Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное
учреждение высшего профессионального образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

М. Э. Абрамян

Платформа .NET

Основные типы стандартной библиотеки. Работа с массивами, строками, файлами. Объекты, интерфейсы, обобщения. Технология LINQ

Учебное пособие

Ростов-на-Дону
Издательство Южного федерального университета
2014

УДК 004
ББК 32.973
А 16

Печатается по решению редакционно-издательского совета
Южного федерального университета

Рецензенты:

*доктор технических наук, профессор кафедры «Информатика»
Ростовского государственного университета путей сообщения (РГУПС)*
М. А. Бутакова

*кандидат физико-математических наук, доцент кафедры алгебры
и дискретной математики института математики, механики
и компьютерных наук Южного федерального университета*
С. С. Михалкович

Абрамян М. Э.

А 16 Платформа .NET: Основные типы стандартной библиотеки. Работа с массивами, строками, файлами. Объекты, интерфейсы, обобщения. Технология LINQ. – Ростов н/Д: Изд-во ЮФУ, 2014. – 216 с.

ISBN 978-5-9275-1356-7

Учебное пособие содержит компактное, но в то же время подробное описание ряда ключевых тем, связанных с платформой .NET Framework. Книга состоит из трех частей. В первой части описываются основные типы, входящие в стандартную библиотеку .NET, в том числе все базовые скалярные типы, структуры для работы с датами и промежутками времени, классы, связанные с массивами, строками и регулярными выражениями, а также классы для работы с файлами, каталогами и файловой системой. Вторая часть посвящена объектным возможностям платформы .NET. В ней рассматриваются основы объектной модели, интерфейсы и обобщения. Третья часть представляет собой введение в технологию LINQ и знакомит читателя с двумя интерфейсами этой технологии: LINQ to Objects и LINQ to XML.

Пособие предназначено для преподавателей программирования и студентов.

УДК 004
ББК 32.973

ISBN 978-5-9275-1356-7

Предисловие

Цель предлагаемой книги – дать компактное, но в то же время подробное описание ряда ключевых тем, связанных с платформой .NET Framework. Не претендуя на исчерпывающую полноту, доступную лишь в фундаментальных руководствах типа [3; 4; 7; 8], данная книга охватывает большую часть возможностей .NET, знание которых необходимо для успешной разработки .NET-приложений.

Книга состоит из трех частей. В первой части дается описание основных типов, входящих в библиотеку FCL (.NET Framework Class Library). Описываются все базовые скалярные типы и связанные с ними классы `Math` и `Random` (глава 1), структуры для работы с датами и промежутками времени (глава 2), массивы (глава 3), строковые классы (глава 4) и связанный с ними класс `Regex`, обеспечивающий работу с регулярными выражениями (глава 5), и, наконец, многочисленные классы для работы с двоичными и текстовыми файлами и с файловой системой (глава 6). Рассматриваются также вопросы, связанные с форматированием данных (п. 4.3) и с форматами кодирования, используемыми при файловом вводе-выводе (п. 4.4).

Для каждого типа приводится достаточно полное описание его свойств и методов за одним основным исключением: не рассматриваются возможности, связанные с многопоточностью и синхронизацией. Приведенные сведения соответствуют последней на момент написания книги версии .NET Framework 4.5. Следует отметить, что базовый состав свойств и методов рассматриваемых типов был реализован уже в версиях .NET Framework 1.0–2.0, и поэтому доступен для использования во всех применяющихся в настоящее время версиях. Дополнения, внесенные в описываемые типы после выхода версии 2.0, особо оговариваются.

Вторая часть книги посвящена объектным возможностям платформы .NET. В главе 7 подробно рассматриваются основы объектной модели, включающие такие темы, как особенности работы с размерными типами (в том числе упаковка и Nullable-структуры), свойства, делегаты и события, а также новые возможности, появившиеся в версии .NET Framework 3.5: анонимные типы, лямбда-выражения и методы расширения. Глава 8 посвящена такой важной составной части объектной модели, как интерфейсы, причем в этой главе не только рассматриваются особенности их описания и использования, но также приводятся основные интерфейсы, входящие в стандартную библиотеку, что дает возможность обсудить такие аспекты платформы .NET, как сборка мусора и освобождение неуправляе-

мых ресурсов (п. 8.1), клонирование объектов (п. 8.2), работа с коллекциями (п. 8.4–8.5), сравнение объектов (п. 8.6). В п. 8.3 рассматриваются достаточно сложные вопросы, посвященные механизму позднего связывания и его особенностям в применении к явно и неявно реализованным интерфейсным методам. Глава 9 посвящена обобщенным классам и методам (generics), которые были введены в .NET Framework 2.0. Здесь же описываются новые возможности для делегатов и интерфейсов, связанные с появлением обобщений, а также приводится сравнительная характеристика необобщенных и обобщенных стандартных коллекций.

Третья часть книги посвящена знакомству с технологией LINQ (технология запросов, встроенных в язык), появившейся в .NET Framework 3.5 и ставшей важной составной частью платформы .NET. Рассматриваются два интерфейса этой технологии: базовый интерфейс для работы с локальными коллекциями LINQ to Objects (главы 10–11) и интерфейс для обработки XML-документов LINQ to XML (глава 12). Глава 12 также содержит краткое введение в язык XML, необходимое для понимания технологии LINQ to XML.

Все рассматриваемые возможности платформы .NET и примеры их применения описываются с использованием языка C# – основного языка данной платформы. Предлагаемая книга не ставит целью изложить основы данного языка; в частности, в ней отсутствуют описания структуры C#-программы и управляющих конструкций. В то же время в ней рассматриваются особенности C# (в том числе добавленные в версии 3.0), связанные с описываемыми в книге возможностями платформы .NET. Следует подчеркнуть, что описанные в книге средства .NET (стандартные классы библиотеки FCL, объектная модель, технология LINQ) можно применять и в других .NET-языках, в том числе в Visual Basic .NET и PascalABC.NET.

При описании методов используется ряд соглашений. Поскольку описываются только *открытые* методы, модификатор `public` в заголовках методов не указывается; опущены также модификаторы, связанные с виртуальностью (за исключением заголовков методов класса `object`, приведенных в п. 7.1). В правой части описания указывается имя класса, к которому относится данный метод или набор методов, или слово «конструктор» (это название выделяется полужирным шрифтом). Заголовки перегруженных вариантов методов обычно объединяются; при этом используются вспомогательные группирующие символы «[» и «]», обрамляющие параметры, которые могут отсутствовать. Приведем условный пример:

```
void Method1(int a[, int b[, double c]]);                    SomeClass
int Method2(string s[, double d][, char c]);
```

Подобное описание означает, что класс `SomeClass` содержит методы `Method1` и `Method2`, причем первый из этих методов имеет три перегруженных

варианта, а второй – четыре. Полное описание всех вариантов данных методов выглядело бы следующим образом:

```
void Method1(int a);
void Method1(int a, int b);
void Method1(int a, int b, double c);
int Method2(string s);
int Method2(string s, double d);
int Method2(string s, char c);
int Method2(string s, double d, char c);
```

При описании *экземплярных* методов класса используется зарезервированное слово языка C# `this`; оно обозначает тот объект, для которого был вызван описываемый метод. Методы, называемые в языке C# *статическими* (описываются с помощью модификатора `static`), часто называются в книге *классовыми* методами, чтобы подчеркнуть то обстоятельство, что они связываются не с отдельным экземпляром класса (т. е. объектом), а с классом в целом.

Книга не содержит упражнений для практического освоения предлагаемого материала. В качестве подобного практикума для первой части можно рекомендовать пособие [1], а для третьей части – пособие [2]. Заметим, что первая часть настоящей книги является переработанным и дополненным вариантом второй части пособия [1]. Дополнительным источником по материалу третьей части могут служить книги [6; 9].

Часть 1
Основные типы стандартной библиотеки. Работа с массивами, строками, файлами

Глава 1. Скалярные типы платформы .NET

1.1. Общие сведения

Для скалярных типов библиотеки FCL, а также классов `System.Object` и `System.String` в языке C# предусмотрены зарезервированные слова (все эти типы называются *элементарными типами* языка C#). В табл. 1 описываются все скалярные типы платформы .NET и указываются связанные с ними зарезервированные слова C#.

Таблица 1

Скалярные типы .NET Framework

C#	Описание	Тип FCL
`bool`	Логическое значение: `true` или `false`	`System.Boolean`
`byte`	8-разрядное беззнаковое целое: `0..255`	`System.Byte`
`sbyte`	8-разрядное знаковое целое: `-128..127`	`System.SByte`
`short`	16-разрядное знаковое целое: `-32 768..32 767`	`System.Int16`
`ushort`	16-разрядное беззнаковое целое: `0..65 535`	`System.UInt16`
`int`	32-разрядное знаковое целое: `-2 147 483 648..2 147 483 647`	`System.Int32`
`uint`	32-разрядное беззнаковое целое: `0..4 294 967 295`	`System.UInt32`
`long`	64-разрядное знаковое целое: `-9 223 372 036 854 775 808..9 223 372 036 854 775 807`	`System.Int64`
`ulong`	64-разрядное беззнаковое целое: `0..18 446 744 073 709 551 615`	`System.UInt64`
`float`	32-разрядное вещественное: точность 7 знаков, порядок от 10^{-45} до 10^{38}	`System.Single`
`double`	64-разрядное вещественное: точность 15 знаков, порядок от 10^{-324} до 10^{308}	`System.Double`
`decimal`	128-разрядное вещественное: точность 28 знаков, порядок от 10^{-28} до 10^{28}	`System.Decimal`
`char`	16-разрядный символ Unicode	`System.Char`

Зарезервированные слова языка C#, указанные в первом столбце таблицы, являются *синонимами* полного имени соответствующего типа FCL (указанного в третьем столбце). Таким образом, следующие описания эквивалентны:

```
int a;
double b;
```

```
using System;
...
Int32 a;
Double b;
```

```
System.Int32 a;
System.Double b;
```

В настоящей книге всегда применяются описания, использующие зарезервированные слова языка; аргументация за и против такого подхода приводится в [5, п. 6.14].

1.2. Тип bool

Операции, предусмотренные для логического типа `bool`, и их обозначения на языке C# приведены в табл. 2.

Таблица 2

Логические операции

Операция	C#	Операция	C#
«И»	`&`	«И» (сокращенная схема)	`&&`
«ИЛИ»	`|`	«ИЛИ» (сокращенная схема)	`||`
Отрицание	`!`	«Исключающее ИЛИ»	`^`

```
static bool Parse(string value);                                    bool
static bool TryParse(string value, out bool result);
```

Метод `Parse` возвращает логическое значение, соответствующее строке `value`. Строка `value` должна содержать текст `"True"` или `"False"` (регистр не учитывается, допускается наличие начальных и конечных пробелов). Если строка содержит другой текст, то метод `Parse` возбуждает исключение `FormatException`. Метод `TryParse` работает аналогично методу `Parse`, за исключением того, что логическое значение, соответствующее строке, возвращается в выходном параметре `result` (при этом сам метод возвращает значение `true`). Основное отличие метода `TryParse` от метода `Parse` состоит в том, что метод `TryParse` не возбуждает исключений; при невозможности интерпретации строки как логического значения он возвращает значение `false`, и это же значение сохраняется в параметре `result`.

```
string ToString();                                                  bool
```

Возвращает строковое представление объекта `this` (`"True"` или `"False"`).

1.3. Числовые типы

1.3.1. Преобразование числовых типов

Преобразование числовых типов может выполняться *неявно* (т. е. автоматически) или *явно*. Для явного преобразования типов в языке C# предназначен *оператор преобразования типа*:

```
(тип)выражение
```

Здесь *тип* обозначает имя типа, к которому должно быть преобразовано указанное *выражение*.

При выполнении явного преобразования вещественных данных к целому типу в C# возвращается ближайшее целое «по направлению к нулю». Например, любое из приведенных ниже выражений вернет целое число 0:

```
(int)0.4
(int)0.9
(int)-0.9
```

Если требуемое преобразование не может привести к потере информации, то оно может быть выполнено неявно (например, тип `byte` может быть неявно преобразован в `int`, тип `int` – в `long`, любой целочисленный тип – в `double`). Неявные преобразования, приводящие к потере информации, в языке C# *запрещены*.

1.3.2. Поля и методы числовых типов

Текст *числовой_тип*, используемый в заголовках полей и методов, обозначает любой из числовых типов (как целочисленных, так и вещественных).

```
static const числовой_тип MaxValue;                    числовой_тип
static const числовой_тип MinValue;
```

Возвращает соответственно максимальное и минимальное значение для данного числового типа (например, `int.MaxValue` равно `2147483647`, а `double.MaxValue` равно `1.79769313486232E+308`).

```
static числовой_тип Parse(string value[,                числовой_тип
                IFormatProvider p]);
static bool TryParse(string value, out числовой_тип result);
```

Преобразует строку `value`, содержащую изображение числа, в число соответствующего типа. В качестве параметра `p` можно использовать объект типа `CultureInfo`, определенный в пространстве имен `System.Globalization`; это позволяет учитывать в методе `Parse` различные *региональные настройки*. Например, если в качестве параметра `p` указать `new CultureInfo("en-US")`, то будут использоваться форматные настройки для США (строка `"ru-RU"` соответствует настройкам для России). Если параметр `p` не указан, то используются *текущие региональные настройки*; по умолчанию они определяются настройками системы Windows. Метод `Try-`

Parse отличается от метода Parse тем, что при невозможности преобразования строки не возбуждает исключения, а возвращает значение false (при этом параметр result полагается равным нулевому значению соответствующего числового типа); при успешном преобразовании результат записывается в параметр result, а метод возвращает значение true. Для метода TryParse также предусмотрен вариант с дополнительным параметром типа IFormatProvider, однако в этом варианте требуется указывать еще один параметр типа NumberStyles, который в данной книге не рассматривается.

Имеется возможность изменить текущие региональные настройки, используемые в программе. Для этого достаточно присвоить новое значение типа CultureInfo свойству CurrentCulture объекта Thread.CurrentThread, определенного в пространстве имен System.Threading. Например, чтобы задать в качестве текущих настроек программы настройки для США, достаточно выполнить следующий оператор присваивания:

```
System.Threading.Thread.CurrentThread.CurrentCulture =
  new System.Globalization.CultureInfo("en-US");
```

```
string ToString([string fmt][, IFormatProvider p]);          числовой_тип
```

Возвращает строковое представление объекта this. Позволяет указывать форматную строку fmt и объект p, определяющий региональные настройки (см. выше описание метода Parse). Различные спецификаторы формата, используемые в форматной строке, приводятся в п. 4.3.2. Если параметр fmt не указан или равен null, то выполняется *форматирование по умолчанию*. Если параметр p не указан или равен null, то используются текущие региональные настройки.

Следующие классовые неизменяемые поля и методы определены только для вещественных типов. Приведем их заголовки для типа double.

```
static const double Epsilon;                                        double
static const double NaN;
static const double NegativeInfinity;
static const double PositiveInfinity;
```

Определяют числовые константы, соответствующие особым значениям: Epsilon определяет минимальное положительное вещественное число, NegativeInfinity и PositiveInfinity определяют значение «минус бесконечность» (–∞) и «плюс бесконечность» (+∞) соответственно (эти значения могут быть получены, например, при делении отрицательного или положительного числа на 0), NaN определяет значение «не число» (Not a Number; это значение, например, является результатом деления 0 на 0). Подчеркнем, что при выполнении арифметических действий над вещественными числами никогда не возбуждается исключений, однако результатом может быть одно из особых значений, приведенных выше.

```
static bool IsInfinity(double value);                                    double
static bool IsNaN(double value);
static bool IsNegativeInfinity(double value);
static bool IsPositiveInfinity(double value);
```

Позволяют проверить, содержит ли параметр `value` особое значение требуемого вида (см. выше описание особых значений).

1.3.3. Стандартные математические функции: класс Math

Стандартные математические функции для любых числовых типов реализованы в библиотеке FCL в виде классовых методов класса `Math`, определенного в пространстве имен `System`. Кроме классовых методов данный класс имеет два классовых неизменяемых поля типа `double`:

- `E` – константа *e*, равная `2.71828182845905`;
- `PI` – константа π, равная `3.14159265358979`.

Все методы класса `Math` можно разбить на три группы. К первой группе относятся четыре метода, которые реализованы для большинства числовых типов.

```
static числовой_тип Min(числовой_тип val1, числовой_тип val2);     Math
static числовой_тип Max(числовой_тип val1, числовой_тип val2);
static числовой_тип Abs(числовой_тип value);
static int Sign(числовой_тип value);
```

Методы `Min` и `Max` возвращают соответственно меньший и больший из своих параметров, метод `Abs` возвращает модуль (абсолютное значение) параметра `value`, а метод `Sign` возвращает –1 для отрицательных параметров `value`, 0 для нулевых и 1 для положительных.

Для каждого варианта перегруженных методов `Min`, `Max`, `Abs` возвращаемый тип совпадает с типом параметров, а для методов `Min` и `Max`, кроме того, совпадают типы параметров. Для беззнаковых типов варианты методов `Abs` и `Sign` не определены.

Вторая группа методов связана с обработкой целых чисел.

```
static long BigMul(int a, int b);                                        Math
static int DivRem(int a, int b, out int rem);
static long DivRem(long a, long b, out long rem);
```

Метод `BigMul` обеспечивает правильное перемножение двух чисел типа `int` в ситуации, когда результат этого перемножения выходит за пределы допустимого для `int` диапазона значений.

Метод `DivRem` выполняет деление нацело параметра `a` на параметр `b`. Возвращаемым значением является частное, а в выходном параметре `rem` возвращается остаток. Если параметр `b` равен 0, то возбуждается исключение `DivideByZeroException`.

Наконец, третья, самая многочисленная, группа методов класса `Math` связана с обработкой вещественных чисел.

```
static double Round(double value[, int d][,                    Math
                MidpointRounding mode]);
static decimal Round(decimal value[, int d][, MidpointRounding mode]);
```

Возвращает значение числа `value`, округленное до ближайшего числа с `d` дробными знаками. Если параметр `d` не указан, то он считается равным 0, т. е. число `value` округляется до целого. В случае значения `value`, равного `NaN` или ±∞, возвращается само значение `value`.

Параметр `mode` перечислимого типа `MidpointRounding` определяет способ округления в ситуации, когда число `value` находится в точности посредине между возможными округленными значениями. Если `mode` равен `MidpointRounding.ToEven`, то подобные числа округляются так, чтобы их последняя цифра была *четной* («*банковское округление*»); если `mode` равен `MidpointRounding.AwayFromZero`, то числа округляются «по направлению от нуля», т. е. из двух возможных округленных значений выбирается более удаленное от 0. Если параметр `mode` не указан, то выполняется банковское округление.

Для остальных методов, связанных с обработкой вещественных чисел, достаточно привести краткие описания (все параметры и возвращаемые значения этих методов имеют тип `double`; для методов `Ceiling`, `Floor` и `Truncate` реализованы также варианты, в которых параметр и возвращаемое значение имеют тип `decimal`).

- `IEEERemainder(x, y)` – остаток от деления числа `x` на число `y`, понимаемый как число, равное `x - y * Round(x/y)` (название метода связано с аббревиатурой организации, которая в 1985 г. разработала стандарт выполнения операций над вещественными числами, используемый в .NET Framework: IEEE – Institute of Electrical and Electronics Engineers).
- `Ceiling(x)`, `Floor(x)`, `Truncate(x)` – *округление* числа `x` соответственно *с избытком*, *с недостатком* и *по направлению к нулю* (`Ceiling` возвращает ближайшее к `x` целое число, большее или равное `x`, а `Floor` – меньшее или равное `x`). Для `x`, равного `NaN` или ±∞, возвращается `x`. Подчеркнем, что возвращаемое значение имеет тип `double`. Метод `Truncate` доступен, начиная с версии .NET Framework версии 3.5.
- `Sqrt(x)` – квадратный корень из `x`. Если `x` отрицателен или равен `NaN`, то возвращается `NaN`, если `x` равен +∞, то возвращается +∞.
- `Pow(x, y)` – степенная функция x^y. Если `x` или `y` равен `NaN`, то возвращается `NaN`; если `x` равен `1`, а `y` не равен `NaN`, то возвращается `1`;

для конечных отрицательных x функция правильно обрабатывает случай целых показателей y. Особые ситуации, связанные с бесконечными значениями, рассматривать не будем.

- Exp(x) – *показательная функция* e^x. Если x равен NaN или +∞, то возвращается x, если x равен –∞, то возвращается 0.
- Log(x), Log10(x), Log(x, a) – *логарифмические функции* по основанию *e*, 10 и a соответственно: ln *x*, lg *x*, $\log_a x$. Если x < 0, то возвращается NaN, если x равен 0 или +∞, то при a > 1 возвращается соответственно –∞ и +∞, а при a ∈ (0, 1) возвращается +∞ и –∞. Если x равен NaN, то возвращается NaN. Если a < 0, a = 1 или a = NaN, то возвращается NaN. Другие особые ситуации рассматривать не будем.
- Sin(x), Cos(x), Tan(x) – *тригонометрические функции*; x измеряется в радианах. Если x равен NaN или ±∞, то возвращается NaN. Функция Tan для конечных x всегда принимает конечные значения (например, для x = ±π/2 функция Tan возвращает значение ±1,63317787283838E+16).
- Asin(x), Acos(x), Atan(x) – *обратные тригонометрические функции*; возвращаемые значения измеряются в радианах и лежат в следующих промежутках: [–π/2, π/2] для Asin и Atan, [0, π] для Acos. Особые ситуации для Asin и Acos: если x не принадлежит промежутку [–1, 1], то возвращается NaN. Особые ситуации для Atan: если x = NaN, то возвращается NaN, если x = ±∞, то возвращается ±π/2.
- Atan2(y, x) – *угол наклона* радиус-вектора с координатами {*x*, *y*} к положительной полуоси *OX*, измеряемый в радианах и лежащий в промежутке (–π, π]. Допускаются как конечные, так и бесконечные координаты. Если один или оба параметра равны NaN, то возвращается NaN.
- Sinh(x), Cosh(x), Tanh(x) – *гиперболические функции*. Если x равен NaN, то возвращается NaN; если x равен ±∞, то Sinh возвращает ±∞, Cosh возвращает +∞, а Tanh возвращает ±1.

1.3.4. Генерация случайных чисел: класс Random

Для генерации случайных чисел (как целых, так и вещественных) предназначен класс Random, определенный в пространстве имен System.

```
Random([int seed]);                                    конструктор
```

Данный конструктор создает объект типа Random – *датчик равномерно распределенных псевдослучайных чисел* – и инициализирует его с помощью целочисленного параметра seed. Если параметр seed не указан, то датчик инициализируется значением, полученным на основе текущего времени (по часам компьютера). Объекты типа Random, инициализированные с по-

мощью одинаковых значений seed, будут генерировать одинаковые последовательности случайных чисел.

```
int Next([int min,] int max);                                    Random
```

Возвращает очередное случайное число типа int. Если параметры отсутствуют, то число выбирается из диапазона [0, int.MaxValue) (где int.MaxValue – максимальное значение типа int); если указан только параметр max, то число выбирается из диапазона [0, max); если указаны два параметра, то число выбирается из диапазона [min, max). Левая граница диапазона включается в него, а правая – нет. Если границы диапазона равны, то всегда возвращается значение левой границы диапазона. Если значение левой границы больше значения правой, то возбуждается исключение.

```
double NextDouble();                                             Random
```

Возвращает очередное случайное число типа double, лежащее в полуинтервале [0, 1).

```
void NextBytes(byte[] buffer);                                   Random
```

Заполняет все элементы массива байтов buffer случайными целыми числами из диапазона 0–255 (включая границы диапазона). Если параметр buffer равен null, то возбуждается исключение.

1.4. Тип char

1.4.1. Символы и их коды. Символьные литералы

Для определения кода по символу c или, наоборот, символа по его коду n в C# достаточно выполнить *явное преобразование типов* (int)c и (char)n.

Символы-литералы в языке C# заключаются в *одинарные* кавычки: 'A', '1' (в отличие от строк-литералов, заключаемых в *двойные* кавычки).

1.4.2. Кодовые таблицы Unicode

При работе с символами любые языки платформы .NET используют *кодировку Unicode*, в которой каждый символ кодируется двухбайтным целым числом (за исключением так называемых *дополнительных символов*, например, очень редко используемых китайских иероглифов или символов вымерших языков, которые кодируются в виде *суррогатных пар* – двух двухбайтных целых из фиксированного диапазона).

Таблица 3 содержит символы кодировки Unicode с кодами 32–127. Коды 1–31 генерируются, как правило, при нажатии *управляющих клавиш*, поэтому соответствующие им символы в таблице не указаны. Данная часть кодировки Unicode совпадает с *кодировкой ASCII*. Отличие кодировки ASCII от кодировки Unicode состоит в том, что в кодировке ASCII для хранения каждого символа выделяется *один* байт.

Таблица 4 содержит ту часть кодировки Unicode, которая связана с русскими буквами и некоторыми дополнительными символами.

Таблица 3

Общая часть кодировок ASCII и Unicode

	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

Таблица 4

Фрагмент кодировки Unicode с русскими буквами и некоторыми дополнительными символами

А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055
Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071
а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087
р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я
1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103
Ё	ё	неразрывный пробел		§	©	«	°	»	–	—	‛	″	...	€	№
1025	1105	160		167	169	171	176	187	8211	8212	8220	8221	8230	8364	8470

1.4.3. Методы типа char

```
static bool IsControl(char c);                                    char
static bool IsDigit(char c);
static bool IsLetter(char c);
static bool IsLetterOrDigit(char c);
static bool IsLower(char c);
static bool IsUpper(char c);
static bool IsPunctuation(char c);
static bool IsSymbol(char c);
static bool IsSeparator(char c);
static bool IsWhiteSpace(char c);
```

Позволяют определить вид символа `c`: `IsControl` – *управляющий символ* (символы с кодами в диапазонах 0–31 и 127–159); `IsDigit` – *цифра* (симво-

лы 0–9, а также символы из других алфавитов, обозначающие цифры); IsLetter – *буква*, IsLower – *строчная* (маленькая) буква; IsUpper – *прописная* (заглавная) буква; IsPunctuation – *знак пунктуации* (символы ! " # % & ' ( ) * , - . / ? @ [\ ] _ { } « » – — ‟ ″ … и некоторые другие); IsSymbol – *специальный символ*, не являющийся знаком пунктуации (символы $ + < = > ^ ` | ~ § © ° € № и некоторые другие); IsSeparator – *разделитель* (пробел (код 32), неразрывный пробел (код 160) и некоторые символы из других алфавитов); IsWhiteSpace – *пробельный символ* (символы с кодами 9 (знак табуляции), 10 (новая строка), 11–12, 13 (возврат каретки), 32, 133, 160 и некоторые символы из других алфавитов).

Для всех этих методов есть вариант с двумя параметрами: (string s, int index); в этом варианте анализируется символ строки s с индексом index.

```
static char ToLower(char c);                                    char
static char ToUpper(char c);
```

Возвращает символ c, преобразованный к нижнему или верхнему регистру соответственно (если символ не является буквой, то возвращается этот же символ).

```
string ToString();                                              char
static string ToString(char c);
```

Экземплярный метод возвращает строковое представление символа this, классовый метод – строковое представление символа c. Типы string и char не являются совместимыми по присваиванию, поэтому для того чтобы присвоить значение символа c строке s, необходимо предварительно преобразовать символ в строку: s = c.ToString() или s = char.ToString(c).

Глава 2. Работа с датами и промежутками времени

Для работы с датами и промежутками времени в .NET Framework предусмотрены две основные структуры DateTime и TimeSpan, определенные в пространстве имен System. Структура DateTime предназначена для манипулирования *абсолютными* календарными датами, а структура TimeSpan позволяет хранить *относительные* промежутки времени. Кроме этих основных структур имеется несколько вспомогательных перечислений и классов, например перечисление DayOfWeek, также определенное в пространстве имен System и содержащее имена дней недели, начиная от воскресенья: Sunday (0), Monday (1), Tuesday (2), Wednesday (3), Thursday (4), Friday (5), Saturday (6). В скобках указаны числовые значения элементов данного перечисления.

Для того чтобы избежать неоднозначности в описаниях методов, дату, дополненную временем, будем называть в дальнейшем *полной датой*.

2.1. Структура DateTime

2.1.1. Возможные значения и варианты создания

```
static readonly DateTime MinValue;                                    DateTime
static readonly DateTime MaxValue;
```

Данные классовые поля возвращают минимальное и максимальное значение объекта DateTime: MinValue соответствует полуночи 1 января 1 года (нашей эры), а MaxValue – полуночи 1 января 10 000 года *минус 1 такт*. Один *такт* (англ. tick) равен 100 наносекундам, или 10^{-7} с; это *минимальная единица измерения времени* для структур DateTime и TimeSpan.

```
DateTime(long ticks);                                                 конструктор
DateTime(int year, int month, int day[, int hour, int minute,
                int second[, int millisecond]]);
```

Наиболее часто используемые варианты конструктора, позволяющего создать объект типа DateTime. Смысл параметров ясен из их названий; годы, месяцы и дни нумеруются от 1, остальные параметры, в том числе такты (ticks), – от 0. Отсутствующие временны*е* компоненты полагаются равными 0. При указании недопустимого значения возбуждается исключение ArgumentOutOfRangeException.

Дата в указанных конструкторах соответствует *григорианскому календарю*. Имеется возможность при создании указать другой календарь (например, юлианский), а также уточнить, какое время подразумевается: *локальное* (local time) или *универсальное по Гринвичу* (coordinated universal time, UTC). Мы не будем рассматривать возможности .NET, связанные с использованием различных календарей и универсального времени.

Создать объект типа DateTime можно также с помощью нескольких классовых свойств и методов. Опишем наиболее полезные из них.

```
static DateTime Now { get; }                                          DateTime
static DateTime Today { get; }
```

Свойство Now возвращает текущую полную дату (т. е. дату и время по встроенным часам компьютера), а свойство Today – только текущую дату (время соответствует полуночи).

```
static DateTime Parse(string s[, IFormatProvider p]);                 DateTime
static bool TryParse(string s, out DateTime result);
```

Возвращает объект типа DateTime, соответствующий строке s, содержащей текстовое представление даты. Параметр p позволяет задать региональные настройки, которые будут учитываться при разборе строки s (см. описание метода Parse в п. 1.3.2); если параметр p отсутствует, то используются текущие региональные настройки. Если дата в строке не указана или указана частично (например, отсутствует год), то отсутствующая информация определяется по текущей дате компьютера. Если время в строке не указано или указано частично (например, отсутствуют секунды), то от-

сутствующие элементы для времени полагаются равными 0. Если метод не может преобразовать содержимое строки `s` в объект типа `DateTime`, то возбуждается исключение `FormatException`. Метод `TryParse` отличается от метода `Parse` тем, что при невозможности преобразования строки не возбуждает исключения, а возвращает значение `false` (при этом параметр `result` полагается равным значению `DateTime.MinValue`); при успешном преобразовании результат записывается в параметр `result`, а метод возвращает значение `true`. Для метода `TryParse` также предусмотрен вариант с дополнительным параметром типа `IFormatProvider`, однако в этом варианте требуется указывать еще один параметр типа `DateTimeStyles`, который в данной книге не рассматривается.

Следует отметить, что методы `Parse` и `TryParse` в состоянии распознать достаточно много вариантов строковых описаний даты и времени. В частности, в качестве разделителей для даты можно указывать как точку «.», так и символ «/» (в качестве разделителей для времени должно использоваться двоеточие «:»), при указании времени можно опускать секунды, при указании *только* даты можно опускать год, между элементами даты и/или времени могут содержаться пробелы.

Региональные настройки могут оказывать существенное влияние на разбор строки. Например, если в качестве параметра `p` указать `new CultureInfo("en-US")` (региональные настройки для США), то строка `"1/12/14"` будет интерпретироваться как 12 января 2014 года, а если параметр `p` не указывать (при работе в русской версии Windows) или указать равным `new CultureInfo("ru-RU")` (настройки для России), то эта же строка будет интерпретироваться как 1 декабря 2014 года.

2.1.2. Свойства

Экземплярные свойства объекта типа `DateTime` являются свойствами только для чтения и позволяют либо выделить из полной даты один элемент или группу элементов, либо рассчитать дополнительные характеристики, связанные с датой. Смысл свойств `Year`, `Month`, `Day`, `Hour`, `Minute`, `Second`, `Millisecond` ясен из их названий (все эти свойства имеют тип `int`). Опишем остальные свойства:

- `Ticks` (типа `long`) – число тактов, содержащихся в указанной полной дате. Например, для начала 3-го тысячелетия (полночь 1 января 2001 г.) данное свойство равно `631 139 040 000 000 000`;
- `Date` (типа `DateTime`) – дата без времени (время соответствует полуночи);
- `TimeOfDay` (типа `TimeSpan`) – время без даты;

- DayOfYear (типа int) – число дней, прошедших от начала года до указанной даты включительно (например, для 1 февраля будет возвращено 32);
- DayOfWeek (типа перечисления DayOfWeek) – день недели, соответствующий указанной дате.

2.1.3. Сравнение дат

```
static bool Equals(DateTime t1, DateTime t2);                    DateTime
```

Возвращает true, если объекты t1 и t2 содержат одинаковую полную дату; в противном случае возвращает false.

```
bool Equals(object value);                                       DateTime
```

Возвращает true, если объект value имеет тип DateTime и содержит ту же полную дату, что и объект this; в противном случае возвращает false.

```
static int Compare(DateTime t1, DateTime t2);                    DateTime
```

Выполняет сравнение полных дат t1 и t2 и возвращает отрицательное число, если полная дата t1 является более ранней, чем t2; положительное число, если t1 является более поздней, чем t2; нулевое число, если объекты t1 и t2 равны.

```
int CompareTo(object value);                                     DateTime
```

Если фактический тип параметра value равен DateTime, то метод выполняет сравнение объектов this и value и возвращает отрицательное, положительное или нулевое число в зависимости от того, является ли полная дата this более ранней, более поздней, чем полная дата value, или эти даты равны. Если параметр value равен null, то метод возвращает положительное число. В остальных случаях возбуждается исключение ArgumentException.

В языке C# для сравнения дат можно использовать любые операции сравнения (==, !=, <, <=, >, >=).

2.1.4. Операции над датами

При выполнении любого из описанных в данном пункте методов объект this, вызвавший метод, *не изменяется*. Если при выполнении любой операции над датами результат выходит за допустимый диапазон значений, то возбуждается исключение ArgumentOutOfRangeException.

```
DateTime Add(TimeSpan value);                                    DateTime
```

Возвращает полную дату, полученную из объекта this прибавлением к нему промежутка времени value.

```
DateTime Subtract(TimeSpan value);                               DateTime
TimeSpan Subtract(DateTime value);
```

Первый вариант метода возвращает полную дату, полученную из объекта this вычитанием из него промежутка времени value. Второй вариант

метода возвращает промежуток времени, полученный при вычитании из объекта this полной даты value.

В языке C# вместо описанных выше методов можно использовать операции «+» и «–». Если первым операндом для операции «+» является объект типа DateTime, то второй операнд должен иметь тип TimeSpan (результат имеет тип DateTime). При использовании операции «–» для первого операнда типа DateTime второй операнд может иметь либо тип TimeSpan (в этом случае результат имеет тип DateTime), либо тип DateTime (в этом случае результат имеет тип TimeSpan).

```
DateTime AddYears(int value);                                    DateTime
DateTime AddMonths(int value);
DateTime AddDays(double value);
DateTime AddHours(double value);
DateTime AddMinutes(double value);
DateTime AddSeconds(double value);
DateTime AddMilliseconds(double value);
DateTime AddTicks(long value);
```

Любой из данных методов возвращает полную дату, полученную из объекта this путем добавления указанного количества соответствующих элементов полной даты. Параметр value может быть как положительным, так и отрицательным. Обратите внимание на то, что при изменении дней, часов, минут, секунд и миллисекунд можно указывать дробные значения.

2.1.5. Форматирование дат

```
string ToString([string fmt][, IFormatProvider p]);              DateTime
```

Возвращает строковое представление объекта this. Позволяет указывать форматную строку fmt и/или объект p, определяющий региональные настройки (см. описание метода Parse в п. 1.3.2). Различные спецификаторы формата, используемые в форматной строке, приводятся в п. 4.3.2. Если параметр fmt не указан или равен null, то выполняется *форматирование по умолчанию* с применением *общего формата* G. Если параметр p не указан или равен null, то используются текущие региональные настройки.

Помимо «универсального» метода ToString имеются специализированные методы, предназначенные для получения конкретного представления даты или времени для *текущих* региональных настроек. Данные методы не имеют параметров и возвращают строку. Приведем имена этих методов, указав также спецификаторы формата для метода ToString, позволяющие получить то же самое строковое представление (в примерах, указанных в скобках, используются региональные настройки для России):

- ToShortDateString – дата в кратком формате d ("27.01.1756");
- ToLongDateString – дата в полном формате D ("27 января 1756 г.");

- `ToShortTimeString` – время в кратком формате `t` (`"10:55"`);
- `ToLongTimeString` – время в полном формате `T` (`"10:55:15"`).

2.1.6. Вспомогательные методы

```
static bool IsLeapYear(int year);                                    DateTime
```

Возвращает `true`, если параметр `year` соответствует *високосному году*, и `false` в противном случае.

```
static int DaysInMonth(int year, int month);                         DateTime
```

Возвращает число дней в месяце `month` года `year` (месяцы нумеруются от `1`).

2.2. Структура TimeSpan

2.2.1. Возможные значения, варианты создания и строковое представление

```
static readonly TimeSpan MinValue;                                   TimeSpan
static readonly TimeSpan MaxValue;
static readonly TimeSpan Zero;
```

Данные классовые поля возвращают минимальное и максимальное значения объекта `TimeSpan`, а также нулевое значение для данного типа. Промежутки времени, определяемые типом `TimeSpan`, измеряются в днях, часах, минутах, секундах, миллисекундах и *тактах* (англ. tick; один такт равен 10^{-7} с) и могут быть как положительными, так и отрицательными. Значение `MaxValue` с точностью до секунд равно `10675199.02:48:05` (перед точкой указывается число дней), значение `MinValue` равно величине `MaxValue`, взятой с противоположным знаком, *минус 1 такт*.

Структура `TimeSpan` содержит также набор полей типа `long` (только для чтения), позволяющих определить, сколько тактов содержится в других единицах измерения времени: `TicksPerDay`, `TicksPerHour`, `TicksPerMinute`, `TicksPerSecond`, `TicksPerMillisecond`.

```
TimeSpan(long ticks);                                              конструктор
TimeSpan(int days, int hours, int minutes, int seconds,
              int milliseconds);
TimeSpan([int days,] int hours, int minutes, int seconds);
```

Смысл параметров данных вариантов конструктора ясен из их названий; все параметры могут быть нулевыми, положительными или отрицательными. В последнем варианте конструктора число миллисекунд (а также число дней, если параметр `days` не указан) полагается равным 0.

```
static TimeSpan FromDays(double value);                              TimeSpan
static TimeSpan FromHours(double value);
static TimeSpan FromMinutes(double value);
static TimeSpan FromSeconds(double value);
```

```
static TimeSpan FromMilliseconds(double value);
static TimeSpan FromTicks(long value);
```

Данные классовые методы позволяют создать объект TimeSpan, указав значение промежутка времени в одной из стандартных единиц измерения. Обратите внимание на то, что количество указываемых единиц (кроме тактов) может быть дробным.

```
static TimeSpan Parse(string s);                                    TimeSpan
static bool TryParse(string s, out TimeSpan result);
```

Метод Parse возвращает объект типа TimeSpan, соответствующий строке s, в которой содержится текстовое представление промежутка времени. Строка s должна иметь следующий формат (необязательные элементы заключены в квадратные скобки): [-][d.]h:m[:s[.f]], где d – количество дней, h – количество часов (от 0 до 23), m и s – количество минут и секунд (от 0 до 59), f – дробная часть секунды, которая может содержать от 1 до 7 цифр. Допускается также строка, содержащая только количество дней (возможно, со знаком «–»). Строка s может также содержать начальные и конечные пробелы. Если метод не может преобразовать содержимое строки s в объект типа TimeSpan, то возбуждается исключение. Метод TryParse работает аналогично методу Parse, за исключением того, что промежуток времени, соответствующий строке, возвращается в выходном параметре result (при этом сам метод возвращает значение true). Основное отличие метода TryParse от метода Parse состоит в том, что метод TryParse не возбуждает исключений; при невозможности интерпретации строки как промежутка времени он возвращает значение false, а в параметр result записывается значение TimeSpan.Zero.

Следует заметить, что приведенный выше формат используется методом ToString (без параметров) для обратного преобразования объекта типа TimeSpan к строковому представлению. В полученной строке обязательно присутствуют элементы h, m и s, причем для каждого из них отводится по две цифры. Элементы d и f указываются только в случае, если они не равны 0, а знак «–», естественно, только если промежуток времени является отрицательным.

2.2.2. Свойства

Все свойства структуры TimeSpan являются свойствами только для чтения. Их можно разделить на две группы.

Первая группа свойств позволяет выделить один из компонентов, входящих во временной промежуток. Это свойства Days, Hours, Minutes, Seconds, Milliseconds, смысл которых очевиден (все эти свойства имеют тип int).

Вторая группа свойств позволяет выразить временной промежуток, содержащийся в объекте TimeSpan, в одной из стандартных единиц измере-

ния времени. Это свойства TotalDays, TotalHours, TotalMinutes, TotalSeconds, TotalMilliseconds (все они имеют тип double, т. е. могут содержать дробную часть), а также свойство Ticks типа long.

2.2.3. Сравнение промежутков времени

```
static bool Equals(TimeSpan t1, TimeSpan t2);                    TimeSpan
```

Возвращает true, если объекты t1 и t2 содержат одинаковые промежутки времени; в противном случае возвращает false.

```
bool Equals(object value);                                       TimeSpan
```

Возвращает true, если объект value имеет тип TimeSpan и содержит тот же промежуток времени, что и объект this; в противном случае возвращает false.

```
static int Compare(TimeSpan t1, TimeSpan t2);                    TimeSpan
```

Выполняет сравнение промежутков времени t1 и t2 и возвращает отрицательное число, если t1 меньше, чем t2; положительное число, если t1 больше, чем t2; нулевое число, если объекты t1 и t2 равны.

```
int CompareTo(object value);                                     TimeSpan
```

Если фактический тип параметра value равен TimeSpan, то метод выполняет сравнение объектов this и value и возвращает отрицательное, положительное или нулевое число в зависимости от того, является ли временной промежуток this меньшим, б*о*льшим, чем промежуток value, или эти промежутки равны. Если параметр value равен null, то метод возвращает положительное число. В остальных случаях возбуждается исключение ArgumentException.

В языке C# для сравнения промежутков времени можно использовать любые операции сравнения (==, !=, <, <=, >, >=).

2.2.4. Операции над промежутками времени

При выполнении любого из описанных в данном пункте методов объект this, вызвавший метод, *не изменяется*. Если при выполнении любой операции над промежутками времени результат выходит за допустимый диапазон значений, то возбуждается исключение ArgumentOutOfRangeException.

```
TimeSpan Add(TimeSpan value);                                    TimeSpan
TimeSpan Subtract(TimeSpan value);
```

Возвращает промежуток времени, полученный из объекта this прибавлением (метод Add) или вычитанием (метод Subtract) промежутка value.

```
TimeSpan Negate();                                               TimeSpan
```

Возвращает промежуток времени, полученный из объекта this переменой его знака на противоположный.

```
TimeSpan Duration();                                    TimeSpan
```

Возвращает промежуток времени, полученный из объекта `this` отбрасыванием его знака.

В языке C# вместо описанных выше методов (кроме `Duration`) можно использовать операции + и –, в том числе унарный минус.

Глава 3. Массивы

3.1. Описание массива и его инициализация

Наряду с *одномерными* (single-dimensional) массивами, элементы которых имеют единственный индекс, в языках платформы .NET можно использовать многомерные *прямоугольные* (multidimensional) массивы, в которых диапазон значений каждого индекса не зависит от значений остальных индексов. Кроме того, можно создать *массив массивов*; в котором размеры элементов-массивов могут различаться; такой массив массивов называется *невыровненным* (jagged) многомерным массивом.

Несмотря на то что для работы с массивами в библиотеке FCL предусмотрен специальный класс `Array`, описание и инициализация массива в каждом языке программирования платформы .NET обычно выполняется по особым правилам, определяемым синтаксисом этого языка.

Класс `Array` включает средства, позволяющие задавать произвольный диапазон значений для каждого индекса, однако индексы массивов, создаваемых с помощью стандартных описателей в языке C#, всегда имеют *фиксированную нижнюю границу*, равную 0.

В последующих примерах предполагается, что массив `a` является одномерным массивом символов и имеет размер 5, а массив `b` является двумерным массивом-матрицей целых чисел размера 3 на 4.

При описании массива достаточно указать тип его элементов и количество индексов (т. е. его *размерность*); размер массива можно не указывать:

```
char[] a;
int[,] b;
```

Затем необходимо выполнить *инициализацию* массива. Простейшим вариантом инициализации является *неявная инициализация*, или *инициализация по умолчанию* (элементы получают нулевые значения соответствующего типа):

```
a = new char[5];
b = new int[3, 4];
```

При инициализации массива в языке C# указывается *количество значений* для каждого индекса (допустимо указывать не только константы, но и *выражения* целого типа).

Описание массива и его инициализацию можно совместить:

```
char[] a = new char[5];
int[,] b = new int[3, 4];
```

При инициализации массива можно явно указать значения его элементов:

```
a = new char[] {'A', 'B', 'C', 'D', 'E'};
b = new int[,] {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
```

В этом случае разрешено не указывать размер массива; если же размер указывается, то он должен соответствовать числу указанных элементов.

Переменную-массив можно инициализировать повторно; при этом прежнее содержимое массива теряется. При повторной инициализации допустимо указывать новый размер массива (однако нельзя изменять *размерность* массива и тип его элементов):

```
a = new char[5];
...
a = new char[10];
```

При передаче входного параметра-массива в какой-либо метод допустимо создавать массив «на лету», используя конструкцию явной инициализации (при этом массив не сохраняется в некоторой переменной, а сразу передается в указанный метод). Если, например, метод `M` принимает входной параметр – символьный массив, то для передачи в него массива из трех элементов «A», «B», «C» можно использовать следующий оператор:

```
M(new char[] {'A', 'B', 'C'});
```

В заключение приведем пример описания и неявной инициализации невыровненного двумерного целочисленного массива `d` с элементами $d_{0,0}$, $d_{1,0}$, $d_{1,1}$, $d_{2,0}$, $d_{2,1}$, $d_{2,2}$ (данный массив можно использовать, например, для хранения ненулевой части нижнетреугольной матрицы порядка 3). В приведенном примере после инициализации массива нулевое значение элемента $d_{1,0}$ заменяется на 5:

```
int[][] d = new int[3][];
d[0] = new int[1];
d[1] = new int[2];
d[2] = new int[3];
d[1][0] = 5;
```

Обратите внимание на то, что для невыровненных массивов каждый индекс необходимо указывать в отдельных скобках (в отличие от многомерных прямоугольных массивов, в которых используется одна пара скобок, а индексы разделяются запятыми).

3.2. Класс Array

3.2.1. Свойства и экземплярные методы

Приведенные ниже свойства и экземплярные методы класса `Array` можно использовать для массивов любого типа и любой размерности (за исключением метода `CopyTo`, предназначенного только для одномерных массивов).

```
int Length { get; }                                                    Array
```

Возвращает *размер* массива `this`, т. е. количество его элементов.

```
int Rank { get; }                                                      Array
```

Возвращает *размерность* массива `this`, т. е. количество его индексов.

```
int GetLength(int dimension);                                          Array
```

Возвращает количество значений для индекса с номером `dimension` (индексы нумеруются от 0).

```
int GetUpperBound(int dimension);                                      Array
```

Возвращает максимальное значение для индекса с номером `dimension` (индексы нумеруются от 0). Имеется также функция `GetLowerBound`, которая возвращает 0 для всех массивов, описанных по стандартным правилам C#.

```
object Clone();                                                        Array
```

Возвращает копию массива `this`. Копия имеет тот же размер и содержит элементы с теми же значениями, что и исходный массив. Копия является *поверхностной* (shallow): если элемент массива `this` содержит ссылку на некоторый объект, то соответствующий элемент массива-копии будет содержать ссылку на тот же объект (иными словами, копирования объектов, на которые ссылаются элементы массива, при поверхностном копировании массива не происходит).

```
void CopyTo(Array a, int start);                                       Array
```

Копирует все элементы *одномерного* массива `this` в существующий одномерный массив `a` (заполняются элементы массива `a`, начиная с индекса `start`). Как и в методе `Clone`, копирование является поверхностным. Если массив `a` содержит недостаточно элементов, то возбуждается исключение.

3.2.2. Классовые методы

Все описанные ниже классовые методы класса `Array`, за исключением методов `Clear` и `Copy`, предназначены для обработки *одномерных* массивов. В .NET Framework версии 2.0 в класс `Array` было добавлено большое количество методов, которые используют в качестве параметров *обобщенные классы* (generic classes). Эти методы здесь не рассматриваются, поскольку вместо них можно применять средства, связанные с технологией LINQ, описанной в третьей части настоящей книги.

```
static void Clear(Array a, int start, int count);                    Array
```

Обнуляет `count` элементов массива `a`, начиная с элемента, имеющего индекс `start`. Если массив `a` является многомерным, то он интерпретируется как *одномерный* массив, причем перебор его элементов выполняется так, чтобы быстрее изменялись *правые* индексы элементов массива `a` (в частности, двумерные массивы-матрицы перебираются *по строкам*). Если массив `a` содержит недостаточно элементов, то возбуждается исключение `IndexOutOfRangeException`.

```
static void Copy(Array src, Array dest, int count);                  Array
static void Copy(Array src, int srcStart, Array dest,
                 int destStart, int count);
```

Копирует `count` элементов из массива `src` в массив `dest`. При наличии параметров `srcStart` и `destStart` чтение элементов в массиве `src` начинается с элемента с индексом `srcStart`, а запись в массив `dest` производится с элемента с индексом `destStart`. При отсутствии параметров `srcStart` и `destStart` чтение и запись выполняются с начала указанных массивов. Позволяет обрабатывать *многомерные* массивы с *одинаковым* количеством индексов; в этом случае перебор элементов производится так же, как и для метода `Clear`. Как и в методе `Clone`, копирование является поверхностным.

```
static void Reverse(Array a[, int start, int count]);                Array
```

Изменяет порядок следования элементов *одномерного* массива `a` на обратный. При наличии параметров `start` и `count` меняется порядок следования только той части массива, которая начинается с элемента с индексом `start` и состоит из `count` элементов. При неверном указании параметров `start` или `count` возбуждается исключение.

```
static void Sort(Array a[, Array b][, int start, int count][,        Array
                 System.Collections.IComparer cmp]);
```

Выполняет сортировку элементов одномерного массива `a` по возрастанию. При отсутствии параметра `cmp` элементы массива `a` должны реализовывать интерфейс `IComparable` (т. е. содержать метод `int CompareTo(object obj)`, позволяющий сравнивать объекты `this` и `obj`). Все элементарные типы языка C# (кроме класса `object`) реализуют интерфейс `IComparable`.

При наличии параметров `start` и `count` сортировка проводится только для `count` элементов массива, начиная с элемента с индексом `start`.

При наличии параметра-массива `b` порядок его элементов изменяется в соответствии с порядком элементов в отсортированном массиве `a` (таким образом, в этом случае массив `a` выступает в роли массива *ключей*, определяющих, в каком порядке располагать элементы в массиве `b`). Следует заметить, что взаимное расположение элементов массива `b`, соответствующих *одинаковым* ключам из массива `a`, в ходе сортировки может измениться.

При наличии параметра cmp сортировка выполняется в соответствии с правилом сравнения, заданным в методе int Compare(object x, object y) объекта cmp (данный метод должен быть определен в объекте cmp, поскольку cmp реализует интерфейс IComparer). При наличии параметра cmp элементы массива a не обязаны реализовывать интерфейс IComparable.

```
static int IndexOf(Array a, object value[,                    Array
                int start[, int count]]);
```

Осуществляет в *одномерном* массиве a поиск первого вхождения элемента value и возвращает его индекс (если элемент не найден, то возвращается –1). Параметр start определяет индекс элемента, начиная с которого выполняется поиск (при отсутствии параметра start поиск осуществляется с начального элемента массива). Параметр count определяет количество элементов, которые анализируются в ходе поиска (элементы перебираются по возрастанию индексов; при отсутствии параметра count поиск выполняется вплоть до последнего элемента массива). При неверном указании параметров start или count возбуждается исключение.

```
static int LastIndexOf(Array a, object value[,                Array
                int start[, int count]]);
```

Осуществляет в *одномерном* массиве a поиск последнего вхождения элемента value и возвращает его индекс (если элемент не найден, то возвращается –1). Параметр start определяет индекс элемента, начиная с которого выполняется поиск (при отсутствии параметра start поиск осуществляется с последнего элемента массива). Параметр count определяет количество элементов, которые анализируются в ходе поиска (элементы перебираются по убыванию индексов; при отсутствии параметра count поиск выполняется вплоть до первого элемента массива). При неверном указании параметров start или count возбуждается исключение.

В классе Array имеется классовый метод BinarySearch, также предназначенный для поиска элементов. Данный метод обеспечивает более быстрый *бинарный поиск*, однако может применяться только к *отсортированным по возрастанию* одномерным массивам. В настоящем пособии этот метод не рассматривается.

3.3. Динамический массив: класс List<T>

Динамические массивы, в отличие от обычных, могут изменять свой размер после создания (с сохранением своего прежнего содержимого). Возможности динамического массива реализованы в обобщенном классе List<T>, определенном в пространстве имен System.Collections.Generic.

Первоначально, в версии .NET 1.0, динамический массив был представлен обычным (необобщенным) классом ArrayList, определенным в пространстве имен System.Collections. В этом же пространстве имен были

определены и другие классы для работы с динамическими структурами данных, например: стек (Stack), очередь (Queue), хэш-таблица (Hashtable), ассоциативный массив, отсортированный по ключу (SortedList), динамический массив битов (BitArray). В .NET Framework 2.0, в связи с появившейся в этой версии возможностью создания обобщенных классов (generic classes, см. главу 9), были разработаны обобщенные варианты для всех коллекций. Эти классы определены в пространстве имен System.Collections.Generic; они обеспечивают создание коллекций, предназначенных для хранения данных определенного типа (необобщенные коллекции .NET 1.0 в качестве типа своих элементов использовали тип object). После появления обобщенных коллекций «старые» необобщенные коллекции практически вышли из употребления (хотя и остаются доступными во всех версиях .NET).

Ниже рассматриваются свойства и методы класса List<T>.

3.3.1. Свойства

```
int Capacity { get; set; }                                    List<T>
```

Возвращает и позволяет изменять *емкость* динамического массива, т. е. размер вспомогательного массива объектов, используемого для хранения элементов. Если в результате добавления в динамический массив новых элементов текущей емкости оказывается недостаточно, она автоматически увеличивается. Следует заметить, что при изменении емкости в памяти создается новый массив объектов (требуемого размера) и в него копируются элементы из прежнего массива, после чего прежний массив считается «мусором» и в дальнейшем уничтожается. Таким образом, изменение емкости объекта List<T> – достаточно ресурсоемкая операция. При попытке установить емкость массива меньшей его текущего размера возбуждается исключение ArgumentOutOfRangeException.

```
int Count { get; }                                            List<T>
```

Возвращает текущий *размер* динамического массива, т. е. количество фактически содержащихся в нем элементов. Данное свойство изменяется автоматически при добавлении элементов к массиву или при их удалении. Оно всегда меньше или равно емкости (Capacity) динамического массива.

```
T this[1 index] { get; set; }                                 List<T>
```

Возвращает и позволяет изменить элемент динамического массива с индексом index (индексация, как и в обычном массиве, начинается от 0). Данное свойство-*индексатор* в языке C# не имеет имени; параметр index в квадратных скобках должен указываться непосредственно после имени динамического массива, например: a[2].

Если значение параметра index является недопустимым (меньше 0 или больше Count - 1), то возбуждается исключение ArgumentOutOfRangeException.

3.3.2. Конструкторы и другие способы создания динамического массива

```
List<T>([int capacity]);                                    конструктор
```

Создает динамический массив заданной емкости capacity. Если параметр capacity отсутствует, то емкость полагается равной 0. Созданный массив не содержит элементов, т. е. его свойство Count равно 0.

```
List<T>(IEnumerable<T> c);                                  конструктор
```

Создает динамический массив и заполняет его элементами, взятыми из коллекции c (в том же порядке, в котором они размещаются в коллекции). Емкость массива при этом полагается равной его фактическому размеру.

В качестве параметра c можно указывать объект любого класса, реализующего интерфейс IEnumerable<T>; примерами таких классов являются обычные массивы, а также динамические массивы List<T> и другие виды обобщенных коллекций.

```
object Clone();                                             List<T>
```

Возвращает копию динамического массива this. Копия имеет ту же емкость, тот же размер и содержит элементы с теми же значениями, что и исходный массив. Как и в случае обычного массива Array, метод Clone осуществляет *поверхностное* копирование.

3.3.3. Совместимость массивов и динамических массивов

Обычные массивы и динамические массивы не совместимы по присваиванию. Для преобразования обычного массива к объекту List<T> необходимо воспользоваться соответствующим конструктором класса List<T>. Для обратного преобразования (объекта List<T> к обычному массиву) надо воспользоваться одним из описанных ниже методов класса List<T>. Заметим, что во всех этих методах выполняется *поверхностное* копирование элементов динамического массива в обычный (по поводу поверхностного копирования см. описание метода Clone для класса Array в п. 3.2.1).

```
T[] ToArray();                                              List<T>
```

Возвращает массив, содержащий все элементы динамического массива this. Размер созданного массива (Length) равен фактическому размеру (Count) динамического массива this.

```
void CopyTo(T[] array[, int arrayStart]);                   List<T>
void CopyTo(int start, T[] array, int arrayStart, int count);
```

Копирует count элементов из динамического массива this (начиная с элемента с индексом start) в одномерный массив array, записывая их в

массив array, начиная с элемента с индексом arrayStart. Если параметры start и count не указаны, то копирование начинается с первого элемента динамического массива (с индексом 0) и выполняется вплоть до последнего элемента динамического массива. Если параметр arrayStart не указан, то заполняются элементы массива array, начиная с первого элемента (с индексом 0).

Размер массива array должен быть достаточным для хранения всех копируемых элементов; при указании параметров start и count динамический массив this должен содержать элементы с индексами от start до start + count - 1. При нарушении любого из этих условий возбуждается исключение.

3.3.4. Преобразование динамического массива

```
void Add(T value);                                                    List<T>
void Insert(int index, T value);
```

Метод Add добавляет в конец динамического массива this объект value. Метод Insert вставляет объект value в позицию с индексом index.

При выполнении данных методов размер массива (Count) увеличивается на 1 и возможно увеличение емкости массива (Capacity). В случае, если T является ссылочным типом, параметр value может быть равен null. Если значение index меньше 0 или больше Count, то возбуждается исключение.

```
void AddRange(IEnumerable<T> c);                                      List<T>
void InsertRange(int start, IEnumerable<T> c);
```

Метод AddRange добавляет в конец динамического массива this все элементы из коллекции c. Метод InsertRange вставляет все элементы из коллекции c в массив this, начиная с позиции с индексом start.

При выполнении данных методов увеличивается размер массива (Count) и возможно увеличение емкости массива (Capacity). Если параметр c равен null, а также если значение start меньше 0 или больше Count, то возбуждается исключение.

```
bool Remove(T value);                                                 List<T>
```

Удаляет из динамического массива this первое вхождение элемента со значением value (данное значение может быть равно null); при этом размер массива (Count) уменьшается. Метод возвращает true, если элемент удален, и false, если элемент не найден.

```
void RemoveAt(int index);                                             List<T>
void RemoveRange(int start, int count);
```

Метод RemoveAt удаляет из динамического массива this элемент с индексом index. Метод RemoveRange удаляет из массива this count элементов,

начиная с элемента с индексом start. При удалении элементов размер массива уменьшается. Если параметр index отрицателен или больше или равен свойству Count, а также если параметры start и count задают диапазон индексов, не соответствующий существующим элементам массива this, то возбуждается исключение.

```
void Clear();                                                    List<T>
```

Удаляет из динамического массива this все элементы и полагает его размер Count равным 0. Емкость массива (Capacity) не изменяется.

```
void TrimExcess();                                               List<T>
```

Уменьшает емкость (Capacity) динамического массива this, полагая ее равной текущему размеру массива Count. Если массив не содержит элементов (т. е. свойство Count равно 0), то устанавливается емкость по умолчанию. Метод может не выполнять никаких действий, если текущий размер массива *почти* достигает текущей емкости.

```
void Reverse([int start, int count]);                            List<T>
```

Изменяет порядок следования элементов динамического массива this на обратный. При наличии параметров start и count меняется порядок следования только той части массива, которая начинается с элемента с индексом start и состоит из count элементов. При неверном указании параметров start или count возбуждается исключение.

```
void Sort([[int start, int count,]                               List<T>
                System.Collections.Generic.IComparer<T> cmp]);
```

Выполняет сортировку элементов динамического массива this по возрастанию. При отсутствии параметра cmp элементы массива должны реализовывать интерфейс IComparable, т. е. содержать метод int CompareTo(object obj), позволяющий сравнивать объекты this и obj. Вместо интерфейса IComparable может использоваться его обобщенный вариант IComparable<T>, содержащий метод int CompareTo(T obj). Все элементарные типы языка C# (кроме класса object) реализуют оба указанных интерфейса.

При наличии параметра cmp сортировка выполняется в соответствии с правилом сравнения, заданным в методе int Compare(T x, T y) объекта cmp (данный метод входит в интерфейс IComparer<T>, который должен реализовывать параметр cmp). Если параметр cmp указан, то элементы массива this не обязаны реализовывать интерфейсы IComparable или IComparable<T>.

При наличии параметров start и count сортировка проводится только для count элементов массива, начиная с элемента с индексом start.

Заметим, что если динамический массив отсортирован по возрастанию, то в нем, как и в обычном массиве, можно выполнять быстрый *бинарный поиск*, применяя для этого метод BinarySearch.

3.3.5. Поиск в динамическом массиве

```
bool Contains(T value);                                          List<T>
```

Возвращает `true`, если динамический массив `this` содержит элемент со значением `value`, и `false` в противном случае.

```
int IndexOf(T value[, int start[, int count]]);                  List<T>
int LastIndexOf(T value[, int start[, int count]]);
```

Метод `IndexOf` осуществляет в динамическом массиве `this` поиск первого вхождения элемента со значением `value`, а метод `LastIndexOf` – поиск последнего вхождения. По поводу возвращаемых значений и назначения параметров `start` и `count` см. описание одноименных методов класса `Array`.

3.4. Перебор элементов в цикле foreach

Удобным способом перебора элементов массива является оператор цикла `foreach`. В заголовке данного цикла указывается переменная (*параметр цикла*), в которую последовательно помещаются значения всех элементов массива, начиная с элемента с индексом 0. Для многомерных массивов перебор элементов выполняется таким образом, чтобы быстрее изменялись *правые* индексы. Из-за подобной организации цикла `foreach` элементы массива в нем можно использовать *только для чтения*. Параметр цикла `foreach` должен иметь тип, совместимый с типом элементов массива; его можно описать в заголовке цикла, в этом случае он будет существовать только при выполнении данного цикла.

Оператор `foreach` может использоваться не только для обычных массивов, но и для объектов других классов, например, для динамических структур данных (в частности, динамических массивов `List<T>`) и строк типа `string` (но не `StringBuilder`). Признаком допустимости использования цикла `foreach` для объекта является наличие у этого объекта интерфейса `IEnumerable`.

В качестве примера использования цикла `foreach` найдем с его помощью сумму всех элементов динамического массива `a` типа `List<int>` (предполагается, что переменная `s` имеет целый тип и инициализирована нулем):

```
foreach (int v in a)
    s += v;
```

Приведем пример решения этой же задачи с применением цикла `for`:

```
for (int i = 0; i < a.Count; i++)
    s += a[i];
```

Дополнительная информация о цикле `foreach` содержится в п. 8.4.

Глава 4. Строки

4.1. Класс string

Класс `string` предназначен для хранения строк с символами из текстового набора Unicode, причем размер строк на практике ограничивается только размерами доступной оперативной памяти. В настоящем пункте описываются все основные возможности класса `string`, за исключением *интернирования* строк (см., например, [7, гл. 12]).

4.1.1. Особенности инициализации. Неизменяемость объектов string

Простейший и самый распространенный способ инициализации объекта типа `string` – присваивание ему *литеральной строки* (т. е. строковой константы, заключенной в кавычки), например:

```
string s = "ABCD";
```

Для инициализации строки можно также использовать несколько вариантов конструктора. Приведем описание двух часто используемых вариантов.

```
string(char value, int count);                                конструктор
```

Создает строку длины `count`, состоящую из одинаковых символов `value`.

```
string(char[] value);                                         конструктор
```

Создает строку, состоящую из символов массива `value`.

При использовании этих и других конструкторов необходимо применять стандартный синтаксис с ключевым словом `new`, например:

```
string s = new string('A', 5);
```

Кроме того, строка может быть создана с помощью различных методов класса `string` (в частности, метода `Format`, описанного в п. 4.3.3), а также метода `ToString` класса `object` и всех его потомков.

После создания и размещения в памяти объект типа `string` *нельзя изменить*; в частности, доступ к символам строки возможен только для чтения. Любые методы, предназначенные для изменения строк, не модифицируют существующую строку, а создают *новую* строку с нужными изменениями.

Класс `string` является *ссылочным* типом, поэтому при присваивании строковых переменных происходит лишь копирование ссылок на одну и ту же неизменяемую строку, размещенную в памяти. Если же в программе, начиная с некоторого момента, не содержится ни одной ссылки на данный строковый объект, то этот объект считается «мусором» и удаляется из памяти при очередной активизации *сборщика мусора* – особой подсистемы среды выполнения .NET Framework (заметим, что подобным образом

уничтожаются и объекты других ссылочных типов). Более подробно механизм сборки мусора описывается во второй части книги (см. п. 8.1.2).

4.1.2. Управляющие последовательности и буквальные строки в C#

Строковые константы в C# могут содержать, помимо обычных символов, *управляющие последовательности* (escape characters), начинающиеся с символа «\». Приведем наиболее важные из управляющих последовательностей:

`\\` – символ «\»;

`\'` – символ «'»;

`\"` – символ «"»;

`\0` – символ с кодом 0;

`\t` – символ табуляции (код 9);

`\n` – символ «переход на новую строку» (код 10);

`\r` – символ «возврат каретки» (код 13);

`\u`*N* – символ Unicode с шестнадцатеричным кодом *N* (*N* должен состоять из 4 цифр и может меняться от `0000` до `FFFF`). Например, символ `'\u0041'` обозначает латинскую букву «A».

Если строка содержит много обычных символов «\», то «экранирование» каждого из них дополнительным символом «\» приводит к тексту, сложному для восприятия, например:

```
string name = "C:\\Windows\\System32\\Notepad.exe";
```

Кроме того, возрастает риск сделать опечатку, пропустив один из экранирующих символов «\». В подобной ситуации удобнее использовать специальное средство языка C#: *буквальные строки* (verbatim strings). Если перед двойной кавычкой «"», открывающей строковую константу, указать символ «@», то все символы до завершающей двойной кавычки будут интерпретироваться как обычные символы. С использованием буквальной строки приведенный выше оператор языка C# примет более наглядный вид:

```
string name = @"C:\Windows\System32\Notepad.exe";
```

Если в буквальной строке надо указать символ «"» (двойная кавычка), то следует ввести его дважды.

4.1.3. Поля и свойства

Класс `string` имеет одно классовое поле и два экземплярных свойства. И поле, и свойства класса `string` доступны только для чтения.

```
static readonly string Empty;                                    string
```

Данное поле возвращает пустую строку. Вместо поля `string.Empty` в языке C# допустимо использовать литеральную строку `""` (две подряд расположенные двойные кавычки).

```
int Length { get; }                                              string
```

Возвращает текущую *длину* строки `this` (т. е. количество содержащихся в ней символов).

```
char this[int index] { get; }                                    string
```

Возвращает символ строки `this` с индексом `index` (символы строки, подобно элементам массива, индексируются от 0). Данное свойство-*индексатор* в языке C# не имеет имени; параметр `index` в квадратных скобках должен указываться непосредственно после имени объекта-строки, например: `s[2]`. При указании недопустимого индекса (меньшего 0 или большего `Length - 1`) возбуждается исключение `IndexOutOfRangeException`.

Отметим, что в качестве индексируемого объекта-строки можно указывать не только переменную, но и любое *выражение* типа `string`. Например, `"ABCD"[2]` вернет символ «C». Экземплярные методы класса `string` также можно применять к любому строковому выражению.

4.1.4. Сравнение строк

Все методы, сравнивающие две строки (`Compare`, `CompareOrdinal` и `CompareTo`), возвращают отрицательное число, если первая строка меньше второй, число 0, если строки равны, и положительное число, если первая строка больше второй (строки сравниваются *посимвольно*). Значение `null` считается меньшим, чем пустая строка `""` (строка длины 0); пустая строка считается меньшей любой непустой строки; строка, все символы которой совпадают с начальными символами более длинной строки, считается меньшей, чем эта более длинная строка.

В языке C# сравнение строк на равенство/неравенство может проводиться с помощью операций `==` и `!=` (при этом учитывается регистр и не учитываются региональные настройки); операции `<`, `<=`, `>`, `>=` для строк *не определены*.

```
static int Compare(string strA, string strB[,                   string
                bool ignoreCase[, CultureInfo culture]]);
static int Compare(string strA, int startA,
                string strB, int startB, int count[,
                bool ignoreCase[, CultureInfo culture]]);
```

Сравнивает строки `strA` и `strB` с учетом региональных настроек.

При наличии параметров `startA`, `startB` и `count` сравниваются подстроки строк `strA` и `strB`, начинающиеся с символов с индексами `startA` и `startB`

соответственно, причем во внимание принимаются не более count символов.

Если параметр ignoreCase не указан или равен false, то при сравнении строк учитывается регистр букв; если указан параметр ignoreCase, равный true, то регистр не учитывается. При наличии параметра culture при сравнении принимаются во внимание региональные настройки, заданные этим параметром (если данный параметр отсутствует, то учитываются текущие региональные настройки).

```
static int CompareOrdinal(string strA, string strB);                    string
static int CompareOrdinal(string strA, int startA,
                string strB, int startB, int count);
```

Сравнивает строки strA и strB путем сравнения *кодов* символов. Таким образом, данный метод не учитывает региональные настройки, но всегда учитывает регистр букв. Это самый быстрый из методов сравнения.

Смысл параметров startA, startB и count – тот же, что и для метода Compare.

```
int CompareTo(string str);                                               string
bool StartsWith(string str);
bool EndsWith(string str);
```

Метод CompareTo сравнивает строки this и str. Метод StartsWith возвращает true, если строка this начинается со строки str, и false в противном случае. Метод EndsWith возвращает true, если строка this оканчивается строкой str, и false в противном случае. Все три метода учитывают регистр букв и текущие региональные настройки.

4.1.5. Поиск в строке

Поиск в строках осуществляется с учетом регистра и без учета региональных настроек.

```
int IndexOf(char value[, int start[, int count]]);                       string
int IndexOf(string value[, int start[, int count]]);
```

Осуществляет в строке this поиск первого вхождения символа (или строки) value и возвращает индекс найденного символа (или начального символа найденной строки); если ни одного вхождения value не найдено, то возвращается –1. Параметр start определяет индекс символа, начиная с которого выполняется поиск (при отсутствии параметра start поиск осуществляется с начала строки). Параметр count определяет количество символов в строке this, которые анализируются в ходе поиска (символы перебираются в направлении от начала к концу строки; при отсутствии параметра count просматривается вся строка). Если параметр value является пустой строкой "", то метод возвращает значение параметра start (или 0, если параметр start не указан). При указании неверных параметров start

или count, а также в случае параметра value, равного null, возбуждается исключение.

```
int LastIndexOf(char value[, int start[, int count]]);          string
int LastIndexOf(string value[, int start[, int count]]);
```

Осуществляет в строке this поиск последнего вхождения символа (или строки) value и возвращает индекс найденного символа (или начального символа найденной строки); если ни одного вхождения не найдено, то возвращается –1. Параметр start определяет индекс символа, начиная с которого выполняется поиск (при отсутствии параметра start поиск осуществляется с конца строки). Параметр count определяет количество символов, которые анализируются в ходе поиска (символы перебираются в направлении от конца к началу строки; при отсутствии параметра count просматривается вся строка). Если параметр value является пустой строкой, то метод возвращает значение параметра start (или индекс последнего символа в строке, если параметр start не указан). При указании неверных параметров start или count, а также в случае параметра value, равного null, возбуждается исключение.

```
int IndexOfAny(char[] values[, int start[, int count]]);        string
```

Осуществляет в строке this поиск любого из символов, указанных в массиве values, и возвращает индекс первого найденного символа (или –1, если ни один из символов в строке не обнаружен). Параметры start и count имеют тот же смысл, что и для метода IndexOf.

```
int LastIndexOfAny(char[] values[, int start[, int count]]);    string
```

Осуществляет в строке this поиск любого из символов, указанных в массиве values, и возвращает индекс последнего найденного символа (или –1, если ни один из символов в строке не обнаружен). Параметры start и count имеют тот же смысл, что и для метода LastIndexOf.

При организации сложных видов поиска в строке обычно используются *регулярные выражения* (см. главу 5).

4.1.6. Преобразование строки

Поскольку объекты типа string являются неизменяемыми, все методы, связанные с преобразованием строки, не изменяют исходную строку, а возвращают новую строку с нужными изменениями.

```
string Insert(int start, string str);                           string
```

Возвращает строку, полученную из строки this вставкой в нее строки str (вставка осуществляется в позицию с индексом start). Если параметр start равен длине строки this, то строка str добавляется в конец строки this; если параметр start отрицателен или превосходит длину строки this, то возбуждается исключение ArgumentOutOfRangeException.

```
string Remove(int start[, int count]);                                    string
```

Возвращает строку, полученную из строки this удалением из нее count символов, начиная с символа с индексом start. Если параметр count не указан, то удаляются все символы вплоть до конца строки.

```
string Replace(char value, char newValue);                                string
string Replace(string value, string newValue);
```

Возвращает строку, полученную из строки this заменой всех вхождений символа (или строки) value на символ (или, соответственно, строку) newValue. Поиск осуществляется с учетом регистра; в случае поиска строки учитываются текущие региональные настройки. Во вставленных фрагментах newValue повторный поиск фрагментов value не производится. Если в качестве newValue указана пустая строка "" или null, то результатом работы метода является удаление всех вхождений строки value; если в качестве value указана пустая строка или null, то возбуждается исключение.

Более сложные виды замены осуществляются обычно с помощью регулярных выражений (см. главу 5).

```
string Substring(int start[, int count]);                                 string
```

Возвращает подстроку строки this. Если параметр count указан, то подстрока содержит count символов строки this, начиная с символа с индексом start; если параметр count не указан, то подстрока содержит все символы строки this, начиная с символа с индексом start. Если параметр start равен длине строки или count равен 0, то возвращается пустая строка. Если параметр start отрицателен или превышает длину строки, а также если в строке содержится менее count символов, начиная с позиции start, то возбуждается исключение ArgumentOutOfRangeException.

```
string PadLeft(int total[, char padChar]);                                string
string PadRight(int total[, char padChar]);
```

Метод PadLeft возвращает строку this, дополненную слева символами padChar так, чтобы суммарная длина строки стала равна total. Если параметр padChar не указан, то строка дополняется слева пробелами. Метод PadRight работает аналогично, за исключением того, что символы padChar добавляются к исходной строке справа. Если значение параметра total меньше исходной длины строки this, то любой из данных методов возвращает строку, равную строке this.

```
string Trim([params char[] trimChars]);                                   string
string TrimStart([params char[] trimChars]);
string TrimEnd([params char[] trimChars]);
```

Метод Trim возвращает строку, полученную из строки this удалением начальных и конечных символов, входящих в массив trimChars. Методы TrimStart и TrimEnd работают аналогично, за исключением того, что TrimStart удаляет символы только из начала строки, а TrimEnd – только из ее

конца. Атрибут params означает, что вместо параметра-массива можно указывать *произвольное* количество параметров типа char. Методы допустимо вызывать без параметров; в этом случае удаляются все *пробельные символы* (т. е. пробелы, символы табуляции, перехода на новую строку и т. п.).

```
string ToLower([CultureInfo culture]);                                    string
string ToUpper([CultureInfo culture]);
```

Метод ToLower возвращает строку, полученную из строки this преобразованием всех букв к *нижнему* регистру; метод ToUpper возвращает строку, полученную из строки this преобразованием всех букв к *верхнему* регистру. Если указан параметр culture, то при преобразовании учитываются региональные настройки, определяемые данным параметром; если параметр не указан, то используются текущие региональные настройки. И в случае региональных настроек для России ("ru-RU"), и в случае настроек для США ("en-US") данные методы правильно обрабатывают как латинские, так и русские буквы.

4.1.7. Объединение и разбиение строк

Для *сцепления* (*конкатенации*) нескольких строк в языке C# предусмотрена операция +. При сцеплении строки s и объекта x *другого типа* (в любом порядке) автоматически выполняется преобразование объекта x к его строковому представлению, т. е. для x вызывается метод ToString.

```
static string Concat(params object[] a);                                  string
```

Возвращает строку, полученную в результате сцепления строковых представлений всех элементов массива a. Атрибут params означает, что вместо параметра-массива можно указывать *произвольное* количество параметров типа object, т. е. фактически *любого типа*, так как любой тип является потомком класса object.

```
static string Join(string sep, string[] a[,                               string
                int start, int count]);
```

Возвращает строку, полученную в результате объединения элементов массива строк a, при котором между соседними элементами помещается строка-*разделитель* sep. Если параметры start и count указаны, то выполняется объединение count элементов массива a, начиная с элемента с индексом start. Если параметры start и count не указаны, то выполняется объединение всех элементов массива a.

```
char[] ToCharArray([int start, int count]);                               string
```

Возвращает массив символов, входящих в строку this. При наличии параметров start и count в массив помещается count символов, начиная с символа с индексом start; если параметры отсутствуют, то в массив помещаются все символы строки. Если параметр count равен 0, то возвращается массив нулевого размера (но не равный null). Если параметр start отрица-

телен или превышает длину строки, а также если в строке содержится менее count символов, начиная с позиции start, то возбуждается исключение ArgumentOutOfRangeException.

```
void CopyTo(int start, char[] array, int arrayStart,                    string
                int count);
```

Копирует count символов из строки this (начиная с символа с индексом start) в символьный массив array, записывая их в элементы массива array, начиная с элемента с индексом arrayStart. Если числовые параметры отрицательны или после стартовых элементов в строке this или массиве array содержится менее count элементов, то возбуждается исключение ArgumentOutOfRangeException.

```
string[] Split([params char[] sep]);                                    string
string[] Split(char[] sep, int count);
string[] Split(char[] sep[, int count], StringSplitOptions opt);
string[] Split(string[] sep[, int count], StringSplitOptions opt);
```

Разбивает строку this на подстроки, разделенные символами (или строками), указанными в массиве sep, и возвращает массив полученных подстрок. Если указан параметр count, то возвращается не более count первых найденных подстрок (если count отрицателен, то возбуждается исключение ArgumentOutOfRangeException). Параметр opt может принимать одно из двух значений перечислимого типа StringSplitOptions: None и RemoveEmptyEntries. При значении StringSplitOptions.RemoveEmptyEntries в массив подстрок не помещаются пустые строки (т. е. подряд идущие разделители считаются одним разделителем, а начальные и завершающие разделители игнорируются); при значении StringSplitOptions.None или при отсутствии параметра opt массив подстрок может содержать пустые строки (если в строке this располагается подряд несколько разделителей или строка this содержит начальные или завершающие разделители). Сами разделители в результирующие подстроки не включаются.

Атрибут params означает, что вместо одного параметра-массива в данном варианте метода можно указывать *произвольное* количество параметров типа char.

При поиске разделителей не учитываются региональные настройки и учитывается регистр символов. Если массив разделителей равен null или метод Split вызван без параметров, то разделителем считается любой *пробельный символ*. В случае строкового массива разделителей важным является *порядок* его элементов, поскольку при поиске разделителей элементы массива sep перебираются по возрастанию индексов. Например, если в массиве разделителей содержатся строки "B" и "BC" в указанном порядке, то строка "ABCD" будет разбита на подстроки "A" и "CD", а если изменить порядок строк в массиве sep, то будут возвращены подстроки "A" и "D".

Для разбиения строк можно также использовать регулярные выражения (см. главу 5).

4.2. Класс StringBuilder

Основной особенностью объектов класса `StringBuilder` является *возможность их изменения*, в том числе *посимвольного* (это отличает данный класс от класса `string`, объекты которого являются неизменяемыми). Класс `StringBuilder` определен в пространстве имен `System.Text`.

4.2.1. Свойства и конструкторы

```
int Capacity { get; set; }                                    StringBuilder
```

Возвращает и позволяет изменять *емкость* строки, т. е. размер массива символов, выделенного для ее хранения. По умолчанию емкость равна 16. Если в результате преобразования строки текущей емкости оказывается недостаточно, она автоматически удваивается. Следует заметить, что при изменении емкости в памяти создается новый массив символов (требуемого размера) и в него копируются символы строки из прежнего массива, после чего прежний массив считается «мусором» и в дальнейшем уничтожается. Таким образом, изменение емкости строки `StringBuilder` – достаточно ресурсоемкая операция.

```
int MaxCapacity { get; }                                      StringBuilder
```

Возвращает максимальную емкость строки. По умолчанию равна значению `int.MaxValue`. Другое ее значение может быть указано в конструкторе при создании объекта типа `StringBuilder`. Изменить значение свойства `MaxCapacity` уже созданного объекта нельзя.

```
int Length { get; set; }                                      StringBuilder
```

Возвращает и позволяет изменить текущую *длину* строки, т. е. количество фактически содержащихся в ней символов. При уменьшении свойства `Length` строка урезается справа, при увеличении – дополняется справа пробелами. Увеличение свойства `Length` может приводить к увеличению емкости (`Capacity`) строки. При попытке задать свойству `Length` отрицательное значение или значение, большее максимальной емкости (`MaxCapacity`), возбуждается исключение.

```
char this[int index] { get; set; }                            StringBuilder
```

Возвращает и позволяет изменить символ строки с индексом `index` (символы, подобно элементам массива, индексируются от 0). Данное свойство-индексатор в языке C# не имеет имени; параметр `index` в квадратных скобках должен указываться непосредственно после имени объекта-строки, например: `s[2]`. Данное свойство отличается от аналогичного свойства-индексатора класса `string` тем, что доступно как для чтения, так и для записи.

Если параметр index является недопустимым (т. е. меньше 0 или больше Length - 1), то возбуждается исключение IndexOutOfRangeException (при доступе на чтение) или ArgumentOutOfRangeException (при доступе на запись).

```
StringBuilder([int capacity[, int maxCapacity]]);                конструктор
```

Создает объект типа StringBuilder заданной емкости capacity и максимальной емкости maxCapacity. Если параметр maxCapacity отсутствует, то максимальная емкость полагается равной int.MaxValue. Если параметр capacity отсутствует, то емкость полагается равной 16. Созданный объект является пустой строкой, т. е. его свойство Length равно 0.

```
StringBuilder(string value[[, int start, int count],          конструктор
                int capacity]);
```

Создает объект типа StringBuilder заданной емкости capacity и инициализирует его строкой value или, при наличии параметров start и count, подстрокой строки value длины count, начинающейся с символа с индексом start. Параметр value может быть равен null, в этом случае созданный объект является пустой строкой. При указании неверных значений параметров start или count возбуждается исключение ArgumentOutOfRangeException. Если параметр capacity не указан или его значение меньше, чем длина L созданной строки, то емкость полагается равной числу $\max\{16, 2^N\}$, где 2^N – минимальная степень, большая или равная L.

4.2.2. Совместимость объектов string и StringBuilder

Объекты string и StringBuilder не совместимы по присваиванию. Для преобразования «обычной» строки (типа string) к объекту StringBuilder необходимо воспользоваться соответствующим конструктором класса StringBuilder. Для обратного преобразования (объекта StringBuilder к обычной строке) надо воспользоваться методом ToString класса StringBuilder, описанным ниже.

```
string ToString([int start, int count]);                         StringBuilder
```

Возвращает строку, содержащуюся в объекте this типа StringBuilder. При наличии параметров возвращает подстроку длины count, которая начинается с символа с индексом start. При указании неверных значений параметров возбуждается исключение ArgumentOutOfRangeException.

Метод ToString выполняется быстро и не приводит к копированию символов исходной строки типа StringBuilder: созданная строка типа string просто ссылается на тот же вспомогательный массив символов (или часть массива), что и исходная строка StringBuilder. Однако *изменение* строки StringBuilder *после* вызова ее метода ToString приводит к копированию ее символов в новый вспомогательный массив (для сохранения постоянства строки, возвращенной методом ToString). Поэтому для повышения

эффективности программы желательно не изменять строку StringBuilder после вызова ее метода ToString (или, иначе говоря, желательно завершить формирование строки StringBuilder перед вызовом ее метода ToString).

4.2.3. Преобразование строки типа StringBuilder

Текст *числовой_тип*, используемый при описании параметра value, обозначает любой из числовых типов.

```
StringBuilder Append(object value);                                    StringBuilder
StringBuilder Append(bool value);
StringBuilder Append(числовой_тип value);
StringBuilder Append(char value[, int charCount]);
StringBuilder Append(char[] value[, int start, int count]);
StringBuilder Append(string value[, int start, int count]);
StringBuilder AppendLine([string value]);
```

Добавляет к объекту-строке this типа StringBuilder строковое представление параметра value (или элементы символьного массива, если value имеет тип char[]) и возвращает измененный объект this. При наличии параметра charCount добавляет к строке charCount копий символа value. Параметры start и count определяют, какая часть символьного массива (или строки) будет добавлена к объекту this (добавляется count элементов, начиная с элемента с индексом start); при отсутствии этих параметров добавляются все элементы массива (соответственно, вся строка). Для получения строкового представления параметра value числового типа, типа bool и object используется метод ToString данного типа. Метод AppendLine добавляет к объекту this маркер конца строки EOLN (при наличии параметра value данный параметр добавляется к объекту this *перед* добавлением маркера EOLN).

Для добавления данных к объекту-строке типа StringBuilder можно также использовать метод AppendFormat, который описывается в п. 4.3.3.

Добавление данных к строке StringBuilder – более эффективная операция, чем сцепление обычных строк, поэтому сцепление заранее неизвестного числа строк рекомендуется проводить с помощью их добавления к строке StringBuilder.

```
StringBuilder Insert(int index, object value);                         StringBuilder
StringBuilder Insert(int index, bool value);
StringBuilder Insert(int index, числовой_тип value);
StringBuilder Insert(int index, char value);
StringBuilder Insert(int index, char[] value[, int start, int count]);
StringBuilder Insert(int index, string value[, int stringCount]);
```

Вставляет в строку this типа StringBuilder, начиная с позиции index, строковое представление параметра value (или элементы символьного массива, если value имеет тип char[]) и возвращает измененный объект this.

При наличии параметра stringCount вставляет stringCount копий строки value. Если указаны параметры start и count, то вставляет count элементов массива, начиная с элемента с индексом start (если параметры не указаны, то вставляет все элементы массива). Для получения строкового представления параметра value типа bool, object и числовых типов используется метод ToString данного типа. Если index меньше 0 или больше Length, то возбуждается исключение ArgumentOutOfRangeException.

```
StringBuilder Remove(int start, int count);                    StringBuilder
```

Удаляет из объекта-строки this count символов, начиная с символа с индексом start и возвращает измененный объект this. При выполнении данного метода емкость (Capacity) объекта this не изменяется, а длина (Length) уменьшается на значение параметра count. Если start меньше 0 или сумма start + count больше Length, то возбуждается исключение ArgumentOutOfRangeException.

```
StringBuilder Replace(char value, char newValue[,              StringBuilder
                int start, int count]);
StringBuilder Replace(string value, string newValue[,
                int start, int count]);
```

Заменяет в объекте-строке this все вхождения символа (или строки) value на символ (или, соответственно, строку) newValue и возвращает измененный объект this. Поиск вхождений выполняется без учета региональных настроек и с учетом регистра. При наличии параметров start и count преобразованию подвергается только часть объекта-строки длины count, начиная с символа с индексом start. Параметр newValue может быть равен null или пустой строке ""; в этом случае все вхождения value удаляются. Если параметр value равен null или является пустой строкой, а также если start меньше 0 или сумма start + count больше Length, то возбуждается исключение.

4.2.4. Дополнительные методы класса StringBuilder

```
void CopyTo(int start, char[] array,                           StringBuilder
                int arrayStart, int count);
```

Копирует count символов из строки this (начиная с символа с индексом start) в символьный массив array, записывая их в элементы массива array, начиная с элемента с индексом arrayStart. Если числовые параметры отрицательны или после стартовых элементов в строке this или массиве array содержится менее count элементов, то возбуждается исключение ArgumentOutOfRangeException.

```
int EnsureCapacity(int capacity);                              StringBuilder
```

Изменяет емкость (Capacity) объекта this следующим образом: если текущая емкость меньше, чем значение параметра, то она полагается рав-

ной данному параметру; в противном случае емкость объекта не изменяется. Метод возвращает новое значение емкости объекта `this`.

4.3. Форматирование данных

4.3.1. Форматирование по умолчанию и явное форматирование

Помимо форматирования по умолчанию, которое преобразует объект к его стандартному строковому представлению и обеспечивается методом `ToString` (без параметров), многие типы допускают явное форматирование. Механизм явного форматирования основан на использовании двух настроечных параметров: *форматной строки* и *регионального стандарта*, поэтому типы, допускающие явное форматирование, снабжаются перегруженным методом `ToString` с двумя параметрами:

```
string ToString(string fmt, IFormatProvider p);
```

Первый параметр (`fmt`) определяет форматную строку, а второй (`p`) – региональные настройки. Данный вариант метода `ToString` входит в интерфейс `IFormattable`, который реализуют все классы, допускающие явное форматирование. Кроме того, многие классы (в частности, все числовые типы, класс `DateTime` и все перечислимые типы) реализуют варианты метода `ToString`, содержащие только один из указанных параметров. В следующих двух пунктах основное внимание будет уделено форматным строкам.

4.3.2. Спецификаторы формата

Простейшим вариантом форматной строки является *спецификатор формата*, задаваемый единственным символом (например, `"D"`), который может быть дополнен *спецификатором точности* (например, `"D6"`). Приведем основные числовые спецификаторы формата.

`C` или `c` – *денежный формат* (currency); спецификатор точности определяет количество дробных знаков. По умолчанию это количество, а также обозначение и размещение знака валюты определяется региональными настройками.

`D` или `d` – *десятичный целочисленный формат* (decimal); спецификатор точности определяет минимальное количество цифр в числе, при необходимости число дополняется слева нулями.

`E` или `e` – *экспоненциальный числовой формат* (exponential); спецификатор точности определяет количество дробных знаков, по умолчанию это количество равно 6. Вид десятичного разделителя определяется региональными настройками; регистр буквы `e`, предваряющей порядок числа, совпадает с регистром спецификатора формата.

`F` или `f` – *числовой формат с фиксированной точкой* (fixed-point); спецификатор точности определяет количество дробных знаков. Количество

дробных знаков по умолчанию и вид десятичного разделителя определяются региональными настройками.

P или p – *процентный формат* (percent); спецификатор точности определяет количество дробных знаков. Число дробных знаков по умолчанию, вид десятичного разделителя и размещение знака % определяются региональными настройками.

X или x – *шестнадцатеричный целочисленный формат* (hexadecimal); спецификатор точности определяет минимальное количество цифр в числе, при необходимости число дополняется слева нулями. Регистр букв, используемых в шестнадцатеричном представлении, совпадает с регистром спецификатора формата.

G или g – *общий формат* (general), в котором используется наиболее краткое из возможных представлений числа данного типа. В частности, для вещественных чисел в качестве G выбирается более короткий из форматов E или F.

В качестве примера в таблице 5 приводятся варианты строкового представления числа 246 для настроек "en-US" (США) и "ru-RU" (Россия).

Таблица 5

Варианты строкового представления числа 246

	en-US	ru-RU
C	$246.00	246,00р.
C1	$246.0	246,0р.
C0	$246	246р.
D	246	246
D2	246	246
D4	0246	0246
E	2.460000E+002	2,460000E+002
e1	2.5e+002	2,5e+002
E0	2E+002	2E+002

	en-US	ru-RU
F	246.00	246,00
f1	246.0	246,0
F0	246	246
P	24,600.00 %	24 600,00%
P1	24,600.0 %	24 600,0%
P0	24,600 %	24 600%
X	F6	F6
x2	f6	f6
X4	00F6	00F6

Большое количество спецификаторов формата предусмотрено для типа DateTime. Среди них d (дата в кратком формате), D (дата в полном формате), t (время в кратком формате), T (время в полном формате), g (дата и время в кратком формате), G (дата в кратком формате, время – в полном), f (дата в полном формате, время – в кратком), F (дата и время в полном формате), M или m (формат «месяц, день»), Y или y (формат «месяц, год»). Вид этих форматов существенно зависит от региональных настроек.

Укажем также спецификаторы формата для *перечислимых типов* (т. е. потомков класса Enum): G (отображение имени перечислимой константы), D (отображение десятичного числа, соответствующего перечислимой кон-

станте), X (отображение шестнадцатеричного числа, соответствующего перечислимой константе).

Общий формат G определен для *всех* типов, допускающих форматирование. Если форматная строка в методе ToString не указана, является пустой строкой "" или равна null, то объект отображается в формате G.

Следует заметить, что для числовых данных и данных типа DateTime можно настраивать формат, используя так называемые *символы-заполнители*. Однако в большинстве случаев форматирования вполне достаточно тех возможностей, которые предоставляются стандартными спецификаторами формата.

4.3.3. Одновременное форматирование нескольких объектов: метод Format

Для одновременного форматирования нескольких объектов с помощью одной форматной строки основным инструментом является метод Format класса string:

```
static string Format([IFormatProvider p,] string fmt,          string
                params object[] args);
```

Параметр args не обязательно должен быть «настоящим» массивом объектов; атрибут params означает, что вместо одного этого параметра можно указывать произвольное число параметров типа object, т. е. фактически *любого типа*, поскольку любой тип является потомком класса object. При выполнении метода все эти параметры считаются элементами *одного массива* args, длина которого равна количеству параметров. Отметим, что имеются перегруженные варианты метода Format, в которых вместо параметра-массива args указывается один, два или три параметра типа object (эти варианты обеспечивают генерацию более эффективного исполняемого кода, поскольку в них не используется вспомогательный массив).

Если указан параметр p, то при форматировании учитываются региональные настройки, соответствующие этому параметру; если параметр p отсутствует, то используются текущие региональные настройки.

В *форматной строке* fmt можно указывать обычный текст и *форматные настройки* для каждого из форматируемых параметров. Эти настройки имеют вид

{*ind*[,*width*][:*spec*]}

Опишем *атрибуты*, входящие в форматные настройки:

ind – целое число, которое определяет индекс форматируемого элемента в массиве args (индексация ведется от 0; данный атрибут является обязательным);

width – целое число, модуль которого задает минимальную *ширину поля вывода* (т. е. минимальное число позиций, отводимое для форматируе-

мого элемента), а знак определяет *способ выравнивания* элемента в пределах поля вывода;

spec – строка, которая задает *спецификатор формата* для данного элемента (эта строка может также содержать формат, явно определяемый с помощью *символов-заполнителей*). Между двоеточием и строкой *spec* ***не должно быть пробелов***.

Если атрибут *width* отсутствует, то используется ширина поля вывода, минимально необходимая для отображения отформатированного элемента. Если отформатированный элемент не занимает всего поля вывода, то он дополняется пробелами: пробелы добавляются слева, если атрибут *width* положителен, и справа, если атрибут *width* отрицателен. Таким образом, при положительном значении *width* выполняется *выравнивание по правой границе* поля вывода, а при отрицательном – *по левой границе*.

Если атрибут *spec* отсутствует, то выбирается вариант форматирования *по умолчанию* (этот вариант соответствует спецификатору формата G).

Для обрамления каждой форматной настройки используются фигурные скобки. Если требуется указать фигурную скобку { или } в обычном тексте, входящем в строку fmt, ее надо ввести дважды: {{ или }}.

При ошибочном задании форматной строки fmt (в частности, при указании недопустимого значения атрибута *ind*) возбуждается исключение FormatException.

Приведем пример. При вызове метода

```
string.Format("Формат D:{0, 6:D4}, формат X:{0, 6:X4}", 246)
```

будет возвращена следующая строка:

```
Формат D:  0246, формат X:  00F6
```

Одновременное форматирование нескольких объектов может выполняться и в некоторых других методах. Например, подобной возможностью обладает метод AppendFormat класса StringBuilder:

```
StringBuilder AppendFormat([IFormatProvider p,]          StringBuilder
                string fmt, params object[] args);
```

Данный метод выполняет форматирование массива объектов args с помощью форматной строки fmt (учитывая региональные настройки, если они переданы с помощью параметра p), после чего добавляет сформированную строку к объекту this, вызвавшему этот метод. Метод возвращает новое значение объекта this.

Возможность форматирования обеспечивается также в методах, связанных с выводом данных в текстовые потоки (см. описания методов Write и WriteLine классов StreamWriter и Console в п. 6.4.4–6.4.5).

Заметим, что метод Format может использоваться и для сцепления строк, причем делает это более эффективно, чем обычная операция сцепления.

4.4. Кодирование и декодирование символьных данных

4.4.1. Кодирование данных по умолчанию: формат UTF-8

Приложения платформы .NET *кодируют* символы и строки при записи их в файл. Основной целью кодирования является уменьшение размера файлов с текстовой информацией (без потери самой информации); кроме того, выбор подходящего способа кодирования необходим, если сохраняемые текстовые данные в дальнейшем предполагается обрабатывать с помощью приложений, использующих какую-либо определенную символьную кодировку. При считывании символьной информации из файлов выполняется ее *декодирование*. В этой ситуации важно правильно указать кодировку символьных данных, используемую в файле, так как в противном случае данные будут декодированы неверно.

По умолчанию при записи символьных данных в файлы применяется формат кодирования UTF-8 (аббревиатура UTF расшифровывается как Unicode Transformation Format, т. е. «формат преобразования Unicode-символов»). В этом формате для хранения символов с кодами от 0 до 127 (*символов ASCII*) используется *один байт*, содержащий код символа (один байт равен 8 битам – именно поэтому в имени данного формата указано число 8). Таким образом, при кодировании этих символов размер необходимой для их хранения памяти уменьшается вдвое (напомним, что в кодировке Unicode, используемой в приложениях .NET, каждый символ, за исключением суррогатных пар, занимает 2 байта). Подобное уменьшение важно по двум причинам. Во-первых, символы ASCII – это наиболее часто используемые символы, поскольку в их число входят цифры, латинские буквы, стандартные знаки препинания и знаки операций, поэтому экономия дискового пространства оказывается довольно значительной. Во-вторых, текст, содержащий *только* символы ASCII в 8-битной кодировке, будет правильно обрабатываться как .NET-приложениями (которые при этом автоматически выполняют его обратное декодирование к стандартному формату Unicode), так и традиционными Windows-приложениями (которые интерпретируют данный текст как текст в «обычной» ASCII-кодировке).

Символы Unicode с кодами от 128 до 2047 (содержащие символы различных европейских и среднеазиатских языков) преобразуются при кодировании UTF-8 в 2 байта, а прочие символы преобразуются в 3 или 4 байта.

Если требуется прочесть или сохранить символьные данные в другом формате, то формат кодирования необходимо явно указать в конструкторах соответствующих потоков-оболочек для чтения (типа `BinaryReader` или `StreamReader`) или для записи (типа `BinaryWriter` или `StreamWriter`).

4.4.2. Явная установка формата кодирования: класс Encoding

Для указания формата кодирования используется класс `Encoding`, определенный в пространстве имен `System.Text` и предоставляющий, наряду с другими свойствами и методами, набор классовых свойств и методов, задающих различные форматы кодирования. Перечислим самые важные из них.

```
static Encoding UTF8 { get; }                                    Encoding
```

Возвращает описанный выше формат кодирования UTF-8 (при файловом вводе-выводе используется по умолчанию).

```
static Encoding Unicode { get; }                                 Encoding
```

Возвращает формат кодирования UTF-16, переводящий символы в Unicode-кодировку. При чтении/записи символов Unicode с использованием данного формата кодирования никакого преобразования символов не происходит. Это обеспечивает более высокую скорость операций ввода-вывода (по сравнению с операциями, использующими другие форматы кодирования), но приводит к увеличению размеров файла.

```
static Encoding Default { get; }                                 Encoding
```

Возвращает формат кодирования, соответствующий той кодовой странице 8-битной ANSI-кодировки, которая используется по умолчанию операционной системой Windows (для русской версии Windows это кодовая страница 1251 «Cyrillic (Windows)»).

```
static Encoding GetEncoding(int codepage);                       Encoding
```

Возвращает формат кодирования, соответствующий кодовой странице 8-битной кодировки с номером `codepage`. Например, для *Windows-кодировки кириллицы* «Cyrillic (Windows)» параметр `codepage` надо положить равным 1251, а для того чтобы записать или считать данные в *альтернативной кодировке кириллицы для DOS* «Cyrillic (DOS)», надо указать кодовую страницу 866.

Глава 5. Регулярные выражения

Язык регулярных выражений предоставляет программисту существенно более широкие возможности строкового поиска, замены и разбиения, чем соответствующие методы класса `string`. В данной главе рассматриваются классы FCL, связанные с регулярными выражениями, а также приводится краткое описание того варианта языка регулярных выражений, который реализован в .NET Framework.

5.1. Класс Regex

Класс `Regex` и связанные с ним классы описаны в пространстве имен `System.Text.RegularExpressions`.

5.1.1. Конструктор и свойства

```
Regex(string pattern[, RegexOptions options]);                конструктор
```

При создании экземпляра Regex регулярное выражение (передаваемое с помощью параметра pattern) специальным образом обрабатывается, что в дальнейшем ускоряет его использование. Дополнительные *опции поиска* могут указываться в качестве необязательного второго параметра типа RegexOptions (см. далее п. 5.1.4). Заметим, что для работы с регулярными выражениями необязательно создавать экземпляры класса Regex; достаточно использовать *класссовые* методы класса Regex, не требующие создания экземпляра. Однако в этом случае предварительная обработка регулярного выражения выполняться не будет.

Для экземпляра класса Regex доступны два свойства (только для чтения): Options (типа RegexOptions) – набор опций поиска, переданных в конструкторе; RightToLeft (типа bool) – порядок поиска регулярного выражения (одна из опций).

5.1.2. Основные методы

Поскольку в классе Regex реализовано много вариантов каждого из его основных методов, приведем вначале список имен и возвращаемых типов этих методов, не указывая для каждого метода варианты его параметров.

```
bool IsMatch(параметры)                                       Regex
```

Возвращает true или false в зависимости от того, найдено или нет требуемое выражение.

```
Match Match(параметры)                                        Regex
```

Возвращает первое найденное выражение.

```
MatchCollection Matches(параметры)                            Regex
```

Возвращает все найденные выражения.

```
string[] Split(параметры)                                     Regex
```

Разбивает строку на фрагменты; разделители фрагментов определяются регулярным выражением.

```
string Replace(параметры)                                     Regex
```

Заменяет найденные выражения.

Теперь укажем варианты списков параметров для каждого из перечисленных выше методов. Смысл параметров следующий: input – строка, в которой выполняется поиск, pattern – регулярное выражение поиска, replacement – выражение для замены, start – индекс первого символа, начиная с которого выполняется поиск, count – число возвращаемых фрагментов (для метода Split) или число выполняемых замен (для метода Replace).

Параметры классовых методов IsMatch, Match, Matches и Split:

```
(string input, string pattern[, RegexOptions options])        Regex
```

Параметры классового метода Replace:

```
(string input, string pattern, MatchEvaluator evaluator[,          Regex
               RegexOptions options])
(string input, string pattern, string replacement[,
               RegexOptions options])
```

Параметры экземплярных методов `IsMatch`, `Match` и `Matches`:

```
(string input[, int start])                                        Regex
```

Параметры экземплярного метода `Split`:

```
(string input[, int count[, start]])                               Regex
```

Параметры экземплярного метода `Replace`:

```
(string input, MatchEvaluator evaluator[,                          Regex
               int count[, int start]])
(string input, string replacement[, int count[, int start]])
```

5.1.3. Вспомогательные методы

```
static string Escape(string str);                                  Regex
```

Возвращает вариант строки `str`, в котором экранированы все специальные символы, используемые в регулярных выражениях: `\ * + ? | { [ ( ) ^ $ . #` и пробельные символы).

```
static string Unescape(string str);                                Regex
```

Восстанавливает строку, символы которой ранее были экранированы.

5.1.4. Вспомогательные классы

`RegexOptions` – перечисление, определяющее *опции поиска*. Основные члены (в скобках указывается значение этой опции при ее задании непосредственно в регулярном выражении – см. п. 5.2.9):

- `IgnoreCase` – игнорировать регистр при поиске (`i`);
- `Multiline` – *режим многострочного текста* (`m`), при котором символы `^` и `$` соответствуют началу и концу каждой строчки текста, а не всей содержащей его строки типа `string` (как в режиме по умолчанию);
- `ExplicitCapture` – считать группами только те пары круглых скобок, которым явно присвоено имя или номер (`n`);
- `Singleline` – режим (`s`), при котором символ `.` (точка) соответствует любому символу, а не любому символу, кроме `\n` (как в режиме по умолчанию);
- `IgnorePatternWhitespaces` – игнорировать неэкранированные пробельные символы в регулярном выражении (`x`);
- `RightToLeft` – выполнять поиск справа налево (`r`);
- `None` – дополнительных опций нет.

Несколько опций должны объединяться операцией `|`.

`Group` – класс, инкапсулирующий свойства *группы* регулярного выра-

жения. Основные свойства (только для чтения):

- `bool Success` – имеет значение `true`, если группа найдена, и `false` в противном случае;
- `int Index` – индекс начала найденной группы;
- `int Length` – длина найденной группы;
- `string Value` – значение найденной группы (если группа не найдена, то равно пустой строке).

Значение группы также возвращается методом `ToString`.

`Match` – класс (потомок `Group`), инкапсулирующий свойства *найденного вхождения* регулярного выражения. Имеет те же свойства, что и класс `Group`, которые в данном случае относятся не к группе, а ко всему найденному вхождению. Кроме того, имеет свойство только для чтения `Groups` типа `GroupCollection` – коллекцию всех групп, связанных с найденным вхождением. Первый элемент этой коллекции (с индексом 0) соответствует *нулевой* группе, т. е. *всему* найденному вхождению.

В классе `Match` также предусмотрен метод `NextMatch` без параметров, возвращающий значение типа `Match`, который позволяет получить *следующее* найденное вхождение (если очередное вхождение отсутствует, то свойство `Success` возвращенного объекта `Match` будет равно `false`).

`GroupCollection` – класс-коллекция групп, реализующий интерфейсы `ICollection` и `IEnumerable`. Имеет свойство `Count` типа `int` – количество групп (только для чтения) и два индексатора типа `Group` (только для чтения) с числовым или строковым параметром – номером (нумерация ведется от 0) или именем группы.

`MatchCollection` – класс-коллекция найденных вхождений, реализующий интерфейсы `ICollection` и `IEnumerable`. Имеет свойство `Count` типа `int` – количество найденных вхождений (только для чтения) и индексатор типа `Match` (только для чтения) с числовым параметром – номером найденного вхождения (нумерация ведется от 0).

`MatchEvaluator` – делегат с сигнатурой `string (Match match)`, используемый в методе `Replace` и определяющий строку, на которую надо заменить найденное вхождение `match`.

5.2. Язык регулярных выражений

В данном пункте описываются основные элементы языка регулярных выражений и приводятся примеры, иллюстрирующие их использование в методах класса `Regex`.

5.2.1. Некоторые специальные символы

- `\t` – табуляция;

- `\r` – возврат каретки;
- `\f` – новая страница;
- `\n` – новая строка;
- `\x`*`NN`* – ASCII-символ в шестнадцатеричной системе счисления;
- `\u`*`NNNN`* – Unicode-символ в шестнадцатеричной системе счисления.

5.2.2. Множества символов

- `[abcd]` – один из символов в списке (отрицание: `[^abcd]`);
- `[a-d]` – один из символов в диапазоне (отрицание: `[^a-d]`);
- `\d` – десятичная цифра, т. е. `[0-9]` (отрицание: `\D`);
- `\w` – *словообразующий символ*; например, для английского языка соответствует множеству `[a-zA-Z_0-9]` (отрицание: `\W`);
- `\s` – *пробельный символ*, т. е. `[ \t\r\f\n]` (отрицание: `\S`);
- `.` – любой символ, кроме `\n` (в режиме `SingleLine` – любой символ).

Символы, указываемые в квадратных скобках, не должны экранироваться (за исключением символа закрывающей квадратной скобки `]`).

5.2.3. Квантификаторы

Любой *квантификатор* означает, что предшествующий ему элемент регулярного выражения (отдельный символ или элемент множества символов) *может повторяться несколько раз* (т. е. иметь в обрабатываемой строке не одно, а несколько соответствий). Ниже перечислены все возможные квантификаторы:

- `*` – 0 или более соответствий;
- `+` – 1 или более соответствий;
- `?` – 0 или 1 соответствие;
- {*N*} – точно *N* соответствий;
- {*N*,} – не менее *N* соответствий;
- {*N*,*M*} – от *N* до *M* соответствий.

Пример 1. Ищется имя файла cv.doc, возможно, снабженное нумерацией. Обратите внимание на экранирование точки и на использование «буквального» режима в регулярном выражении:

```
Regex.IsMatch("cv12.doc", @"cv\d*\.doc") // вернет true
```

Пример 2. Имя doc-файла, которое начинается с символов cv и оканчивается произвольным текстом:

```
Regex.IsMatch("cvnew.doc", @"cv.*\.doc") // вернет true
```

Все указанные выше квантификаторы являются «жадными». Добавление суффикса `?` превращает квантификатор в «ленивый»:

```
Regex.Match("zz<i>A<i>B</i>C</i>zz", @"<i>.*</i>")
  // вернет <i>A<i>B</i>C</i>
```

```
Regex.Match("zz<i>A<i>B</i>C</i>zz", @"<i>.*?</i>")
  // вернет <i>A<i>B</i>
```

5.2.4. Директивы нулевой длины

Свое название директивы нулевой длины получили потому, что они не соответствуют какому-либо набору символов найденного вхождения. Назначение этих директив – уточнить дальнейшие действия «поискового автомата» в зависимости от его текущего состояния. В частности, поиск считается успешным (и продолжается далее), если текущая позиция строки соответствует текущей директиве нулевой длины, указанной в регулярном выражении. Ниже приводятся основные директивы нулевой длины:

- `^` – начало строки (в режиме `Multiline` – начало любой строчки многострочного текста);
- `$` – конец строки (в режиме `Multiline` – конец любой строчки многострочного текста);
- `\A` – начало строки (в любом режиме);
- `\z` – конец строки (в любом режиме);
- `\Z` – конец строки или строчки многострочного текста;
- `\b` – позиция на границе слова;
- `\B` – позиция не на границе (т. е. внутри) слова;
- `(?=`*expr*`)` – продолжать поиск, если для выражения *expr* есть соответствие справа (*положительный просмотр вперед*);
- `(?!`*expr*`)` – продолжать поиск, если для выражения *expr* нет соответствия справа (*отрицательный просмотр вперед*);
- `(?<=`*expr*`)` – продолжать поиск, если для выражения *expr* есть соответствие слева (*положительный просмотр назад*);
- `(?<!`*expr*`)` – продолжать поиск, если для выражения *expr* нет соответствия слева (*отрицательный просмотр назад*);

Границей слова считается позиция, в которой словообразующий символ `\w` соседствует либо с началом/концом строки, либо с символом, который не является словообразующим (`\W`).

Пример 1:

```
Regex.Match("zz<i>A<i>B</i>C</i>zz", @".*(?=</i>)")
  // вернет zz<i>A<i>B</i>C
```

Пример 2:

```
Regex.Match("zz<i>A<i>B</i>C</i>zz", @"(?<=<i>).*(?=</i>)")
  // вернет A<i>B</i>C
```

Пример 3. Распознавание концов строчек многострочного текста, которые в Windows помечаются двумя символами: `\r\n`:

```
foreach (Match m in Regex.Matches("a.txt\r\nb.doc\r\nc.txt\r\nd.doc",
  @".*\.txt(?=\r?$)", RegexOptions.Multiline))
```

```
    Console.Write(m+" "); // будет напечатано a.txt c.txt
```

Пример 4. Подсчет числа пустых строк в исходном тексте `text` (квантификатор `?` нужен для того, чтобы распознать последнюю пустую строку, после которой может отсутствовать символ `\r`):

```
Regex.Matches(text, @"^\r?$", RegexOptions.Multiline).Count
```

Пример 5. Поиск отдельных слов в строке `s`:

```
foreach (Match m in Regex.Matches("  aa   ss cc ", @"\b\w+\b"))
    Console.Write("("+m+")"); // будет напечатано (aa)(ss)(cc)
```

5.2.5. Группировка и ссылки на группы

- `(`*expr*`)` – включить соответствие для выражения *expr* в нумерованную группу (группы нумеруются от 1 в соответствии с порядком следования их открывающих скобок; группа 0 соответствует всему найденному вхождению);
- `(?<`*name*`>`*expr*`)` или `(?'`*name*`'`*expr*`)` – включить соответствие для выражения *expr* в именованную группу с именем *name*;
- `(?:`*expr*`)` – группирующее выражение, не связываемое с нумерованной или именованной группой;
- `\`*N* – ссылка на ранее найденную группу с номером *N*;
- `\k<`*name*`>` – ссылка на группу с именем *name*.

Пример 1. Чтобы выделить в регулярном выражении для телефонного номера `\d{3}-\d{3}-\d{4}` начальную группу из трех цифр (код региона) и завершающую группу из 7 цифр (собственно телефонный номер), достаточно заключить их в круглые скобки и воспользоваться свойством `Groups`:

```
Match m = Regex.Match("123-456-7890", @"(\d{3})-(\d{3}-\d{4})");
Console.WriteLine(m.Groups[0]); // будет напечатано 123-456-7890
Console.WriteLine(m.Groups[1]); // будет напечатано 123
Console.WriteLine(m.Groups[2]); // будет напечатано 456-7890
```

Найденные группы можно использовать при продолжении поиска, указывая в регулярном выражении *ссылки* на них.

Пример 2. Поиск слова, начинающегося и оканчивающегося на одну и ту же букву:

```
Regex.Match("push pop peek", @"\b(\w)\w*\1\b") // вернет pop
```

5.2.6. Альтернативные варианты

- `a|b` – подходит один из указанных вариантов; при прочих равных условиях предпочтение отдается левому варианту. Можно указывать более двух операндов.

Примеры:

```
Regex.Matches("10", "1|10")   // вернет 1
Regex.Matches("10", "10|1")   // вернет 10
Regex.Matches("10", "0|1|10") // вернет 1 и 0
```

5.2.7. Комментарии

- `(?#comment)` – комментарий;
- `#comment` – комментарий до конца строки (только в режиме `IgnorePatternWhitespaces`).

5.2.8. Некоторые подстановки в выражениях замены

В строковом параметре `replacement` метода `Replace` указывается выражение, которое надо использовать при выполнении замены каждого найденного вхождения. Этот параметр не может содержать «обычные» элементы регулярных выражений, однако для него также определен некоторый язык подстановок, основные элементы которого перечислены ниже:

- `$$` – символ `$`;
- `$0` – все найденное вхождение;
- `$_` – вся исходная строка;
- `$N` – найденная группа с номером `N` (или пустая строка, если группа не найдена);
- `${name}` – найденная группа с именем `name` (или пустая строка, если группа не найдена).

Пример 1. Все найденные числа заключаются в угловые скобки:

```
Regex.Replace("10+2=12", @"\d+", "<$0>")  // вернет <10>+<2>=<12>
```

В случае сложных видов замены удобнее использовать вариант метода `Replace` с параметром-делегатом `MatchEvaluator` (обычно этот параметр представляется в виде лямбда-выражения – см. п. 7.10).

Пример 2. Значения всех найденных чисел удваиваются:

```
Regex.Replace("10+2=12", @"\d+",
  m => (int.Parse(m.Value)*2).ToString()) // вернет 20+4=24
```

5.2.9. Опции поиска

- `(?opt)` – в качестве `opt` указывается буква соответствующей опции (см. описание класса `RegexOptions` в п. 5.1.4). Любая опция, кроме `(?r)`, может указываться в любом месте регулярного выражения и впоследствии может быть отменена директивой `(?-opt)`. Опция `(?r)` должна быть указана в начале регулярного выражения и не может быть отменена. Опции можно объединять следующим образом: `(?i-ms)` – опция `i` включена, опции `m` и `s` отключены.

Примеры:

```
Regex.Match("a", "A", RegexOptions.IgnoreCase) // вернет a
Regex.Match("a", "(?i)A")                      // вернет a
Regex.Match("BaAaAab", "(?i)A+")               // вернет aAaAa
Regex.Match("BaAAaab", "(?i)a(?-i)a")          // вернет Aa
```

Глава 6. Файлы

6.1. Перечисления, связанные с обработкой файлов

Все типы-перечисления, описанные в данном пункте, определены в пространстве имен `System.IO`. После указания имени, входящего в перечислимый тип, в скобках указывается связанное с ним числовое значение.

6.1.1. FileMode

Перечисление `FileMode` определяет *режим открытия* файла:

- `CreateNew` (1) – создать новый файл (если файл уже существует, то возбуждается исключение `IOException`);
- `Create` (2) – создать новый файл; если файл уже существует, то его содержимое очищается, после чего он открывается;
- `Open` (3) – открыть существующий файл (если файл не существует, то возбуждается исключение `FileNotFoundException`);
- `OpenOrCreate` (4) – открыть существующий файл; если файл не существует, то он создается;
- `Truncate` (5) – очистить содержимое существующего файла, после чего открыть его (если файл не существует, то возбуждается исключение `FileNotFoundException`);
- `Append` (6) – открыть существующий файл на запись и переместиться в его конец; если файл не существует, то он создается.

6.1.2. FileAccess

Перечисление `FileAccess` определяет *способ доступа* к файлу:

- `Read` (1) – доступ для чтения;
- `Write` (2) – доступ для записи;
- `ReadWrite` (3) – доступ для чтения и записи.

6.1.3. SeekOrigin

Перечисление `SeekOrigin` определяет позицию, от которой отсчитывается смещение файлового указателя при выполнении метода `Seek`:

- `Begin` (0) – смещение определяется относительно начала файла (допускаются только неотрицательные смещения);
- `Current` (1) – смещение определяется относительно текущей позиции файлового указателя;
- `End` (2) – смещение определяется относительно конца файла (допускаются только неположительные смещения).

6.2. Двоичный файловый поток: класс FileStream

Класс `FileStream` (*файловый поток*) определен в пространстве имен `System.IO`. Он обеспечивает базовые возможности для работы с файлами (открытие, определение и изменение размера файла, позиционирование файлового указателя, чтение/запись байтов и массивов байтов, закрытие). В данном пункте приведены лишь наиболее важные из свойств, конструкторов и методов класса `FileStream`. В частности, не рассматриваются возможности, связанные с файловыми дескрипторами, с совместным доступом к файлу и с асинхронным вводом-выводом.

6.2.1. Свойства

```
string Name { get; }                                                FileStream
```

Возвращает полное имя файла, связанного с файловым потоком `this`.

```
bool CanRead { get; }                                               FileStream
```

Возвращает `true`, если файл открыт на чтение; в противном случае (если файл закрыт или открыт только на запись) возвращает `false`.

```
bool CanWrite { get; }                                              FileStream
```

Возвращает `true`, если файл открыт на запись; в противном случае (если файл закрыт или открыт только на чтение) возвращает `false`.

```
bool CanSeek { get; }                                               FileStream
```

Возвращает `true`, если для файла разрешен *прямой доступ* к элементам; в противном случае (если файл закрыт или обеспечивает только *последовательный доступ*) возвращает `false`.

```
long Length { get; }                                                FileStream
long Position { get; set; }
```

Свойство `Length` возвращает размер открытого файла в байтах, свойство `Position` возвращает и позволяет изменить текущую позицию файлового указателя (следует обратить внимание на то, что данные свойства имеют тип `long`, поэтому в принципе позволяют хранить размер и позицию файлового указателя для файлов размера до 9 223 372 036 854 775 807 байт, или около *9 миллионов терабайт*). Если файл закрыт или не поддерживает прямой доступ (т. е. если свойство `CanSeek` возвращает `false`), попытка вызвать данные свойства приводит к возбуждению исключения. Исключение возбуждается также при попытке присвоить свойству `Position` отрицательное значение, однако допускается присваивать свойству `Position` значение, большее значения свойства `Length`.

Если присвоить свойству `Position` значение, большее `Length`, то автоматического изменения размера файла не произойдет. Для увеличения размера файла необходимо произвести запись новых элементов в установленную позицию (при этом значения байтов, расположенных между старыми и новыми элементами, полагаются равными нулю). Для увеличения

размера файла без записи в него новых элементов можно использовать метод SetLength.

6.2.2. Создание

Стандартным способом создания объекта FileStream, как и объекта любого класса, является вызов конструктора. Ниже описывается наиболее часто используемый вариант конструктора FileStream.

```
FileStream(string name, FileMode mode[, FileAccess access]); конструктор
```

Создает объект типа FileStream, связывает данный объект с файлом, имеющим имя name, и открывает данный файл в режиме, указанном в параметре mode. Если параметр access указан, то он определяет способ доступа к данному файлу; в противном случае устанавливается доступ для чтения и записи (FileAccess.ReadWrite). Совместный доступ к файлу из нескольких файловых потоков возможен только в случае, если для всех этих потоков установлен доступ только для чтения (FileAccess.Read).

В качестве имени name можно указывать как полное имя (включающее имя диска, путь, собственно имя и расширение), так и краткое имя (без имени диска и пути). Если указано краткое имя, то файл ищется в *текущем каталоге*, т. е. в рабочем каталоге приложения (work directory). Если в имени файла указан несуществующий диск или каталог, то возбуждается исключение DirectoryNotFoundException; если полное имя файла имеет слишком большую длину (более 259 символов), то возбуждается исключение PathTooLongException. В языке C# при указании полных имен файлов удобно использовать *буквальные строки* (см. п. 4.1.2).

Помимо вызова конструктора имеются и другие способы создать объект типа FileStream. В частности, объект типа FileStream возвращают перечисленные ниже классовые методы класса File.

```
static FileStream Create(string name);                    File
```

Создает файл с именем name (или очищает файл, если он уже существует) и открывает его на чтение и запись (FileAccess.ReadWrite).

```
static FileStream OpenRead(string name);                  File
```

Открывает *существующий* файл с именем name на чтение (FileAccess.Read); если файл не существует, то возбуждается исключение FileNotFoundException. При открытии файла с помощью этого метода возможен совместный доступ к файлу из нескольких файловых потоков.

```
static FileStream OpenWrite(string name);                 File
```

Открывает файл с именем name на запись (FileAccess.Write); если файл не существует, то он создается.

```
static FileStream Open(string name, FileMode mode[,                File
                FileAccess access]);
```

Работает аналогично описанному выше конструктору класса FileStream, за исключением того, что файл всегда открывается в режиме монопольного доступа, даже при открытии файла только на чтение.

Главным преимуществом методов Create, OpenRead и OpenWrite класса File является более краткая форма их вызова (по сравнению с вызовом конструктора класса FileStream).

Аналогичные методы, возвращающие объект класса FileStream, реализованы в классе FileInfo. Однако в классе FileInfo эти методы являются *экземплярными*, и перед их вызовом надо создать объект типа FileInfo, вызвав его конструктор с единственным параметром name (именем файла). Поскольку имя файла уже определено при вызове конструктора, методы Create, OpenRead, OpenWrite и Open класса FileInfo не содержат параметр name.

6.2.3. Методы

```
long Seek(long offset, SeekOrigin origin);                    FileStream
```

Изменяет текущую позицию файлового указателя для файлового потока this и возвращает новую позицию файлового указателя. Параметр offset определяет смещение указателя, а параметр origin – позицию в файле, относительно которой отсчитывается смещение (см. описание перечисления SeekOrigin в п. 6.1.3). Вместо вызова данного метода достаточно изменить свойство Position:

f.Seek(4, SeekOrigin.Begin) эквивалентно f.Position = 4

f.Seek(4, SeekOrigin.Current) эквивалентно f.Position += 4

f.Seek(-4, SeekOrigin.End) эквивалентно f.Position = f.Length - 4

(в последнем примере предполагается, что файл имеет размер не менее 4 байт).

```
void SetLength(long value);                                   FileStream
```

Изменяет размер файла (в байтах), полагая его равным значению value. Если параметр value является отрицательным, то возбуждается исключение ArgumentOutOfRangeException. Допустимо как уменьшать размер файла (при этом удаляются его последние байты), так и увеличивать его размер (при этом в конец файла добавляются новые байты с нулевыми значениями).

При уменьшении размера файла может измениться значение свойства Position. Это происходит в ситуации, когда прежнее значение свойства Position оказывается б*о*льшим, чем новый размер файла. В подобной ситуации свойство Position полагается равным новому размеру файла Length (иными словами, файловый указатель перемещается на маркер конца файла EOF).

```
int ReadByte();                                                    FileStream
```

Считывает значение байта из текущей позиции файла, перемещает файловый указатель к следующему байту (т. е. увеличивает значение свойства `Position` на 1) и возвращает значение считанного байта, преобразованное к типу `int`.

Если предпринимается попытка прочесть байт за концом файла, то метод возвращает –1, а значение свойства `Position` не изменяется. Таким образом, если файл открыт и доступен для чтения, то выполнение данного метода никогда не приведет к возбуждению исключения.

```
int Read(byte[] array, int start, int count);                      FileStream
```

Считывает `count` или менее байтов из файлового потока `this` (начиная с байта, на который указывает файловый указатель), последовательно записывает их в элементы массива байтов `array`, начиная с элемента с индексом `start`, и возвращает количество фактически считанных байтов. Возвращаемое значение будет равно `count`, если успешно считаны все требуемые байты. В противном случае (если конец файла будет достигнут раньше, чем закончится считывание `count` байтов) возвращаемое значение будет меньше параметра `count`. После выполнения метода файловый указатель перемещается вперед на количество фактически прочитанных байтов (иными словами, значение свойства `Position` увеличивается на величину, равную возвращаемому значению метода `Read`).

При выполнении данного метода исключение возбуждается в следующих случаях:

- файл открыт только на запись;
- в процессе чтения произошла какая-либо ошибка (например, стал недоступен диск с данным файлом);
- параметр `count` отрицателен;
- массив `array` равен `null` или не содержит элементов с индексами в диапазоне от `start` до `start + count - 1`.

Если при выполнении метода возбуждается исключение, то позиция файлового указателя не изменяется.

```
void WriteByte(byte value);                                        FileStream
```

Записывает в текущую позицию файлового потока `this` один байт, равный `value`, и перемещает файловый указатель к следующему байту.

Если текущая позиция находится за последним байтом файла (т. е. значение свойства `Position` равно свойству `Length` или превышает его), то размер файла увеличивается, а значение свойства `Position` становится равным новому значению `Length`. Кроме того, выполняется обнуление всех байтов, находящихся между последним из «старых» байтов и только что записанным байтом.

Если файл открыт только на чтение, то возбуждается исключение; в этом случае позиция файлового указателя не изменяется.

```
void Write(byte[] array, int start, int count);                FileStream
```

Записывает `count` байтов из массива байтов `array`, начиная с элемента с индексом `start`, в файловый поток `this`, начиная с байта, на который указывает файловый указатель. После выполнения метода файловый указатель перемещается вперед на `count` байтов (иными словами, значение свойства `Position` увеличивается на величину `count`).

При выполнении данного метода возможно увеличение размера файла. Если запись выполняется с позиции за концом файла (т. е. значение свойства `Position` превышает значение свойства `Length`), то выполняется обнуление всех байтов, находящихся между последним из «старых» байтов и только что записанными байтами.

При выполнении данного метода исключение возбуждается в следующих случаях:

- файл открыт только на чтение;
- в процессе записи произошла какая-либо ошибка (например, на диске недостаточно места для записи новых байтов);
- параметр `count` отрицателен;
- массив `array` равен `null` или не содержит элементов с индексами в диапазоне от `start` до `start + count - 1`.

Если при выполнении метода возбуждается исключение, то позиция файлового указателя не изменяется.

```
void Flush();                                                  FileStream
```

Записывает в файл данные, содержащиеся в *файловом буфере* (т. е. в области оперативной памяти, в которой накапливаются данные, полученные из программы, перед их записью в файл). После этого очищает файловый буфер. Данный метод автоматически вызывается при закрытии файла методом `Close`, поэтому в его явном вызове, как правило, нет необходимости.

```
void Close();                                                  FileStream
```

Закрывает файл, связанный с файловым потоком `this`, и освобождает неуправляемые ресурсы, выделенные для работы с данным файлом. Перед закрытием файла для него выполняется метод `Flush`.

Когда объект, связанный с файловым потоком, разрушается (т. е. удаляется из памяти), для него автоматически вызывается метод `Close`. Несмотря на эту «страховочную» возможность, *следует всегда закрывать файловый поток сразу после завершения работы с ним, явным образом вызывая метод* `Close` (или вызывая одноименный метод потока-оболочки, связанного с данным потоком, – см. описание метода `Close` для классов `BinaryReader` и `BinaryWriter` в п. 6.3.1).

После вызова метода Close все действия, связанные с доступом к элементам файла, их чтением или записью, будут приводить к возбуждению исключения. Однако повторный вызов метода Close игнорируется, не возбуждая исключения. После закрытия файла можно обращаться к свойствам CanRead, CanWrite, CanSeek (все они будут возвращать false), а также к свойству Name, которое по-прежнему будет возвращать полное имя файла.

6.3. Двоичные потоки-оболочки: классы BinaryReader и BinaryWriter

Рассмотренный в предыдущем пункте класс FileStream позволяет осуществлять ввод-вывод файловых данных только в виде *наборов байтов*. Для возможности чтения или записи более сложных структур данных необходимо использовать «надстройки» над стандартным файловым потоком: класс BinaryReader (*двоичный поток-оболочка для чтения*) или класс BinaryWriter (*двоичный поток-оболочка для записи*) соответственно. Оба эти класса определены в пространстве имен System.IO. Они обеспечивают чтение из файла и запись в файл всех элементарных типов языка C#, кроме класса object.

6.3.1. Конструкторы, общие свойства и методы двоичных потоков

Хотя классы BinaryReader и BinaryWriter предназначены для выполнения принципиально различных действий, у них имеется ряд одинаковых свойств и методов, которые удобно описать совместно. Кроме того, похожими являются и конструкторы данных классов.

```
BinaryReader(Stream stream[, Encoding encoding]);          конструктор
BinaryWriter(Stream stream[, Encoding encoding]);
```

Каждый из конструкторов создает соответствующий двоичный поток-оболочку, которая связывается с базовым потоком stream. Тип Stream является общим предком всех классов-потоков; при работе с файлами в качестве параметра stream указывается объект типа FileStream. Базовый поток stream необязательно предварительно сохранять в отдельной переменной; допустимо создавать его «на лету», указывая в качестве первого параметра вызов конструктора класса FileStream или подходящий метод классов File или FileInfo. В дальнейшем доступ к базовому потоку можно получить, используя свойство BaseStream потоков-оболочек (см. ниже его описание).

Параметр encoding определяет для класса BinaryReader *формат декодирования* символьных данных при их чтении из файла, а для класса BinaryWriter – *формат кодирования* символьных данных при их записи в файл. Если данный параметр не указан, то используется формат UTF-8.

Если поток stream не поддерживает требуемую операцию ввода-вывода (чтение в случае BinaryReader или запись в случае BinaryWriter) или

если поток уже закрыт, то возбуждается исключение ArgumentException. Если какой-либо из параметров равен null, то возбуждается исключение ArgumentNullException.

```
Stream BaseStream { get; }                    BinaryReader, BinaryWriter
```

Свойство только для чтения, возвращающее базовый поток для потока-оболочки this. Приводить данное свойство (типа Stream) к типу FileStream следует только в случае, если требуется обратиться к свойству Name или методу SetLength класса FileStream, так как все прочие свойства и методы уже определены в классе Stream, являющемся предком всех классов-потоков. Доступ к базовому потоку с помощью свойства BaseStream возможен только при *открытом* потоке-оболочке (до вызова его метода Close).

Следует заметить, что действия по *прямому доступу* к файловым элементам для потока-оболочки BinaryReader возможны только с помощью свойств и методов потока BaseStream, поскольку поток-оболочка BinaryReader аналогичных средств для прямого доступа не имеет. Для потока-оболочки BinaryWriter реализован метод Seek, однако доступ к свойствам базового потока (в частности, Position и Length) возможен только через свойство BaseStream.

```
void Close();                                 BinaryReader, BinaryWriter
```

Закрывает базовый поток BaseStream, связанный с потоком-оболочкой this, и освобождает неуправляемые ресурсы, выделенные для работы с этими потоками. Перед закрытием потока-оболочки BinaryWriter для него выполняется метод Flush. Повторное выполнение метода Close игнорируется, не возбуждая исключения.

Необходимо *обязательно* вызывать метод Close потока-оболочки, так как данный метод (в отличие от одноименного метода класса FileStream) *не вызывается автоматически при разрушении объекта типа* BinaryReader *или* BinaryWriter.

Поскольку метод Close потока-оболочки автоматически закрывает базовый поток, явно вызывать метод Close базового потока после закрытия потока-оболочки *не требуется*.

Если к одному и тому же файлу одновременно подключены два потока-оболочки (BinaryReader и BinaryWriter), то нельзя закрывать один из них до завершения работы с другим, так как закрытие одного из потоков-оболочек автоматически приведет к закрытию базового потока, связанного с файлом, а это сделает невозможным доступ к файлу для оставшегося потока-оболочки.

6.3.2. Чтение данных с помощью объекта BinaryReader

В любом из указанных ниже методов считывание данных начинается с текущей позиции файла (т. е. с позиции файлового указателя). После вы-

полнения любой операции по считыванию данных файловый указатель перемещается вперед на количество прочитанных байтов.

При работе с символьными данными (типа `char` и `string`) следует учитывать, что в файле они хранятся в закодированном виде, поэтому для их правильного считывания необходимо при создании потока-оболочки `BinaryReader` указать тот же *формат кодирования*, который использовался при записи этих символьных данных в файл.

Для чтения каждого элементарного типа данных (кроме класса `object`) в классе `BinaryReader` предусмотрен особый метод. Ниже приводятся только те методы, которые связаны с основными элементарными типами.

```
bool ReadBoolean();                                          BinaryReader
byte ReadByte();
int ReadInt32();
long ReadInt64();
double ReadDouble();
char ReadChar();
string ReadString();
```

Каждый из методов данной группы считывает из потока один элемент требуемого типа и возвращает его значение (за исключением метода `ReadBoolean`, который читает из потока один байт и возвращает `false`, если прочитанный байт равен 0, и `true` в противном случае).

При попытке прочесть данные за концом файла возбуждается исключение `EndOfStreamException`.

Метод `ReadString` вначале читает из потока информацию о длине строки (*в байтах*), а затем считывает указанное количество байтов и преобразует прочитанные байты в символы, учитывая использованный в файле формат кодирования. Следует заметить, что информация о длине строки может занимать от 1 до 5 байт, в зависимости от размеров строки. Например, если символы строки занимают не более 127 байт (наиболее распространенная ситуация), то длина строки кодируется в *одном* байте, причем значение этого байта равно количеству байтов текста. Случаи, когда символы строки в двоичном файле занимают более 127 байтов, встречаются достаточно редко и поэтому здесь подробно не описываются; отметим лишь, что если строка занимает от 128 до 16383 байтов, то ее длина кодируется *двумя* байтами. Подчеркнем, что длина строки определяется *в байтах*, а не в символах; в кодировках UTF-8 и UTF-16 (Unicode) это различие является существенным, так как, например, русские буквы в этих кодировках занимают по 2 байта.

```
byte[] ReadBytes(int count);                                 BinaryReader
char[] ReadChars(int count);
```

Метод `ReadBytes` считывает из потока `count` или менее байтов и возвращает прочитанные байты в виде массива. Размер массива будет равен

count, если успешно считаны все требуемые байты. В противном случае (если конец файла будет достигнут раньше, чем закончится считывание count байтов) размер массива будет меньше параметра count; исключение в этой ситуации не возбуждается. Параметр count должен быть неотрицательным.

Метод ReadChars выполняется аналогично, за исключением того, что считываются не байты, а символы.

```
int Read();                                                    BinaryReader
```

Считывает из потока один символ и возвращает его значение, преобразованное к типу int (т. е. возвращается *код* символа в кодировке Unicode). Если предпринимается попытка прочесть символ за концом файла, то метод возвращает –1. Таким образом, данный метод работает аналогично методу ReadByte класса FileStream, за исключением того, что считывается *символ*, а не байт данных (как в методе ReadByte).

```
int Read(byte[] bytes, int start, int count);                  BinaryReader
int Read(char[] chars, int start, int count);
```

Вариант метода Read с параметром bytes (массивом байтов) работает в точности так же, как метод Read класса FileStream. Вариант метода Read с параметром chars (массивом символов) работает аналогично, за исключением того, что из файла производится считывание не байтов, а символов.

Напомним, что во всех методах, считывающих символы, *учитывается формат кодирования*, указанный при создании объекта-оболочки BinaryReader.

6.3.3. Запись данных с помощью объекта BinaryWriter

Для записи данных в классе BinaryWriter предусмотрен единственный метод, который перегружен для различных типов записываемых данных, в том числе для всех элементарных типов (кроме класса object). Ниже приводятся только те варианты метода Write, которые связаны с основными элементарными типами.

В любом из указанных методов запись данных начинается с текущей позиции файла (т. е. с позиции файлового указателя). После выполнения любой операции по записи данных файловый указатель перемещается вперед на количество записанных байтов; при этом возможно увеличение размера файла.

Текст *числовой_тип*, используемый при описании параметра value, обозначает любой из числовых типов.

Если параметры методов равны null, то возбуждается исключение ArgumentNullException.

При работе с символьными данными (типа char и string) следует учитывать, что при их записи в файл будет выполняться *кодирование*, формат которого определяется при создании потока-оболочки BinaryWriter.

```
void Write(bool value);                                          BinaryWriter
void Write(числовой_тип value);
void Write(char value);
void Write(string value);
```

Каждый из методов данной группы записывает в поток значение параметра value соответствующего типа. Исключение составляет параметр value типа bool, вместо которого в файл записывается один байт со значением 0 (если параметр равен false) или 1 (если параметр равен true).

При записи строки в файл вначале записывается информация о длине строки (указывается длина *уже закодированной* строки *в байтах*), а затем – сами символы строки (символы кодируются с учетом формата кодирования, определенного для потока BinaryWriter). По поводу способа хранения в файле информации о длине строки см. описание метода ReadString.

```
void Write(byte[] array[, int start, int count]);          BinaryWriter
void Write(char[] array[, int start, int count]);
```

Каждый из данных методов записывает в поток элементы массива array (байты или символы). Если параметры start и count указаны, то записывается count элементов массива array, начиная с элемента с индексом start; в противном случае в поток записываются все элементы массива. Таким образом, первый из данных методов работает аналогично методу Write класса FileStream.

6.3.4. Дополнительные поля и методы двоичных потоков

Кроме рассмотренных выше свойств и методов класс BinaryReader содержит полезный метод PeekChar.

```
int PeekChar();                                                  BinaryReader
```

Возвращает код текущего символа из базового потока (преобразованный к типу int) или –1, если, начиная с текущей позиции, поток не содержит символов. Изменения позиции файлового указателя при выполнении метода PeekChar *не происходит*.

Данный метод удобно использовать для проверки того, достигнут ли конец файла при считывании данных с помощью потока-оболочки BinaryReader:

```
while (r.PeekChar() != -1)
{
  <чтение и обработка очередного элемента из потока-оболочки r>
}
```

Следует, однако, иметь в виду, что при использовании стандартного формата кодирования UTF-8 данный метод может возвращать –1, не дохо-

дя до фактического конца файла. Например, если последними элементами двоичного файла являются отрицательные целые числа (типа int) в диапазоне от –1 до –128, то уже при анализе первого из этих чисел метод PeekChar вернет –1 (это «странное» поведение метода связано с особенностями формата UTF-8). Чтобы избежать подобных проблем и обеспечить безопасную обработку *числовых* двоичных файлов с использованием метода PeekChar, необходимо при создании потока-оболочки BinaryReader указывать формат кодирования Encoding.Default, соответствующий ANSI-кодировке, используемой системой Windows по умолчанию.

В заключение опишем дополнительные поля и методы класса BinaryWriter.

```
static readonly BinaryWriter Null;                    BinaryWriter
```

Данное классовое поле позволяет использовать в программе «пустой» двоичный поток-оболочку, который записывает данные «в никуда». Эта возможность оказывается полезной при отладке, а также при реализации «молчаливых» режимов работы программы.

```
void Flush();                                         BinaryWriter
```

Записывает в файл данные, содержащиеся в *буфере* потока-оболочки this. После этого очищает данный буфер.

```
long Seek(int offset, SeekOrigin origin);             BinaryWriter
```

Вызывает одноименный метод базового потока BaseStream, связанного с потоком-оболочкой this.

Подчеркнем, что два последних метода реализованы только для класса BinaryWriter (у класса BinaryReader они отсутствуют).

6.4. Текстовые файловые потоки: классы StreamReader и StreamWriter

Для работы с *текстовыми файлами* предусмотрены два основных класса: *текстовый поток-оболочка для чтения* StreamReader и *текстовый поток-оболочка для записи* StreamWriter. Эти классы определены в пространстве имен System.IO.

6.4.1. Создание текстовых потоков

В отличие от двоичных потоков-оболочек (BinaryReader и BinaryWriter) при создании текстовых потоков не требуется явно создавать «базовый» файловый поток типа FileStream; достаточно лишь указать имя требуемого текстового файла. Заметим, что базовый файловый поток все же создается, и к нему можно получить доступ с помощью свойства BaseStream, однако при работе с текстовыми потоками данное свойство, как правило, не используется. Имеется также возможность связать текстовый поток с уже существующим файловым потоком (типа FileStream); для этого предусмот-

рены соответствующие конструкторы. В настоящем пособии эти конструкторы (как и некоторые другие конструкторы со специальными возможностями) не рассматриваются.

```
StreamReader(string name[, Encoding encoding]);                конструктор
StreamWriter(string name[, bool append[, Encoding encoding]]);
```

Создает объект-поток указанного типа и связывает его с файлом с именем `name` (дополнительные сведения о параметре `name` приводятся в описании конструктора `FileStream` – см. п. 6.2.2).

В случае потока для чтения `StreamReader` указанный файл должен существовать (иначе возбуждается исключение `FileNotFoundException`). Файл открывается на чтение, и файловый указатель устанавливается на начало файла.

В случае потока для записи `StreamWriter` файл может отсутствовать; в этом случае он автоматически создается. Файл открывается для записи. Если параметр `append` не указан или равен `false`, то содержимое существующего файла очищается; если указан параметр `append`, равный `true`, то файл открывается *для дополнения*, т. е. его содержимое сохраняется, а файловый указатель устанавливается на маркер конца файла. Если файл является пустым (например, только что создан), то значение параметра `append` при его открытии может быть любым.

Если указан параметр `encoding`, то он определяет *формат кодирования* файловых данных; при отсутствии этого параметра используется формат UTF-8. Для установки формата кодирования, который соответствует ANSI-кодировке, используемой системой Windows по умолчанию, в качестве параметра `encoding` следует указать `Encoding.Default`.

Совместный доступ к файлу из нескольких текстовых потоков возможен только в случае, если все эти потоки являются потоками типа `StreamReader`.

Текстовые потоки можно также создать с помощью перечисленных ниже классовых методов класса `File`.

```
static StreamReader OpenText(string name);                          File
static StreamWriter CreateText(string name);
static StreamWriter AppendText(string name);
```

Действие метода `OpenText` аналогично действию конструктора класса `StreamReader` с единственным параметром `name`; действие методов `CreateText` и `AppendText` аналогично действию конструктора класса `StreamWriter` с первым параметром `name` и вторым параметром, равным `false` и `true` соответственно. К сожалению, в данных методах не предусмотрен параметр, определяющий формат кодирования; при создании потока для него всегда устанавливается формат UTF-8.

Аналогичные методы реализованы в классе FileInfo. Однако в классе FileInfo эти методы являются *экземплярными* и не имеют параметров (перед их вызовом надо создать объект типа FileInfo, вызвав его конструктор с единственным параметром name – именем файла).

6.4.2. Закрытие текстовых потоков. Пустые текстовые потоки

```
void Close();                                    StreamReader, StreamWriter
```

Закрывает текстовый поток this (при этом закрывается и базовый поток BaseStream, связанный с потоком this) и освобождает неуправляемые ресурсы, выделенные для работы с данным потоком. Повторное выполнение метода Close игнорируется, не возбуждая исключения.

Перед закрытием потока StreamWriter выполняется его метод Flush, записывающий в файл текст, оставшийся в файловом буфере. Заметим, что класс StreamReader не имеет метода Flush.

Для текстовых потоков, как и для двоичных, следует *всегда* вызывать метод Close после завершения работы с ними.

```
static readonly StreamReader Null;                          StreamReader
static readonly StreamWriter Null;                          StreamWriter
```

Данные классовые поля позволяют использовать в программе «пустые» текстовые потоки для чтения и записи (пустой поток StreamReader.Null обрабатывается как файл с пустым содержимым; текст, записываемый в пустой поток StreamWriter.Null, нигде не сохраняется). Эта возможность оказывается полезной при отладке, а также при реализации «молчаливых» режимов работы программы.

6.4.3. Чтение данных из текстового потока

В любом из описанных ниже методов чтения считывание данных начинается с текущей позиции файла (т. е. с позиции файлового указателя). После выполнения любой операции по считыванию данных файловый указатель перемещается вперед на количество прочитанных символов.

Поскольку символы в текстовых файлах хранятся в закодированном виде, для их правильного считывания необходимо при создании потока StreamReader указать тот же формат кодирования, который использовался при записи текста в файл.

```
int Read();                                                 StreamReader
int Read(char[] chars, int start, int count);
```

Данные методы работают аналогично методам Read класса BinaryReader с теми же параметрами (см. п. 6.3.2).

```
string ReadLine();                                          StreamReader
```

Считывает и возвращает очередную строку из текстового потока this. Признаком конца строки считается конец файла или наличие одного из

двух вариантов *маркеров конца строки*: символ с кодом 10 ('\n') или пара символов с кодами 13 и 10 ('\r', '\n'); маркер конца строки в возвращаемую строку не включается. Если данный метод вызывается после достижения конца файла, то он возвращает значение null.

```
string ReadToEnd();                                          StreamReader
```

Считывает все оставшиеся символы из текстового потока this (начиная с текущего символа) и возвращает их в виде одной строки, содержащей как «обычные символы», так и маркеры конца строк (маркер конца файла в возвращаемую строку не включается). Если данный метод вызывается после достижения конца файла, то он возвращает пустую строку "".

При организации считывания данных из текстовых файлов могут оказаться полезными следующие метод и свойство класса StreamReader.

```
int Peek();                                                  StreamReader
```

Данный метод работает так же, как метод PeekChar класса BinaryReader.

```
bool EndOfStream { get; }                                    StreamReader
```

Возвращает true, если достигнут конец файла, и false в противном случае.

6.4.4. Запись данных в текстовый поток

Для записи данных в классе StreamWriter предусмотрены методы Write и WriteLine, которые перегружены для различных типов записываемых данных, в том числе для всех элементарных типов, включая класс object. Ниже приводятся только те варианты метода Write, которые связаны с основными элементарными типами.

В любом из приведенных методов записи данные добавляются в конец файла. Текст *числовой_тип*, указанный при описании параметра value, обозначает любой из числовых типов.

При работе с символьными данными (типа char и string) следует учитывать, что при их записи в файл будет выполняться кодирование, формат которого определяется при создании текстового потока StreamWriter.

```
void Write(object value);                                    StreamWriter
void Write(bool value);
void Write(числовой_тип value);
void Write(char value);
void Write(string value);
```

Каждый из методов данной группы записывает в поток текстовое представление параметра value (для получения текстового представления вызывается метод ToString указанного параметра; в частности, для символьных и строковых параметров в файл записываются сами эти параметры). Если параметр равен null, то для него метод ToString не вызывается и в файл ничего не записывается.

```
void Write(char[] array[, int start, int count]);                StreamWriter
```

Оба варианта данного метода работают аналогично методам Write класса BinaryWriter с теми же параметрами, за исключением того, что в случае единственного параметра array, равного null, исключение не возбуждается и в файл ничего не записывается.

```
void Write(string fmt, params object[] args);                    StreamWriter
```

Данный метод обеспечивает *форматный вывод данных*, использующий форматную строку fmt. Фактически он вызывает метод Format класса string с указанными параметрами (см. п. 4.3.3), после чего записывает строку, возвращенную этим методом, в текстовый поток.

Для любого из перечисленных выше методов Write имеется «парный» к нему метод WriteLine с тем же набором параметров. Метод WriteLine записывает в текстовый поток те же данные, что и соответствующий ему метод Write, после чего дописывает в поток *маркер конца строки*. Имеется также вариант метода WriteLine без параметров, который записывает в файл только маркер конца строки:

```
void WriteLine();                                                StreamWriter
```

Используемый во всех вариантах метода WriteLine маркер конца строки берется из свойства NewLine класса StreamWriter:

```
string NewLine { get; set; }                                     StreamWriter
```

По умолчанию это свойство возвращает строку из двух символов с кодами 13 и 10 ("\r\n"). Значение свойства NewLine можно изменять (например, на строку "\n" или "\r\n\r\n"), но обычно необходимости в этом не возникает.

6.4.5. Стандартный текстовый поток для ввода-вывода: класс Console

В любых консольных приложениях платформы .NET доступен особый текстовый поток для ввода-вывода: *консольное окно*. Для управления этим окном предназначен класс Console, определенный в пространстве имен System.

В оконных приложениях .NET консольное окно по умолчанию не создается, однако попытки ввода-вывода данных с использованием класса Console не приводят к возбуждению исключений, поскольку в этой ситуации класс Console связывается с «пустыми» текстовыми потоками для ввода и вывода. Для возможности использования консольного окна в оконных приложениях необходимо указать в качестве типа приложения вариант «Console Application».

В .NET Framework версии 2.0 класс Console был дополнен большим количеством свойств и методов, предназначенных для гибкого управления внешним видом консольного окна и обеспечения дополнительных воз-

можностей при консольном вводе данных. В настоящей книге эти свойства и методы не рассматриваются.

Объект типа `Console` создавать не требуется; все действия с консольным окном выполняются с помощью классовых свойств и методов класса `Console`.

```
static TextReader In { get; }                                    Console
static TextWriter Out { get; }
static TextWriter Error { get; }
```

Каждое из данных свойств обеспечивает доступ к соответствующему текстовому потоку, связанному с консольным окном: `In` – стандартный поток для ввода, `Out` – стандартный поток для вывода, `Error` – поток для вывода сообщений об ошибках. Данные свойства предназначены только для чтения; в качестве их типа указаны классы `TextReader` и `TextWriter` – базовые предки любых текстовых потоков ввода-вывода (в том числе файловых). Заметим, что классы `TextReader` и `TextWriter` имеют те же методы для ввода и вывода, что и рассмотренные ранее классы `StreamReader` и `StreamWriter`.

В стандартных консольных потоках используется формат кодирования, принятый по умолчанию для консольных окон в текущей версии Windows; в частности, в русской версии Windows используется кодовая страница 866 «Cyrillic (DOS)» ANSI-кодировки.

```
static void SetIn(TextReader newIn);                             Console
static void SetOut(TextWriter newOut);
static void SetError(TextWriter newError);
```

Данные методы предназначены для перенаправления стандартных консольных потоков ввода-вывода; в качестве их новых значений можно указать, например, текстовые файлы или пустые потоки.

```
static int Read();                                               Console
static string ReadLine();
```

Данные методы предназначены для ввода символов и строк из стандартного потока ввода `In`. Они работают аналогично соответствующим методам класса `StreamReader`, однако следует учитывать, что текст, набранный в консольном окне, передается программе (и, следовательно, обрабатывается методами ввода) только после нажатия клавиши [Enter]. Если же поток ввода пуст, то выполнение программы приостанавливается до тех пор, пока с клавиатуры не будут введены данные и не будет нажата клавиша [Enter]. Следует также иметь в виду, что нажатие клавиши [Enter] записывает во входной поток два символа с кодами 13 и 10, которые могут считываться методом `Read` как обычные символы.

```
static void Write(object value);                                    Console
static void Write(bool value);
static void Write(числовой_тип value);
static void Write(char value);
static void Write(string value);
static void Write(char[] array[, int start, int count]);
static void Write(string fmt, params object[] args);
```

Эти методы предназначены для вывода данных различных типов в стандартный поток вывода `Out`. Кроме того, у класса `Console` имеются «парные» к `Write` методы `WriteLine` с тем же набором параметров и метод `WriteLine` без параметров:

```
static void WriteLine();                                            Console
```

Все эти методы работают аналогично одноименным методам класса `StreamWriter`, поскольку вызов любого варианта классового метода `Write` или `WriteLine` класса `Console` приводит к вызову соответствующего варианта одноименного экземплярного метода для объекта `Console.Out`.

Для вывода данных в поток сообщений об ошибках `Error` специальных методов не предусмотрено, однако для этих целей можно использовать соответствующие методы потока `Error`, например: `Console.Error.WriteLine()`.

6.5. Вспомогательные классы для работы с файлами, каталогами и дисками

Все типы, описанные в данном пункте, определены в пространстве имен `System.IO`.

6.5.1. Перечисление FileAttributes

Перечисление `FileAttributes` содержит имена различных *файловых атрибутов*. В переменных типа `FileAttributes` эти имена могут комбинироваться с помощью побитовой операции «ИЛИ» (операция | в C#). Ниже приведены имена наиболее распространенных атрибутов; в скобках после них указываются связанные с ними числовые значения:

- `ReadOnly` (1) – файл только для чтения;
- `Hidden` (2) – скрытый файл;
- `System` (4) – системный файл;
- `Directory` (16) – каталог.

6.5.2. Перечисление DriveType

Перечисление `DriveType` определяет *типы логических дисков*. Приведем имена, входящие в это перечисление, указывая в скобках связанные с ними числовые значения:

- Unknown (0) – неизвестный тип логического диска;
- NoRootDirectory (1) – логический диск, не имеющий корневого каталога (как правило, это означает, что указанный диск отсутствует);
- Removable (2) – устройство для сменных носителей (например, дисковод или USB-порт);
- Fixed (3) – жесткий локальный диск;
- Network (4) – сетевой диск;
- CDRom (5) – устройство для чтения компакт-дисков;
- Ram (6) – виртуальный диск, созданный в оперативной памяти.

6.5.3. Класс Path

Класс Path предназначен для манипулирования *именами файлов*. Он не требует создания экземпляров; все его поля и методы являются классовыми.

Поля данного класса определяют различные типы символов, используемых (или запрещенных для использования) в именах файлов в различных операционных системах. В данном пособии поля класса Path не рассматриваются.

Основная часть методов предназначена для выделения требуемого элемента из имени файла или каталога. Перечислим эти методы, описав для каждого возвращаемое значение (все методы имеют единственный параметр name типа string и возвращают объект типа string):

- GetPathRoot – имя корневого каталога (вида "C:\", "C:" или "\"). *Особые ситуации*: если name равно null, то возвращается null; если name не содержит имени корневого каталога в какой-либо из указанных выше форм, то возвращается пустая строка;
- GetDirectoryName – путь к файлу, включающий имя диска, но не содержащий завершающий символ «\» (если name завершается символом «\», то возвращается строка name без завершающего символа «\»). *Особые ситуации*: если name представляет собой имя корневого каталога вида "C:\", "C:" или "\", то возвращается null; если имя файла name не содержит имени диска и пути, то возвращается пустая строка ""; если файл находится в корневом каталоге (например, "C:\data" или "\data"), то возвращаемая строка *содержит* завершающий символ «\» (для приведенных примеров будет возвращено "C:\" или "\" соответственно); если параметр name равен null, является пустой строкой или состоит только из пробелов, то возбуждается исключение;
- GetFileName – имя файла вместе с расширением. *Особые ситуации*: если name является пустой строкой или оканчивается символом-

разделителем для диска или каталога, то возвращается пустая строка; если name равно null, то возвращается null;

- GetFileNameWithoutExtension – имя файла *без расширения*, т. е. строка, возвращенная методом GetFileName, у которой дополнительно удален последний из имеющихся символов «.» (точка) и все следующие за ним символы;
- GetExtension – расширение файла, включая предшествующую точку. *Особые ситуации*: если name не содержит расширения или оканчивается символом «.», то возвращается пустая строка; если name равно null, то возвращается null;
- GetFullPath – полное имя файла (если name содержит относительное имя, то к нему добавляется имя текущего каталога вместе с именем диска; если name начинается с символа «\», то к нему добавляется имя текущего диска). *Особые ситуации*: если name равно null, является пустой строкой или состоит только из пробелов, то возбуждается исключение.

```
static bool HasExtension(string name);                                    Path
```

Возвращает true, если имя файла name содержит непустое расширение, и false в противном случае.

```
static bool IsPathRooted(string name);                                    Path
```

Возвращает true, если имя файла name содержит корневой каталог (т. е. является абсолютным), и false, если имя является относительным.

```
static string ChangeExtension(string name, string ext);                  Path
```

Возвращает имя name, расширение которого заменено на расширение ext. Параметр ext может либо содержать, либо не содержать начальный символ «.». *Особые ситуации*: если параметр name равен null или является пустой строкой, то он возвращается без изменений; если параметр ext равен null, то возвращается строка name *без расширения* (и без предшествующей расширению точки); если параметр ext равен пустой строке или строке " .", то возвращается строка name *с пустым расширением* – точкой.

Все вышеперечисленные методы класса Path возбуждают исключение, если имя name содержит недопустимые символы (например, «<», «>», «|», «"»); при этом символы «*» и «?», используемые в *масках файлов*, считаются допустимыми.

В заключение опишем методы класса Path, связанные с созданием *временных файлов*.

```
static string GetTempPath();                                              Path
```

Возвращает имя системного каталога, предназначенного для хранения временных файлов.

```
static string GetTempFileName();                                    Path
```

Создает пустой файл с уникальным именем и расширением .tmp в системном каталоге, предназначенном для хранения временных файлов, и возвращает полное имя созданного файла.

```
static string GetRandomFileName();                                  Path
```

Возвращает случайную строку, которую можно использовать в качестве имени файла или каталога. Строка состоит из цифр и строчных латинских букв и включает собственно имя из 8 символов и расширение из 3 символов.

6.5.4. Классы File и FileInfo

Классы `File` и `FileInfo` предназначены в основном для обработки файлов как элементов файловой системы, без доступа к их содержимому. Класс `File` содержит только классовые методы; при этом первым параметром любого метода является имя `name` обрабатываемого файла. Класс `FileInfo` содержит свойства и экземплярные методы, для доступа к которым необходимо создать объект данного класса, указав в его конструкторе имя обрабатываемого файла. В конструкторе класса `FileInfo` нельзя указывать значение `null`, пустую строку или строку, состоящую только из пробелов, а также строку, содержащую недопустимые символы (например, «<», «>», «|», «"», «*», «?»).

Если для некоторого файла требуется выполнить *единственное* действие, то удобнее воспользоваться для этого классом `File`; если же для файла надо выполнить *несколько* действий, то целесообразно создать для данного файла объект типа `FileInfo`, с помощью которого и выполнить нужные действия.

При описании файловых потоков уже рассматривались методы классов `File` и `FileInfo`, позволяющие открыть файл с нужным именем: `Create`, `OpenRead`, `OpenWrite`, `Open` (п. 6.2.2) и `OpenText`, `CreateText`, `AppendText` (п. 6.4.1). В данном пункте мы не будем возвращаться к этим методам.

Поскольку все методы класса `File` имеют свои аналоги (свойства или методы) в классе `FileInfo`, ниже приводятся описания только методов класса `File`, после которых кратко описываются их аналоги для класса `FileInfo`. Для всех описанных ниже методов в качестве параметра `name` можно указывать как абсолютные, так и относительные имена файлов; в последнем случае файлы ищутся в текущем каталоге (т. е. в рабочем каталоге приложения).

```
static void Copy(string name, string newName[, bool overwrite]);   File
```

Создает копию файла `name` с именем `newName`. Файл `name` и путь, указанный в имени `newName`, должны существовать; имя `newName` не должно быть именем существующего каталога; если параметр `overwrite` не указан или

равен false, то имя newName не может быть именем существующего файла (при нарушении любого из этих условий возбуждается исключение). Если параметр overwrite равен true и файл newName существует, то его содержимое заменяется на содержимое файла name (если же файл newName закрыт на запись, то возбуждается исключение).

В классе FileInfo данному методу соответствует метод CopyTo с теми же параметрами, за исключением параметра name. В отличие от метода Copy, не возвращающего значение, метод CopyTo возвращает объект типа FileInfo, соответствующий созданной копии (с полным файловым именем).

```
static void Move(string name, string newName);                    File
```

Переименовывает файл name, заменяя его имя на newName. В качестве имени name нельзя указывать имя каталога; файл name и путь, указанный в имени newName, должны существовать; имя newName не должно быть именем существующего каталога или файла (при нарушении любого из этих условий возбуждается исключение). В имени newName можно указывать каталог, находящийся на другом диске. Новое имя файла может совпадать со старым; в этом случае метод не выполняет никаких действий.

В классе FileInfo данному методу соответствует метод MoveTo с параметром newName; в результате успешного выполнения метода MoveTo вызвавший его объект типа FileInfo связывается с именем newName.

```
static void Delete(string name);                                  File
```

Удаляет файл с именем name. Если файл не существует, то метод не выполняет никаких действий. Если в имени файла указан несуществующий диск и/или каталог или если файл существует, но удален быть не может (например, если он в данный момент используется другим приложением), а также если в качестве name указано имя существующего каталога, то возбуждается исключение.

В классе FileInfo данному методу соответствует метод Delete без параметров.

```
static bool Exists(string name);                                  File
```

Возвращает true, если файл с именем name существует, и false в противном случае. В частности, метод возвращает false, если параметр name равен null, является пустой строкой или является именем *каталога* (пусть даже и существующего). Данный метод никогда не возбуждает исключения.

В классе FileInfo данному методу соответствует логическое свойство Exists, доступное только для чтения.

```
static FileAttributes GetAttributes(string name);                 File
static void SetAttributes(string name, FileAttributes newAttr);
```

Метод GetAttributes позволяет определить, а метод SetAttributes – изменить атрибуты файла или каталога с именем name. Если name не является

именем существующего файла или каталога, то возбуждается исключение.

В классе FileInfo данной паре методов соответствует свойство Attributes типа FileAttributes, доступное как для чтения, так и для записи. Попытка изменения свойства Attributes для *несуществующего* файла или каталога приводит к возбуждению исключения, а при чтении свойства возвращается –1.

```
static DateTime GetCreationTime(string name);                    File
static void SetCreationTime(string name, DateTime newTime);
```

Метод GetCreationTime позволяет определить, а метод SetCreationTime – изменить дату и время создания файла или каталога с именем name. Если name не содержит имя существующего файла или каталога, то возбуждается исключение. Для хранения даты и времени используется объект типа DateTime.

В классе FileInfo данной паре методов соответствует свойство CreationTime типа DateTime, доступное как для чтения, так и для записи. Если файл/каталог не существует, то при доступе к этому свойству на чтение возвращается дата 01.01.1601 3:00:00, а при доступе на запись возбуждается исключение.

Помимо даты и времени создания файла/каталога классы File и FileInfo позволяют определить и изменить ряд других его «временны***х***» характеристик. Для доступа к каждой такой характеристике в классе File предусмотрены два метода (с префиксами Get и Set), а в классе FileInfo – свойство типа DateTime, доступное для чтения и записи. Примерами таких характеристик являются *дата и время последнего доступа* (методы GetLastAccessTime и SetLastAccessTime, свойство LastAccessTime) и *дата и время последней модификации* (методы GetLastWriteTime и SetLastWriteTime, свойство LastWriteTime). В случае отсутствия файла/каталога эти методы и свойства ведут себя аналогично методам и свойству, связанным с датой создания.

Если после последнего обращения к одному из указанных свойств класса FileInfo оно могло измениться, то перед повторным обращением к нему необходимо обновить информацию о файле/каталоге, вызвав метод Refresh (без параметров) класса FileInfo.

Класс FileInfo содержит также ряд экземплярных свойств (только для чтения), которые не имеют соответствий в классе File. Перечислим эти свойства (все они, кроме Length, могут использоваться и в том случае, когда объект this типа FileInfo связан не с файлом, а с каталогом):

- Directory – объект типа DirectoryInfo, содержащий информацию о каталоге, в котором содержится файл (если файл не существует, то при обращении к данному свойству возбуждается исключение);

- DirectoryName – строка, содержащая полный путь к файлу (данный путь не обязан существовать; завершающий символ «\» указывается только в случае корневого каталога);
- Extension – строка, содержащая расширение файла (непустое расширение дополняется слева точкой);
- Name – строка, содержащая имя файла (с расширением) без предшествующего пути;
- FullName – строка, содержащая полное имя файла;
- Length – число типа long, равное длине файла в байтах (если файл не существует или объект this типа FileInfo связан с каталогом, то при обращении к данному свойству возбуждается исключение).

6.5.5. Чтение и запись данных с помощью методов класса File

Класс File включает ряд методов, позволяющих организовать чтение или запись файловых данных, не требующие начальных действий по открытию файла и завершающих действий по его закрытию (указанные действия выполняются автоматически).

```
static byte[] ReadAllBytes(string path);                                File
static string[] ReadAllLines(string path[, Encoding encoding]);
static string ReadAllText(string path[, Encoding encoding]);
```

Эти методы обеспечивают чтение содержимого файла с именем path и его запись в оперативную память. Для чтения двоичных данных предназначен метод ReadAllBytes, записывающий файловое содержимое в массив байтов. Для чтения текстовых данных предназначены методы ReadAllLines и ReadAllText, первый из которых записывает файловое содержимое в массив строк, а второй – в одну строку, содержащую, помимо текста, и маркеры конца строк. При чтении из текстового файла можно дополнительно указать его кодировку, используя параметр encoding.

```
static void WriteAllBytes(string path, byte[] bytes);                   File
static void WriteAllLines(string path, string[] contents[,
                Encoding encoding]);
static void WriteAllText(string path, string contents[,
                Encoding encoding]);
static void AppendAllText(string path, string contents[,
                Encoding encoding]);
```

Эти методы предназначены для записи данных в файл с именем path. Если файл с указанным именем не существует, то он создается. Метод WriteAllBytes записывает в файл набор байтов bytes, прочие методы – строковые данные contents, представленные либо в виде массива строк, либо в виде одной строки, включающей маркеры конца строк. Метод AppendAllText, в отличие от остальных методов, не удаляет прежнее содержимое

файла (новые данные дописываются в конец файла). Все методы, связанные с записью текстовых данных, могут содержать параметр `encoding`, определяющий используемую при записи кодировку.

Применение этих методов требует размещения всех файловых данных в оперативной памяти, что, как правило, оказывается менее эффективным по сравнению с алгоритмами, выполняющими поэлементную файловую обработку.

Чтобы снизить затраты, связанные с выделением оперативной памяти, в версии .NET Framework 4.0 в указанный набор методов были включены следующие варианты методов, обрабатывающих наборы строк:

```
static IEnumerable<string> ReadLines(string path[,                File
                Encoding encoding]);
static void WriteAllLines(string path,
                IEnumerable<string> contents[, Encoding encoding]);
static void AppendAllLines(string path,
                IEnumerable<string> contents[, Encoding encoding]);
```

В этих методах строки могут размещаться в любых коллекциях, реализующих интерфейс `IEnumerable<string>` (например, в динамических массивах `List<string>`), и, главное, при использовании последовательностей, связанных с технологией LINQ (см. часть 3), оказывается возможным «поэлементное» выполнение операций файлового чтения/записи. Например, 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 полученной последовательности в цикле `foreach`, что позволяет получить более эффективный *построчный* вариант обработки текстовых файлов. Новые варианты методов `WriteAllLines` и `AppendAllLines` также оказываются более эффективными, если указываемая в них строковая последовательность `contents` генерируется в ходе выполнения запросов LINQ, поскольку в этой ситуации строковые элементы последовательности записываются в файл *по мере их формирования*, и в оперативной памяти не требуется выделять место для *одновременного* хранения всех элементов.

Указанные преимущества методов, использующих последовательности LINQ, обеспечиваются благодаря особому свойству запросов LINQ – их *отложенному*, или «ленивому» характеру (см. п. 10.2).

6.5.6. Классы Directory и DirectoryInfo

Классы `Directory` и `DirectoryInfo` предназначены для работы с *каталогами*. Класс `Directory`, подобно ранее рассмотренному классу `File`, содержит только классовые методы; при этом первым параметром большинства методов является имя `name` обрабатываемого каталога. Класс `DirectoryInfo`, подобно классу `FileInfo`, содержит свойства и экземплярные методы, для

доступа к которым необходимо создать объект данного класса, указав в его конструкторе имя обрабатываемого каталога. В конструкторе класса DirectoryInfo нельзя указывать значение null, пустую строку или строку, состоящую только из пробелов, а также строку, содержащую недопустимые символы (например, «<», «>», «|», «"», «*», «?»).

Для всех описанных ниже методов в качестве параметра name можно указывать как абсолютные, так и относительные имена каталогов; в последнем случае каталог считается подкаталогом текущего каталога.

Вначале опишем важные методы класса Directory, которые не имеют соответствий в классе DirectoryInfo.

```
static string GetCurrentDirectory();                                Directory
static void SetCurrentDirectory(string newName);
```

Метод GetCurrentDirectory позволяет определить, а метод SetCurrentDirectory – изменить текущий каталог, т. е. *рабочий каталог приложения*. Если параметр newName не является именем существующего каталога, то возбуждается исключение. Параметр newName может содержать или не содержать в конце символ «\»; строка, возвращаемая методом GetCurrentDirectory, оканчивается символом «\» только в случае, если текущий каталог является корневым каталогом.

```
static string[] GetLogicalDrives();                                 Directory
```

Возвращает массив строк с именами корневых каталогов имеющихся логических дисков (проверка наличия носителей в дисководах и устройствах для чтения компакт-дисков при этом не производится). Каждая строка имеет вид "*<буква>*:\", например: "C:\".

Все прочие методы класса Directory имеют аналоги в классе DirectoryInfo, поэтому при их описании сразу будут указываться имена и особенности соответствующих свойств или методов класса DirectoryInfo.

```
static string[] GetFiles(string name[, string mask[,               Directory
                SearchOption option]]);
static string[] GetDirectories(string name[, string mask[,
                SearchOption option]]);
static string[] GetFileSystemEntries(string name[, string mask]);
```

Данные методы возвращают массив строк с полными именами файлов (метод GetFiles), подкаталогов (метод GetDirectories) или одновременно файлов и подкаталогов (метод GetFileSystemEntries) из каталога name. Возвращаемые имена подкаталогов не оканчиваются символом «\».

Если указан параметр mask, то возвращаются имена только тех файлов/каталогов, которые удовлетворяют указанной *маске*; если параметр mask не указан, то его значение считается равным «*», что соответствует *любым* именам файлов/каталогов. Помимо символа «*», обозначающего любое количество любых символов, в маске можно указывать обычные

символы, а также символ «?», обозначающий ровно один произвольный символ.

Если указан параметр option перечислимого типа SearchOption, то, в зависимости от его значения, поиск файлов/каталогов может проводиться не только в указанном каталоге (вариант SearchOption.TopDirectoryOnly), но и во всех его подкаталогах любого уровня вложенности (вариант SearchOption.AllDirectories). Если параметр option отсутствует, то поиск проводится только в указанном каталоге.

Если каталог с именем name не существует, то возбуждается исключение.

В классе DirectoryInfo данным методам соответствуют методы GetFiles, GetDirectories и GetFileSystemInfos с теми же параметрами, за исключением параметра name. Тип возвращаемого значения методов класса DirectoryInfo отличается от массива строк: метод GetFiles возвращает массив объектов типа FileInfo, метод GetDirectories возвращает массив объектов DirectoryInfo. Метод GetFileSystemInfos возвращает массив объектов типа FileSystemInfo (данный тип является общим предком типов FileInfo и DirectoryInfo), причем элементы данного массива, связанные с *файлами,* фактически будут иметь тип FileInfo, а элементы, связанные с *каталогами*, – тип DirectoryInfo.

```
static DirectoryInfo CreateDirectory(string name);                Directory
```

Создает последовательность вложенных каталогов, указанных в строке name, и возвращает объект типа DirectoryInfo, связанный с созданным каталогом. Если указанный каталог уже существует, то метод возвращает данный каталог, не выполняя никаких дополнительных действий. Если параметр name равен null, является пустой строкой или строкой, состоящей из пробелов, а также если создать требуемые каталоги невозможно (например, из-за ошибочно указанного имени диска), то возбуждается исключение. Если параметр name является именем *существующего файла*, то возбуждается исключение.

В классе DirectoryInfo данному методу соответствует метод Create без параметров (создается последовательность каталогов, определяемая объектом типа DirectoryInfo, вызвавшим метод Create). Метод Create не имеет возвращаемого значения.

```
static void Move(string name, string newName);                    Directory
```

Переименовывает файл или каталог name, заменяя его имя на newName. Файл/каталог name, а также путь, указанный в параметре newName, должны существовать; имя newName не должно быть именем существующего файла/каталога (при нарушении любого из этих условий возбуждается исключение). Заметим, что в данном методе (в отличие от метода Move класса File) новое имя не может совпадать со старым, а переименование должно

проводиться в пределах одного и того же диска (при нарушении этих условий возбуждается исключение).

В классе DirectoryInfo данному методу соответствует метод MoveTo с параметром newName; в результате успешного выполнения метода MoveTo вызвавший его объект типа DirectoryInfo связывается с именем newName.

```
static void Delete(string name[, bool recursive]);                Directory
```

Если параметр recursive не указан или равен false, то метод обеспечивает удаление *пустого* каталога с именем name (если каталог не является пустым, то возбуждается исключение). Если параметр recursive равен true, то удаляемый каталог может содержать файлы и подкаталоги, которые также удаляются. Если каталог с указанным именем не существует или доступен только для чтения, то возбуждается исключение. Исключение возбуждается также в случае, когда указанный каталог является рабочим каталогом какого-либо работающего приложения.

В классе DirectoryInfo имеются оба варианта данного метода; эти варианты не содержат параметра name.

```
static bool Exists(string name);                                  Directory
```

Возвращает true, если каталог с именем name существует, и false в противном случае. В частности, метод возвращает false, если параметр name равен null, является пустой строкой или является именем *файла* (пусть даже и существующего). Данный метод никогда не возбуждает исключения.

В классе DirectoryInfo данному методу соответствует логическое свойство Exists, доступное только для чтения.

```
static string GetDirectoryRoot(string name);                      Directory
static DirectoryInfo GetParent(string name);
```

Метод GetDirectoryRoot возвращает имя корневого каталога вида "<*буква*>:\" для файла или каталога с именем name. Если имя диска в параметре name не указано, то возвращается корневой каталог рабочего каталога приложения. Метод GetParent возвращает объект типа DirectoryInfo, соответствующий каталогу, в котором содержится файл/каталог name. Если name является корневым каталогом, то возвращается null.

В качестве имени name в обоих методах можно указывать любое допустимое имя файла/каталога, в том числе и с несуществующим именем диска. Если имя является недопустимым (например, является пустым), то возбуждается исключение.

В классе DirectoryInfo методам GetDirectoryRoot и GetParent соответствуют свойства Root и Parent типа DirectoryInfo, доступные только для чтения.

Класс Directory, подобно классу File, содержит ряд методов для получения «временны**х**» характеристик файлов и каталогов (дата и время соз-

дания, дата и время последнего доступа и т. д.). Данные методы имеют в точности такие же имена, что и методы класса `File`, и работают совершенно аналогично. Это же относится и к набору «временны*ы*х» свойств, который является одинаковым у классов `FileInfo` и `DirectoryInfo`.

В заключение перечислим экземплярные свойства класса `DirectoryInfo`, которые не имеют соответствий в классе `Directory` (все они, кроме свойства `Attributes`, доступны только для чтения; следует также отметить, что все эти свойства могут использоваться и в том случае, когда объект `this` типа `DirectoryInfo` связан не с каталогом, а с файлом):

- `Attributes` – объект типа `FileAttributes`, содержащий информацию об атрибутах каталога и доступный как для чтения, так и для записи. В случае *несуществующего* каталога данное свойство ведет себя так же, как одноименное свойство класса `FileInfo`;
- `Extension` – строка, содержащая расширение каталога (непустое расширение дополняется слева точкой);
- `Name` – строка, содержащая имя каталога (с расширением) без предшествующего пути;
- `FullName` – строка, содержащая полное имя каталога.

6.5.7. Класс DriveInfo

Класс `DriveInfo` предназначен для получения информации о *логических дисках* компьютера.

Для получения объекта типа `DriveInfo` можно вызвать конструктор класса, передав ему в качестве строкового параметра букву требуемого логического диска (`"A"–"Z"`, регистр не важен) или любое допустимое имя файла или каталога, содержащее имя данного диска (например, `"C:\"`). У данного класса имеется также классовый метод `GetDrives` (без параметров), который возвращает массив объектов `DriveInfo`, связанных со *всеми* обнаруженными на компьютере логическими дисками.

Вся информация о логическом диске, связанном с объектом типа `DriveInfo`, доступна через его свойства, которые можно разбить на две группы. К первой группе относятся свойства, обращение к которым никогда не приводит к возбуждению исключения (все эти свойства доступны только для чтения):

- `Name` – строка, содержащая имя корневого каталога диска в формате `"<`*буква*`>:\"`, например: `"C:\"`;
- `RootDirectory` – объект типа `DirectoryInfo`, связанный с корневым каталогом диска;
- `DriveType` – перечисление типа `DriveType`, определяющее *тип* диска (см. п. 6.5.2);

- `IsReady` – свойство логического типа, определяющее, доступен ли указанный диск.

Вторую группу образуют свойства, имеющие смысл только для доступных дисков; если диск недоступен, то обращение к ним приводит к возбуждению исключения. Все эти свойства, кроме свойства `VolumeLabel`, доступны только для чтения:

- `DriveFormat` – строка с описанием формата диска, например: `"FAT"`, `"FAT32"`, `"NTFS"` (жесткие диски), `"CDFS"` (CD- и DVD-диски) и т. д.;
- `TotalSize` – размер диска в байтах (целое число типа `long`);
- `TotalFreeSize` – размер свободного пространства диска в байтах (целое число типа `long`);
- `AvailableFreeSize` – размер свободного пространства диска в байтах, которое доступно для текущего пользователя (целое число типа `long`);
- `VolumeLabel` – строка с именем метки диска; данное свойство доступно и для чтения, и для записи.

Часть 2
Объекты, интерфейсы, обобщения

Глава 7. Основы объектной модели .NET

7.1. Класс object и его методы

Все типы платформы .NET Framework являются потомками ссылочного типа `System.Object`. На языке C# ссылочные типы (reference types) определяются с использованием описателя `class` и называются также *классовыми типами*, или *классами*.

Для базового класса `System.Object` в C# можно использовать его синоним `object`. Поскольку все типы – потомки класса `object`, все описываемые ниже методы класса `object` имеются у всех типов .NET Framework.

Класс `object` является общим предком всех классов, структур и перечислений, определенных в библиотеке FCL. Объекты данного класса практически никогда не создаются, хотя в классе предусмотрен конструктор без параметров. Кратко опишем все методы этого класса.

```
virtual bool Equals(object obj);                                    object
static bool Equals(object objA, object objB);
```

Возвращает `true`, если сравниваемые объекты являются *равными*, и `false` в противном случае. В экземплярном методе сравниваются объекты `this` и `obj`, в классовом методе – `objA` и `objB`. Экземплярный метод обычно переопределяется в типах-потомках; в нем, как правило, проводится *поparное сравнение полей* сравниваемых объектов. Выполнение классового метода сводится в вызову переопределенного экземплярного метода. Примеры методов сравнения для некоторых потомков класса `object` приведены в п. 2.1.3 и 2.2.3.

```
static bool ReferenceEquals(object objA, object objB);              object
```

Возвращает `true`, если объекты `objA` и `objB` являются *тождественными* (т. е. двумя ссылками на один и тот же объект), и `false` в противном случае.

```
Type GetType();                                                     object
```

Возвращает объект типа `Type`, содержащий информацию о *фактическом* типе объекта `this`. Пример использования объектов типа `Type` приводится в п. 9.4.

```
virtual int GetHashCode();                                    object
```

Возвращает целочисленный *хэш-код* для данного объекта, используемый при хранении объектов в *ассоциативных массивах* (примерами ассоциативных массивов являются классы `Hashtable` и `Dictionary<TKey, TValue>` – см. п. 9.3 и 9.8). Хэш-код должен удовлетворять двум условиям: (1) равные объекты должны иметь одинаковый хэш-код, (2) значения хэш-кодов для различных объектов данного типа должны быть по возможности равномерно распределены в некотором диапазоне (последнее условие повышает производительность ассоциативных массивов). Напрямую в программах данный метод обычно не используется. Рекомендуется переопределять его во всех типах-потомках, для которых переопределяется метод `Equals`.

```
virtual string ToString();                                    object
```

Возвращает строковое представление объекта `this`. Его реализация в классе `object` возвращает *полное имя типа* (т. е. имя типа, дополненное именем пространства имен, например `"System.Object"`). Переопределяется для многих потомков класса `object`, в частности, для всех элементарных типов.

7.2. Размерные типы

Некоторые потомки `object` являются *размерными типами* (value type). На языке C# размерные типы определяются с использованием описателя `struct` и называются *структурами*. Хотя для описания нового размерного типа достаточно указать в его определении слово `struct`, фактически любой такой тип будет унаследован от типа `System.ValueType`, который, в свою очередь, является прямым потомком класса `System.Object`. Тип `System.ValueType` в программах на C# не используется, его основная роль – служить «фундаментом» для всех размерных типов. Заметим, что этот «прародитель» размерных типов является потомком *ссылочного* типа `object`.

Важным частным случаем размерных типов являются *перечисления*, фактически представляющие собой набор целочисленных констант. Для определения нового перечисления на C# предназначен описатель `enum`. При этом каждый тип-перечисление является потомком `System.Enum`, который, в свою очередь, является прямым потомком класса `System.ValueType`.

Структурами являются все числовые типы, тип `char` и тип `bool` (таким образом, структурами являются все элементарные типы C#, за исключением классов `object` и `string`).

Перечислим *особенности размерных типов* (т. е. структур):

- структуры не поддерживают наследования (за исключением возможности *упаковки* – неявного приведения к типу `object`);

- структуры не могут содержать описания конструктора без параметров (хотя он всегда доступен и обеспечивает инициализацию членов структуры по умолчанию – посредством побитового обнуления). Если конструктор структуры не вызван, то необходимо явно задать значения всех ее полей, иначе ее нельзя будет использовать;
- структуры не могут содержать финализаторы и виртуальные методы;
- для полей структуры нельзя использовать инициализаторы;
- в явно определяемых конструкторах необходимо задавать значения всех полей.

В качестве примера приведем описание структуры с тремя ошибками, которое станет правильным, если заменить `struct` на `class`:

```
public struct Point
{
  int x = 1;
  int y;
  public Point() {}
  public Point(int x) { this.x = x; }
}
```

7.3. Nullable-структуры

Nullable-структуры – это структуры с возможностью хранения «неопределенного» (indefined) значения – `null`. Они реализуются с помощью обобщенной структуры `Nullable<T>`, однако в языке C# предусмотрен синтаксис, упрощающий описание Nullable-типов и работу с ними. Nullable-тип может быть получен из обычного структурного типа добавлением знака «?», например: `int?`, `bool?`, `Point?`. Преобразование `vtype` → `vtype?` выполняется неявно (т. е. автоматически), обратное преобразование требует явного *приведения типа* (cast), например `i = (int)inull` (но если при выполнении данного оператора переменная `inull` окажется равной `null`, то будет возбуждено исключение `InvalidOperationException`). Nullable-типы могут использоваться в выражениях; для большинства операций наличие null-операнда возвращает `null`. Исключением является операция сравнения на равенство `==`, которая возвращает `true`, если оба операнда равны `null`; кроме того, любая операция отношения (`<`, `>`, `<=`, `>=`) с `null` возвращает `false`; для `bool?` значение null интерпретируется как *неизвестное* значение, поэтому для `b`, равного `null`, выражение `b && false` вернет `false`, а `b || true` вернет `true` (в других случаях, например `b && true`, возвращается `null`).

В языке C# имеется операция `??`, которая возвращает значение первого операнда, если он не равен `null`, и второго операнда в противном случае, например `i = inull ?? 0`. В отличие от операции явного приведения типа,

рассмотренной ранее, применение операции ?? никогда не приводит к возбуждению исключения. Заметим, что операция ?? может применяться не только к Nullable-типам, но и к обычным ссылочным типам.

7.4. Упаковка и распаковка данных размерного типа

При присваивании объекта размерного типа переменной, которая имеет тип object или интерфейсный тип (в частности, при передаче такого объекта в качестве object-параметра), выполняется *упаковка* этого объекта. Обратное действие называется *распаковкой*. Упаковка выполняется неявно, распаковка требует явного преобразования.

При упаковке всегда создается *копия* исходного объекта, она размещается в куче (динамической памяти), и ссылка на нее возвращается ссылочному объекту.

Пример:

```
int i = 10;
object o = i;
int j = (int)o;
```

Для упакованных числовых данных правила приведения числовых типов не действуют; упакованные данные можно преобразовать только в «родной» тип. Продолжим предыдущий пример:

```
double d = (double)o; // ошибка времени выполнения
double d = (int)o;    // d равно 10.0
```

Если в объекте o типа object упаковано число типа double, то для его сохранения в переменной int надо выполнить *два* приведения типа:

```
o = 5.7;
int k = (int)(double)o; // k равно 5, поскольку при приведении (int)
                        // дробная часть отбрасывается
```

Приведем более содержательный пример упаковки:

```
ArrayList L = new ArrayList();
L.Add(10);   // число 10 упаковывается,
             // так как метод Add описан как Add(object value)
int i = (int)L[0];
```

По возможности упаковки следует избегать. Иногда это можно сделать простой модификацией исходного кода. В последующих примерах предполагается, что i – переменная типа int, o – переменная типа object, содержащая упакованное целое.

```
Console.WriteLine(i + ", " + (int)o); // 2 упаковки и 1 распаковка
```

Напомним, что приведенное выражение преобразуется к вызову статического метода Concat класса string с тремя параметрами типа object.

```
Console.WriteLine(i + ", " + o);            // 1 упаковка
Console.WriteLine(i.ToString() + ", " + o); // 0 упаковок
```

Хотя, казалось бы, ToString – виртуальный метод, вызов которого требует предварительной упаковки (поскольку для позднего связывания надо иметь таблицу виртуальных методов, присутствующую только у ссылочных типов), компилятор обходится без этого, так как он ориентируется на тип i. Это размерный тип, поэтому в позднем связывании нет необходимости (у размерных типов не может быть потомков, в которых будет переопределен виртуальный метод), и поэтому уже на этапе компиляции правильно выбирается метод ToString для типа int.

```
Console.WriteLine(i); // 0 упаковок
```

В данном примере не используется упаковка, так как в классе Console имеется перегруженный вариант метода WriteLine с параметром типа int. Этот пример показывает, почему разработчики класса Console не ограничились реализацией метода с «универсальным» параметром WriteLine(object o): такой метод будет недостаточно эффективен для размерных типов.

С появлением *обобщенных коллекций* количество ситуаций, в которых требуется упаковка, существенно уменьшилось (см. п. 9.2–9.3).

7.5. Модификаторы доступа

Приведем модификаторы доступа, предусмотренные в C# для *типов*:

- internal, public (по умолчанию для невложенных типов используется модификатор internal, означающий доступность типа в пределах содержащей его *сборки*, т. е. exe-файла или библиотеки – dll-файла);
- abstract (если класс имеет данный модификатор, то для него нельзя создавать экземпляры, однако этот класс может содержать абстрактные методы);
- sealed («запечатанный» класс, для которого нельзя порождать классы-потомки);
- static (имеет тот же смысл, что и модификатор sealed, но должен применяться к классу, состоящему исключительно из статических членов).

Один и тот же класс не может одновременно иметь модификаторы abstract и sealed; если для некоторого запечатанного класса надо запретить создавать экземпляры, то достаточно определить для него единственный конструктор без параметров, являющийся закрытым (private).

Для *членов класса* предусмотрены основные модификаторы private, protected, internal, public (по умолчанию для члена класса используется модификатор private).

Дополнительно для *полей* могут быть указаны следующие модификаторы:

- `static` (статическое, или классовое, поле);
- `readonly` (значение поля с таким модификатором можно задать только в конструкторе или при описании поля, потом оно доступно только для чтения); данный модификатор можно использовать совместно с модификатором `static`;
- `const` (значение поля с таким модификатором указывается при описании поля и вычисляется на этапе компиляции, после чего это значение подставляется всюду в код); const-поле неявно считается статическим, однако для него модификатор `static` не указывается; атрибут `const` допустимо указывать только для числовых типов, `bool`, `char`, `string`, а также перечислимых типов.

Дополнительно для *методов* могут быть указаны следующие модификаторы:

- `static` (статический, или классовый, метод);
- `virtual` (виртуальный метод); может использоваться только для нестатических методов; определяет первый метод в *цепочке виртуальных методов*; для виртуальных методов реализуется *позднее связывание* (связывание на этапе выполнения), при этом выбирается метод класса, ближайшего в цепочке виртуальности к *фактическому* классу объекта, причем перебор методов начинается с метода того класса, который указан при *описании* объекта;
- `override` (метод с таким модификатором *продолжает* цепочку виртуальности в классе-потомке);
- `new` (используется при разрыве цепочки виртуальности или переопределении невиртуального метода в классе-потомке);
- `abstract` (используется при описании абстрактного метода, т. е. метода без реализации); допустим только в классах с модификатором `abstract`);
- `sealed` (используется при описании методов, которые нельзя переопределить).

7.6. Перегрузка методов и конструкторы

В языках платформы .NET активно используется *перегрузка методов*. Перегруженные методы имеют одинаковые имена, но списки их параметров должны различаться. В частности, в C# для разных перегруженных методов могут использоваться параметры, имеющие и не имеющие модификатор `ref`, например `int x` и `ref int x` (именно поэтому при вызове методов с ref-параметрами необходимо указывать эти модификаторы). Однако не допускаются перегруженные методы, различающиеся только *типом* модификатора параметра (`ref` или `out`); это объясняется тем, что различие между

модификаторами учитывается только на этапе компиляции. Тип возвращаемого значения при перегрузке методов также не принимается во внимание; поэтому методы, различающиеся только возвращаемым типом, перегружены быть не могут.

Конструкторы экземпляров, в отличие от остальных членов класса, в C# *не наследуются*. Если ни один конструктор не определен в классе явным образом, то компилятор автоматически добавляет в класс конструктор без параметров. После заголовка конструктора можно указывать вариант вызова конструктора предка (`: base(…)`) или другого конструктора этого же типа (`: this(…)`). Если после заголовка не указан вызов конструктора, то по умолчанию вызывается конструктор предка без параметров (`: base()`), а если его нет, то возникает ошибка компиляции. Конструкторы могут быть закрытыми.

Класс может содержать *статический конструктор*, который используется для инициализации статических членов и выполнения действий, общих для класса в целом. Такой конструктор всегда описывается с атрибутом `static`, не имеет параметров и всегда является закрытым (хотя атрибут `private` для него не указывается). Статический конструктор вызывается *один* раз для данного типа, этот вызов гарантированно выполняется перед первым созданием экземпляра или первым обращением к статическому полю или методу. Но в остальном порядок вызова статических конструкторов определяется средой выполнения. Например, статический конструктор класса-потомка может быть вызван раньше статического конструктора класса-предка.

7.7. Свойства

Свойства (properties) выглядят как поля, но обрабатываются с помощью методов.

Приведем пример описания свойства (в этом примере *свойство* `Name` просто обеспечивает доступ к закрытому *полю* `name`):

```
string name;
public string Name
{
  get { return name; }
  set { name = value; }  // value – имя параметра метода get,
            // тип параметра value совпадает с типом свойства
}
```

Методы `get` и `set` (*аксессоры*) автоматически генерируются компилятором и получают имена `get_XXX` и `set_XXX`, где `XXX` – имя свойства. Если метод `set` отсутствует, то свойство будет доступно только для чтения.

Свойства используются исключительно для повышения наглядности (по сравнению с вызовом соответствующих методов). Например:

```
a.Name += "***";
  // выполняется следующий код: a.set_Name(a.get_Name() + "***");
```

Главное отличие свойств от полей состоит в том, что для свойства можно обеспечить *контроль за его изменением*, а также *выполнение побочных действий*. Приведем более содержательный пример свойства:

```
class Button
{
  int width;
  public Width
  {
    get { return width; }
    set
    {
      if (value == width) return; // если новое значение совпадает
                  // со старым, то никакие действия не выполняются
      if (value <= 0)
        throw
          new ArgumentOutOfRangeException("(сообщение об ошибке)");
            // обработка ошибочного нового значения
      width = value;
      Redraw(); // выполнение побочного действия (перерисовка кнопки)
    }
  }
…
}
```

Свойства с параметрами называются *индексаторами*. Приведем пример описания индексатора для доступа к элементам закрытого массива `a`:

```
string[] a;
public string this[int index]
{
  get { return a[index]; }
  set { a[index] = value; }
}
```

Методы `get` и `set` индексатора по умолчанию генерируются компилятором с именами `set_Item` и `get_Item` и соответствующим списком параметров (параметр `value` в методе `set_Item` указывается последним).

Индексаторы применяются непосредственно к имени экземпляра. В качестве примера можно указать индексатор класса `string`, возвращающий символ с требуемым индексом: `s[i]` (данный индексатор является свойством только для чтения, в то время как аналогичный индексатор класса `StringBuilder` доступен как для чтения, так и для записи). Индекс не обязан быть целочисленным (в отличие от индексов массива). Индексов может быть несколько; допускается перегрузка индексаторов.

7.8. Инициализаторы объектов, анонимные типы и описатель var (C# 3.0)

Сразу после вызова конструктора можно указать в фигурных скобках *инициализаторы* для требуемых полей или свойств. Эта новая возможность, появившаяся в C# 3.0, используется исключительно для более краткой записи, хотя разновидность подобной конструкции необходима для создания анонимных типов, описываемых далее.

Пример:

```
public class Pupil
{
  public string Name;
  public int Age;

  public Pupil() {}
  public Pupil(string n) { Name = n; }
}
…
Pupil p1 = new Pupil { Name = "Петя", Age = 14 }
Pupil p2 = new Pupil("Вася") { Age = 15 }
```

Обратите внимание на то, что пустые скобки `()` после имени конструктора в инициализаторе можно опускать, а в фигурных скобках указываются не операторы, а список пар, разделяемых запятой, причем в конце списка знак препинания не ставится.

Еще одним нововведением C# 3.0 являются *анонимные типы* – безымянные классы, создаваемые «на лету» и содержащие набор свойств, *доступных только для чтения*. Для создания экземпляра анонимного типа используется конструкция, аналогичная инициализатору объекта, за исключением того, что после слова `new` не указывается имя типа:

```
var p3 = new { Name = "Коля", Age = 14 };
```

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(p3.Name + ", " + p3.Age); // Коля, 14
p3.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 = "Дима";
int Age = 10;
var p4 = new { Name, Age };
Console.WriteLine(p4.Name + ", " + p4.Age); // Дима, 10
```

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 p3 и p4 имеют один и тот же анонимный тип и поэтому совместимы по присваиванию:

```
p4 = p3;
```

Если определить объекты p5 и p6 так, как указано ниже, то они будут иметь различные типы и, кроме того, их типы будут отличаться от типа объектов p3 и p4:

```
var p5 = new { Name1 = "Ваня", Age = 12 };
var p6 = new { Age = 13, Name = "Витя" };
```

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(p3);  // { Name = Коля, Age = 14 }
Console.WriteLine(p6);  // { Age = 13, Name = Витя }
```

Анонимные типы применяются, в основном, при работе с запросами LINQ (см. часть 3). Описатель var может также использоваться в качестве заменителя длинного имени типа (пример подобного использования описателя var приводится далее в п. 8.1.1).

7.9. Делегаты и события

Делегат – аналог процедурного типа. Любой делегат реализуется компилятором в виде специального класса – потомка класса System.MulticastDelegate, однако C# позволяет обойтись без явного описания класса; вместо этого достаточно указать одну строку:

```
public delegate int Transformer(int i); // новый тип делегата,
         // позволяющий хранить метод с сигнатурой int (int i)
```

Теперь в каком-либо классе можно описать переменную типа Transformer и присвоить ей какой-либо метод с соответствующей сигнатурой:

```
public int Triple(int i)
  { return 3*i; }
…
Transformer t = Triple;
```

В ранних версиях C# приходилось вызывать *конструктор делегата* (начиная с версии 2.0 это является необязательным):

```
t = new Transformer(Triple);
```

В случае экземплярных методов делегату передается метод вместе с именем экземпляра:

```
t = a.Triple;
```

В случае статических методов для метода указывается имя класса:

```
t = Math.Abs;
```

Обратите внимание на то, что при указании метода, который присваивается делегату, круглые скобки не используются.

Все делегаты обладают способностью *группировать методы* (эта способность определена в классе `MulticastDelegate`). Приведем пример:

```
public delegate void InfoDelegate(string s);

class Informer
{
  public static InfoDelegate Info;
  public static void NewInfo(string s)
  {
    if (Info != null)
      Info(s);
  }
// этот метод вызывается, когда классу Informer надо проинформировать
// своих "подписчиков"; если не выполнять проверку, то при отсутствии
// подписчиков будет возбуждено исключение
}

class A
{
  public string InfoLog;
  public void GetInfo(string s) { InfoLog += s + ". "; }
}

class B
{
  public static string InfoLog;
  public static void GetInfo(string s) { InfoLog += s + ". "; }
}

class Program
{
  static void Main(string[] args)
  {
    Informer.NewInfo("Event0");   // подписчиков нет
```

```
        A a = new A();
        Informer.Info += a.GetInfo;
        Informer.NewInfo("Event1");
        Console.WriteLine(a.InfoLog); // Event1.
        Console.WriteLine(B.InfoLog); // пустая строка
        Informer.Info += B.GetInfo;
        Informer.NewInfo("Event2");
        Console.WriteLine(a.InfoLog); // Event1. Event2.
        Console.WriteLine(B.InfoLog); // Event2.
        Informer.Info -= a.GetInfo;
        Informer.Info -= a.GetInfo;   // при повторном отказе от подписки
                                      // не выполняется никаких действий
        Informer.NewInfo("Event3");
        Console.WriteLine(a.InfoLog); // Event1. Event2.
        Console.WriteLine(B.InfoLog); // Event2. Event3.
        Console.ReadLine();
    }
}
```

Если в делегате группируются несколько методов, возвращающих значение, то при вызове делегата он вернет значение *последнего* метода из группы, а остальные значения пропадут (хотя имеются средства, позволяющие вызвать по отдельности все методы, сгруппированные в делегате, и тем самым получить все их возвращаемые значения).

Если значение делегата равно `null`, то операция `+=` эквивалентна обычной инициализации `=`.

В реализованной в примере схеме информирования о событиях отсутствует защита от возможных попыток одного из подписчиков вмешаться в действия информатора, связанные с другими подписчиками:

- если один из подписчиков вместо операции `+=` использует операцию `=`, то все остальные подписчики отсоединятся от источника (в нашем примере это произойдет, если заменить оператор `Informer.Info += B.GetInfo` на `Informer.Info = B.GetInfo`);
- если один из подписчиков вместо операции `-=` присвоит делегату значение `null`, то будут отсоединены все подписчики (это произойдет, если заменить `Informer.Info -= a.GetInfo` на `Informer.Info = null`);
- один из подписчиков может «сымитировать» событие, явным образом вызвав делегат информатора (например, с помощью оператора `Informer.Info("False event")`).

События (events) формализуют рассмотренную ранее схему обмена сообщениями между источником широковещательных сообщений и его

подписчиками, повышая ее надежность. Определим основные понятия, связанные с функционированием событий:

- *источник* – это тип, содержащий делегат, связанный с некоторым событием; он определяет, в какой момент надо вызывать этот делегат, инициируя соответствующее событие;
- *подписчик* – это тип, получающий от источника информацию о событии посредством своего метода, присоединенного к этому событию (*обработчика события*); подписчик определяет, когда начинать и заканчивать *прослушивание события*, используя операции += и -= соответственно для подключения обработчика к событию и его отключения от события; подписчик ничего не знает о других подписчиках и не должен вмешиваться в их работу;
- *событие* – это фактически оболочка для делегата, оставляющая открытой только ту часть его функциональных возможностей, которые необходимы для модели «источник–подписчик»; главное предназначение событий – не допускать, чтобы подписчики мешали друг другу.

Для того чтобы использовать возможности событий в приведенном выше примере, достаточно добавить слово `event` к описанию делегата `Info` класса `Informer`:

```
public static event InfoDelegate Info;
```

Если теперь попытаться выполнить любое из «запрещенных» действий, описанных ранее, возникнет ошибка компиляции. В том классе, в котором определено событие, с ним можно работать как с обычным делегатом.

Модифицируем предыдущий пример, используя *модель событий*, применяемую в классах стандартной библиотеки .NET. Кроме того, сделаем событие не классовым, а экземплярным.

```
public class InfoEventArgs : EventArgs
{
  public readonly string Info;
  public InfoEventArgs(string s)
  { Info = s; }
}

public delegate void InfoEventHandler(object sender, InfoEventArgs e);

public class Informer
{
  public event InfoEventHandler Info;
  protected virtual void OnInfo(InfoEventArgs e)
    // диспетчер события Info
```

```
    {
      if (Info != null)
        Info(this, e);
    }
    public void NewInfo(string s)
      // метод, имитирующий наступление события
    {
      OnInfo(new InfoEventArgs(s));
    }
    public string Id;
  }

  class A
  {
    public string InfoLog;
    public void GetInfo(object sender, InfoEventArgs e)
    {
      InfoLog += string.Format("From {0}: {1}. ",
        (sender as Informer).Id, e.Info);
    }
  }

  class B
  {
    public static string InfoLog;
    public static void GetInfo(object sender, InfoEventArgs e)
    {
      InfoLog += string.Format("From {0}: {1}. ",
        (sender as Informer).Id, e.Info);
    }
  }

  class Program
  {
    static void Main(string[] args)
    {
      Informer i1 = new Informer { Id = "i1"},
        i2 = new Informer { Id = "i2"};
      A a = new A();
      i1.Info += a.GetInfo;
      i2.Info += a.GetInfo;
      i1.NewInfo("Event1");
      i2.NewInfo("Event2");
      Console.WriteLine(a.InfoLog); // From i1: Event1. From i2: Event2.
      Console.WriteLine(B.InfoLog); // пустая строка
```

```
        i2.Info -= a.GetInfo;
        i2.Info += B.GetInfo;
        i1.NewInfo("Event3");
        i2.NewInfo("Event4");
        Console.WriteLine(a.InfoLog);
                    // From i1: Event1. From i2: Event2. From i1: Event3.
        Console.WriteLine(B.InfoLog); // From i2: Event4.
        Console.ReadLine();
    }
}
```

Действия компилятора при обработке события:

- создается закрытый делегат;
- определяются два открытых метода, обеспечивающих применение операций += и -= к закрытому делегату; именно эти методы и вызываются при применении операций += и -= к событию;
- любое обращение к событию в пределах содержащего его класса переводится в обращение к соответствующему закрытому делегату.

В C# можно «взять на себя» перечисленные функции компилятора для того, чтобы организовать нестандартные действия по подключению или отключению обработчика:

```
InfoEventHandler info; // закрытый делегат
public event InfoEventHandler Info;
  // открытое событие с явно определенными методами
{
  add { info += value; }
  remove { info -= value; }
}
```

В этом примере явная реализация на самом деле работает аналогично реализации, которую по умолчанию обеспечивает компилятор. Однако теперь при добавлении или удалении обработчиков мы можем выполнять дополнительные действия, например инициировать другие события или выводить отладочную информацию.

Если событие не несет дополнительной информации, а важен лишь факт его наступления (и, возможно, данные о его отправителе), то вместо определения нового делегата можно использовать стандартный делегат `EventHandler`:

```
public delegate void EventHandler(object sender, EventArgs e);
```

7.10. Анонимные методы (C# 2.0) и лямбда-выражения (C# 3.0)

В предыдущем примере мы многократно обращались к методам, связанным с делегатами (добавляли их к делегату или отсоединяли от него). Однако часто единственное назначение метода – присоединить его к неко-

торому делегату. В этом случае нет необходимости описывать этот метод стандартным образом, достаточно создать *анонимный метод* и сразу связать его с делегатом:

```
public delegate int Transformer(int i);
…
Transformer t = delegate(int i) { return 3 * i; };
t(4); // 12
```

Если некоторый метод принимает параметр типа делегата, то в качестве этого параметра можно передать анонимный метод. Если параметры в анонимном методе не используются, то их можно не указывать (вместе с круглыми скобками):

```
button1.Click += delegate{ MessageBox.Show(" … "); };
```

В C# 3.0 концепция анонимных методов получила дальнейшее развитие в *лямбда-выражениях*. С использованием лямбда-выражения определение делегата `t` из предыдущего примера можно переписать так:

```
Transformer t = (int i) => { return 3 * i; };
```

или так:

```
Transformer t = (int i) => 3 * i;
```

или даже так:

```
Transformer t = i => 3 * i;
```

Синтаксис лямбда-выражения:

параметры => выражение_или_операторный_блок

Если параметров нет, или их больше одного, или для них указывается тип, то список параметров заключается в круглые скобки. Как правило, тип параметров указывать не требуется, так как он автоматически определяется («выводится») компилятором по тому делегату, с которым связывается лямбда-выражение. Если метод состоит из одного оператора `return` *выражение*, то вместо операторного блока достаточно указать возвращаемое выражение.

Код анонимного метода или лямбда-выражения может ссылаться на локальные переменные того метода, в котором он определен (так называемые *внешние переменные*). Пример:

```
delegate int IntSequence();
static void Main(string[] args)
{
  int n = 0;
  IntSequence s = () => n++;
  //  или IntSequence s = delegate { return n++; };
  Console.WriteLine(s());  //  0
  Console.WriteLine(s());  //  1
  Console.ReadLine();
}
```

При включении внешней переменной в анонимный метод или лямбда-выражение она *захватывается*, т. е. ее время жизни продлевается до времени жизни соответствующего анонимного метода или лямбда-выражения (это объясняется тем, что все внешние переменные «поднимаются» компилятором до поля некоторого закрытого класса). Приведем модификацию предыдущего примера, в которой благодаря захвату продлевается время жизни локальной переменной `n` метода `Seq`:

```
static IntSequence Seq()
{
  int n = 0;
  return () => n++;
}
static void Main(string[] args)
{
  IntSequence s = Seq();
  Console.WriteLine(s()); //  0
  Console.WriteLine(s()); //  1
  Console.ReadLine();
}
```

7.11. Методы расширения (C# 3.0)

Методы расширения (extensions) позволяют дополнить существующий класс новыми методами, не изменяя определение 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 пример:

```
public static class ExtDemo
{
  public static bool Odd(this int n)
  {
    return n % 2 != 0;
  }
  public static int CharCount(this string s, params char[] charSet)
  {
    int n = 0;
    foreach (char c in s)
      if (charSet.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 определен метод CharCount, позволяющий подсчитать количество символов из определенного набора, содержащихся в строке.

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 требуемые символы 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 (см. п. 4.1.7):

```
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). Кроме того, параметр, определяющий, 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 (см. часть 3).

Если в классе определен метод экземпляра с тем же именем, что и метод расширения, то предпочтение всегда отдается методу экземпляра (метод расширения в подобной ситуации можно вызывать только как статический метод класса, в котором этот метод определен).

Глава 8. Интерфейсы

Интерфейс – это именованный набор сигнатур методов. Все члены интерфейса неявно абстрактны (не содержат реализации и по умолчанию являются открытыми). В интерфейсе могут также определяться события и свойства (в том числе индексаторы), т. е. именно те члены класса, которые могут быть абстрактными. Поскольку статические члены класса не могут быть абстрактными, их нельзя включать в интерфейсы.

Особенности интерфейсов:

- класс может реализовывать несколько интерфейсов, но наследовать он может только от одного класса;
- хотя структура не может наследовать от класса, она может реализовывать интерфейсы.

8.1. Интерфейс IDisposable и освобождение ресурсов

В качестве первого примера интерфейса рассмотрим стандартный интерфейс `IDisposable`, связанный с освобождением неуправляемых ресурсов:

```
public interface IDisposable
{
  void Dispose();
}
```

Любой класс, реализующий интерфейс `IDisposable`, должен реализовать открытый метод `Dispose` (или унаследовать его от класса-предка). Для проверки того, реализует ли класс некоторый интерфейс, можно использовать операцию `is`, а для приведения класса к типу интерфейса можно использовать операцию `as` (или обычную операцию приведения типа):

```
object fs = new System.IO.FileStream(@"c:\a.txt",
  System.IO.FileMode.Create);
if (fs is IDisposable)
```

```
{
  IDisposable d = (IDisposable)fs;
  d.Dispose();
}
```

Заметим, что указанный вариант требует двойной проверки возможности приведения объекта к типу интерфейса. Более эффективный вариант будет следующим:

```
object fs = new System.IO.FileStream(@"c:\a.txt",
  System.IO.FileMode.Create);
IDisposable d = fs as IDisposable;
  if (d != null)
    d.Dispose();
```

Во всех файловых потоках (см. п. 6.2.3, 6.3.1, 6.4.2) метод `Close` является синонимом метода `Dispose`.

Имеется набор рекомендаций, связанных с семантикой удаления объекта:

- однажды удаленный объект восстановлению не подлежит; при обращении к некоторым его методам или свойствам вполне может возникнуть исключение (а при обращении к другим может и не возникнуть);
- повторный вызов метода `Dispose` не приводит к ошибке;
- если удаляемый объект «владеет» другими объектами с интерфейсом `IDisposable`, то, как правило, при своем удалении он должен удалить свои подчиненные объекты (например, так ведет себя форма по отношению к своим компонентам).

Объекты, являющиеся оболочками для неуправляемых ресурсов (например, файловые или сетевые потоки, сетевые сокеты, перья, кисти и растровые изображения из библиотеки GDI+), следует удалять *немедленно* после завершения работы с ними. Имеются исключения из этого правила:

- не следует удалять объект, полученный в виде статического свойства другого объекта (пример: `Brushes.Blue`);
- не следует немедленно удалять один из объектов `BinaryReader` или `BinaryWriter`, если к файловому потоку подключен также другой из указанных объектов (поскольку при этом будет удален сам файловый поток – см. п. 6.3.1).

8.1.1. Оператор using

Для работы с объектами, реализующими интерфейс `IDisposable`, в языке C# предусмотрен особый оператор `using`, обеспечивающий автоматический вызов метода `Dispose` при выходе объекта из области видимости:

```
using (var fs = new System.IO.FileStream(@"c:\a.txt",
  System.IO.FileMode.Create))
```

```
{
  fs.Write(new byte[]{0, 0, 0}, 0, 3);
}
```

В отличие от фрагментов программ, приведенных в предыдущем пункте, в данном случае описать `fs` как `object` нельзя, так как тип, указанный в заголовке `using`, должен реализовывать интерфейс `IDisposable`. Однако в C# 3.0 можно использовать новую возможность: ключевое слово `var`, означающее, что тип объекта должен определяться по его инициализирующему выражению (в частности, конструктору). Если в операторе `using` указывается один оператор, то фигурные скобки можно опустить.

Компилятор преобразует данный фрагмент в следующий (здесь все фигурные скобки обязательны):

```
var fs = new System.IO.FileStream(@"c:\a.txt",
  System.IO.FileMode.Create);
try
{
  fs.Write(new byte[]{0, 0, 0}, 0, 3);
}
finally
{
  if (fs != null)
    fs.Dispose();
}
```

Блок `finally` гарантирует, что метод `Dispose` будет вызван даже в случае возбуждения исключения или преждевременного завершения блока `try` (например, при выполнении оператора `return`).

8.1.2. Сборщик мусора и финализаторы

Следует обратить внимание на то, что метод `Dispose` освобождает именно *неуправляемые ресурсы*. Память, выделенная для размещения объекта в куче, освобождается автоматически специальной системой среды выполнения платформы .NET, называемой *сборщиком мусора* (garbage collector, GC). С классом может быть связан особый метод – *финализатор*, который выполняется в тот момент (точнее, непосредственно перед тем моментом), когда сборщик мусора разрушает объект. Финализатор в основном используется для того, чтобы гарантировать освобождение неуправляемых ресурсов даже в том случае, если ранее они не были освобождены явным вызовом метода `Dispose`. Надо, однако, учитывать, что вызов финализаторов может существенно замедлить процедуру разрушения объектов. «Финализаторы подобны адвокатам: хотя бывают случаи, когда они вам действительно нужны, обычно вы не стремитесь прибегать к их услугам без крайней необходимости» [3, с. 489].

Стандартная схема реализации освобождения неуправляемых ресурсов в финализаторе:

```
class DisposeDemo: IDisposable
{
  public void Dispose()
  // этот метод не требуется переопределять в классах-потомках
  {
    Dispose(true);
    GC.SuppressFinalize(this);
    // подавление вызова финализатора при разрушении объекта
  }
  protected virtual void Dispose(bool disposeIsCalled)
  // этот метод можно переопределять
  {
    if (disposeIsCalled)
    {
      // освобождение ресурсов подчиненных объектов
    }
    // освобождение собственных ресурсов
  }
  ~DisposeDemo() // финализатор
  {
    Dispose(false);
  }
}
```

При вызове метода `Dispose` из финализатора уже нельзя обращаться к подчиненным объектам, так как для них к этому моменту могли быть вызваны финализаторы, обеспечившие их полное разрушение. Собственные ресурсы объект может освобождать всегда; примерами таких ресурсов могут являться непосредственные ссылки на ресурсы операционной системы и временные файлы, которые надо удалить при разрушении объекта.

Почему финализатор замедляет разрушение объектов? Опишем схему работы сборщика мусора:

- сборщик мусора начинает процесс сборки, если памяти недостаточно для размещения очередного объекта или в других ситуациях, требующих уменьшения объема памяти, например, если объем объектов, созданных после последней сборки мусора (объектов поколения 0 – см. далее), превысил некоторое пороговое значение (обычно 256 К – объем кэша процессора);
- сборщик мусора приостанавливает выполнение приложения, начинает перебирать корневые ссылки на объекты и обходит граф объектов, помечая объекты, которых он коснется, как доступные (при попадании на уже помеченные объекты он их пропускает, чтобы не

попасть в бесконечный цикл). По окончании этого процесса все непомеченные объекты считаются мусором;

- объекты, объявленные мусором и не имеющие финализаторов, уничтожаются немедленно. Если же они имеют финализаторы, то они помечаются как подлежащие дополнительной обработке, и для них финализаторы вызываются в отдельном потоке приложения (при этом происходит как бы их временное «воскрешение»);
- все оставшиеся объекты (в том числе и те, для которых были вызваны финализаторы) перемещаются вниз по куче (происходит *уплотнение* кучи). При этом сборщик мусора изменяет значения ссылок в «живых» объектах, которые были перемещены. Кроме того, все обработанные и при этом выжившие объекты переходят в следующее поколение.

Следует подчеркнуть, что объекты, имеющие финализаторы, не разрушаются за одну сборку мусора: на этапе этой первой сборки для них только вызываются финализаторы, а сами они остаются «живыми» (и даже переходят в следующее поколение). Только при очередной сборке мусора в этом поколении данные объекты разрушаются окончательно.

Для оптимизации сборки мусора все объекты разделяются на три поколения: только что созданные (поколение 0), объекты, пережившие одну сборку мусора (поколение 1), и объекты, пережившие хотя бы одну сборку мусора в поколении 1 («долгожители» – поколение 2). Если размер объектов из поколения 0 превышает 256 К, то они подвергаются сборке мусора; «выжившие» помещаются в поколение 1. Если при сборке мусора выясняется, что поколение 1 занимает более 2 М, выполняется сборка мусора в поколении 1, а «выжившие» при этом объекты переходят в поколение 2. Поколение 2 подвергается сборке мусора только в случае, если его размер превысит 10 М. Заметим, что сборщик мусора может адаптироваться к особенностям приложения, изменяя пороговые значения для поколений.

Сборка мусора может быть запущена принудительно: вызовом метода `GC.Collect()` для полной сборки мусора или метода `GC.Collect(n)` для сборки мусора в поколениях, номер которых не превышает `n`; например, для быстрой сборки мусора в поколении 0 достаточно вызвать `GC.Collect(0)`.

Чтобы гарантировать уничтожение объектов, срок жизни которых продлен финализаторами, можно выполнить следующую последовательность действий:

```
GC.Collect();
GC.WaitForPendingFinalizers();
  // ждать, пока не завершится поток с запущенными финализаторами
GC.Collect();  // повторная сборка мусора,
  // уничтожающая объекты с проработавшими финализаторами
```

Принудительную сборку мусора выполнять не рекомендуется. Исключение – приложения, которые большую часть своей работы простаивают (например, приложение, активизирующееся один раз в сутки для отслеживания обновлений); для такого приложения имеет смысл принудительно освободить ресурсы сразу после завершения его «фазы активности».

8.2. Интерфейс ICloneable и явная реализация интерфейсов

Приведенная выше реализация метода `Dispose` – пример *неявной* реализации интерфейса `IDisposable`: метод `Dispose` можно вызвать как из класса, так и из интерфейсного объекта.

При *явной реализации интерфейса* перед именем интерфейсных методов указывается имя интерфейса; в этом случае методы могут быть вызваны только из интерфейсных объектов. Зачем нужна явная реализация? Например, для разрешения конфликтов между интерфейсами, когда в двух интерфейсах содержатся одноименные методы с одинаковыми сигнатурами, или для разрешения конфликтов между интерфейсом и классом, если в классе удобно определить другой метод с тем же именем.

Рассмотрим в качестве примера интерфейс `ICloneable`:

```
public interface ICloneable
{
  object Clone();  // возвращает копию (клон) объекта
}
```

Предположим, что в классе `Demo` реализован этот интерфейс:

```
class Demo: ICloneable
{
  public int X, Y;
  public object Clone()
  {
    return new Demo { X = this.X, Y = this.Y };
  }
}
```

Пример его применения:

```
Demo a = new Demo { X = 10, Y = 20 }:
Demo b = (Demo)a.Clone();
Console.WriteLine(b.X + " " + b.Y); // 10 20
```

Неудобство использования метода `Clone` состоит в том, что, поскольку тип возвращаемого значения метода `Clone` описан как `object`, требуется явное приведение этого объекта к его фактическому типу `Demo`.

Было бы удобно реализовать метод `Clone` с возвращаемым значением типа `Demo`. Но перегрузить методы, различающиеся только типами возвращаемого значения, нельзя. Отказаться от метода `Clone` с типом `object` тоже

нельзя: тогда класс не будет реализовывать интерфейс ICloneable. Выход состоит в явной реализации интерфейса ICloneable:

```
class Demo: ICloneable
{
  public int X, Y;
  public Demo Clone()
  {
    return new Demo { X = this.X, Y = this.Y };
  }
  object ICloneable.Clone()
  {
    return Clone(); // вызывается метод Clone класса, а не интерфейса
  }
}
```

Обратите внимание на то, что при явной реализации метода интерфейса атрибут public указывать нельзя. Явно реализованный интерфейс также нельзя описать как виртуальный, однако в классах-потомках его можно реализовать заново. Атрибут виртуальности при явной реализации не нужен, так как при вызове метода с помощью интерфейсного объекта позднее связывание реализуется автоматически (т. е. метод по умолчанию ведет себя как виртуальный).

Варианты вызова методов Clone:

```
Demo a = new Demo { X = 10, Y = 20 }:
Demo b = a.Clone(); // вызывается метод Clone класса,
                    // приведение типа не требуется
Console.WriteLine(b.X + " " + b.Y); // 10 20
Demo c = (Demo)(a as ICloneable).Clone();
                    // вызывается метод Clone интерфейса
Console.WriteLine(c.X + " " + c.Y); // 10 20
```

8.3. Интерфейсные методы и позднее связывание

Как уже было сказано, при вызове метода посредством интерфейсного объекта он всегда ведет себя как виртуальный, т. е. для него всегда выполняется позднее связывание (даже если он реализован неявно и при этом не имеет атрибута виртуальности). Однако при вызове этого же метода непосредственно из классового объекта (при условии, что этот метод реализован неявно) позднее связывание будет доступно только при включении этого метода в цепочку виртуальности.

Рассмотрим пример.

```
using System;

public interface ICommon
{
```

```
    void DoIt();
}

public class Base: ICommon
{
  void ICommon.DoIt()
  { Console.Write("ICommon(Base) "); }
  public virtual void DoIt()
  { Console.Write("Base "); }
}

public class Derived1: Base, ICommon
{
  void ICommon.DoIt()
  { Console.Write("ICommon(Derived1) "); }
  public new virtual void DoIt()
  { Console.Write("Derived1 "); }
}

public class Derived2: Derived1
{
  public override void DoIt()
  { Console.Write("Derived2 "); }
}

class Program
{
  static void Main()
  {
    Derived2 r1 = new Derived2();
    Derived1 r2 = r1;
    Base r3 = r1;
    ICommon r4 = r1;
    r1.DoIt(); r2.DoIt(); r3.DoIt(); r4.DoIt();
    Console.ReadLine();
  }
}
```

Результат работы программы:

```
Derived2 Derived2 Base ICommon(Derived1)
```

Второй вызов напечатает «Derived2», так как методы `DoIt` для классов `Derived1` и `Derived2` связаны в цепочку виртуальности.

В результате третьего вызова будет напечатано «Base», так как, хотя `Base.DoIt` и был объявлен как виртуальный, соответствующие методы производных типов создают новую цепочку виртуальности, не связанную с

методом DoIt класса Base (следует заметить, что если в описании метода в классе Derived1 не указать слово new, в ходе компиляции данной программы будет выведено предупреждение «'Derived1.DoIt' hides inherited member 'Base.DoIt'»).

Четвертый вызов выведет «ICommon (Derived1)», так как метод DoIt класса Derived2, перекрывая метод DoIt класса Derived1, не перекрывает метод ICommon.DoIt этого же класса. Поэтому вызывается метод ICommon.DoIt того класса, *подключившего интерфейс* ICommon, который является ближайшим по иерархии к Derived2 (т. е. класса Derived1).

Что произойдет, если в описании класса Derived1 убрать реализацию метода ICommon.DoIt? В этом случае четвертый вызов выведет «Derived2». Это связано с тем, что в классе Derived1 указано подключение к интерфейсу ICommon, и, поскольку теперь в нем нет метода ICommon.DoIt, методом, реализующим интерфейс, считается «обычный» метод DoIt, который перекрывается в классе Derived2, образуя цепочку виртуальности, благодаря которой для интерфейса ICommon и вызывается метод DoIt из класса Derived2 (заметим, что если разорвать эту цепочку, например закомментировав атрибут override в описании метода DoIt класса Derived2, то для интерфейса будет вызван метод DoIt класса Derived1).

Если же дополнительно убрать подключение интерфейса ICommon к классу Derived1, то четвертый вызов выведет «ICommon (Base)», поскольку теперь ближайший в иерархии класс, подключивший интерфейс ICommon, – это класс Base, в котором метод интерфейса задан явно (в виде ICommon.DoIt).

Если, далее, заменить атрибуты new virtual в методе DoIt класса Derived1 на атрибут override, то вывод будет следующим (обратите внимание на то, что выбор четвертого метода (для интерфейса ICommon) не изменился):

```
Derived2 Derived2 Derived2 ICommon(Base)
```

Если после всех описанных изменений убрать реализацию метода ICommon.DoIt в описании класса Base, то все четыре вызова методов выведут «Derived2». Почему теперь последний вызов дает «Derived2»? По-прежнему ближайший к Derived2 класс, подключивший интерфейс ICommon, – это Base, однако теперь в нем метод интерфейса описан неявно. Учитывая, что для «обычного» метода DoIt в данном случае имеет место цепочка виртуальности, вызов метода DoIt для интерфейса ICommon (формально выполняемый для ближайшего класса, реализующего интерфейс ICommon, т. е. для класса Base) фактически приводит к вызову метода, последнего в цепочке виртуальности, т. е. метода DoIt класса Derived2.

Убрав дополнительно все атрибуты `virtual` и `override`, мы получим следующий результат (теперь цепочка виртуальности разорвана в том числе и для метода `DoIt`, связанного с интерфейсом `ICommon`):

```
Derived2 Derived1 Base Base
```

Наконец, добавив к описанию класса `Derived2` подключение интерфейса `ICommon`, мы получим следующий результат:

```
Derived2 Derived1 Base Derived2
```

Мы видим, что результат работы первых трех методов не изменился. Однако теперь ближайшим классом, реализующим интерфейс `ICommon`, для класса `Derived2` является сам этот класс `Derived2`, поэтому при вызове метода `DoIt` как метода интерфейса вызывается метод из класса `Derived2`.

Если теперь убрать реализацию метода `DoIt` в классе `Derived2`, то получим следующий результат:

```
Derived1 Derived1 Base Base
```

В пояснении нуждается только результат работы последнего метода. Напомним, что класс `Derived1` *не реализует* интерфейс `ICommon`, поэтому для поиска *метода интерфейса*, унаследованного классом `Derived2`, мы поднимаемся до класса `Base`. Естественно, если подключить к классу `Derived1` интерфейс `ICommon`, то четвертый вызов выведет «Derived1».

8.4. Базовые интерфейсы для коллекций и цикл foreach

Под *коллекциями* в широком смысле подразумеваются классы, содержащие набор элементов. Понятно, что классы-коллекции нельзя выводить от одного общего предка («базовой» коллекции), поскольку природа коллекций может быть самой разнообразной: это и массивы, и динамические массивы (например, `ArrayList`), и даже строки (`string`), поскольку строку можно рассматривать как коллекцию образующих ее символов. Для единообразной обработки различных коллекций необходимо, чтобы все они реализовывали соответствующие интерфейсы, важнейшим из которых является `IEnumerator`, определенный в пространстве имен `System.Collections`:

```
public interface IEnumerator
{
  bool MoveNext();
  object Current { get; }
  void Reset();
}
```

Следует обратить внимание на то, что в данный интерфейс входят не только методы, но и свойство.

Объект, реализующий интерфейс `IEnumerator`, называется *перечислителем*; он может рассматриваться как *курсор*, доступный только для чте-

ния и перемещающийся по элементам коллекции только в направлении «вперед».

Метод MoveNext пытается перейти от текущего элемента коллекции к следующему и возвращает false, если это не удалось (т. е. если предыдущий элемент был последним элементом коллекции); при первом вызове данный метод пытается перейти на первый элемент коллекции. Можно сказать, что метод MoveNext переводит перечислитель вперед на один элемент (при этом перечислитель может оказаться в позиции «за последним элементом»). Свойство Current (только для чтения) возвращает текущий элемент, т. е. элемент коллекции, на который указывает перечислитель (если перечислитель находится в позиции «перед первым» или «после последнего элемента», то обращение к свойству Current возбуждает исключение InvalidOperationException). Элемент возвращается как object, поэтому, как правило, требуется дополнительное приведение значения свойства Current к фактическому типу элемента. Чтобы обеспечить корректную обработку пустых коллекций, перед обращением к свойству Current необходимо предварительно вызвать метод MoveNext. Наконец, метод Reset предназначен для перевода перечислителя в исходное состояние – в позицию «перед первым элементом». Следует заметить, что коллекции могут не поддерживать возврат курсора; в этом случае вызов метода Reset приводит к исключению NotSupportedException.

Для корректного выполнения перебора необходимо, чтобы в ходе перебора анализируемая коллекция оставалась неизменной, поэтому методы MoveNext и Reset реализуются таким образом, чтобы возбуждать исключение при попытке их вызова после какой-либо модификации коллекции.

Сам перечислитель, реализующий интерфейс IEnumerator, как правило, оформляется в виде отдельного класса (этот класс может быть вложенным в класс-коллекцию), а для его получения в классе-коллекции предусматривается специальный метод GetEnumerator – «поставщик перечислителя», являющийся единственным членом интерфейса IEnumerable:

```
public interface IEnumerable
{
  IEnumerator GetEnumerator();
}
```

Таким образом, для того чтобы некоторый объект допускал перебор своих элементов, достаточно, чтобы он реализовывал интерфейс IEnumerable. Вот как должен выглядеть код, обеспечивающий подобный перебор (в приведенном примере мы предполагаем, что объект s является коллекцией, состоящей из элементов типа char; например, s может иметь тип string или char[], или быть динамическим массивом ArrayList при условии, что все его элементы имеют тип char):

```
IEnumerator en = s.GetEnumerator();
while (en.MoveNext())
{
  char c = (char)en.Current;
  Console.WriteLine(c); // или другая обработка элемента коллекции
}
```

Заметим, что данный фрагмент будет корректно выполняться и в случае, если коллекция не содержит ни одного элемента (например, если `s` является пустой строкой). В то же время если среди элементов коллекции окажутся объекты типа, который не приводится к типу `char`, то будет возбуждено исключение.

Замечание. Чтобы указанный фрагмент успешно откомпилировался, к программе необходимо подключить пространство имен `System.Collections`, поскольку именно в нем реализованы интерфейсы `IEnumerator` и `IEnumerable`.

Подобно тому как оператор `using` упрощает использование объектов, реализующих интерфейс `IDisposable` (см. п. 8.1.1), цикл `foreach` упрощает перебор объектов, реализующих интерфейс `IEnumerable`. Вот как выглядит реализация приведенного выше фрагмента с использованием цикла `foreach` (фигурные скобки в данном случае можно не указывать):

```
foreach (char c in s)
{
  Console.WriteLine(c); // или другая обработка элемента коллекции
}
```

Таким образом, цикл `foreach` является синтаксическим сокращением, позволяющим скрыть от программиста действия программы, связанные с созданием перечислителя и вызовом его методов и свойств. Знание механизма работы цикла `foreach` позволяет понять, почему его нельзя использовать, например, для перебора символов объекта `StringBuilder`; это связано с тем, что `StringBuilder` не реализует интерфейс `IEnumerable`.

8.5. Итераторы и конструкция yield

Наряду со средствами, упрощающими *использование* коллекций (цикл `foreach`), в C# предусмотрены специальные средства, упрощающие их *создание*. Таким средством являются *итераторы*, реализуемые с помощью специальной конструкции `yield`.

Начнем с примера использования итератора. Опишем класс `YieldDemo`, содержащий массив строк `S`, и реализуем перечислитель этого класса таким образом, чтобы он последовательно перебирал все символы всех непустых строк, входящих в массив s.

```
class YieldDemo: IEnumerable
```

```
{
  public string[] S = new string[10];
  public IEnumerator GetEnumerator()
  {
    foreach (string s in S)
      if (s != null)
        foreach (char c in s)
          yield return c;
  }
}
static void Main(string[] args)
{
  YieldDemo d = new YieldDemo();
  d.S[0] = "aaa";
  d.S[5] = "ccc";
  foreach (char c in d)
    Console.Write(c);
      // будут выведены 6 символов: aaaccc
  Console.WriteLine();
  d.S[3] = "bb";
  d.S[5] = "cc";
  foreach (char c in d)
    Console.Write(c);
      // будут выведены 7 символов: aaabbcc
  Console.ReadLine();
}
```

Мы видим, что если использовать итератор, то необходимости в создании специального класса-перечислителя не возникает. Подобно тому как оператор `foreach` «скрывает» от программиста действия по использованию перечислителя, оператор `yield` позволяет обойтись без явной реализации подобного класса. «В то время как оператор `foreach` является *потребителем* перечислителя, итератор выступает в роли *производителя* перечислителя» [3, с.153].

Итак, что же делает оператор `yield`? Если «обычный» оператор `return` как бы говорит «Вот значение, которое возвращает этот метод», оператор `yield return` говорит «Вот следующий элемент, полученный от этого перечислителя» (обратите внимание на то, что возвращаемое значение метода, использующего `yield`, имеет тип `IEnumerator`). При каждом появлении оператора `yield` управление передается вызывающему коду, но состояние вызванного метода сохраняется, чтобы он мог продолжить выполнение, когда будет запрошен очередной элемент перечислителя. Срок жизни состояния привязан к сроку жизни перечислителя.

Метод, использующий yield, может иметь возвращаемый тип IEnumerable. В этом случае его возвращаемое значение считается коллекцией, к которой уже присоединен перечислитель, и поэтому данное возвращаемое значение может использоваться непосредственно в цикле foreach:

```
static IEnumerable Fib(int n)
{
  int a = 1, b = 1;
  yield return a;
  yield return b;
  for (int i = 2; i < n; i++)
  {
    int c = a + b;
    yield return c;
    a = b;
    b = c;
  }
}
static void Main(string[] args)
{
  foreach (int f in Fib(10))
    Console.WriteLine(f);
      // будут выведены первые 10 чисел Фибоначчи:
      // 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
  Console.ReadLine();
}
```

Как и в предыдущих примерах, оператор foreach служит удобной «оберткой» для перечислителя, однако можно было бы обойтись и без нее:

```
static void Main(string[] args)
{
  IEnumerator en = Fib(10).GetEnumerator();
  while (en.MoveNext())
  {
    int f = (int)en.Current;
    Console.WriteLine(f);
  }
}
```

После завершения перебора элементов перечислитель, созданный с помощью оператора yield, не может быть использован повторно, даже если он сохранен в переменной типа IEnumerator, поскольку метод Reset для подобного перечислителя не реализован: при попытке его вызова возбуждается исключение NotSupportedException.

Обратите внимание на то, что в методе, реализующем перечислитель с помощью оператора yield, нельзя указывать «обычный» оператор return. В

ситуации, когда требуется досрочно прервать перечисление последовательности элементов, надо использовать специальный оператор `yield break`:

```
static IEnumerable TwoOrThree(bool onlyTwo)
{
  yield return 1;
  yield return 2;
  if (onlyTwo)
    yield break;
  yield return 3;
}
```

Разумеется, в приведенном примере можно обойтись и без специальной конструкции `yield break`:

```
static IEnumerable TwoOrThree(bool onlyTwo)
{
  yield return 1;
  yield return 2;
  if (!onlyTwo)
    yield return 3;
}
```

8.6. Интерфейс IComparable и объекты, допускающие сравнение

В классе `Array` реализовано несколько статических перегруженных методов, позволяющих выполнять сортировку массива (см. п. 3.2.2). Простейший из них принимает единственный параметр – сортируемый массив:

```
Array.Sort(a);
```

В качестве параметра в данном случае допускается передавать только массив, элементы которого можно сравнивать между собой. Для этого достаточно, чтобы тип элементов массива реализовывал интерфейс `IComparable`:

```
public interface IComparable
{
  int CompareTo(object other);
}
```

Вызов метода `a.CompareTo(b)` возвращает отрицательное число, если `a` «меньше» `b`, возвращает 0, если `a` «равно» `b`, и возвращает положительное число, если `a` «больше» `b`. Слова «меньше», «равно» и «больше» взяты в кавычки, поскольку они должны пониматься в более широком смысле, чем обычные арифметические операции сравнения, – как «предшествует», «находится на том же уровне» и «следует после». Хотя параметр метода `CompareTo` описан как `object`, он должен иметь фактический тип, совпадающий с

типом объекта, вызвавшего этот метод; в противном случае возбуждается исключение ArgumentException. В то же время допускается параметр, равный null (даже для объектов размерных типов); при этом считается, что значение null «меньше» любого другого значения любого типа.

Все числовые типы (а также типы bool, char и string) реализуют интерфейс IComparable.

А как быть с классами, для которых интерфейс IComparable не реализован? Например, рассмотрим массив структур Point (из пространства имен System.Drawing):

```
Point[] p = new Point[5];
```

При попытке вызова метода Array.Sort(p) будет возбуждено исключение InvalidOperationException с сообщением «Не удалось сравнить два элемента массива». В подобной ситуации можно определить «пользовательский» алгоритм сравнения, оформив его в виде класса, реализующего интерфейс IComparer:

```
public interface IComparer
{
  int Compare(object a, object b);
}
```

Возвращаемое значение метода Compare имеет тот же смысл, что и возвращаемое значение метода CompareTo, если считать, что параметр a метода Compare соответствует объекту, вызывающему метод CompareTo, а параметр b – параметру other этого метода. Допускаются параметры, равные null; два параметра null считаются равными, в противном случае объект со значением null считается меньшим любого другого объекта.

Для возможности сортировки массива элементов Point достаточно определить для них вспомогательный класс-*компаратор* с интерфейсом IComparer, обеспечивающий сравнение двух объектов типа Point (будем считать меньшей ту точку, которая расположена ближе к началу координат), после чего вызвать вариант метода Array.Sort, принимающий два параметра: сортируемый массив и объект-компаратор:

```
using System;
using System.Drawing;
using System.Collections;

class PointComparer: IComparer
{
  public int Compare(object a, object b)
  {
    Point pa = (Point)a, pb = (Point)b;
    long da = (long)pa.X * pa.X + (long)pa.Y * pa.Y,
      db = (long)pb.X * pb.X + (long)pb.Y * pb.Y;
```

```
        return Math.Sign(da - db);
    // тип long и функция Sign использованы,
    // чтобы избежать целочисленного переполнения
      }
    }
    …
    static void Main(string[] args)
    {
      Point[] p = new Point[4];
      p[0].X = int.MaxValue;
      p[2].Y = 100;
      p[3].X = 40;
      Array.Sort(p, new PointComparer());
      foreach (Point a in p)
        Console.Write(a);
    }
```

В данном случае мы не стали особым образом обрабатывать в методе `Compare` значение `null`, поскольку при сортировке массива структур (т. е. размерных типов) это значение появиться не может. При выполнении метода `Main` будет выведен следующий текст:

```
{X=0,Y=0} {X=40,Y=0} {X=0,Y=100} {X=2147483647,Y=0}
```

Завершая данный пункт, упомянем еще один интерфейс, связанный со сравнением объектов, – `IEqualityComparer`:

```
public interface IEqualityComparer
{
  bool Equals(object a, object b);
  int GetHashCode(object obj);
}
```

Интерфейс `IEqualityComparer` предназначен для того, чтобы переопределить сравнение объектов определенного типа на *равенство* (речь идет именно о переопределении, так как методы `Equals`, как статический, так и экземплярный, имеются у всех классов .NET – см. описание методов базового класса `object` в п. 7.1). Как правило, необходимость в подобном переопределении возникает только в ситуации, когда объекты данного типа используются в качестве ключа некоторого ассоциативного массива (в стандартной библиотеке .NET ассоциативные массивы реализованы в виде двух классов: `Hashtable` и `Dictionary<TKey, TValue>`). Поскольку при реализации ассоциативного массива применяется хэширование, необходимо переопределить не только метод `Equals`, но и метод `GetHashCode`, обеспечив согласованное поведение этих методов (см. требования к хэш-кодам, приведенные в п. 7.1).

Глава 9. Обобщения

Обобщения (generics) были введены в .NET Framework 2.0. Их наличие позволяет уменьшить размер программного кода и сделать его более наглядным, безопасным и эффективным.

В языке C# можно использовать обобщенные методы и обобщенные типы; при этом обобщенные методы могут описываться и в обычных, необобщенных классах.

9.1. Обобщенные методы

Поставим задачу: описать универсальный метод, позволяющий менять местами содержимое двух переменных. Учитывая, что все типы являются потомками `object`, можно попытаться реализовать этот метод следующим образом:

```
static void Swap(ref object a, ref object b)
{
  object v = a;
  a = b;
  b = v;
}
```

Этот метод успешно откомпилируется, однако при попытке его использования возникнут проблемы.

Например, попытаемся поменять местами две строки:

```
string s1 = "a", s2 = "b";
Swap(ref s1, ref s2); // ошибка компиляции
```

При компиляции этого фрагмента будет выведено сообщение о том, что тип `ref string` нельзя преобразовать к типу `ref object`.

Попытка явного приведения типа также к успеху не приведет:

```
Swap(ref (object)s1, ref (object)s2); // ошибка компиляции
```

Теперь будет выведено сообщение о том, что ref-параметр должен быть переменной, допускающей присваивание (assignable variable), в то время как выражение `(object)s1` переменной не является (в частности, его нельзя указывать в левой части операции присваивания).

Проблему можно решить, введя две вспомогательные переменные типа `object`:

```
object o1 = s1, o2 = s2;
Swap(ref o1, ref o2); // ошибки нет
```

Однако в результате выполнения метода `Swap` поменяются местами значения *ссылок* `o1` и `o2`, но не `s1` и `s2`. Поэтому для перемены местами `s1` и `s2` потребуется выполнить еще два присваивания, которые, к тому же, должны сопровождаться приведением типов:

```
s1 = (string)o1;
```

```
s2 = (string)o2;
```

В случае размерных типов возникнут аналогичные проблемы:

```
int n1 = 1, n2 = 2;
object o1 = n1, o2 = n2; // упаковка целых n1 и n2
Swap(ref o1, ref o2);
  // ошибки нет, но в n1 по-прежнему 1, а в n2 по-прежнему 2
```

Неизменность значений переменных `n1` и `n2` объясняется тем, что в ходе упаковки (см. п. 7.4) создаются *новые объекты* (размещаемые в куче), поэтому перемена их местами также никак не скажется на значениях исходных целочисленных переменных.

Таким образом, поставленная задача не может быть удовлетворительно решена без использования обобщений. С применением же обобщений она решается тривиально:

```
static void Swap<T>(ref T a, ref T b)
{
  T v = a;
  a = b;
  b = v;
}
```

Параметр `T`, указанный в угловых скобках, называется *обобщенным параметром*. При вызове обобщенного метода в качестве обобщенного параметра указывается какой-либо конкретный тип, например:

```
Swap<string>(ref s1, ref s2);
```

или

```
Swap<int>(ref n1, ref n2);
```

Однако в случае, если значение обобщенного параметра может быть определено самим компилятором по типу фактических параметров метода, его можно не указывать:

```
Swap(ref s1, ref s2);
Swap(ref n1, ref n2);
```

Эта важная особенность компилятора называется *логическим выводом типов* (type inference).

Ясно, что при логическом выводе компилятор может принимать во внимание только *объявленный* тип переменных, поэтому, например, следующий фрагмент программы откомпилирован не будет:

```
int n1 = 1;
object n2 = 2;
Swap(ref n1, ref n2); // ошибка компиляции
```

При анализе данного фрагмента компилятор выберет в качестве значения обобщенного параметра тип `object` (поскольку фактический параметр `n2` имеет *объявленный* тип `object`), однако тип параметра `n1` (`ref int`), как мы уже выяснили, нельзя привести к типу `ref object`, в результате чего возникнет ошибка компиляции.

Наряду с обобщенными вариантами метода можно описывать и его необобщенные варианты. Если компилятор должен выбирать между необобщенным методом и одним из вариантов обобщенного метода, то он всегда выбирает необобщенный метод:

```
static void Swap(ref string a, ref string b)
{
    string v = a;
    a = b;
    b = v;
    Console.WriteLine("Non-generic");
}
…
Swap(ref s1, ref s2); // будет напечатан текст "Non-generic"
```

В то же время можно явно указать, что требуется вызов именно обобщенного метода:

```
Swap<string>(ref s1, ref s2); // текст "Non-generic" напечатан не будет
```

Метод может иметь несколько обобщенных параметров; в этом случае они перечисляются в угловых скобках через запятую, например:

```
public T3 Test<T1,T2,T3>(T1 a, T2 b) { … }
```

При наличии нескольких обобщенных перегруженных методов с разным числом обобщенных параметров компилятор не оказывает какому-либо из них предпочтения при логическом выводе.

9.2. Обобщенные типы

Задача реализации универсального класса-коллекции для хранения данных любых типов, казалось бы, может быть решена и без привлечения обобщенных типов. Действительно, для хранения данных любого типа достаточно использовать коллекцию объектов типа `object`. Примером подобной коллекции является класс `Stack` из пространства имен `System.Collections`. Однако при использовании данного класса неизбежны потери в производительности, а также могут возникать ошибки, которые проявятся только на этапе выполнения программы. Рассмотрим примеры.

```
Stack s = new Stack();
s.Push(14);
int a = s.Pop(); // ошибка компиляции
```

При выполнении данного фрагмента будет выведено сообщение компилятора о том, что объект типа `object` нельзя неявно привести к типу `int`.

Исправленный вариант будет компилироваться и правильно выполняться, однако при этом будет происходить упаковка и распаковка целочисленного значения:

```
s.Push(14); // упаковка при приведении числа 14 к типу object
int a = (int)s.Pop(); // распаковка при приведении
```

```
                    // возвращаемого значения к типу int
```

Наконец, попытка занесения в стек данных ошибочного типа будет выявлена только на этапе выполнения:

```
s.Push("14"); // ошибки компиляции нет,
              // при выполнении в стек s будет помещена строка
int a = (int)s.Pop();
  // ошибка времени выполнения при попытке приведения строки к типу int
```

Все отмеченные проблемы снимаются при использовании обобщенного класса `Stack<T>` из пространства имен `System.Collections.Generic`:

```
Stack<int> s = new Stack<int>();
s.Push(14);      // упаковка не производится
int a = s.Pop(); // ошибки компиляции нет,
                 //   поскольку Pop возвращает значение типа int
s.Push("14");    // ошибка компиляции: в стек Stack<int>
                 //   можно помещать только данные типа int
```

9.3. Стандартные коллекции

В пространстве имен `System.Collections.Generic` определен ряд обобщенных классов-коллекций, большинство из которых является обобщенным вариантом соответствующего необобщенного класса-коллекции из пространства имен `System.Collections`.

В табл. 6 указаны соответствия для «старых» (необобщенных) и «новых» (обобщенных) коллекций, а также даны их краткие описания. Во всех необобщенных коллекциях элементы хранятся как данные типа `object`; в обобщенных коллекциях тип данных определяется обобщенным параметром (или двумя обобщенными параметрами в случае обобщенных ассоциативных массивов).

Таблица 6

Коллекции .NET Framework

Необобщенный вариант коллекции	Обобщенный вариант коллекции	Описание
`ArrayList`	`List<T>`	Динамический массив с возможностью вставки и удаления элементов
`Stack`	`Stack<T>`	Стек
`Queue`	`Queue<T>`	Очередь
`Hashtable`	`Dictionary<TKey, TValue>`	Ассоциативный массив (набор пар «ключ-значение»)
`SortedList`	`SortedList<TKey, TValue>`	Ассоциативный массив, отсортированный по ключу, с возможностью доступа к элементам по индексу

Все методы и свойства обобщенных коллекций имеют свои аналоги в необобщенных коллекциях; отличие состоит лишь в том, что в случае обобщенных коллекций используемые в них типы учитывают тип элементов. Благодаря этому обстоятельству, в частности, отпадает необходимость в упаковке при работе с коллекциями размерных типов, что обеспечивает существенное повышение производительности.

При организации перебора элементов обобщенных классов `Dictionary` и `SortedList` используется еще один обобщенный тип – структура `KeyValuePair<TKey, TValue>`, предоставляющая доступ к компонентам элемента данных классов: его *ключу* `Key` (типа `TKey`) и его *значению* `Value` (типа `TValue`). Необобщенным аналогом данной структуры является структура `DictionaryEntry`, оба свойства которой – `Key` и `Value` – имеют тип `object`.

Учитывая сложность класса `Dictionary`, приведем пример его использования для решения следующей задачи: подсчитать количество появлений одинаковых значений в целочисленном массиве `data` и вывести результат для каждого значения в виде «значение – число его появлений».

```
int[] data = new int[] { 3, 6, 6, 2, 3, 4, 6, 7, 2,
                         3, 4, 3, 2, 5, 4, 3, 2, 45 };
Dictionary<int, int> dict = new Dictionary<int, int>();
foreach (int d in data)
  if (dict.ContainsKey(d))
    dict[d]++;
  else
    dict[d] = 1;
foreach (KeyValuePair<int, int> kv in dict)
  Console.Write(kv.Key + "-" + kv.Value + " ");
```

При выполнении данного фрагмента на экран будут выведены следующие результаты:

```
3-5 6-3 2-4 4-3 7-1 5-1 45-1
```

Доступ по ключу к элементу ассоциативного массива *для его чтения* (в нашей программе это `dict[d]++`) возможен только в случае, если элемент с этим ключом уже существует (в противном случае возбуждается исключение `KeyNotFoundException`). Однако доступ на запись разрешен и для отсутствующих ключей (в нашей программе это `dict[d] = 1`); при этом в ассоциативном массиве создается элемент с указанным ключом.

Опишем еще одну стандартную обобщенную коллекцию, которая, в отличие от ранее рассмотренных, не имеет необобщенного аналога и появилась только в версии .NET Framework 3.5: это *обобщенное множество* `HashSet<T>` элементов типа `T`. Данная коллекция не может содержать одинаковые элементы (при попытке добавления уже имеющегося элемента не выполняется никаких действий), ее элементы не располагаются в определенном порядке, и к ним нельзя обратиться по индексу. Однако с помощью

метода `s.Contains(item)` можно быстро (без линейного перебора всей коллекции) определить, входит ли элемент `item` в коллекцию `s`. Кроме того, в данном классе определены все стандартные операции над множествами (сравнение множеств, пересечение, объединение, разность, симметрическая разность); можно также организовать перебор всех элементов множества, используя цикл `foreach`.

Еще одной стандартной обобщенной коллекцией, не имеющей необобщенного аналога, является *двусвязный список* `LinkedList<T>`. Мы не будем подробно описывать этот вид коллекции.

Во всех рассмотренных классах-коллекциях можно определить количество элементов, используя свойство `Count`, доступное только для чтения.

9.4. Реализация обобщенных типов

В качестве примера описания обобщенного класса приведем упрощенную реализацию стека:

```
public class SimpleStack<T>
{
  int top;
  T[] data = new T[100];
  public void Push(T d) { data[top++] = d; }
  public T Pop() { return data[--top]; }
  public T Peek() { return data[top - 1]; }
  public int Count { get { return top; } }
}
```

Обобщенный параметр `T`, указанный в угловых скобках после имени типа в заголовке его описания, может в дальнейшем использоваться при описании любых членов этого типа. Заметим, что методы обобщенного типа не считаются обобщенными, если они сами не содержат обобщенные параметры (указываемые в угловых скобках после их имени).

Как и в случае обобщенных методов, обобщенный тип может иметь несколько обобщенных параметров, указываемых в угловых скобках через запятую.

Обобщенными могут быть как классы (т. е. ссылочные типы), так и структуры (т. е. размерные типы). Имеется единственное исключение: не может быть обобщенных перечислений.

Обобщенный тип, в имени которого хотя бы один обобщенный параметр является «заглушкой», называется *открытым* типом. Если же все обобщенные параметры заменены на конкретные типы, то такой обобщенный тип называется *замкнутым*. Для создания экземпляра можно использовать только замкнутые обобщенные типы. В то же время и открытые, и замкнутые типы являются «полноценными» типами языка; в частности,

для любого из них можно получить информацию о типе, используя операцию typeof:

```
Type t1 = typeof(SimpleStack<>);
  // t1 содержит информацию об открытом типе SimpleStack<T>
Type t2 = typeof(SimpleStack<int>);
  // t2 содержит информацию о замкнутом типе SimpleStack<int>
Console.Write(t1.Name + " " + t1.ContainsGenericParameters);
foreach (Type t in t1.GetGenericArguments())
  Console.Write(" " + t.Name);
Console.Write("\n" + t2.Name + " " + t2.ContainsGenericParameters);
foreach (Type t in t2.GetGenericArguments())
  Console.Write(" " + t.Name);
```

При выполнении данного фрагмента на экран будет выведен следующий текст:

```
SimpleStack`1 True T
SimpleStack`1 False Int32
```

Если обобщенный тип имеет несколько обобщенных параметров, то при применении к нему операции typeof надо указывать их количество с помощью запятых:

```
class Test<T> {}
class Test<T1,T2> {}
…
Type t1 = typeof(Test<>), t2 = typeof(Test<,>);
Console.Write(t1.Name);
foreach (Type t in t1.GetGenericArguments())
  Console.Write("  " + t.Name);
Console.Write("\n" + t2.Name);
foreach (Type t in t2.GetGenericArguments())
  Console.Write("  " + t.Name);
```

При выполнении данного фрагмента на экран будет выведен следующий текст:

```
Test`1 T
Test`2 T1 T2
```

Имеется возможность указать *значение по умолчанию* для данных, имеющих тип T. Для этого надо использовать конструкцию default(T), которая означает null для всех ссылочных типов и структуру с побитово обнуленными полями для размерных типов.

При определении классов-потомков обобщенных классов можно использовать как открытые, так и замкнутые варианты класса-предка; можно также добавлять новые обобщенные параметры и использовать имя класса при замыкании его предка:

```
class Test1<T>: Test<T> {}
class Test2: Test<int> {}
```

```
class Test3<T1, T2>: Test<T1> {}
class Test4: Test<Test4> {}
```

Если в обобщенном типе определены статические поля, то их значения будут уникальными для любого замыкания этого типа (иными словами, статическое поле `f` типа `Test<int>` не будет иметь никакого отношения к статическому полю `f` типа `Test<string>`).

9.4.1. О сокращенном именовании обобщенных типов

При конструировании замкнутых обобщенных классов их имена часто оказываются длинными и сложными для восприятия. Например, для реализации графа можно использовать динамический массив `List`, *i*-й элемент которого будет, в свою очередь, коллекцией-множеством `HashSet` с индексами вершин графа, инцидентных вершине *i*: `List<HashSet<int>>`. Можно ли связать с данной коллекцией более кратное наименование, например `Graph`? Казалось бы, хорошим вариантом является порождение необобщенного класса-потомка от данного замкнутого обобщенного класса:

```
internal sealed class Graph: List<HashSet<int>> { }
```

При этом не потребуется определять ни одного нового члена, поскольку все открытые члены класса `List` будут унаследованы, а конструктор без параметров будет создан компилятором по умолчанию. Однако подобный вариант *не обеспечивает тождественности типов*. В частности, если в некотором методе будет указан параметр типа `Graph`, то передать ему объект типа `List<HashSet<int>>` будет нельзя без явного приведения типа.

Для решения указанной проблемы в языке C# имеется специальное средство: определение *псевдонимов* (синонимов) типов с помощью директивы `using`. В нашем случае достаточно добавить в начало программы следующую директиву:

```
using Graph =
  System.Collections.Generic.List<System.Collections.Generic.HashSet<int>>;
```

Обратите внимание на то, что в правой части данного определения необходимо указывать *полные* имена всех типов. Теперь в программе имя `Graph` будет считаться синонимом типа, указанного справа от знака равенства (подобно тому как имя `int` считается синонимом типа `System.Int32`).

Заметим, что описанное средство предназначено, прежде всего, для разрешения конфликтов, возникающих, если одноименные типы определены в разных пространствах имен.

9.5. Ограничения для обобщенных параметров

По умолчанию в качестве обобщенного параметра `T` можно использовать любой тип. Однако в этом случае набор действий, доступных для дан-

ных типа T, является очень ограниченным. Например, следующие обобщенные методы не будут компилироваться:

```
public static void M1<T>(ref T a)
{
  a = null; // если T – структура,
            // то переменной a нельзя присваивать null
}
public static void M2<T>(out T a)
{
  a = new T(); // тип T не обязан иметь конструктор без параметров
}
public static T? M3<T>(ref T a, bool b)
  // тип T? определен только в случае, если T является структурой
{
  T? res = null;
  if (b)
    res = default(T);
  return res;
}
public static T M4<T>(T a, T b)
{
  return a.CompareTo(b) < 0 ? a : b;
  // тип T не обязан реализовывать интерфейс IComparable
}
```

Для того чтобы данные, имеющие тип T, могли выполнять более разнообразные действия, к обобщенному параметру T необходимо применить одно или несколько *ограничений* (constraints). Имеются 6 видов ограничений:

- `where T : class` – тип T должен быть ссылочным (т. е. классом; в частности, массивом, интерфейсом или делегатом);
- `where T : struct` – тип T должен быть размерным (т. е. структурой или перечислением), за исключением Nullable-типов;
- `where T :` *имя_класса* – тип T должен либо быть указанным классом, либо порождаться от указанного класса (можно указывать любой незапечатанный класс, за исключением классов `Array`, `Delegate`, `MulticastDelegate`, `ValueType`, `Enum`);
- `where T : new()` – тип T должен иметь открытый конструктор без параметров;
- `where T :` *имя_интерфейса* – тип T должен реализовывать указанный интерфейс;
- `where T :` *имя_другого_обобщенного_параметра* – тип T должен быть совместим с указанным обобщенным параметром (т. е. либо совпа-

дать с ним, либо порождаться от указанного параметра, либо реализовывать его – в случае, если другой обобщенный параметр является интерфейсом).

С применением ограничений нам удастся обеспечить компиляцию всех приведенных выше методов:

```
public static void M1<T>(ref T a) where T : class
{
  a = null;
}
public static void M2<T>(out T a) where T : new()
{
  a = new T();
}
public static T? M3<T>(ref T a, bool b) where T : struct
{
  T? res = null;
  if (b)
    res = default(T);
  return res;
}
public static T M4<T>(T a, T b) where T : IComparable
{
  return a.CompareTo(b) < 0 ? a : b;
}
```

Ограничения делятся на основные и дополнительные. Для любого обобщенного параметра `T` может быть определено не более одного основного ограничения и произвольное число дополнительных. Основными являются первые три из перечисленных видов ограничений, дополнительными – последние три. При этом ограничение на открытый конструктор, очевидно, также не должно указываться более одного раза.

Замечание. Реализация метода `M4`, возвращающего минимальный из указанных параметров, является несимметричной: если второй параметр равен `null`, то возвращается `null`, если же первый параметр равен `null`, то возбуждается исключение. Приведем симметричный вариант метода `M4`:

```
public T M4<T>(T a, T b) where T : IComparable
{
  try
  {
    return a.CompareTo(b) < 0 ? a : b;
  }
  catch
  {
    return default(T);
```

```
  }
}
```

Методы и типы являются единственными конструкциями языка C#, которые могут иметь обобщенные параметры. Поля, а также свойства (в том числе индексаторы) и события обобщенных параметров иметь не могут, однако любые из этих членов могут использовать обобщенные параметры, указанные в охватывающих их типах.

Например, мы можем «расширить» возможности класса `SimpleStack<T>`, добавив к нему индексатор, возвращающий элемент обобщенного типа:

```
public T this[int index] { get { return data[index]; } }
```

9.6. Обобщенные делегаты

При определении делегата можно также указывать обобщенные параметры. В результате будет определен *обобщенный делегат*. Основное достоинство обобщенных делегатов – в том, что при их использовании можно обойтись без определения новых (необобщенных) делегатов. Вспомним пример с определением события (см. п. 7.9). В нем мы определили новый делегат `InfoEventHandler`, отличающийся от стандартного делегата `EventHandler` тем, что его второй параметр имеет тип `InfoEventArgs`:

```
public delegate void InfoEventHandler(object sender, InfoEventArgs e);
```

Однако этого можно и не делать, если воспользоваться еще одним стандартным делегатом, который имеет то же имя `EventHandler`, однако является обобщенным:

```
public delegate void EventHandler<T>(object sender, T e)
  where T : EventArgs;
```

В качестве его обобщенного параметра можно указывать любой тип, являющийся потомком типа `EventArgs`. В нашем случае вместо нового делегата `InfoEventHandler` мы можем использовать при описании события `Info` замкнутый обобщенный делегат `EventHandler<InfoEventArgs>`:

```
public event EventHandler<InfoEventArgs> Info;
```

Обратимся к еще одному ранее рассмотренному делегату (см. п. 7.9):

```
public delegate int Transformer(int i);
```

Этот делегат позволяет хранить только методы, преобразующие целые числа.

Если же определить обобщенный делегат `Transformer<T>`, то с его помощью можно будет работать с методами, преобразующими данные любого типа:

```
public delegate T Transformer<T>(T x);
…
Transformer<int> t1 = i => -i;
Transformer<string> t2 = s => '*' + s + '*';
```

```
Console.WriteLine(t1(1));     // -1
Console.WriteLine(t2("abc")); // *abc*
```

В версии .NET Framework 2.0 в пространстве имен `System` было определено семейство обобщенных делегатов, с помощью которых можно сконструировать замкнутый делегат для хранения практически любого метода (не использующего `ref`- и `out`-параметров). Перечислим эти делегаты:

```
public delegate TResult Func<TResult>();
public delegate TResult Func<T, TResult>(T arg);
public delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2);
public delegate TResult Func<T1, T2, T3, TResult>(T1 arg1, T2 arg2,
  T3 arg3);
public delegate TResult Func<T1, T2, T3, T4, TResult>(T1 arg1, T2 arg2,
  T3 arg3, T4 arg4);
public delegate void Action();
public delegate void Action<T>(T arg);
public delegate void Action<T1, T2>(T1 arg1, T2 arg2);
public delegate void Action<T1, T2, T3>(T1 arg1, T2 arg2, T3 arg3);
public delegate void Action<T1, T2, T3, T4>(T1 arg1, T2 arg2, T3 arg3,
  T4 arg4);
```

В версии .NET Framework 4.0 данное семейство было расширено таким образом, чтобы можно было использовать делегаты, содержащие до 16 параметров.

Любые делегаты из ранее приведенных примеров могут быть заменены на один из указанных стандартных делегатов. Например, вместо необобщенного делегата `Transformer` можно использовать делегат `Func<int, int>`, вместо делегата `Transformer<string>` можно использовать `Func<string, string>`, а вместо делегата `InfoEventHandler` – `Action<object, InfoEventArgs>`.

9.7. Обобщенные интерфейсы

Механизм обобщенных параметров типа позволяет определять эффективные и надежные *обобщенные интерфейсы*. Рассмотрим уже известный нам интерфейс `IComparable`:

```
public interface IComparable
{
  int CompareTo(object other);
}
```

То обстоятельство, что параметр описан как `object`, приводит к двум проблемам. Во-первых, компилятор не будет считать ошибкой передачу в качестве `other` объекта любого типа, однако при выполнении программы будет возбуждено исключение, если параметр `other` будет иметь тип, не приводимый к типу объекта, вызвавшего метод `CompareTo`. Во-вторых, при

выполнении данного метода для размерных типов размерный параметр other будет подвергаться упаковке.

В то же время есть очевидное средство, позволяющее повысить как надежность, так и эффективность метода CompareTo: описать параметр other, указав тип, совпадающий с типом, для которого реализуется интерфейс. Ясно, что обеспечить это можно, сделав соответствующей интерфейс обобщенным:

```
public interface IComparable<T>
{
  int CompareTo(T other);
}
```

Для целей обратной совместимости имеет смысл реализовать для типа *оба* указанных интерфейса: и необобщенный, и обобщенный. При этом нет необходимости в явной реализации какого-либо из методов данных интерфейсов, поскольку методы не будут конфликтовать: они просто будут считаться перегруженными вариантами одного метода CompareTo.

После того как мы узнали о наличии обобщенного интерфейса IComparable<T>, имеет смысл вернуться к рассмотренному в п. 9.5 обобщенному методу M4 с ограничением и изменить его следующим образом (приведем первый вариант данного метода; усовершенствованный вариант изменяется аналогично):

```
public static T M4a<T>(T a, T b) where T : IComparable<T>
{
  return a.CompareTo(b) < 0 ? a : b;
}
```

Это повысит эффективность метода при его использовании для размерных типов, поскольку в данном случае параметр b метода CompareTo не будет подвергаться упаковке.

Для того чтобы продемонстрировать различия в работе методов M4 и M4a, можно создать демонстрационный класс, реализующий оба интерфейса, и поместить в методы этих интерфейсов отладочную печать:

```
class CompDemo : IComparable, IComparable<CompDemo>
{
  public int CompareTo(object other)
  {
    Console.WriteLine("object");
    return 0;
  }
  public int CompareTo(CompDemo other)
  {
    Console.WriteLine("CompDemo");
    return 0;
  }
```

```
}
```

При вызове метода M4 с параметрами типа CompDemo будет напечатан текст «object», а при вызове модифицированного метода M4a – текст «CompDemo».

Ранее, в п. 8.6, мы описали еще два необобщенных интерфейса, связанных с операцией сравнения: IComparer и IEqualityComparer. Для них также имеются обобщенные варианты с очевидной реализацией:

```
public interface IComparer<T>
{
  int Compare(T a, T b);
}
public interface IEqualityComparer<T>
{
  bool Equals(T a, T b);
  int GetHashCode(T obj);
}
```

Упомянем также обобщенный интерфейс IEquitable<T>, который не имеет необобщенного аналога. Он содержит метод Equals(T value) типа bool, обеспечивающий более эффективную реализацию сравнения на равенство, чем одноименный экземплярный метод класса object (см. п. 7.1).

Имеется еще один рассмотренный нами интерфейс, в котором использовался тип object, – это интерфейс IEnumerator (см. п. 8.4):

```
public interface IEnumerator
{
  bool MoveNext();
  object Current { get; }
  void Reset();
}
```

Ясно, что было бы гораздо удобнее, если бы свойство Current возвращало значение того же типа, что и перебираемая перечислителем последовательность. Поэтому, наряду с необобщенным интерфейсом IEnumerator, реализован также его обобщенный вариант IEnumerator<T> (он определен в пространстве имен System.Collections.Generic):

```
public interface IEnumerator<T> : IDisposable, IEnumerator
{
  T Current { get; }
}
```

В этом примере мы впервые познакомились с *порождением* («наследованием») одного интерфейса от других. Порожденный интерфейс-потомок «наследует» все методы своих интерфейсов-предков; это означает, что тот класс, который реализует подобный интерфейс-потомок, должен реализовать не только собственные методы потомка, но и все методы предков. Кроме того, подключение к классу интерфейса-потомка будет оз-

начать и автоматическое подключение к нему всех интерфейсов-предков (поэтому перечислять их в списке реализуемых интерфейсов не требуется, хотя и не запрещается).

Проиллюстрируем особенности, связанные с порождением интерфейсов, на примере обобщенного интерфейса IEnumerator<T>. Поскольку он порожден от интерфейсов IDisposable и IEnumerator, в любом классе, реализующем интерфейс IEnumerator<T>, следует реализовать не только его свойство Current (типа T), но и свойство Current типа object, методы MoveNext и Reset, а также метод Dispose. В результате этот класс будет автоматически реализовывать не только интерфейс IEnumerator<T>, но и интерфейсы IEnumerator и IDisposable. Отметим, что все требуемые методы не удастся описать неявно, поскольку два свойства Current будут конфликтовать. В подобной ситуации разумнее всего явно реализовать «менее удобное» свойство Current, возвращающее значение типа object. Разумеется, допустимо описать явно и другие члены интерфейсов, например метод IDisposable.Dispose, однако в этом нет необходимости:

```
class EnumDemo: IEnumerator<EnumDemo>
  public EnumDemo Current
  {
    get { … }
  }
  object IEnumerator.Current
  {
      get { return Current; }
      // возвращается значение неявно реализованного свойства
  }
  public void Dispose() { … }
  public bool MoveNext() { … }
  public void Reset() { … }
}
```

Еще раз отметим, что для объекта eDemo типа EnumDemo операции eDemo is IEnumerator, eDemo is IEnumerator<EnumDemo> и eDemo is IDisposable будут возвращать true.

Подобно тому как интерфейс IEnumerable предоставляет классу необобщенный перечислитель (посредством вызова метода GetEnumerator, возвращающего результат типа IEnumerator), обобщенный интерфейс IEnumerable<T> позволяет получить *обобщенный перечислитель* типа IEnumerator<T>:

```
public interface IEnumerable<T> : IEnumerable
{
  IEnumerator<T> GetEnumerator();
}
```

Этот интерфейс, как и все базовые обобщенные интерфейсы, связанные с коллекциями, описан в пространстве имен System.Collections.Generic. Обратите внимание на то, что данный интерфейс порождается от необобщенного интерфейса IEnumerable, поэтому в классе, реализующем интерфейс IEnumerable<T>, необходимо описать два варианта метода GetEnumerator (при этом, разумеется, один из них должен быть явно реализован для соответствующего интерфейса).

Важнейшая особенность интерфейса IEnumerable<T> (по сравнению с его необобщенным аналогом) состоит в обеспечении большей надежности программного кода. Рассмотрим пример, в котором для автоматической генерации перечислителя, связанного с данным классом, будем использовать оператор yield (см. п. 8.5):

```
class EnumDemo : IEnumerable
{
  public IEnumerator GetEnumerator()
  {
    yield return 1;
    yield return 2;
  }
}
```

Протестируем работу перечислителя с помощью цикла foreach:

```
EnumDemo en = new EnumDemo();
foreach (int a in en)
  Console.WriteLine(a); // 1 2
```

В результате на экран будут выведены два целых числа: 1 и 2. Однако, поскольку перечислитель IEnumerator не «знает» о том, что перечисляемые элементы должны иметь целый тип, следующий фрагмент также успешно откомпилируется, хотя при его выполнении будет возбуждено исключение InvalidCastException:

```
EnumDemo en = new EnumDemo1();
foreach (string a in en)
  Console.WriteLine(a); // ошибка времени выполнения
```

Изменим наш класс, реализовав в нем обобщенный интерфейс IEnumerable<T>:

```
class EnumDemo : IEnumerable<int>
{
  public IEnumerator<int> GetEnumerator()
  {
    yield return 1;
    yield return 2;
  }
  IEnumerator IEnumerable.GetEnumerator()
  {
```

```
    return GetEnumerator();
  }
}
```

Теперь ошибочная попытка организовать перебор элементов типа `string` будет выявлена уже на этапе компиляции, при этом будет выведено сообщение о том, что нельзя преобразовать тип `int` в `string`.

Заметим, что подобный контроль типов на этапе компиляции будет выполнен и в том случае, если оба метода `GetEnumerator` будут явно реализованы для соответствующих интерфейсов:

```
class EnumDemo : IEnumerable<int>
{
  IEnumerator<int> IEnumerable<int>.GetEnumerator()
  {
    yield return 1;
    yield return 2;
  }
  IEnumerator IEnumerable.GetEnumerator()
  {
    return (this as IEnumerable<int>).GetEnumerator();
  }
}
```

В данном случае для вызова метода `GetEnumerator` требуется выполнить приведение объекта к типу интерфейса.

Замечание. Однако, если в классе имеется *экземплярный* метод `GetEnumerable`, пусть даже и не связанный с каким-либо интерфейсом, то именно этот метод будет использован при выполнении цикла `foreach`. Поэтому, например, при компиляции следующего фрагмента контроль типов в цикле `foreach` проводиться не будет, несмотря на то что класс реализует обобщенный интерфейс:

```
class EnumDemo : IEnumerable<int>
{
  IEnumerator<int> IEnumerable<int>.GetEnumerator()
  {
    yield return 1;
    yield return 2;
  }
  IEnumerator IEnumerable.GetEnumerator()
  {
    return (this as IEnumerable<int>).GetEnumerator();
  }
  public IEnumerator GetEnumerator()
  {
    return (this as IEnumerable<int>).GetEnumerator();
```

```
  }
}
…
EnumDemo en = new EnumDemo1();
foreach (string a in en)
  Console.WriteLine(a); // исключение InvalidCastException
```

Поэтому в случае, если класс должен вести себя как перечислимая коллекция строго типизированных элементов, желательно «открыть» в нем интерфейс `IEnumerable<T>`, т. е. реализовать его неявно, и «закрыть» интерфейс `IEnumerable` путем его явной реализации. Именно таким образом ведут себя все обобщенные коллекции из пространства имен `System.Collections.Generic`.

Впрочем, иногда эта рекомендация не соблюдается. В частности, в классе `Array` открыт *необобщенный* интерфейс `IEnumerable` и «закрыт» (явно реализован) интерфейс `IEnumerable<T>` (это сделано для обеспечения совместимости с ранее разработанным программным кодом). Поэтому приведенный ниже фрагмент не будет откомпилирован:

```
int[] a = new int[10];
IEnumerator<int> en = a.GetEnumerator(); // ошибка компиляции
```

В данном случае необходимо выполнить явное приведение массива к типу соответствующего обобщенного интерфейса:

```
IEnumerator<int> en = (a as IEnumerable<int>).GetEnumerator();
```

Впрочем, подобное поведение не вызывает дополнительных проблем и не влияет на надежность кода, так как обычно метод `GetEnumerator` используется самим компилятором при реализации цикла `foreach`, а для массивов проверка типов в цикле `foreach` на этапе компиляции была реализована еще в первой версии языка C#.

Следует также учитывать, что, в отличие от класса `Array`, все «старые» необобщенные классы-коллекции из пространства имен `System.Collections` с появлением возможности использования обобщенных интерфейсов в версии C# 2.0 *не были модифицированы*, и по-прежнему реализуют только *необобщенные* интерфейсы (в частности, интерфейс `IEnumerable`).

9.8. Обобщенные интерфейсы, связанные с коллекциями

Рассмотренные выше интерфейсы `IEnumerable` и `IEnumerable<T>` предназначены для организации *перебора* элементов коллекций; в частности, для обеспечения *унификации* этого перебора посредством цикла `foreach`. Для унификации других методов и свойств, связанных с коллекциями, в стандартной библиотеке .NET Framework имеются дополнительные интерфейсы, как необобщенные, так и обобщенные. Основными из них являются `ICollection`, `ICollection<T>`, `IList`, `IList<T>`, `IDictionary`, `IDictionary<TKey,`

TValue>.

Надо отметить, что необобщенные и обобщенные варианты интерфейсов ICollection и IList различаются между собой гораздо существеннее, чем ранее рассмотренные необобщенные и обобщенные варианты интерфейсов IComparable, IEnumerator и IEnumerable. По-видимому, это объясняется недостаточной продуманностью более «старых», необобщенных интерфейсов, из-за которой разработчики их обобщенных вариантов были вынуждены внести в них существенные модификации. Часть функциональности, связанной с необобщенным интерфейсом IList, при реализации обобщенных интерфейсов была перенесена в интерфейс ICollection<T>, после чего эта часть была просто унаследована интерфейсом IList<T>. В результате интерфейс ICollection<T> оказался «богаче», чем интерфейс ICollection, однако при этом интерфейс IList<T>, с учетом членов, унаследованных от ICollection<T>, получил практически ту же функциональность, что и интерфейс IList. При этом «расширение» интерфейса ICollection<T> не отразилось на большинстве стандартных обобщенных классов-коллекций, поскольку они либо, подобно своим необобщенным аналогам, реализуют лишь необобщенный интерфейс ICollection (таковы классы Stack<T> и Queue<T>), либо реализуют более полный интерфейс IList<T> (класс List<T>) или IDictionary<TKey, TValue> (класс Dictionary<TKey, TValue>). Из всех рассмотренных нами стандартных обобщенных коллекций только класс HashSet<T>, появившийся в .NET Framework 3.5, служит некоторым «оправданием» факта добавления в интерфейс ICollection<T> новых членов, поскольку этот класс реализует интерфейс ICollection<T> и *не реализует* интерфейс IList<T>.

Вместо того чтобы приводить полные описания данных интерфейсов, перечислим их наиболее важные члены.

ICollection (представители – Stack, Queue, Stack<T>, Queue<T>):

```
int Count { get; }
void CopyTo(Array array, int index);
```

ICollection<T> (представитель – HashSet<T>):

```
int Count { get; }
void CopyTo(T[] array, int index);
void Clear();
bool Contains(T item);
void Add(T item);
bool Remove(T item);
```

Заметим, что в классах Stack<T>, Queue<T> реализован более безопасный с точки зрения типов метод void CopyTo(T[] array, int index), а реализация одноименного метода интерфейса ICollection скрыта. Кроме того, методы Clear и Contains также реализованы в классах Stack, Queue, Stack<T>, Queue<T> (хотя в интерфейсе ICollection они отсутствуют).

IList (представитель – ArrayList):

```
object this[int index] { get; set; }
int IndexOf(object value);
void Insert(int index,
  object value);
void RemoveAt(int index);
void Clear();
bool Contains(object value);
int Add(object value);
bool Remove(object value);
```

IList<T> (представитель – List<T>):

```
T this[int index] { get; set; }
int IndexOf(T item);
void Insert(int index, T item);

void RemoveAt(int index);
```

Обратите внимание на то, что интерфейс IList<T>, как и IList, включает методы Clear, Contains, Add и Remove, поскольку интерфейс IList<T> порождается от интерфейса ICollection<T> (при этом метод Add, в отличие от одноименного метода интерфейса IList, имеет возвращаемый тип void). Так как интерфейс IList порождается от интерфейса ICollection, оба интерфейса – и IList, и IList<T> – включают свойство Count и метод CopyTo.

IDictionary (представитель – Hashtable):

```
object this[object key] { get; set; }
ICollection Keys { get; }
ICollection Values { get; }
void Add(object key, object value);
void Remove(object key);
bool Contains(object key);
void Clear();
```

IDictionary<TKey, TValue> (представитель – Dictionary<TKey, TValue>):

```
TValue this[TKey key] { get; set; }
ICollection<TKey> Keys { get; }
ICollection<TValue> Values { get; }
void Add(TKey key, TValue value);
bool Remove(TKey key);
bool ContainsKey(TKey key);

bool TryGetValue(TKey key,
  out TValue value);
```

Интерфейс IDictionary порождается от интерфейса ICollection, а интерфейс IDictionary<TKey, TValue> – от интерфейса ICollection<KeyValuePair<TKey, TValue>>. Поэтому и IDictionary, и IDictionary<TKey, TValue> содержат также свойство Count и метод CopyTo (а интерфейс IDictionary<TKey, TValue> – метод Clear). Интересно отметить, что в классе Hashtable метод CopyTo реализован в виде экземплярного метода класса и поэтому может быть вызван непосредственно для объекта Hashtable, тогда как в классе Dictionary<TKey, TValue> метод CopyTo реализован лишь как *метод интерфейса*, поэтому для его вызова необходимо привести объект Dictionary<TKey, TValue> к типу ICollection<KeyValuePair<TKey, TValue>>. Метод CopyTo класса Hashtable возвращает массив объектов типа DictionaryEntry, одноименный метод класса Dictionary<TKey, TValue> (вызываемый через интерфейс) возвращает массив объектов типа KeyValuePair<TKey, TValue>.

Часть 3
Технология LINQ

Глава 10. Основы технологии LINQ

Как нам уже известно, для того чтобы можно было использовать цикл `foreach`, достаточно, чтобы перебираемый набор данных реализовывал интерфейс `IEnumerable` или `IEnumerable<T>`. Таким образом, наличие у класса, реализующего коллекцию, одного из указанных интерфейсов позволяет организовать единообразный *перебор* элементов этой коллекции. Однако действия над коллекциями не ограничиваются перебором их элементов. Обычно сам перебор требуется для того, чтобы выполнить над коллекцией некоторое действие, результатом которого будет либо новая, преобразованная коллекция, либо некоторое скалярное значение. Разумеется, имея в своем распоряжении цикл `foreach`, нетрудно получить требуемый результат, обрабатывая каждый элемент исходной коллекции нужным образом. Однако было бы еще удобнее воспользоваться некоторым методом, выполняющим требуемую обработку коллекции и «скрывающим» детали этой обработки, подобно тому как цикл `foreach` «скрывает» детали функционирования перечислителя. Подобным образом ведут себя *запросы* – операторы языка SQL, обеспечивающие обработку наборов записей, полученных из одной или нескольких связанных между собой таблиц базы данных.

Технология LINQ (**L**anguage **In**tegrated **Q**uery – «запрос, интегрированный в язык») как раз и обеспечивает «встраивание» в язык программирования возможности использования запросов, подобных запросам SQL. «LINQ – это технология Microsoft, предназначенная для обеспечения механизма поддержки уровня языка для опроса данных всех типов» [6, 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» [3, с. 315].

Программный интерфейс LINQ (LINQ API) состоит из компонентов, каждый из которых связан с определенной категорией обрабатываемых наборов данных. В версии .NET Framework 3.5 были представлены следующие компоненты LINQ API:

- *LINQ to Objects* – базовый интерфейс для стандартных запросов к локальным коллекциям; основан на методах класса `Enumerable` из пространства имен `System.Linq` (включает около 40 видов запросов);
- *LINQ to XML* – интерфейс, предназначенный для обработки XML-документов; 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 (X-DOM), реализованную в виде иерархии классов из пространства имен `System.Xml.Linq`;
- *LINQ to SQL* и *LINQ to Entities* – интерфейсы, предназначенные для взаимодействия с удаленными базами данных в качестве источников наборов данных; основаны на методах класса `Queryable` из пространства имен `System.Linq`.

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.ParallelEnumerable` (около 10 методов) и обеспечивает обработку локальных коллекций типа `ParallelQuery<T>`. 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 [4] интерфейс PLINQ описывается в главе 23 «Параллельное программирование», а не в главах 8–10, специально посвященных технологии LINQ и ее интерфейсам.

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 кроссплатформенное представление любых структур данных. 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 настоящей книге, наряду с интерфейсом LINQ to Objects, подробно рассматривается и интерфейс LINQ to XML.

Для возможности применения к коллекции запросов LINQ необходимо, чтобы коллекция реализовывала *обобщенный* интерфейс `IEnumerable<T>`. «Старые» коллекции из пространства имен `System.Collections` (например, `ArrayList`) не реализуют указанный интерфейс, однако с помощью вспомогательных *запросов импортирования* `OfType` и `Cast` подобные коллекции также можно преобразовать к типу `IEnumerable<T>`, допускающему применение запросов LINQ. В дальнейшем все классы, реализующие интерфейс `IEnumerable<T>`, будем называть *последовательностями*.

10.1. Технология LINQ и новые возможности языка C# 3.0

Большинство нововведений языка C# версии 3.0 связано именно с технологией LINQ и призвано максимально упростить код, связанный с запросами LINQ. Прежде всего, следует отметить *лямбда-выражения*, предназначенные для замены анонимных методов (см. п. 7.10).

Приведем пример. Пусть в программе дан набор фамилий, из которого надо отобрать фамилии, длина которых превосходит 6 символов. В качестве исходной последовательности можно использовать обычный массив, например:

```
string[] data = { "Владимиров", "Петров", "Сидоров", "Васильев", "Яшин" };
```

Для отбора требуемых элементов в классе `Enumerable` предусмотрен статический метод `Where` с двумя параметрами: исходной последовательностью и делегатом, играющим роль *предиката*, отбирающего элементы для помещения в результирующую последовательность:

```
IEnumerable<string> res = Enumerable.Where(data,
  delegate(string s){ return s.Length > 6; });
```

С использованием лямбда-выражений приведенный фрагмент будет существенно более кратким:

```
IEnumerable<string> res = Enumerable.Where(data, s => s.Length > 6);
```

Заметим, что в методах класса `Enumerable` для описания любых параметров-делегатов используются *обобщенные делегаты* из уже известного нам семейства `Func` (см. п. 9.6).

Представить запрос в еще более кратком виде можно с помощью другого нововведения C# 3.0 – ключевого слова `var`, предназначенного для автоматического определения типа элемента по типу его инициализирующего выражения (см. п. 7.8):

```
var res = Enumerable.Where(data, s => s.Length > 6);
```

Для вывода полученной последовательности проще всего воспользоваться циклом `foreach`:

```
foreach (string s in res)
  Console.WriteLine(s);
```

При обработке набора, указанного выше в качестве образца исходных данных, будет выведено:

```
Владимиров
Сидоров
Васильев
```

Для того чтобы по достоинству оценить еще одно нововведение C# 3.0, предположим, что нам требуется дополнительно отсортировать полученную последовательность по алфавиту (в лексикографическом порядке). Для указанного вида запроса в классе `Enumerable` предусмотрен статический метод `OrderBy`, также имеющий два параметра: входную последовательность и делегат – *селектор ключа*, возвращающий ключ, по которому должна сортироваться выходная последовательность. В нашем случае в качестве ключей надо использовать сами элементы-строки. Любой метод, выполняющий запрос, *оставляет исходную последовательность неизменной* и возвращает полученную последовательность в качестве результата своей работы. Для отбора нужных элементов и их последующей сортировки можно использовать два оператора:

```
var res = Enumerable.Where(data, s => s.Length > 6);
res = Enumerable.OrderBy(res, s => s);
```

В результате элементы полученной последовательности будут отсортированы по алфавиту:

```
Васильев
Владимиров
Сидоров
```

Можно также использовать один оператор с вложенными запросами:

```
var res = Enumerable.OrderBy(Enumerable.Where(data, s => s.Length > 6),
  s => s);
```

Однако данный вариант не обладает достаточной наглядностью, во-первых, из-за многократно используемого имени `Enumerable` и, во-вторых, из-за того, что порядок *записи* запросов не совпадает с порядком их *вызова* (вначале вызывается запрос `Where`, записанный вторым, а затем – запрос `OrderBy`, указанный первым).

Поскольку использование последовательных вызовов запросов, преобразующих исходную последовательность, является стандартной практикой, в версию 3.0 языка C# была добавлена новая возможность, позволившая придать записи подобных вызовов особую наглядность. Эта возможность – *методы расширения* (extensions, см. п. 7.11). Все методы класса `Enumerable` являются статическими методами расширения, поэтому их можно вызывать, указывая в качестве вызывающего класса саму исходную последовательность:

```
var res = data.Where(s => s.Length > 6);
res = res.OrderBy(s => s);
```

При таком способе записи последовательный вызов запросов принимает вид *цепочки* запросов (каждый последующий запрос применяется к результату предыдущего запроса):

```
var res = data.Where(s => s.Length > 6).OrderBy(s => s);
```

Ясно, что все элементы цепочки (кроме, быть может, последнего) должны представлять запросы, возвращающие последовательности. Последний запрос цепочки не обязан удовлетворять указанному требованию. В качестве примера приведем цепочку из двух запросов, первый из которых отбирает фамилии длиной более 6 символов, а второй возвращает *количество элементов* полученной последовательности:

```
Console.WriteLine(data.Where(s => s.Length > 6).Count()); // 3
```

Важную роль в применении технологии LINQ играют также *анонимные типы* – еще одно нововведение C# 3.0 (см. п. 7.8). Они обычно используются при преобразовании исходной последовательности в последовательность элементов другого типа (так называемое *проецирование* последовательности). Основным методом проецирования является метод `Select`. Усложним наш пример, предположив, что в исходном наборе строк указываются не только фамилии, но и имена, причем имя следует перед фамилией и отделяется от нее единственным пробелом:

```
string[] src = { "Иван Владимиров", "Сергей Петров", "Виктор Сидоров",
  "Анатолий Васильев", "Лев Яшин" };
```

Для дальнейших преобразований исходной последовательности может оказаться удобным предварительно выполнить ее проецирование в последовательность элементов с двумя полями: `FirstName` (имя) и `LastName` (фамилия):

```
var res = src.Select(s => { string[] ss = s.Split();
    return new { FirstName = ss[0], LastName = ss[1] }; })
  .Where(s => s.LastName.Length > 6)
  .OrderBy(s => s.LastName);
foreach (var s in res)
  Console.WriteLine(s);
```

В результате выполнения данного фрагмента на экране будет выведен следующий текст:

```
{ FirstName = Анатолий, LastName = Васильев }
{ FirstName = Иван, LastName = Владимиров }
{ FirstName = Виктор, LastName = Сидоров }
```

Следует заметить, что в данном случае использование ключевого слова `var` является обязательным как при описании возвращаемой последовательности `res`, так и при описании параметра `s` цикла `foreach`. Действительно, `res` имеет тип `IEnumerable<T>`, где `T` – *анонимный* тип, возвращаемый лямбда-выражением метода `Select`, и этот же тип `T` имеет параметр цикла `s`. Поскольку данный тип не имеет имени, единственная возможность описания соответствующих объектов – использование слова `var`.

Обратите внимание также на то, что лямбда-выражение метода `Select` содержит не возвращаемое значение (в виде выражения), а *операторный блок*. Для сложных цепочек запросов целесообразно отображать их на нескольких строках, начиная каждую строку с вызова очередного запроса, предваренного точкой. Именно такой способ мы использовали в приведенном выше примере.

10.2. Особенности технологии LINQ

Запросы LINQ *никогда не изменяют входную последовательность*: если запрос предназначен для преобразования входной последовательности, то преобразованная последовательность возвращается как результат выполнения запроса. Такое поведение соответствует парадигме *функционального программирования*, на которой основана технология LINQ.

Запросы LINQ стараются сохранить исходный порядок элементов при преобразовании входной последовательности в выходную. В частности, порядок элементов сохраняется при выполнении запросов `Where` и `Select`; более того, в лямбда-выражениях для данных запросов предусмотрена возможность использования *индекса* элемента входной последовательности. Во всех случаях, когда в результате выполнения запроса отбрасываются повторяющиеся элементы (запросы `Distinct`, `Union`, `Intersect` и `Except`), в результирующей последовательности остается *первое вхождение* каждого повторяющегося элемента. К числу немногих запросов, явным образом изменяющих порядок следования элементов, относится запрос инвертирования `Reverse` и все виды запросов сортировки (`OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending`).

Важной особенностью запросов, возвращающих последовательность, является их *отложенный* (или «*ленивый*») характер: все подобные запросы выполняются не в момент конструирования (и присваивания некоторому объекту типу `IEnumerable<T>`), а при *переборе* элементов полученной после-

довательности (обычно этот перебор реализуется посредством цикла foreach). Это, в частности, означает, что если между конструированием отложенного запроса и перебором выходной последовательности будут внесены изменения во входную последовательность, то эти изменения будут учтены при переборе результата. Далее, это означает, что отложенный запрос будет *выполнен повторно*, если будет организован повторный перебор результирующей последовательности. Наконец, если при конструировании запроса в его лямбда-выражениях используются *внешние переменные*, то будет приниматься во внимание значение этих переменных не в момент *конструирования* запроса, а в момент *перебора* выходной последовательности. Последнее обстоятельство может приводить к странным на первый взгляд результатам. Приведем пример:

```
int[] src = { 12, 14, 60, 32, 48, 70 };
int k = 0;
var res = src.Where(n => n % 10 != k);
k = 2;
res = res.Where(n => n % 10 != k);
foreach (var e in res)
  Console.WriteLine(e); // 14 60 48 70
```

Казалось бы, при выполнении сконструированного запроса из исходной последовательности будут удалены все числа, оканчивающиеся на 0 и 2. Однако в результате мы получим набор чисел 14, 60, 48, 70. Это объясняется тем обстоятельством, что в момент *выполнения* запроса (который наступит при начале цикла foreach) значение переменной k будет равно 2, и это значение будет использовано во *всех* лямбда-выражениях, входящих в сконструированный запрос. Фактически, несмотря на то что запрос строился «в несколько этапов», в результате будет выполнен запрос, который выглядит следующим образом:

```
src.Where(n => n % 10 != k).Where(n => n % 10 != k)
```

Для получения ожидаемого результата (14, 48) можно, например, использовать во втором запросе *другую внешнюю переменную* со значением 2, оставив значение переменной k равным 0.

Сконструированный запрос выполняется немедленно, если его результатом является скалярное значение или коллекция определенного типа (например, массив – см. п. 10.3.8).

10.3. Обзор запросов LINQ to Objects

Благодаря технологии IntelliSense быстрое ознакомление с запросами LINQ не представляет труда: достаточно набрать после имени последовательности точку и выбрать из появившегося списка требуемый метод расширения (после имен методов расширения в списке указываются угловые скобки, например: Aggregate <>). После выбора требуемого метода расши-

рения и ввода скобки «(» во всплывающей подсказке будет выведен список параметров этого метода или варианты списка параметров, если метод является перегруженным.

Тем не менее, мы дадим краткий обзор запросов LINQ to Object, разделив запросы на 12 групп и описав их в 12 подпунктах настоящего пункта (п. 10.3.1–10.3.12). Кроме того, наиболее сложные виды запросов будут проиллюстрированы примерами.

В приведенных ниже описаниях используются следующие соглашения:

- в каждом подпункте вначале указывается тип объекта, к которому должны применяться запросы рассматриваемой группы (почти всегда этим типом будет `IEnumerable<TSource>`; исключение составляют группы 2, 5, 7 и 12), а затем (после символа «→») – тип возвращаемого значения для запросов этой группы (в некоторых случаях, если запросы данной группы могут возвращать значения различных типов, тип возвращаемого значения, предваряемый символом «→», указывается после описания каждого запроса – см. группы 6, 8, 10, 12); все указанные типы выделяются полужирным шрифтом.
- элементы описания запроса, которые могут быть опущены, заключаются, как обычно, в квадратные скобки **[ ]**;
- дополнительные параметры типа `IComparer<T>` и `IEqualityComparer<T>`, присутствующие в реализации некоторых перегруженных методов (см. группы 2, 5, 6, 8, 11), но используемые достаточно редко, не указываются в списке параметров; вместо этого о возможности использования соответствующих перегруженных вариантов сообщается при описании особенностей данных методов;
- если параметр запроса представляет собой делегат (заметим, что это всегда обобщенный делегат из семейства `Func`), то указывается не имя этого делегата, снабженное описателем типа, а соответствующее *лямбда-выражение*, вместе с типами входных параметров и типом возвращаемого значения (например, вместо параметра-предиката `Func<TSource, [int,] bool> predicate`, входящего в запросы группы 1, указывается лямбда-выражение `(TSource[, int]) => bool`).

10.3.1. Фильтрация, инвертирование и преобразование пустой последовательности

```
                              IEnumerable<TSource> → IEnumerable<TSource>
Where((TSource[, int]) => bool)
TakeWhile((TSource[, int]) => bool)
SkipWhile((TSource[, int]) => bool)
Take(int count)
Skip(int count)
```

```
Distinct()
Reverse()
DefaultIfEmpty([TSource defaultValue])
```

Метод `Where` возвращает те элементы входной последовательности (в том же порядке), для которых указанный предикат возвращает значение `true`; метод `TakeWhile` заносит в выходную последовательность элементы входной последовательности, пока указанный предикат возвращает значение `true`; метод `SkipWhile` пропускает начальные элементы входной последовательности, пока предикат возвращает значение `true`, после чего заносит в выходную последовательность все оставшиеся элементы. Во всех трех методах предикат может содержать дополнительный параметр – индекс анализируемого элемента.

Методы `Take` и `Skip`, подобно методам `TakeWhile` и `SkipWhile`, возвращают начальную или, соответственно, конечную часть исходной последовательности, однако для этих методов явно указывается число начальных элементов, которые надо возвратить (метод `Take`) или пропустить (метод `Skip`).

Метод `Distinct` возвращает последовательность без повторяющихся элементов (в последовательности оставляются только первые вхождения повторяющихся элементов).

Метод `Reverse` возвращает последовательность, в которой порядок следования элементов изменен на обратный.

Метод `DefaultIfEmpty` возвращает непустую входную последовательность в неизменном виде, а в случае пустой входной последовательности возвращает последовательность, содержащую *единственный* элемент, значение которого равно `defaultValue`, если этот параметр указан, или `default(TSource)`, если метод вызван без параметра (напомним, что `default(TSource)` означает `null` для ссылочных типов и структуру с побитово обнуленными полями для размерных типов). Пример использования метода `DefaultIfEmpty` будет приведен далее (см. замечание 1 в п. 10.3.5).

10.3.2. Упорядочивание

```
                    IEnumerable<TSource> → IOrderedEnumerable<TSource>
OrderBy(TSource => TKey)
OrderByDescending(TSource => TKey)
ThenBy(TSource => TKey)
ThenByDescending(TSource => TKey)
```

Методы возвращают элементы исходной последовательности, отсортированные по указанному *ключу*. Ключ определяется лямбда-выражением и должен иметь тип, реализующий интерфейс `IComparable`. Если тип ключа не реализует указанный интерфейс (или если при сортировке надо использовать способ упорядочивания, отличный от стандартного), можно исполь-

зовать перегруженный вариант любого из приведенных методов, содержащий дополнительный второй параметр comparer типа IComparer<TKey>.

Методы, содержащие в своем имени слово Descending, обеспечивают сортировку по убыванию.

Все рассматриваемые методы выполняют *устойчивую* сортировку; это означает, что исходный порядок следования элементов с *одинаковыми* ключами в результате сортировки не изменяется*.

Методы ThenBy и ThenByDescending используются, если последовательность требуется отсортировать по *набору ключей*; эти методы переупорядочивают (в соответствии со своим ключом) только те элементы последовательности, у которых были одинаковые ключи на предыдущем этапе сортировки. Поскольку методы ThenBy и ThenByDescending могут вызываться только для уже отсортированных последовательностей (типа IOrderedEnumerable<TSource>), первым методом в цепочке сортирующих методов обязательно должен быть либо метод OrderBy, либо OrderByDescending.

Тип IOrderedEnumerable<T> может неявно приводиться к типу IEnumerable<T>, поэтому к отсортированной последовательности можно в дальнейшем применять любые другие операции запросов, однако при этом преобразованные последовательности уже будут иметь «неотсортированный» тип IEnumerable<T>. Заметим, что, как правило, сортировка выполняется на завершающем этапе формирования выходной последовательности.

10.3.3. Сцепление и теоретико-множественные операции

```
                              IEnumerable<TSource> → IEnumerable<TSource>
Concat(IEnumerable<TSource> second)
Union(IEnumerable<TSource> second)
                              IEnumerable<TSource> → IEnumerable<TSource>
Intersect(IEnumerable<TSource> second)
Except(IEnumerable<TSource> second)
```

Метод Concat возвращает последовательность, содержащую все элементы первой входной последовательности (т. е. последовательности, для которой вызван данный метод), после которых следуют все элементы второй последовательности (указанной в качестве параметра second).

Остальные методы реализуют теоретико-множественные операции (объединение, пересечение и разность) для двух исходных последовательностей. Последовательность, полученная в результате выполнения любого из этих методов, не содержит повторяющихся элементов. Порядок следования элементов определяется порядком их первых вхождений в первую

* Следует заметить, что в книге [6] – одной из немногих книг на русском языке, посвященных LINQ, – ошибочно сказано, что методы OrderBy и OrderByDescending выполняют неустойчивую сортировку.

исходную последовательность. При выполнении метода Union ко всем (различным) элементам первой последовательности добавляются элементы второй последовательности, отсутствующие в первой. При выполнении методов Intersect или Except выходная последовательность содержит только те (различные) элементы первой последовательности, которые присутствуют или, соответственно, отсутствуют во второй.

10.3.4. Проецирование

```
                                IEnumerable<TSource> → IEnumerable<TResult>
Select((TSource[, int]) => TResult)
SelectMany((TSource[, int]) => IEnumerable<TResult>)
```

Метод Select выполняет преобразование каждого элемента входной последовательности в соответствии с указанным лямбда-выражением и помещает результат этого преобразования в выходную последовательность. Таким образом, количество элементов во входной и выходной последовательностях всегда будет одинаковым, однако тип их элементов может различаться (в качестве типа элементов выходной последовательности может использоваться анонимный тип). При преобразовании элемента может учитываться его индекс в исходной последовательности, который указывается в качестве второго, необязательного параметра лямбда-выражения.

Метод SelectMany отличается от метода Select тем, что в нем каждый элемент входной последовательности может быть преобразован в *несколько* (0 или более) элементов выходной последовательности. В частности, данный метод может использоваться для преобразования иерархической последовательности («последовательности последовательностей») в «плоскую» последовательность. Приведем пример:

```
string[] src = { "AB CD", "E F G", "XYZ" };
var res = src.SelectMany(s => s.Split());
```

В результате последовательность res (типа IEnumerable<string>) будет представлять собой последовательность *слов*, полученных из последовательности src: AB, CD, E, F, G, XYZ.

Если бы во втором операторе вместо метода SelectMany был вызван метод Select, то последовательность res состояла бы из трех элементов – *строковых массивов*, каждый из которых содержал бы все слова из соответствующей строки последовательности src; в этом случае последовательность res имела бы тип IEnumerable<string[]>.

Если бы в приведенном выше методе SelectMany лямбда-выражение имело вид s => s, то в результате была бы получена последовательность *символов*, содержащихся в элементах массива src. Это связано с тем, что класс string реализует интерфейс IEnumerable<char> и, таким образом, может интерпретироваться как символьная последовательность.

С помощью комбинации методов Select и SelectMany можно организовать перебор *упорядоченных пар* элементов двух последовательностей (т. е. получить *перекрестное объединение* последовательностей, или, иначе говоря, их *декартово произведение*) и вернуть «плоскую» последовательность – результат этого перебора. Приведем пример:

```
int[] src1 = { 10, 20, 30 }, src2 = { 0, 1, 2, 3 };
var res = src1.SelectMany(a => src2.Select(b => a + b));
```

В результате последовательность res (типа IEnumerable<int>) будет содержать значения всех попарных сумм элементов двух исходных последовательностей в следующем порядке: 10, 11, 12, 13, 20, 21, 22, 23, 30, 31, 32, 33.

10.3.5. Объединение

```
                                IEnumerable<TOuter> → IEnumerable<TResult>
Join(IEnumerable<TInner> inner, TOuter => TKey, TInner => TKey,
                (TOuter, TInner) => TResult)
GroupJoin(IEnumerable<TInner> inner, TOuter => TKey, TInner => TKey,
                (TOuter, IEnumerable<TInner>) => TResult)
```

В методах Join и GroupJoin, как и в методах группы 3, используются две исходных последовательности: первая («внешняя») последовательность вызывает данные методы, а вторая («внутренняя») указывается в качестве первого параметра inner.

Метод Join выполняет *внутреннее объединение* двух исходных последовательностей по ключу. Ключи для внешней и внутренней последовательности определяются первыми двумя лямбда-выражениями метода; к каждой паре элементов внешней и внутренней последовательности, имеющих одинаковые ключи, применяется третье лямбда-выражение, результат которого заносится в выходную последовательность. Подобное объединение называется внутренним, поскольку выходная последовательность будет формироваться на основе только тех элементов внешней входной последовательности, для которых найден *хотя бы один* элемент внутренней последовательности с таким же ключом.

Метод GroupJoin также выполняет объединение двух последовательностей по ключу, однако в нем каждый результирующий элемент определяется по элементу внешней последовательности и *всем* элементам внутренней последовательности с тем же ключом. Подобное объединение называется *левым внешним объединением*; в нем любой элемент из первой («левой») последовательности примет участие в формировании выходной последовательности, даже если для него не найдется ни одного «парного» элемента из второй последовательности.

Приведем примеры использования данных методов.

```
int[] src1 = { 10, 21, 33, 84 };
```

```
int[] src2 = { 40, 51, 52, 53, 60 };
var res = src1.Join(src2, a => a % 10, b => b % 10, (a, b) => a + "-" + b);
```

Ключом для сравнения двух чисел в данном случае является их последняя цифра, поэтому для каждого числа из массива src1 парными будут считаться числа из массива src2, оканчивающиеся на ту же цифру. Полученная последовательность будет состоять из следующих строк: «10-40», «10-60», «21-51», «33-53». Порядок следования пар определяется порядком элементов внешней последовательности, а для пар с одним и тем же первым элементом – порядком элементов внутренней последовательности.

Для применения к тому же набору исходных данных метода GroupJoin следует откорректировать последнее лямбда-выражение, поскольку в данном случае его второй параметр будет не числом, а *числовой последовательностью* (для большей наглядности мы изменили имя этого параметра на bb):

```
var res = src1.GroupJoin(src2, a => a % 10, b => b % 10,
  (a, bb) => a + "-" + bb.Count());
```

В данном случае с каждым элементом внешней последовательности src1 будет связано *количество* парных для него элементов, найденных во внутренней последовательности src2: «10-2», «21-1», «33-1», «84-0». Для подсчета количества элементов мы воспользовались методом Count, описанным далее (см. п. 10.3.10).

Для любого из рассматриваемых методов предусмотрен перегруженный вариант с дополнительным (последним) параметром comparer типа IEqualityComparer<TKey>, позволяющим переопределить способ сравнения ключей на равенство.

Заметим, что для методов Join и GroupJoin используется эффективная реализация, не сводящаяся к двойному циклу с попарными проверками ключей: внутренняя последовательность предварительно преобразуется к специальному виду – индексированной по ключу *таблице просмотра* типа ILookup (см. п. 10.3.8), что позволяет находить ее элементы, парные к элементам внешней последовательности, не прибегая к их многократному перебору.

Замечание 1. Построение плоского внешнего объединения. Метод Join возвращает «плоское» внутреннее объединение, а метод GroupJoin – иерархическое левое внешнее объединение. Для того чтобы получить плоское внешнее объединение, необходимо применить цепочку из запросов GroupJoin и SelectMany. Однако простое применение к результату запроса GroupJoin метода SelectMany не решит задачу, так как в результате будет получено *внутреннее* объединение. Приведем пример, использующий описанные выше массивы src1 и src2:

```
var res = src1.GroupJoin(src2, a => a % 10, b => b % 10,
  (a, bb) => bb.Select(e => a + "-" + e))
```

```
  .SelectMany(e => e);
```

Полученная последовательность будет состоять из строк «10-40», «10-60», «21-51», «33-53», т. е. будет совпадать с последовательностью, полученной с применением метода Join. Объясняется это тем, что метод SelectMany игнорирует все пустые последовательности. Проблема решается с помощью вспомогательного метода DefaultIfEmpty, позволяющего преобразовать пустую последовательность в одноэлементную последовательность (см. ее описание в п. 10.3.1):

```
var res = src1.GroupJoin(src2, a => a % 10, b => b % 10,
  (a, bb) => bb.DefaultIfEmpty().Select(e => a + "-" + e))
  .SelectMany(e => e);
```

В результате выполнения данного запроса дополнительно будет выведена строка «84-0», соответствующая элементу последовательности src1, не имеющего парных элементов в последовательности src2. Таким образом, будет действительно получено плоское левое внешнее объединение. Заметим, что вместо числа 0 в подобных строках можно использовать любое другое подходящее значение; для этого его достаточно указать в качестве параметра метода DefaultIfEmpty.

Замечание 2. Запрос Zip (.NET 4.0). В версии .NET 4.0 набор методов LINQ, связанных с объединением, был дополнен методом Zip, позволяющим комбинировать две последовательности с элементами типа TFirst и TSecond, объединяя их элементы *с одинаковыми индексами* (способ объединения определяется вторым параметром – лямбда-выражением; «лишние» элементы более длинной последовательности не обрабатываются):

```
                                IEnumerable<TFirst> → IEnumerable<TResult>
Zip(IEnumerable<TSecond> second, (TFirst, TSecond) => TResult)
```

10.3.6. Группировка

```
                IEnumerable<TSource> → последовательность другого типа
GroupBy(TSource => TKey) → IEnumerable<IGrouping<TKey, TSource>>
GroupBy(TSource => TKey, TSource => TElement)
                → IEnumerable<IGrouping<TKey, TElement>>
GroupBy(TSource => TKey, (TKey, IEnumerable<TSource>) => TResult)
                → IEnumerable<TResult>
GroupBy(TSource => TKey, TSource => TElement,
                (TKey, IEnumerable<TElement>) => TResult)
                → IEnumerable<TResult>
```

Метод GroupBy реорганизует элементы «плоской» входной последовательности в последовательность *групп* элементов (т. е. в иерархическую последовательность, или «последовательность последовательностей»). С этой точки зрения данный метод является в некоторым смысле «обрат-

ным» к методу SelectMany, который преобразует иерархическую последовательность в плоскую.

Группировка производится по *ключу*; лямбда-выражение для ключа является первым параметром в любом перегруженном варианте метода GroupBy. Если этот параметр является единственным, то выходная последовательность содержит элементы типа IGrouping<TKey, TSource>, которые сами являются подпоследовательностями с элементами типа TSource и дополнительно содержат свойство для чтения Key – ключ, связанный с соответствующей подпоследовательностью. Приведем определение данного обобщенного интерфейса:

```
public interface IGrouping<TKey, TElement> :
  IEnumerable<TElement>, IEnumerable
{
  TKey Key { get; }
}
```

Если в качестве второго параметра метода GroupBy указать лямбда-выражение, преобразующее параметр типа TSource к некоторому другому объекту типа TElement, то в подпоследовательность будут включаться объекты типа TElement.

Элементы результирующей последовательности можно определить явно; для этого надо использовать лямбда-выражение, которое по ключу и связанной с ним подпоследовательности определяет элемент выходной последовательности типа TResult (при этом может использоваться как подпоследовательность исходных элементов типа TSource, так и подпоследовательность новых объектов типа TElement, определенных с помощью соответствующего лямбда-выражения).

В качестве примера применения данного метода разобьем набор целых чисел на группы, в каждую из которых будут входить числа, оканчивающиеся на одну и ту же цифру:

```
int[] src = { 10, 21, 33, 84, 40, 51, 52, 53, 60 };
var res = src.GroupBy(i => i % 10);
foreach (var g in res)
{
  Console.Write("Key = " + g.Key + ":");
  foreach (var n in g)
    Console.Write(" " + n);
  Console.WriteLine();
}
```

В данном примере приведен также фрагмент, обеспечивающий вывод результата; этот фрагмент представляет собой вложенный цикл. Во внешнем цикле перебираются группы, входящие в полученную последовательность (т. е. подпоследовательности); эти группы имеют тип IGrouping<int,

int>; во внутреннем цикле перебираются элементы этих групп, т. е. целые числа, соответствующие одному и тому же ключу. В результате на экран будет выведен следующий текст:

```
Key = 0: 10 40 60
Key = 1: 21 51
Key = 3: 33 53
Key = 4: 84
Key = 2: 52
```

Для демонстрации второго перегруженного варианта метода GroupBy можно сформировать выходную последовательность, в группах которых у чисел будут отброшены последние цифры (поскольку эти цифры всегда можно восстановить по ключу):

```
var res = src.GroupBy(i => i % 10, i => i / 10);
```

Остальные операторы не изменятся. В данном случае вывод будет следующим:

```
Key = 0: 1 4 6
Key = 1: 2 5
Key = 3: 3 5
Key = 4: 8
Key = 2: 5
```

С помощью двух последних перегруженных вариантов метода мы можем явным образом сформировать выходную последовательность; при этом она не обязана состоять из подпоследовательностей, однако разумно предусмотреть для каждого элемента поле, содержащее связанный с ним ключ (таким образом, нам придется воспользоваться анонимным типом). Можно, например, связать с каждым ключом *сумму* значений всех чисел, имеющих данный ключ:

```
var res = src.GroupBy(i => i % 10,
  (k, nn) => new { Key = k, Sum = nn.Sum() });
foreach (var g in res)
  Console.WriteLine("Key = " + g.Key + ": " + g.Sum);
```

В данном примере мы воспользовались методом Sum, описанным далее (см. п. 10.3.10). Результат:

```
Key = 0: 110
Key = 1: 72
Key = 3: 86
Key = 4: 84
Key = 2: 52
```

Чтобы проиллюстрировать последний, четвертый вариант метода GroupBy, свяжем с каждым ключом сумму чисел, в которых отброшена последняя цифра (цикл для вывода полученной последовательности не изменится):

```
var res = src.GroupBy(i => i % 10, i => i / 10,
```

```
  (k, nn) => new { Key = k, Sum = nn.Sum() });
```

Результат:

```
Key = 0: 11
Key = 1: 7
Key = 3: 8
Key = 4: 8
Key = 2: 5
```

Для любого из приведенных вариантов метода GroupBy предусмотрен перегруженный вариант с дополнительным (последним) параметром comparer типа IEqualityComparer<TKey>, позволяющим переопределить способ сравнения ключей на равенство.

10.3.7. Импортирование

OfType<TResult>()	IEnumerable → IEnumerable<TResult>
Cast<TResult>()	

Методы OfType и Cast предназначены, прежде всего, для преобразования коллекций, реализующих только необобщенный интерфейс IEnumerable (например, коллекций типа ArrayList), в последовательности IEnumerable<T>, к которым уже можно применять все виды запросов LINQ. Различаются данные методы способом обработки особой ситуации, когда тип элемента входной коллекции отличен от TResult и, кроме того, тип TResult не входит в число его предков: в этом случае метод OfType просто игнорирует подобный элемент, не включая его в выходную последовательность, а метод Cast возбуждает исключение.

Следует заметить, что с помощью метода OfType нельзя выполнять статические преобразования скалярных типов (например, int в long, long в int, int в char и т. п.), поскольку участвующие в этих преобразованиях типы не связаны отношениями «предок – потомок». В частности, при выполнении следующего фрагмента будет получена пустая последовательность res:

```
int[] src = {1, 2, 3};
var res = src.OfType<long>(); // res не содержит ни одного элемента
```

Имеется простой способ определить, будет ли метод OfType выполнять преобразование элемента e к типу TResult: это будет происходить только в том случае, если операция e is TResult вернет значение true.

Заметим, что для преобразования, например, последовательности типа IEnumerable<int> в последовательность типа IEnumerable<long> можно воспользоваться другим видом запроса, а именно методом Select:

```
var res = src.Select(n => (long)n);
```

В данном случае на этапе компиляции известно, что n имеет тип int, и поэтому компилятор может выполнить статическое преобразование типа int к типу long.

Метод OfType может также применяться для *отбора* из исходной коллекции элементов-потомков определенного типа. Например, если у класса T1 имеются два класса-потомка T2 и T3, то для отбора из коллекции src типа IEnumerable<T1> только тех элементов, фактический тип которых равен T2, достаточно выполнить следующий оператор:

```
var res1 = src.OfType<T2>();
```

Подобного результата можно добиться, и используя метод Where:

```
var res2 = src.Where(e => e is T2);
```

Последовательности res1 и res2 будут содержать одни и те же элементы, однако их *тип* будет различным: последовательность res1 имеет тип IEnumerable<T2>, в то время как последовательность res2 – «старый» тип IEnumerable<T1>. Таким образом, чтобы получить последовательность, полностью идентичную res1, не применяя метод OfType, придется воспользоваться менее эффективной цепочкой из двух запросов:

```
var res3 = src.Where(e => e is T2).Select(e => (T2)e);
```

10.3.8. Экспортирование

```
                    IEnumerable<TSource> → коллекция определенного типа
ToArray() → TSource[]
ToList() → List<TSource>
ToDictionary(TSource => TKey) → Dictionary<TKey, TSource>
ToDictionary(TSource => TKey, TSource => TElement)
                → Dictionary<TKey, TElement>
ToLookup(TSource => TKey) → ILookup<TKey, TSource>
ToLookup(TSource => TKey, TSource => TElement)
                → ILookup<TKey, TElement>
AsEnumerable<TSource>() → IEnumerable<TSource>
AsQueryable<TSource>() → IQueryable<TSource>
```

Методы ToArray, ToList, ToDictionary преобразуют входную последовательность в коллекцию указанного типа; при этом сконструированный запрос выполняется *немедленно*. При выполнении метода ToDictionary обязательно указывается лямбда-выражение для вычисления *ключа* ассоциативного массива Dictionary по элементу входной последовательности; можно также указать лямбда-выражение для вычисления *значения* элемента, связанного с данным ключом; если второе лямбда-выражение не указано, то этим значением считается значение соответствующего элемента входной последовательности. Необходимо, чтобы для каждого элемента входной последовательности генерировался *уникальный* ключ; если это условие нарушается, то метод ToDictionary возбуждает исключение.

Метод ToLookup преобразует последовательность в так называемую *таблицу просмотра* – индексированную ключом типа TKey коллекцию типа

ILookup<TKey, TElement>, которая отличается от ассоциативного массива Dictionary тем, что с каждым ключом можно связывать *несколько* значений (т. е. *последовательность* значений), а также тем, что полученная таблица просмотра будет доступна только для чтения. Таблицы просмотра используются в реализациях методов Join и GroupJoin для повышения эффективности их работы. Приведем интерфейс таблиц просмотра:

```
public interface ILookup<TKey, TElement> :
  IEnumerable<IGrouping<TKey, TElement>>, IEnumerable
{
  int Count { get; }
  bool Contains(TKey key);
  IEnumerable<TElement> this[TKey key] { get; }
}
```

Для любого из приведенных вариантов методов ToDictionary и ToLookup предусмотрен перегруженный вариант с дополнительным (последним) параметром comparer типа IEqualityComparer<TKey>, позволяющим переопределить способ сравнения ключей на равенство.

Метод AsEnumerable предназначен, в основном, для преобразования к типу IEnumerable<T> последовательностей, используемых в других интерфейсах LINQ (и, прежде всего, в запросах к удаленным базам данных); метод AsQueryable, наоборот, обеспечивает преобразование «обычных» последовательностей (типа IEnumerable<T>) к типу IQueryable<T>, используемому в интерфейсах LINQ, предназначенных для работы с базами данных. Поскольку мы не рассматриваем использование технологии LINQ для работы с базами данных, укажем лишь один пример применения метода AsEnumerable: для приведения типа IOrderedEnumerable<T> к типу IEnumerable<T>. Как правило, этого не требуется делать явно, поскольку при присваивании подобное приведение выполняется автоматически:

```
IEnumerable<int> s2 = s1.OrderBy(i => i); // ошибки нет
```

Однако при использовании описателя var могут возникнуть проблемы:

```
var s2 = s1.OrderBy(i => i);
  // ошибки нет; s2 получает тип IOrderedEnumerable<int>
s2 = s2.Where(i => i > 0); // ошибка компиляции
```

Ошибка произойдет из-за того, что метод Where возвращает последовательность типа IEnumerable<int>, а эта последовательность не может быть неявно приведена к типу последовательности-потомка IOrderedEnumerable<int>, который имеет переменная s2. Для решения проблемы достаточно дополнить первый из двух операторов вызовом метода AsEnumerable:

```
var s2 = s1.OrderBy(i => i).AsEnumerable();
  // ошибки нет; s2 получает тип IEnumerable<int>
```

10.3.9. Поэлементные операции

```
First([TSource => bool])                    IEnumerable<TSource> → TSource
FirstOrDefault([TSource => bool])
Last([TSource => bool])
LastOrDefault([TSource => bool])
Single([TSource => bool])
SingleOrDefault([TSource => bool])
ElementAt(int index)
ElementAtOrDefault(int index)
```

Все перечисленные в данной группе методы возвращают одно скалярное значение, поэтому запросы, содержащие эти методы, выполняются немедленно после конструирования (т. е. не являются отложенными).

Если имя метода оканчивается на `OrDefault`, то при отсутствии требуемого элемента метод не возбуждает исключение (в отличие от парного к нему метода без суффикса «OrDefault»); вместо этого он возвращает значение `default(TSource)` – `null` для ссылочных типов и структуру с побитово обнуленными полями для размерных типов (в частности, 0 для числовых типов).

Варианты методов `First`, `Last`, `Single` без лямбда-выражений возвращают соответственно первый, последний и единственный элемент входной последовательности; наличие предиката (лямбда-выражения) приводит к тому, что первый, последний и единственный элемент выбирается только среди элементов, удовлетворяющих указанному предикату. Если требуемых элементов больше одного, то методы `Single` и `SingleOrDefault` возбуждают исключение (таким образом, метод `Single` не будет возбуждать исключение, только если в последовательности имеется *единственный* требуемый элемент, а метод `SingleOrDefault` – если количество требуемых элементов в последовательности равно 0 или 1).

В методах `ElementAt` и `ElementAtOrDefault` индексирование, как обычно, ведется от 0; метод `ElementAtOrDefault` возвращает значение `default(TSource)` даже для отрицательных индексов. Следует отметить, что если входная последовательность поддерживает интерфейс `IList<T>`, то эти методы выполняются быстро, так как для доступа к требуемому элементу вызывается соответствующий индексатор.

10.3.10. Агрегирование

```
                                    IEnumerable<TSource> → скалярный_тип
Count([TSource => bool]) → int
LongCount([TSource => bool]) → long
Average([TSource => числовой_тип]) → double или decimal
Sum([TSource => числовой_тип]) → числовой_тип
```

```
Max() → TSource
Max(TSource => TResult) → TResult
Min() → TSource
Min(TSource => TResult) → TResult
Aggregate((TSource, TSource) => TSource) → TSource
Aggregate(TResult seed, (TResult, TSource) => TResult) → TResult
Aggregate(TAccumulate seed, (TAccumulate, TSource) => TAccumulate,
                TAccumulate => TResult) → TResult
```

Методы `Count` и `LongCount` возвращают количество элементов во входной коллекции; при наличии дополнительного параметра-предиката возвращается количество элементов, удовлетворяющих указанному предикату. Если входная последовательность реализует интерфейс `ICollection<T>`, то для подсчета числа элементов просто вызывается метод `Count` данного интерфейса; в противном случае выполняется перебор элементов последовательности.

Методы `Average` и `Sum` могут вызываться без параметра, если элементы входной последовательности имеют числовой тип `int`, `long`, `float`, `double`, `decimal`. Метод `Sum` возвращает значение, тип которого совпадает с типом элементов последовательности или, при наличии лямбда-выражения, – с типом, возвращаемым лямбда-выражением. Метод `Average` возвращает значение `double` при обработке числовых данных любых типов, кроме `decimal`; для данных типа `decimal` возвращаемый результат также имеет тип `decimal`. Наряду с любым из указанных числовых типов в методах `Sum` и `Average` можно использовать его Nullable-варианты `int?`, `long?`, `float?`, `double?`, `decimal?` (см. п. 7.3); в этом случае возвращаемое значение тоже имеет Nullable-тип, обрабатываются только элементы последовательности, не равные `null`, а если таких элементов нет, то возвращается значение `null`.

Методы `Max` и `Min` могут вызываться без параметра, если тип элементов входной последовательности реализует интерфейс `IComparable<T>` (таковы, в частности, все элементарные типы, за исключением типа `object`). Этот же интерфейс должен реализовываться типом `TResult`, используемым в параметре – лямбда-выражении.

Метод `Aggregate` предназначен для реализации *нестандартного агрегирования*. В любом его варианте в качестве одного из параметров указывается лямбда-выражение с двумя параметрами `(seed, elem)`, определяющее, каким образом к уже имеющемуся значению *аккумулятора* `seed` будет добавляться значение очередного элемента `elem` исходной последовательности типа `IEnumerable<TSource>`.

В первом варианте метода `Aggregate` лямбда-выражение является единственным параметром, оно обрабатывает все элементы последовательности, начиная со второго, а первый элемент последовательности использу-

ется для задания начального значения аккумулятора seed. В данном случае все параметры и возвращаемое значение лямбда-выражения должны иметь тип TSource. В качестве примера приведем реализацию вычисления факториала (в ней используется метод Range из п. 10.3.12):

```
static int Fact1(int n)
{
  return Enumerable.Range(1, n).Aggregate((seed, elem) => seed * elem);
}
```

Данная реализация будет возвращать корректные значения факториала только в случае, если эти значения не будут превосходить int.MaxValue.

Первый вариант метода Aggregate может быть также использован, например, для нахождения минимального или максимального элемента последовательности, если сравнение элементов должно проводиться нестандартным образом. Приведенная функция возвращает первую строку максимальной длины из исходной последовательности:

```
static string MaxLength(Enumerable<string> s)
{
  return s.Aggregate((seed, elem) => seed.Length > elem.Length ?
    seed : elem);
}
```

Заметим, что попытка использования выражения s.Max() приведет к неверному результату, поскольку в этом случае будет выполняться *лексикографическое* сравнение строк.

Во втором варианте метода Aggregate начальное значение аккумулятора задается в качестве первого параметра, а лямбда-выражение обрабатывает все элементы последовательности, включая и первый. В подобной ситуации аккумулятор может иметь тип, отличный от типа TSource, причем тип возвращаемого значения должен совпадать с типом аккумулятора. С помощью этого варианта метода Aggregate можно реализовать более надежный вариант вычисления факториала, при котором произведение будет накапливаться в переменной вещественного типа:

```
static double Fact2(int n)
{
  return Enumerable.Range(1, n).Aggregate(1.0,
    (seed, elem) => seed * elem);
}
```

Например, Fact1(32) вернет неправильное значение -2147483648, а Fact2(32) – хотя и приближенное, но правильное по порядку величины значение 2.63130836933694E+35. Заметим, что вызов Fact2(100) вернет значение double.PositiveInfinity.

В третьем, наиболее гибком варианте метода Aggregate накопление результата выполняется так же, как и во втором, однако предусмотрено еще

одно лямбда-выражение, предназначенное для преобразования полученного значения аккумулятора к окончательному результату. При этом результат может иметь тип, отличающийся и от типа элементов обрабатываемой последовательности, и от типа аккумулятора. В качестве примера применения третьего варианта метода Aggregate приведем реализацию метода расширения (аналогичного методу Join класса string), выполняющего объединение строковых представлений всех элементов последовательности, при котором между соседними элементами помещается указанная строка-разделитель separator (разделитель может быть пустой строкой). Поскольку среди методов, связанных с обработкой последовательностей, уже имеются методы с именами Join и Concat, для данного метода выберем имя Combine:

```
public static class ExtCombine
{
  public static string Combine<T>(this IEnumerable<T> src,
    string separator)
  {
    return src.Aggregate("", (seed, s) =>
      seed + s.ToString() + separator, s => separator.Length > 0 ?
      s.Remove(s.Length - separator.Length) : s);
  }
}
```

Завершая описание метода Aggregate, отметим, что во многих ситуациях вместо данного метода проще использовать непосредственный алгоритм вычисления требуемой величины с применением цикла foreach.

10.3.11. Квантификаторы

```
All(TSource => bool)                          IEnumerable<TSource> → bool
Any([TSource => bool])
Contains(TSource value)
SequenceEqual(IEnumerable<TSource> second)
```

Метод All возвращает true, если *все* элементы последовательности удовлетворяют указанному параметру-предикату; метод Any возвращает true, если *какие-либо* элементы последовательности удовлетворяют указанному параметру-предикату (при отсутствии параметра метод Any возвращает true, если последовательность является непустой).

Метод Contains возвращает true, если в последовательности имеется *хотя бы один* элемент со значением value; метод SequenceEqual возвращает true, если последовательность, которая вызвала этот метод, совпадает с последовательностью second, указанной в качестве его параметра (последовательности считаются равными, если они содержат одни и те же элементы в том же самом порядке). Для методов Contains и SequenceEqual предусмотрен

перегруженный вариант с дополнительным (вторым) параметром comparer типа IEqualityComparer<TSource>, позволяющим переопределить способ сравнения элементов последовательности на равенство.

10.3.12. Генерирование последовательностей

```
                                        Enumerable → последовательность
Empty<TResult>() → IEnumerable<TResult>
Repeat<TResult>(TResult element, int count) → IEnumerable<TResult>
Range(int start, int count) → IEnumerable<int>
```

Метод Empty создает пустую последовательность типа IEnumerable<TResult>, метод Repeat создает последовательность типа IEnumerable<TResult>, содержащую count копий элемента element; метод Range создает числовую последовательность, содержащую count последовательных целых чисел, начиная с числа start.

Данные методы не являются методами расширения, поэтому их необходимо вызывать как обычные статические методы класса Enumerable.

Метод Empty оказывается полезным в ситуации, когда какие-либо из обрабатываемых коллекций могут оказаться отсутствующими, т. е. равными null (не следует путать эту ситуацию с ситуацией, когда коллекция существует, но является пустой). Рассмотрим пример:

```
string[] src = { "ABC", null, "10" };
var res = src.SelectMany(s => s);
foreach (var e in res)
  Console.Write(e); // будет возбуждено исключение
```

Данный фрагмент предназначен для получения «плоской» последовательности всех символов, входящих в строки из массива src. Однако из-за того что одна из строк равна null, выполнение запроса завершается аварийно. Для исправления данного фрагмента проще всего воспользоваться операцией ?? совместно с методом Empty:

```
var res = src.SelectMany(s => s ?? Enumerable.Empty<char>());
foreach (var e in res)
  Console.Write(e); // будут выведены символы ABC10
```

При обработке null-строки возвращается пустая коллекция, которая никак не влияет на выходную последовательность, но предотвращает возбуждение исключения.

Глава 11. Выражения запросов

Чтобы упростить применение запросов LINQ для программистов, знакомых с построением запросов SQL, в синтаксис языка C# 3.0 были введены вспомогательные конструкции, позволяющие представить наиболее

сложные виды запросов LINQ «в стиле SQL». Эти конструкции называются *выражениями запросов* (query expressions). Любое выражение запроса в конечном итоге преобразуется компилятором в цепочку обычных методов запросов; таким образом, выражения запросов являются конструкциями языка C#, упрощающими кодирование (подобно циклу `foreach`).

В форме выражений запросов можно записать следующие виды запросов LINQ:

- `Where` (п. 10.3.1),
- `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending` (п. 10.3.2),
- `Select` и `SelectMany` (п. 10.3.4),
- `Join` и `GroupJoin` (п. 10.3.5),
- `GroupBy` (п. 10.3.6).

Легко видеть, что выражения запросов ориентированы на наиболее сложные виды запросов LINQ, которые к тому же «наиболее близки» языку SQL. В то же время ничто не мешает комбинировать выражения запросов и вызовы обычных методов LINQ.

11.1. Описание грамматики выражений запросов

(1) Выражение должно начинаться с конструкции `from`. Конструкция `from` определяет переменную-перечислитель для одной из последовательностей, участвующих в запросе:

```
from [тип] перечислитель in последовательность
```

В качестве последовательности может указываться не только переменная типа `IEnumerable<T>`, но и любое выражение, возвращающее значение этого типа. Конструкция `from` не имеет аналога среди методов запросов; она лишь вводит в рассмотрение последовательность и связывает с ней перечислитель. Если тип перечислителя не указан, то он определяется по типу элементов последовательности; если тип указан, то элементы последовательности приводятся к этому типу с помощью метода `Cast`.

(2) Следующая часть выражения может содержать 0 или более конструкций четырех видов, описываемых ниже (эти конструкции могут располагаться в любом порядке).

(2.1) Конструкция `join` позволяет организовать объединение последовательностей методом `Join` или `GroupJoin`:

```
join [тип] внутренний_перечислитель in внутренняя_последовательность
  on внешний_ключ equals внутренний_ключ [into идентификатор]
```

Завершающая часть конструкции (`into идентификатор`) является необязательной; при ее наличии объединение выполняется не методом `Join`, а методом `GroupJoin` (при этом идентификатор, указываемый после слова `into`, связывается с последовательностью элементов внутренней последовательности, имеющих тот же ключ, что и текущее значение переменной-

перечислителя внешней последовательности). Если тип внутреннего перечислителя не указан, то он определяется по типу элементов внутренней последовательности; если тип указан, то элементы внутренней последовательности приводятся к этому типу (с помощью метода Cast).

(2.2) Конструкция let вводит в запрос переменную и присваивает ей значение:

```
let идентификатор = выражение
```

Конструкция let не имеет аналога среди методов запросов.

(2.3) Конструкция where обеспечивает фильтрацию элементов для последовательности или объединения последовательностей (методом Where):

```
where логическое_выражение
```

(2.4) Конструкция orderby состоит из одного или более полей сортировки с необязательным указанием направления сортировки (поля разделяются запятыми):

```
orderby выражение1 [ascending | descending],
  выражение2 [ascending | descending] ...
```

Данная конструкция аналогична вызову цепочки методов OrderBy, OrderByDescending, ThenBy, ThenByDescending (начинающейся с метода OrderBy или OrderByDescending, после которого следует 0 или более методов ThenBy или ThenByDescending).

(3) В данном месте выражения могут располагаться 0 или более повторений конструкций (1)–(2), начиная с конструкции (1). Заметим, что с помощью нескольких конструкций from обеспечивается реализация метода запроса SelectMany.

(4) Следующая часть выражения должна содержать одну из двух конструкций, завершающих выражение запроса: select или group.

(4.1) Конструкция select обеспечивает проецирование исходных элементов на элементы выходной последовательности (в зависимости от количества предваряющих конструкций from конструкция select обеспечивает вызов либо метода Select, либо метода SelectMany):

```
select выражение
```

(4.2) Конструкция group также обеспечивает проецирование исходных элементов на элементы выходной последовательности, но дополнительно выполняет группировку полученных элементов по ключу методом GroupBy:

```
group выражение by ключ_группировки
```

(5) После завершающей конструкции (4) может располагаться необязательная конструкция *продолжения запроса* into. Она связывает перечислитель последовательности, полученной в результате выполнения предшествующей части запроса, с идентификатором, указанным после into:

```
into идентификатор
```

После конструкции into должны располагаться другие части выраже-

ния запроса, описываемые конструкциями (1)–(4), начиная с конструкций (1) или (2).

Не следует смешивать конструкцию `into`, с которой начинается продолжение запроса, с одноименным фрагментом, которым может завершаться конструкция `join`.

11.2. Обзор конструкций выражения запросов

В настоящем пункте даются краткие характеристики всех конструкций, входящих в выражения запросов, и описывается, каким образом компилятор преобразует эти конструкции в методы LINQ.

11.2.1. Конструкции from и select

Если в конструкции `from` указывается тип перечислителя, то к последовательности применяется метод `Cast`:

```
from T e in s          |   from e in s.Cast<T>()
…                      |   …
```

Если после `select` указывается перечислитель из `from`, то обе эти конструкции просто заменяются на имя последовательности:

```
from e in s            |   s
  select e             |
```

В любом другом случае к последовательности применяется метод `Select`, а имя перечислителя и выражение, указанное после `select`, используются в лямбда-выражении для этого метода:

```
from e in s            |   s.Select(e => f)
  select f             |
```

Если перед конструкцией `select` указаны две конструкции `from`, то весь этот фрагмент заменяется на вызов метода `SelectMany`, причем вторая конструкция `from` переносится в лямбда-выражение для этого метода:

```
from e1 in s1          |   s1.SelectMany(e1 =>
from e2 in s2          |     from e2 in s2 select f)
  select f             |
```

В случае, если после двух конструкций `from` следует конструкция, отличная от `select`, выполняется преобразование, вводящее дополнительный перечислитель `x` и вспомогательный анонимный класс (в дальнейшем полученное выражение подвергается повторному преобразованию, описанному выше, поскольку теперь две конструкции `from` оканчиваются конструкцией `select`):

```
from e1 in s1
from e2 in s2
…
```

```
from x in
  from e1 in s1
  from e2 in s2
  select new { e1, e2 }
…
```

Приведем пример. Рассмотрим выражение запроса, обеспечивающее построение перекрестного объединения (т. е. декартова произведения) трех экземпляров одной и той же двухэлементной последовательности:

```
string[] src = { "0", "1" };
var res = from e1 in src
          from e2 in src
          from e3 in src
          select e1 + e2 + e3;
foreach (var e in res)
  Console.Write(e + " ");
```

Смысл данного выражения ясен: перебираются всевозможные комбинации троек `e1`, `e2`, `e3`, причем элементы этих троек берутся из одной и той же последовательности `src`. Естественно ожидать, что в результирующих тройках быстрее будут изменяться «правые» элементы (т. е. элементы, соответствующие «внутренним» конструкциям `from`) – по аналогии с поведением параметров вложенных циклов. Действительно, при выполнении данного фрагмента на экран будет выведен следующий текст:

```
000 001 010 011 100 101 110 111
```

Выясним, в какую цепочку методов запросов будет преобразовано данное выражение. Поскольку оно начинается с двух конструкций `from`, за которыми *не следует* конструкция `select`, эти первые две конструкции `from` заключаются в «оболочку» вспомогательного запроса с новым перечислителем:

```
var res = from x in
              from e1 in src
              from e2 in src
              select new { e1, e2 }
          from e3 in src
          select x.e1 + x.e2 + e3;
```

Заметим, что теперь перечислители `e1` и `e2` «погружены» в новый анонимный тип и превратились в его поля, поэтому всюду далее нам пришлось изменить обращения к этим перечислителям, дополнив их префиксом «`x.`». Полученный фрагмент можно откомпилировать; результат его выполнения не будет отличаться от приведенного выше.

Теперь можно преобразовать вспомогательный запрос, поскольку он представляет собой две конструкции `from`, после которых следует конструкция `select`. Как было указано выше, подобные запросы заменяются на

вызов метода SelectMany:

```
var res = from x in
            src.SelectMany(e1 => from e2 in src select new { e1, e2 })
          from e3 in src
          select x.e1 + x.e2 + e3;
```

Результат выполнения данного запроса не отличается от предыдущих.

Внешний запрос теперь также представляет собой две конструкции from, завершающиеся конструкцией select (при этом в первой конструкции from в качестве последовательности выступает возвращаемое значение метода SelectMany). Значит, его также можно преобразовать в вызов метода SelectMany:

```
var res = src.SelectMany(e1 => from e2 in src
                               select new { e1, e2 })
              .SelectMany(x => from e3 in src
                               select x.e1 + x.e2 + e3);
```

Для того чтобы полностью избавиться от элементов выражения запроса, осталось преобразовать две конструкции from … select в лямбда-выражениях, заменив эти конструкции на вызовы метода Select:

```
var res = src.SelectMany(e1 => src.Select(e2 => new { e1, e2 }))
              .SelectMany(x => src.Select(e3 => x.e1 + x.e2 + e3));
```

Разумеется, результат выполнения данного запроса также не будет отличаться от предыдущих.

Сравним полученную цепочку вызовов методов запросов с исходным выражением запроса:

```
var res = from e1 in src
          from e2 in src
          from e3 in src
          select e1 + e2 + e3;
```

Выражение запроса является существенно более наглядным, поскольку не содержит ни вложенных вызовов методов, ни лямбда-выражений, ни анонимных типов, ни вспомогательных перечислителей. Здесь уместна аналогия с циклом foreach, «скрывающим» от программиста детали реализации перебора последовательности. Впрочем, надо отметить, что подобное существенное повышение наглядности обеспечивается не всегда; оно наиболее характерно для запросов, в которых используются методы SelectMany.

11.2.2. Конструкция group

Если после group указывается перечислитель из from, то используется метод GroupBy с одним параметром:

```
from e in s                 |   s.GroupBy(e => k)
  group e by k              |
```

В любом другом случае используется метод GroupBy с двумя параметрами:

```
from e in s                    |   s.GroupBy(e => k, e => f)
  group f by k                 |
```

11.2.3. Конструкция orderby

Приведем пример, в котором упорядочивание по всем ключам производится по возрастанию:

```
from e in s                    |   from e in s.OrderBy(e => o1)
  orderby o1, o2, …            |     .ThenBy(e => o2) …
```

Если после какого-либо элемента списка в конструкции orderby указывается модификатор descending, то в соответствующем месте цепочки методов запросов используется метод с суффиксом «Descending».

11.2.4. Конструкция let

Конструкция from, за которой следует конструкция let, преобразуется следующим образом:

```
from e in s                    |   from x in
  let l = v                    |     from e in s
                               |     select new { e, l = v }
```

При преобразовании вводится дополнительный перечислитель x и вспомогательный анонимный класс.

Данная конструкция позволяет в дальнейшем использовать в выражении запроса более краткий идентификатор l вместо связанного с ним выражения. Приведем пример:

```
string[] src = { "Иван Владимиров", "Сергей Петров", "Виктор Сидоров",
  "Анатолий Васильев", "Лев Яшин" };
var res = from s in src
          let fam = s.Split()[1]
          orderby fam.Length
          select fam;
```

Приведенный запрос возвращает фамилии, отсортированные по возрастанию их длины:

```
Яшин Петров Сидоров Васильев Владимиров
```

Выясним, в какую цепочку методов запросов преобразуется данное выражение. Начнем с преобразования конструкции let:

```
var res = from x in
            from s in src
            select new { s, fam = s.Split()[1] }
          orderby x.fam.Length
```

```
          select x.fam;
```

Вложенный запрос преобразуется в метод `Select`:

```
var res = from x in src.Select(s => new { s, fam = s.Split()[1] })
          orderby x.fam.Length
          select x.fam;
```

Теперь можно преобразовать конструкцию `orderby`:

```
var res = from x in src.Select(s => new { s, fam = s.Split()[1] })
            .OrderBy(x => x.fam.Length)
          select x.fam;
```

Наконец, преобразуем конструкцию `select` в метод `Select`:

```
var res = src.Select(s => new { s, fam = s.Split()[1] })
          .OrderBy(x => x.fam.Length).Select(x => x.fam);
```

Сравним полученную цепочку методов с исходным выражением запроса:

```
var res = from s in src
          let fam = s.Split()[1]
          orderby fam.Length
          select fam;
```

Как и в предыдущем примере, выражение запроса выглядит гораздо нагляднее. Однако это объясняется тем, что мы рассматриваем цепочку методов, полученную путем формального преобразования исходного выражения. Если попытаться реализовать требуемый запрос «на пустом месте», не ориентируясь на вид выражения запроса, то полученная цепочка методов будет даже короче выражения запроса, не уступая ему в наглядности:

```
var res = src.Select(s => s.Split()[1]).OrderBy(x => x.Length);
```

11.2.5. Конструкция where

Конструкция `from`, за которой следует конструкция `where`, преобразуется следующим образом (к исходной последовательности `s` применяется метод `Where`):

`from e in s` `  where w`	`from e in s.Where(e => w)`

11.2.6. Конструкция join

Конструкция `join`, вместе с предшествующей ей конструкцией `from` и последующей конструкцией `select`, преобразуется двумя различными способами, в зависимости от наличия или отсутствия элемента `into`:

`from e1 in s1`
`  join e2 in s2`
`  on k1 equals k2`
`  select f`
	`from x in s1.Join(s2, e1 => k1,`
	`  e2 => k2, (e1, e2) => f)`
	`select x`

```
from e1 in s1
  join e2 in s2
  on k1 equals k2
  into i
  select f
```

```
from x in s1.GroupJoin(s2, e1 => k1,
  e2 => k2, (e1, i) => f)
select x
```

Элемент `into i` является не просто формальным признаком того, что следует использовать метод `GroupJoin` вместо `Join`. Определяемый в этом элементе идентификатор `i` обозначает *последовательность* тех элементов последовательности `s2`, которые имеют тот же ключ, что и элемент `e1` последовательности `s1`. Обратите внимание на то, что именно идентификатор `i` входит в качестве второго параметра в последнее лямбда-выражение метода `GroupJoin`. Таким образом, в первом варианте конструкции `join` (без элемента `into`) в выражении `f` можно использовать перечислители `e1` и `e2`, тогда как во втором варианте этой конструкции (с элементом `into`) в выражении `f` следует использовать перечислитель `e1` и последовательность `i`. В обоих вариантах при определении ключа `k1` надо использовать перечислитель `e1`, а при определении ключа `k2` – перечислитель `e2`.

Если конструкция `join` не предшествует непосредственно конструкции `select`, то подобный фрагмент преобразуется во вложенный запрос, содержащий дополнительный перечислитель и вспомогательный анонимный класс (в дальнейшем полученное выражение подвергается повторному преобразованию, описанному выше, поскольку теперь конструкция `join` оканчивается конструкцией `select`):

```
from e1 in s1
  join e2 in s2
  on k1 equals k2
…
```

```
from x in
  from e1 in s1
  join e2 in s2
  on k1 equals k2
  select new { e1, e2 }
…
```

```
from e1 in s1
  join e2 in s2
  on k1 equals k2
  into i
…
```

```
from x in
  from e1 in s1
  join e2 in s2
  on k1 equals k2
  into i
  select new { e1, i }
…
```

Если в конструкции `join` указывается тип перечислителя `e2`, то к последовательности `s2` применяется метод `Cast`:

```
join T e2 in s2
…
```

```
join e2 in s2.Cast<T>()
…
```

В качестве примера реализуем в виде выражений запросов три запроса, рассмотренных при описании методов `Join` и `GroupJoin` (см. п. 10.3.5):

```
int[] src1 = { 10, 21, 33, 84 };
int[] src2 = { 40, 51, 52, 53, 60 };
var res1 = src1.Join(src2, a => a % 10, b => b % 10,
            (a, b) => a + "-" + b);
var res2 = src1.GroupJoin(src2, a => a % 10, b => b % 10,
            (a, bb) => a + "-" + bb.Count());
var res3 = src1.GroupJoin(src2, a => a % 10, b => b % 10,
            (a, bb) => bb.DefaultIfEmpty().Select(e => a + "-" + e))
           .SelectMany(e => e);
```

Во всех запросах ключом является последняя (правая) цифра числа. В первом запросе строится плоское внутреннее объединение, во втором – иерархическое левое внешнее объединение (при этом элементы выходной последовательности включают значение левого элемента объединения и *количество* правых элементов, входящих в это же объединение) и, наконец, в третьем – плоское левое внешнее объединение.

Приведем элементы полученных последовательностей:

```
res1: 10-40, 10-60, 21-51, 33-53
res2: 10-2, 21-1, 33-1, 84-0
res3: 10-40, 10-60, 21-51, 33-53, 84-0
```

Те же самые запросы, оформленные в виде выражений запросов, будут выглядеть следующим образом:

```
int[] src1 = { 10, 21, 33, 84 };
int[] src2 = { 40, 51, 52, 53, 60 };
var res1 = from a in src1
           join b in src2
           on a % 10 equals b % 10
           select a + "-" + b;
var res2 = from a in src1
           join b in src2
           on a % 10 equals b % 10
           into bb
           select a + "-" + bb.Count();
var res3 = from a in src1
           join b in src2
           on a % 10 equals b % 10
           into bb
           from e in bb.DefaultIfEmpty()
           select a + "-" + e;
```

Рассмотрим, как будет выполняться преобразование последнего запроса. В нем конструкция `join` не завершается конструкцией `select`, поэтому данная конструкция преобразуется во вложенный запрос, связанный с

дополнительным перечислителем x (при этом имя перечислителя x будет префиксом для имен bb и a, используемых в двух последних конструкциях запроса):

```
var res3 = from x in
             from a in src1
             join b in src2
             on a % 10 equals b % 10
             into bb
             select new { a, bb }
           from e in x.bb.DefaultIfEmpty()
           select x.a + "-" + e;
```

Данное выражение запроса содержит две «внешние» конструкции from (from x и from e), завершающиеся конструкцией select, поэтому оно преобразуется к методу SelectMany, возвращающему «плоскую» последовательность элементов.

11.2.7. Конструкция into

Конструкция into i (i – идентификатор), с которой начинается продолжение запроса, преобразуется следующим образом:

```
предшествующий запрос
into i
продолжение запроса
```

```
from i in
  предшествующий запрос
продолжение запроса
```

Таким образом, конструкция into позволяет размещать фрагменты сложных запросов в последовательную цепочку, не оформляя начальную часть в виде вложенного запроса.

Глава 12. LINQ to XML

12.1. Язык XML

Язык XML (eXtensible Markup Language) – универсальный язык разметки структурированных (иерархических) данных. В отличие от языка HTML, также предназначенного для разметки данных, XML не содержит фиксированного набора тегов; еще одним отличием является более строгий набор правил, связанных с использованием тегов в XML-документах. С помощью языка XML удобно выполнять сериализацию данных (например, для их последующей передачи по сети или для сохранения текущего состояния приложения). Поскольку документ XML хранится в обычном текстовом формате, он не зависит от используемой операционной системы и архитектуры компьютера. При надлежащем форматировании документ

XML вполне может восприниматься человеком (хотя главное его назначение – служить источником данных для программ). «XML – это платформенно-, программно- и аппаратно-независимое средство для передачи информации» (см., например, [10]).

12.1.1. XML-документ и его составляющие

Приведем пример простого XML-документа, иллюстрирующего основные особенности языка XML.

```
<?xml version="1.0" encoding="windows-1251"?>
<описания-книг>
<!-- Использованы теги формата FB2 (Fiction Book 2.0) -->
  <title-info>
    <genre match="90">sf_fantasy</genre>
    <author>
      <first-name>Джон</first-name>
      <middle-name>Рональд Руэл</middle-name>
      <last-name>Толкин</last-name>
    </author>
    <book-title>Возвращение Короля</book-title>
    <lang>ru</lang>
    <src-lang>en</src-lang>
    <translator>
      <first-name>Мария</first-name>
      <last-name>Каменкович</last-name>
    </translator>
    <translator>
      <first-name>Валерий</first-name>
      <last-name>Каррик</last-name>
    </translator>
    <sequence name="Властелин Колец" number="3" />
  </title-info>
  <title-info>
    <genre>sf</genre>
    <author>
      <first-name>Аркадий</first-name>
      <last-name>Стругацкий</last-name>
    </author>
    <author>
      <first-name>Борис</first-name>
      <last-name>Стругацкий</last-name>
    </author>
    <book-title>Улитка на склоне</book-title>
    <lang>ru</lang>
  </title-info>
```

```
</описания-книг>
```

Как правило, любой XML-документ начинается с *объявления XML* (XML declaration), в котором, в частности, указывается версия стандарта XML и кодировка документа. В данном случае это однобайтная ANSI-кодировка кириллицы, хотя обычно в XML-документах используется более универсальная кодировка UTF-8 (см. п. 4.4.1), которая задается строкой «utf-8» (или вообще не указывается).

Объявление XML входит в так называемый *пролог XML* (XML prolog), в котором, наряду с объявлением, может содержаться дополнительная информация, связанная с документом в целом, например *объявление типа документа* (type document declaration). В нашем примере пролог состоит только из объявления XML.

Документ XML должен содержать *корневой элемент*, или *элемент документа* (root element, document element); в нашем случае корневой документ имеет имя `описания-книг`. Из приведенного примера видно, что в именах элементов можно использовать буквы различных алфавитов, хотя, как правило, применяются латинские буквы.

Любой элемент, в том числе корневой, может содержать *дочерние элементы* (child elements), а также *дочерние узлы* других видов (child nodes). В нашем случае элемент `описания-книг` содержит два дочерних элемента `title-info`, каждый из которых, в свою очередь, содержит набор элементов третьего уровня, а некоторые из этих элементов (например, `author`) – элементы четвертого уровня. На любом уровне вложенности элементы могут иметь совпадающие имена; в нашем примере – это имена `title-info`, `translator` (в первом элементе `title-info`), `author` (во втором элементе `title-info`).

Элементы могут включать *атрибуты* (attributes), которые указываются сразу за именем элемента в открывающем или комбинированном теге в формате `имя="значение"` (значения должны заключаться в кавычки, обычно двойные; атрибутов с одинаковыми именами в одном элементе быть не должно). В нашем примере атрибуты имеют два дочерних элемента первого элемента `title-info`: `genre` (атрибут `match`) и `sequence` (атрибуты `name` и `number`).

Примером содержимого элемента XML, отличного от его дочерних элементов, является обычный текст.

Если элемент имеет непустое содержимое, то это содержимое должно располагаться между открывающим и закрывающим *тегом* (tag) – маркером, заключенным в угловые скобки; при этом открывающий тег содержит имя элемента (например, `<author>`), возможно, дополненное набором атрибутов, а закрывающий тег – имя элемента, предваренное косой чертой (`</author>`). Если элемент не имеет содержимого, то его допустимо указывать с помощью единственного «комбинированного» тега, начинающегося

с имени элемента и оканчивающегося пробелом и косой чертой (в нашем примере так представлен элемент sequence). Впрочем, в данном случае можно использовать и парные теги, между которыми ничего не содержится (например, <sequence name="Властелин Колец" number="3"></sequence>).

В любом месте XML-документа можно помещать *комментарий* (comment) – текст, заключенный между специальными маркерами <!-- и -->, и *инструкции обработки* (processing instructions), заключенные между маркерами <? и ?>. XML-элементы могут также содержать «буквальные» *символьные данные* CDATA. Все эти особые виды XML-содержимого будут кратко описаны далее, при рассмотрении связанных с ними классов.

12.1.2. Корректные и действительные XML-документы

Любой XML-документ должен удовлетворять набору правил, при нарушении которых он считается *некорректным* (*неправильно сформированным)*. Перечислим основные из этих правил:

- если в документе присутствует описание, то оно должно располагаться в начале документа;
- в любом документе должен быть *единственный* корневой элемент;
- помимо буквенных и цифровых символов имена элементов и атрибутов могут содержать только символы «.» (точка), «_» (подчеркивание) и «-» (дефис); прочие символы, в том числе пробелы, не допускаются;
- имена элементов и атрибутов не могут начинаться с цифры, точки или дефиса (но могут начинаться с символа подчеркивания);
- имена элементов и атрибутов чувствительны к регистру (таким образом, элементы author и Author считаются различными);
- любой открывающий тег должен иметь парный ему закрывающий тег; при этом должна обеспечиваться правильная вложенность тегов (например, фрагмент <author><Author></author></Author> считается ошибочным); на комбинированный тег вида <имя /> данное правило не распространяется;
- между начальными символами тегов («<» для открывающего или комбинированного, «</» для закрывающего) и именем элемента не должно быть пробелов (перед символом «>» пробелы допускаются);
- один элемент не может иметь несколько атрибутов с одинаковыми именами (хотя может содержать несколько одноименных дочерних элементов);
- значения любых атрибутов должны заключаться в однотипные кавычки, при этом само значение атрибута не должно содержать кавычки данного типа;

- в числовых значениях в качестве разделителя всегда используется точка;
- для некоторых символов требуется использовать специальные обозначения (называемые *предопределенными символьными сущностями*): `&` для символа «&», `<` для символа «<», `>` для символа «>».

Кроме трех предопределенных символьных сущностей, перечисленных в последнем пункте правил, обеспечивающих корректность XML-документа, имеются еще две: `&apos;` для символа «'» (одинарная кавычка-апостроф), `&quot;` для символа «"» (двойная кавычка). Эти сущности обычно используются в тексте значений атрибутов (напомним, что эти значения должны заключаться в одинарные или двойные кавычки). Впрочем, без них почти всегда можно обойтись, поскольку, если значение атрибута содержит двойные кавычки, оно может быть заключено в одинарные кавычки, и наоборот. Например, оба приведенных далее значения атрибутов являются допустимыми: `"a'b'c"`, `'a"b"c'`.

Наряду с предопределенными символьными сущностями в XML можно использовать *нумерованные символьные сущности*, которые позволяют указать любой символ из набора Unicode с помощью его номера. Предусмотрены два формата для нумерованных символьных сущностей: `&#`*десятичный_номер*`;` и `&#x`*шестнадцатеричный_номер*`;`. Например, указать в тексте документа XML символ «&» (имеющий кодовый номер 38) можно тремя способами: `&`, `&#38;` или `&#x26;`.

Помимо основных правил, выполнение которых необходимо для того, чтобы любой XML-документ считался корректным (well-formed), на формат XML-документа могут накладываться дополнительные правила, вытекающие из предметной области, связанной с данным форматом. Например, при описании литературного произведения естественно предполагать, что оно имеет название, причем единственное, и автора (одного или многих); кроме того, это произведение должно быть отнесено к одному или нескольким жанрам (причем если это произведение не вполне вписывается в рамки какого-либо жанра, то можно дополнительно указать «вес» этого жанра с помощью атрибута `weight`). Любое произведение представлено на определенном языке; если же это переводное произведение, то для него можно дополнительно указать язык оригинала (единственный) и переводчиков (одного или нескольких). Учитывая подобные естественные предположения, можно потребовать, чтобы любой документ, описывающий литературные произведения и подобный приведенному выше, удовлетворял дополнительным правилам:

- любой элемент `title-info` должен содержать:
  - не менее одного элемента `author` и `genre`;

  - ровно один элемент `book-title` и `lang`;
  - не более одного элемента `src-lang` и `sequence`;
  - любое количество элементов `translator`;
- элемент `author` должен содержать ровно один элемент `last-name` и не более одного элемента `first-name` и `middle-name`;
- элементы `book-title` и `lang` должны иметь только текстовое содержимое, причем содержимое `book-title` может быть произвольным текстом, а содержимое `lang` должно совпадать с одним из стандартных имен, определенных для различных языков («ru» для русского, «en» для английского и т. д.);
- элемент `genre` может содержать дополнительный атрибут `weight`, значение которого должно представлять собой целое число из диапазона 1–100.

Все подобные дополнительные правила описываются на специальном языке и могут оформляться либо в виде *определения типа документа* (document type definition, DTD), либо в виде *схемы* (schema) XML-документа. Если XML-документ удовлетворяет требуемому DTD или схеме, то он считается не только корректным, но и *действительным*, или *валидным* (valid).

12.2. Объектные модели XML-документа

Все конструкции, входящие в правильно сформированный XML-документ (объявление, элемент, атрибут, текстовое содержимое), могут быть представлены классами, образующими *объектную модель документа* (Document Object Model, DOM). Можно говорить также о программном интерфейсе объектной модели документа – DOM API. Стандартной моделью DOM является модель W3C DOM XML API, предложенная консорциумом W3C – официальным разработчиком языка XML. Эта модель реализована в стандартной библиотеке .NET Framework (в пространстве имен `System.Xml`), однако ее использование требует от программиста значительных усилий. В качестве примера приведем фрагмент программы, формирующий первую часть документа, приведенного выше (в начале данной программы необходимо указать директиву `using System.Xml`):

```
XmlDocument d = new XmlDocument();
d.AppendChild(d.CreateXmlDeclaration("1.0", "windows-1251", null));
XmlElement bookDescriptions = d.CreateElement("описания-книг");
d.AppendChild(bookDescriptions);
XmlElement titleInfo = d.CreateElement("title-info");
XmlElement genre = d.CreateElement("genre");
XmlAttribute attr = d.CreateAttribute("match");
attr.InnerText = "90";
```

```
genre.Attributes.Append(attr);
genre.InnerText = "sf_fantasy";
titleInfo.AppendChild(genre);
XmlElement child = d.CreateElement("author");
XmlElement name = d.CreateElement("first-name");
name.InnerText = "Джон";
child.AppendChild(name);
name = d.CreateElement("middle-name");
name.InnerText = "Рональд Руэл";
child.AppendChild(name);
name = d.CreateElement("last-name");
name.InnerText = "Толкин";
child.AppendChild(name);
titleInfo.AppendChild(child);
child = d.CreateElement("book-title");
child.InnerText = "Возвращение Короля";
titleInfo.AppendChild(child);
child = d.CreateElement("lang");
child.InnerText = "ru";
titleInfo.AppendChild(child);
child = d.CreateElement("sequence");
attr = d.CreateAttribute("name");
attr.InnerText = "Властелин Колец";
child.Attributes.Append(attr);
attr = d.CreateAttribute("number");
attr.InnerText = "3";
child.Attributes.Append(attr);
titleInfo.AppendChild(child);
bookDescriptions.AppendChild(titleInfo);
d.Save("MyBooks.xml");
```

В результате в файл MyBooks.xml будет записан следующий текст в кодировке Windows-1251:

```
<?xml version="1.0" encoding="windows-1251"?>
<описания-книг>
  <title-info>
    <genre match="90">sf_fantasy</genre>
    <author>
      <first-name>Джон</first-name>
      <middle-name>Рональд Руэл</middle-name>
      <last-name>Толкин</last-name>
    </author>
    <book-title>Возвращение Короля</book-title>
    <lang>ru</lang>
    <sequence name="Властелин Колец" number="3" />
```

```
  </title-info>
</описания-книг>
```

Приведенный код программы сложен для восприятия, и по нему трудно определить вид формируемого XML-документа. Обработка полученного документа непосредственно в программе также является непростой задачей. Например, вывести содержимое документа или какого-либо его узла, указав связанный с ним объект в методе `WriteLine`, не удастся, поскольку при выполнении следующего фрагмента

```
Console.WriteLine(d);
Console.WriteLine(titleInfo);
```

будет выведен текст

```
System.Xml.XmlDocument
System.Xml.XmlElement
```

Если исправить фрагмент, обратившись к свойству `OuterXml`,

```
Console.WriteLine(d.OuterXml);
Console.WriteLine(titleInfo.OuterXml);
```

то в консольном окне будет выведен соответствующий текст без форматирования, например (при выполнении первого оператора):

```
<?xml version="1.0" encoding="windows-1251"?><описания-книг><title-info><genre m
atch="90">sf_fantasy</genre><author><first-name>Джон</first-name><middle-name>Po
нальд Руэл</middle-name><last-name>Толкин</last-name></author><book-title>Возвра
щение Короля</book-title><lang>ru</lang><sequence name="Властелин Колец" number=
"3" /></title-info></описания-книг>
```

Применение методов LINQ to Objects для перебора дочерних элементов также затруднено, так как дочерние элементы содержатся в свойстве `ChildNodes`, представляющем собой коллекцию типа `XmlNodeList`, в котором *не реализован* обобщенный интерфейс `IEnumerable<T>`:

```
var res = titleInfo.ChildNodes.Cast<XmlNode>().Select(e => e.Name);
foreach (var a in res)
  Console.WriteLine(a);
```

При выполнении данного фрагмента будут выведены имена всех дочерних элементов элемента `titleInfo`:

```
genre
author
book-title
lang
sequence
```

Учитывая сложность работы с W3C DOM API, разработчики версии .NET Framework 3.5 включили в нее альтернативную объектную модель для XML-документов, обладающую рядом преимуществ по сравнению со стандартной моделью W3C. Эта новая модель (для которой мы будем использовать обозначение *X-DOM*) вместе с дополнительными запросами,

оформленными в виде методов расширения класса `Extensions` из пространства имен `System.Xml.Linq`, образует интерфейс *LINQ to XML*.

Приведенный ниже фрагмент, формирующий тот же документ, что и рассмотренный ранее фрагмент для модели W3C DOM, весьма наглядно демонстрирует преимущества новой модели X-DOM:

```
XDocument d = new XDocument(
  new XDeclaration("1.0", "windows-1251", null),
  new XElement("описания-книг",
    new XElement("title-info",
      new XElement("genre",
        new XAttribute("match", "90"),
        "sf_fantasy"),
      new XElement("author",
        new XElement("first-name", "Джон"),
        new XElement("middle-name", "Рональд Руэл"),
        new XElement("last-name", "Толкин")),
      new XElement("book-title", "Возвращение Короля"),
      new XElement("lang", "ru"),
      new XElement("sequence",
        new XAttribute("name", "Властелин Колец"),
        new XAttribute("number", "3")))));
```

Заметим, что для вывода полученного документа на экран *в отформатированном виде* достаточно выполнить оператор `Console.WriteLine(d)`.

12.3. Основные классы модели X-DOM

В данном пункте описываются (в порядке наследования) основные классы, входящие в объектную модель X-DOM. При описании свойств и методов данных классов слова «предок» и «потомок» будут относиться не к иерархии наследования данных классов, а к *иерархии расположения объектов этих классов в дереве объектов, моделирующем XML-документ*; при описании иерархии наследования классов будет использоваться слово «наследник».

В описаниях классов основное внимание уделяется тому, как в модели X-DOM реализуются *связи* между различными компонентами XML-документа. Методы, связанные с такими «внешними» действиями, как разбор, загрузка и сохранение XML-документа или его элемента, описываются далее, в п. 12.6.1. Кроме того, в настоящем пункте очень кратко описываются средства, предназначенные для работы со *значениями* атрибутов и элементов (более подробно эти средства рассмотрены в п. 12.6.3).

Для любого неабстрактного класса модели X-DOM предусмотрен конструктор с параметром `other`, тип которого совпадает с типом конструируемого объекта. Подобные конструкторы позволяют создавать *копии* уже

существующих объектов. В дальнейшем, при описании конкретных классов, мы не будем повторно упоминать о наличии подобных конструкторов.

12.3.1. XObject

XObject – базовый тип в иерархии X-DOM; он является абстрактным классом и наследуется непосредственно от класса object. Его основными свойствами, передаваемыми всем своим классам-наследникам и доступными только для чтения, являются Document – ссылка на документ (типа XDocument) и Parent – ссылка на элемент-родитель (типа XElement). Кроме того, в классе XObject определены два события: Changed (наступает при изменении данного объекта или его объектов-потомков) и Changing (наступает непосредственно перед изменением данного объекта или его объектов-потомков). Прямыми наследниками класса XObject являются два класса: XAttribute и XNode.

12.3.2. XAttribute

XAttribute – класс для хранения информации об одном *атрибуте* элемента; он наследуется от класса XObject.

Его основными свойствами являются Name (имя атрибута, только для чтения) и Value (значение атрибута, доступно для чтения и записи). Свойство Name имеет тип XNode, совместимый по присваиванию с типом string; свойство Value имеет тип string. Атрибуты любого элемента организованы в последовательность типа IEnumerable<XAttribute>, причем в самом классе XAttribute предусмотрены два свойства типа XAttribute (только для чтения), обеспечивающие перемещение по этой последовательности: NextAttribute (следующий атрибут) и PreviousAttribute (предыдущий атрибут). Если следующий или предыдущий атрибут отсутствует или текущий атрибут не связан с элементом (т. е. его свойство Parent равно null), то соответствующее свойство равно null.

Для задания или изменения значения атрибута можно, помимо свойства Value, использовать метод SetValue типа void с единственным параметром value типа object. Особенности данного метода подробно описываются в п. 12.6.3.

Для удаления атрибута из списка атрибутов элемента XML достаточно вызвать для удаляемого атрибута метод Remove типа void без параметров (если свойство Parent удаляемого атрибута равно null, то данный метод возбуждает исключение).

Для создания объекта типа XAttribute предусмотрен конструктор с параметрами (XName name, object value). Параметр value обрабатывается аналогично одноименному параметру метода SetValue.

12.3.3. XNode

`XNode` – абстрактный базовый класс для всех компонентов XML-документа, кроме атрибутов. Подобные компоненты являются *узлами* дерева, порождающего XML-документ. К узлам дерева относятся не только его элементы, но и другие компоненты – текстовое содержимое элементов, комментарии и т. д. Все узлы, имеющие общего родителя (*узлы-братья*, англ. sibling nodes), хранятся в последовательности типа `IEnumerable<XNode>`, и в классе `XNode` предусмотрены методы и свойства для работы с элементами последовательности, содержащей узел, для которого вызывается данное свойство или метод (для краткости будем называть такой узел self-узлом). Прежде всего, это два свойства типа `XNode` (только для чтения), обеспечивающие перемещение по этой последовательности: `NextNode` (следующий узел) и `PreviousNode` (предыдущий узел). Как и в случае аналогичных свойств класса `XAttribute`, если следующий или предыдущий узел для self-узла отсутствует или self-узел не имеет элемента-родителя (т. е. его свойство `Parent` равно `null`), то соответствующее свойство равно `null`. Следует учитывать, что фактически узлы-братья хранятся в виде *односвязного списка*, поэтому значение свойства `NextNode` вычисляется гораздо быстрее, чем значение свойства `PreviousNode`.

Кроме данных свойств в классе `XNode` предусмотрено несколько методов, связанных с последовательностью узлов-братьев:

- `IsAfter(XNode node)` – метод типа `bool`; возвращает `true`, если self-узел расположен в последовательности после узла `node`, в противном случае (в частности, если self-узел не имеет родителя) возвращает `false`;
- `AddAfterSelf(object content)` и `AddAfterSelf(params object[] content)` – методы типа `void`; добавляют непосредственно после self-узла один или несколько новых узлов (если self-узел не имеет родителя, то возбуждается исключение `InvalidOperationException`);
- `NodesAfterSelf()` – метод типа `IEnumerable<XNode>`; возвращает последовательность узлов-братьев, расположенных в последовательности после self-узла;
- `ElementsAfterSelf()` – метод типа `IEnumerable<XElement>`; возвращает последовательность элементов-братьев, расположенных в последовательности после self-узла; имеется перегруженный вариант данного метода с параметром `name` типа `XName`, оставляющий в результирующей последовательности только элементы с именем `name`.

Имеются также «парные» методы, в которых слово «`After`» заменено словом «`Before`». Эти парные методы работают аналогичным образом, но анализируют, добавляют или возвращают не последующие, а *предшест-*

вующие узлы.

Класс XNode содержит также методы для удаления или замены self-узла:

- void Remove() – удаляет self-узел из последовательности его узлов-братьев;
- void ReplaceWith(object content) и void ReplaceWith(params object[] content) – заменяют self-узел в последовательности его узлов-братьев на один или несколько узлов, указанных в качестве параметра content.

Если свойство Parent self-узла равно null, то вызов методов Remove или ReplaceWith приводит к возбуждению исключения InvalidOperationException.

Отметим также метод Ancestors типа IEnumerable<XElement>, возвращающий последовательность всех элементов, являющихся *предками* self-узла, начиная с ближайшего предка, т. е. *родителя* (сам self-узел в эту последовательность не входит, потому что его тип может отличаться от типа XElement). Имеется перегруженный вариант метода Ancestors с одним параметром name типа XName; при наличии этого параметра в возвращаемую последовательность включаются только те предки self-узла, которые имеют имя name.

Таким образом, в классе XNode предусмотрена функциональность, связанная не только с данным узлом XML-документа, но и с его узлами-*братьями* и элементами-*предками*.

12.3.4. XText

XText – простейший узел XML-документа; представляет собой обычный *текст*. Наследуется от класса XNode, содержит свойство Value типа string, доступное как для чтения, так и для записи.

Для создания объекта типа XText предусмотрен конструктор с параметром value типа string. Как правило, для создания объекта типа XText не требуется вызывать конструктор, так как если в качестве очередного узла XML-дерева указывается строковое выражение (или выражение, которое может быть преобразовано к строковому), то это выражение автоматически преобразуется к типу XText с соответствующим значением свойства Value.

Важно иметь в виду, что при работе со свойством Value *не следует* в качестве значений каких-либо специальных символов использовать их предопределенные или нумерованные символьные сущности. Например, если узел XText должен содержать значение «&», то это значение надо задавать обычным образом: "&" (при визуализации XML-дерева с данным уз-

лом это значение будет автоматически преобразовано в предопределенную сущность &):

```
var e = new XElement("demo", "&");
Console.WriteLine(e); // <demo>&amp;amp;</demo>
```

Приведем еще один пример:

```
var e = new XElement("demo", "<\">");
Console.WriteLine(e); // <demo>&amp;lt;"&amp;gt;</demo>
```

Обратите внимание на то, что в качестве двойных кавычек в полученном тексте XML-элемента используется обычный символ «"», а не его предопределенная сущность, поскольку это не приводит к каким-либо неоднозначностям при анализе данного элемента (в то же время по правилам языка C# символ «"» в тексте строки необходимо *экранировать*, указывая перед ней обратную косую черту).

12.3.5. XCData

`XCData` – узел XML-документа с текстовым содержимым, которое сохраняется в документе *буквально*, без привлечения сущностей-заместителей. Наследуется от класса `XText`. Подобно классу `XText` имеет свойство `Value` типа `string`, доступное как для чтения, так и для записи, и конструктор с параметром `value` типа `string`. При записи данного узла в XML-документ используются открывающий и закрывающий маркеры специального вида:

```
var e = new XElement("demo", new XCData("<\">"));
Console.WriteLine(e); // <demo><![CDATA[<">]]></demo>
```

Если не учитывать особый формат представления узлов типа `XCData` в XML-документе, работа с ними ничем не отличается от работы с обычными текстовыми узлами типа `XText`.

12.3.6. XComment

`XComment` – узел XML-документа, представляющий собой *комментарий*. Наследуется от класса `XNode`. Содержит свойство `Value` типа `string`, доступное как для чтения, так и для записи. Имеет конструктор с параметром `value` типа `string`.

Поскольку текст комментария игнорируется при разборе XML-документа, он может содержать практически любые символы и их комбинации, которые никогда не будут заменяться на предопределенные или нумерованные сущности:

```
var e = new XElement("demo", new XComment("<\">"));
Console.WriteLine(e); // <demo><!--<">--></demo>
```

Единственное исключение делается при попытке указать в тексте комментария завершающий маркер комментария `-->`; в этом случае между соседними дефисами автоматически добавляется символ пробела: `- ->`.

12.3.7. XProcessingInstruction

XProcessingInstruction – узел XML-документа, содержащий *инструкции обработки* (processing instructions) – вспомогательные данные, не относящиеся к содержимому XML-документа, но доступные для использования каким-либо *процессором XML*, т. е. программой, предназначенной для анализа или преобразования XML-документов.

Инструкции обработки в XML-документе обрамляются маркерами вида <? и ?> и состоят из двух элементов, разделенных пробелом:

- target – имя получателя инструкции (обязательный элемент, который должен удовлетворять правилам именования элементов XML);
- data – данные для обработки (необязательный элемент, являющийся произвольной текстовой строкой).

Класс XProcessingInstruction наследуется от класса XNode и имеет свойства Target и Data, доступные как для чтения, так и для записи. Его конструктор также принимает два параметра target и data, которые не могут быть равны null (параметр data может быть произвольной строкой, в том числе пустой, а параметр target должен удовлетворять правилам именования элементов XML). Пример:

```
var e = new XElement("demo", new XProcessingInstruction("SetMark",
  "Value='&' Count=1"));
Console.WriteLine(e); // <demo><?SetMark Value='&' Count=1?></demo>
```

Если свойство Data равно пустой строке, то пробел между именем получателя инструкции и закрывающим маркером ?> не указывается:

```
var e = new XElement("demo", new XProcessingInstruction("SetMark", ""));
Console.WriteLine(e); // <demo><?SetMark?></demo>
```

12.3.8. XContainer

XContainer – абстрактный класс для реализации *XML-контейнеров* – узлов дерева XML, которые могут содержать другие узлы (*узлы-потомки*). Наследуется от класса XNode. В дополнение к унаследованным от XNode свойствам содержит свойства FirstNode и LastNode типа XNode (только для чтения), возвращающие первый и последний дочерний узел self-узла соответственно (*дочерним узлом* называется любой узел-потомок, родителем которого, т. е. *ближайшим* предком, является self-узел).

Для добавления дочерних узлов класс XContainer имеет методы Add и AddFirst, которые позволяют добавлять один или несколько дочерних узлов в конец или начало списка дочерних узлов соответственно; оба метода имеют один параметр content, который может иметь как тип object, так и тип params object[].

Для удаления или замены всех дочерних узлов класс XContainer имеет методы RemoveNodes (без параметров; удаляет всех дочерние узлы) и Repla-

ceNodes (с параметром content типа object или params object[]; заменяет все дочерние узлы на один или несколько узлов, указанных в параметре content).

Имеется несколько методов, позволяющих получить различные варианты последовательностей, связанных с узлами-потомками данного XML-контейнера:

- Nodes() и DescendantNodes() – возвращают последовательность типа IEnumerable<XNode>; в случае метода Nodes последовательность содержит дочерние узлы, в случае DescendantNodes последовательность содержит узлы-потомки (любого уровня вложенности); порядок возвращаемых узлов соответствует порядку, в котором они расположены в XML-документе;
- Elements() и Descendants() – возвращают последовательность типа IEnumerable<XElement>, в случае метода Elements последовательность содержит дочерние элементы, в случае Descendants – элементы-потомки (любого уровня вложенности); имеются перегруженные варианты этих методов с параметром name типа XName; при наличии этого параметра в возвращаемую последовательность включаются только те элементы-потомки self-узла, которые имеют имя name.

Метод Element(XName name) возвращает первый дочерний элемент XML-контейнера с указанным именем name (или null, если ни одного требуемого элемента не найдено).

12.3.9. XElement

XElement – основной класс модели X-DOM, предназначенный для хранения *именованного элемента* XML-документа. Наследуется от XContainer. В дальнейшем будем для краткости называть тот элемент, для которого вызывается описываемое свойство или метод, self-элементом.

Это единственный вид узла XML-дерева, который может иметь атрибуты, поэтому класс XElement содержит набор свойств и методов, предназначенных для работы с последовательностью атрибутов (все свойства доступны только для чтения):

- HasAttributes – свойство типа bool; равно true, если self-элемент содержит хотя бы один атрибут;
- FirstAttribute и LastAttribute – свойства типа XAttribute, возвращают первый и, соответственно, последний атрибут из последовательности атрибутов self-элемента; если элемент не содержит атрибутов, возвращают null;
- Attribute(XName name) – метод типа XAttribute, возвращающий атрибут self-элемента с указанным именем name (или null, если атрибут с указанным именем не существует);

- `Attributes()` и `Attributes(XName name)` – методы типа `IEnumerable<XAttribute>`, возвращающие последовательность атрибутов self-элемента; если указан параметр `name`, то последовательность содержит только атрибут с указанным именем (поскольку атрибуты XML-элемента не могут иметь одинаковые имена, второй вариант метода всегда возвращает последовательность, содержащую не более одного атрибута);
- `RemoveAttributes()` – метод типа `void`, удаляющий все атрибуты self-элемента;
- `ReplaceAttributes(object content)` и `ReplaceAttributes(params object[] content)` – методы типа `void`, заменяющие все атрибуты self-элемента на один или несколько атрибутов, указанных в параметре `content` (в данном параметре можно указывать не только атрибуты, но и *узлы*, которые будут добавлены в последовательность дочерних узлов self-элемента);
- `SetAttributeValue(XName name, object value)` – метод типа `void`, позволяющий создавать, изменять и удалять атрибуты self-элемента. Если параметр `value` равен `null`, то атрибут с именем `name` удаляется; в противном случае, если атрибут с именем `name` отсутствует, то он создается, иначе его значение заменяется на `value` (в качестве `value` можно указывать любой объект, кроме объектов-наследников класса `XObject`; по поводу преобразования этих объектов к строковому типу см. п. 12.6.3).

Следует отметить, что для добавления атрибута к элементу, кроме метода `SetAttributeValue`, можно использовать метод `Add`, поскольку в классе `XElement` этот метод может использоваться не только для добавления дочерних узлов, но и для добавления атрибутов (в то же время метод `AddFirst` *не позволяет* добавлять к элементу атрибуты).

Класс `XElement` имеет свойство `Value` типа `string`, доступное для чтения и записи. Это свойство возвращает объединенное текстовое содержимое самого self-элемента и его потомков (всех уровней). При этом предопределенные и нумерованные символьные сущности, используемые в XML для обозначения символов, заменяются на сами эти символы. Если же присвоить свойству `Value` новое строковое значение, то все дочерние узлы будут заменены на единственный дочерний узел типа `XText`, содержащий указанную строку. Свойству `Value` нельзя присвоить значение `null`. Если у элемента и его потомков отсутствует текстовое содержимое (в частности, если элемент представлен единственным комбинированным тегом вида `<name />`), то свойство `Value` возвращает *пустую строку*.

Метод `SetValue(object value)` типа `void` также позволяет изменить свойство `Value`, при этом параметр `value` может иметь любой тип (кроме

типов-наследников класса XObject, при использовании которых возбуждается исключение ArgumentException). Перед присваиванием свойству Value параметр value преобразуется к строковому типу (см. п. 12.6.3).

Свойство IsEmpty типа bool (только для чтения) позволяет распознать элемент, представленный комбинированным тегом; в этом случае оно будет равно true (наличие атрибутов в расчет не принимается). Например, для элемента свойство IsEmpty будет равно true, а для элемента <a> это свойство будет равно false.

Для работы с дочерними *элементами* в классе XElement предусмотрены следующие дополнительные свойства и методы:

- HasElements – свойство типа bool (только для чтения); равно true, если self-элемент содержит хотя бы один дочерний элемент;
- SetElementValue(XName name, object value) – метод типа void, позволяющий создавать, изменять и удалять дочерние элементы self-элемента. Если параметр value равен null, то дочерний элемент с именем name удаляется; в противном случае, если дочерний элемент с именем name отсутствует, то он создается и его свойству Value присваивается значение value, иначе его свойство Value заменяется на value (в качестве параметра value можно указывать любой объект – см. п. 12.6.3).

Есть также методы класса XElement, имеющие отношение ко всему содержимому элемента: и его атрибутам, и его дочерним узлам:

- RemoveAll() – метод типа void, удаляющий у self-элемента все его атрибуты и дочерние узлы;
- ReplaceAll(object content) и ReplaceAll(params object[] content) – методы типа void, заменяющие все атрибуты и дочерние узлы self-элемента на один или несколько атрибутов и/или дочерних узлов, указанных в параметре content.

Наконец, упомянем модификации унаследованных методов, возвращающих последовательности узлов или элементов. Имена этих модификаций оканчиваются на «AndSelf»; все они отличаются от своих исходных вариантов тем, что включают в возвращаемую последовательность сам self-элемент:

- AncestorsAndSelf() и AncestorsAndSelf(XName name) – модификация метода Ancestors (см. описание класса XNode, п. 12.3.3);
- DescendantNodesAndSelf() – модификация метода DescendantNodes (см. описание класса XContainer, п. 12.3.8);
- DescendantsAndSelf() и DescendantsAndSelf(XName name) – модификация метода Descendants (см. описание класса XContainer).

Для создания объекта типа XElement предусмотрено несколько конструкторов со следующими наборами параметров (параметр name определяет имя создаваемого элемента):

- XElement(XName name) – создает элемент без содержимого;
- XElement(XName name, object content) и XElement(XName name, params object[] content) – создает элемент с содержимым content (одним или несколькими дочерними узлами и/или атрибутами).

Если требуется представить элемент в виде одного комбинированного тега, необходимо, чтобы при его создании содержимое content не включало данных, отличных от null (если задать в содержимом content хотя бы один параметр, отличный от null, например пустую строку, то элемент будет отображаться в виде открывающего и закрывающего тега). Элемент отображается в виде комбинированного тега также в случае, если для него вызван метод RemoveNodes или RemoveAll.

12.3.10. XDocument

XDocument – класс, предназначенный для хранения «оболочки» дерева XML. Наследуется от XContainer. В любом XML-документе имеется единственный объект типа XDocument, причем любой объект-наследник XObject имеет свойство Document, ссылающееся на документ, с которым связан данный объект. По сравнению с классом XElement класс XDocument содержит немного собственных свойств и методов. Перечислим основные из них:

- Declaration – свойство типа XDeclaration, доступное как для чтения, так и для записи; предназначено для хранения объявления XML (см. п. 12.4.1);
- Root – свойство типа XElement, доступное только для чтения; содержит корневой элемент XML-документа.

Заметим, что объект XDocument не является родителем своего корневого элемента, поскольку родителем элемента (как и любого узла дерева XML) может быть только элемент, т. е. объект типа XElement. Поэтому свойство Parent корневого элемента полагается равным null.

Для создания объекта типа XDocument предусмотрено несколько конструкторов со следующими наборами параметров:

- XDocument() – создает «пустой» XML-документ;
- XDocument(params object[] content) – создает XML-документ с содержимым content (в наборе параметров должно быть не более одного объекта типа XElement, определяющего корневой элемент XML-документа, однако может присутствовать, например, любое количество XML-комментариев);

- `XDocument(XDeclaration declaration, params object[] content)` – действует подобно ранее рассмотренному конструктору, но дополнительно позволяет определить *объявление XML* (параметр `declaration`).

Из примера использования модели W3C DOM (см. п. 12.2) нетрудно заметить, что объект типа `XmlDocument` является основным, «цементирующим» компонентом документа XML. В частности, в модели W3C DOM любой компонент XML-документа создается с помощью одного из методов объекта типа `XmlDocument`. Это означает, что в программе нельзя создать фрагмент XML-документа, не связанный с объектом типа `XmlDocument`, что в ряде случаев является неудобным ограничением.

Модель X-DOM свободна от указанного ограничения. Хотя в ней и предусмотрен класс `XDocument`, создавать фрагменты XML-документа можно без его участия.

12.3.11. XDocumentType

`XDocumentType` – специализированный узел XML-документа, содержащий *объявление типа документа* (type document declaration; не следует путать объявление типа документа с объявлением XML, для которого предусмотрен особый класс `XDeclaration` – см. п. 12.4.1). Наследуется от `XNode`. Этот узел может содержаться только в прологе XML-документа, поэтому его можно включать только в XML-контейнер типа `XDocument`. Данный класс связан со специальными возможностями XML-документов, которые мы не будем рассматривать.

12.4. Вспомогательные классы модели X-DOM

12.4.1. XDeclaration

`XDeclaration` – вспомогательный класс модели X-DOM, предназначенный для хранения *объявления XML* (XML declaration). Наследуется от класса `object`, не содержит дополнительных методов. Для настройки трех компонентов объявления класс `XDeclaration` имеет три свойства типа `string`, доступных как для чтения, так и для записи:

- `Version` – свойство, определяющее версию стандарта XML; при сохранении XML-документа в качестве версии стандарта XML *всегда* используется значение `"1.0"`;
- `Encoding` – свойство, определяющее кодировку XML-документа, которая учитывается при записи документа на диск; в качестве примеров кодировок укажем `"utf-8"` и `"windows-1251"` (кодировка `"utf-8"` используется по умолчанию);

- `Standalone` – свойство, которое определяет, является ли данный XML-документ *автономным* или он связан с какими-либо внешними документами; данное свойство может принимать только два непустых допустимых значения: `"yes"` (автономный документ) и `"no"`; в случае любого другого значения (в частности, пустой строки или `null`) компонент `Standalone` в объявлении сохраненного XML-документа не указывается.

Конструктор объекта `XDeclaration` принимает три строковых параметра – значения компонентов объявления `version`, `encoding` и `standalone` соответственно. Если какой-либо параметр является пустой строкой или равен `null`, то при сохранении XML-документа в качестве соответствующего компонента объявления будет использовано значение по умолчанию (хотя соответствующие свойства объекта `XDeclaration` будут содержать значения, указанные в конструкторе, и эти же значения будут использованы при преобразовании объекта `XDeclaration` к строковому представлению с помощью метода `ToString`).

12.4.2. XName

`XName` – вспомогательный запечатанный класс, предназначенный для хранения *имен элементов* XML-документа. Наследуется от класса `object`, реализует интерфейс `IEquitable<XName>` (таким образом, имена могут сравниваться на равенство). К типу `XName` может быть неявно преобразована строка типа `string`; обратное преобразование должно выполняться явно, с помощью метода `ToString`. В классе `XName` предусмотрены специальные механизмы для поддержки имен, включающих *пространства имен XML* (XML namespaces, см. п. 12.6.4). В частности, имеются три свойства (только для чтения), позволяющие получить различные компоненты имени:

- `LocalName` (типа `string`) – локальное имя;
- `Namespace` (типа `XNamespace`) – необязательное пространство имен, присоединяемое к локальному имени для получения полного имени;
- `NamespaceName` (типа `string`) – имя пространства имен.

Полное имя, связанное с объектом `XName` (qualified name), может быть получено с помощью метода `ToString`.

12.4.3. XNamespace

`XNamespace` – вспомогательный запечатанный класс, предназначенный для работы с *пространствами имен XML*. В качестве имени пространства имен XML обычно используется *универсальный идентификатор ресурса* (Uniform Resource Identifier, URI), связанный с той организацией, которая

разработала спецификацию соответствующего XML-документа (хотя допустимым является любое строковое выражение).

Класс `XNamespace` наследуется от класса `object`. К типу `XNamespace` может быть неявно преобразована строка типа `string`; обратное преобразование должно выполняться явно, с помощью метода `ToString`.

Для класса `XNamespace` определена операция «+», применяемая к операндам типа `XNamespace` (первый операнд – пространство имен) и `string` (второй операнд – локальное имя). Результатом этой операции будет полное имя XML-элемента (типа `XName`).

Класс `XNamespace` имеет единственное экземплярное свойство `NamespaceName` типа `string`, возвращающее имя пространства имен, и три статических свойства типа `XNamespace`, возвращающих специальные виды пространств имен:

- `None` – пустое пространство имен (его имя – пустая строка);
- `Xml` – пространство имен, соответствующее XML URI (его имя – `"http://www.w3.org/XML/1998/namespace"`);
- `Xmlns` – пространство имен, соответствующее xmlns URI (его имя – `"http://www.w3.org/2000/xmlns/"`).

Все перечисленные свойства являются свойствами только для чтения.

Дополнительные сведения об использовании пространств имен приводятся далее, в п. 12.6.4.

12.5. Методы расширения LINQ to XML

При рассмотрении классов модели X-DOM мы уже познакомились с набором методов, которые можно рассматривать как запросы, специфические для интерфейса LINQ to XML. К таким методам можно отнести все методы, возвращающие последовательности `IEnumerable<T>`. Приведем список этих методов, сгруппировав его по «иерархической природе» возвращаемых последовательностей. Если метод может содержать необязательный параметр `name`, то он указывается в квадратных скобках. В правой части описания указывается тип объекта, в котором впервые определен этот метод, и (после символа «→») тип возвращаемой последовательности.

(1) Компоненты одного уровня (узлы-братья и элементы-братья):

`NodesAfterSelf()`	`XNode → IEnumerable<XNode>`
`NodesBeforeSelf()`	`XNode → IEnumerable<XNode>`
`ElementsAfterSelf([XName name])`	`XNode → IEnumerable<XElement>`
`ElementsBeforeSelf([XName name])`	`XNode → IEnumerable<XElement>`

(2) Элементы-предки:

`Ancestors([XName name])`	`XNode → IEnumerable<XElement>`
`AncestorsAndSelf([XName name])`	`XElement → IEnumerable<XElement>`

(3) Дочерние компоненты (атрибуты, узлы и элементы):

```
Attributes([XName name])              XElement → IEnumerable<XAttribute>
Nodes()                                  XContainer → IEnumerable<XNode>
Elements([XName name])                XContainer → IEnumerable<XElement>
```

(4) Компоненты-потомки любого уровня вложенности (узлы и элементы):

```
DescendantNodes()                        XContainer → IEnumerable<XNode>
DescendantNodesAndSelf()                   XElement → IEnumerable<XNode>
Descendants([XName name])              XContainer → IEnumerable<XElements>
DescendantsAndSelf([XName name])        XElement → IEnumerable<XElements>
```

Для всех перечисленных методов, *кроме методов группы (1)*, в интерфейсе LINQ to XML определены одноименные методы расширения, которые вызываются не для отдельного объекта типа `T`, а для *последовательности* `IEnumerable<T>`. Все методы расширения являются статическими методами класса `Extensions` из пространства `System.Xml.Linq`. Благодаря методам расширения можно обрабатывать наборы узлов, атрибутов или элементов в одном запросе. Методы расширения могут возвращать последовательность, содержащую одинаковые элементы; для удаления одинаковых элементов можно использовать стандартный запрос `Distinct`.

В качестве примера приведем описание методов расширения, соответствующих методам группы (2):

```
Ancestors([XName name])
                  IEnumerable<T> where T : XNode → IEnumerable<XElement>
AncestorsAndSelf([XName name])
                           IEnumerable<XElement> → IEnumerable<XElement>
```

В классе `Extensions` определены еще два метода расширения:

```
InDocumentOrder()               IEnumerable<XNode> → IEnumerable<XNode>
```

Данный метод изменяет порядок исходной последовательности узлов некоторого XML-документа, располагая их в порядке следования в этом документе.

```
Remove()                                  IEnumerable<XAttribute> → void
Remove()                             IEnumerable<T> where T : XNode → void
```

Перегруженные варианты данного метода удаляют из документа атрибуты или узлы, содержащиеся в исходной последовательности. В отличие от всех остальных методов расширения метод `Remove` имеет тип `void`. Приведем пример использования перегруженного метода `Remove` для удаления из XML-элемента `src` всех комментариев:

```
src.DescendantNodes().OfType<XComment>().Remove();
```

12.6. Дополнительные возможности технологии LINQ to XML

12.6.1. Загрузка, разбор и сохранение элементов и документов XML

Операции, связанные с загрузкой, разбором и сохранением, предусмотрены как для всего дерева XML (представленного в модели X-DOM классом `XDocument`), так и для отдельного элемента XML (представленного классом `XElement`).

Для *загрузки* предназначен статический метод `Load` классов `XDocument` и `XElement`; среди его перегруженных вариантов отметим вариант с параметром типа `string` – именем файла – и вариант с параметром типа `TextReader` – читающим потоком, который может быть связан как с текстовым файлом, так и со строкой в памяти. Аналогичные параметры (имя файла или пишущий поток) предусмотрены и для экземплярного метода `Save` этих же классов, предназначенного для *сохранения* документа или элемента XML. При сохранении можно указать дополнительный параметр `SaveOptions.DisableFormatting`, отменяющий автоматическое форматирование сохраняемого документа или элемента XML.

Для *преобразования в строку* документа/элемента XML достаточно использовать метод `ToString`, в котором также можно использовать необязательный параметр `SaveOptions.DisableFormatting`. Для обратного преобразования, т. е. *разбора строки*, в которой содержится документ/элемент XML, предназначен статический метод `Parse` классов `XDocument` и `XElement` с параметром-строкой, содержащей текст для разбора.

При загрузке или разборе документа/элемента XML выполняется проверка того, что он правильно сформирован, и в случае, если это не так, возбуждается исключение `XmlException`.

При загрузке или разборе документа можно указать дополнительные опции (в виде значений перечисления `LoadOptions`, объединенных операцией «|»). Из возможных значений перечисления `LoadOptions` упомянем `PreserveWhitespace`, сохраняющее незначащие пробелы документа/элемента XML при его загрузке (по умолчанию все подобные пробелы удаляются). Чтобы после обработки документа незначащие пробелы были сохранены, необходимо использовать уже упомянутый ранее параметр `SaveOptions.DisableFormatting`.

12.6.2. Функциональное конструирование дерева XML

Существенное упрощение действий, необходимых для конструирования дерева XML в модели X-DOM, достигается за счет механизма *функционального конструирования*, в котором все дерево может быть построено в одном выражении. Эта возможность обеспечивается с помощью методов, принимающих параметр типа `params object[]`. В качестве такого пара-

метра можно передавать не только массив данных произвольного типа, но и набор данных, разделяя их запятыми, что позволяет структурировать полученный программный код в соответствии с содержимым создаваемого XML-документа (см. пример в п. 12.2). Кроме того, в качестве данного параметра можно передавать последовательность `IEnumerable<T>`, полученную в результате выполнения некоторого запроса. Эта возможность *проецирования последовательностей* в дерево XML является еще одним аспектом технологии LINQ to XML, дополняющим рассмотренные ранее возможности LINQ to XML, связанные с извлечением и удалением последовательностей из XML-документов посредством соответствующих запросов.

Приведем список методов, принимающих параметр `content` типа `params object[]` (все они, за исключением конструкторов, имеют тип `void`):

`AddAfterSelf(params object[] content)`
`AddBeforeSelf(params object[] content)`
`ReplaceWith(params object[] content)`
	XNode

`Add(params object[] content)`
`AddFirst(params object[] content)`
`ReplaceNodes(params object[] content)`
	XContainer

`ReplaceAttributes(params object[] content)` `ReplaceAll(params object[] content)`	**XElement**

`XElement(XName name, params object[] content)`
`XDocument(params object[] content)`
`XDocument(XDeclaration declaration, params object[] content)`
	конструктор

В качестве элементов набора `content` могут выступать данные любого типа (для некоторых методов имеются исключения, описанные ниже). При этом каждый элемент набора обрабатывается в зависимости от его фактического типа с последовательным применением следующих правил:

- если элемент равен `null`, то он игнорируется;
- если элемент является наследником `XObject` (но не является объектом типа `XAttribute`), то он добавляется без изменений в соответствующую последовательность узлов дерева/элемента XML (например, в случае использования в методе `ReplaceAll` или `ReplaceAttributes` – в последовательность узлов self-элемента);
- если элемент имеет тип `XAttribute`, то для конструктора класса `XElement` и его методов `ReplaceAttributes` и `ReplaceAll` выполняется добавление этого элемента в последовательность атрибутов; для остальных методов, а также конструктора класса `XDocument` возбуждается исключение;
- если элемент имеет тип `string`, то он неявно преобразуется к объекту `XText` с указанным текстовым содержимым, после чего добавляется в соответствующую последовательность узлов дерева XML;

- если тип элемента отличен от `string`, но при этом реализует интерфейс `IEnumerable` или `IEnumerable<T>`, т. е. элемент представляет собой *последовательность*, то выполняется перебор всех элементов этой последовательности с учетом приведенных здесь правил;
- если ни одно из вышеперечисленных условий не выполняется, то элемент преобразуется к своему строковому представлению, которое в дальнейшем обрабатывается как элемент типа `string` (преобразование выполняется по тем же правилам, что и аналогичное преобразование параметра `value` в методах `SetValue`, описываемое далее в п. 12.6.3).

При добавлении к новому элементу существующего узла или атрибута, имеющего непустое свойство `Parent` (т. е. уже имеющего родителя), всегда выполняется *глубокое копирование (клонирование)* – не просто копируется ссылка на существующий объект, а создаются новые объекты-копии как копируемого объекта, так и всех его потомков. Это обеспечивает корректное изменение свойств `Parent` и позволяет избежать нежелательных побочных эффектов.

12.6.3. Работа со значениями атрибутов и элементов

Напомним, что у классов `XAttribute` и `XElement` имеется свойство `Value`, позволяющее получить значение соответствующего атрибута или элемента и доступное как для чтения, так и для записи. Свойства `Value` имеют тип `string`, однако нередко в качестве значений атрибутов и элементов приходится использовать данные других типов, например числовых, логических или связанных с датами или промежутками времени (`DateTime` и `TimeSpan` – см. главу 2). При этом возникает необходимость в корректном преобразовании этих данных к их строковому представлению, соответствующему требованиям спецификации XML. Например, логические значения должны записываться в нижнем регистре (`"true"` и `"false"`), а в представлении дробных чисел с качестве десятичного разделителя должна использоваться *точка*. Не менее актуальной является и обратная задача: преобразование значения атрибута или элемента к типу данных, отличному от `string`. Эта последняя задача осложняется еще тем обстоятельством, что заранее неизвестно, существует ли в анализируемом документе атрибут или элемент с требуемым именем.

В модели X-DOM для решения перечисленных задач имеется удобная и гибкая система методов и преобразований.

В то время как свойству `Value` можно присвоить только значение типа `string`, метод `SetValue` с параметром `value` типа `object` позволяет передавать в качестве значения данные *любых типов*; при этом для большинства типов в качестве значения атрибута будет использоваться значение, возвра-

щаемое методом ToString параметра value. Если параметр value имеет тип float, double, decimal, bool, DateTime, TimeSpan, то его значение преобразуется к специальному строковому представлению, удовлетворяющему спецификации языка XML. Значение value, указываемое в методе SetValue, не может иметь тип, унаследованный от класса XObject (в этом случае возбуждается исключение ArgumentException).

Методы SetAttributeValue и SetElementValue с параметрами name типа XName и value типа object, предусмотренные в классе XElement, позволяют настраивать свойства атрибутов и дочерних узлов self-элемента; при этом действуют те же правила по преобразованию параметра value, что и для метода SetValue. Кроме того, с помощью этих методов можно быстро *добавлять* атрибуты и дочерние элементы (добавление выполняется автоматически, если атрибут или элемент с именем name не найден) и *удалять* их из списка атрибутов и дочерних элементов self-элемента (для этого достаточно в качестве параметра value указать значение null).

Возможность указания данных различных типов в качестве значения атрибута или элемента предусмотрена и в *конструкторах* соответствующих классов. Методы AddBeforeSelf, AddAfterSelf, Add и AddFirst также обладают подобной возможностью.

Для решения обратной задачи – преобразования значения атрибута или элемента из строкового представления к его исходному типу – в модели X-DOM используется *операция приведения к соответствующему типу*, применяемая непосредственно к объекту типа XAttribute или XElement. Для объектов типа XAttribute или XElement можно выполнять приведение ко всем стандартным числовым типам, типам bool, DateTime, TimeSpan, string, а также к Nullable-вариантам всех перечисленных размерных типов (см. п. 7.3). Возможность приведения к Nullable-типу полезна, если заранее неизвестно, существует ли атрибут или дочерний элемент с требуемым именем.

Рассмотрим несколько примеров.

Предположим, что некоторые дочерние элементы элемента src имеют атрибут mark со значением, приводимым к типу int, а некоторые не имеют подобного атрибута. Для того чтобы найти среднее значение всех атрибутов mark и записать его в качестве значения нового элемента res, достаточно выполнить следующий оператор:

```
var res = new XElement("res", src.Elements()
          .Select(e => (int?)e.Attribute("mark")).Average());
```

Для записи результата в обычную переменную следует учесть, что метод Average, примененный к последовательности Nullable-элементов, также возвращает Nullable-результат:

```
double? avr = src.Elements().Select(e => (int?)e.Attribute("mark"))
```

```
                .Average();
```

Если при обработке дочерних элементов, вместо того чтобы игнорировать элементы с отсутствующим атрибутом `mark`, необходимо считать, что для этих элементов атрибут `mark` имеет *значение по умолчанию*, равное, например, нулю, то для вычисления среднего значения всех атрибутов надо изменить предыдущий вариант следующим образом:

```
var res = new XElement("res", src.Elements()
          .Select(e => (int?)e.Attribute("mark") ?? 0).Average());
```

В данном случае, благодаря операции `??`, все значения `null` будут заменены на значения 0. При этом, в частности, результат метода `Average` уже не будет иметь Nullable-тип:

```
double avr = src.Elements().Select(e => (int?)e.Attribute("mark") ?? 0)
                .Average();
```

Возможность приведения к типу `string` также может быть использована для корректной обработки отсутствующих данных. Предположим, что некоторые дочерние элементы элемента `src` имеют атрибут `info`, а некоторые не имеют. Требуется преобразовать все дочерние элементы таким образом, чтобы значения их атрибутов были добавлены к их текстовому содержимому, а сами атрибуты удалены. Задача может быть решена следующим образом:

```
foreach (var e in src.Elements())
  e.SetValue(e.Value + (string)e.Attribute("info"));
(from e in src.Elements() select e.Attribute("info")).Remove();
```

Заметим, что удалить только что обработанный атрибут непосредственно в цикле `foreach` с помощью выражения `e.Attribute("info").Remove()` *нельзя*, так как при отсутствии данного атрибута метод `Attribute` вернет `null`, что приведет к возбуждению исключения при попытке последующего вызова метода `Remove`. С другой стороны, в цикле `foreach` можно использовать выражение `e.Attributes("info").Remove()`, содержащее метод `Attributes` (обратите внимание на букву «s» в конце), который *всегда* возвращает некоторую последовательность – либо содержащую атрибут с указанным именем, либо пустую, если требуемый атрибут отсутствует. В любом случае к полученной последовательности можно применить метод `Remove` (для пустой последовательности он не будет выполнять никаких действий).

Заметим также, что если не указать операцию приведения к типу `string`, то для атрибута, не равного `null`, будет вызван метод `ToString`, в результате чего к прежнему содержимому элемента будет добавлено не *значение* атрибута (например, `abc`), а *полный его текст* (например, `info="abc"`).

12.6.4. Работа с пространствами имен XML-документа

Здесь и далее мы будем использовать в качестве примеров XML-документы, соответствующие спецификации «Fiction Book 2.0» (данный

формат в настоящее время является одним из популярных форматов хранения электронных книг; файлы в этом формате имеют расширение fb2).

Если открыть для просмотра какой-либо файл с расширением fb2, то в его начале можно будет увидеть примерно такой текст:

```
<?xml version="1.0" encoding="windows-1251"?>
<FictionBook xmlns="http://www.gribuser.ru/xml/fictionbook/2.0"
  xmlns:l="http://www.w3.org/1999/xlink">
  <description>
    <title-info>
…
```

Первая строка показывает, что данный файл представляет собой XML-документ в указанной кодировке. Из второй строки следует, что корневым элементом данного документа является элемент с именем `FictionBook`. Далее следуют элементы второго (в частности, `description`) и более глубоких (в частности, `title-info`) уровней. Однако если загрузить данный файл в переменную `d` типа `XDocument` и попытаться получить ее элемент с именем `FictionBook` (`d.Element("FictionBook")`) или дочерний элемент `description` корневого элемента (`d.Root.Element("description")`), мы получим в качестве результата значение `null`.

Объясняется это тем, что для корневого элемента нашего документа (а тем самым и для всех его элементов-потомков) определено *непустое пространство имен*, образующее начальную часть имени этого элемента (и имен всех его потомков). Пространство имен определяется с помощью стандартного атрибута XML с именем `xmlns`; этот атрибут автоматически обрабатывается при загрузке документа, поэтому вместо его анализа в нашей программе достаточно обратиться к той части имени корневого элемента, которая отвечает за пространство имен:

```
XNamespace ns = d.Root.Name.Namespace;
Console.WriteLine(ns); // http://www.gribuser.ru/xml/fictionbook/2.0
```

На экран будет выведено *строковое представление* объекта `ns` типа `XNamespace` (благодаря неявному вызову для него функции `ToString`); это же представление может быть получено с помощью его свойства `NamespaceName`, доступного только для чтения:

```
Console.WriteLine(ns.NamespaceName);
  // http://www.gribuser.ru/xml/fictionbook/2.0
```

Выведем на экран имя корневого элемента (точнее, строковое представление этого имени):

```
Console.WriteLine(d.Root.Name);
  // {http://www.gribuser.ru/xml/fictionbook/2.0}FictionBook
```

Мы видим, что при наличии в имени непустого пространства имен это пространство указывается в качестве префикса и дополнительно заключается в фигурные скобки. Таким образом, если мы хотим обратиться по

имени к корневому элементу нашего документа, мы можем поступить следующим образом (в результате на экран будет выведено полное содержимое корневого элемента):

```
Console.WriteLine(d.Element("{" + ns + "}FictionBook"));
```

Оказывается, однако, что можно поступить проще, применив операцию *сложения* к объекту типа `XNamespace` и строке, представляющей локальное имя:

```
Console.WriteLine(d.Element(ns + "FictionBook"));
```

Эта возможность обеспечивается благодаря тому, что в классе `XNamespace` переопределена операция «+» с операндами типа `XNamespace` и `string`, которая возвращает объект `XName` – полное имя XML-элемента.

Аналогичным образом можно получить доступ к дочерним элементам корневого элемента, например:

```
Console.WriteLine(d.Root.Element(ns + "description"));
```

Заметим, что в тексте XML-документа элемент-потомок `description` не содержит определения пространства имен. Это объясняется следующим правилом: *пространство имен любого элемента XML распространяется на любые его элементы-потомки, если в них это пространство не переопределено*.

В качестве примера программно переопределим имя элемента `description`, удалив из него «старое» пространство имен, и затем сохраним полученный XML-документ:

```
d.Root.Element(ns + "description").Name = "new-description";
d.Save("Demo.fb2");
```

Начало нового документа будет выглядеть следующим образом:

```
<?xml version="1.0" encoding="windows-1251"?>
<FictionBook
  xmlns:l="http://www.w3.org/1999/xlink"
  xmlns="http://www.gribuser.ru/xml/fictionbook/2.0">
  <new-description xmlns="">
    <title-info xmlns="http://www.gribuser.ru/xml/fictionbook/2.0">
```

Мы видим, что теперь с элементом `new-description` связывается *пустое* пространство имен. Мы видим также, что у потомка данного элемента автоматически восстанавливается пространство имен, определенное в корневом элементе. Таким образом, *программное изменение пространства имен элемента не приводит к автоматическому изменению пространства имен всех его потомков*. Подобное «групповое» изменение пространств имен можно выполнить путем обработки соответствующей последовательности, например:

```
foreach (var e in d.Root.Element("new-description").Descendants())
  e.Name = e.Name.LocalName;
```

В результате начало документа изменится следующим образом:

```
<?xml version="1.0" encoding="windows-1251"?>
<FictionBook
  xmlns:l="http://www.w3.org/1999/xlink"
  xmlns="http://www.gribuser.ru/xml/fictionbook/2.0">
  <new-description xmlns="">
    <title-info>
```

Мы видим, что теперь пространство имен для элемента `title-info` (а также всех остальных потомков элемента `new-description`) не переопределяется.

12.6.5. Префиксы пространства имен

В стандарте XML предусмотрен еще один способ задания пространства имен – с помощью *префикса пространства имен*. Префикс для пространства имен задается с помощью атрибута следующего вида:

`xmlns:`*`имя_префикса`*`="`*`имя_пространства_имен`*`"`

От атрибута `xmlns`, задающего пространство имен для текущего элемента и всех его потомков, данный атрибут отличается наличием дополнительного двоеточия, после которого и указывается имя префикса (в элементе `FictionBook` из п. 12.6.4 таким образом определен префикс `l`, связываемый с пространством имен http://www.w3.org/1999/xlink).

Определение в некотором элементе префикса пространства имен *не приводит* к автоматическому связыванию этого элемента с данным пространством имен. Для того чтобы это связывание произошло, необходимо явно указать префикс перед именем элемента (как в открывающем, так и в закрывающем теге), отделив его от имени двоеточием. Однако даже в этом случае указанное пространство имен *не будет по умолчанию связано со всеми потомками данного элемента*.

Префиксы позволяют задать пространства имен не только для элементов XML, но и для их *атрибутов*. В качестве примера можно привести само определение префикса, которое выполняется с помощью атрибута, снабженного специальным префиксом `xmlns`. Префикс `xmlns` всегда связан со стандартным пространством имен http://www.w3.org/2000/xmlns/; для доступа к этому пространству имен можно использовать статическое свойство `Xmlns` класса `XNamespace`. Например, получить имена всех пространств имен, для которых в корневом элементе XML-документа `d` определены префиксы, можно следующим образом:

```
foreach (var e in d.Root.Attributes()
  .Where(a => a.Name.Namespace == XNamespace.Xmlns))
    Console.WriteLine(e.Value);
```

При выполнении данного фрагмента для рассмотренного выше документа на экран будет выведено единственное имя:

```
http://www.w3.org/1999/xlink
```

Как изменится вид XML-документа, если мы явным образом свяжем один из его элементов с пространством имен, для которого задан префикс? Чтобы это выяснить, изменим в документе, который мы преобразовывали в предыдущим пункте, имя пространства имен для элемента `new-description` на `http://www.w3.org/1999/xlink` (напомним, что к настоящему моменту элемент `new-description` имеет пустое пространство имен):

```
XElement e = d.Root.Element("new-description");
e.Name = (XNamespace)"http://www.w3.org/1999/xlink" + e.Name.LocalName;
```

Сохранив измененный документ и открыв его для просмотра, мы увидим в начале документа следующий текст:

```
<?xml version="1.0" encoding="windows-1251"?>
<FictionBook
  xmlns:l="http://www.w3.org/1999/xlink"
  xmlns="http://www.gribuser.ru/xml/fictionbook/2.0">
  <l:new-description xmlns="">
    <title-info>
```

Итак, с элементом `new-description` теперь связано пространство имен, определяемое префиксом `l`. При этом в элементе `new-description` по-прежнему определено пустое пространство имен, которое действует на всех потомков данного элемента (но не на сам элемент, поскольку для него имя указано явно с помощью префикса). При необходимости можно было бы использовать префикс и для каких-либо потомков, поскольку его область видимости распространяется на всех потомков элемента, в котором префикс определен.

Интересно отметить, что если бы мы вывели измененный элемент `e` на экран оператором `Console.WriteLine(e)`, то его начало выглядело бы так:

```
<l:new-description xmlns="" xmlns:l="http://www.w3.org/1999/xlink">
  <title-info>
```

В данном случае префикс `l` определяется в самом элементе, поскольку «выше» этого элемента ничего нет, и получить другим способом информацию о значении префикса `l` невозможно.

Мы видим, что для программного связывания элемента или атрибута с некоторым префиксом, уже определенным в документе, нет необходимости знать имя префикса – достаточно знать имя связанного с ним пространства имен. Имя префикса требуется указывать лишь при *формировании* данного префикса, например:

```
d.Root.Add(new XAttribute(XNamespace.Xmlns + "pref", "aa/bb/cc"));
```

В результате открывающий тег корневого элемента будет дополнен новым атрибутом:

```
xmlns:pref="aa/bb/cc"
```

В файлах fb2 пространство имен http://www.w3.org/1999/xlink используется в специальных атрибутах `href`, применяемых при организации раз-

личных ссылок (перекрестных ссылок, подстрочных примечаний, ссылок на иллюстрации). При этом для поиска атрибутов href несущественно, связан ли с ними префикс соответствующего пространства имен и каково имя этого префикса, – достаточно знать, какое именно пространство имен нам требуется. В качестве примера выведем все элементы, у которых имеется атрибут href:

```
XNamespace ns1 = "http://www.w3.org/1999/xlink";
foreach (var a in d.Descendants()
                   .Where(e => e.Attributes(ns1 + "href").Any()))
  Console.WriteLine(a);
```

В результате на экран может быть выведен, например, следующий текст:

```
<image l:href="#cover.jpg" xmlns:l="http://www.w3.org/1999/xlink" />
```

Заметим, что в самом файле данный тег, как правило, не имеет атрибута xmlns:l (поскольку префикс l уже определен в корневом элементе XML-документа):

```
<image l:href="#cover.jpg" />
```

12.6.6. Хранение в XML-документе двоичных данных

Продолжим анализ документа, начатый в предыдущем пункте. Мы обнаружили, что в нем имеется элемент image, имеющий атрибут href со значением #cover.jpg. Естественно предположить, что элемент image ссылается на некоторое изображение, соответствующее обложке книги. Символ # имеет тот же смысл, что и в языке HTML: он означает, что ссылка является *внутренней* и связана с тегом в этом же документе, имеющим *идентификатор* cover.jpg (идентификатор элемента задается специальным атрибутом id).

Осталось выяснить, каким образом в текстовом документе XML можно обеспечить хранение *двоичных* данных (в частности, изображений). Для этого используется специальный формат, называемый Base64. В этом формате байты исходного двоичного набора объединяются в тройки (размером 24 бита), каждая из которых интерпретируется как четыре 6-разрядных двоичных числа. Затем каждое из таких двоичных чисел (находящихся в диапазоне от 0 до 63) переводится в соответствующий алфавит формата Base64, содержащий только латинские буквы (26 прописных и 26 строчных), цифры (10) и еще два символа: «*» и «/» – всего 64 символа.

Специальным образом обрабатывается ситуация, когда в конце двоичного набора данных имеются один или два отдельных байта, не образующих тройку. В этом случае к этим байтам добавляются нулевые биты, дополняющие их до целого количества 6-битных кодов: 4 бита в случае одиночного байта (в результате будут получены 12 бит, которые будут закодированы двумя символами) и 2 бита в случае двух байт (в результате бу-

дут получены 18 бит, которые будут закодированы тремя символами). Для того чтобы дополнить полученный «неполный» набор из двух или трех символов до полной четверки символов, к нему добавляется специальный *символ-заполнитель* «=». Этот 65-й символ, в отличие от перечисленных выше 64 основных символов формата Base64, может встретиться только в конце закодированных данных, причем количество его вхождений лежит в диапазоне от 0 до 2.

Преимущество описанного формата Base64 состоит в том, что он использует лишь 65 отображаемых символов из набора ASCII и, следовательно, будет корректно храниться и отображаться во всех распространенных текстовых кодировках.

Для преобразования массива байтов в формат Base64 (и обратного преобразования) проще всего воспользоваться соответствующими статическими методами класса `System.Convert`:

```
string ToBase64String(byte[] inArray[, int offset,                Convert
               int length][, Base64FormattingOptions options]);
int ToBase64CharArray(byte[] inArray, int offsetIn,
               int length, char[] outArray[, int offsetOut][,
               Base64FormattingOptions options]);
byte[] FromBase64String(string s);
byte[] FromBase64CharArray(char[] inArray, int offset, int length);
```

Параметры `offset` определяют индекс, начиная с которого будут обрабатываться элементы исходного массива, параметр `length` – количество обрабатываемых элементов; возвращаемое значение в методе `ToBase64CharArray` равно количеству заполненных элементов в результирующем символьном массиве `outArray`. Параметр `options` имеет перечислимый тип `Base64FormattingOptions` с двумя возможными значениями: `None` (результирующая строка не снабжается дополнительными символами перехода на новую строку), `InsertLineBreaks` (результирующая строка разбивается на части по 76 символов, после каждой из которых вставляются символы перехода на новую строку `"\r\n"`). Отсутствие параметра `options` означает режим без разбиения на фрагменты.

В файлах fb2 все двоичные данные (хранящиеся в формате Base64) размещаются в элементах с именем `binary`. Эти элементы содержат два атрибута: `id` (уникальный идентификатор элемента, позволяющий ссылаться на него с помощью атрибута `href`) и `content-type` (строка, определяющая формат двоичных данных; может принимать значения `image/jpeg`, `image/png`, `image/bmp`).

Как правило, идентификатор элемента, связанного с графическим файлом, совпадает с именем этого файла, поэтому для того чтобы выделить из fb2-файла все иллюстрации и сохранить их в виде «обычных» гра-

фических файлов, достаточно выполнить следующий фрагмент программы (предполагаем, что файл имеет имя demo.fb2):

```
var d = XDocument.Load("demo.fb2");
XNamespace ns = d.Root.Name.Namespace;
foreach (var e in d.Descendants(ns + "binary"))
{
  byte[] ch = Convert.FromBase64String(e.Value);
  using (var fs = File.Create(e.Attribute("id").Value))
    fs.Write(ch, 0, ch.Length);
}
```

12.7. Приложение. Класс XObject и его наследники

В табл. 7 приводится алфавитный список членов (свойств, методов и событий) классов, порожденных от `XObject`. Для свойств в скобках указываются имеющиеся аксессоры: `get` – для чтения, `set` – для записи. События помечаются словом `event`. Во втором столбце указано имя класса, в котором данный член появляется впервые в цепочке наследования.

Таблица 7

Свойства и методы классов, порожденных от XObject

Член класса	Класс
Add	XContainer
AddAfterSelf	XNode
AddBeforeSelf	XNode
AddFirst	XContainer
Ancestors	XNode
AncestorsAndSelf	XElement
Attribute	XElement
Attributes	XElement
Changed (event)	XObject
Changing (event)	XObject
Data (get, set)	XProcessingInstruction
Declaration (get, set)	XDocument
DescendantNodes	XContainer
DescendantNodesAndSelf	XElement
Descendants	XContainer
DescendantsAndSelf	XElement
Document (get)	XObject
Element	XContainer
Elements	XContainer
ElementsAfterSelf	XNode
ElementsBeforeSelf	XNode

Член класса	Класс
FirstAttribute (get)	XElement
FirstNode (get)	XContainer
HasAttributes (get)	XElement
HasElements (get)	XElement
IsAfter	XNode
IsBefore	XNode
IsEmpty (get)	XElement
LastAttribute (get)	XElement
LastNode (get)	XContainer
Name (get)	XAttribute
NextAttribute (get)	XAttribute
NextNode (get)	XNode
Nodes	XContainer
NodesAfterSelf	XNode
NodesBeforeSelf	XNode
Parent (get)	XObject
PreviousAttribute (get)	XAttribute
PreviousNode (get)	XNode
Remove	XAttribute
Remove	XNode
RemoveAll	XElement
RemoveAttributes	XElement
RemoveNodes	XContainer
ReplaceAll	XElement
ReplaceAttributes	XElement
ReplaceNodes	XContainer
ReplaceWith	XNode
Root (get)	XDocument
SetAttributeValue	XElement
SetElementValue	XElement
SetValue	XAttribute
SetValue	XElement
Target (get, set)	XProcessingInstruction
Value (get, set)	XAttribute
Value (get, set)	XText
Value (get, set)	XCData
Value (get, set)	XComment
Value (get, set)	XElement

Литература

1. *Абрамян М. Э.* Практикум по программированию на языках C# и VB .NET. – 2-е изд. – Ростов н/Д: ЦВВР, 2007. – 220 с.
2. *Абрамян М. Э.* Технология LINQ на примерах. Практикум с использованием электронного задачника Programming Taskbook for LINQ. – М.: ДМК Пресс, 2014. – 326 с.
3. *Албахари Дж., Албахари Б.* C# 3.0. Справочник. – СПб.: БХВ-Петербург, 2009. – 944 с.
4. *Албахари Дж., Албахари Б.* C# 5.0. Справочник. Полное описание языка. – М.: Вильямс, 2013. – 1054 с.
5. *Балена Ф., Димауро Дж.* Современная практика программирования на Microsoft Visual Basic и Visual C#. – М.: Русская редакция, 2006. – 640 с.
6. *Раттц-мл. Дж.* LINQ: язык интегрированных запросов в C# 2008 для профессионалов. – М.: Вильямс, 2008. – 560 с.
7. *Рихтер Дж.* Программирование на платформе Microsoft .NET Framework. – 3-е изд. – М.: Русская редакция; СПб.: Питер, 2005. – 512 с.
8. *Рихтер Дж.* CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#. – СПб.: Питер, 2013. – 896 с.
9. *Фримен А., Раттц-мл. Дж.* LINQ. Язык интегрированных запросов в C# 2010 для профессионалов. – М.: Вильямс, 2011. – 656 с.
10. XML Tutorial. URL: www.w3schools.com/xml/ (дата обращения: 10.06.2014).

Содержание