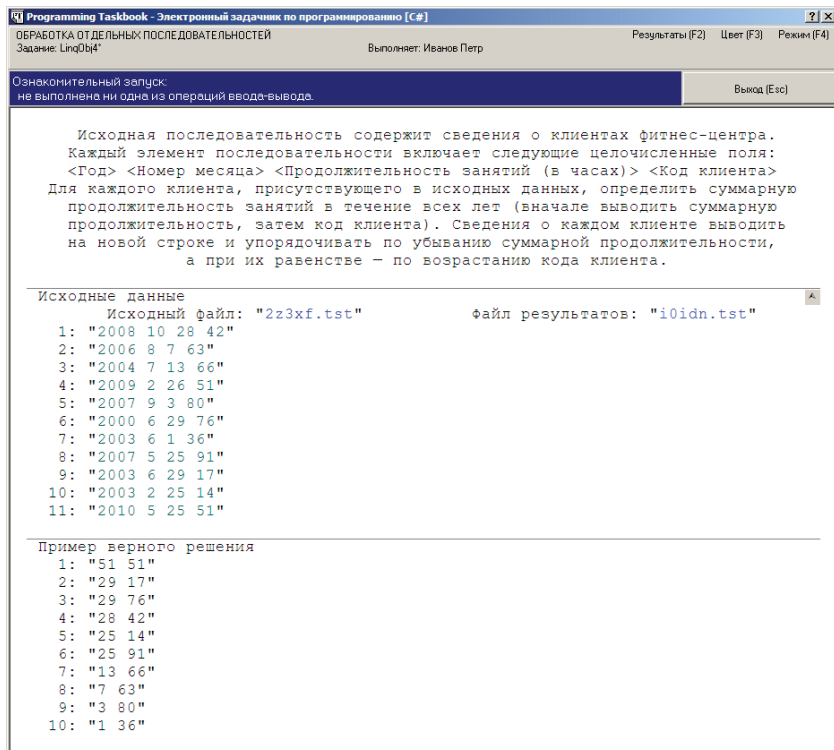


Рис. 26. Режим полного отображения текстовых файлов без полосы прокрутки

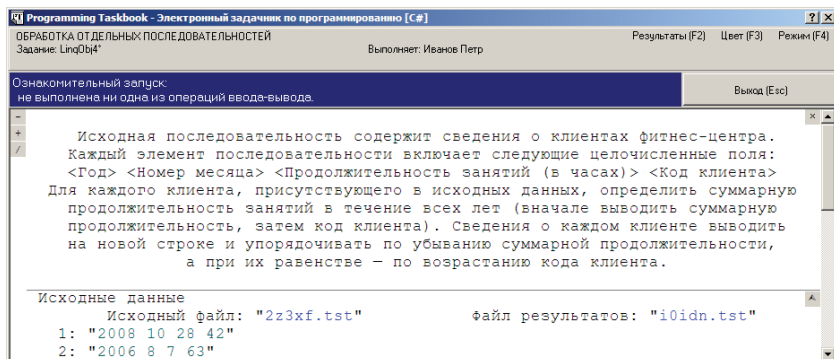





Рис. 27. Режим полного отображения текстовых файлов с полосой прокрутки

Группа маркеров, отображаемая в левом верхнем углу раздела с формулировкой задания, предназначена для быстрого отображения различных разделов, связанных с заданием: щелчок на маркере  обеспечивает переход к началу предыдущего раздела, щелчок на маркере  – к началу следующего раздела (рис. 28). Перебор разделов выполняется циклически. Маркер  позволяет быстро переключаться между разделами с результатами и примером правильного решения, если окно содержит оба этих раздела. Вместо щелчка на любом из указанных маркеров достаточно нажать клавишу, соответствующую изображенному на нем символу: **[-]**, **[+]**, **[/]**.

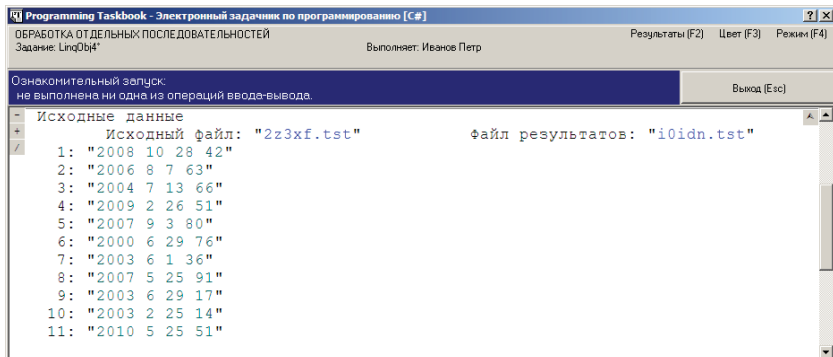




Рис. 28. Отображение начала раздела с исходными данными

Наконец, отметим маркер , появляющийся в правом верхнем углу раздела с формулировкой задания при отображении в окне полосы прокрутки. Этот маркер (и связанная с ним клавиша **Del**) позволяет скрыть раздел с формулировкой, увеличив тем самым область окна для отображения данных из других разделов. Заметим, что клавиша **Del** дает возможность скрыть раздел с формулировкой и в том случае, если маркер  не отображается на экране; эта возможность уже обсуждалась в п. 5.3.

6.1.2. Выполнение задания

Завершив обзор дополнительных возможностей окна задачника и ознакомившись с заданием LinqObj4, приступим к его выполнению.

В этом задании, как и во всех заданиях группы LinqObj, исходная последовательность содержится во внешнем текстовом файле.

Для чтения всех строк этого файла проще всего воспользоваться методом `ReadAllLines` класса `File`, который возвращает содержимое файла в виде строкового массива:

```
File.ReadAllLines(GetString(), Encoding.Default)
```

Первый параметр этого метода определяет имя файла; мы получаем это имя с помощью функции `GetString`. Второй, необязательный параметр определяет кодировку файла; по умолчанию используется кодировка UTF-8. Поскольку во всех заданиях электронного задачника, связанных с обработкой файлов, предполагается, что файлы имеют кодировку «windows-1251» (которая является кодировкой по умолчанию для русской версии Windows), нам необходимо явным образом указать эту кодировку во втором параметре, используя свойство `Default` класса `Encoding`.

Примечание. Считывание содержимого всего текстового файла в оперативную память, выполняемое методом `ReadAllLines`, как правило, оказывается менее эффективным, чем построчное считывание данных и их немедленная обработка, особенно для файлов большого размера. При выполнении заданий группы `LinqObj` метод `ReadAllLines` следует рассматривать как один из «вспомогательных» методов, используемых для изучения технологии LINQ. В таком качестве его применение вполне оправдано, поскольку он обеспечивает наиболее простой способ получения исходной последовательности.

В версии .NET 4.0 в класс `File` был добавлен метод `ReadLines` с тем же набором параметров, что и `ReadAllLines`, но возвращающий не массив `string[]`, а последовательность `IEnumerable<string>`. В случае использования метода `ReadLines` считывание элементов из файла выполняется не в момент выполнения данного метода (как при использовании метода `ReadAllLines`), а впоследствии, при обработке каждого элемента последовательности в цикле `foreach`, что позволяет получить более эффективный *построчный* вариант обработки файлов, по сравнению с методом `ReadAllLines`.

При последующей обработке исходной последовательности нам потребуется обращаться к отдельным полям ее элементов, поэтому после считывания строк из исходного файла необходимо преобразовать их в набор полей. Для этого достаточно применить к полученному строковому массиву метод `Select`, определяющий по исходному строковому элементу элемент *анонимного типа* с требуемыми полями (анонимные типы ранее использовались в п. 5.6):

```
File.ReadAllLines(GetString(), Encoding.Default)
    .Select(e =>
    {
        string[] s = e.Split(' ');
        return new
        {
            hours = int.Parse(s[2]),
            code = int.Parse(s[3])
        };
    })
```

Мы использовали лямбда-выражение, содержащее не возвращаемое значение, а *набор операторов*, включающий оператор `return`. Это связано с тем, что для определения полей надо предварительно выполнить расщепление исходной строки методом `Split`. Заметим, что если разбить текст лямбда-выражения на отдельные строки, то редактор кода автоматически выполнит форматирование полученного текста, представив результат в виде, подобном приведенному выше.

Возможен и вариант лямбда-выражения с возвращаемым значением, но в нем требуется дважды выполнить вызов метода `Split`:

```
.Select(e => new
{
    hours = int.Parse(e.Split[2]),
    code = int.Parse(e.Split[3])
})
```

Такой вариант вполне допустим для простых наборов данных, содержащих небольшое число полей.

Мы включили в анонимный тип лишь данные, связанные с продолжительностью занятий (поле `hours`) и кодом клиента (поле `code`), поскольку сведения о годе и месяце для выполнения задания LinqObj4 не требуются.

На данном этапе решения задачи можно выполнить отладочную печать полученной последовательности, добавив к цепочке вызванных методов вспомогательный метод `Show`. Приведем текст функции `Solve`, соответствующий текущему этапу решения:

```
public static void Solve()
{
    Task("LinqObj4");
    File.ReadAllLines(GetString(), Encoding.Default)
        .Select(e =>
        {
```

```

string[] s = e.Split(' ');
return new
{
    hours = int.Parse(s[2]),
    code = int.Parse(s[3])
};
})
.Show();
}

```

При запуске программы в окне задачника появится раздел отладки, содержащий информацию о полученной последовательности. Приведем вид этого окна в режиме сокращенного отображения файловых данных, дополнительно скрыв раздел с формулировкой (рис. 29). Информационная панель содержит сообщение «Введены не все требуемые исходные данные», поскольку в нашей программе еще не введено имя файла результатов.

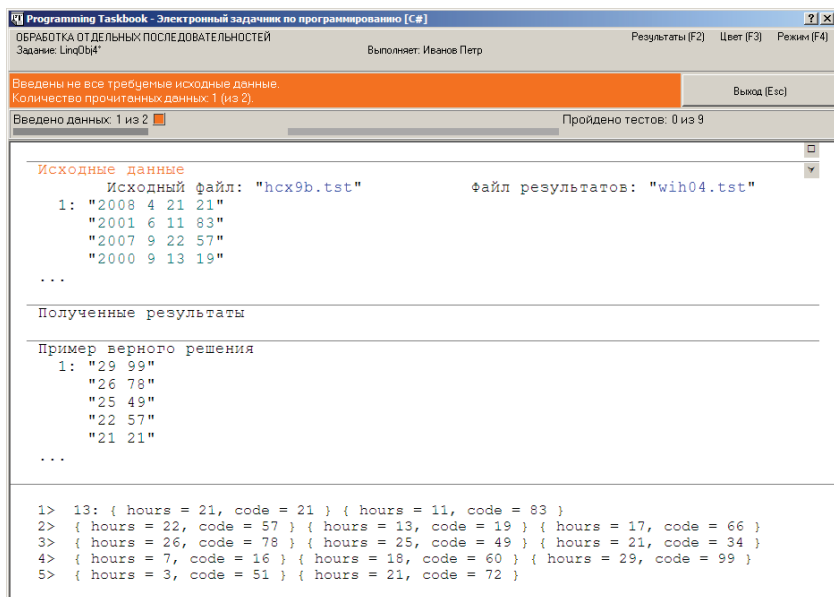


Рис. 29. Окно с выводом промежуточной последовательности

Раздел отладки демонстрирует, каким образом данные анонимного типа преобразуются к их строковому представлению: это представление содержит *список пар* имя поля = значение поля, заключен-

ный в фигурные скобки. Сравнивая исходные строковые данные и данные, выведенные в разделе отладки, убеждаемся, что преобразование выполнено правильно.

В задании требуется определить суммарную продолжительность занятий для каждого клиента; для этого следует выполнить группировку полученной последовательности по кодам клиентов (см. п. 5.6). Затем надо отсортировать сгруппированную последовательность (см. п. 5.3) по двум ключам: главному (суммарная продолжительность) – по убыванию и подчиненному (код клиента) – по возрастанию. Отсортированную последовательность надо преобразовать в последовательность строк с помощью метода проецирования `Select`.

Используя простейший вариант метода `GroupBy`, получаем следующую цепочку методов, которая должна продолжить ранее построенную цепочку:

```
.GroupBy(e => e.code)
.OrderByDescending(e => e.Sum(c => c.hours))
.ThenBy(e => e.Key)
.Select(e => e.Sum(c => c.hours) + " " + e.Key)
```

В приведенном варианте приходится дважды вычислять сумму полей `hours`: при сортировке и при последующем проецировании. Чтобы этого избежать, можно использовать вариант метода `GroupBy` с двумя параметрами, определив для элементов сгруппированной последовательности новый анонимный тип:

```
.GroupBy(e => e.code,
    (k, ee) => new {k, sum = ee.Sum(c => c.hours)})
.OrderByDescending(e => e.sum).ThenBy(e => e.k)
.Select(e => e.sum + " " + e.k)
```

В выражении, использованном в методе `Select`, не требуется выполнять специальные действия по преобразованию числовых данных в их строковые представления, так как, согласно правилам языка C#, подобное преобразование выполняется автоматически для одного из операндов суммы, если другой операнд имеет тип `string` (в нашем случае таким операндом является разделитель-пробел).

Осталось записать полученную строковую последовательность в текстовый файл с указанным именем. Если предварительно связать последовательность с переменной `r`, то для сохранения ее в текстовом файле достаточно выполнить единственный оператор

(напомним, что этот оператор указан в комментарии, включенном в заготовку к заданию – см. п. 6.1.1):

```
File.WriteAllLines(GetString(), r.ToArray(), Encoding.Default);
```

При выполнении оператора выполняются следующие действия:

- ❑ определяется имя файла результатов, которое считывается из набора исходных данных с помощью функции `GetString`;
- ❑ последовательность `r` преобразуется в массив; для этого используется метод `ToArray`, входящий в набор методов LINQ (данный метод упоминался ранее в конце п. 5.5.3);
- ❑ выполняется запись элементов полученного строкового массива в файл; при этом используется кодировка, указанная в третьем параметре (в данном случае – кодировка «windows-1251»).

Примечание. Начиная с версии .NET 4.0, можно пользоваться вариантом метода `WriteAllLines` со вторым параметром типа `IEnumerable<string>`. Таким образом, в этом варианте не требуется преобразовывать последовательность в массив, вызывая для нее метод `ToArray`.

Объединяя полученные фрагменты, получаем первый вариант правильного решения:

```
public static void Solve()
{
    Task("LinqObj4");
    var r = File.ReadAllLines(GetString(), Encoding.Default)
        .Select(e =>
        {
            string[] s = e.Split(' ');
            return new
            {
                hours = int.Parse(s[2]),
                code = int.Parse(s[3])
            };
        })
        .GroupBy(e => e.code,
            (k, ee) => new { k, sum = ee.Sum(c => c.hours) })
        .OrderByDescending(e => e.sum).ThenBy(e => e.k)
        .Select(e => e.sum + " " + e.k);
    File.WriteAllLines(GetString(), r.ToArray(), Encoding.Default);
}
```

Примечание. В некоторых задачах группы `LinqObj` в начало файла требуется записать отдельные строки (не входящие в последовательность). Чтобы преобразовать строку `s` в одноэлементный

массив, достаточно использовать выражение `new string[] { s }`; напомним, что для объединения последовательностей можно использовать метод `Concat`.

Для проверки правильности полученного решения (как и решений других задач группы LinqObj) его надо протестировать на девяти различных тестовых наборах исходных данных, среди которых будут встречаться и наборы сравнительно большого размера. В качестве примера приведем окно задачника с сообщением об успешном выполнении задания, в котором для обработки был предложен набор из 78 записей (рис. 30).

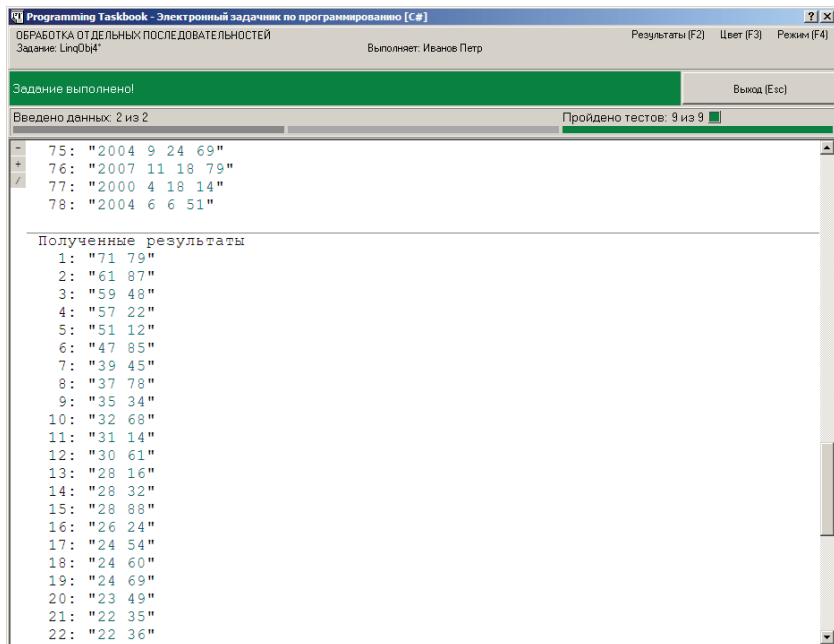


Рис. 30. Успешное выполнение задания LinqObj4

Завершая обсуждение задания LinqObj4, приведем вариант его решения, использующий *выражение запроса* (см. п. 5.5.3):

```
public static void Solve()
{
    Task("LinqObj4");
}
```



```
var r =
    from e in File.ReadAllLines(GetString(), Encoding.Default)
    let s = e.Split(' ')
    select new
    {
        hours = int.Parse(s[2]),
        code = int.Parse(s[3])
    }
    into e
    group e.hours by e.code
    into e
    let sum = e.Sum()
    orderby sum descending, e.Key
    select sum + " " + e.Key;
File.WriteAllLines(GetString(), r.ToArray(), Encoding.Default);
}
```

Прокомментируем особенности данного варианта.

Поскольку в конструкции `select` можно указывать только *выражение*, определяющее элемент возвращаемой последовательности, вызов метода `Split` мы выполнили в конструкции `let`, сохранив его результат во вспомогательной переменной `s` и используя его далее в конструкции `select`.

Результаты выполнения конструкций `select` и `group` (то есть промежуточные последовательности) мы передаем в следующие фрагменты выражения запроса, используя конструкцию *продолжения запроса* `into` (см. п. 5.6). Напомним, что при автоматическом форматировании выражения запроса, выполняемом редактором среды Visual Studio, каждый фрагмент, начинающийся с конструкции продолжения запроса, выделяется дополнительным отступом. Обратите внимание на то, что конструкция продолжения запроса *завершает область действия всех перечислителей предыдущего запроса*, поэтому в ней в качестве имени нового перечислителя можно использовать одно из ранее применявшихся имен.

Конструкция `group`, в отличие от метода `GroupBy`, не позволяет явно определять вид элементов получаемой последовательности (в результате группировки *всегда* возвращается последовательность элементов типа `IGrouping<K, E>` – см. п. 5.6), однако имеется возможность явно указать, какие поля из исходного набора следует оставить в сгруппированной последовательности. Указав после слова `group` поле `e.hours`, мы обеспечили построение последовательности элементов, включающих *ключ* (свойство `Key`) и *числовую последовательность* – набор чисел, взятых из полей `hours` и соответствующих

данному ключу. Это позволило при вызове метода `Sum` обойтись без уточняющего лямбда-выражения. Для того чтобы избежать двойного вызова метода `Sum`, мы сохранили его результат во вспомогательной переменной `sum`, которая была определена с помощью еще одной конструкции `let`.

Полученный вариант решения обладает несколько большей наглядностью, чем первый вариант, прежде всего за счет отсутствия лямбда-выражений.

6.2. Более сложные задания на обработку отдельных последовательностей: `LinqObj41`, `LinqObj61`

В данном пункте мы рассмотрим еще два задания группы `LinqObj`, связанные с обработкой отдельных последовательностей. Каждое из них имеет ряд особенностей, которые могут оказаться полезными и при выполнении других заданий этой группы.

В задании `LinqObj41`, связанном с обработкой последовательности данных об автозаправочных станциях (АЗС), в набор исходных данных, помимо самой последовательности, входит целое число – значение марки бензина *M*:

LinqObj41. Дано целое число *M* – значение одной из марок бензина. Исходная последовательность содержит сведения об автозаправочных станциях (АЗС). Каждый элемент последовательности включает следующие поля:

| | | | |
|-----------------|---------|------------|-----------------------------|
| <Марка бензина> | <Улица> | <Компания> | <Цена 1 литра (в копейках)> |
|-----------------|---------|------------|-----------------------------|

Названия компаний и улиц не содержат пробелов. В качестве марки бензина указываются числа 92, 95 или 98. Каждая компания имеет не более одной АЗС на каждой улице; цены на разных АЗС одной и той же компании могут различаться. Для каждой улицы, на которой имеются АЗС с бензином марки *M*, определить максимальную цену бензина этой марки (вначале выводить максимальную цену, затем название улицы). Сведения о каждой улице выводить на новой строке и упорядочивать по возрастанию максимальной цены, а для одинаковой цены – по названиям улиц в алфавитном порядке.

Если ни одной АЗС с бензином марки M не найдено, то записать в результирующий файл строку «Нет».

Приведем вид окна задачника, полученного при запуске созданной программы-заготовки (рис. 31). В окне установлен режим сокращенного отображения файловых данных.

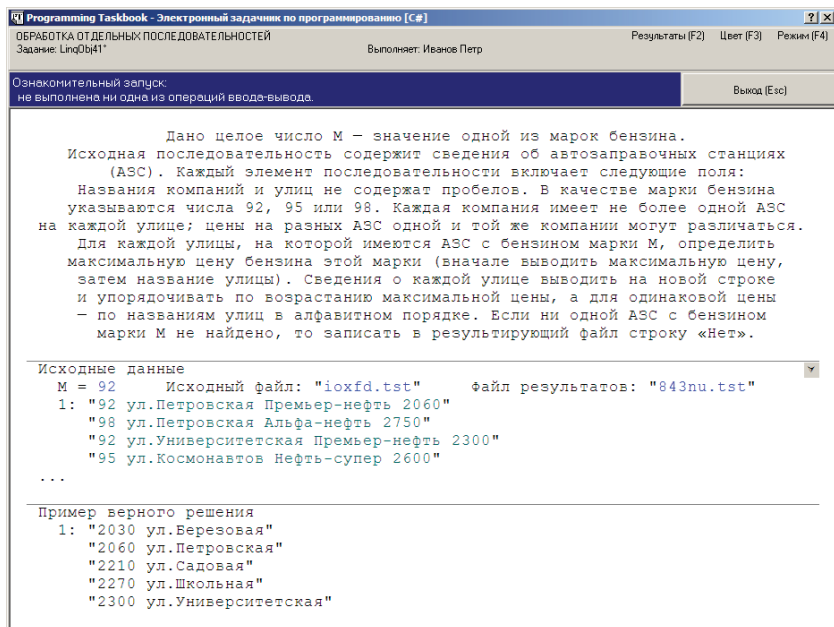


Рис. 31. Ознакомительный запуск задания LinqObj41

При организации ввода следует учитывать, что дополнительный элемент исходных данных (в нашем случае — марка бензина M) должен считываться *перед* именами файлов, связанных с исходной и результирующей последовательностями. Порядок ввода определяется как в формулировке задания, так и в разделе исходных данных, в котором число M указывается перед именами файлов. Напомним, что для ввода целочисленных данных следует использовать специальную функцию задачника GetInt.

Для отбора только тех данных, которые связаны с определенной маркой бензина, нам потребуется использовать метод фильтрации Where (см. п. 5.3).

Задание LinqObj41 является примером заданий, в которых необходимо предусматривать обработку особой ситуации, связанной с отсутствием требуемых элементов (в данном случае – отсутствием записей, связанных с маркой бензина *M*). В подобной ситуации результирующая последовательность будет пустой; для ее обработки удобно использовать метод `DefaultIfEmpty`, ранее рассмотренный в п. 5.5.4.

Организуя ввод и вывод с применением методов `ReadAllLines` и `WriteAllLines` и учитывая сделанные замечания, получаем первый вариант решения:

```
public static void Solve()
{
    Task("LinqObj41");
    int m = GetInt();
    var r = File.ReadAllLines(GetString(), Encoding.Default)
        .Select(e =>
        {
            string[] s = e.Split(' ');
            return new
            {
                brand = int.Parse(s[0]),
                street = s[1],
                price = int.Parse(s[3])
            };
        })
        .Where(e => e.brand == m)
        .GroupBy(e => e.street,
            (k, ee) => new { street = k, max = ee.Max(e => e.price) })
        .OrderBy(e => e.max).ThenBy(e => e.street)
        .Select(e => e.max + " " + e.street)
        .DefaultIfEmpty("Her");
    File.WriteAllLines(GetString(), r.ToArray(), Encoding.Default);
}
```

При расщеплении строк исходного файла на отдельные поля мы не стали сохранять данные, связанные с названиями компаний, поскольку для решения задачи эти данные не требуются.

В методе группировки `GroupBy` мы ввели дополнительный анонимный тип, что позволило избавиться от многократных вызовов агрегирующей функции `Max`.

Заметим, что вместо сортировки по двум ключам можно было бы отсортировать в лексикографическом порядке саму результирующую строковую последовательность, однако при подобной сорти-

ровке результат будет правильным только в случае, если все числовые значения цен имеют *одинаковое количество цифр* (в противном случае лексикографическая сортировка строк будет отличаться от сортировки числовых значений).

Вызов метода `DefaultIfEmpty` удобно выполнять в конце цепочки методов, применяя его к последовательности *строковых* элементов.

Приведем также вариант решения задачи, использующий выражение запроса:

```
public static void Solve()
{
    Task("LinqObj41");
    int m = GetInt();
    var r =
        (from e in File.ReadAllLines(GetString()), Encoding.Default)
        let s = e.Split(' ')
        select new
        {
            brand = int.Parse(s[0]),
            street = s[1],
            price = int.Parse(s[3])
        }
        into e
        where e.brand == m
        group e.price by e.street
        into e
        let max = e.Max()
        orderby max, e.Key
        select max + " " + e.Key)
    .DefaultIfEmpty("Her");
    File.WriteAllLines(GetString(), r.ToArray(), Encoding.Default);
}
```

Структура данного варианта решения аналогична структуре соответствующего варианта решения задачи `LinqObj4`, рассмотренного в предыдущем пункте. Следует лишь обратить внимание на использование конструкции `where` и на применение к выражению запроса метода `DefaultIfEmpty`, для которого в выражениях запросов не предусмотрено особой конструкции (для возможности применения метода к выражению запроса это выражение необходимо заключить в круглые скобки).

Во втором задании, которое мы рассмотрим в данном пункте, наряду с группировкой данных потребуется выполнить и их *внешнее объединение* (см. п. 5.5.4).

LinqObj61. Исходная последовательность содержит сведения об оценках учащихся по трем предметам: алгебре, геометрии и информатике. Каждый элемент последовательности содержит данные об одной оценке и включает следующие поля:

<Фамилия> <Инициалы> <Класс> <Название предмета> <Оценка>

Полных однофамильцев (с совпадающей фамилией и инициалами) среди учащихся нет. Класс задается целым числом, оценка – целое число в диапазоне 2–5. Название предмета указывается с заглавной буквы. Для каждого учащегося определить среднюю оценку по каждому предмету и вывести ее с двумя дробными знаками (если по какому-либо предмету учащийся не получил ни одной оценки, то вывести для этого предмета 0.00). Сведения о каждом учащемся выводить на отдельной строке, указывая фамилию, инициалы и средние оценки по алгебре, геометрии и информатике. Данные располагать в алфавитном порядке фамилий и инициалов.

На рис. 32 приведен вид окна задачника (в режиме сокращенного отображения файловых данных), которое будет отображено на экране при запуске созданной программы-заготовки.

Особенность данного задания состоит в необходимости вывода информации не только о парах «учащийся–предмет», которые встречаются в исходном наборе данных, но и о тех парах, которые в наборе отсутствуют (поскольку у некоторых учащихся могут отсутствовать оценки по некоторым предметам).

Если бы задача не содержала указанное дополнительное условие, то она решалась бы группировкой исходных данных по составному ключу – паре полей «учащийся–предмет». В нашем случае, с учетом дополнительного условия, потребуется выполнить два действия: (1) сгруппировать данные по учащимся и (2) в каждом элементе сгруппированной последовательности (который, в свою очередь, является последовательностью) построить его внешнее объединение со вспомогательной последовательностью subjects, содержащей названия предметов в требуемом порядке (последовательность subjects должна быть внешней, то есть левой, последовательностью). При этом необходимо предусмотреть особую обработку пустых последовательностей в полученном объединении. Для нахождения средних оценок следует использовать метод агрегирования Average.

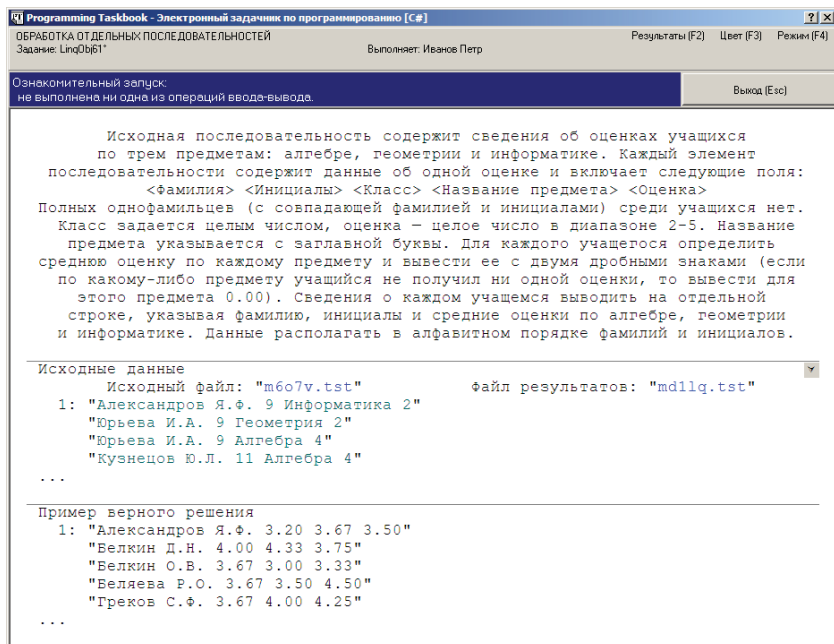


Рис. 32. Ознакомительный запуск задания LinqObj61

Отметим еще две особенности данной задачи. Имена учащихся включают фамилию и инициалы, поэтому при определении соответствующего поля надо объединить указанные поля. Далее, при выводе полученных средних оценок (вещественных чисел) необходимо выполнять их форматирование: числа должны выводиться с двумя дробными знаками и использовать точку в качестве десятичного разделителя.

Приведем вариант правильного решения задачи, а затем прокомментируем его:

```
public static void Solve()
{
    Task("LinqObj61");
    string[] subjects = { "Алгебра", "Геометрия", "Информатика" };
    var culture = new System.Globalization.CultureInfo("en-US");
    var r = File.ReadAllLines(GetString(), Encoding.Default)
        .Select(e =>
        {
            string[] s = e.Split(' ');
```

```
return new
{
    name = s[0] + " " + s[1],
    subj = s[3],
    mark = int.Parse(s[4])
};
})
.OrderBy(e => e.name)
.GroupBy(e => e.name, (k, ee) => new
{
    name = k,
    avrs = subjects
        .GroupJoin(ee, s => s, e => e.subj,
            (s1, ee1) => ee1.Select(e1 => e1.mark)
            .DefaultIfEmpty().Average())
})
.Select(e => e.name + e.avrs.Aggregate("",
    (a, d) => a + " " + d.ToString("f2", culture)));
File.WriteAllLines(GetString(), r.ToArray(), Encoding.Default);
}
```

Последовательность `subjects` определяется в виде обычного строкового массива с применением списка инициализирующих значений.

В программе описывается переменная `culture`, в которую заносятся региональные настройки «en-US» (*американского английского*). В этих настройках, в отличие от настроек «ru-RU», используемых по умолчанию в русской версии Windows, десятичным разделителем является точка. Соответствующий класс .NET `CultureInfo` описан в пространстве имен `System.Globalization`. Так как обращение к этому классу выполняется в единственном операторе, мы использовали в конструкторе полное имя класса, не добавляя директиву `using System.Globalization` в начало программы.

При формировании полей исходной последовательности мы занесли в поле `name` фамилию и инициалы, разделив их пробелом.

В результате группировки этой последовательности по полю `name` мы получаем последовательность из двух полей: `name` (определяется по ключу группировки) и `avrs`, которое является результатом левого внешнего объединения массива `subjects` и набора элементов, имеющих общее ключевое поле `name`. Объединение выполняется с помощью метода `GroupJoin`, ключом объединения является название предмета. Полученная последовательность `ee1` элементов с общим ключом объединения представляет собой набор элементов, соответствующих некоторому учащемуся и одному из трех анализируемых предметов. Поскольку нас интересуют только *оценки*,



входящие в эти элементы, выполняется проецирование данного набора в числовой набор оценок (методом `Select`). Полученный набор может быть пустым; с помощью метода `DefaultIfEmpty` пустой набор преобразуется в набор, содержащий единственный элемент, равный нулю. Наконец, к полученному набору применяется метод агрегирования `Average`. В результате в поле `avgs` помещается тройка вещественных чисел – средних оценок учащегося с именем `pame` по трем предметам (порядок предметов соответствует их порядку в массиве `subjects`). Если учащийся не имел оценок по некоторому предмету, то соответствующий элемент набора `avgs` равняется нулю.

Действия по сортировке полученной последовательности в комментариях не нуждаются.

При формировании строкового представления полученных данных необходимо добавить в строку все элементы поля `avgs`, отформатировав их требуемым образом. Чтобы не обращаться явным образом к каждому элементу, мы организовали их перебор с помощью «универсального» метода агрегирования `Aggregate` (см. п. 5.2), инициализируя связанный с ним аккумулятор пустой строкой и добавляя к нему каждую из трех средних оценок, снабженную начальным пробелом (первый из этих пробелов будет отделять полученный список средних оценок от имени учащегося `pame`).

Форматирование вещественного числа выполняется с помощью варианта метода `ToString` с двумя параметрами: формирующими атрибутами `f2`, определяющими, что число должно отображаться в формате с *фиксированным* разделителем и *двумя* дробными знаками, и региональными настройками `culture`, благодаря которым в качестве разделителя используется точка.

Заметим, что вместо выражения

```
a + " " + d.ToString("f2", culture)
```

можно использовать метод `Format` класса `string`:

```
string.Format(culture, "{0} {1:f2}", a, d)
```

В этом методе вначале указывается параметр, определяющий региональные настройки, затем *форматная строка*, а далее – список выражений, предназначенных для вставки в форматную строку на позициях, отмеченных *форматными настройками*. Форматные настройки заключаются в фигурные скобки и обязательно содержат индекс выражения из последующего списка, которое будет добавле-

но в указанную позицию (индексация начинается от 0). В форматных настройках могут дополнительно указываться ширина поля вывода (отделяется от индекса *запятой*) и форматирующие атрибуты (отделяются от предшествующих настроек *двоеточием*), например {1,6:f2} (ширина поля вывода здесь равна 6). Форматная строка может содержать и обычный текст, возвращаемый без изменений (в нашем примере это *пробел*, размещенный между форматными настройками). Заметим, что при форматировании сложных выражений использовать метод Format предпочтительнее, чем операцию сцепления «+».

Примечание. Изменить региональные настройки можно и для программы в целом. Если, например, требуется установить для программы настройки «en-US», то достаточно указать в начале программы единственный оператор:

```
System.Threading.Thread.CurrentThread.CurrentCulture =  
    new System.Globalization.CultureInfo("en-US");
```

Если подключить к программе пространства имен System.Threading и System.Globalization, то данный оператор примет более простой вид:

```
Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
```

Можно выполнить и более тонкую корректировку региональных настроек, изменив в них только вид десятичного разделителя. Приведем соответствующий фрагмент программы, предполагая, что к ней подключены указанные выше пространства имен:

```
var culture =  
    (CultureInfo)Thread.CurrentThread.CurrentCulture.Clone();  
culture.NumberFormat.NumberDecimalSeparator = ".";  
Thread.CurrentThread.CurrentCulture = culture;
```

После глобального изменения настроек отпадает необходимость в их указании в методах ToString и Format.

Все описанные действия можно реализовать и с применением выражения запроса. Приведем соответствующий вариант решения:

```
public static void Solve()  
{  
    Task("LinqObj61");  
    string[] subjects = { "Алгебра", "Геометрия", "Информатика" };  
    var culture = new System.Globalization.CultureInfo("en-US");
```

```

var r =
    from e in File.ReadAllLines(GetString(), Encoding.Default)
    let s = e.Split(' ')
    select new
    {
        name = s[0] + " " + s[1],
        subj = s[3],
        mark = int.Parse(s[4])
    }
    into e
    group e by e.name
    into e
    let avrs =
        from e1 in subjects
        join e2 in e
        on e1 equals e2.subj
        into ee
        select
            (from eel in ee
             select eel.mark)
            .DefaultIfEmpty().Average()
    orderby e.Key
    select e.Key + avrs.Aggregate("",
        (a, d) => a + " " + d.ToString("f2", culture));
File.WriteAllLines(GetString(), r.ToArray(), Encoding.Default);
}

```

В данном варианте решения мы использовали конструкцию `join`. Напомним, что выражение `into ee` является частью конструкции `join`, а не отдельной конструкцией продолжения запроса (как два ранее указанных выражения `into e`). Особенностью данного варианта является наличие двух *вложенных* подзапросов: первый из них возникает в определении переменной `avrs`, а второй – в выражении, указанном в операции `select`, входящей в определение `avrs` (к этому вложенному выражению запроса применяются методы `DefaultIfEmpty` и `Average`).

6.3. Обработка взаимосвязанных последовательностей: LinqObj98

Завершающая подгруппа группы LinqObj содержит 30 задач, посвященных обработке нескольких взаимосвязанных последовательностей.

Все задачи этой подгруппы относятся к одной предметной области (реализация товаров в сети магазинов); для того чтобы из-

бежать дублирования при описании исходных последовательностей, эти описания вынесены в преамбулу к данной подгруппе (см. п. 3.2).

Задачи могут включать две (LinqObj71–LinqObj80), три (LinqObj81–LinqObj92) или четыре (LinqObj93–LinqObj100) исходные последовательности. Мы рассмотрим задачу LinqObj98 – одну из задач повышенной сложности, в которой требуется обработать четыре последовательности.

LinqObj98. Даны последовательности B , C , D и E , включающие следующие поля:

| | | | |
|----|------------------------|-------------------|---------------------|
| B: | <Страна-производитель> | <Артикул товара> | <Категория> |
| C: | <Скидка (в процентах)> | <Код потребителя> | <Название магазина> |
| D: | <Артикул товара> | <Цена (в рублях)> | <Название магазина> |
| E: | <Код потребителя> | <Артикул товара> | <Название магазина> |

Свойства последовательностей описаны в преамбуле к данной подгруппе заданий. Для каждой категории товаров и каждого магазина, указанного в E , определить суммарный размер скидки на все товары данной категории, проданные в данном магазине (вначале выводится название категории, затем название магазина, потом суммарная скидка). При вычислении размера скидки на товар копейки отбрасываются. Если на проданный товар скидка отсутствует, то ее размер полагается равным 0. Если для некоторой категории товаров в каком-либо магазине не было продаж, то суммарная скидка для этой пары «категория–магазин» полагается равной -1 . Сведения о каждой паре «категория–магазин» выводить на новой строке и упорядочивать по названиям категорий в алфавитном порядке, а для одинаковых названий категорий – по названиям магазинов (также в алфавитном порядке).

Приведем вид окна задачника, полученного при запуске созданной программы-заготовки (рис. 33). В окне установлен режим сокращенного отображения файловых данных; для уменьшения размеров окна в нем скрыт раздел с формулировкой.

Исходные последовательности, созданные задачиком при данном запуске, имели следующий размер: 56 элементов (последовательность B), 42 элемента (последовательность C), 23 элемента (последовательность D) и 96 элементов (последовательность E). На рис. 34 приведен фрагмент раздела исходных данных в режиме полного отображения, содержащий заключительную часть последо-

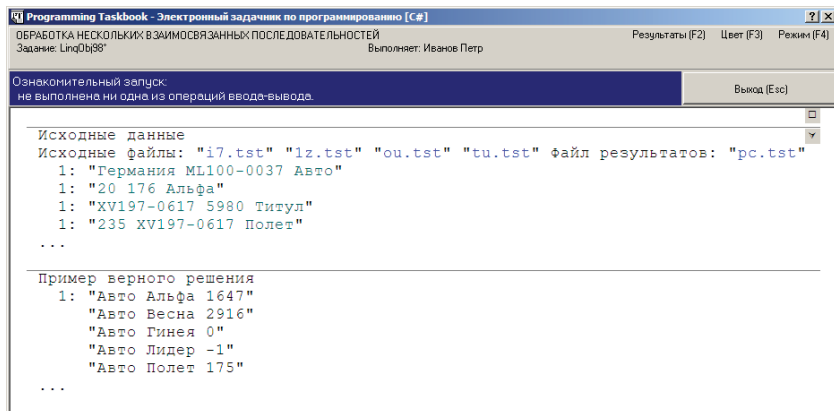


Рис. 33. Ознакомительный запуск задания LinqObj98
(сокращенное отображение файловых данных)

вательности C , всю последовательность D и начальную часть последовательности E .

Перед тем как приступить к выполнению задания, уточним «роль», которую играет в нем каждая последовательность. Последовательность B устанавливает связь между конкретным товаром (товар задается артикулом) и его категорией. Последовательность E содержит данные о товарах, проданных в различных магазинах; при этом указывается идентификатор потребителя, купившего товар. В результирующей последовательности следует отразить *все* категории, присутствующие в последовательности B , и *все* магазины, присутствующие в последовательности E (точнее, требуется вывести информацию о каждом элементе *перекрестного объединения* набора категорий товаров из B и набора магазинов из E). Последовательности C и D требуются для определения размера скидки для товара, проданного в данном магазине: с помощью последовательности D определяется цена товара в данном магазине, а с помощью последовательности C – величина скидки в процентах для потребителя, купившего этот товар в данном магазине (процент скидки не зависит от вида товара и определяется только парой «потребитель–магазин»; следует также учитывать, что потребитель может не иметь скидки в данном магазине).

Первый этап решения состоит в формировании последовательностей анонимных типов на основе исходных строковых последо-

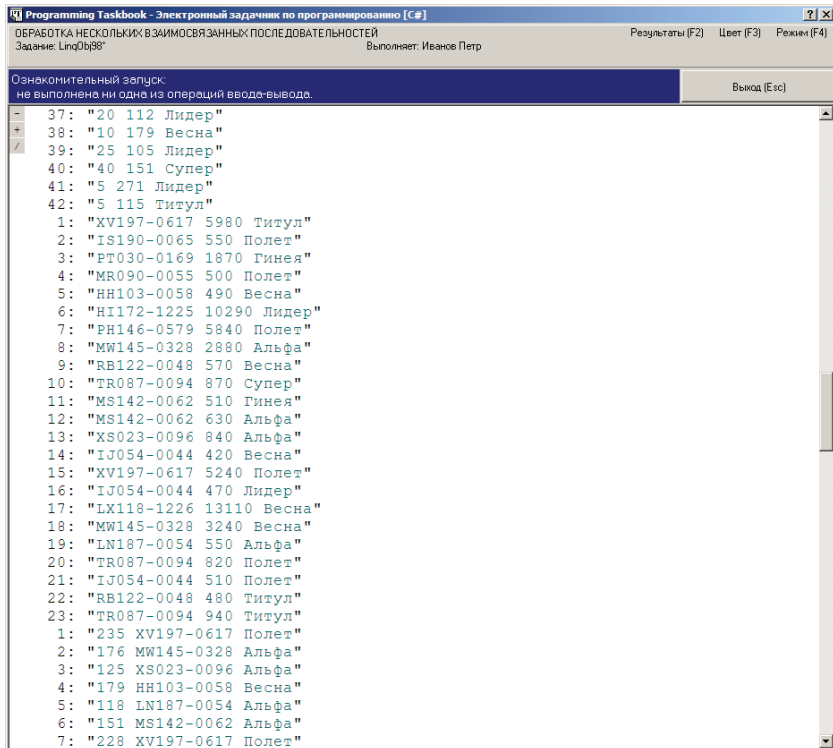


Рис. 34. Ознакомительный запуск задания LinqObj98
(полное отображение исходных файловых данных)

вательностей. Для большей наглядности присвоим этим последовательностям имена, которые используются в условии задачи:

```

var B = File.ReadAllLines(GetString(), Encoding.Default)
    .Select(e =>
    {
        string[] s = e.Split(' ');
        return new
        {
            art = s[1],
            cat = s[2]
        };
    });
var C = File.ReadAllLines(GetString(), Encoding.Default)
    .Select(e =>

```

```
{
    string[] s = e.Split(' ');
    return new
    {
        discount = int.Parse(s[0]),
        code = s[1],
        shop = s[2]
    };
});
var D = File.ReadAllLines(GetString(), Encoding.Default)
.Select(e =>
{
    string[] s = e.Split(' ');
    return new
    {
        art = s[0],
        price = int.Parse(s[1]),
        shop = s[2]
    };
});
var E = File.ReadAllLines(GetString(), Encoding.Default)
.Select(e =>
{
    string[] s = e.Split(' ');
    return new
    {
        code = s[0],
        art = s[1],
        shop = s[2]
    };
});
});
```

При решении задачи необходимо использовать все поля исходных последовательностей, кроме поля «Страна-производитель» из последовательности *B*. Для кода потребителя (представляющего по условию задачи целое число) допустимо в данном случае оставить его строковое представление. Приведем названия идентификаторов полей и их типы:

- ❑ art – «Артикул товара» (строка);
- ❑ cat – «Категория» (строка);
- ❑ discount – «Скидка (в процентах)» (целое число);
- ❑ code – «Код потребителя» (строка);
- ❑ shop – «Название магазина» (строка);
- ❑ price – «Цена (в рублях)» (целое число).

Чтобы убедиться в правильности построения последовательностей, достаточно добавить к их инициализирующим выражениям

вызов метода Show со строковым параметром – именем последовательности. Фрагмент данных, выведенных в раздел отладки, показан на рис. 35 (он содержит два последних элемента последовательности *C*, всю последовательность *D* и три начальных элемента последовательности *E*). Заметим, что в окне задачника на приведенном рисунке скрыты все разделы, кроме раздела отладки; для выполнения этого действия достаточно нажать клавишу пробела (повторное нажатие клавиши пробела восстанавливает разделы).

```

56> { discount = 20, code = 233, shop = Лидер }
57> { discount = 10, code = 104, shop = Люкс }
58> D 18: { art = MH115-0569, price = 5630, shop = Весна }
59> { art = IP019-0090, price = 910, shop = Титул }
60> { art = CL072-0025, price = 230, shop = Титул }
61> { art = MH115-0569, price = 6250, shop = Дукал }
62> { art = RS054-1024, price = 11570, shop = Весна }
63> { art = QY139-0025, price = 260, shop = Альфа }
64> { art = GI138-0161, price = 1910, shop = Титул }
65> { art = RD184-1857, price = 18570, shop = Альфа }
66> { art = RS071-0297, price = 2820, shop = Альфа }
67> { art = RD184-1857, price = 18750, shop = Титул }
68> { art = PG033-0072, price = 620, shop = Альфа }
69> { art = GU073-0089, price = 830, shop = Титул }
70> { art = CU173-0036, price = 410, shop = Люкс }
71> { art = QZ159-1072, price = 10180, shop = Дукал }
72> { art = CU173-0036, price = 400, shop = Весна }
73> { art = RS054-1024, price = 10240, shop = Люкс }
74> { art = CL072-0025, price = 260, shop = Весна }
75> { art = QG129-0692, price = 7260, shop = Альфа }
76> E 47: { code = 197, art = GU073-0089, shop = Титул }
77> { code = 287, art = RS054-1024, shop = Весна }
78> { code = 117, art = QG129-0692, shop = Альфа }
  
```

Рис. 35. Отладочная печать исходных последовательностей

На втором этапе решения преобразуем последовательность *E*, добавив в нее дополнительные данные из последовательностей *B*, *C* и *D*.

Напомним, что последовательность *E* содержит информацию о проданных товарах (указываются код потребителя, купившего товар, артикул товара и название магазина).

Прежде всего дополним каждый элемент последовательности *E* информацией о *категории* проданного товара; для этого достаточно выполнить внутреннее объединение последовательностей *E* и *B* по ключу – артикулу товара:

```

E.Join(B, e1 => e1.art, e2 => e2.art,
      (e1, e2) => new { e1.code, e1.art, e1.shop, e2.cat })
  
```

При определении анонимного типа для элементов полученного объединения мы указали для каждого поля только инициализирую-

щие значения (определяемые полями исходных последовательностей), не указывая *имена* определяемых полей; в этом случае поле получает имя, совпадающее с именем поля-инициализатора.

Далее нам необходимо включить в информацию о проданном товаре его цену, которая определяется в последовательности *D* и зависит от магазина, в котором товар был продан. Для этого к последовательности, полученной в результате выполнения указанного выше метода `Join`, надо применить еще один метод `Join`, выполняющий внутреннее объединение по *составному ключу* – артикулу товара и названию магазина:

```
.Join(D, e1 => e1.art + e1.shop, e2 => e2.art + e2.shop,
    (e1, e2) => new { e1.code, e1.art, e1.shop, e1.cat, e2.price })
```

Для просмотра полученной последовательности применим к ней метод `Show`:

```
E.Join(B, e1 => e1.art, e2 => e2.art,
    (e1, e2) => new { e1.code, e1.art, e1.shop, e2.cat })
.Join(D, e1 => e1.art + e1.shop, e2 => e2.art + e2.shop,
    (e1, e2) => new { e1.code, e1.art, e1.shop, e1.cat, e2.price })
.Show();
```

Раздел отладки с начальным содержимым полученной последовательности приведен на рис. 36; для сравнения на рис. 37 приведено начало последовательности *E* из раздела исходных данных.

Осталось выполнить объединение полученной последовательности и последовательности *C*, в результате которого мы найдем *размер скидки* для каждого проданного товара. Скидка определяется парой «потребитель–магазин»: магазин может предоставлять некоторым потребителям фиксированную скидку *d* (в процентах) на

```
1> 73: { code = 132, art = TC173-1507, shop = Весна, cat = Игрушки, price = 15070 }
2> { code = 277, art = TM168-0226, shop = Весна, cat = Игрушки, price = 2070 }
3> { code = 122, art = SS016-0316, shop = Гинья, cat = Быт.техника, price = 2970 }
4> { code = 133, art = CW197-0117, shop = Титул, cat = Компьютеры, price = 1280 }
5> { code = 122, art = SS016-0316, shop = Гинья, cat = Быт.техника, price = 2970 }
6> { code = 290, art = TQ057-0589, shop = Гинья, cat = Игрушки, price = 6120 }
7> { code = 272, art = CW197-0117, shop = Титул, cat = Компьютеры, price = 1280 }
8> { code = 156, art = UB161-0263, shop = Гинья, cat = Мебель, price = 2650 }
9> { code = 132, art = NR190-0335, shop = Титул, cat = Компьютеры, price = 3910 }
10> { code = 149, art = TQ057-0589, shop = Гинья, cat = Игрушки, price = 6120 }
11> { code = 185, art = CW197-0117, shop = Титул, cat = Компьютеры, price = 1280 }
12> { code = 130, art = SQ200-0071, shop = Титул, cat = Быт.техника, price = 830 }
13> { code = 103, art = TM168-0226, shop = Весна, cat = Игрушки, price = 2070 }
```

Рис. 36. Отладочная печать промежуточного объединения

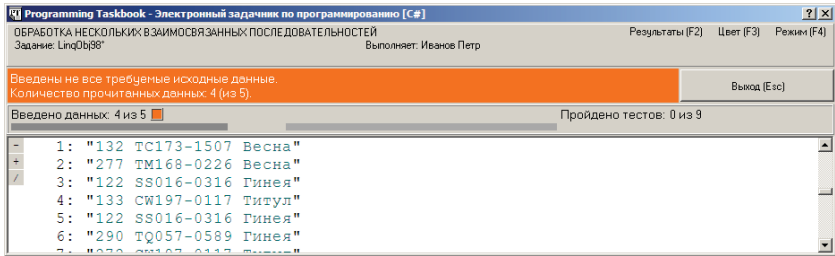


Рис. 37. Начало последовательности E, приведенной в разделе исходных данных

все товары. Формула для вычисления величины скидки для товара стоимостью p приведена в преамбуле к рассматриваемой подгруппе группы LinqObj (см. п. 3.2): $p * d / 100$, где символ «/» обозначает операцию *целочисленного деления* (таким образом, скидка для каждого товара выражается целым числом).

В данном случае нельзя применить метод Join, обеспечивающий получение внутреннего объединения, так как для некоторых возможных пар «потребитель–магазин» скидка не определена (то есть некоторые потребители не имеют скидки в некоторых магазинах). Вместо этого используем метод GroupJoin, возвращающий *левое внешнее объединение*, в котором каждому элементу внешней последовательности (полученной ранее в результате выполненных объединений) сопоставляется *набор* элементов внутренней последовательности C. Объединение выполняется по составному ключу – коду потребителя и названию магазина. В результирующем объединении достаточно сохранять три поля: категорию товара, название магазина и размер скидки для данного товара (прочие поля были нужны для построения объединений и вычисления скидки и в дальнейшем не потребуются):

```
.GroupJoin(C, e1 => e1.code + e1.shop, e2 => e2.code + e2.shop,
    (e1, ee2) => new {e1.shop, e1.cat,
        discount = ee2.Select(e => e.discount)
            .FirstOrDefault() * e1.price / 100 })
```

Прокомментируем способ вычисления скидки. Набор ee2 (включающий элементы внутренней последовательности C с тем же ключом, что и элемент e1 внешней последовательности) может содержать либо 1 элемент (если для текущего ключа «потребитель–магазин» скидка определена), либо 0 элементов (если скидка от-

существует). Мы выполняем проецирование последовательности `ee2` элементов анонимного типа в последовательность числовых элементов (данное проецирование будет успешно выполнено и в случае пустой последовательности `ee2`), а затем применяем к полученной числовой последовательности метод `FirstOrDefault`, возвращающий первый (и в данном случае единственный) элемент непустой последовательности или значение по умолчанию для целого типа (то есть число 0, соответствующее нулевой скидке) в случае пустой последовательности. Полученное число, равное скидке в процентах, используется в формуле для вычисления скидки для конкретного товара.

Примечание. Применять метод `FirstOrDefault` непосредственно к последовательности `ee2` не следует, так как в этом случае для пустой последовательности метод вернул бы значение `null` (значение по умолчанию для анонимного типа), которое нельзя использовать в числовых выражениях. Впрочем, требуемый результат можно получить, не выполняя проецирования, с помощью тернарной операции:

```
discount = ee2.Any() ? ee2.First().discount * e1.price / 100 : 0
```

В данном выражении мы использовали метод `Any`, возвращающий значение `true` для непустой последовательности и `false` в противном случае. Во втором операнде тернарной операции для доступа к первому элементу последовательности вместо метода `FirstOrDefault` можно использовать метод `First`.

Метод `Any` является запросом-квантификатором; к этой же категории относятся методы `All`, `Contains` и `SequenceEqual`; все они возвращают логическое значение.

Метод `All` имеет параметр-предикат; этот метод возвращает `true`, если *все* элементы последовательности удовлетворяют указанному предикату. Метод `Any` также может иметь параметр-предикат; в этом случае метод возвращает `true`, если найдется хотя бы один элемент последовательности, удовлетворяющий указанному предикату.

Метод `Contains` имеет параметр `value`, тип которого совпадает с типом элементов последовательности; он возвращает `true`, если последовательность содержит элемент со значением `value`.

Метод `SequenceEqual` позволяет сравнить две последовательности на равенство (последовательности считаются равными, если они содержат одинаковый набор элементов в том же самом порядке). Данный метод применяется к первой из сравниваемых последовательностей, а вторая последовательность указывается в качестве его параметра.

Представим в одном выражении все описанные выше операции объединения и сохраним полученную последовательность под именем discounts:

```
var discounts = E.Join(B, e1 => e1.art, e2 => e2.art,
    (e1, e2) => new { e1.code, e1.art, e1.shop, e2.cat })
    .Join(D, e1 => e1.art + e1.shop, e2 => e2.art + e2.shop,
    (e1, e2) => new { e1.code, e1.art, e1.shop, e1.cat, e2.price })
    .GroupJoin(C, e1 => e1.code + e1.shop, e2 => e2.code + e2.shop,
    (e1, ee2) => new
    {
        e1.shop,
        e1.cat,
        discount = ee2.Select(e => e.discount)
            .FirstOrDefault() * e1.price / 100
    });
```

Перейдем к третьему, завершающему этапу решения задачи: обработке полученной последовательности.

Если бы в задаче требовалось вывести данные только о тех парах «категория–магазин», которые встречаются в последовательности discounts, то было бы достаточно выполнить группировку этой последовательности по составному ключу, включающему поля cat и shop, отсортировать полученную последовательность и найти сумму скидок по всем товарам, имеющим общий ключ:

```
var r = discounts
    .GroupBy(e => e.cat + " " + e.shop, e => e.discount)
    .OrderBy(e => e.Key)
    .Select(e => e.Key + " " + e.Sum());
```

Добавив к решению указанный фрагмент вместе с завершающим оператором

```
File.WriteAllLines(GetString(), r.ToArray(), Encoding.Default);
```

обеспечивающим сохранение последовательности в текстовом файле, и запустив программу, мы получим результат, подобный приведенному на рис. 38. Отличие от правильного решения состоит в том, что в полученной последовательности отсутствует информация о парах «категория–магазин», которым *не соответствует ни одного проданного товара*.

Для того чтобы учесть отсутствующие пары, поступим следующим образом. Вначале сформируем набор cats всех категорий и на-

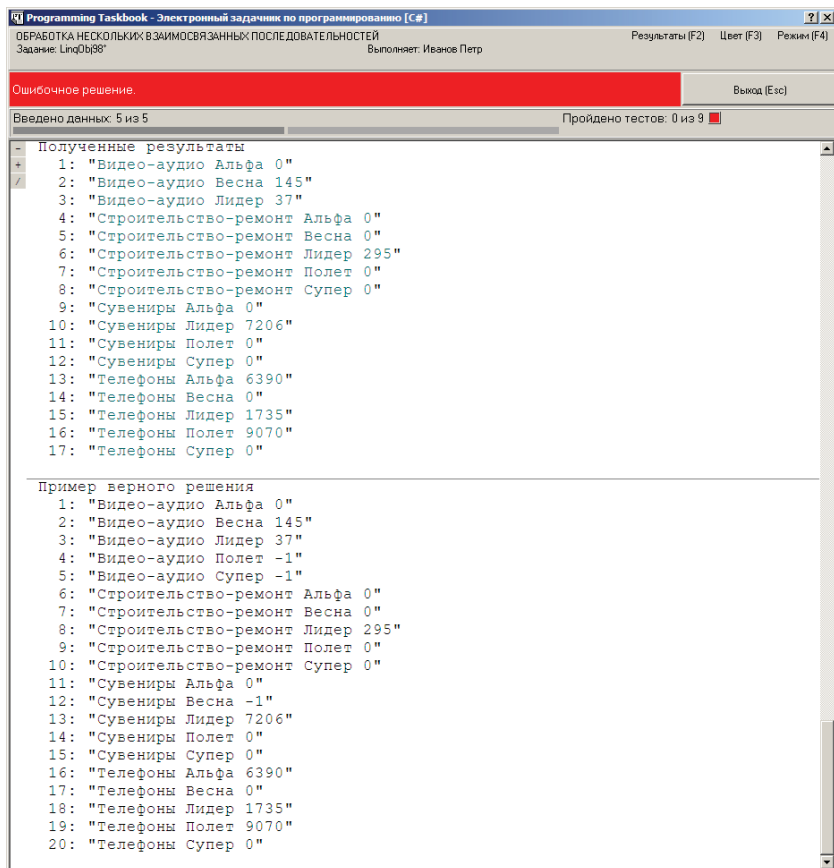


Рис. 38. Ошибочное выполнение задания LinqObj98

бор shops всех магазинов, используя группировку последовательности *B* по ключу *cat* и группировку последовательности *E* по ключу *shop* (в сгруппированных последовательностях достаточно сохранить только набор ключей группировки):

```
var cats = B.GroupBy(e => e.cat, (k, ee) => k);
var shops = E.GroupBy(e => e.shop, (k, ee) => k);
```

Затем получим перекрестное объединение наборов *cats* и *shops* в виде набора строк, используя комбинацию методов *SelectMany* и *Select* (см. п. 5.5.2), и отсортируем результат:

```
var cats_shops =  
    cats.SelectMany(e1 => shops.Select(e2 => e1 + " " + e2))  
        .OrderBy(e => e);
```

И наконец, выполним левое внешнее объединение последовательностей `cats_shops` и `discounts` методом `GroupJoin`, вернув результат в виде набора строк (дополнительная сортировка этого набора уже не потребуется):

```
var r = cats_shops.GroupJoin(discounts, e1 => e1,  
    e2 => e2.cat + " " + e2.shop, (e1, ee2) => e1 + " " +  
    ee2.Select(e => e.discount).DefaultIfEmpty(-1).Sum());
```

При обработке последовательности `ee2` мы учли, что она может оказаться пустой, причем в этом случае следует вернуть значение `-1`.

Добавив оператор, обеспечивающий сохранение последовательности `r` в текстовом файле и запустив программу, мы получим сообщение о первом успешном тестовом запуске, а запустив программу девять раз – сообщение о том, что задание выполнено.

Данное решение допускает две модификации: во-первых, при выполнении объединения промежуточной последовательности с последовательностью *D* в результирующей последовательности можно не сохранять поле «Артикул», так как в дальнейшем оно не требуется; во-вторых, при последующем объединении этой последовательности с последовательностью *C* (методом `GroupJoin`) можно объединить поля «Категория» и «Магазин», поскольку в дальнейшем они всегда будут использоваться совместно.

Анализируя решение, нетрудно заметить, что все именованные вспомогательные последовательности (`discounts`, `cats`, `shops`, `cats_shops`) в дальнейшем коде используются по одному разу, и поэтому обращение к ним можно заменить их инициализирующими выражениями. Однако к подобной возможности следует относиться с осторожностью, чтобы не получить излишне громоздкое и поэтому сложное для восприятия выражение. Можно, например, сохранить имена `discounts`, `cats`, `shops` и отказаться от именования перекрестного объединения `cats_shops`.

Приведем, с учетом этих замечаний, решение задачи, опустив для краткости фрагмент, связанный с определением исходных последовательностей (место этого фрагмента помечено соответствующим комментарием):

```

public static void Solve()
{
    Task("LinqObj98");

    // здесь должен находиться фрагмент, обеспечивающий
    // ввод исходных последовательностей B, C, D и E

    var discounts = E.Join(B, e1 => e1.art, e2 => e2.art,
        (e1, e2) => new { e1.code, e1.art, e1.shop, e2.cat })
        .Join(D, e1 => e1.art + e1.shop, e2 => e2.art + e2.shop,
            (e1, e2) => new { e1.code, e1.shop, e1.cat, e2.price })
        .GroupJoin(C, e1 => e1.code + e1.shop, e2 => e2.code + e2.shop,
            (e1, ee2) => new
            {
                cat_shop = e1.cat + " " + e1.shop,
                discount = ee2.Select(e => e.discount)
                    .FirstOrDefault() * e1.price / 100
            });
    var cats = B.GroupBy(e => e.cat, (k, ee) => k);
    var shops = E.GroupBy(e => e.shop, (k, ee) => k);
    var r = cats
        .SelectMany(e1 => shops.Select(e2 => e1 + " " + e2))
        .OrderBy(e => e)
        .GroupJoin(discounts, e1 => e1, e2 => e2.cat_shop,
            (e1, ee2) => e1 + " " +
                ee2.Select(e => e.discount).DefaultIfEmpty(-1).Sum());
    File.WriteAllLines(GetString(), r.ToArray(), Encoding.Default);
}

```

Приведем также вариант решения, использующий выражение запроса. Благодаря более компактному способу записи операций, связанных с объединением, в нем можно обойтись без именования промежуточных последовательностей. При этом удобно вначале построить перекрестное объединение категорий и магазинов, а затем обратиться в преобразованию последовательности проданных товаров.

```

public static void Solve()
{
    Task("LinqObj98");

    // здесь должен находиться фрагмент, обеспечивающий
    // ввод исходных последовательностей B, C, D и E

    var r =
        from e1 in
            // нахождение последовательности cats
            from e in B
            group e by e.cat

```

```

    // конец вложенного запроса
    from e2 in
    // нахождение последовательности shops
    from e in E
    group e by e.shop
    // конец вложенного запроса
    let s = e1.Key + " " + e2.Key
    orderby s
    select s
    into e1
    join e2 in
    // нахождение последовательности discounts
    from e1 in E
    join e2 in B
    on e1.art equals e2.art
    join e3 in D
    on e1.art + e1.shop equals e3.art + e3.shop
    join e4 in C
    on e1.code + e1.shop equals e4.code + e4.shop
    into ee4
    select new
    {
        cat_shop = e2.cat + " " + e1.shop,
        discount = (from e in ee4 select e.discount)
            .FirstOrDefault() * e3.price / 100
    }
    // конец вложенного запроса
    on e1 equals e2.cat_shop
    into ee2
    select e1 + " " +
        (from e in ee2 select e.discount)
            .DefaultIfEmpty(-1).Sum();
    File.WriteAllLines(GetString(), r.ToArray(), Encoding.Default);
}

```

В построенном выражении запроса содержатся три вложенных запроса, связанных с нахождением последовательностей, которые в предыдущем варианте решения имели имена `cats`, `shops` и `discounts`. Начало и конец каждого из вложенных запросов для наглядности помечены соответствующими комментариями.

Следует обратить внимание на возможность указания «цепочек» конструкций `join`, которая была использована при построении последовательности `discounts`. После каждой конструкции `join` указывается выражение `on ... equals`, определяющее ключи данного объединения; в случае внешнего объединения (аналога метода `GroupJoin`) дополнительно указывается выражение `into`, вводящее переменную-перечислитель для набора элементов внутренней последователь-

ности, соответствующих элементу внешней последовательности. В конструкции `select`, завершающей цепочку, можно использовать переменные-перечислители, связанные со *всеми* последовательностями, указанными в предыдущих конструкциях `join`.

Завершая обсуждение задачи LinqObj98, приведем фрагмент кода, содержащий выражение запроса для более простой задачи, в которой требуется вывести информацию только о тех парах «категория–магазин», которые присутствуют в последовательности проданных товаров (ранее мы приводили соответствующий фрагмент, использующий вызовы методов LINQ):

```
var r =
    from e1 in E
    join e2 in B
    on e1.art equals e2.art
    join e3 in D
    on e1.art + e1.shop equals e3.art + e3.shop
    join e4 in C
    on e1.code + e1.shop equals e4.code + e4.shop
    into ee4
    select new
    {
        cat_shop = e2.cat + " " + e1.shop,
        discount = (from e in ee4 select e.discount)
                    .FirstOrDefault() * e3.price / 100
    }
    into e
    orderby e.cat_shop
    group e.discount by e.cat_shop
    into e
    select e.Key + " " + e.Sum();
```

При нахождении последовательности с применением указанного выражения запроса результат выполнения программы будет аналогичен приведенному на рис. 38.



Глава 7. Примеры решения задач из группы LinqXml

Задания группы LinqXml посвящены технологии LINQ to XML – интерфейсу LINQ, предназначенному для обработки документов XML.

Интерфейс LINQ to XML включает, помимо дополнительных методов расширения, набор классов, связанных с различными компонентами XML. Этот набор образует *объектную модель документа XML* (XML Document Object Model), для которой используются сокращенные обозначения XML DOM или X-DOM. Основные классы, входящие в X-DOM, а также понятия, относящиеся к XML-документам, кратко описаны в преамбуле к группе LinqXml (см. гл. 4). Многие классы, входящие в X-DOM, имеют свойства, возвращающие различные последовательности; некоторые методы классов (в частности, их конструкторы) могут принимать последовательности в качестве своих параметров. Во всех ситуациях, связанных с обработкой последовательностей, можно использовать базовые запросы LINQ, входящие в интерфейс LINQ to Objects и изученные нами при выполнении заданий из групп LinqBegin и LinqObj.

7.1. Создание XML-документа: LinqXml10

Знакомство с возможностями LINQ to XML естественно начать с создания XML-документов. Этой теме посвящена первая подгруппа группы LinqXml (см. п. 4.1). Рассмотрим последнее из заданий, входящих в эту подгруппу.

LinqXml10. Даны имена существующего текстового файла и создаваемого XML-документа. Создать XML-документ с кор-

невым элементом `root`, элементами первого уровня `line` и инструкциями обработки (инструкции обработки являются дочерними узлами корневого элемента). Если строка текстового файла начинается с текста «data:», то она (без текста «data:») добавляется в XML-документ в качестве данных к очередной инструкции обработки с именем `instr`, в противном случае строка добавляется в качестве дочернего текстового узла в очередной элемент `line`.

После создания с помощью программного модуля PT4Load проекта-заготовки для данного задания (см. п. 5.1.1), автоматического запуска среды Visual Studio и загрузки в нее созданного проекта на экране будет отображен файл `LinqXml10.cs`. Приведем начальную часть этого файла:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Xml.Linq;
using PT4;

namespace PT4Tasks
{
    public class MyTask : PT
    {
        // При решении задач группы LinqXml доступны следующие
        // дополнительные методы расширения, определенные в задачнике:
        //
        // Show() и Show(cmt) – отладочная печать последовательности,
        // cmt – строковый комментарий;
        //
        // Show(e => r) и Show(cmt, e => r) – отладочная печать
        // значений r, полученных из элементов e последовательности,
        // cmt – строковый комментарий.

        public static void Solve()
        {
            Task("LinqXml10");
        }
    }
}
```

Подобно ранее рассмотренным заготовкам, создаваемым для заданий групп LinqBegin и LinqObj, этот файл содержит набор директив using, описание класса MyTask с функцией Solve, в которую требуется ввести решение задачи, и описание вариантов вспомогательного метода расширения Show, оформленных в виде методов класса LinqXml (в приведенном тексте описание класса LinqXml опущено). В комментарии, предшествующем функции Solve, кратко описываются варианты метода Show. Напомним, что этот метод предназначен для отладочной печати последовательностей (см. п. 5.3).

Обратите внимание на директиву подключения пространства имен System.Xml.Linq, с которым связаны все классы модели X-DOM.

В созданной заготовке отсутствуют дополнительные методы для ввода-вывода последовательностей (подобные методам GetEnumerableInt и GetEnumerableString и методу расширения Put, используемым при выполнении заданий группы LinqBegin). Это связано с тем, что в большинстве заданий требуется обработать XML-документ, хранящийся во внешнем файле, и записать в этот же файл результат обработки. Для операций файлового чтения-записи XML-документов предусмотрены специальные методы класса XmlDocument, которые и следует использовать при выполнении заданий. Исключение составляют задания на создание XML-документов, исходные данные для которых содержатся в «обычных» текстовых файлах. При выполнении подобных заданий, как и заданий группы LinqObj, следует использовать метод File.ReadAllLines, обеспечивающий чтение всех строк исходного текстового файла в строковый массив (см. п. 6.1.2). Для ввода имен файлов и дополнительных строковых данных необходимо использовать функцию задачника GetString; для вывода данных базовых типов (подобный вывод требуется только в заданиях подгруппы, связанной с анализом содержимого XML-документа, – см. п. 4.2) надо использовать метод Put, также определенный в задачнике.

После запуска созданной программы-заготовки мы увидим на экране окно задачника, содержащее формулировку задания, а также пример исходных данных и правильных результатов (рис. 39).

Хотя файловые данные на приведенном рисунке отображаются в сокращенном виде, указанные фрагменты демонстрируют все особенности этих данных. В исходном текстовом файле содержатся строки, представляющие собой наборы слов, причем некоторые строки начинаются с текста «data:» (на рисунке этот текст содержит третья строка); результирующий файл содержит XML-документ

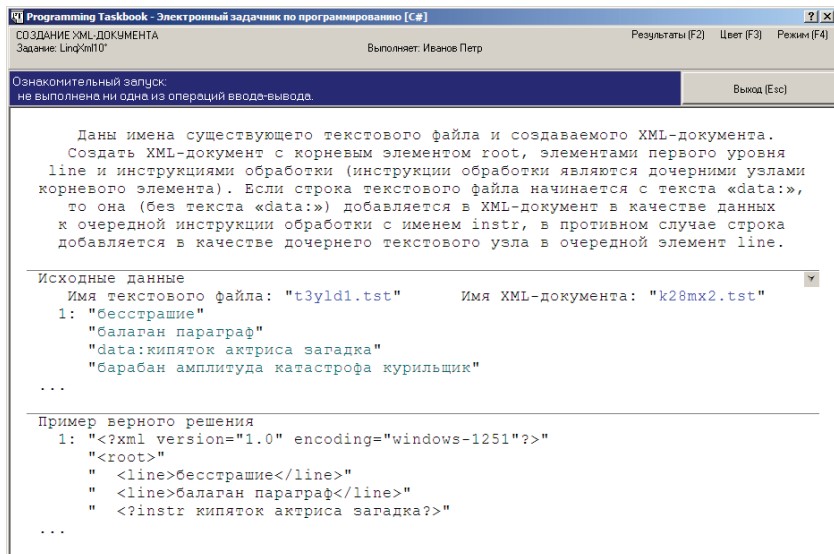


Рис. 39. Ознакомительный запуск задания LinqXml10

в кодировке «windows-1251», включающий корневой элемент `root`, дочерними узлами которого (то есть узлами *первого уровня*, поскольку корневой элемент считается узлом нулевого уровня) являются элементы `line` и инструкции обработки `instr`.

Элемент XML-документа обязательно имеет *имя*; он может быть представлен в виде *парных тегов* вида `<имя>...</имя>` (где на месте многоточия могут располагаться дочерние узлы этого элемента) или в виде *комбинированного тега* `<имя />`. Элемент может также содержать *атрибуты* (соответствующий пример мы рассмотрим в следующем пункте).

Инструкция обработки является особым компонентом XML-документа, который заключается в скобки вида `<? ?>` (заметим, что *объявление* XML-документа, с которого он начинается, также оформляется подобным образом). Первое слово в этих скобках считается *именем* инструкции обработки (точнее, именем ее программы-получателя), а последующий текст – *данными*, связанными с инструкцией. Инструкции обработки называются так потому, что они определяют не содержимое XML-документа, а дополнительные данные, предназначенные для программ, обрабатывающих этот документ. Разумеется, в образцах XML-документов, генерируемых

задачиком, используются инструкции обработки, не связанные ни с какими программами и имеющие лишь формальные признаки «настоящих» инструкций.

Данные для инструкций `instr` должны извлекаться из строк исходного файла, начинающихся с текста «data:»; прочие строки файла должны использоваться в качестве содержимого элементов `line`. Таким образом, при выполнении задания нам потребуется сформировать на основе строковой последовательности, полученной из исходного файла, последовательность дочерних узлов корневого элемента `root` для созданного XML-документа.

Как уже было отмечено, исходную строковую последовательность проще всего получить с помощью метода `File.ReadAllLines`. Так как все текстовые файлы, связанные с заданиями, имеют кодировку «windows-1251», в методе `File.ReadAllLines` необходимо указать дополнительный параметр `Encoding.Default`:

```
var a = File.ReadAllLines(GetString(), Encoding.Default);
```

Для создания как самого XML-документа, так и его компонентов следует использовать *конструкторы* соответствующих классов. При выполнении данного задания нам потребуются следующие классы, входящие в X-DOM: `XDocument` (XML-документ), `XDeclaration` (объявление документа), `XElement` (элемент) и `XProcessingInstruction` (инструкция обработки).

Важной особенностью конструкторов классов `XDocument` и `XElement` является то, что они позволяют сразу указывать содержимое создаваемого документа (соответственно, элемента) в виде дополнительных параметров, среди которых могут быть и параметры-последовательности (в случае параметров-последовательностей в создаваемый документ или элемент добавляются все элементы этих последовательностей). Указанной особенностью обладают и некоторые другие методы классов X-DOM. Благодаря этой особенности документ XML вместе с его содержимым может быть построен *в одном выражении*, причем при формировании содержимого можно использовать последовательности, получаемые с помощью методов LINQ. Подобный способ формирования XML-документа называется *функциональным конструированием*. Его достоинствами являются краткость и наглядность получаемого кода, по виду которого можно легко определить структуру определяемого XML-документа или любого его фрагмента.

В качестве примера функционального конструирования XML-документа приведем фрагмент, который по последовательности строк **a** (полученной ранее из исходного текстового файла) формирует документ с корневым элементом **root** и элементами первого уровня **line** с текстовыми значениями, взятыми из последовательности **a** в том же порядке:

```
XDocument d = new XDocument(  
    new XDeclaration(null, "windows-1251", null),  
    new XElement("root",  
        a.Select(e => new XElement("line", e))));
```

В данном фрагменте вначале вызывается конструктор класса **XDocument**; в списке его параметров указывается конструктор, создающий объявление документа (в нем будет отличаться от **null** лишь второй параметр, определяющий кодировку документа), и конструктор, создающий элемент нулевого уровня **root**. В конструкторе этого элемента вначале указывается его имя, а затем, в дополнительных параметрах, – его содержимое. В данном случае содержимое представляет собой *последовательность* элементов (то есть объектов типа **XElement**), полученную из исходной строковой последовательности с помощью метода проецирования **Select** (см. п. 5.4). Лямбда-выражение метода **Select** содержит вызов конструктора для объекта **XElement**; в нем в качестве имени указывается строковая константа «**line**», а в качестве содержимого – строка **e**.

Примечание. При определении содержимого в конструкторе класса **XElement** (и в других методах **X-DOM**, обеспечивающих функциональное конструирование) строки преобразуются в простейший вид узла XML-документа – текстовый узел, которому в модели **X-DOM** соответствует класс **XText**. Подчеркнем, что строки при функциональном конструировании не интерпретируются как символьные последовательности (в отличие от любых других параметров-последовательностей), а рассматриваются как единые объекты.

В приведенном фрагменте не обрабатываются особым образом строки исходной последовательности, начинающиеся с текста «**data:**» (которые надо преобразовывать не в элементы, а в инструкции обработки). Этот недочет мы исправим позже.

Осталось сохранить полученный XML-документ под требуемым именем. Для этого достаточно использовать метод **Save** класса **XDocument**, указав имя в качестве параметра:

```
d.Save(GetString());
```

При вызове этого метода не требуется дополнительно указывать используемую кодировку, поскольку она была ранее указана в объявлении XML-документа. Заметим также, что метод `Save` по умолчанию *форматирует* XML-документ при его сохранении в файле, выводя каждый узел на отдельной строке (или на нескольких строках, если узел является элементом, содержащим набор дочерних элементов) и добавляя в начало каждой строки необходимые отступы.

Примечание. При выполнении заданий возможность автоматического форматирования XML-документа оказывается очень полезной, так как она упрощает анализ полученного XML-документа и его сравнение с «правильным» образцом. В ситуации, когда программа генерирует XML-документ, не предназначенный для непосредственного просмотра человеком, можно отключить возможность форматирования, указав в методе `Save` необязательный второй параметр `SaveOptions.DisableFormatting`; это позволит уменьшить размер файла, содержащего документ, а также сохранить в нем все незначащие пробелы (по поводу незначащих пробелов см. также примечание в п. 7.3).

Можно получить текстовое представление XML-документа, не сохраняя его в файле; для этого достаточно вызвать для объекта типа `XDocument` метод `ToString`, возвращающий строку (этот метод тоже может иметь необязательный параметр `SaveOptions.DisableFormatting`).

Метод `Save` предусмотрен и для объектов типа `XElement`; он позволяет сохранить в файле текстовое представление отдельного элемента XML. Метод `ToString` объекта `XElement` возвращает строку с текстовым представлением XML-элемента.

Объединяя указанные выше операторы, получаем первый (пока еще не вполне правильный) вариант решения:

```
public static void Solve()
{
    Task("LinqXml10");
    var a = File.ReadAllLines(GetString(), Encoding.Default);
    XDocument d = new XDocument(
        new XDeclaration(null, "windows-1251", null),
        new XElement("root",
            a.Select(e => new XElement("line", e))));
    d.Save(GetString());
}
```

При запуске программы в окне будет выведено сообщение об ошибочном решении (см. рис. 40; для уменьшения размеров окна в нем скрыт раздел с формулировкой задания).

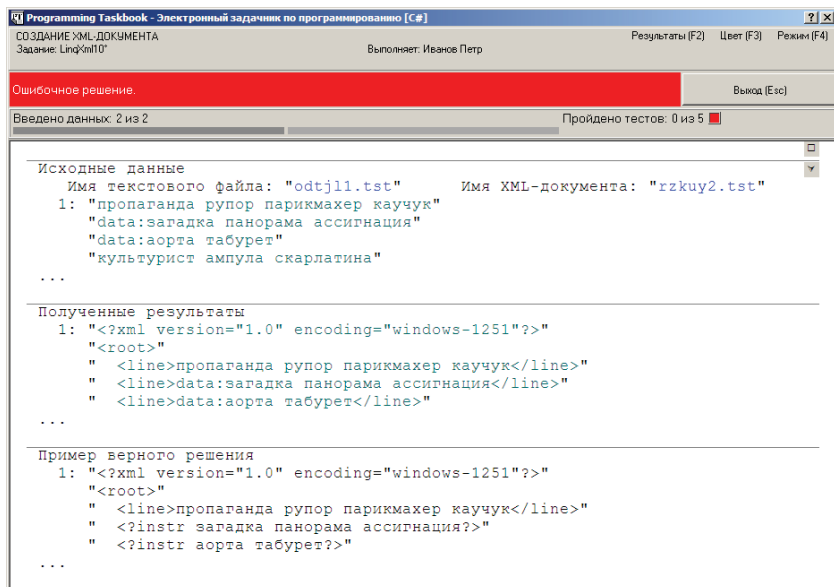


Рис. 40. Ошибочное выполнение задания LinqXml10

Для получения XML-документа, соответствующего условиям задачи, надо при создании документа `d` предусмотреть специальную обработку строк исходного файла, начинающихся с текста «data:», создавая для них вместо объектов `XElement` объекты `XProcessingInstruction`. Подобную обработку можно реализовать, включив в лямбда-выражение тернарную операцию:

```
e => e.StartsWith("data:") ?
    new XProcessingInstruction("instr", e.Substring(5)) :
    new XElement("line", e)
```

При создании инструкции обработки в качестве ее данных указывается подстрока строки `e` без начального текста «data:» (число 5 определяет индекс начального символа подстроки; напомним, что символы строки индексируются от 0).

Однако при попытке откомпилировать полученную программу будет выведено сообщение об ошибке компиляции. Ошибка заключается в том, что по приведенному лямбда-выражению нельзя определить *тип* элементов возвращаемой последовательности (часть элементов будет иметь тип `XProcessingInstruction`, а часть – тип `XElement`). Чтобы исправить данную ошибку, следует дополнить тернарное выражение операцией приведения к некоторому *общему типу*, который и будет считаться типом элементов возвращаемой последовательности. В качестве такого общего типа можно указать любой тип, являющийся предком как класса `XProcessingInstruction`, так и класса `XElement`. Ближайшим общим предком этих классов выступает класс `XNode`, являющийся абстрактным базовым классом для всех компонентов-узлов XML-документа (класс `XDocument` – также потомок класса `XNode`). Используя операцию приведения типа `as`, получаем следующий вариант лямбда-выражения (добавленный фрагмент выделен полужирным шрифтом):

```
e => e.StartsWith("data:") ?  
    new XProcessingInstruction("instr", e.Substring(5)) :  
    new XElement("line", e) as XNode
```

Примечание. Операция `as` как имеющая более высокий приоритет, чем тернарная операция, относится только к выражению, указанному после двоеточия. Однако этого достаточно для успешной компиляции программы, поскольку в ходе анализа типов возвращаемых значений (теперь это `XProcessingInstruction` и `XNode`) компилятор обнаружит, что тип `XProcessingInstruction` может быть приведен к типу `XNode`, и выполнит это приведение автоматически, получив в результате последовательность элементов типа `XNode`. Того же результата мы добились бы, указав текст `as XNode` перед двоеточием или заключив все тернарное выражение в круглые скобки.

Вместо типа `XNode` мы могли бы использовать и другие общие классы-предки классов `XProcessingInstruction` и `XElement`, например `XObject` (общий абстрактный базовый класс для *узлов* и *атрибутов* XML-документа) или `object` (общий предок всех классов .NET).

Подчеркнем, что приведение типа не изменяет *фактического типа* элементов последовательности, оно лишь обеспечивает единообразную интерпретацию всех этих элементов как *XML-узлов*, давая тем самым компилятору возможность определить тип получаемой последовательности (в нашем случае `IEnumerable<XNode>`).

Приведем окончательный вариант решения:

```
public static void Solve()
{
    Task("LinqXml10");
    var a = File.ReadAllLines(GetString(), Encoding.Default);
    XDocument d = new XDocument(
        new XDeclaration(null, "windows-1251", null),
        new XElement("root",
            a.Select(e => e.StartsWith("data:") ?
                new XProcessingInstruction("instr", e.Substring(5)) :
                new XElement("line", e) as XNode)));
    d.Save(GetString());
}
```

После пяти тестовых запусков программы мы получим сообщение о том, что задание выполнено.

7.2. Анализ содержимого XML-документа: LinqXml20

Задания LinqXml11–LinqXml20, входящие во вторую подгруппу группы LinqXml, посвящены анализу существующего XML-документа (см. п. 4.2). В этих и последующих заданиях группы LinqXml основным элементом исходных данных является XML-документ, который автоматически генерируется задачиком и сохраняется в файле (имя файла передается программе, выполняющей задание).

В заданиях данной подгруппы не требуется изменять исходный XML-документ; необходимо лишь проанализировать его содержимое и вывести результаты этого анализа. Для этого следует использовать дополнительные методы расширения, входящие в интерфейс LINQ to XML и позволяющие представить компоненты XML-документа (узлы и атрибуты) в виде *последовательностей* различных видов, анализ которых, в свою очередь, удобно проводить с помощью методов, входящих в интерфейс LINQ to Objects.

Продemonстрируем приемы совместного использования базовых методов LINQ и специализированных методов расширения LINQ to XML на примере последнего из заданий, входящих во вторую подгруппу.

LinqXml20. Дан XML-документ, содержащий хотя бы один элемент первого уровня. Для каждого элемента первого уров-

ня найти его элементы-потомки, имеющие максимальное количество атрибутов. Перебирая элементы первого уровня в порядке их появления в XML-документе, вывести имя элемента, число N – максимальное количество атрибутов у его потомков (значение N может быть равно 0), и имена потомков, имеющих N атрибутов (имена потомков выводить в алфавитном порядке; среди этих имен могут быть совпадающие). Если элемент первого уровня не содержит элементов-потомков, то в качестве значения N выводить -1 , а в качестве имени потомка – текст «no child».

После создания программы-заготовки для этого задания и ее запуска на экране появится окно, подобное приведенному на рис. 41.

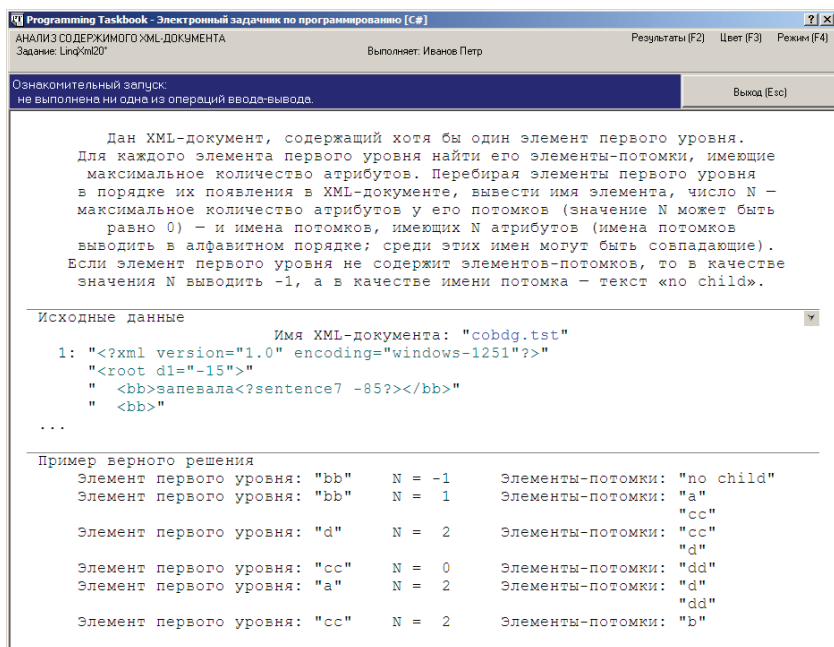


Рис. 41. Ознакомительный запуск задания LinqXml20 (режим сокращенного отображения исходного XML-документа)

Проанализируем данные, указанные в этом окне.

Несмотря на режим сокращенного отображения исходного документа, первый из элементов первого уровня отображен в окне пол-

ностью. Этот элемент имеет имя **bb** и включает два дочерних узла: текстовое содержимое (слово «запевала») и инструкцию обработки. Таким образом, данный элемент не содержит *элементов-потомков*, поэтому для него требуется вывести число **-1** и текст «no child».

Для анализа последующих элементов первого уровня следует перейти в режим полного отображения файловых данных (описание этого режима и связанных с ним возможностей содержится в п. 6.1.1). Соответствующий вариант окна приведен на рис. 42 (на этом рисунке, кроме того, скрыт раздел с формулировкой задания).

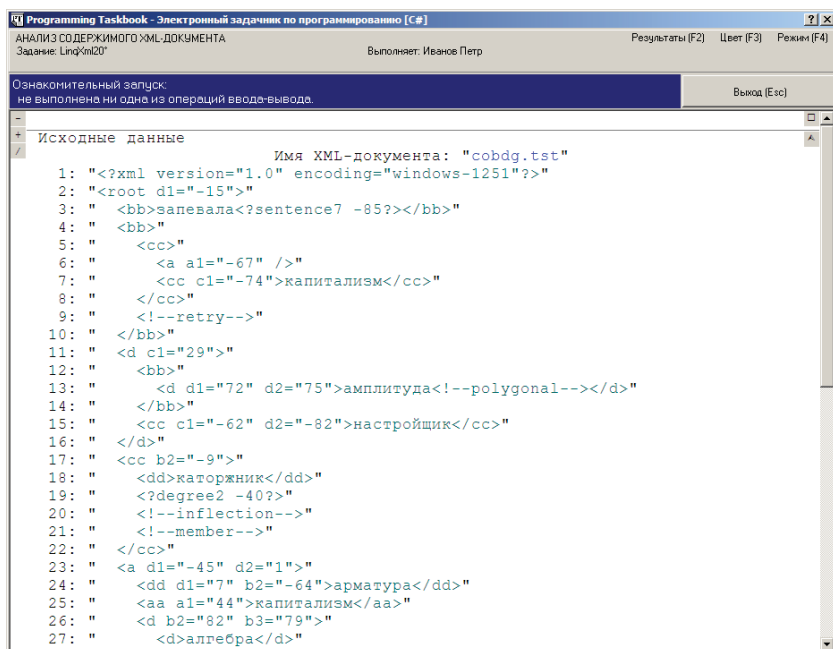


Рис. 42. Ознакомительный запуск задания LinqXml20 (режим полного отображения исходного XML-документа)

Анализируя приведенную часть документа, мы видим, что второй элемент первого уровня (который, как и первый, имеет имя **bb**), содержит три элемента-потомка: это его *дочерний* элемент (то есть непосредственный потомок) **cc** и два *потомка второго уровня* (то есть дочерних элемента его дочернего элемента): **a** и **cc**. Кроме того, ана-

лизируемый элемент **bb** включает *комментарий*, содержащий слово «retry» (комментарии в XML-документе заключаются в скобки вида `<!-- -->`). Оба потомка второго уровня содержат по одному *атрибуту*: элемент **a** имеет атрибут **a1**, элемент **cc** – атрибут **c1** (атрибуты указываются после имени элемента в его открывающем теге и всегда имеют вид **имя атрибута**=**"значение атрибута"**; при этом значение обязательно заключается в кавычки). Кроме того, элемент **cc** имеет текстовое содержимое «капитализм». Таким образом, максимальное количество атрибутов (равное 1) имеют два потомка элемента **bb**, поэтому в примере верного решения после имени второго элемента **bb** выведены число 1 и имена **a** и **cc**.

У следующего (третьего) элемента первого уровня (с именем **d**) имеется потомок второго уровня с именем **d** с двумя атрибутами (**d1** и **d2**) и дочерний элемент **cc**, также имеющий два атрибута (**c1** и **d2**). Поэтому после имени **d** выводятся число 2 и имена **cc** и **d** (согласно условию задачи имена должны быть отсортированы по алфавиту). Заметим, что элемент XML не может содержать атрибуты с одинаковыми именами (хотя его *дочерние элементы*, как мы уже заметили, могут иметь одинаковые имена).

Возможна ситуация, когда элемент первого уровня *содержит* элементы-потомки, но ни один из них не имеет атрибутов (в приведенном примере такими свойствами обладает четвертый элемент первого уровня **cc**). В этом случае максимальное число атрибутов равно 0, и после имени данного элемента первого уровня надо вывести число 0 и имена *всех* его элементов-потомков (в примере выведено единственное имя **dd**, поскольку остальные потомки элемента **cc** элементами не являются: это *текст* «каторжник», являющийся одновременно текстовым содержимым дочернего элемента **dd**, *инструкция обработки* и два *комментария*).

Мы видим, что для решения данной задачи (как и многих других задач из этой подгруппы и последующих подгрупп) было бы удобно использовать такие характеристики элементов XML-документа, как последовательность их *дочерних элементов*, последовательность всех их *элементов-потомков* (любого уровня), а также последовательность их *атрибутов*. В других ситуациях может потребоваться использовать последовательность *дочерних узлов* элемента и последовательность всех его *узлов-потомков* любого уровня (в эти последовательности, помимо элементов, будут входить и другие компоненты XML-документа, например комментарии, инструкции обработки и текстовое содержимое).

Все перечисленные характеристики предусмотрены в модели X-DOM и реализованы в виде методов, возвращающих последовательности типа `IEnumerable<T>`, где тип `T` может принимать значение `XElement` (для последовательности элементов), `XNode` (для последовательностей узлов) и `XAttribute` (для последовательности атрибутов).

Наиболее богатой коллекцией подобных методов обладает класс `XElement`. Приведем основные из них (все они возвращают последовательность, связанную с тем элементом, для которого они вызваны):

- ❑ `Attributes` – возвращает последовательность атрибутов;
- ❑ `Elements` – возвращает последовательность дочерних элементов;
- ❑ `Nodes` – возвращает последовательность дочерних узлов;
- ❑ `Descendants` – возвращает последовательность всех элементов-потомков любого уровня;
- ❑ `DescendantNodes` – возвращает последовательность всех узлов-потомков любого уровня.

Все эти методы, за исключением `Attributes`, определены и для класса `XDocument`, поскольку он также содержит узлы и элементы (хотя и не может содержать атрибутов). Полезно также помнить, что доступ к корневому элементу XML-документа можно получить с помощью свойства `Root` класса `XDocument`.

Класс `XElement` включает также методы `DescendantsAndSelf` и `DescendantNodesAndSelf`, которые отличаются от методов `Descendants` и `DescendantNodes` тем, что включают в результирующую последовательность тот элемент, для которого вызван метод.

При работе с последовательностями, возвращаемыми описанными выше методами, следует учитывать, что их элементы располагаются в том же порядке, в котором соответствующие им компоненты XML располагаются в XML-документе.

Для методов `Attributes`, `Elements` и `Descendants` предусмотрены перегруженные варианты с одним параметром `name` типа `XName`, позволяющие отфильтровать из соответствующих последовательностей только те атрибуты или элементы, которые имеют указанное имя `name`. Имеются методы `Attribute` и `Element` с таким же параметром, возвращающие *первый* атрибут или элемент с указанным именем (или `null` в случае их отсутствия).

Следует обратить внимание на то, что имена атрибутов и элементов имеют тип `XName`, и для преобразования имен в строку необхо-

можно явно вызвать метод ToString. Причина введения специального типа для имен атрибутов и элементов связана с тем, что эти имена могут включать дополнительную начальную часть – *пространство имен* (namespace). Работе с пространствами имен посвящена специальная подгруппа группы LinqXml (см. п. 4.5; пример решения одной из задач этой подгруппы приводится в п. 7.5); во всех предшествующих задачах имена атрибутов и элементов имеют *пустое* пространство имен, поэтому для получения строкового представления имени можно также использовать свойство LocalName класса XName, возвращающее *локальное имя* (без пространства имен).

Наконец, следует упомянуть метод, возвращающий последовательность *предков*: это метод Ancestors, который можно вызывать для любых объектов типа XElement, то есть для любых узлов. Последовательность начинается с ближайшего предка, называемого *родителем*, и заканчивается наиболее удаленным предком – корневым элементом XML-документа. Все предки являются *элементами XML*, то есть имеют тип XElement. Заметим, что количество элементов в последовательности Ancestors, возвращаемой для данного узла, равно *уровню* этого узла в документе (например, для корневого элемента Root, то есть элемента нулевого уровня, метод Ancestors возвращает пустую последовательность).

Важной особенностью интерфейса LINQ to XML является наличие вариантов всех перечисленных выше методов, которые можно вызывать не для отдельных элементов, а для их *последовательности*. С помощью подобных методов расширения можно, например, легко получить последовательность всех атрибутов, связанных с дочерними элементами некоторого элемента *e*; для этого достаточно использовать цепочку из двух методов: *e.Elements().Attributes()*. Можно также получить последовательность всех элементов-потомков второго уровня (то есть дочерних элементов дочерних элементов), используя цепочку из вызовов двух методов Elements: *e.Elements().Elements()*.

Воспользуемся некоторыми из перечисленных методов для решения задачи LinqXml20.

Прежде всего необходимо загрузить из файла исходный XML-документ. Для этого достаточно вызвать статический метод Load класса XDocument, передав ему в качестве параметра имя файла (кодировку указывать не требуется, так как информация о кодировке содержится в самом файле – в объявлении XML-документа):

```
XDocument d = XDocument.Load(GetString());
```

Примечание. В классе `XDocument` имеется еще один статический метод, позволяющий сформировать XML-документ: это метод `Parse` со строковым параметром, содержащим текстовое представление XML-документа.

Статические методы `Load` и `Parse` имеются и в классе `XElement`; с их помощью можно конструировать *элемент* XML на основе его текстового представления, хранящегося в файле или в строке.

Теперь следует организовать перебор всех элементов первого уровня (в порядке их следования в документе). Для этого достаточно использовать последовательность, возвращаемую методом `Elements` для корневого элемента `d.Root`:

```
foreach (var e1 in d.Root.Elements())  
{  
    // обработка элемента e1  
}
```

При обработке элемента `e1` следует вначале вывести его имя, затем максимальное число атрибутов у его элементов-потомков и, наконец, имена элементов-потомков, имеющих максимальное число атрибутов.

Будем последовательно добавлять в цикл `foreach` те данные, которые требуется вывести. Для того чтобы задачник не проверял промежуточные результаты решения, организуем их вывод в разделе отладки, используя функции `Show` и `ShowLine`.

Вначале просто выведем имена всех элементов первого уровня:

```
foreach (var e1 in d.Root.Elements())  
{  
    Show(e1.Name.LocalName);  
}
```

Сравнение выведенных данных с примером правильного решения показывает, что программа действительно перебирает все элементы первого уровня, причем в порядке их следования в документе (см. рис. 43).

Для определения максимального числа атрибутов среди элементов-потомков следует использовать метод `Descendants`, возвращающий элементы-потомки всех уровней, и метод `Attributes`, возвращающий все атрибуты элемента. Необходимо также предусмотреть особую обработку ситуации, когда анализируемый элемент не содержит ни одного элемента-потомка:

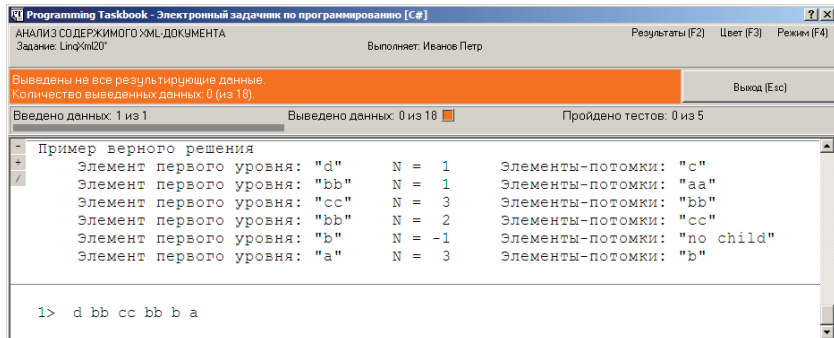


Рис. 43. Первый этап выполнения задания LinqXml20

```
int max = e1.Descendants()
    .Select(e => e.Attributes().Count())
    .DefaultIfEmpty(-1).Max();
```

Используя метод `Select`, мы построили вспомогательную последовательность, содержащую *количество атрибутов* для всех элементов-потомков, после чего применили к ней метод `Max`. Перед применением метода `Max` мы вызвали метод `DefaultIfEmpty`, который преобразует пустую последовательность в последовательность с единственным элементом, значение которого указывается в качестве параметра этого метода. Метод `DefaultIfEmpty` позволяет нам правильно обрабатывать элементы, не содержащие элементов-потомков, возвращая для них значение `-1`.

Добавим к полученному фрагменту решения вывод значения `max` в раздел отладки:

```
ShowLine(max);
```

В данном случае мы использовали функцию `ShowLine`; это позволит выводить информацию о каждом элементе первого уровня на отдельной экранной строке.

Запуск полученной программы показывает, что данный этап решения мы также прошли успешно (см. рис. 44).

Завершающий этап, посвященный нахождению имен элементов-потомков, имеющих максимальное число атрибутов, реализуется аналогичным образом:

```
var a = e1.Descendants()
    .Where(e => e.Attributes().Count() == max)
```

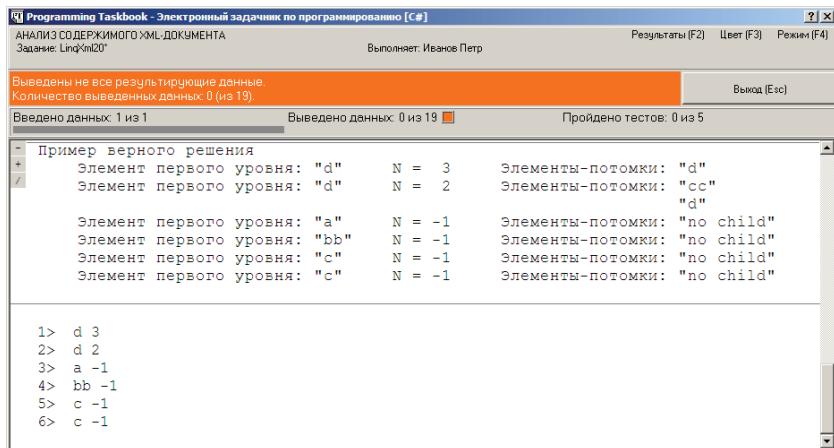


Рис. 44. Второй этап выполнения задания LinqXml20

```

.Select(e => e.Name.LocalName)
.OrderBy(e => e)
.DefaultIfEmpty("no child");

```

Здесь вначале выполняется фильтрация последовательности Descendants с помощью метода Where, после чего на основе отфильтрованной последовательности элементов формируется последовательность их имен, которая сортируется и в заключение обрабатывается методом DefaultIfEmpty, чтобы получить последовательность с единственным элементом «no child» вместо пустой последовательности. Для вывода элементов полученной последовательности надо использовать цикл foreach.

Объединив все описанные этапы решения задачи и заменив методы Show и ShowLine на метод Put, обеспечивающий вывод данных в раздел результатов, получаем первый вариант правильного решения:

```

public static void Solve()
{
    Task("LinqXml20");
    XDocument d = XDocument.Load(GetString());
    foreach (var e1 in d.Root.Elements())
    {
        Put(e1.Name.LocalName);
        int max = e1.Descendants()
            .Select(e => e.Attributes().Count())

```

```

        .DefaultIfEmpty(-1).Max();
    Put(max);
    var a = e1.Descendants()
        .Where(e => e.Attributes().Count() == max)
        .Select(e => e.Name.LocalName)
        .OrderBy(e => e)
        .DefaultIfEmpty("no child");
    foreach (var e in a)
        Put(e);
}
}

```

Обе цепочки методов LINQ, использованные в решении, можно представить в виде *выражений запросов* (см. п. 5.5.3); в результате получаем еще один вариант правильного решения:

```

public static void Solve()
{
    Task("LinqXml20");
    XDocument d = XDocument.Load(GetString());
    foreach (var e1 in d.Root.Elements())
    {
        Put(e1.Name.LocalName);
        var max = (from e in e1.Descendants()
                    select e.Attributes().Count())
                    .DefaultIfEmpty(-1).Max();

        Put(max);
        var a = (from e in e1.Descendants()
                  where e.Attributes().Count() == max
                  let name = e.Name.LocalName
                  orderby name
                  select name)
                .DefaultIfEmpty("no child");
        foreach (var e in a)
            Put(e);
    }
}

```

7.3. Преобразование XML-документа: LinqXml28, LinqXml32, LinqXml37

Обратимся к подгруппе группы LinqXml, посвященной простейшим преобразованиям XML-документа (см. п. 4.3). Задания этой подгруппы можно разбить на две части: первая часть (LinqXml21–LinqXml30) содержит задания на *удаление* компонентов XML-до-

кумента, вторая часть (LinqXml31–LinqXml40) – задания на *вставку и изменение* компонентов.

В этих и последующих заданиях группы LinqXml программа должна преобразовать исходный XML-документ и сохранить его в том же файле, в котором содержался исходный документ.

Вначале рассмотрим одно из заданий на удаление компонентов.

LinqXml28. Дан XML-документ. Удалить дочерние текстовые узлы для всех элементов третьего уровня. Если текстовый узел является единственным дочерним узлом элемента, то после его удаления элемент должен быть представлен в виде комбинированного тега.

Указание. Использовать метод `OfType<XText>`.

Для удаления компонентов (атрибутов и узлов) в модели X-DOM предусмотрено несколько методов, основным из которых является метод `Remove`, имеющийся и в классе `XAttribute`, и в классе `XNode` (а следовательно, и в классе `XElement`, который является потомком класса `XNode`). Вызов этого метода для некоторого компонента `e` удаляет этот компонент из списка компонентов (списка атрибутов или списка узлов некоторого элемента XML), в котором он содержался. Если компонент `e` не содержится в списке компонентов (в этом случае его поле `Parent`, хранящее ссылку на его родительский элемент, равно `null`), то вызов `e.Remove()` приводит к возбуждению исключения.

Важной особенностью метода `Remove` является то, что он реализован также в виде метода расширения и может вызываться не только для отдельного атрибута или узла, но и для *последовательности* атрибутов или узлов. Таким образом, для удаления из документа набора узлов или атрибутов достаточно сформировать последовательность удаляемых компонентов (используя методы LINQ), после чего вызвать для нее метод `Remove`.

Для объектов класса `XElement` определены методы без параметров `RemoveAttributes` и `RemoveNodes`, удаляющие соответственно все атрибуты и все дочерние узлы того элемента, для которого эти методы вызваны (метод `RemoveNodes` имеется и в классе `XDocument`). Таким образом, вызов `e.RemoveAttributes()` (для некоторого элемента `e`) равносителен вызову цепочки методов `e.Attributes().Remove()`, а вызов `e.RemoveNodes()` равносителен вызову `e.Nodes().Remove()`. В классе `XElement` предусмотрен также метод `RemoveAll`, удаляющий и атрибуты, и дочерние узлы элемента.

Обратимся к заданию LinqXml28. В нем требуется удалить *текстовые* дочерние узлы (то есть узлы типа *XText*) для всех элементов третьего уровня. Если считать, что исходный документ связан с переменной *d*, то для получения последовательности элементов третьего уровня достаточно использовать следующую цепочку методов:

```
d.Root.Elements().Elements().Elements()
```

К полученной последовательности элементов можно сразу применить метод *Nodes*; в результате мы получим последовательность всех дочерних узлов элементов третьего уровня:

```
d.Root.Elements().Elements().Elements().Nodes()
```

Чтобы отобрать из данной последовательности только текстовые узлы, можно было бы применить к ней запрос *Where(e => e is XText)*, однако возможен более наглядный способ, основанный на применении метода *OfType*:

```
d.Root.Elements().Elements().Elements().Nodes().OfType<XText>()
```

Метод *OfType<T>* относится к категории *методов импортирования*; он включает в выходную последовательность только те элементы исходной последовательности, для которых может быть выполнена операция приведения к типу *T* (иными словами, включаются те элементы, фактический тип которых либо совпадает с *T*, либо унаследован от типа *T*). Полученная последовательность имеет тип *IEnumerable<T>*.

Для удаления требуемых узлов достаточно применить к полученной последовательности метод *Remove*. Получаем первый вариант решения:

```
public static void Solve()
{
    Task("LinqXml28");
    string name = GetString();
    XDocument d = XDocument.Load(name);
    d.Root.Elements().Elements().Elements()
        .Nodes().OfType<XText>().Remove();
    d.Save(name);
}
```

Если в результате выполнения метода *Remove* у элемента с именем *a* удаляются все его дочерние узлы, то этот элемент представ-

ляется в виде *комбинированного тега* `<a />`. Таким образом, при использовании метода `Remove` дополнительное условие задачи `LinqXml28` выполняется автоматически.

Обрабатываемая в задаче последовательность дочерних узлов всех элементов третьего уровня может интерпретироваться как последовательность *всех узлов четвертого уровня*. Для отбора всех таких узлов `e` можно использовать метод `Where` с условием `e.Ancestors().Count() == 4` (поскольку только у узлов четвертого уровня имеется ровно четыре предка). Условия такого вида удобны, если требуется выбрать элементы с большой глубиной вложенности или элементы из некоторого *диапозона* уровней, например все элементы уровней 2, 3 и 4. В метод `Where` можно сразу включить и условие отбора текстовых узлов `e is XText`. Приведем соответствующий вариант решения, используя в нем выражение запроса:

```
public static void Solve()
{
    Task("LinqXml28");
    string name = GetString();
    XDocument d = XDocument.Load(name);
    (from e in d.DescendantNodes()
     where e is XText && e.Ancestors().Count() == 4
     select e).Remove();
    d.Save(name);
}
```

Теперь рассмотрим задачу, в которой требуется добавить в XML-документ новые компоненты.

LinqXml32. Даны XML-документ и строки S_1 и S_2 . В строке S_1 записано имя одного из элементов исходного документа, строка S_2 содержит допустимое имя элемента XML. Перед каждым элементом второго уровня с именем S_1 добавить элемент с именем S_2 . Добавленный элемент должен содержать последний атрибут и первый дочерний элемент последующего элемента (если они есть). Если элемент S_1 не имеет дочерних элементов, то добавленный перед ним элемент S_2 должен быть представлен в виде *комбинированного тега*.

Указание. Использовать метод `FirstOrDefault`.

Для добавления новых компонентов в XML-документ, как и для их удаления, в модели X-DOM предусмотрено несколько методов.

В классе `XNode` имеются методы `AddAfterSelf` и `AddBeforeSelf`, позволяющие добавить один или более узлов после или, соответственно, перед тем узлом, для которого вызваны эти методы (добавленные узлы будут иметь тот же уровень, что и узел, вызвавший метод). В классе `XElement` (и в классе `XDocument`) имеются методы `Add` и `AddFirst`, позволяющие добавить один или более узлов в конец или, соответственно, в начало набора дочерних узлов того элемента (XML-документа), для которого были вызваны методы. В случае класса `XElement` метод `Add` можно использовать и для добавления новых атрибутов (атрибуты добавляются в конец списка атрибутов).

Важной особенностью всех перечисленных методов является то, что в качестве их параметров можно указывать не только объекты типа `XNode`, но и *данные других типов*, в том числе числа, строки, значения даты и времени (типа `DateTime` и `TimeSpan`); все эти данные будут автоматически преобразованы к *строковым узлам* `XText`. Кроме того, параметры могут быть *последовательностями*; в этом случае методы обрабатывают все элементы этих последовательностей (по указанным выше правилам).

В задании `LinqXml32` требуется добавить новый элемент перед каждым элементом второго уровня с именем `s1`. Получить последовательность требуемых элементов можно с помощью следующей цепочки методов (предполагая, что XML-документ связан с переменной `d`):

```
d.Root.Elements().Elements(s1)
```

Указание строки `s1` в качестве параметра последнего метода `Elements` обеспечивает отбор только тех элементов, которые имеют данное имя. Заметим, что при вызове последнего метода выполняется неявное преобразование параметра `s1` (типа `string`) к типу `XName`.

Для перебора отобранных элементов надо использовать цикл `foreach`:

```
foreach (var e in d.Root.Elements().Elements(s1))
{
    // обработка элементов e
}
```

Для добавления элемента с именем `s2` перед элементом `e` воспользуемся методом `AddBeforeSelf`:

```
e.AddBeforeSelf(new XElement(s2, ...));
```

В позицию, отмеченную многоточием, можно добавлять дочерние узлы и атрибуты для созданного элемента (см. п. 7.1). В нашем случае элемент должен содержать *последний* атрибут и *первый* дочерний элемент элемента *e* (если они есть).

Для доступа к последнему атрибуту элемента *e* достаточно воспользоваться свойством `LastAttribute` (имеется также парное свойство `FirstAttribute`, обеспечивающее доступ к первому атрибуту). Если элемент не имеет атрибутов, то оба эти свойства возвращают `null`. Подобное поведение нас устраивает, поскольку конструктор класса `XElement` не обрабатывает параметры, равные `null` (за исключением первого параметра, определяющего имя элемента, которое не может быть равно `null`).

Примечание. Первый параметр конструкторов `XAttribute` и `XElement`, определяющий имя создаваемого атрибута или элемента, должен удовлетворять дополнительным условиям, налагаемым на имена атрибутов и элементов XML: имена должны быть непустыми, содержать только цифровые и буквенные (не обязательно латинские) символы, а также символы «.» (точка), «-» (дефис) и «_» (подчеркивание). Имена могут начинаться только с буквенных символов или подчеркивания. Следует также учитывать, что имена XML чувствительны к регистру.

Для доступа к первому дочернему элементу специального свойства не предусмотрено (хотя у классов `XDocument` и `XElement` имеются свойства для доступа к первому и последнему *дочернему узлу*: `FirstNode` и `LastNode`). Однако мы можем воспользоваться методом `Elements`, возвращающим последовательность дочерних элементов, и извлечь из этой последовательности первый элемент. Следует лишь учесть, что последовательность может оказаться пустой; в этом случае желательно возвращать значение `null`. Подобное поведение обеспечивает метод `FirstOrDefault` (мы уже использовали этот метод при решении задач – см. п. 6.3).

Таким образом, конструктор нового элемента должен иметь вид:

```
new XElement(s2, e.LastAttribute, e.Elements().FirstOrDefault())
```

Если в конструкторе элемента отсутствуют параметры, связанные с дочерними узлами, то этот элемент представляется в виде комбинированного тега (как и требуется в условии задачи). Если по каким-либо причинам желательно, чтобы элемент имел вид парного тега с пустым содержимым (например, `<a>`), то в конструкторе,

кроме имени элемента, достаточно указать дополнительный параметр, являющийся *пустой строкой* "".

Объединив приведенные выше фрагменты и дополнив их операторами, обеспечивающими ввод исходных данных и сохранение измененного документа, получаем следующий вариант правильного решения задачи:

```
public static void Solve()
{
    Task("LinqXml32");
    string name = GetString(), s1 = GetString(), s2 = GetString();
    XDocument d = XDocument.Load(name);
    foreach (var e in d.Root.Elements().Elements(s1))
        e.AddBeforeSelf(new XElement(s2,
            e.LastAttribute, e.Elements().FirstOrDefault()));
    d.Save(name);
}
```

При преобразовании XML-документа часто приходится изменять *значения* атрибутов и узлов. В модели X-DOM для этих целей предусмотрен набор удобных свойств и методов, которые мы обсудим, решая следующую задачу.

LinqXml37. Дан XML-документ. Для каждого элемента второго уровня, имеющего потомков, добавить к его текстовому содержимому текстовое содержимое всех элементов-потомков, после чего удалить все его узлы-потомки, кроме дочернего текстового узла.

Указание. Использовать свойство Value класса XElement.

Упомянутое в указании к задаче свойство Value типа string имеется у класса XAttribute и у некоторых классов, связанных с узлами XML, в том числе XText, XComment, XElement. Для атрибута оно возвращает строковое значение этого атрибута, для текстового узла и комментария – значения этих узлов. Для элемента XML свойство Value возвращает текстовое содержимое этого элемента, *дополненное текстовым содержимым всех его элементов-потомков*, взятым в порядке следования этих элементов.

Свойство Value доступно для изменения. Для атрибутов и комментариев изменение этого свойства приводит к изменению их значений. Для элемента XML изменение свойства Value приводит к тому, что *все дочерние узлы данного элемента заменяются на единственный дочерний текстовый узел*, значение которого будет совпадать с новым значением свойства Value.

Таким образом, благодаря «сложному» поведению свойства Value для элементов XML требуемое в задаче LinqXml37 преобразование элемента *e* может быть выполнено с помощью *единственной* операции присваивания, имеющей, правда, несколько странный вид:

```
e.Value = e.Value;
```

Хотя левая и правая части в указанной операции присваивания совпадают, эта операция приведет к *изменению* элемента *e*, поскольку фактически она будет заменена на вызов специального метода, связанного со свойством Value и выполняющего действия, описанные выше.

Осталось обсудить, как получить последовательность элементов, удовлетворяющих условиям задачи, то есть элементов второго уровня, имеющих потомков. Для отбора элементов второго уровня достаточно вызвать цепочку из двух методов Elements (подобные действия мы уже выполняли при решении двух предыдущих задач). Чтобы проверить, имеет ли элемент XML узлы-потомки, можно использовать свойство IsEmpty класса XElement. Данный метод возвращает true, если элемент представлен комбинированным тегом, и false в противном случае, то есть когда он представлен парным тегом. Если элемент имеет вид «пустого» парного тега (вида <a>), то считается, что он содержит дочерний текстовый узел XText, имеющий значение пустой строки "". Заметим, что свойство Value возвращает пустую строку и для элемента, представленного комбинированным тегом, и для элемента, представленного пустым парным тегом, поэтому условие *e.Value != ""* *не следует* использовать для проверки того, содержит ли элемент узлы-потомки (поскольку этому условию не удовлетворяют элементы, содержащие *единственный пустой дочерний текстовый узел*).

В классе XElement имеются также свойства, позволяющие проверить, содержит ли элемент *атрибуты* (свойство HasAttributes) или дочерние *элементы* (свойство HasElements). Разумеется, все подобные свойства можно реализовать с помощью обработки соответствующих последовательностей, вызвав для них метод Any (данный метод LINQ наряду с прочими методами-квантификаторами описан в примечании в п. 6.3). Например, для проверки того, имеет ли элемент *e* дочерние узлы, достаточно проанализировать значение выражения *e.Nodes().Any()*.

Применяя описанные выше средства X-DOM, получаем следующее решение задачи LinqXml37 (в этом решении мы использовали выражение запроса):

```

public static void Solve()
{
    Task("LinqXml37");
    string name = GetString();
    XDocument d = XDocument.Load(name);
    var a = from e in d.Root.Elements().Elements()
            where !e.IsEmpty
            select e;
    foreach (var e in a)
        e.Value = e.Value;
    d.Save(name);
}

```

Приведем пример выполнения данной программы. На рис. 45 указан раздел исходных данных с начальной частью XML-документа, а на рис. 46 – раздел результатов с преобразованным XML-документом.

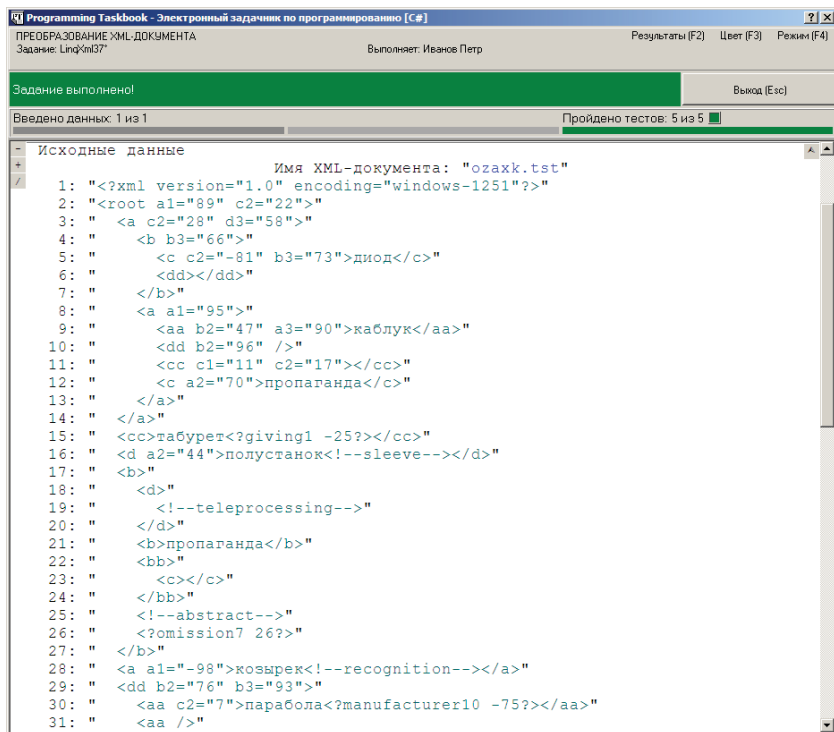


Рис. 45. Успешное выполнение задания LinqXml37
(раздел исходных данных)

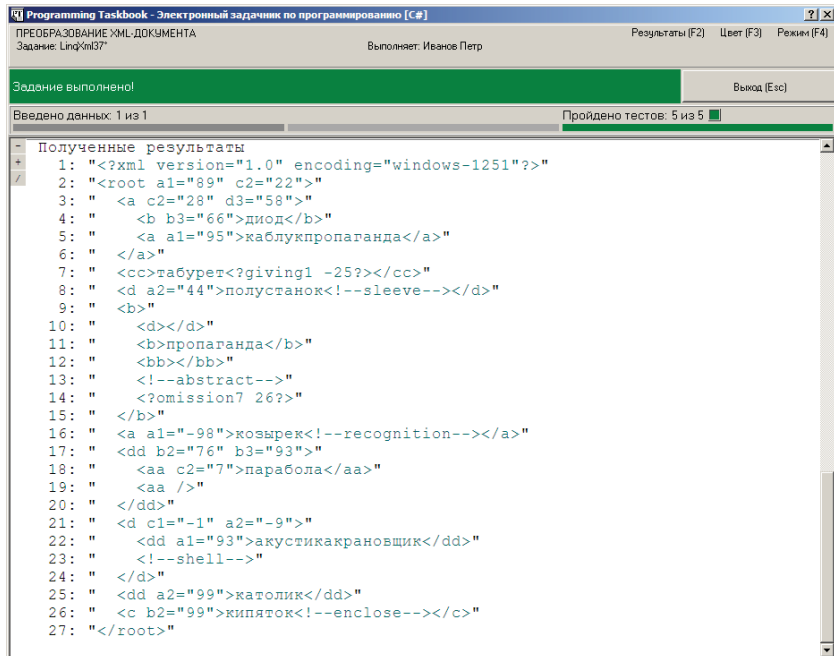


Рис. 46. Успешное выполнение задания LinqXml37
(раздел результатов)

Первый элемент второго уровня в исходном документе имеет имя **b**; он содержит два дочерних элемента: **c** с текстовым значением «диод» и **dd** с пустым текстовым значением. В результате преобразования элемент **b** получает единственный текстовый узел «диод» (его атрибут **b3** не изменяется).

Второй элемент второго уровня (**a**) имеет четыре дочерних элемента, среди которых имеются элемент, представленный комбинированным тегом, элемент, представленный парным пустым тегом, и два элемента со значениями «каблук» и «пропаганда». В результате элемент **a** получает «объединенное» текстовое значение «каблукпропаганда».

Элемент второго уровня **d**, начинающийся в строке номер 18 на рис. 45, имеет единственный дочерний узел-комментарий. Поскольку он содержит дочерние узлы, для него выполняется требуемое преобразование, однако среди его потомков нет ни одного элемента XML, поэтому в результате он получает *пустое* текстовое значение

и отображается в виде пустого парного тега (см. строку номер 10 на рис. 46).

Тот же самый результат получается и для элемента **bb**, начинающегося в строке 22 на рис. 45: хотя он и содержит дочерний элемент, текстовое значение этого элемента является пустым, поэтому элемент **bb** преобразуется в пустой парный тег (см. строку 12 на рис. 46).

Следует отметить, что узлы второго уровня, не являющиеся элементами XML, например комментарий **abstract** (строка 25 на рис. 45) или инструкция обработки **omission7** (строка 27), в полученном документе не изменяются. Не изменяется и элемент **a**, представленный комбинированным тегом (строка 31).

Примечание. При форматировании XML-документа в него добавляются дополнительные пробелы, однако при последующем чтении отформатированного документа из файла эти пробелы игнорируются. Более точно: если между соседними тегами различных элементов (не важно, открывающими или закрывающими) располагаются только пробелы и символы конца строк, то все они считаются незначащими и игнорируются. Именно поэтому, например, элемент **b**, приведенный на рис. 45 в строках 4–7, будет содержать только два дочерних элемента (**c** и **dd**) и ни одного дочернего текстового узла, хотя между его тегами и тегами его дочерних элементов содержится ряд дополнительных пробелов. Если пробелы располагаются между открывающим и закрывающим тегами одного и того же элемента, то считается, что этот элемент содержит единственный пустой текстовый узел. Однако если в пространстве между тегами содержится хотя бы один значащий символ, то весь текст между тегами, *включая пробелы*, считается текстовым узлом. Например, при чтении из файла элемента, представленного в виде `<a> x ` (слева и справа от символа «**x**» расположено по одному пробелу), будет сформирован элемент **a** с текстовым содержимым " **x** ", включающим начальный и конечный пробелы.

Если при загрузке XML-документа требуется учесть все незначащие пробелы, то в методе **Load** надо указать второй, дополнительный параметр **LoadOptions.PreserveWhiteSpace**. Для учета незначащих пробелов при последующем сохранении XML-документа надо отключить режим автоматического форматирования, использовав метод **Save** с дополнительным параметром **SaveOptions.DisableFormatting**.

Завершая обсуждение приемов преобразования XML-документов, упомянем еще несколько методов, которые могут оказаться полезными при выполнении подобных преобразований.



Наряду с различными вариантами метода Remove, обеспечивающими *удаление* узла из документа или дочерних узлов или атрибутов из некоторого элемента, в классах XNode и XElement предусмотрен набор методов, начинающихся со слова «Replace» и позволяющих *заменить* отдельные компоненты XML-документа другими.

Класс XNode содержит метод ReplaceWith, позволяющий заменить узел, вызвавший данный метод, на один или несколько узлов, указанных в качестве параметра метода.

Класс XElement дополнительно содержит методы ReplaceAttributes, ReplaceNodes и ReplaceAll, которые заменяют соответственно атрибуты, дочерние узлы и все дочерние компоненты (атрибуты и узлы) на набор атрибутов и узлов, указанных в качестве параметров. Подчеркнем, что в любом из этих трех методов можно указывать как параметры-атрибуты, так и параметры-узлы, причем, как в случае ранее описанных методов Add, параметрами могут быть и базовые типы (числа, строки, даты, промежутки времени), которые автоматически преобразуются в текстовые узлы.

Заметим, что метод ReplaceNodes реализован и в классе XDocument.

Варианты методов Replace, в отличие от вариантов метода Remove, не могут вызываться для последовательностей элементов, поскольку в подобных вариантах методов было бы непонятно, в какую часть документа XML следует помещать добавляемые данные.

В процессе преобразования документа XML может потребоваться изменить *имена* содержащихся в нем элементов или атрибутов (напомним, что из всех видов узлов только элементы XML имеют имена). Для изменения имени элемента XML, то есть объекта типа XElement, достаточно изменить его свойство Name. Одноименное свойство атрибута, то есть объекта типа XAttribute, доступно только для чтения, поэтому для изменения имени атрибута придется удалить исходный атрибут и создать новый, с новым именем и прежним содержимым (см. задачу LinqXml40).

Удобные и гибкие средства преобразования значений атрибутов и элементов предоставляет набор методов SetValue. Метод SetValue имеется в классе XAttribute и XElement и позволяет задать значение атрибута и элемента соответственно, причем в качестве параметра можно указывать не только строки, но и данные других типов (числа, даты, промежутки времени), которые будут автоматически преобразованы в их строковые представления *с учетом правил XML* (например, для вещественных чисел всегда будет использоваться точка в качестве разделителя).

В классе `XElement` имеются также методы `SetAttributeValue` и `SetElementValue` с двумя параметрами – именем атрибута/элемента и его новым значением (которое может представляться не только строкой, но и данными других типов). Если элемент содержит несколько дочерних элементов с одинаковыми именами, то метод `SetElementValue` действует на первый из таких элементов (напомним, что атрибутов с одинаковыми именами элемент содержать не может). Интересно отметить, что эти методы могут использоваться не только для изменения, но также и для создания или удаления атрибутов или дочерних элементов: если атрибут или элемент с указанным именем отсутствует, то он создается (и добавляется в конец списка атрибутов или, соответственно, дочерних узлов); если для существующего атрибута/элемента в качестве нового значения указывается `null`, то он удаляется (если значение `null` указано для несуществующего атрибута/элемента, то метод не выполняет никаких действий).

7.4. Преобразование типов при обработке XML-документа: LinqXml50

Во многих XML-документах значения атрибутов и элементов связываются не с текстовыми данными, а с данными других типов: логическими значениями, числами, датами, промежутками времени. Строковые представления подобных данных, содержащиеся в XML-документах, должны удовлетворять специальным правилам, определенным в стандарте XML (ранее мы уже упоминали об одном таком правиле: независимо от региональных настроек операционной системы для представления вещественных чисел в XML-документе должен использоваться десятичный разделитель *точка*).

В модели X-DOM предусмотрен ряд средств, облегчающих обработку атрибутов и элементов, значения которых связаны с данными специального типа. Этим средствам посвящены задачи, объединенные в особую подгруппу группы LinqXml (см. п. 4.4) и связанные с обработкой в XML-документе вещественных чисел, логических значений, промежутков времени и дат. В качестве примера рассмотрим одну из задач, связанную с обработкой промежутков времени.

LinqXml50. Дан XML-документ. С каждым элементом документа связывается некоторый промежуток времени (в днях,

часах, минутах и секундах). Этот промежуток либо явно указывается в атрибуте `time` данного элемента (в формате, принятом в стандарте XML), либо, если данный атрибут отсутствует, считается равным одному дню. Добавить в начало набора дочерних узлов корневого элемента элемент `total-time` со значением, равным суммарному значению промежутков времени, связанных со всеми элементами первого уровня.

Указание. Использовать приведение объекта `XAttribute` к `Nullable-типу TimeSpan?` и операцию `??`.

На рис. 47 приведено окно задачника, которое будет отображено на экране при запуске программы-заготовки для данной задачи (для уменьшения размеров окна в нем скрыт раздел с формулировкой).

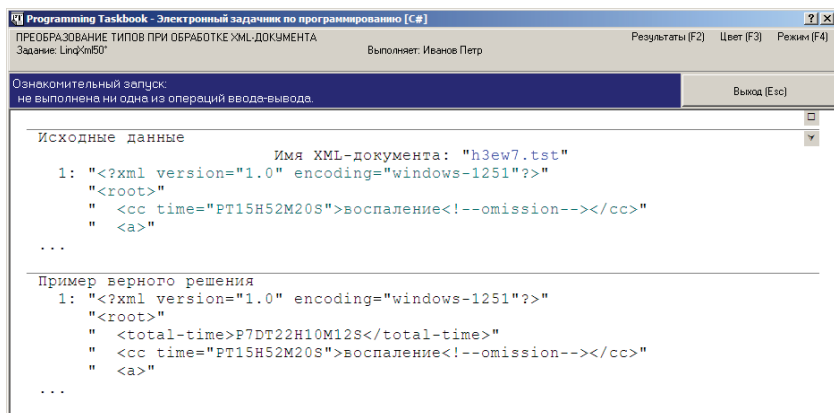


Рис. 47. Ознакомительный запуск задания LinqXml50

Анализируя фрагменты XML-документов, приведенные на рисунке, можно заметить, что для представления промежутков времени используется достаточно сложный формат. Однако нам не требуется работать непосредственно с этим форматом; достаточно использовать объекты типа `TimeSpan` из стандартной библиотеки .NET, поскольку все необходимые преобразования объектов этого типа выполняются «внутри» соответствующих методов X-DOM.

Наиболее просто обеспечивается задание значений специальных типов; для этого следует использовать методы `SetValue`, `SetAttributeValue` и `SetElementValue`, описанные в конце предыдущего пунк-

та, или указать требуемые значения в конструкторе элемента или атрибута (особенности конструкторов объектов X-DOM были рассмотрены в п. 7.1). В частности, если в качестве значения указать объект типа `TimeSpan`, то он будет автоматически преобразован к строковому представлению, удовлетворяющему стандарту XML. Заметим, что вызов метода `ToString` для объекта `TimeSpan` даст *неверное* (с точки зрения стандарта XML) строковое представление промежутка времени.

Получение значений атрибутов и элементов XML, связанных со специальными типами данных, реализовано с помощью явного приведения самого атрибута или элемента (то есть объекта класса `XAttribute` или `XElement`) к требуемому типу данных. Следует подчеркнуть, что приведению должен подвергаться атрибут или элемент в целом, а не его значение (доступное с помощью строкового свойства `Value`). Например, если известно, что атрибут `a1` имеет значение, определяющее некоторый промежуток времени (и заданное в соответствии со стандартом XML), то для получения этого значения в виде объекта `TimeSpan` достаточно использовать выражение `(TimeSpan)a1`. Однако при попытке использовать выражение `(TimeSpan)a1.Value` компилятор выведет сообщение о том, что нельзя преобразовать тип `string` к типу `TimeSpan`.

Если значение атрибута или элемента не удовлетворяет стандарту XML, принятому для представления требуемого типа данных, то в результате попытки явного приведения к этому типу будет возбуждено исключение `FormatException`.

При обработке XML-документов может оказаться, что атрибут или элемент с требуемым именем `name` отсутствует. В этом случае при попытке использования методов `Attribute(name)` или `Element(name)` будет возвращено значение `null`, для которого, разумеется, нельзя вызывать никакие методы или свойства, в том числе и `Value`. Однако к нему *можно применить операцию приведения типа*; требуется лишь, чтобы этот тип включал значение `null`. Значение `null` могут принимать любые ссылочные типы, в частности строковый тип `string`. Поэтому, например, выражение `(string)e.Attribute(name)` не приведет к возбуждению исключения даже при отсутствии у элемента `e` атрибута с именем `name`, а при наличии этого атрибута вернет его строковое значение (при отсутствии атрибута будет возвращено значение `null`).

Аналогичный прием может быть использован и при обработке элементов или атрибутов, содержащих числовые значения или зна-

чения других *размерных типов* (несмотря на то что размерные типы не могут принимать значение null). В данном случае можно применять так называемые *Nullable-типы* – расширения размерных типов, включающие значение null. Имя Nullable-типа может быть получено из имени соответствующего размерного типа добавлением символа «?», например `int?`. Таким образом, получить либо целочисленное значение атрибута с именем `name`, либо null, если такого атрибута не существует, можно с помощью выражения `(int?)e.Attribute(name)` (если атрибут `name` существует, но содержит значение, которое нельзя интерпретировать как целое число, то при обработке данного выражения будет возбуждено исключение `FormatException`).

Часто в ситуации, когда требуемый атрибут или элемент отсутствует, надо использовать специальное *значение по умолчанию*. Соответствующее выражение можно представить в наиболее кратком виде, если использовать операцию `??` (см. п. 5.1.2). Например, если для элемента `e` требуется получить значение строкового атрибута с именем `name` или пустую строку, при условии отсутствия атрибута с таким именем, то достаточно воспользоваться следующим выражением:

```
(string)e.Attribute(name) ?? ""
```

Применим полученные сведения для решения задачи LinqXml50.

Будем считать, что исходный XML-документ связан с переменной `d`.

На первом этапе решения найдем суммарный промежуток времени, связанный со всеми элементами первого уровня. Для отбора всех элементов первого уровня достаточно использовать выражение

```
d.Root.Elements()
```

Для каждого элемента `e` надо определить связанный с ним промежуток времени. При этом необходимо учитывать, что если элемент не содержит атрибута `time`, то с ним связывается промежуток времени, равный одному дню:

```
d.Root.Elements()  
    .Select(e => (TimeSpan?)e.Attribute("time") ??  
        new TimeSpan(24, 0, 0))
```

Поскольку тип `TimeSpan` является *структурой*, то есть размерным, а не ссылочным типом, нам пришлось использовать его

Nullable-вариант. Для получения промежутка, равного одному дню, мы использовали конструктор структуры TimeSpan с тремя параметрами, определяющими количество часов, минут и секунд.

Полученные промежутки времени надо просуммировать. Методом агрегирования Sum в данном случае воспользоваться нельзя, поскольку этот метод позволяет суммировать только *числовые значения*. Однако мы можем использовать «универсальный» метод агрегирования Aggregate (см. п. 5.2), указав в его лямбда-выражении операцию «+» для объектов типа TimeSpan:

```
d.Root.Elements()
    .Select(e => (TimeSpan?)e.Attribute("time") ??
        new TimeSpan(24, 0, 0))
    .Aggregate(TimeSpan.Zero, (a, e) => a + e)
```

Напомним, что первый параметр метода Aggregate определяет начальное значение переменной-*аккумулятора* (в этой переменной накапливается вычисляемое выражение), а второй параметр является лямбда-выражением, определяющим способ изменения аккумулятора *a* при обработке очередного элемента *e* исходной последовательности. В качестве начального значения аккумулятора мы использовали статическое свойство Zero структуры TimeSpan, равное нулевому промежутку времени.

Типы параметров лямбда-выражения выводятся компилятором из типов аккумулятора и элементов обрабатываемой последовательности; таким образом, оба параметра имеют тип TimeSpan, и для них определена операция «+», возвращающая суммарный промежуток времени.

На данном этапе решения задачи мы уже можем проверить правильность нахождения суммарного промежутка времени, выведя его в разделе отладки. Приведем соответствующий вариант функции Solve:

```
public static void Solve()
{
    Task("LinqXml50");
    string name = GetString();
    XDocument d = XDocument.Load(name);
    TimeSpan t = d.Root.Elements()
        .Select(e => (TimeSpan?)e.Attribute("time") ??
```

```
        new TimeSpan(24, 0, 0))  
        .Aggregate(TimeSpan.Zero, (a, e) => a + e);  
    Show(t);  
}
```

При запуске программы мы, разумеется, получим сообщение об ошибке (поскольку исходный XML-документ не изменился), однако выведенное в разделе отладки значение промежутка времени должно совпасть со значением, указанным в элементе `total-time` XML-документа, приведенного в разделе с примером правильного решения (см. рис. 48). Следует обратить внимание на различия в способах представления промежутков времени (в разделе отладки приведено строковое представление, возвращаемое методом `To-String` структуры `TimeSpan`).

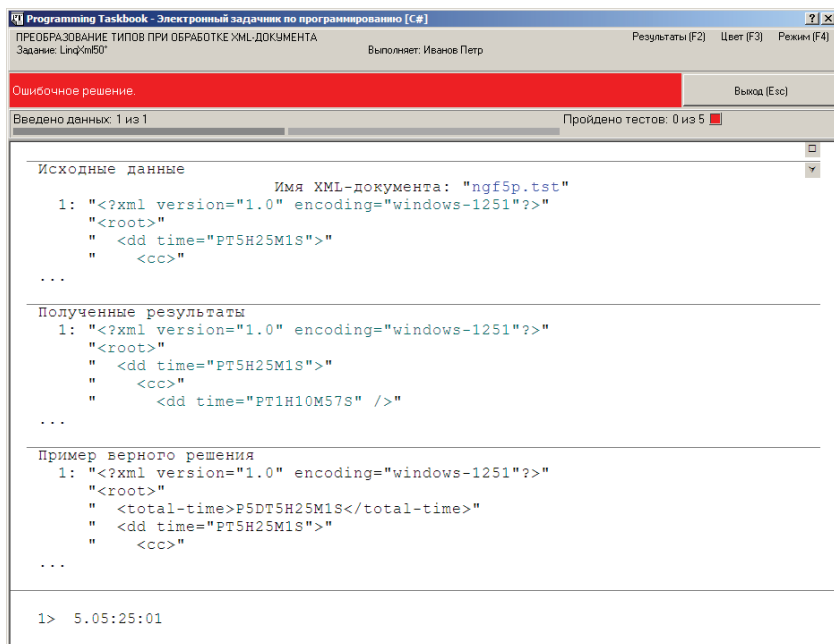


Рис. 48. Первый этап выполнения задания LinqXml50

Обратимся ко второму этапу решения, на котором нам требуется добавить в начало набора дочерних узлов корневого элемента исходного XML-документа элемент с именем `total-time` и значением, равным значению найденного промежутка времени `t`, после чего сохранить измененный XML-документ под тем же именем.

Этот этап является существенно более простым, чем рассмотренный ранее. Для добавления нового элемента достаточно воспользоваться методом `AddFirst` (см. п. 7.3); в конструкторе добавляемого элемента следует указать переменную `t` без каких-либо преобразований, поскольку все требуемые преобразования будут выполнены автоматически:

```
d.Root.AddFirst(new XElement("total-time", t));
d.Save(name);
```

Заметим, что в программе, решающей задачу, можно обойтись без вспомогательной переменной `t`, поместив выражение, вычисляющее суммарный промежуток времени, непосредственно в конструктор элемента `total-time`. В результате получим следующий вариант правильного решения:

```
public static void Solve()
{
    Task("LinqXml50");
    string name = GetString();
    XDocument d = XDocument.Load(name);
    d.Root.AddFirst(new XElement("total-time",
        d.Root.Elements()
            .Select(e => (TimeSpan?)e.Attribute("time") ??
                new TimeSpan(24, 0, 0))
            .Aggregate(TimeSpan.Zero, (a, e) => a + e)));
    d.Save(name);
}
```

7.5. Работа с пространствами имен XML-документа: LinqXml57

Отдельная подгруппа группы LinqXml посвящена средствам X-DOM, связанным с *пространствами имен* (англ. namespace) XML-документа (см. п. 4.5). Возможность дополнять имена элементов и атрибутов пространствами имен введена в XML для того, чтобы обеспечить уникальность используемых имен, а также «при-

вязать» эти имена к организации, разработавшей спецификацию для соответствующей категории XML-документов; по этой причине в качестве пространств имен обычно используются *универсальные идентификаторы ресурсов* Интернета (Uniform Resource Identifier – URI). Благодаря указанной привязке имени компонентов XML приобретают «абсолютный» смысл, обеспечивающий их однозначную интерпретацию. Например, имя XML-элемента **author** может встретиться во многих XML-документах, однако это имя, дополненное пространством имен `http://www.gribuser.ru/xml/fictionbook/2.0`, однозначно указывает, что имеется в виду имя элемента, определяющего автора литературного произведения в XML-документе, соответствующем спецификации «Fiction Book 2.0» (формат «Fiction Book 2.0» в настоящее время является одним из популярных форматов хранения электронных книг; файлы в этом формате имеют расширение fb2).

Компоненты XML могут иметь *пустое пространство имен*; так, в частности, все компоненты в XML-документах из задач первых четырех подгрупп группы LinqXml (LinqXml1–LinqXml52). Однако в «реальных» XML-документах, подготовленных в соответствии с определенными правилами, подобная ситуация маловероятна.

Основные приемы определения и корректировки пространств имен мы обсудим на примере задачи LinqXml57.

LinqXml57. Даны XML-документ и строки S_1 и S_2 , содержащие различные пространства имен. Удалить в документе определения исходных пространств имен и определить в корневом элементе два префикса пространств имен: префикс x , связанный с S_1 , и префикс y , связанный с S_2 . Снабдить префиксом x элементы нулевого и первого уровней, а префиксом y – элементы последующих уровней.

Данная задача посвящена специальному средству задания пространств имен – *префиксам пространств имен*, однако анализ связанного с ней XML-документа позволит познакомиться и с более простым способом задания пространств имен, основанном на применении атрибута `xmlns`.

Прежде чем обсуждать средства X-DOM, связанные с пространствами имен, полезно рассмотреть примеры использования пространств имен в XML-документах. Источником примеров нам послужит сама задача LinqXml57, а именно образцы ее исходных и результирующих XML-документов. Чтобы проиллюстрировать не-

которые особенности, связанные с пространствами имен, дополним программу-заготовку, созданную для задания LinqXml57, фрагментом, выводящим в раздел отладки имена всех элементов исходного XML-документа. Перед этим, разумеется, необходимо прочесть исходный документ из файла. Организуем также считывание строк s1 и s2, связанных с пространствами имен, хотя в программе они пока не будут использоваться:

```
public static void Solve()
{
    Task("LinqXml57");
    string name = GetString(), s1 = GetString(), s2 = GetString();
    XDocument d = XDocument.Load(name);
    d.Descendants().Show(e => e.Name + "\n");
}
```

Чтобы вывести в раздел отладки имена всех элементов, мы воспользовались методом Descendants для исходного документа d. К полученной последовательности элементов XML мы применили метод расширения Show с параметром – лямбда-выражением, которое определяет, какую именно информацию о каждом элементе последовательности требуется вывести на экран. Благодаря слагаемому "\n" каждое имя будет выводиться на новой экранной строке.

При запуске полученной программы на экране появится окно загрузчика с сообщением об ошибочном решении (поскольку исходный XML-документ не изменился). После прокрутки содержимого окна к разделу полученных результатов вид окна будет подобен приведенному на рис. 49.

Данный рисунок является хорошей иллюстрацией различных аспектов использования пространств имен.

Прежде всего отметим, что в разделе полученных результатов приводится *исходный* XML-документ, так как содержащий его файл не был изменен нашей программой. Таким образом, информация, выведенная в разделе отладки, соответствует документу из раздела результатов. Мы видим, что первый элемент (root) не связан с каким-либо пространством имен; его имя в разделе результатов в точности соответствует тексту его открывающего тега. Следующий элемент (d) содержит атрибут xmlns. Этот атрибут определяет пространство имен для данного элемента и *всех его элементов-потомков*, если в них не определен атрибут xmlns с новым значением. В частности, дочерний элемент ss, не имеющий атрибута xmlns, получает пространство имен своего родителя, а элемент dd (третьего уровня)

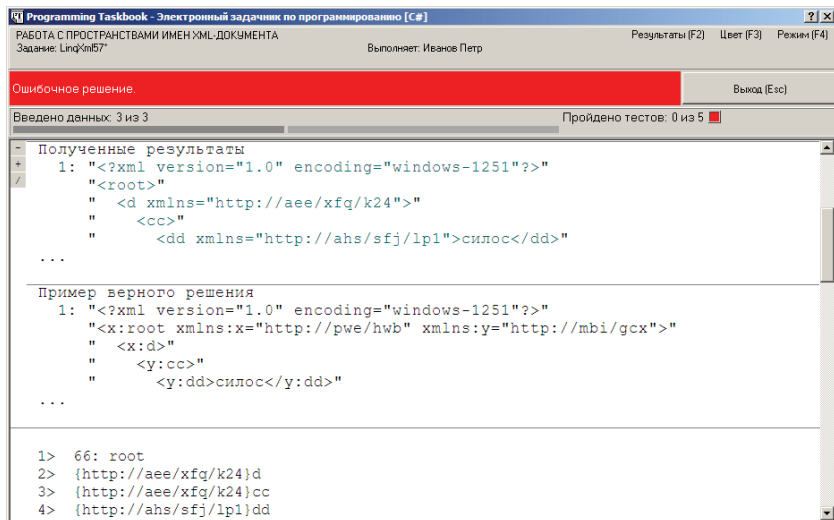


Рис. 49. Отладочная печать имен исходного XML-документа в задании LinqXml57

переопределяет атрибут `xmlns` и благодаря этому получает новое пространство имен. Вся эта информация подтверждается данными, выведенными в разделе отладки. Текст этого раздела показывает, что если элемент имеет непустое пространство имен, то оно заключается в фигурные скобки `{ }` и приписывается слева к локальному имени элемента. В то же время *теги* исходного XML-документа, связанные с элементами, содержат только их локальные имена (пространства имен в тегах не указываются).

Теперь обратимся к разделу с примером верного решения. В этом разделе содержится XML-документ, в котором для задания пространств имен применяются не атрибуты `xmlns`, а так называемые *префиксы пространства имен*. Для определения префикса пространства имен необходимо использовать специальный атрибут, начинающийся с префикса `xmlns`, после которого указываются двоеточие и имя определяемого префикса. Значение префикса задается значением атрибута. В документе из раздела с примером верного решения определены два префикса пространства имен с именами `x` и `y`. Оба они определены в корневом элементе и поэтому доступны для всех элементов документа (префиксы доступны в том элементе, в котором они определены, и во всех его потомках).

В отличие от определения пространства имен, рассмотренного нами на примере исходного документа, определение префикса пространства имен *не приводит* к автоматическому присоединению данного пространства имен к имени того элемента, в котором префикс определен. Для подобного присоединения необходимо явно указать имя префикса перед локальным именем элемента в его тегах, отделив его от локального имени двоеточием. В приведенном на рис. 49 примере префикс `x` связывается с элементами нулевого (`root`) и первого (`d`) уровней, а префикс `y` – с элементами более высоких уровней (`ss` и `dd`). На примере элемента `dd` видно, что префикс должен указываться и в открывающем, и в закрывающем теге элемента. Префиксам пространств имен посвящены задания LinqXml57–LinqXml60.

Еще одна особенность способа задания пространства имен с помощью префиксов состоит в том, что пространство имен, заданное посредством префикса, *не распространяется* по умолчанию на элементы-потомки. Так, если в нашем примере для элемента `dd` префикс `y` не был бы указан явным образом, то элемент `dd` имел бы пустое пространство имен.

Пространства имен можно задавать не только для элементов XML, но и для их атрибутов. При этом атрибут можно снабдить пространством имен *только с помощью префикса*. Примерами таких атрибутов на рис. 49 являются сами атрибуты задания префикса: `xmlns:x` и `xmlns:y`; в данном случае префикс `xmlns` предварительно определять не требуется, так как он является стандартным префиксом XML. Префиксам атрибутов посвящены задания LinqXml58–LinqXml59.

Примечание. Некоторую путаницу порождает то обстоятельство, что и определение пространства имен, и определение префикса пространства имен начинается с одного и того же идентификатора `xmlns` (являющегося аббревиатурой выражения XML namespace, набранной строчными буквами). При определении пространства имен идентификатор `xmlns` является именем стандартного атрибута, в то время как при определении префикса пространства имен идентификатор `xmlns` является именем стандартного префикса, который должен указываться перед именем атрибута (совпадающего с именем определяемого префикса). Итак, если `xmlns` – префикс, то он определяет префикс, если `xmlns` – атрибут, то он определяет имя пространства имен.

Теперь обратимся к средствам X-DOM, предназначенным для работы с пространствами имен.

Мы уже отмечали (см. п. 7.2), что причиной введения специального типа `XName` для имен элементов и атрибутов является возможность связывания собственно имен (локальных имен) с дополнительным компонентом имени – пространством имен. Строка типа `string` может быть неявно преобразована в тип `XName`; обратное преобразование должно выполняться явно с помощью метода `ToString`.

Класс `XName` включает следующие свойства (доступные только для чтения): `LocalName` (типа `string`) – локальное имя, `Namespace` (типа `XNamespace`) – пространство имен, `NamespaceName` (типа `string`) – строковое представление пространства имен.

Класс `XNamespace` предназначен для хранения пространств имен. Как и для класса `XName`, для класса `XNamespace` возможно неявное преобразование строки к типу `XNamespace`, а обратное преобразование должно быть выполнено явно, с использованием метода `ToString`. Кроме того, строковое представление пространства имен может быть получено с помощью свойства `NamespaceName`. Удобным средством, позволяющим быстро формировать полные имена по пространствам имен и локальным именам, является переопределенная операция «+», первый операнд которой определяет пространство имен и имеет тип `XNamespace`, а второй операнд определяет локальное имя и имеет тип `string`. С помощью подобного варианта операции «+» можно получать полные имена, не используя описанный выше формат представления полного имени (в котором пространство имен должно заключаться в фигурные скобки – см. раздел отладки на рис. 49).

В классе `XNamespace` определены также статические (то есть классовые) свойства типа `XNamespace`, связанные со специальными пространствами имен; среди них следует отметить свойство `Xmlns`, связанное с `xmlns` URI (<http://www.w3.org/2000/xmlns/>) и используемое при определении префиксов пространств имен.

При создании и преобразовании XML-документов практически никогда не требуется явным образом *определять* атрибуты `xmlns`, задающие пространства имен. Вместо этого требуемые пространства имен должны *явным образом указываться в именах элементов*. Также не требуется напрямую обращаться к префиксам пространств имен (хотя, разумеется, их необходимо явно определить в XML-документе): вместо этого следует дополнять имена элементов или атрибутов тем пространством имен, которое связано с префиксом.

При программном задании пространства имен для некоторого элемента его элементы-потомки *не получают этого же простран-*

ства имен автоматически. Для каждого элемента пространство имен необходимо задавать явным образом.

Когда сформированный XML-документ сохраняется в файле или преобразуется в текст, все необходимые атрибуты `xmlns`, а также все префиксы пространств имен добавляются в XML-документ автоматически, причем с учетом описанных выше правил, касающихся областей видимости пространств имен. Например, если все элементы документа имеют одно и то же пространство имен (не связанное с префиксом), то это пространство имен будет определено в атрибуте `xmlns` корневого элемента, после чего его действие по умолчанию распространится на все элементы документа (и поэтому в других элементах атрибут `xmlns`, определяющий пространство имен, использоваться не будет). Если же некоторые элементы имеют другие пространства имен, то эти пространства имен будут определены в элементах явным образом (и распространятся на все их элементы-потомки).

Примечание. При изменении пространств имен у некоторых элементов может потребоваться явным образом удалить в этих элементах атрибуты `xmlns`, задающие «старые» пространства имен. Если этого не сделать, то при автоматическом добавлении к элементу атрибута `xmlns` с именем нового пространства имен будет возбуждено исключение, связанное с тем, что элемент не может иметь два атрибута с одинаковыми именами (сообщение об ошибке будет иметь вид «Префикс ... не может быть переопределен с ... на ... внутри того же начального тега элемента»).

Используем средства работы с пространствами имен, описанные выше, для решения задачи LinqXml57. Напомним, что в имеющемся варианте программы мы уже обеспечили ввод всех строковых данных и загрузили XML-документ из файла, связав с ним переменную `d`.

Вначале определим в корневом элементе префиксы пространств имен `x` и `y`, связав их с пространствами имен, хранящимися в строках `s1` и `s2`. Для этого нам необходимо использовать статическое свойство `Xmlns` класса `XNamespace` (определяющее стандартный префикс пространства имен `xmlns`), объединив его с именами определяемых префиксов операцией «+»:

```
d.Root.Add(new XAttribute(XNamespace.Xmlns + "x", s1),  
            new XAttribute(XNamespace.Xmlns + "y", s2));
```

Теперь нам необходимо перебрать все элементы документа и изменить в их именах пространства имен в соответствии с условием задачи: элементы нулевого и первого уровней должны получить пространство имен `s1`, а прочие элементы – пространство имен `s2`:

```
foreach (var e in d.Descendants())
{
    string s = e.Ancestors().Count() <= 1 ? s1 : s2;
    e.Name = (XNamespace)s + e.Name.LocalName;
}
```

В последнем операторе мы еще раз использовали операцию «+», при этом потребовалось явным образом преобразовать строку `s` к типу `XNamespace` (если не выполнить подобного преобразования, то к операндам будет применена «обычная» операция сцепления строк, в результате которой строка `s` станет входить в локальное имя элемента, превратив его в недопустимое, так как символы «\», присутствующие в пространствах имен, в локальное имя входить не могут – см. первое примечание в п. 7.3).

Чтобы не выполнять преобразования типа на каждой итерации цикла, целесообразно выполнить их один раз до цикла:

```
XNamespace ns1 = (XNamespace)s1, ns2 = (XNamespace)s2;
foreach (var e in d.Descendants())
{
    XNamespace ns = e.Ancestors().Count() <= 1 ? ns1 : ns2;
    e.Name = ns + e.Name.LocalName;
}
```

Подчеркнем, что при изменении имен элементов мы не обращались к ранее определенным префиксам пространств имен, а использовали лишь связанные с ними значения (то есть сами пространства имен).

После цикла останется сохранить измененный документ под тем же именем:

```
d.Save(name);
```

При сохранении документа с каждым элементом будет связан префикс, соответствующий тому пространству имен, которое входит в имя элемента.

Оператор отладочной печати (добавленный в программу на стадии ознакомления с задачей) разместим в конце функции `Solve`.

Приведем окно задачника, которое появится на экране при запуске полученного варианта решения (рис. 50).

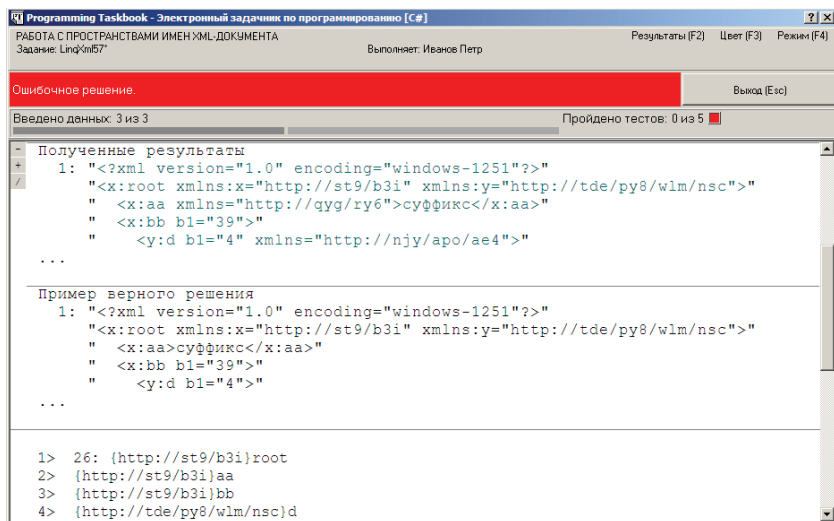


Рис. 50. Ошибочное выполнение задания LinqXml57

Мы видим, что имена элементов, указанные в разделе отладки, действительно содержат пространства имен, связанные с требуемыми префиксами. Кроме того, при сохранении полученного документа префиксы были автоматически указаны во всех тегах элементов.

Тем не менее решение, полученное программой, признано ошибочным. Ошибка состоит в том, что из документа не были удалены атрибуты `xmlns`, определяющие «старые» пространства имен. Еще раз подчеркнем, что *автоматического удаления атрибутов `xmlns` в результате изменения пространств имен не происходит* (см. предыдущее примечание). В нашем случае сохранение «старых» атрибутов `xmlns` не привело к возбуждению исключения, поскольку новые пространства имен связаны с префиксами и их указание не потребовало добавления в элементы новых атрибутов `xmlns`.

Для получения правильного решения достаточно организовать в цикле `foreach` удаление атрибутов `xmlns`. Однако при этом следует учитывать, что некоторые элементы не имеют подобного атрибута, поэтому для удаления нельзя использовать следующее выражение:

```
e.Attribute("xmlns").Remove();
```

При отсутствии атрибута `xmlns` метод `Attribute` вернет значение `null`, а при попытке применить к этому значению метод `Remove` будет возбуждено исключение `NullReferenceException` («В экземпляре объекта не задана ссылка на объект»).

Данная проблема решается путем использования варианта метода `Remove`, предназначенного для обработки *последовательности* атрибутов:

```
e.Attributes("xmlns").Remove();
```

Метод `Attributes` возвращает последовательность атрибутов `xmlns` для данного элемента (эта последовательность будет содержать не более одного элемента, так как элемент не может иметь атрибутов с одинаковыми именами). Метод `Remove` применяется к каждому элементу полученной последовательности; для пустой последовательности он не выполняет никаких действий.

Действия по удалению атрибутов не обязательно выполнять в цикле. Поскольку в задаче требуется удалить *все* атрибуты `xmlns`, входящие в исходный документ, мы можем выполнить это действие с помощью единственного оператора, указав его после (или до) цикла `foreach`:

```
d.Descendants().Attributes("xmlns").Remove();
```

Приведем полный текст полученного правильного решения (без отладочной печати):

```
public static void Solve()
{
    Task("LinqXml57");
    string name = GetString(), s1 = GetString(), s2 = GetString();
    XDocument d = XDocument.Load(name);
    d.Root.Add(new XAttribute(XNamespace.Xmlns + "x", s1),
        new XAttribute(XNamespace.Xmlns + "y", s2));
    XNamespace ns1 = (XNamespace)s1, ns2 = (XNamespace)s2;
    foreach (var e in d.Descendants())
    {
        XNamespace ns = e.Ancestors().Count() <= 1 ? ns1 : ns2;
        e.Name = ns + e.Name.LocalName;
    }
    d.Descendants().Attributes("xmlns").Remove();
    d.Save(name);
}
```

7.6. Дополнительные задания на обработку XML-документов: LinqXml61, LinqXml82

Завершающая подгруппа группы LinqXml (см. п. 4.6) содержит 30 задач, предназначенных для закрепления изученных ранее приемов обработки XML-документов. В отличие от задач предшествующих подгрупп, эти задачи не связаны с определенными категориями средств технологии LINQ to XML; в них требуется самостоятельно выбрать методы LINQ и классы X-DOM, позволяющие выполнить требуемое преобразование исходного XML-документа наиболее эффективным образом. Еще одним отличием данной подгруппы является близкий к реальному характер исходных XML-документов, каждый из которых связан с определенной предметной областью.

Можно провести аналогию между последней подгруппой группы LinqXml и группой LinqObj: входящие в их состав задачи посвящены закреплению ранее изученных технологий (LINQ to XML и LINQ to Objects соответственно) и близки к реальным задачам, возникающим при обработке сложных структур данных.

Еще одной чертой, приближающей обрабатываемые XML-документы к реальным, является наличие в них пространства имен, определенного в корневом элементе каждого документа.

Подобно задачам группы LinqObj, задачи из рассматриваемой подгруппы группы LinqXml можно разбить на *серии*, каждая из которых посвящена определенной предметной области (например, задачи LinqXml61–LinqXml67 связаны с обработкой данных о клиентах фитнес-центра). В пределах каждой серии задачи расположены по увеличению сложности: если для начальных задач требуется лишь изменить способ представления данных в XML-документе, не меняя его структуры, то в последующих задачах необходимо дополнительно выполнить группировку или объединение исходных данных, а также включить в документ новые данные, полученные из исходных путем их некоторого преобразования (в частности, агрегирования).

Вначале рассмотрим сравнительно простую задачу, в которой требуется только изменить способ представления данных.

LinqXml61. Дан XML-документ с информацией о клиентах фитнес-центра. Образец элемента первого уровня:

```
<record>  
  <id>10</id>
```



```
<date>2000-05-01T00:00:00</date>
<time>PT5H13M</time>
</record>
```

Здесь `id` – код клиента (целое число), `date` – дата с информацией о годе и месяце, `time` – продолжительность занятий (в часах и минутах) данного клиента в течение указанного месяца. Преобразовать документ, изменив элементы первого уровня следующим образом:

```
<time id="10" year="2000" month="5">PT5H13M</time>
```

Порядок следования элементов первого уровня не изменять.

На рис. 51 и 52 приведено окно задачника, которое будет отображено на экране при запуске программы-заготовки, созданной для этой задачи. Начальная часть раздела исходных данных показана на рис. 51, а ее завершающая часть, вместе с примером правильного решения, – на рис. 52.

Задача состоит в следующем изменении представления для элементов первого уровня: три его *дочерних элемента*, содержащих код

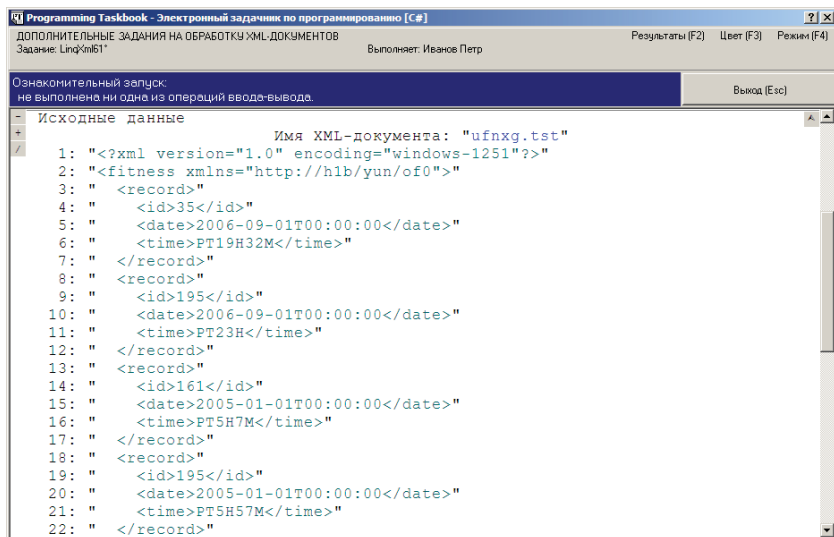


Рис. 51. Ознакомительный запуск задания LinqXml61 (начальная часть раздела исходных данных)

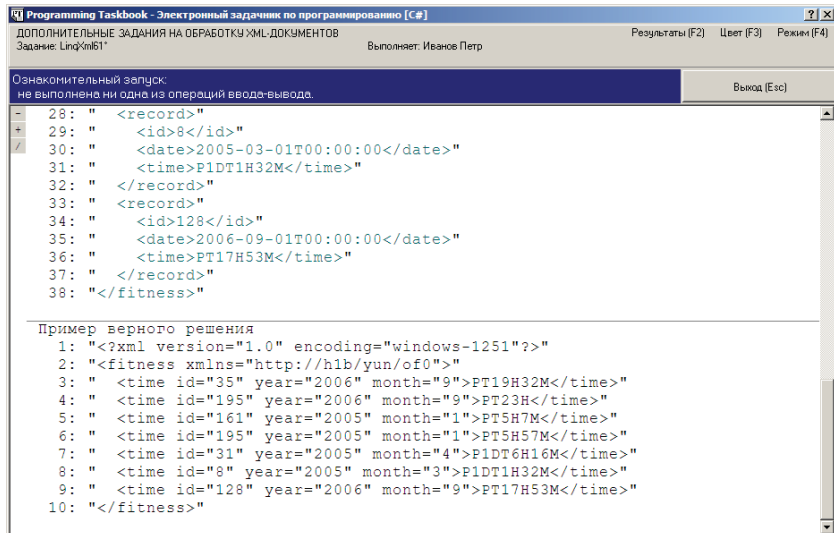


Рис. 52. Ознакомительный запуск задания LinqXml61
(завершающая часть раздела исходных данных
и раздел с примером верного решения)

клиента, дату (включающую год и месяц) и продолжительность его занятий в течение указанного месяца, надо заменить на три *атрибута* (со значениями, равными коду клиента, году и месяцу) и *строковый узел*, определяющий продолжительность занятий. Кроме того, следует изменить имя элементов первого уровня (прежнее имя *record* надо заменить на *time*). Все новые элементы должны иметь то же пространство имен, что и элементы исходного документа (это пространство имен задано в корневом элементе).

Будем считать, что исходный документ связан с переменной *d* типа *XDocument*, а определенное в нем пространство имен связано с переменной *ns* типа *XNamespace*.

Поскольку в задаче не требуется изменять структуру исходных данных, нам достаточно заменить каждый элемент первого уровня на элемент, в котором изменено представление соответствующих данных. Для этого удобно использовать метод *ReplaceNodes*, вызвав его для корневого элемента:

```
d.Root.ReplaceNodes(...);
```

На месте многоточия надо указать новую последовательность элементов, которые должны заменить исходные элементы первого уровня. Эту последовательность можно получить на основе исходной последовательности элементов первого уровня, выполнив ее проецирование методом `Select`. Приведем выражение, позволяющее получить требуемую последовательность элементов:

```
d.Root.Elements()  
  .Select(e => new XElement(ns + "time",  
    new XAttribute("id", e.Element(ns + "id").Value),  
    new XAttribute("year",  
      ((DateTime)e.Element(ns + "date")).Year),  
    new XAttribute("month",  
      ((DateTime)e.Element(ns + "date")).Month),  
    e.Element(ns + "time").Value));
```

При формировании нового элемента (обратите внимание на то, что к его имени `time` слева прибавляется пространство имен `ns`) мы сразу определяем его содержимое, используя механизм функционального конструирования (см. п. 7.1): к созданному элементу добавляются три атрибута и одно значение строкового типа, которое автоматически преобразуется в текстовый узел (типа `XText`).

При обращении ко всем элементам исходного документа необходимо явно указывать связанное с ними пространство имен (напомним, что пространство имен, определенное в корневом элементе, распространяется по умолчанию на все его элементы-потомки – см. п. 7.5).

Для задания значения атрибута `id` и строкового содержимого элемента `time` достаточно использовать значения исходных элементов с именами `id` и `time` соответственно (то есть их свойства `Value`). Значения атрибутов `year` и `month` необходимо *извлечь* из значения исходного элемента `date`; для этого удобно выполнить приведение элемента `date` к типу `DateTime` и воспользоваться свойствами `Year` и `Month` данного типа (приемы преобразования типов при обработке XML-документов были подробно рассмотрены в п. 7.4). Подчеркнем, что приведение типа необходимо применить не к значению элемента `date`, а к самому этому элементу.

Рассмотренное выражение следует указать в качестве параметра метода `ReplaceNodes` (вместо многоточия).

После выполнения указанного преобразования останется сохранить измененный XML-документ под тем же именем.

Получаем следующий вариант решения задачи:

```
public static void Solve()
{
    Task("LinqXml61");
    string name = GetString();
    XDocument d = XDocument.Load(name);
    XNamespace ns = d.Root.Name.Namespace;
    d.Root.ReplaceNodes(d.Root.Elements()
        .Select(e => new XElement(ns + "time",
            new XAttribute("id", e.Element(ns + "id").Value),
            new XAttribute("year",
                ((DateTime)e.Element(ns + "date")).Year),
            new XAttribute("month",
                ((DateTime)e.Element(ns + "date")).Month),
            e.Element(ns + "time").Value)));
    d.Save(name);
}
```

Можно избавиться от единственного лямбда-выражения, используя вместо метода `Select` соответствующее выражение запроса (приведенный ниже фрагмент следует указать вместо фрагмента, выделенного в первом варианте решения полужирным шрифтом, удалив также одну из трех стоящих подряд скобок):

```
d.Root.ReplaceNodes(
    from e in d.Root.Elements()
    select new XElement(ns + "time",
```

После пяти тестовых запусков любого из указанных вариантов решения будет выведено сообщение о том, что задание выполнено.

В качестве второго примера рассмотрим задачу LinqXml82, в которой требуется изменить *структуру* исходного XML-документа.

LinqXml82. Дан XML-документ с информацией о задолженности по оплате коммунальных услуг. Образец элемента первого уровня (смысл данных тот же, что и в LinqXml76, в качестве имени элемента первого уровня указываются номера дома и квартиры, разделенные символом «-» (дефис) и снабженные префиксом `addr`, а в качестве значения этого элемента указывается размер задолженности для данной квартиры):

```
<addr12-23>1245.64</addr12-23>
```

Преобразовать документ, выполнив группировку данных по номеру дома, а в пределах каждого дома – по номеру этажа. Изменить элементы первого уровня следующим образом:

```
<house12>
  <floor1 count="0" total-debt="0" />
  ...
  <floor6 count="1" total-debt="1245.64" />
  ...
  <floor9 count="3" total-debt="3142.7" />
</house12>
```

Имя элемента первого уровня должно иметь префикс **house**, после которого указывается номер дома, имя элемента второго уровня должно иметь префикс **floor**, после которого указывается номер этажа. Атрибут **count** равен числу задолжников на данном этаже, атрибут **total-debt** определяет суммарную задолженность по данному этажу, округленную до двух дробных знаков (незначащие нули не отображаются). Если на данном этаже отсутствуют задолжники, то для соответствующего элемента второго уровня значения атрибутов **count** и **total-debt** должны быть равны 0. Элементы первого уровня должны быть отсортированы по возрастанию номеров домов, а их дочерние элементы – по возрастанию номеров этажей.

При решении этой задачи также следует заменить исходную последовательность элементов первого уровня на новую последовательность, используя метод **ReplaceNodes** для корневого элемента **Root**. Однако в данном случае отсутствует однозначное соответствие между элементами первого уровня исходного и нового XML-документов (поскольку в новом документе требуется выполнить дополнительную группировку данных). В подобной ситуации удобно предварительно сформировать вспомогательную последовательность данных анонимного типа, которую затем подвергнуть требуемым преобразованиям и использовать при определении компонентов нового XML-документа.

Приведем вариант функции **Solve**, в которой реализован первый этап решения: загружен исходный документ и на его основе определена вспомогательная последовательность **a**. Для проверки правильности полученной последовательности она выведена в разделе отладки (с помощью метода расширения **Show**).

```
public static void Solve()
{
    Task("LinqXml82");
    string name = GetString();
    XDocument d = XDocument.Load(name);
    XNamespace ns = d.Root.Name.Namespace;
    var a = d.Root.Elements()
        .Select(e =>
        {
            string[] s = e.Name.LocalName.Substring(4).Split('-');
            return new
            {
                house = int.Parse(s[0]),
                floor = (int.Parse(s[1]) - 1) % 36 / 4 + 1,
                debt = (double)e
            };
        })
        .Show();
}
```

Перед определением анонимного типа мы выполнили расщепление имени элемента методом `Split`, предварительно удалив из него начальный текст «`addr`» (методом `Substring`). Полученные фрагменты строки (представляющие собой строковые представления целых чисел) были преобразованы в целые числа методом `Parse`. Вместо номера квартиры (который не потребуется при обработке полученной последовательности) мы сохранили *номер этажа* (`floor`), соответствующий данной квартире; при этом была использована формула, в которой учитывается, что каждый подъезд включает 36 квартир, на каждом этаже имеются 4 квартиры, а нумерация квартир выполняется в каждом подъезде снизу вверх.

Следует также обратить внимание на способ определения значения задолженности. По условию задачи задолженность является *значением* элемента `e` (представленным в формате вещественного числа, соответствующем стандарту XML), поэтому для ее получения в числовом виде достаточно преобразовать элемент `e` к типу `double` (см. п. 7.4).

На рис. 53 и 54 приведено окно задачника, которое будет отображено на экране при запуске полученной программы. Раздел исходных данных показан на рис. 53, а раздел отладки – на рис. 54. Напомним, что для быстрого перехода от одного из разделов с заданием к разделу отладки, как и для обратного перехода, достаточно нажать клавишу пробела (подобную возможность мы уже использовали в п. 6.3).

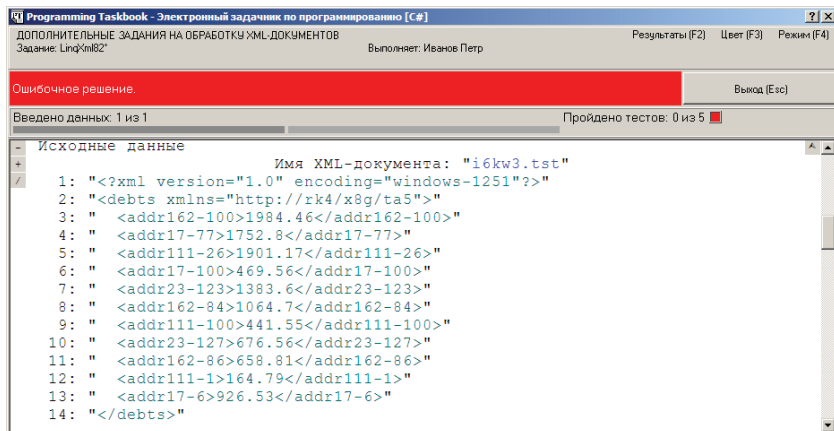


Рис. 53. Первый этап выполнения задания LinqXml82 (раздел исходных данных)

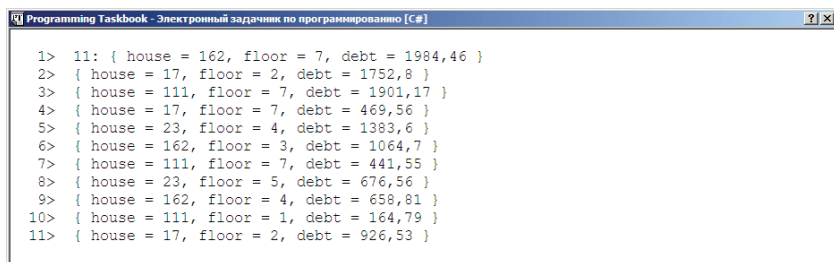


Рис. 54. Первый этап выполнения задания LinqXml82 (раздел отладки)

Сравнивая содержимое исходного XML-документа и полученной последовательности, можно убедиться в том, что последовательность построена правильно (заметим, что в разделе отладки вещественные данные отображаются с *запятой* в качестве десятичного разделителя).

Разумеется, данный вариант решения будет считаться ошибочным, поскольку мы еще не приступили к изменению исходного документа. Однако использование последовательности **a** позволит нам существенно упростить построение нового варианта XML-документа.

Приступим ко второму этапу решения – определению элементов первого уровня для нового XML-документа.

Прежде всего необходимо выполнить группировку элементов последовательности **a** по номерам домов. По каждому элементу сгруппированной последовательности надо определить XML-элемент, локальное имя которого получается присоединением текста «house» к номеру дома и дополняется пространством имен, определенным в исходном документе:

```
a.GroupBy(e => e.house,
(k, ee) => new XElement(ns + ("house" + k)))
```

Обратите внимание на скобки, использованные при определении имени элемента XML: они необходимы, поскольку *вначале* требуется полностью сформировать локальное имя элемента (типа string; при этом целочисленное значение *k* автоматически преобразуется к своему строковому представлению), а *затем* прибавить к нему слева пространство имен *ns* (типа XNamespace). Если бы скобки отсутствовали (то есть имя задавалось бы выражением *ns + "house" + k*), то компилятор вывел бы сообщение о том, что нельзя применить операцию «+» к операндам типа XName и int.

Требуется также учесть условие задачи о *порядке следования элементов*: элементы, соответствующие различным домам, должны быть отсортированы по возрастанию номеров домов. Проще всего обеспечить это условие, предварительно отсортировав нужным образом исходную последовательность **a**:

```
a.OrderBy(e => e.house)
  .GroupBy(e => e.house,
(k, ee) => new XElement(ns + ("house" + k)))
```

Полученную последовательность элементов XML надо указать в качестве параметра метода `ReplaceNodes` для корневого элемента `d.Root`, заменив тем самым исходную последовательность элементов первого уровня:

```
d.Root.ReplaceNodes(a.OrderBy(e => e.house)
  .GroupBy(e => e.house,
(k, ee) => new XElement(ns + ("house" + k))));
```

Протестируем полученный вариант решения, добавив в конец функции `Solve` оператор, обеспечивающий сохранение нового XML-документа:

d. Save (name);

В программе целесообразно сохранить отладочную печать вспомогательной последовательности **a**. На рис. 55 и 56 приведен результат запуска данного варианта (рис. 55 содержит раздел результатов, а рис. 56 – раздел отладки). Мы видим, что элементы первого уровня содержат все номера домов и упорядочены нужным образом.

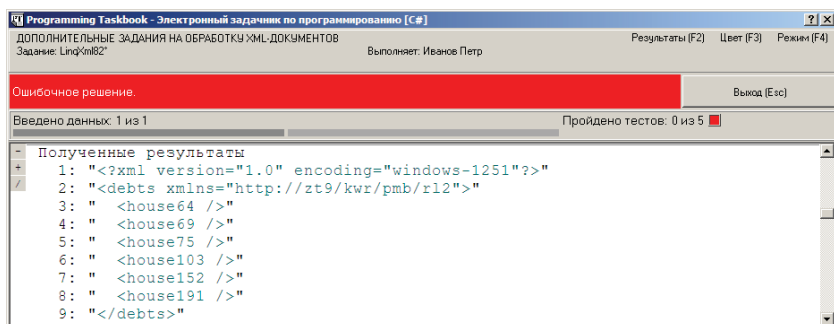


Рис. 55. Второй этап выполнения задания LinqXml82
(раздел результатов)

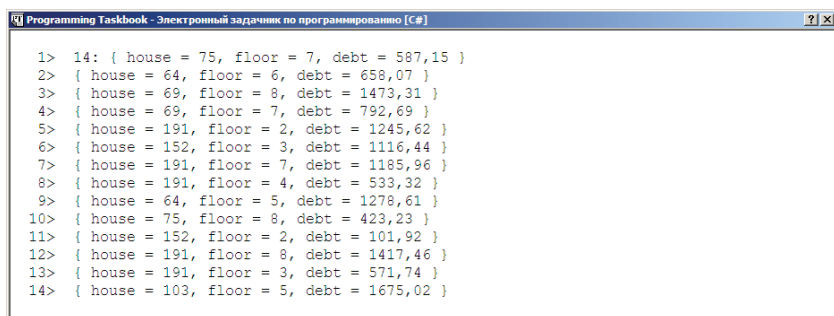


Рис. 56. Второй этап выполнения задания LinqXml82
(раздел отладки)

Нам осталось определить содержимое элементов первого уровня. По условию задачи каждый элемент первого уровня должен включать элементы, связанные с каждым этажом дома и содержащие информацию о задолжниках на этом этаже: число задолжников и сум-

марный размер их задолженности (информация должна быть представлена в виде двух атрибутов: `count` и `total-debt` соответственно).

Если бы в задаче требовалось вывести данные только о тех этажах, на которых имеются задолжники, то достаточно было бы еще раз применить метод `GroupBy`, выполнив группировку по номеру этажа в пределах каждого дома. Однако в этом случае в полученной последовательности будут отсутствовать данные о тех этажах, на которых нет задолжников. Поэтому воспользуемся методом `GroupJoin`, построив *левое внешнее объединение* вспомогательной целочисленной последовательности {1, 2, ..., 9} и последовательности задолжников для каждого дома по ключу, определяющему номер этажа (построение объединений подробно обсуждалось в п. 5.5).

Вспомогательную последовательность определим с помощью метода `Range` класса `Enumerable` (см. п. 5.2) и свяжем ее с переменной `floors`:

```
var floors = Enumerable.Range(1, 9);
```

Результат левого внешнего объединения преобразуем в последовательность элементов XML и, используя технологию функционального конструирования, добавим эту последовательность в конструкторы, определяющие элементы первого уровня. В приведенном операторе добавленный фрагмент выделен полужирным шрифтом:

```
d.Root.ReplaceNodes(a.OrderBy(e => e.house)
    .GroupBy(e => e.house,
        (k, ee) => new XElement(ns + ("house" + k),
            floors.GroupJoin(ee, e1 => e1, e2 => e2.floor,
                (e1, ee2) => new XElement(ns + ("floor" + e1),
                    new XAttribute("count", ee2.Count()),
                    new XAttribute("total-debt", ee2.Sum(e => e.debt)))))));
```

Прокомментируем добавленный фрагмент. В качестве второй (внутренней) последовательности, участвующей в построении левого внешнего объединения, используется последовательность `ee`, содержащая все элементы последовательности `a`, относящиеся к определенному дому. Результат левого внешнего объединения определяется в лямбда-выражении с параметрами (`e1`, `ee2`); в нем первый параметр равен номеру этажа, а второй параметр представляет собой последовательность всех задолжников, живущих на этом этаже (последовательность может быть пустой). По данным параметрам определяется XML-элемент, имя которого получается присоеди-

нием к строке «floor» номера этажа (и добавлением пространства имен ns). В конструкторе этого элемента определяются его атрибуты: count, равный количеству элементов последовательности ee2, и total-debt, равный сумме всех полей debt для элементов последовательности ee2. Заметим, что в данном случае нет необходимости прибегать к использованию метода DefaultIfEmpty для обработки пустых последовательностей ee2, так как и метод Count, и метод Sum корректно обрабатывают пустые последовательности, возвращая нулевые значения соответствующего типа.

Приведем полученный вариант решения задачи (в данном варианте удален вызов отладочного метода Show):

```
public static void Solve()
{
    Task("LinqXml82");
    string name = GetString();
    XDocument d = XDocument.Load(name);
    XNamespace ns = d.Root.Name.Namespace;
    var a = d.Root.Elements()
        .Select(e =>
        {
            string[] s = e.Name.LocalName.Substring(4).Split('-');
            return new
            {
                house = int.Parse(s[0]),
                floor = (int.Parse(s[1]) - 1) % 36 / 4 + 1,
                debt = (double)e
            };
        });
    var floors = Enumerable.Range(1, 9);
    d.Root.ReplaceNodes(a.OrderBy(e => e.house)
        .GroupBy(e => e.house,
            (k, ee) => new XElement(ns + ("house" + k),
                floors.GroupJoin(ee, e1 => e1, e2 => e2.floor,
                    (e1, ee2) => new XElement(ns + ("floor" + e1),
                        new XAttribute("count", ee2.Count()),
                        new XAttribute("total-debt", ee2.Sum(e => e.debt))))));
    d.Save(name);
}
```

Некоторые тестовые запуски полученного варианта решения будут успешными, однако возможны случаи, в которых созданный XML-документ будет отличаться от требуемого. Один из таких случаев приведен на рис. 57, на котором представлено окно задачника в режиме сокращенного отображения файловых данных и со

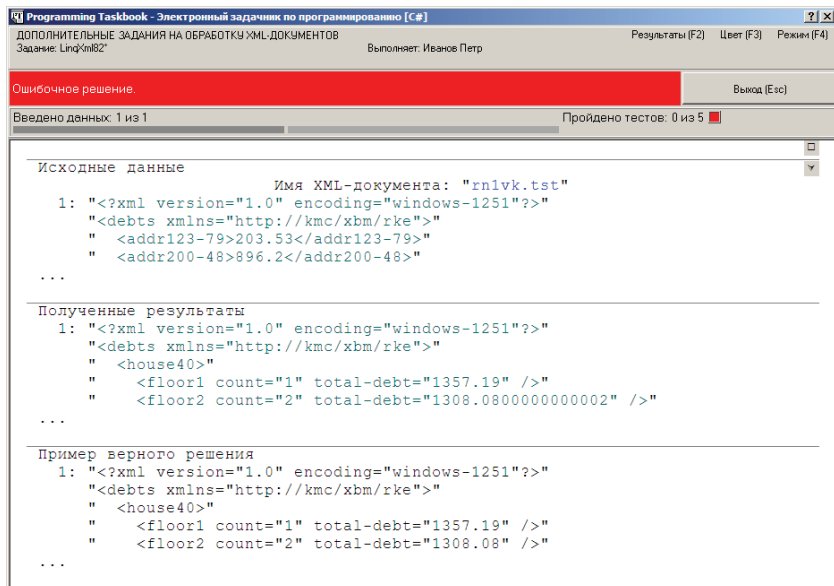


Рис. 57. Ошибочное выполнение задания LinqXml82

скрытой формулировкой задачи.

Анализируя полученный XML-документ, нетрудно заметить, что единственное его отличие от правильного варианта состоит в том, что некоторые вещественные числовые значения содержат более двух дробных знаков. Это обстоятельство может быть объяснено лишь накоплением погрешностей при вычислении суммы (поскольку по условию задачи все исходные вещественные данные имеют не более двух дробных знаков).


Для исправления подобной ошибки достаточно воспользоваться методом `Round` класса `Math`, позволяющим выполнять *округление* вещественных чисел до любого дробного знака (округляемое значение является первым параметром, число сохраняемых дробных знаков – вторым). В нашем случае следует применить метод `Round` к результату, возвращаемому методом `Sum`:

```
Math.Round(ee2.Sum(e => e.debt), 2)
```

После указанного исправления любой тестовый запуск будет успешным, и после пяти запусков в окне задачника появится сообщение о том, что задание выполнено.

Завершая рассмотрение этой задачи, приведем вариант построения новой последовательности XML-элементов первого уровня, использующий выражение запроса:

```
d.Root.ReplaceNodes(  
    from e in a  
    orderby e.house  
    group e by e.house  
    into ee  
    select new XElement(ns + ("house" + ee.Key),  
        from e1 in floors  
        join e2 in ee  
        on e1 equals e2.floor  
        into ee2  
        select new XElement(ns + ("floor" + e1),  
            new XAttribute("count", ee2.Count()),  
            new XAttribute("total-debt",  
                Math.Round(ee2.Sum(e => e.debt), 2)))))
```



Глава 8. Новые средства языка C# 3.0, связанные с технологией LINQ

8.1. Лямбда-выражения

Перед описанием лямбда-выражений, появившихся в версии C# 3.0, напомним о некоторых возможностях языка C#, доступных в более ранних версиях и тесно связанных с лямбда-выражениями.

В большинстве методов LINQ предусмотрены параметры, с помощью которых определяются функции, обеспечивающие обработку каждого элемента некоторой последовательности. Например, метод фильтрации *Where* имеет параметр-*предикат* *predicate*, принимающий на входе элемент последовательности и возвращающий значение *true*, если данный элемент надо включить в результирующую («отфильтрованную») последовательность, и *false*, если элемент включать не следует.

В интерфейсе LINQ to Objects подобные параметры описываются как *делегаты*. Делегаты (*delegates*) являются аналогами процедурных типов и позволяют хранить ссылки на конкретные методы; при этом *сигнатура* метода должна соответствовать данному делегату (сигнатура определяется количеством и типами параметров метода, а также типом возвращаемого значения).

Хотя делегаты, как и прочие типы данных, в языке C# являются классами, для их определения предусмотрен упрощенный синтаксис, «скрывающий» их классовую природу. Приведем пример описания делегата, принимающего один целочисленный параметр и возвращающего логическое значение:

```
public delegate bool IntPredicate(int e);
```

После определения предиката `IntPredicate` можно описывать методы, принимающие параметры данного типа. При этом если, например, некоторый метод имеет параметр `pred` типа `IntPredicate`, то при вызове этого метода в качестве параметра `pred` можно указать имя любого метода, принимающего целое число и возвращающего значение логического типа.

Легко понять, что для фильтрации последовательностей различных типов методом `Where` потребуется использовать различные делегаты (например, определенный выше делегат `IntPredicate` можно использовать только при обработке *целочисленных* последовательностей). Более того, предусмотреть заранее полный набор предикатов для обработки любых последовательностей просто невозможно, так как последовательности могут содержать элементы, тип которых определен в самой программе.

Для решения этой проблемы в язык C# 2.0 была добавлена возможность использования *обобщенных делегатов* (*generic delegates*), при определении которых можно указывать обобщенные параметры-типы. В качестве примера приведем определение обобщенного делегата `Predicate<T>`, который можно использовать при фильтрации последовательности с элементами типа `T`:

```
public delegate bool Predicate<T>(T e);
```

Наличие такого обобщенного делегата позволяет конструировать необобщенные делегаты-предикаты для любых типов данных, например `Predicate<int>`, `Predicate<string>` и т. д.

Таким образом, используя обобщенные делегаты, можно легко определять целые семейства делегатов с нужными характеристиками. Более того, необходимости в определении новых обобщенных делегатов обычно не возникает, поскольку в стандартной библиотеке .NET Framework уже определено семейство перегруженных обобщенных делегатов `Func`, с помощью которого можно сконструировать большинство необобщенных делегатов, возвращающих значения. Эти делегаты параметризуются несколькими обобщенными типами, причем последний из них определяет тип возвращаемого значения, а все предыдущие – типы параметров, например:

```
public delegate TResult Func<TResult>();  
public delegate TResult Func<T1, T2, T3, T4, TResult>(T1 arg1,  
    T2 arg2, T3 arg3, T4 arg4);
```

В версии .NET Framework 2.0 максимальное число обобщенных параметров, определяющих типы параметров делегата, равно 4 (см. второй из приведенных примеров); в версии 4.0 число подобных параметров было увеличено до 16). Отметим, что имеется также семейство обобщенных делегатов Action, не возвращающих значения:

```
public delegate void Action<T1, T2>(T1 arg1, T2 arg2);
```

Именно делегаты типа Func используются во всех методах, входящих в интерфейс LINQ to Object. Для того чтобы в этом убедиться, достаточно добавить в текст программы вызов метода Where для некоторой последовательности (для определенности будем считать, что элементы последовательности имеют тип double). После ввода текста «Where» и следующей за ним открывающей скобки на экране появится всплывающая подсказка о параметрах метода, которая будет иметь следующий вид:

```
(Func<double, bool> predicate)
```

Подсказка означает, что в качестве параметра метода Where в данном случае надо указать метод, принимающий единственный параметр типа double и возвращающий значение типа bool. Если в этот момент нажать клавишу со стрелкой, направленной вниз или вверх, то в подсказке будет приведен другой вариант параметров для метода Where:

```
(Func<double, int, bool> predicate)
```

Данный вариант метода Where принимает метод с двумя параметрами (первый типа double, второй типа int) и по-прежнему возвращает логическое значение. В обоих вариантах параметр типа double определяет обрабатываемый элемент исходной последовательности; в варианте с двумя параметрами параметр типа int определяет индекс этого элемента.

Как задать требуемый параметр-делегат? Можно описать соответствующий вспомогательный метод, после чего указать его имя в методе Where, например:

```
public static bool IsPositive(double e)
{
    return e > 0;
}
...
```



```
b = a.Where(IsPositive);  
...
```

Понятно, что такой способ является крайне неудобным и фактически сводит на нет все преимущества технологии LINQ. Уже в C# 2.0 была добавлена возможность создания методов «на лету» для передачи их в качестве параметров-делегатов. Подобные методы получили название *анонимных методов*. С применением анонимного метода вызов `Where` для отбора всех положительных элементов последовательности `a` можно оформить следующим образом:

```
b = a.Where(delegate(double e) { return e > 0; });
```

Такой способ задания предиката не требует предварительного описания соответствующего метода, но все же является излишне многословным. Для того чтобы сделать определение анонимных методов наиболее кратким и наглядным, в C# 3.0 были введены *лямбда-выражения* (данное название было заимствовано из терминологии функциональных языков программирования).

Лямбда-выражение имеет следующий синтаксис:

параметры => выражение или операторный блок

Если параметров нет, или их больше одного, или для них указывается описатель типа, то список параметров заключается в круглые скобки. Как правило, тип параметров указывать не требуется, так как он автоматически определяется («выводится») компилятором из определения того делегата, с которым связывается лямбда-выражение. Если метод состоит из одного оператора `return` *выражение*, то вместо операторного блока достаточно указать возвращаемое выражение.

Таким образом, лямбда-выражение представляет собой упрощенную запись анонимного метода, в которой можно не указывать те элементы, которые могут быть однозначно восстановлены компилятором.

При использовании лямбда-выражения приведенный выше оператор, обеспечивающих отбор всех положительных элементов, примет наиболее наглядный вид:

```
b = a.Where(e => e > 0);
```

Анализируя данное лямбда-выражение, компилятор, во-первых, определит, что параметр `e` должен иметь тип `double` (поскольку этот

тип имеют элементы последовательности **a**) и, во-вторых, заменит выражение `e > 0` на оператор `return`, возвращающий значение этого выражения. Таким образом, лямбда-выражение будет автоматически преобразовано компилятором к следующему виду:

```
b = a.Where((double e) => { return e > 0; });
```

Разумеется, при написании программы гораздо удобнее использовать более краткую форму лямбда-выражения, приведенную ранее. Следует, однако, заметить, что в некоторых (достаточно редких) случаях может потребоваться явно указать типы параметров, используемых в лямбда-выражениях (подобный пример приведен в п. 5.2).

При обработке как анонимных методов, так и сводящихся к ним лямбда-выражений, компилятор определяет на их основе вспомогательные методы того класса, в котором они были использованы. Таким образом, фактически все сводится к той ситуации, с которой мы начали обсуждение способов передачи параметра-делегата. Однако указанные действия компилятора выполняются «за кадром», позволяя описывать параметры-делегаты наиболее кратким и наглядным образом.

Код анонимного метода или лямбда-выражения может ссылаться на локальные переменные того метода, в котором он определен (такие переменные называются *внешними переменными*). Например, используя внешнюю переменную `c`, можно осуществить отбор элементов последовательности **a**, не превосходящих некоторого значения (записанного в переменной `c`):

```
double c;  
// ввод или явное определение значения переменной c  
b = a.Where(e => e <= c);
```

Обработка столь просто записываемого лямбда-выражения требует от компилятора достаточно сложных действий, поскольку при оформлении этого лямбда-выражения в виде вспомогательного метода в него необходимо включать и переменную `c`. Для того чтобы это оказалось возможным, переменная `c` «повышается» компилятором из разряда локальной переменной до поля некоторого вспомогательного класса, после чего она уже может быть указана во вспомогательном методе, созданном компилятором на основе лямбда-выражения (указанное действие носит название *захвата*

внешней переменной). Захват внешней переменной является хорошей иллюстрацией той большой и скрытой от глаз программиста работы, которую проводит компилятор при разборе лямбда-выражений.

Примечание. Описанная выше интерпретация лямбда-выражений как способа представления параметров-делегатов характерна для интерфейсов LINQ to Object и LINQ to XML, связанных с обработкой локальных данных (коллекций и XML-документов). В интерфейсах LINQ to SQL и LINQ to Entities, связанных с обработкой удаленных источников данных (см. гл. 9), лямбда-выражения используются для определения так называемых *деревьев выражений* (expression tree). Это связано с тем обстоятельством, что при обработке удаленных источников данных требуется не вызывать соответствующие методы-делегаты, а *интерпретировать их код*, преобразуя его в запросы, характерные для тех хранилищ данных, к которым осуществляется доступ (например, в запросы SQL). Для хранения кода лямбда-выражений и его последующей интерпретации и используются деревья выражений. Следует подчеркнуть, что вид лямбда-выражений во всех LINQ-интерфейсах остается неизменным; изменяется лишь его интерпретация компилятором. Более подробно особенности лямбда-выражений для интерфейсов LINQ to SQL и LINQ to Entities описываются в п. 9.2.

8.2. Анонимные типы и описатель var

Одним из нововведений языка C# версии 3.0 являются *анонимные типы* – безымянные классы, создаваемые «на лету» и содержащие набор свойств. Для создания экземпляра анонимного типа используется конструкция, которая начинается со слова `new` и содержит список свойств и их инициализаторов, заключенный в фигурные скобки, например:

```
var pupil1 = new { Name = "Петя", Age = 14, AvgMark = 4.5 };
```

Тип каждого свойства выводится из типа его инициализатора. Поскольку созданный тип не имеет имени, для описания экземпляра этого типа требуется использовать ключевое слово `var`. В дальнейшем можно обращаться ко всем свойствам созданного объекта, но *только для чтения* (здесь и далее комментарий к фрагменту программы, набранный полужирным шрифтом, содержит текст, который будет выведен на экран):

```
Console.WriteLine(pupill.Name + ", " + pupill.Age); // Петя, 14  
pupill.Age += 2; // ошибка компиляции
```

Имя свойства анонимного типа можно не указывать, если в качестве инициализатора этого свойства используется *переменная*, имя которой совпадает с именем определяемого свойства. В этом случае в списке свойств достаточно указать только инициализирующую переменную для данного свойства, например:

```
string Name = "Вася";  
int Age = 15;  
var pupil2 = new { Name, Age, AvrMark = 4.7 };  
Console.WriteLine(pupil2.Name + ", " + pupil2.Age); // Вася, 15
```

Для того чтобы два объекта имели один и тот же анонимный тип, необходимо, чтобы при их определении использовался одинаковый набор имен свойств и эти имена располагались в одинаковом порядке. Например, определенные выше объекты `pupil1` и `pupil2` имеют один и тот же анонимный тип и поэтому совместимы по присваиванию:

```
pupil2 = pupil1;
```

Если определить объекты `pupil3` и `pupil4` так, как указано ниже, то они будут иметь различные типы и, кроме того, их типы будут отличаться от типа объектов `pupil1` и `pupil2`:

```
var pupil3 = new { Name1 = "Ваня", Age = 12, AvrMark = 4.2 };  
var pupil4 = new { Age = 11, Name = "Коля", AvrMark = 4.8 };
```

Для экземпляра анонимного типа метод `ToString` возвращает строку, содержащую список имен всех его свойств и связанных с ними значений, причем список заключается в фигурные скобки. Эта особенность, в частности, проявляется при выводе экземпляров анонимных типов на экран (напомним, что метод `WriteLine` класса `Console` автоматически вызывает для своего параметра метод `ToString`, чтобы получить его строковое представление):

```
Console.WriteLine(pupill);  
// { Name = Петя, Age = 14, AvrMark = 4,5 }  
Console.WriteLine(pupil4);  
// { Age = 11, Name = Коля, AvrMark = 4,8 }
```

Анонимные типы применяются в основном при работе с запросами LINQ. Описатель var может также использоваться в качестве заменителя длинного имени типа.

8.3. Методы расширения

Методы расширения (extensions), введенные в C# 3.0, позволяют дополнить существующий класс новыми методами, не изменяя определения этого класса. Метод расширения всегда описывается в виде статического метода вспомогательного статического класса, причем первый параметр подобного метода должен снабжаться специальным модификатором this и иметь тип, совпадающий с типом того класса, расширением которого является данный метод.

Приведем два примера методов расширения (обратите внимание на слово static, которое надо указывать не только в заголовке каждого метода, но и в заголовке описания класса):

```
public static class ExtDemo
{
    public static bool Odd(this int n)
    {
        return n % 2 != 0;
    }
    public static int CharCount(this string s,
        params char[] chars)
    {
        int n = 0;
        foreach (char c in s)
            if (chars.Contains(c))
                n++;
        return n;
    }
}
```

В классе ExtDemo определены два статических метода расширения: один для типа int, второй для типа string. Для типа int определен метод Odd, позволяющий проверить целое число на нечетность, для типа string – метод CharCount, позволяющий подсчитать количество символов строки, входящих в символьный набор chars.

Эти методы можно вызывать, как обычные методы соответствующих классов:

```
Console.WriteLine(5.Odd()); // True
Console.WriteLine("123.123.456".CharCount('1', '2', '5')); // 5
```

Разумеется, никакого «дополнения» классов при определении их методов расширения не происходит; вместо этого компилятор, встретив метод расширения, автоматически заменяет его вызов на вызов соответствующего метода статического класса, в котором определено данное расширение. В нашем примере код будет автоматически преобразован к следующему:

```
Console.WriteLine(ExtDemo.Odd(5));  
Console.WriteLine(ExtDemo.CharCount("123.123.456", '1', '2', '5'));
```

Заметим, что благодаря использованию модификатора `params` в методе `CharCount` символы, включаемые в набор `chars`, можно указывать через запятую, не «превращая» их в обычный массив, хотя передача всех символов как одного массива также допускается:

```
"123.123.456".CharCount(new char[] { '1', '2', '5' })
```


Приведем еще один пример, в котором определяются два удобных перегруженных варианта метода `Split` класса `string`:

```
public static class ExtSplit  
{  
    public static string[] Split(this string src,  
        string separator, bool removeEmptyEntries)  
    {  
        return src.Split(new string[] { separator },  
            removeEmptyEntries ?  
                StringSplitOptions.RemoveEmptyEntries :  
                StringSplitOptions.None);  
    }  
    public static string[] Split(this string src,  
        string separator)  
    {  
        return Split(src, separator, true);  
    }  
}
```

В приведенных реализациях метода `Split` можно указывать строковый разделитель, не превращая его в одноэлементный массив (как это требуется в стандартном варианте `Split`, содержащем параметр типа `StringSplitOptions`). Кроме того, параметр, который определяет, надо ли считать расположенные рядом разделители одним разделителем, имеет не тип перечисления `StringSplitOptions`, а обычный

логический тип, причем имеется вариант метода, в котором данный параметр по умолчанию полагается равным `true`.

Основным достоинством методов расширения является возможность их использования в стиле, аналогичном стилю вызова «обычных» методов. Особенно явно это достоинство проявляется при использовании методов расширения, определенных в классе `Enumerable` и подобных ему классах, связанных с технологией LINQ.



Глава 9. Технологии LINQ для обработки удаленных источников данных

9.1. Интерфейсы LINQ to SQL и LINQ to Entities

Для обработки удаленных источников данных предусмотрены два стандартных интерфейса LINQ: LINQ to SQL и LINQ to Entities. Интерфейс LINQ to Entities является составной частью технологии Entity Framework, поэтому в книгах по LINQ названия LINQ to Entities и Entity Framework часто используются как синонимы.

Интерфейс LINQ to SQL обеспечивает преобразование запросов LINQ в запросы языка SQL – стандартного языка структурированных запросов, этот интерфейс ориентирован на SQL-серверы баз данных (в частности, Microsoft SQL Server).

Интерфейс LINQ to Entities преобразует запросы LINQ в запросы языка Entity SQL (ESQL) – усовершенствованного варианта SQL, реализованного в рамках технологии Entity Framework. Благодаря гибкости технологии Entity Framework интерфейс LINQ to Entities может использоваться не только для SQL-серверов, но и для других видов серверов баз данных.

Для возможности обработки базы данных с применением интерфейсов LINQ to SQL и LINQ to Entities необходимо предварительно построить *объектную модель* этой базы, определив *классы сущностей* (entity classes), связанные с ее компонентами. Наличие подобной модели позволяет организовать обработку базы данных на более высоком уровне, используя возможности объектно-ориентированного программирования.

Объектная модель, создаваемая для применения в рамках интерфейса LINQ to SQL, является более простой для реализации, в то

время как объектная модель для LINQ to Entities (называемая Entity Data Model – EDM) предоставляет больше гибкости при построении классов сущностей и связывании их с компонентами базы данных (например, можно связать одну таблицу базы данных с несколькими сущностями объектной модели или связать несколько таблиц с одной сущностью).

Универсальность технологии LINQ для обработки баз данных состоит в том, что после определения объектной модели и ее связывания с базой данных последующая обработка данных с применением как интерфейса LINQ to SQL, так и интерфейса LINQ to Entities выполняется посредством стандартного набора запросов LINQ, совпадающего в основном с набором запросов LINQ to Objects. Запросы LINQ to SQL и LINQ to Entities, как и запросы LINQ to Objects, записываются либо в виде цепочки методов с параметрами – лямбда-выражениями, либо в виде выражений запросов.

Таким образом, большинство приемов составления запросов для обработки локальных последовательностей, изученных нами при рассмотрении технологии LINQ to Objects, может быть без изменений перенесено в программы, связанные с обработкой удаленных наборов данных. Однако, несмотря на то что *по форме* запросы к локальным и удаленным наборам данных практически совпадают, *механизм* их выполнения при обработке удаленных наборов данных имеет существенные отличия. Эти отличия описываются в следующем пункте.

9.2. Интерфейс IQueryable<T> и интерпретируемые запросы

Напомним, что все запросы, входящие в интерфейс LINQ to Objects, реализованы в виде методов расширения класса Enumerable и могут применяться к любым *локальным* последовательностям, то есть объектам, реализующим интерфейс IEnumerable<T>, например строкам (интерпретируемым как последовательность символов), массивам, спискам List<T> и т. д. При этом лямбда-выражения в этих запросах определяют *делегаты*, то есть методы, которые вызываются при обработке каждого элемента обрабатываемой последовательности.

Интерфейсы LINQ to SQL и LINQ to Entities содержат другую реализацию запросов LINQ. Эти запросы реализованы в виде методов расширения класса Queryable и могут применяться к последо-

вательностям, реализующим интерфейс `IQueryable<T>`. Такой интерфейс реализуют все последовательности, связанные с объектной моделью базы данных.

Основным отличием методов расширения класса `Queryable` от одноименных методов расширения класса `Enumerable` является то, что используемые в методах класса `Queryable` в качестве параметров лямбда-выражения интерпретируются не как делегаты, а как *деревья выражений* (expression trees). В то время как делегат представляет собой обычную функцию, которая преобразуется компилятором в машинный код (точнее, в специальный код на низкоуровневом языке платформы .NET) и затем выполняется, дерево выражений преобразуется компилятором в иерархическую структуру объектов, полностью описывающую лямбда-выражение, на основании которого дерево было построено. На этапе выполнения дерева выражений анализируются (*интерпретируются*) и преобразуются в элементы запроса SQL (или ESQL), который посылается серверу базы данных и возвращает требуемый результат (отдельный элемент или последовательность). Чтобы подчеркнуть эту особенность запросов LINQ to SQL и LINQ to Entities, они называются *интерпретируемыми запросами*, в отличие от *локальных запросов* LINQ to Objects, основанных на использовании делегатов.

Разумеется, для успешной интерпретации запроса необходимо иметь дополнительную информацию о структуре той базы данных, для обработки которой используется запрос. Эту информацию компилятор извлекает из объектной модели базы данных.

Необходимо отметить еще одну особенность интерпретируемых запросов. В то время как при выполнении цепочки *локальных* запросов каждый запрос выполняется последовательно, вызывая указанные в нем делегаты, в случае *интерпретируемых* запросов *вся* цепочка запросов (включая все содержащиеся в ней вложенные запросы) преобразуется в один запрос, посылаемый на сервер базы данных. Тем самым минимизируется количество обращений к удаленному хранилищу данных.

Интерпретируемые запросы, подобно локальным запросам, имеют отложенный («ленивый») характер. Иными словами, они не выполняются до тех пор, пока в программе не встречается цикл `foreach` для перебора элементов результирующей последовательности. Исключение составляют запросы, возвращающие отдельные элементы или завершающиеся вызовом методов экспортирования (например, `ToArray`). При этом если в дальнейшем потребуется выполнить пере-

бор последовательности еще раз, запрос к базе данных будет выполнен повторно (то есть он опять будет послан серверу для получения результирующей последовательности).

Приведем пример. Будем предполагать, что в программе установлена связь с базой данных, содержащей таблицу Customer с информацией о потребителях. Таблица включает следующие поля:

- ❑ Id – код потребителя (ключевое поле целого типа);
- ❑ Year – год рождения (поле целого типа);
- ❑ Street – улица проживания (поле строкового типа).

Будем также предполагать, что в программе определен класс Customer с этими же полями, входящий в состав объектной модели базы данных для интерфейса LINQ to SQL, и описана переменная cust типа IQueryable<Customer>, для которой установлена связь с таблицей Customer базы данных (вопросы, связанные с созданием базы данных, подключением к базе данных и ее таблицам, а также настройкой ее объектной модели, обсуждаются в п. 9.4).

Имея переменную cust, мы можем проанализировать содержимое таблицы Customer с помощью запросов LINQ. Например, определим переменные целого типа count и max и найдем count самых молодых потребителей из числа тех, год рождения которых меньше значения max, после чего выведем для этих потребителей поля Id и Year, отсортировав полученных потребителей по убыванию года рождения:

```
int count = 5, max = 1990;
var r = cust.Where(e => e.Year < max)
    .OrderByDescending(e => e.Year)
    .Take(count);
foreach (var e in r)
    Console.WriteLine(e.Id + " " + e.Year);
```

Мы видим, что запись методов LINQ ничем не отличается от записи, применявшейся нами при обработке локальных коллекций. Однако если с помощью технологии IntelliSense отобразить во всплывающей подсказке типы параметров одного из методов, например Where, то в качестве типа предиката (задаваемого лямбда-выражением) будет указан тип

Expression<Func<Customer, bool>>

Тип Func<Customer, bool> определяет *делегата*, имеющего параметр типа Customer и возвращающего логическое значение (см. п. 8.1). В данном случае этот делегат является параметром-типом

для класса Expression, определенного в пространстве имен System.Linq.Expressions и предназначенного для хранения деревьев выражений.

При выполнении программы цепочка запросов LINQ будет преобразована в *единственный* SQL-запрос примерно такого вида:

```
SELECT TOP 5 *  
  FROM Customer  
 WHERE Customer.Year < @max  
 ORDER BY Customer.Year DESC
```

Обратите внимание на то, что в выражении WHERE указывается *переменная* @max (поскольку она входит в состав соответствующего дерева выражений), тогда как в выражении TOP указано *значение* переменной count (так как данный параметр в методе Take передается по значению).

Пересылка сформированного запроса на сервер произойдет при выполнении цикла foreach; в результате на экран будет выведен текст, подобный приведенному ниже:

```
30 1989  
23 1989  
13 1987  
20 1984  
7 1981
```

Дополним полученную программу следующим фрагментом:

```
Console.WriteLine();  
count = 10;  
max = 1988;  
foreach (var e in r)  
    Console.WriteLine(e.Id + " " + e.Year);
```

Подчеркнем, что мы *не изменили* оператор, связанный с определением результирующей последовательности r. Однако в силу отложенного характера выполнения запросов результат выполнения второго цикла foreach будет отличаться от результата первого (второй набор данных отделяется от первого пустой строкой):

```
30 1989  
23 1989  
13 1987  
20 1984
```



7 1981

13 1987

20 1984

7 1981

17 1981

14 1979

При выполнении второго цикла SQL-запрос будет повторно отправлен на сервер. Поскольку в этом запросе используется переменная *max*, значение которой было изменено, вторая последовательность будет содержать другой набор значений. Заметим, что изменение значения переменной *count* *не приведет* к изменению количества полученных записей. Это связано с тем, что значение *count* используется в запросе в качестве обычного целочисленного параметра (а не в составе некоторого лямбда-выражения).

Аналогичные эффекты, связанные с отложенным характером выполнения запросов LINQ, могут возникать не только для интерпретируемых, но и для локальных запросов (см. примечание в п. 5.4).

9.3. Основные ограничения на запросы LINQ для удаленных источников данных

Некоторые методы LINQ, а также некоторые варианты перегруженных методов, доступные для локальных коллекций, нельзя применять в запросах к базам данных. Основное ограничение здесь связано с тем, что к записям таблицы базы данных нельзя обратиться по индексу.

Перечислим те методы, использование которых в запросах LINQ to SQL и LINQ to Entity приводит к возбуждению исключения:

- ☐ TakeWhile;
- ☐ SkipWhile;
- ☐ Reverse;
- ☐ ElementAt;
- ☐ ElementAtOrDefault;
- ☐ Aggregate;
- ☐ Zip.

Для методов *Where*, *Select* и *SelectMany* запрещено использовать варианты лямбда-выражений с дополнительным вторым параметром — индексом элемента.

Еще одно ограничение, связанное с запросами к базам данных, состоит в том, что применяемые в этих запросах лямбда-выражения

должны содержать только *возвращаемые выражения* (а не набор операторов, что допускается в локальных запросах).

Необходимо отметить, что в данном пункте перечислены лишь *основные* ограничения на запросы LINQ для интерфейсов LINQ to SQL и LINQ to Entity. В силу специфики подобных интерфейсов некоторые запросы и их комбинации, успешно выполняющиеся для интерфейса LINQ to Objects, будут приводить к ошибкам времени выполнения в случае их использования для интерфейса LINQ to SQL, LINQ to Entity или обоих этих интерфейсов.

9.4. Пример применения интерфейса LINQ to SQL: LinqObj71

Для того чтобы привести пример применения технологии LINQ для обработки баз данных, обратимся к задаче LinqObj71 из второй подгруппы группы LinqObj (см. п. 3.2). Описываемые в задаче LinqObj71 (как и в других задачах этой подгруппы) наборы исходных данных можно интерпретировать как содержимое *нескольких связанных таблиц* некоторой базы данных. Определив и заполнив таблицы этой базы (для заполнения будем использовать наборы данных, сгенерированные задачиком), мы обработаем базу данных с помощью запросов LINQ, получив требуемые результаты, что позволит задачику проверить правильность наших действий.

В качестве интерфейса LINQ для доступа к базе данных будем использовать интерфейс LINQ to SQL. Этот интерфейс поддерживается во всех рассматриваемых версиях Visual Studio (2008, 2010, 2012) и является более простым и чаще используемым, чем интерфейс LINQ to Entities. Кроме того, применение интерфейса LINQ to SQL не требует знания особенностей технологии Entity Framework, составной частью которой является интерфейс LINQ to Entities.

Вначале мы рассмотрим пример, связанный с обработкой локальной базы данных и использующий простейшую объектную модель. Затем мы опишем особенности работы с базой данных, управляемой SQL-сервером, для доступа к которой используем автоматически сгенерированную объектную модель с расширенными возможностями.

9.4.1. Создание и настройка локальной базы данных

В задаче LinqObj71 используются две последовательности с именами *A* и *C*:

A: <Код потребителя> <Год рождения> <Улица проживания>
C: <Код потребителя> <Скидка (в процентах)> <Название магазина>

Данные из последовательности A будем хранить в таблице Customer, имеющей следующие поля:

- Id – код потребителя (ключевое поле целого типа);
- Year – год рождения (поле целого типа);
- Street – улица проживания (поле строкового типа).

Данные из последовательности C будем хранить в таблице Discount, снабдив ее дополнительным полем Id, уникальным для каждой записи:

- Id – идентификатор записи (ключевое поле целого типа);
- IdCust – код потребителя (поле целого типа);
- Value – значение скидки в процентах (поле целого типа);
- Shop – название магазина (поле строкового типа).

Для таблиц следует установить связь «один ко многим» («один потребитель – много скидок»), используя для этой связи поля Id таблицы Customer и IdCust таблицы Discount.

В качестве проекта-заготовки будем использовать проект, созданный для задания LinqObj71 (создание проекта с помощью программы PT4Load описывается в п. 5.1.1). При описании последующих действий будем предполагать, что используется среда программирования Microsoft Visual Studio 2010. В среде Visual Studio 2008 все действия выполняются аналогично. Версия 2012 имеет незначительные отличия, которые мы будем специально оговаривать. Поскольку для версии 2012 распространен локализованный вариант с русскоязычным интерфейсом, при первом упоминании команды меню или другого интерфейсного элемента наряду с английским вариантом названия будем приводить и его русский эквивалент.

Начнем с создания базы данных. Для этого выполним команду меню **Project ⇒ Add New Item...** (Проект ⇒ Добавить новый элемент...). В появившемся окне будет выведен список всех объектов, которые можно добавить к нашему проекту. Для создания базы данных можно использовать два объекта: **Local Database** (Локальная база данных), позволяющий создать локальную базу данных, сохранив ее в файле с расширением sdf, и **Service-based Database** (База данных, основанная на службах), позволяющий создать базу данных, управляемую SQL-сервером, сохранив ее в файле с расширением mdf. Локальная база данных может использоваться одновременно

только одним приложением и не требует подключения SQL-сервера для доступа к ее данным.

Так как локальную базу данных проще создать и настроить, выберем вариант **Local Database** (действия по созданию базы данных, основанной на службах, будут описаны далее, в п. 9.4.3). В качестве имени базы данных укажем LocalDB.sdf и нажмем кнопку **Add** (Добавить). Если после этого действия появится дополнительное окно **Data Source Configuration Wizard** (Мастер настройки источника данных), то сразу закроем его, нажав кнопку **Cancel** (Отмена).

В результате к проекту будет добавлен файл LocalDB.sdf. Выполнив двойной щелчок на его имени в окне **Solution Explorer** (Обозреватель решений), мы отобразим на экране еще одно окно, имеющее заголовок **Server Explorer** (Обозреватель серверов) и позволяющее настроить конфигурацию созданной базы данных. Заметим, что файл LocalDB.sdf будет размещен в том же каталоге, в котором содержатся и остальные файлы проекта, то есть в *рабочем каталоге* задачника (по умолчанию это каталог c:\PT4Work).

Приступим к настройке базы данных. Для создания новой таблицы достаточно вызвать контекстное меню подпункта **Tables** (Таблицы) для базы данных LocalDB.sdf в окне **Server Explorer** (щелкнув на этом подпункте правой кнопкой мыши) и выполнить в этом меню команду **Create Table** (Создать таблицу). Появится диалоговое окно **New Table** (Новая таблица), в котором следует ввести имя таблицы Customer и определить ее поля (см. табл. 9.1, а также рис. 58).

Таблица 9.1. Поля таблицы Customer базы данных LocalDB.sdf

| Column Name | Data Type | Length | Allow Nulls | Unique | Primary Key |
|-------------|-----------|--------|-------------|--------|-------------|
| Id | int | 4 | No | Yes | Yes |
| Year | int | 4 | No | No | No |
| Street | nvarchar | 50 | No | No | No |

Нажатие кнопки **OK** обеспечит создание таблицы Customer, которая будет отображена в группе **Tables** окна **Server Explorer**.

Аналогичными действиями добавим в базу данных таблицу Discount и определим ее поля (см. табл. 9.2).

Таблица 9.2. Поля таблицы Discount базы данных LocalDB.sdf

| Column Name | Data Type | Length | Allow Nulls | Unique | Primary Key |
|-------------|-----------|--------|-------------|--------|-------------|
| Id | int | 4 | No | Yes | Yes |
| IdCust | int | 4 | No | No | No |
| Value | int | 4 | No | No | No |
| Shop | nvarchar | 50 | No | No | No |

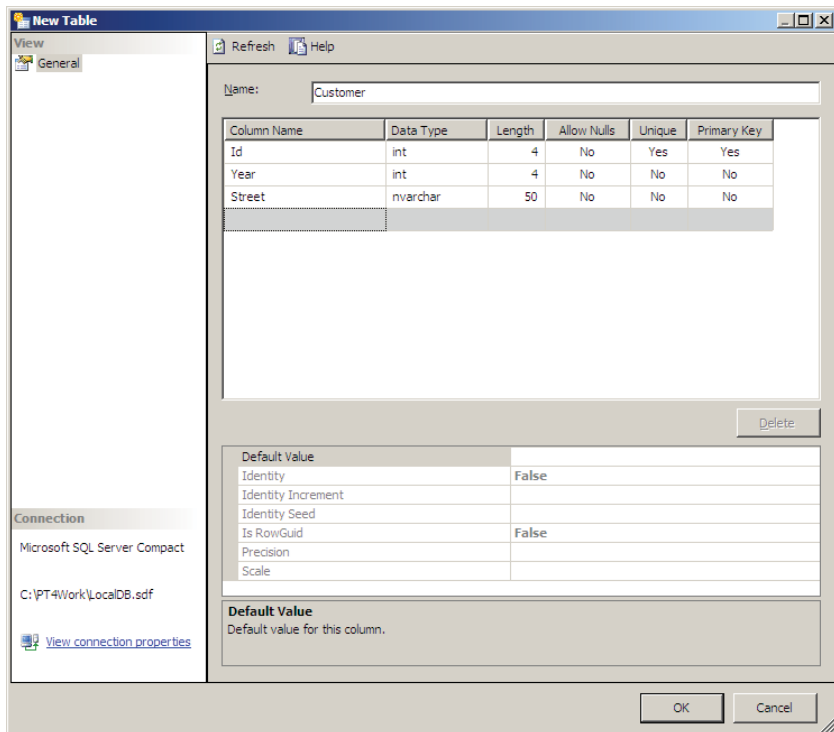


Рис. 58. Создание таблицы Customer

Осталось установить связь между созданными таблицами. Для этого необходимо вызвать в окне **Server Explorer** контекстное меню таблицы Discount (то есть той таблицы, которая в устанавливаемой связи соответствует варианту «ко многим»), выполнить команду **Table Properties** (Свойства таблицы) и в появившемся окне выбрать пункт **Add Relations** (Добавить связи). При настройке связи необходимо указать ее название (например, FK_Discount_Customer), а также имена таблиц и имена полей, по которым будет установлена эта связь. Заметим, что и имена таблиц, и имя главного ключа (**Primary Key**) для таблицы Customer, равное Id, уже будут указаны требуемым образом. Поэтому нам достаточно, кроме ввода названия связи, указать имя внешнего ключа (**Foreign Key**) для таблицы Discount, равное IdCust, после чего нажать кнопку **Add Columns** (Добавить столбцы) – см. рис. 59.

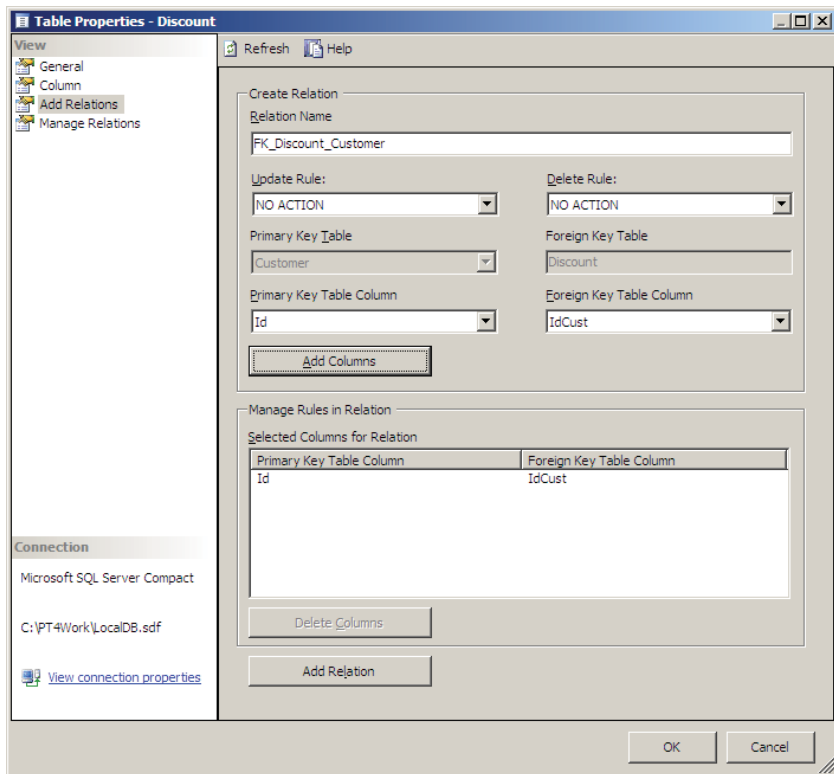


Рис. 59. Создание связи между таблицами Customer и Discount

В результате кнопка **Add Relation** (Добавить связь) станет доступна, и, нажав ее, мы установим требуемую связь, о чем будет выведено сообщение в информационном окне. Для завершения настройки связи останется закрыть диалоговое окно свойств, нажав кнопку **OK**.

На этом настройка конфигурации базы данных завершается.

Примечание. При последующей работе с нашим проектом его необходимо открывать способом, стандартным для среды Visual Studio (например, выполняя двойной щелчок на файле LinqObj71.sln или LinqObj71.csproj или загружая его в уже открытую среду Visual Studio с помощью соответствующих команд меню). Использовать для открытия данного проекта программу PT4Load не следует, так как при этом из проекта будут автоматически удалены все «лишние», с точки зрения задачника, компоненты,

в том числе файл базы данных, и придется повторно добавлять к проекту удаленные компоненты (заметим, что все компоненты, удаленные из проекта, сохраняются в рабочем каталоге).

9.4.2. Создание и использование простейшей объектной модели базы данных

Созданная в предыдущем пункте база LocalDB пока еще не содержит данных. Для занесения этих данных в таблицы можно использовать ввод с клавиатуры, выбрав требуемую таблицу в окне **Server Explorer** и выполнив команду **Show Table Data** (Показать таблицу данных) из ее контекстного меню. С помощью появившейся на экране таблицы можно осуществлять добавление новых и редактирование существующих записей.

Мы поступим по-другому, записав в нашу базу те данные, которые будут созданы задачиком при инициализации задания LinqObj71. Для подобного изменения базы данных необходимо обеспечить доступ к ее таблицам из программы, а для обеспечения доступа следует использовать *объектную модель* базы данных.

Реализуем простейшую объектную модель, позволяющую получить доступ к базе данных методами интерфейса LINQ to SQL.

Наша объектная модель будет включать два класса, соответствующих записям в таблицах Customer и Discount. Эти классы и входящие в них поля (соответствующие полям таблиц) необходимо снабдить специальными *атрибутами* Table и Column, которые будут учитываться при выполнении методов, обеспечивающих доступ к базе данных. Эти атрибуты определены в пространстве имен System.Data.Linq.Mapping, входящем в сборку System.Data.Linq. Для того чтобы атрибуты можно было указывать в программе, требуется выполнить два действия:

- 1) подключить к проекту сборку System.Data.Linq; для этого следует в окне **Solution Explorer** вызвать контекстное меню для раздела **References**, выполнить в нем команду **Add Reference...** (Добавить ссылку...), в появившемся окне выбрать в списке сборок компонент System.Data.Linq (в версии Visual Studio 2012 необходимо дополнительно *установить флажок* рядом с данным компонентом) и нажать кнопку **OK**;
- 2) добавить в начало файла LinqObj71.cs директиву using для требуемого пространства имен:

```
using System.Data.Linq.Mapping;
```

После выполнения указанных действий добавим в файл `Linq-Obj71.cs` описания двух классов, разместив их перед описанием класса `MyTask`:

```
[Table]
public class Customer
{
    [Column(IsPrimaryKey = true)]
    public int Id;
    [Column]
    public int Year;
    [Column]
    public string Street;
}

[Table]
public class Discount
{
    [Column(IsPrimaryKey = true)]
    public int Id;
    [Column]
    public int IdCust;
    [Column]
    public int Value;
    [Column]
    public string Shop;
}
```

Атрибуты `Table` и `Column` заключаются в квадратные скобки и могут снабжаться дополнительными параметрами в круглых скобках (в частности, с помощью параметра `IsPrimaryKey` можно пометить ключевое поле таблицы, а с помощью параметра `Name` – указать имя таблицы или ее поля, если оно отличается от имени класса или, соответственно, поля класса).

Заметим, что если какое-либо числовое поле таблицы может содержать пустые значения, то в соответствующем классе это поле должно иметь `Nullable`-тип (например, `int?`). В определенных нами таблицах ни одно поле не может содержать значение `NULL`, поэтому `Nullable`-типы использовать не требуется.

Имея классы `Customer` и `Discount`, мы можем создать последовательности объектов этих типов, получив значения их полей из файлов, подготовленных задачей. Для этого достаточно добавить в метод `Solve` следующий фрагмент (в качестве имен после-

довательностей будем использовать имена *A* и *C* из условия задачи LinqObj71):

```
var A = File.ReadAllLines(GetString(), Encoding.Default)
    .Select(e =>
    {
        string[] s = e.Split();
        return new Customer
        {
            Id = int.Parse(s[0]),
            Year = int.Parse(s[1]),
            Street = s[2]
        };
    });
var C = File.ReadAllLines(GetString(), Encoding.Default)
    .Select((e, i) =>
    {
        string[] s = e.Split();
        return new Discount
        {
            Id = i + 1,
            IdCust = int.Parse(s[0]),
            Value = int.Parse(s[1]),
            Shop = s[2]
        };
    });
```

В приведенном фрагменте мы считываем строки из исходных файлов, после чего методом `Select` преобразуем каждую строку в объект требуемого класса. При этом мы используем новую возможность языка *C#*, добавленную в версии 3.0: *вызов конструктора с явным заданием значений всех полей создаваемого объекта* (подобный вызов похож на оператор создания анонимного типа; отличается он тем, что после слова `new` указывается имя типа создаваемого объекта). Обратите также внимание на использование во втором методе `Select` лямбда-выражения с двумя параметрами; параметр `i`, определяющий индекс обрабатываемой строки, используется при задании ключевого поля `Id` таблицы `Discount` (это поле будет содержать целочисленные данные, начиная с 1).

Осталось занести полученные данные в таблицы базы `LocalDB`. Для этого программе необходимо установить связь с базой и создать сущности, связанные с ее таблицами. В рамках интерфейса `LINQ to SQL` указанные действия выполняются с помощью классов `DataContext` (отвечает за связь с базой данных) и `Table<T>` (отвечает

за доступ к таблице базы данных с записями типа T). Эти классы определены в пространстве имен System.Data.Linq, поэтому в начало файла LinqObj71.cs необходимо добавить директиву

```
using System.Data.Linq;
```

Связь с базой данных и ее двумя таблицами обеспечивается тремя операторами:

```
DataContext db = new DataContext(@"c:\PT4Work\LocalDB.sdf");  
Table<Customer> customers = db.GetTable<Customer>();  
Table<Discount> discounts = db.GetTable<Discount>();
```

Обратите внимание на то, что для установки связи с локальной базой данных, хранящейся в файле sdf, в конструкторе DataContext достаточно указать *полный путь к этому файлу* (в приведенном фрагменте предполагается, что файл LocalDB.sdf находится в каталоге c:\PT4Work).

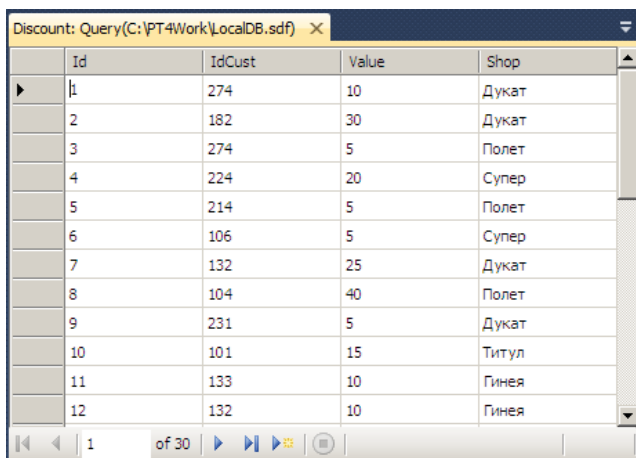
Примечание. Символ @, указанный перед строкой с именем файла, приводит к тому, что все символы, входящие в эту строку, интерпретируются буквально и не связываются с управляющими последовательностями. При отсутствии данного символа все символы «\» (обратная косая черта) требовалось бы экранировать, указывая их дважды ("c:\\PT4Work\\LocalDB.sdf"), поскольку в противном случае они интерпретировались бы как начало управляющей последовательности.

Имея доступ к таблицам базы данных, мы можем дополнять их новыми записями. Для этого в классе Table<T> предусмотрены два метода: InsertOnSubmit и InsertAllOnSubmit. Первый из них добавляет в таблицу одну запись, второй – последовательность записей. Поскольку мы уже сформировали последовательности записей A и C, нам удобнее воспользоваться методом InsertAllOnSubmit. После вызовов любых методов, изменяющих содержимое базы данных, необходимо *актуализировать* эти изменения в базе данных, вызвав метод SubmitChanges класса DataContext. Итак, для заполнения таблиц базы данных LocalDB нам осталось добавить в конец метода Solve три оператора:

```
customers.InsertAllOnSubmit(A);  
discounts.InsertAllOnSubmit(C);  
db.SubmitChanges();
```

При запуске полученного варианта программы в окне задачника появится сообщение о том, что были введены не все исходные данные (поскольку мы не прочли имя файла результатов), однако наша ближайшая цель – заполнение таблиц базы данных LocalDB – будет достигнута.

Если теперь, используя окно **Server Explorer**, отобразить, например, содержимое таблицы Discount (требуемые для этого действия описаны в начале пункта), то на экране мы увидим набор данных, подобный приведенному на рис. 60. В этой таблице значения поля Id представляют собой последовательные целые числа, начиная от 1, а значения остальных полей соответствуют данным, полученным из исходного текстового файла, созданного задачником.



| | Id | IdCust | Value | Shop |
|---|----|--------|-------|--------|
| ▶ | 1 | 274 | 10 | Дукат |
| | 2 | 182 | 30 | Дукат |
| | 3 | 274 | 5 | Полет |
| | 4 | 224 | 20 | Супер |
| | 5 | 214 | 5 | Полет |
| | 6 | 106 | 5 | Супер |
| | 7 | 132 | 25 | Дукат |
| | 8 | 104 | 40 | Полет |
| | 9 | 231 | 5 | Дукат |
| | 10 | 101 | 15 | Титул |
| | 11 | 133 | 10 | Гинейя |
| | 12 | 132 | 10 | Гинейя |

Рис. 60. Просмотр заполненной таблицы Discount

При повторном запуске программы мы получим сообщение об исключении типа `SqlException`. Это связано с тем обстоятельством, что при добавлении в базу данных новых записей, сгенерированных при очередном запуске программы, возникнет ситуация, когда добавляемая запись будет иметь *то же самое значение ключевого поля*, что и одна из уже имеющихся записей. Поскольку в таблице не могут находиться записи с одинаковым значением ключевого поля, попытка добавления таких записей приведет к возбуждению исключения.

Чтобы избежать подобной ошибки, достаточно перед занесением в таблицы новых данных *удалять из них прежнее содержимое*.

Для выполнения этого действия, как и для вставки новых записей, в классе `Table<T>` предусмотрены два метода: `DeleteOnSubmit` и `DeleteAllOnSubmit` (первый удаляет одну запись, второй – последовательность записей). Добавим в программу перед операторами вставки записей в таблицы следующие четыре оператора:

```
discounts.DeleteAllOnSubmit(discounts);  
db.SubmitChanges();  
customers.DeleteAllOnSubmit(customers);  
db.SubmitChanges();
```

Обратите внимание на то, что вначале необходимо очистить таблицу `Discount`, а затем – таблицу `Customer`. Порядок удаления данных обусловлен наличием связи между таблицами: запись в таблице `Customer` нельзя удалять, пока в таблице `Discount` остаются связанные с ней записи. По этой же причине необходимо вызывать метод `SubmitChanges` после каждого действия, связанного с удалением.

Теперь при каждом запуске программы в базе `LocalDB` будет обновляться набор данных.

Воспользуемся полученной программой, для того чтобы показать «в действии» технологию `LINQ to SQL`. Для этого решим задачу `LinqObj71`, предполагая, что исходные данные, требующие обработки, содержатся в базе `LocalDB`.

Приведем завершающую часть формулировки задачи `LinqObj71`.

Для каждого магазина определить потребителей, имеющих максимальную скидку в этом магазине (вначале выводится название магазина, затем код потребителя, его год рождения и значение максимальной скидки). Если для некоторого магазина имеется несколько потребителей с максимальной скидкой, то вывести данные о потребителе с минимальным кодом. Сведения о каждом магазине выводить на новой строке и упорядочивать по названиям магазинов в алфавитном порядке.

Используя объекты `customers` и `discounts`, обеспечивающие связь с таблицами `Customer` и `Discount`, получаем следующий вариант решения, который необходимо добавить в конец метода `Solve`:

```
var r = discounts.Join(customers, e1 => e1.IdCust, e2 => e2.Id,  
    (e1, e2) => new { e1.Shop, e1.Value, e1.IdCust, e2.Year })  
    .GroupBy(e => e.Shop, (k, ee) =>  
        new { k, d = ee.OrderByDescending(e => e.Value)  
            .ThenBy(e => e.IdCust).First() })
```

```
.OrderBy(e => e.k)
.Select(e => e.k + " " + e.d.IdCust + " " +
    e.d.Year + " " + e.d.Value);
File.WriteAllLines(GetString(), r.ToArray(), Encoding.Default);
```

После девяти тестовых запусков полученной программы будет выведено сообщение о том, что задание выполнено.

Мы видим, что при обработке таблиц баз данных можно использовать те же методы LINQ, что и при обработке локальных последовательностей, хотя механизм выполнения методов LINQ будет другим; в частности, результирующая последовательность *r* будет иметь тип `IQueryable<string>`, а все лямбда-выражения будут интерпретироваться как *деревья выражений* (см. п. 9.2).

Интересно отметить, что программа по-прежнему будет выводить правильные результаты, если в первой строке приведенного выше фрагмента заменить обращения к объектам-таблицам `customers` и `discounts` на обращения к локальным последовательностям *A* и *C*, которые не связаны с базой LocalDB, но содержат тот же набор исходных данных:

```
var r = C.Join(A, e1 => e1.IdCust, e2 => e2.Id,
...

```

При этом, однако, все лямбда-выражения будут интерпретироваться как *делегаты*, а последовательность *r* будет иметь тип `IEnumerable<string>` (в чем можно убедиться с помощью контекстных подсказок).

9.4.3. Создание и настройка базы данных, основанной на службах

Для определения объектной модели, описанной в предыдущем пункте, нам потребовалось ввести программный код достаточно большого размера. При этом объектная модель определяла лишь структуру таблиц базы данных, не учитывая связи между ними.

В среду программирования Visual Studio встроены средства, позволяющие *автоматизировать* процесс создания объектной модели как для интерфейса LINQ to SQL, так и для интерфейса LINQ to Entities. Эти средства освобождают программиста от необходимости набирать код вручную и позволяют получить объектную модель, обладающую дополнительными возможностями.

Однако выполнить автоматическую генерацию объектной модели, связанной с интерфейсом LINQ to SQL, можно лишь для баз данных, основанных на службах. Поэтому, для того чтобы привести пример автоматически сгенерированной объектной модели и рассмотреть особенности ее использования, нам необходимо создать и настроить соответствующий вариант базы данных. Все действия будем выполнять в том же проекте, связанном с заданием LinqObj71, который рассматривался в двух предыдущих пунктах (это, в частности, продемонстрирует общность приемов работы с обоими вариантами баз данных при использовании интерфейса LINQ to SQL).

Для создания базы данных, основанной на службах, после выполнения команды меню **Project** ⇒ **Add New Item...** необходимо выбрать в появившемся диалоговом окне вариант **Service-based Database**, указать имя создаваемой базы данных, например ServiceDB.mdf, и нажать кнопку **Add**. Если после этого на экране появится окно **Data Source Configuration Wizard**, то можно сразу закрыть его, нажав кнопку **Cancel**.

В результате к проекту будет добавлен файл ServiceDB.mdf; он сразу будет отображен и в окне **Solution Explorer**, и в окне **Server Explorer**. Файл ServiceDB.mdf, как и ранее созданный файл LocalDB.sdf, будет размещен в том же каталоге, в котором содержатся остальные файлы проекта.

Настроим конфигурацию созданной базы данных. Для создания новой таблицы достаточно вызвать контекстное меню подпункта **Tables** для базы данных ServiceDB.mdf в окне **Server Explorer** и выполнить в этом меню команду **Add New Table** (Добавить новую таблицу). При этом в редакторе появится новая вкладка с именем, начинающимся с текста dbo.Table1 (dbo.Table в Visual Studio 2012). Вкладка будет содержать таблицу, позволяющую задать имена и свойства полей создаваемой таблицы базы данных. В этой таблице достаточно заполнить поля **Column Name** (Имя) и **Data Type** (Тип данных) (см. табл. 9.3, а также рис. 61).

Таблица 9.3. Поля таблицы Customer базы данных ServiceDB.mdf

| Column Name | Data Type |
|-------------|--------------|
| Id | int |
| Year | int |
| Street | nvarchar(50) |



| | Column Name | Data Type | Allow Nulls |
|---|-------------|--------------|--------------------------|
|  | Id | int | <input type="checkbox"/> |
| | Year | int | <input type="checkbox"/> |
| | Street | nvarchar(50) | <input type="checkbox"/> |
|  | | | <input type="checkbox"/> |

Рис. 61. Определение полей таблицы Customer

Заметим, что в Visual Studio 2012 поле Id добавится в таблицу автоматически, причем оно уже будет иметь метку ключевого поля (в виде изображения ключа, указанного слева от имени поля). В версиях 2008 и 2010 необходимо выполнить специальные действия, для того чтобы пометить поле Id как ключевое: вызвать контекстное меню для данного поля (щелкнув на нем правой кнопкой мыши) и выполнить команду **Set Primary Key**.

Созданной таблице надо присвоить имя Customer. Для этого в версиях 2008 и 2010 достаточно указать требуемое имя в окне **Properties** рядом со свойством **(Name)**.

В версии 2012 соответствующее свойство **(Имя)** доступно только для чтения, однако требуемое имя можно указать в первой строке SQL-запроса, который приводится в нижней части вкладки с настройками создаваемой таблицы (введенное имя выделено полужирным шрифтом):

```
CREATE TABLE [dbo].[Customer]
```

После указанных действий в версиях 2008 и 2010 достаточно сохранить созданную таблицу (если при этом появится окно **Save Change Script** с запросом о сохранении протокола внесенных изменений, то в этом окне можно выбрать вариант **No**). В версии 2012 следует нажать кнопку **Обновить**, расположенную в левом верхнем углу вкладки с настройками таблицы, и в появившемся окне **Предварительный просмотр обновлений базы данных** нажать кнопку **Обновить базу данных** (заметим, что в этом окне в качестве действий пользователя должно быть указано «Создать [dbo].[Customer] (Таблица)»).

В результате имя созданной таблицы Customer появится в разделе **Tables** для базы данных ServiceDB в окне **Server Explorer** (если имя не появится, то следует выполнить дополнительное обновление базы данных; для этого достаточно воспользоваться командой **Refresh** (Обновить) из контекстного меню этой базы данных в окне **Server Explorer**).

Аналогичными действиями надо добавить в базу данных таблицу Discount, настроив ее поля так, как указано в табл. 9.4. Поле Id необходимо сделать ключевым.

**Таблица 9.4. Поля таблицы Discount
базы данных ServiceDB.mdf**

| Column Name | Data Type |
|-------------|--------------|
| Id | int |
| IdCust | int |
| YearValue | int |
| Shop | nvarchar(50) |

Завершая определение структуры базы данных, настроим связь между созданными таблицами по полям Customer.Id – Discount.Id-Cust. Указанная настройка выполняется для той таблицы, которая в устанавливаемой связи соответствует варианту «ко многим», то есть в нашем случае – для таблицы Discount. Если в данный момент на экране отсутствует вкладка с определением свойств таблицы Discount, то надо вызвать контекстное меню этой таблицы в окне **Server Explorer** и выполнить его команду **Open Table Definition** (Открыть определение таблицы).

Действия по настройке связи, как и действия по заданию имени таблицы, выполняются по-разному в версиях 2008–2010 и 2012.

В версиях 2008 и 2010 достаточно вызвать контекстное меню вкладки с определением таблицы Discount и выполнить его команду **Relationships....** На экране появится окно настройки связей **Foreign Key Relationships**, в котором для создания новой связи надо нажать кнопку **Add**. В появившемся списке свойств созданной связи (рис. 62) надо выделить раздел **Tables And Columns Specification**, щелкнуть мышью на кнопке с многоточием, расположенной справа от названия этого раздела, и в появившемся диалоговом окне **Tables and Columns** задать имена таблиц и полей (см. табл. 9.5, а также рис. 63). При этом имя связи, которое указывается в верхней части этого окна, примет вид FK_Discount_Customer.

Таблица 9.5. Настройка связи между таблицами базы данных ServiceDB.mdf

| Primary key table | Foreign key table |
|-------------------|-------------------|
| Customer | Discount |
| Id | IdCust |

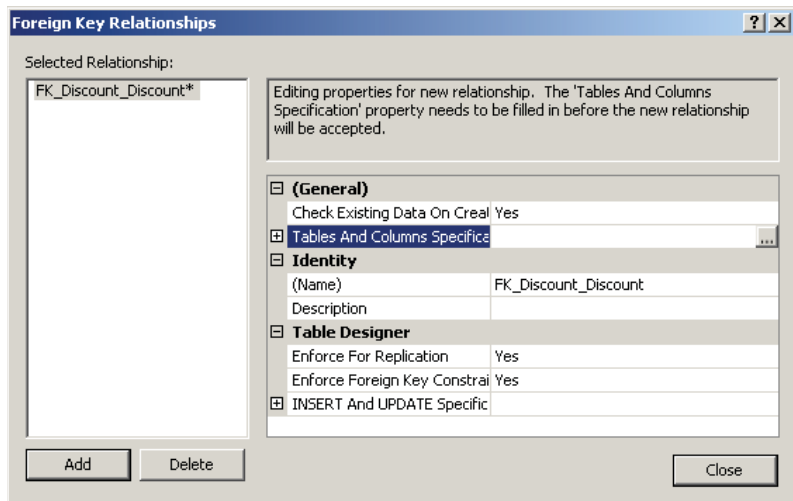
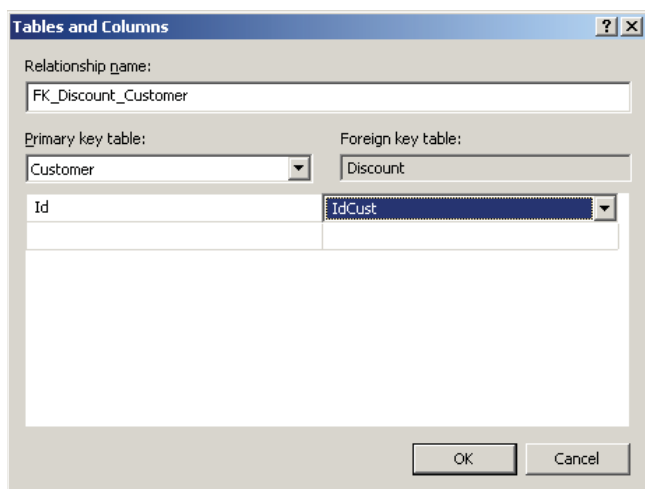


Рис. 62. Настройка связи между таблицами: окно свойств

Рис. 63. Настройка связи между таблицами:
окно параметров связи

Нажатие кнопки **OK** установит требуемую связь и закроет окно **Tables and Columns**, после чего диалоговое окно **Foreign Key Relationships** можно будет закрыть, нажав в нем кнопку **Close**.

В версии 2012 следует вызвать контекстное меню раздела **Внешние ключи**, расположенного в правой части вкладки с настройками таблицы Discount, и выполнить его команду **Добавить новый внешний ключ**. При этом в SQL-запрос, связанный с формированием таблицы Discount и расположенный в нижней части данной вкладки, будет добавлена новая строка – заготовка для создаваемой связи, имеющая следующий вид:

```
CONSTRAINT [FK_Discount_ToTable] FOREIGN KEY ([COLUMN])  
REFERENCES [ToTable]([ToTableColumn])
```

В эту заготовку надо ввести имена таблиц и полей. Кроме того, можно откорректировать имя создаваемой связи. В результате строка SQL-запроса должна принять следующий вид (измененные фрагменты строки выделены полужирным шрифтом):

```
CONSTRAINT [FK_Discount_Customer] FOREIGN KEY ([IdCust])  
REFERENCES [Customer]([Id])
```

После изменения текста SQL-запроса следует выполнить обновление настроек таблицы Discount, нажав кнопку **Обновить** на вкладке с настройками этой таблицы и кнопку **Обновить базу данных** в появившемся окне просмотра обновлений базы данных (описание обновления должно иметь вид «Создать [dbo].[FK_Discount_Customer] (Внешний ключ)»).

На этом настройка конфигурации базы данных ServiceDB.mdf завершается.

Несмотря на различия в способе создания локальных баз данных и баз данных, основанных на службах, программный код, обеспечивающий их обработку методами интерфейса LINQ to SQL, практически не отличается. Имеется единственное отличие, связанное с указанием строки подключения к базе данных. Ранее было отмечено, что для локальных баз данных в качестве строки подключения достаточно указывать полный путь к соответствующему sdf-файлу. Подобный способ подключения можно использовать и для mdf-файлов, связанных с базами данных, основанными на службах, однако он не всегда приводит к успешному подключению. Более надежным является использование строки подключения, автоматически формируемой при создании базы данных и указываемой в ее свойстве **ConnectionString** (Строка подключения) в окне **Properties**. Содержащийся в этом свойстве текст достаточно вставить в качестве строкового пара-

метра в конструктор DataContext. Приведем пример использования подобной строки подключения (конкретный вид строки зависит не только от имени mdf-файла с базой данных, но и от конфигурации SQL-сервера, а также от используемой версии Visual Studio):

```
DataContext db = new DataContext(@"Data Source=.\SQLEXPRESS;  
AttachDbFilename=|DataDirectory|\ServiceDB.mdf;  
Integrated Security=True;User Instance=True");
```

Поскольку строка подключения представлена в виде *буквальной* строки (благодаря наличию перед ней символа @), ее содержимое допустимо разбивать на несколько строк кода.

Внеся это *единственное* изменение в программу, разработанную в п. 9.4.2, и запустив ее, мы получим сообщение о верном решении, несмотря на то что теперь программа работает не с локальной базой LocalDB.sdf, а с базой ServiceDB.mdf, основанной на службах. Таким образом, все реализованные в программе действия, а также входящая в нее простейшая объектная модель могут использоваться для обоих вариантов баз данных.

9.4.4. Автоматическая генерация объектной модели базы данных и особенности ее использования

Имея базу данных ServiceDB.mdf, мы можем сгенерировать для нее объектную модель, используя встроенные средства среды Visual Studio. Полученная модель позволит упростить действия, связанные с подключением к базе и ее обработкой.

Автоматически созданная объектная модель оформляется в виде отдельного компонента проекта, поэтому для ее создания надо выполнить команду меню **Project** ⇒ **Add New Item...** и выбрать в появившемся диалоговом окне вариант **LINQ to SQL Classes** (Классы LINQ to SQL). По умолчанию создаваемый компонент получает имя DataClasses1.dbml. Изменим имя этого компонента на DB.dbml и нажмем кнопку **Add**.

В результате в разделе редактора кода Visual Studio появится вкладка с именем DB.dbml. Текст в этой вкладке описывает действия, требуемые для определения тех баз данных и их таблиц, для которых надо создать объектную модель. Эти действия состоят в перетаскивании с помощью мыши имен нужных таблиц требуемой базы

данных из окна Server Explorer на вкладку DB.dbml. Заметим, что созданный компонент не поддерживает работу с локальными базами данных, поэтому при попытке перетаскивания на него таблиц из базы LocalDB появится окно с сообщением об ошибке.

После перетаскивания таблиц Customer и Discount базы данных ServiceDB на вкладку DB.dbml содержимое вкладки примет вид, приведенный на рис. 64.

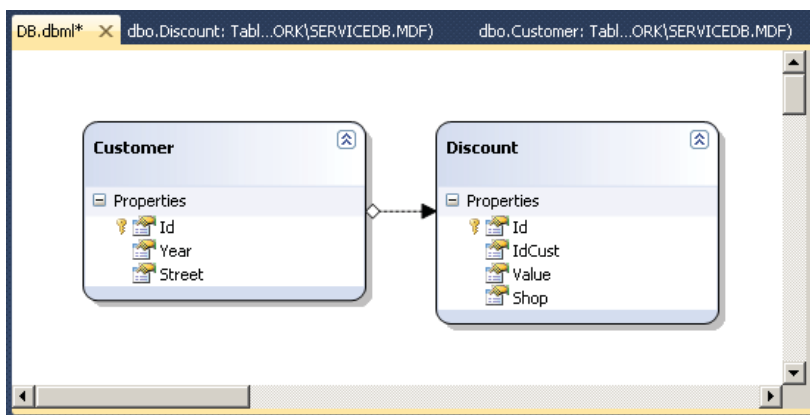


Рис. 64. Графическое представление созданной объектной модели

При генерации объектной модели должны учитываться связи между таблицами; эти связи отображаются на рисунке в виде стрелок (стрелка идет от таблицы, соответствующей отношению «один», к таблице, соответствующей отношению «ко многим»).

Если на схеме объектной модели не отображается связь между таблицами, то ее можно добавить, выполнив следующие действия. Используя контекстное меню вкладки DB.dbml (достаточно щелкнуть правой кнопкой мыши в любой точке области вкладки вне содержащихся в ней изображений), выполните команду **Add ⇒ Association** (Добавить ⇒ Связь). На экране появится окно **Association Editor** (Редактор ассоциаций), в верхней части которого надо указать имена родительского класса (**Parent Class**) и дочернего класса (**Child Class**); в нашем случае Customer и Discount соответственно (в качестве родительского класса указывается тот класс, поле связи которого является главным ключом).

После выбора классов следует задать их поля связи Id и IdCust; они указываются в том же окне в разделе **Association Properties** (см. рис. 65).

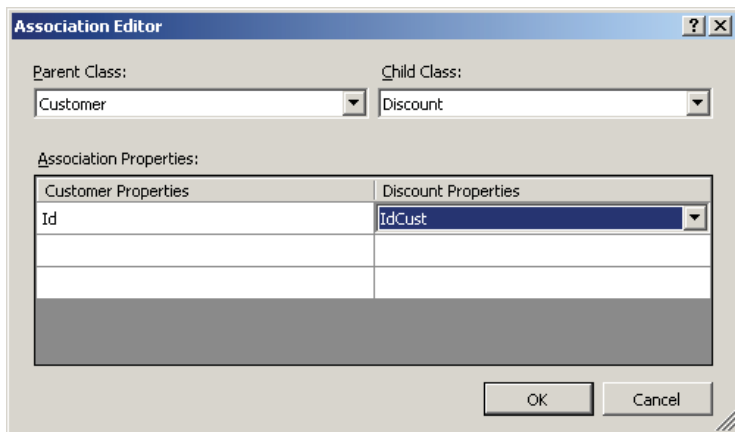


Рис. 65. Окно настройки свойств связи для объектной модели

Для завершения определения связи надо закрыть окно, нажав кнопку **ОК**. В результате между изображениями объектов Customer и Discount должно появиться изображение связи.

Заметим, что свойства установленной связи можно редактировать; для этого достаточно щелкнуть на изображении связи правой кнопкой мыши и выполнить команду **Edit Association** (Изменить связь) из появившегося контекстного меню.

Код, определяющий созданную объектную модель, содержится в файле DB.Designer.cs. Все классы, входящие в эту модель, определены в пространстве имен, совпадающем с именем проекта (в нашем случае LinqObj71).

Помимо классов Customer и Discount, связанных с отдельными таблицами базы данных, в созданную объектную модель входит класс, связанный с базой данных в целом. Некоторые его свойства можно отобразить в окне **Properties**, если выполнить щелчок мышью на вкладке DB.dbml вне содержащихся в ней изображений (щелчок на изображении приводит к отображению в окне **Properties** свойств объекта, связанного с выбранным изображением – таблицей или связью). В частности, в окне свойств указывается имя этого класса; оно формируется на основе имени созданного компонента и

в нашем случае имеет вид `DBDataContext`. Используя окно **Properties**, это имя можно изменить.

Каждый из классов `Customer` и `Discount` содержит свойства, связанные со всеми полями соответствующей таблицы, однако в дополнение к ним он включает свойства, отражающие *связь между таблицами*. В классе `Customer` таким свойством является `Discounts` (типа `EntitySet<Discount>`); оно содержит последовательность всех записей таблицы `Discount`, связанных с данной записью таблицы `Customer` (то есть набор всех скидок, связанных с данным потребителем); в классе `Discount` имеется дополнительное свойство `Customer` (типа `Customer`), определяющее ту запись из таблицы `Customer`, которая связана с данной записью таблицы `Discount`.

Класс `DBDataContext` является потомком класса `DataContext` и содержит конструкторы, упрощающие подключение к базе данных (в частности, предусмотрен конструктор без параметров, не требующий указания строки подключения). Кроме того, в этом классе предусмотрены свойства `Customers` (типа `Table<Customer>`) и `Discounts` (типа `Table<Discount>`), обеспечивающие доступ к таблицам `Customer` и `Discount` соответственно.

Примечание. Имена свойств класса `DBDataContext`, предназначенных для доступа к таблицам, имеют форму множественного числа (то есть оканчиваются буквой «s»). Эту же форму имеет вспомогательное свойство `Discounts` класса `Customer`, содержащее все скидки, связанные с данным потребителем. Подобные модификации имен выполняются средой Visual Studio автоматически. Следует, однако, отметить, что в версии 2012 форма множественного числа не используется.

Для того чтобы созданная объектная модель была доступна в функции `Solve`, содержащей решение задачи, в файл `LinqObj71.cs` необходимо внести следующие изменения:

- ❑ добавить в начало файла `LinqObj71.cs` директиву `using` для пространства имен `LinqObj71`:

```
using LinqObj71;
```

- ❑ удалить или закомментировать описания классов `Customer` и `Discount`, содержащиеся в файле `LinqObj71.cs` и относящиеся к предыдущей (простейшей) объектной модели.

Полученная программа будет по-прежнему правильно решать поставленную задачу. Однако в ней будет использоваться только та

часть возможностей новой объектной модели, которая имелась и у простейшей модели, созданной нами вручную.

Дополнительные возможности новой объектной модели позволяют (1) упростить действия по подключению к базе данных, (2) избежать введения дополнительных переменных, связанных с таблицами, (3) получить более простую цепочку запросов LINQ, обеспечивающую решение поставленной задачи. Приведем новый вариант завершающей части функции Solve, начиная от действий по подключению к базе данных, и затем прокомментируем сделанные изменения:

```
DBDataContext db = new DBDataContext();           // (1)

db.Discounts.DeleteAllOnSubmit(db.Discounts);    // (2)
db.SubmitChanges();
db.Customers.DeleteAllOnSubmit(db.Customers);
db.SubmitChanges();
db.Customers.InsertAllOnSubmit(A);
db.Discounts.InsertAllOnSubmit(C);
db.SubmitChanges();

var r = db.Discounts.GroupBy(e => e.Shop,          // (3)
    (k, ee) =>
        new { k, d = ee.OrderByDescending(e => e.Value)
            .ThenBy(e => e.IdCust)
            .First() })
    .OrderBy(e => e.k)
    .Select(e => e.k + " " + e.d.IdCust + " " +
        e.d.Customer.Year + " " + e.d.Value);
File.WriteAllLines(GetString(), r.ToArray(), Encoding.Default);
```

Для того чтобы в дальнейшем можно было использовать дополнительные средства объектной модели, относящиеся к базе данных в целом, подключение к базе данных (см. оператор, помеченный комментарием (1)) выполняется с помощью специализированного класса `DBDataContext`, в отличие от ранее использовавшегося класса `DataContext` (см. вариант подключения, приведенный в конце п. 9.4.3). В новом варианте программы вызывается конструктор `DBDataContext` *без параметров*, таким образом, строка подключения в тексте программы не указывается. Это оказывается возможным благодаря тому, что значение строки подключения хранится в классе `DBDataContext` и автоматически используется при выполнении его конструктора без параметров.

Фрагмент программы, начало которого помечено комментарием (2), обеспечивает очистку таблиц базы и их заполнение новыми данными. Отличие этого фрагмента от аналогичного фрагмента преды-

дущего варианта программы (см. п. 9.4.2) состоит в том, что в нем не используются вспомогательные переменные `customers` и `discounts`, связанные с соответствующими таблицами базы данных. Вместо них выполняется обращение к свойствам `Customers` и `Discounts` объекта `db` типа `DBDataContext`.

В цепочке запросов, помеченной комментарием (3) и обеспечивающей получение требуемого набора данных, отсутствует запрос `Join`, с которого начиналась цепочка запросов в предыдущем варианте программы (см. п. 9.4.2). Запрос `Join` требовался для того, чтобы установить связь между набором скидок и набором потребителей и благодаря этому получить для каждой скидки полную информацию о связанном с ней потребителе. В данном случае подобной необходимости нет, так как указанная связь обеспечивается средствами, встроенными в объектную модель: для получения информации о потребителе, связанном с некоторой скидкой (типа `Discount`), достаточно обратиться к ее свойству `Customer`. Этим способом мы получаем информацию о годе рождения потребителя, связанного со скидкой `d` в запросе `Select`, завершающем цепочку запросов: `e.d.Customer.Year`. Таким образом, в цепочке запросов фактически производится обращение только к таблице `Discount` (это обращение, как и в операторах фрагмента (2), выполняется с помощью выражения `db.Discounts`).

При использовании нового варианта программы в версии Visual Studio 2012 следует изменить имена свойств `Customers` и `Discounts` объекта `db`, удалив из них завершающую букву «s» (см. предыдущее примечание).

Таким образом, применение автоматически сгенерированной объектной модели базы данных позволило нам получить более простой вариант решения задачи. Следует подчеркнуть, что этот вариант можно использовать только *совместно с базой данных*. Если попытаться изменить цепочку запросов, указав в ее начале вместо последовательности `db.Discounts`, связанной с таблицей базы данных, имя *локальной* последовательности *C* (ср. с аналогичным действием, описанным в конце п. 9.4.2), то программа успешно откомпилируется, однако на этапе выполнения будет выведено сообщение об ошибке. Это объясняется тем, что элементы сформированной последовательности *C* *не содержат дополнительных данных о связанных с ними потребителях* (свойство `Customer` у всех этих элементов равно `null`). Указанная связь устанавливается только при формировании объекта `db`; она обеспечивается благодаря наличию связи между соответствующими таблицами базы данных.



Литература

1. *Албахари Дж., Албахари Б.* С# 3.0. Справочник. – СПб.: БХВ-Петербург, 2009. – 944 с.
2. *Албахари Дж., Албахари Б.* С# 5.0. Справочник. Полное описание языка. – М.: Вильямс, 2013. – 1054 с.
3. *Албахари Дж., Албахари Б.* LINQ. Карманный справочник. – СПб.: БХВ-Петербург, 2012. – 240 с.
4. *Раттиц-мл. Дж.* LINQ: язык интегрированных запросов в С# 2008 для профессионалов. – М.: Вильямс, 2008. – 560 с.
5. *Фримен А., Раттиц-мл. Дж.* LINQ. Язык интегрированных запросов в С# 2010 для профессионалов. – М.: Вильямс, 2011. – 656 с.



Предметный указатель

Методы, определенные в классе `System.Linq.Enumerable` и реализующие базовый набор запросов LINQ, описываются в указателе как *методы LINQ to Objects*. Методы, определенные в классе `Extensions` из пространства имен `System.Xml.Linq` и реализующие дополнительный набор запросов интерфейса LINQ to XML, описываются в указателе как *методы LINQ to XML*.

Символы

??, операция, 127, 251

+, операция, переопределение
для `XNamespace`, 259, 272

А

`Action< >`, семейство

обобщенных делегатов, 280

`Add`, метод классов `XDocument`
и `XElement`, 240

`AddAfterSelf`, метод класса
`XNode`, 240

`AddBeforeSelf`, метод класса
`XNode`, 240

`AddFirst`, метод классов
`XDocument`
и `XElement`, 240, 254

`Aggregate`, метод LINQ
to Objects, 18, 130, 132, 201,
252, 293

`All`, метод LINQ to Objects, 211

`Ancestors`, метод класса

`XNode`, 232

`Any`, метод LINQ
to Objects, 211, 243

`as`, операция, 226

`Attribute`, метод класса
`XElement`, 231, 233

`Attributes`, метод класса
`XElement` и метод LINQ
to XML, 231, 250

`Average`, метод LINQ
to Objects, 18, 130, 198

С

`Column`, атрибут, 299

`Concat`, метод LINQ
to Objects, 25, 153, 192

`Contains`, метод LINQ
to Objects, 211

`Count`, метод LINQ
to Objects, 18, 129, 275

CultureInfo, класс, 200
CurrentCulture, свойство класса
Thread, 202

D

DataContext, класс LINQ
to SQL, 301
DateTime, структура, 267
Default, свойство класса
Encoding, 187, 191, 222
DefaultIfEmpty, метод LINQ
to Objects, 25, 170, 196, 201,
234, 235
DeleteAllOnSubmit, метод
класса Table< >, 304
DeleteOnSubmit, метод класса
Table< >, 304
DescendantNodes, метод
классов XDocument
и XElement и метод LINQ
to XML, 231
DescendantNodesAndSelf, метод
класса XElement, 231
Descendants, метод классов
XDocument и XElement
и метод LINQ
to XML, 231, 233
DescendantsAndSelf, метод
класса XElement, 231
Dictionary< , >, класс, 164
Distinct, метод LINQ
to Objects, 20

E

Element, метод классов
XDocument
и XElement, 231, 250
ElementAt, метод LINQ
to Objects, 123, 293

ElementAtOrDefault, метод
LINQ to Objects, 123, 126, 293
Elements, метод классов
XDocument и XElement
и метод LINQ to XML, 231
EndsWith, метод класса
string, 124
Enumerable, класс LINQ
to Objects, 10, 113, 130
Except, метод LINQ
to Objects, 20, 135
Expression, класс LINQ
to SQL/Entities, 292
Extensions, класс LINQ
to XML, 10

F

First, метод LINQ
to Objects, 18, 123
FirstAttribute, свойство класса
XElement, 241
FirstNode, свойство классов
XDocument и XElement, 241
FirstOrDefault, метод LINQ
to Objects, 18, 123, 126,
211, 241
Format, метод класса
string, 201
FormatException,
исключение, 251
from, конструкция выражения
запроса, 161
Func< >, семейство
обобщенных
делегатов, 279, 291

G

group, конструкция выражения
запроса, 177, 193

GroupBy, метод LINQ to
Objects, 25, 173, 175, 190, 196
GroupJoin, метод LINQ to
Objects, 25, 154, 167, 210, 274

H

HasAttributes, свойство класса
XElement, 243
HasElements, свойство класса
XElement, 243

I

Comparable, интерфейс, 144
Comparer< >,
интерфейс, 145
IEnumerable< >,
интерфейс, 10, 305
IGrouping< >,
интерфейс, 173, 193
IndexOutOfRangeException,
исключение, 124
InsertAllOnSubmit, метод
класса Table< >, 302
InsertOnSubmit, метод класса
Table< >, 302
Intersect, метод LINQ
to Objects, 20, 135
into
конструкция выражения
запроса, 178, 193
продолжение конструкции
join, 171
InvalidOperationException,
исключение, 125
IQueryable< >,
интерфейс, 10, 290, 305
is, операция, 239
IsDigit, метод структуры
char, 156

IsEmpty, свойство класса
XElement, 243

J

join, конструкция выражения
запроса, 171, 216
Join, метод LINQ to Objects, 25,
154, 167, 209

K

Key, свойство интерфейса
IGrouping< >, 173, 193

L

Last, метод LINQ to Objects, 18,
123, 156
LastAttribute, свойство класса
XElement, 241
LastNode, свойство классов
XDocument и XElement, 241
LastOrDefault, метод LINQ
to Objects, 18, 123, 126
let, конструкция выражения
запроса, 171, 178, 193
LINQ to Entities (Entity
Framework), 288
ограничения
на запросы, 293
LINQ To Objects, 113
LINQ to SQL, 288
ограничения
на запросы, 293
LINQ to XML, 218
List< >, класс, 164
Load, метод классов
XDocument и XElement, 232
LoadOptions,
перечисление, 246

LocalName, свойство класса
XName, 232, 259

M

Max, метод LINQ to Objects,
18, 130, 196, 234
Min, метод LINQ
to Objects, 18, 130

N

Name, свойство классов
XAttribute и XElement, 247
Namespace, свойство класса
XName, 259
NamespaceName
свойство класса XName, 259
свойство класса
XNamespace, 259
Nodes, метод классов
XDocument и XElement
и метод LINQ to XML, 231
Nullable-тип, 251, 300

O

OfType< >, метод LINQ
to Objects, 238
on, продолжение конструкции
join, 171
orderby, конструкция
выражения запроса, 162
OrderBy, метод LINQ
to Objects, 20, 144, 159, 168
OrderByDescending, метод
LINQ to Objects, 20, 144

P

ParallelEnumerable, класс
PLINQ, 11

ParallelQuery< >, класс
PLINQ, 11

Parse

метод классов XDocument
и XElement, 233
метод структуры
int, 176, 230

PLINQ (Parallel LINQ), 11

Programming Taskbook
for LINQ, 15

GetBool, метод ввода, 120
GetChar, метод ввода, 120
GetDouble, метод
ввода, 120
GetEnumerableInt, метод
ввода, 17, 122
GetEnumerableString, метод
ввода, 17, 122, 138
GetInt, метод ввода, 120, 138
GetString, метод
ввода, 120, 187
Put, метод вывода, 17, 125
Show, метод отладочной
печати, 17, 140, 141, 151,
156, 188
ShowLine, метод отладочной
печати, 140
демонстрационный
режим, 118
отображение файловых
данных, 184
протокол выполнения
задания, 128
создание
проекта-заготовки, 114

Q

Queryable, класс LINQ
to SQL/Entities, 10

R

Range, метод LINQ to Objects,
18, 130, 135, 274
ReadAllLines, метод класса
File, 187, 222
ReadLines, метод класса
File, 187
Remove, метод классов
XAttribute и XElement и метод
LINQ to XML, 237
RemoveAll, метод класса
XElement, 237
RemoveAttributes, метод класса
XElement, 237
RemoveNodes, метод классов
XDocument и XElement, 237
ReplaceAll, метод класса
XElement, 247
ReplaceAttributes, метод класса
XElement, 247
ReplaceNodes, метод классов
XDocument
и XElement, 247, 266
ReplaceWith, метод класса
XNode, 247
Reverse, метод LINQ
to Objects, 20, 139, 293
Root, свойство класса
XDocument, 231
Round, метод класса
Math, 276

S

Save, метод классов XDocument
и XElement, 223
SaveOptions,
перечисление, 224, 246
select, конструкция выражения
запроса, 161, 193

Select, метод LINQ to Objects,
23, 147, 157, 167, 234, 293
SelectMany, метод LINQ
to Objects, 23, 147, 150, 152,
157, 168, 293
SequenceEqual, метод LINQ
to Objects, 211
SetAttributeValue, метод класса
XElement, 248
SetElementValue, метод класса
XElement, 248
SetValue, метод классов
XAttribute и XElement, 247
Single, метод LINQ
to Objects, 18, 123
SingleOrDefault, метод LINQ
to Objects, 18, 123, 126
Skip, метод LINQ
to Objects, 20, 139, 152
SkipWhile, метод LINQ
to Objects, 20, 139, 293
Split, метод класса
string, 188, 270, 286
SQL-запрос, 292
StartsWith, метод класса
string, 225
StringSplitOptions,
перечисление, 286
SubmitChanges, метод класса
DataContext, 302
Substring, метод класса
string, 270
Sum, метод LINQ
to Objects, 18, 130, 175, 275

T

Table, атрибут, 298
Table< >, класс LINQ
to SQL, 301

Take, метод LINQ
to Objects, 20, 139, 152
TakeWhile, метод LINQ
to Objects, 20, 139, 293
ThenBy, метод LINQ
to Objects, 20, 144
ThenByDescending, метод LINQ
to Objects, 20, 144, 159
this, модификатор параметра
в методе расширения, 285
TimeSpan, структура, 251
ToArray, метод LINQ
to Objects, 164, 191
ToDictionary, метод LINQ
to Objects, 164
ToList, метод LINQ
to Objects, 164
ToString
 вариант метода с двумя
 параметрами, 201
 метод классов XDocument
 и XElement, 224
try-catch, конструкция
 перехвата и обработки
 исключений, 125

U

Union, метод LINQ to Objects,
20, 135

V

Value, свойство классов
X-DOM, 242
var, описатель, 123

W

where, конструкция выражения
запроса, 162

Where, метод LINQ to Objects,
20, 139, 150, 235, 293
WriteAllLines, метод класса
File, 191

X

XAttribute, класс X-DOM, 78
XComment, класс X-DOM, 78
XDeclaration, класс X-DOM, 222
XDocument, класс
X-DOM, 78, 222
X-DOM (XML DOM),
 объектная модель
 XML-документа, 78, 218
XElement, класс
X-DOM, 78, 222
Xmlns, свойство класса
XNamespace, 259
xmlns, стандартный атрибут
и стандартный префикс, 258
XML-документ
 компоненты, 78
 префикс пространства
 имен, 257
 пространство имен, 254
 форматирование, 246
XName, класс X-DOM, 231, 259
XNamespace, класс X-DOM, 259
XNode, класс X-DOM, 78, 226
XObject, класс X-DOM, 78, 226
XProcessingInstruction, класс
X-DOM, 78, 222
XText, класс
X-DOM, 78, 223, 267

Z

Zero, свойство структуры
TimeSpan, 252
Zip, метод LINQ to Objects,
155, 293

А

Автоматическая генерация
объектной модели базы
данных, 311

Агрегирование, категория
запросов LINQ, 129

Анонимный метод, 281

Анонимный тип, 177, 187,
196, 283

Б

База данных, основанная
на службах
подключение, 310
создание и настройка, 306

В

Выражение запроса, 160
перечислитель в выражении
запроса, 161

Г

Группировка, категория
запросов LINQ, 172

Д

Делегат, 278
Дерево выражений, 283, 290

И

Импортирование, категория
запросов LINQ, 238
Инвертирование, запрос
LINQ, 135, 139

К

Квантификаторы, категория
запросов LINQ, 211

Компоненты XML-документа.
См. XML-документ

Л

Лексикографический порядок
строк, 138

Локальная база данных
подключение, 302
создание и настройка, 295

Лямбда-выражение, 124, 281
внешние переменные
в лямбда-выражении, 282
представление в виде
набора операторов, 188, 293

М

Метод расширения, 285

О

Обобщенный делегат, 279

Объединение
внутреннее, 154
категория запросов
LINQ, 153
левое внешнее, 154
перекрестное, 154, 157
плоское левое
внешнее, 155, 165

Отложенное выполнение
запроса LINQ, 151, 163, 292

П

Поэлементные операции,
категория запросов LINQ, 123

Префикс пространства имен
XML-документа.

См. XML-документ

Приведение атрибута/элемента
XML к новому типу, 250, 270

Проецирование, категория
запросов LINQ, 146

Пространство имен
XML-документа.
См. XML-документ

С

Сортировка, категория запросов
LINQ, 135, 144

Т

Теоретико-множественные
операции, категория запросов
LINQ, 135

Технология LINQ, общее
описание, 9

Ф

Фильтрация, категория
запросов LINQ, 135, 139

Форматирование XML-документа.
См. XML-документ

Функциональное
конструирование
компонентов XML, 222

Э

Экспортирование, категория
запросов LINQ, 164

Электронный задачник
Programming Taskbook
for LINQ. *См.* Programming
Taskbook for LINQ

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЪЯНС БУКС» наложенным платежом, выслав открытку или письмо по почтовому адресу: 123242, Москва, а/я 20 или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. (499) 725-54-09, 725-50-27; электронный адрес **books@alians-kniga.ru**.

Абрамян Михаил Эдуардович

Технология LINQ на примерах
Практикум с использованием электронного задачника
Programming Taskbook for LINQ

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Верстка *Чаннова А. А.*

Дизайн обложки *Мовчан А. Г.*

Подписано в печать 18.09.2013. Формат 60×90 1/16.

Гарнитура «Петербург». Печать офсетная.

Усл. печ. л. 20,375. Тираж 200 экз.

Веб-сайт издательства: www.dmk.ru