Борис Пахомов

bhv®

C# для начинающих

Основные элементы языка C#
Среда программирования SharpDevelop
Создание основных типов приложений

для начинающих

Борис Пахомов

C# для начинающих

Санкт-Петербург
«БХВ-Петербург»
2014

УДК 004.438 C#
ББК 32.973.26-018.1
П12

Пахомов Б. И.

П12 C# для начинающих. — СПб.: БХВ-Петербург, 2014. — 432 с.: ил.

ISBN 978-5-9775-0943-5

Книга является руководством для начинающих по разработке приложений на языке C#. Приведены общие сведения о языке C# и платформе .NET. Рассмотрены базовые типы данных, переменные, функции и массивы. Показана работа с датами и перечислениями. Описаны основные элементы и конструкции языка: классы, интерфейсы, сборки, манифесты, пространства имен, коллекции, обобщения, делегаты, события и др. Приведены сведения о процессах и потоках Windows, а также примеры организации работы в многопоточном режиме. Рассмотрено создание консольных приложений, приложений типа Windows Forms и приложений для работы с базами данных. В качестве среды разработки в книге использован бесплатный пакет SharpDevelop.

Для начинающих программистов

УДК 004.438 C#
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Екатерина Капалыгина*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Марины Дамбиевой*

Подписано в печать 30.12.13.
Формат 60×90 1/16. Печать офсетная. Усл. печ. л. 27.
Тираж 700 экз. Заказ №
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9 линия, 12/28

ISBN 978-5-9775-0943-5

Оглавление

Введение

Предлагаемая читателю книга по современному языку C# — результат спонтанного решения автора, долго занимавшегося языком C/C++ и интегрированными средами разработки, такими как Borland C++Builder и Visual C++. Но первой уже нет, а вторая все еще дышит. Но с каждым разом — все реже и реже. Так, по крайней мере, мне кажется. Тут я подвожу к мысли, что не такое уж и спонтанное было мое решение. Все дело в том, что первый звонок прозвенел, когда вышла среда Visual C++ 2008. С удивлением обнаружил, что из среды разработки исчез целый раздел работы с базами данных. Кое-что там, конечно, осталось, но вот основного, увы, не стало. Сколько было вопросов к Microsoft по этому поводу в Интернете! Сколько негодований! Но фирма уклонялась от ответа. Мол, якобы, да, того... Были намеки, что в следующей версии среды все поправится. Нет. Не поправилось ни в следующей (2010), ни в недавней (2012). Стало ясно, что это уже политика фирмы. Пользователей упорно отворачивали от C++, развивая C#. Но так как в мире уже много чего сделано на C++ и в ближайшее даже десятилетие-двадцатилетие придется пользоваться этим языком хотя бы для сопровождения уже наработанного, фирма прозорливо поступает, не отказываясь от выпуска изделий для работы в C++. Но в нем уже столько латок, столько заплаток, столько всего подобного, что порой кажется, что и сами разработчики потеряли контроль над языком. Не зря компания, не бросая разработки по C++, начала развивать более "отточенный" язык C#. Разработчики учли все неприятности, через которые сами проходили и заставили спотыкаться и пользователей-программистов. Я всегда удивлялся, глядя на язык Java. Вроде бы не очень заметный, не очень распространенный, мало рекламируемый, но какой удобный! А фирма, создавшая Java, учла неприятности, заложенные в C++, и избежала их. А теперь настала очередь разработчиков C#: они не потеряли хорошего из C++, взяли замечательное из Java и получили более совершенное из-

делие — C#. По сравнению с C++ он действительно отточенный. Это понимаешь, когда принимаешься за его изучение. Те, кто не мучился с C++, станут уныло изучать C# и не заметят его прелести, как не заметил сначала ее я. Пугают классы. В этом языке даже обычные, казалось, типы переменных, такие привычные как `int`, `float`, `string`, `double`, тоже, оказываются, классы! Но, слава Богу, для этих классов введены ключевые слова, которые я только что написал, и поэтому, не зная классов, можно вначале работать с такими типами как бы по-старому, не пугаясь.

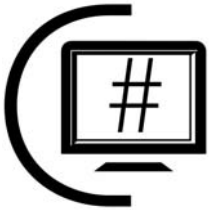
Материал книги построен так, чтобы не попадать сразу в неведомое новое, хотя бы тем, кто раньше изучал языки программирования. Но не всегда удается это сделать. Поэтому на определенных этапах изучения придется кое-что принимать просто на веру, а потом уже в дальнейшем материале видеть, откуда что бралось и почему.

В книге много примеров, которые надо не только разбирать (хотя в них есть комментарии), но и желательно самому их записывать, а не скачивать сразу в приложение, если представится такая возможность. Один читатель моих предыдущих книг как-то прислал мне письмо, что, мол, неудобно из книги заводить себе примеры, составляя из них свое приложение. Муторно. Нельзя ли, мол... Нельзя. Я не сторонник. Когда вы вручную заносите текст в поле редактора, вы одновременно изучаете язык, сами того не подозревая. Пробуете его на вкус. Запоминаете его правила и тем самым избегаете в будущем большого количества ошибок при вводе текста. Хотя сегодняшние компиляторы большинство ошибок быстро отлавливают. Но вот как раз в этот-то момент и происходит ваше близкое знакомство с языком, потому что компилятор вам подсказывает, мол, так нельзя, надо вот так. А если вы отлаженный текст вычлените из книги и вставите его в поле редактора, то ничего этого не увидите и в дальнейшем окажетесь беспомощным при вводе собственного текста.

Чем еще интересен C#? Пользуясь его средствами, вы можете решать проблемы создания графических интерфейсов, не прибегая непосредственно к таким средам, как Windows Forms, ADO.NET и др. Хотя это не рационально, потому что, пользуясь этими средами напрямую, вы пользуетесь теми сервисами, которые в них уже заложены. В C# надо будет делать все вручную. Но сделав хотя бы один примерчик для одной из сред, вы при изучении отдельно этих сред увидите и поймете, почему там в них все так устроено. Такой пример приведен в последней, 18-й главе книги для среды Windows Forms.

Приятного изучения C#.

Автор



ЧАСТЬ I

Базовые сведения о программировании на языке C#

ГЛАВА 1

Общие сведения о языке C# и платформе .NET

.NET (читается "дот нэт") или .NET Framework — это платформа программирования. Вообще, компьютерная платформа — это аппаратный и/или программный комплекс, служащий основой для различных вычислительных систем. Примером платформы программирования может служить операционная система компьютера. Алгоритмический язык C# (читается "си шарп") как раз и создан для работы на платформе .NET.

Разработка программного обеспечения (ПО) на платформах операционных систем (ОС) семейства Windows подразумевала использование языка программирования C (читается "си" в соответствии с английской фонетикой) в сочетании со специальными средствами ОС Windows, которые называются сокращенно API (читается не по буквам "эй пи ай", а "апи"). Это аббревиатура от Application Programming Interface — интерфейс прикладного программирования. В этом интерфейсе сосредоточены крупные программные структуры, позволяющие путем их настройки на конкретное приложение автоматизировать процесс трудоемкого программирования на C. Тот, кому "повезло" испробовать на себе это "удовольствие", думаю, до сих пор видит по ночам кошмарные сны. Но это и понятно: все вновь созданное обычно очень несовершенно и дорабатывается в процессе длительной эксплуатации. Необходимость уйти от использования напрямую в программировании средств API привела к созданию более совершенных систем программирования типа, например, Borland C++Builder, которые значительно облегчили и облагородили тяжелый труд программиста. Однако жизнь не стоит на месте, и язык C на определенном этапе перестал обеспечивать потребности программирования. На горизонте появилась концепция так называемого *объектно-ориентированного программирования* (ООП), которая позволяла посмотреть на сам процесс создания программного продукта со-

всем с другой стороны, предоставляя программисту более широкие возможности для автоматизации его труда и создания более качественной программной продукции. Основой ООП явились понятия *класса* и *объекта*. Разработчики языка C пошли путем добавки к C структуры "класс". Получился язык C++. Этот процесс оказался настолько непростым, что, думаю, в свое время сами разработчики очень пожалели, что приняли именно такую концепцию быть на уровне современных требований к процессу создания программного продукта. В погоне за скоростью обработки приложениями данных и за необходимой надежностью и безопасностью работы приложений разработчикам пришлось организовывать два вида памяти при обработке данных: неуправляемую (в C памятью приходится управлять вручную) и управляемую (в C++ эту функцию берет на себя специальная среда, так называемая *управляемая куча*, поэтому управление памятью — автоматическое), организовывать специальный и довольно неприятный аппарат указателей. Но мы знаем, что чем дальше в лес, тем больше дров. Разработчикам пришлось строить аппарат перехода между данными из управляемой памяти в неуправляемую и наоборот. Легче было похоронить C и создать заново другой язык на новой концепции. Но разработчики были связаны по рукам: очень много программного продукта на C уже работало в мире, и поставить на нем крест значило подорвать производственный процесс множества предприятий и организаций. Поэтому приходилось не только заботиться о сохранении C, но и придерживаться современных требований (создание C++), поддерживать совместимость старых программ при работе в новых средах. То есть надо было тащить за собой хвосты C в новый язык C++, которые только мешали новому языку и осложняли процесс разработки программ на этом языке. В конце концов, видимо, у разработчиков терпение лопнуло, и они создали новый язык под названием C#, учитывающий новые веяния в программировании (ООП) и свободный от недостатков C++. Однако и C++ не оказался заброшенным по причине, отмеченной ранее (совместимость и поддержка уже работающих в мире программ). Да и большое количество программистов, работающих на C++, не очень жаждут изучать новый язык, зная, что переход на более высокий уровень всегда есть шаг назад на некоторое время. В заключение своего пассажа на тему старых-новых программ приведу пример из собственного наблюдения. Одна испанская транснациональная компания, приобретя предприятие, на котором я работал, стала внедрять, что вполне естественно, свою технологию (передовую по тем временам) управления производственным процессом. Привезла с собой свои программы, которые у нее давно работали в других ее "дочках". Оказалось, что многие из программ написаны на языке КОБОЛ, о котором мы забыли еще лет пятнадцать назад.

Но руководство не собиралось из-за наших принципов терять свои деньги и приказало вспомнить КОБОЛ для сопровождения старых программ. Думаю, что я убедил читателя в необходимости создания C# и, тем более, в необходимости его изучения. Замечу также, что C# — это язык семейства языков C, он является гибридом языков C, Java, Visual Basic 6. Следуя за М. В. Ломоносовым, сказавшем о русском языке, что он содержит в себе "великолепие ишпанского, живость французского, крепость немецкого, нежность итальянского, сверх того богатство и сильную в изображениях краткость греческого и латинского языка", про C# можно сказать, что он с синтаксической точки зрения является таким же чистым, как Java, столь же простым, как Visual Basic 6, и таким же гибким и мощным, как C++. Если установить бесплатный продукт фирмы Microsoft .NET 4.0 Framework Software Development Kit (SDK) или среду Visual Studio 2010, то для программирования на основе платформы .NET становятся доступными языки C#, F#, JScript .NET, Visual Basic, C++/CLI. Здесь CLI (Common Language Infrastructure, общеязыковая инфраструктура) — привязка C++ к платформе .NET. Вернемся все-таки к платформе .NET, на базе которой функционирует C#. Эта платформа представляет собой программную платформу для создания приложений не только на базе ОС семейства Windows, но и других операционных систем, которые создавались не фирмой Microsoft, как Windows. Это системы Mac OS X, UNIX, Linux. Платформа обеспечивает взаимодействие с уже существующим программным обеспечением. Приложения на платформе .NET можно создавать с помощью многих языков программирования, таких как C#, F#, S#, Visual Basic и др. Сегодня фирма Microsoft выпускает продукт под названием Visual Studio (2008, 2010, 2012), который дает возможность создавать приложения на разных языках на платформе .NET. Все языки, поддерживаемые .NET, имеют общий исполняющий механизм. Здесь уже нет такой неразберихи, как в C++ (управляемая и неуправляемая память, разные указатели для обоих видов памяти, аппарат перехода от одного вида памяти к другому). Платформа содержит в себе обширную и, что важно, общую для всех поддерживаемых языков библиотеку базовых классов, которые обеспечивают, например, ввод-вывод данных, работу приложений с графическими объектами, создание не только веб-интерфейсов, но и обычных (настольных) и консольных (без графики) приложений, работу с базами данных, дают возможность создавать интерфейсы для работы с удаленными объектами. В частности, платформа .NET Framework — это управляемая среда выполнения, предоставляющая разнообразные службы работающим в ней приложениям. Она состоит из двух основных компонентов: исполняющей среды общего языка (Common Language Runtime, CLR), являющейся механизмом, управляющим выполняющие-

ся приложения, и библиотеки классов .NET Framework, предоставляющей библиотеку проверенного кода, предназначенного для повторного использования, который разработчики могут вызывать из своих приложений. Службы (точнее — сервисы, а еще точнее — услуги), которые платформа .NET Framework предоставляет работающим приложениям:

- *управление памятью.* Во многих языках программирования разработчики самостоятельно назначают и выделяют ресурсы памяти и решают вопросы, связанные со временем жизни объектов. В приложениях платформы .NET Framework среда CLR предоставляет эти сервисы автоматически;
- *система общего типа.* В традиционных языках программирования базовые типы определяются компилятором, что осложняет взаимодействие между языками. В платформе .NET Framework базовые типы определяются единственной системой типа .NET Framework, называемой CTS (Common Type System). При этом используются одни и те же базовые типы для всех языков .NET Framework;
- *расширенная библиотека классов.* Вместо того чтобы писать много кода для выполнения стандартных низкоуровневых операций программирования, разработчики могут использовать легкодоступную библиотеку типов и члены из библиотеки классов .NET Framework;
- *платформы и технологии разработки.* Платформа .NET Framework включает библиотеки для конкретных областей разработки приложений, например ASP.NET для веб-приложений, ADO.NET для доступа к данным и Windows Communication Foundation для приложений, ориентированных на службы (сервисы);
- *взаимодействие языков.* Языковые компиляторы на платформе .NET Framework компилируют приложение не в исполнительный код сразу, а в промежуточный код, называемый языком CIL (Common Intermediate Language), который впоследствии компилируется во время исполнения приложения средой CLR. Такой подход приводит к тому, что программы, написанные на одном языке, доступны в других языках, а разработчики могут сосредоточиться на создании приложений на предпочитаемом языке или языках;
- *совместимость версий.* За редкими исключениями, приложения, которые разрабатываются с помощью платформы .NET Framework определенной версии, могут выполняться без изменений на более поздней версии;
- *параллельное выполнение.* Платформа .NET Framework помогает в разрешении конфликтов версий, разрешая установку нескольких

версий среды CLR на одном компьютере. Это означает, что несколько версий приложений также могут сосуществовать, и что приложение может выполняться на версии платформы .NET Framework, для которой оно было создано;

- *настройка для различных версий.* Ориентируясь на переносимую библиотеку классов платформы .NET Framework, разработчики могут создавать сборки (exe- или dll-файлы, предназначенные для исполнения), которые работают на нескольких платформах .NET Framework. Например, на .NET Framework, Silverlight, Windows Phone 7 или Xbox 360.

.NET Framework для пользователей

Если вы не разрабатываете приложения .NET Framework, но используете их, вам не требуется обладать какими-либо специальными знаниями о платформе .NET Framework или ее работе.

Если используется операционная система Windows, платформа .NET Framework может быть уже установлена на компьютере. Кроме того, если устанавливается приложение, требующее платформу .NET Framework, программа установки приложения может инсталлировать конкретную версию .NET Framework на вашем компьютере. В некоторых случаях можно увидеть диалоговое окно, которое запрашивает установку платформы .NET Framework.

Как правило, не требуется удалять какие-либо версии .NET Framework, уже установленные на вашем компьютере, потому что используемое приложение может зависеть от конкретной версии. В случае удаления какой-либо версии его исполнение может завершиться ошибкой. Обратите внимание, что на одном компьютере может быть одновременно загружено несколько версий платформы .NET Framework. Это означает, что не нужно удалять предыдущие версии для установки более поздней версии.

.NET Framework для разработчиков

Разработчик может выбрать любой язык программирования, который поддерживает платформу .NET Framework, для создания приложения. Поскольку платформа .NET Framework обеспечивает независимость и взаимодействие языков, можно взаимодействовать с другими приложениями и компонентами платформы .NET Framework независимо от языка, с помощью которого они были разработаны.

Для разработки приложений или компонентов платформы .NET Framework выполните следующие действия:

1. Установите версию платформы .NET Framework, на которую будет нацелено ваше приложение. Последняя рабочая версия — это .NET Framework 4.5.
2. Выберите язык или языки платформы .NET Framework, которые вы будете использовать для разработки приложений. Доступны языки Visual Basic, C#, F# и C++ от Microsoft. Язык программирования, который позволяет разрабатывать приложения для платформы .NET Framework, соответствует спецификации Common Language Infrastructure (CLI). В спецификации описываются требования к исполняемому коду приложения и среде исполнения этого кода. Определяется среда, позволяющая многим языкам высокого уровня использоваться на различных компьютерных платформах. Иначе говоря, если ваша программа создана на одном из языков из платформы .NET, то она сможет работать на другом компьютере с другой платформой (не на любой, конечно, а на той, что согласована с .NET).
3. Выберите и установите среду разработки, которая будет использоваться для создания приложений и которая поддерживает выбранный вами язык программирования. Интегрированная среда разработки (Integrated Development Environment, IDE) Microsoft для приложений .NET Framework — это Microsoft Visual Studio, например, версии 2010. Она доступна бесплатно в версии Express. Но существует и другой бесплатный продукт, на котором можно создавать приложения в среде .NET. О нем будет рассказано в *главе 2*.

ГЛАВА 2

Средства создания приложений на языке C#

Описание средств

Чтобы создавать приложения на C#, как и на любом другом языке программирования, надо иметь возможность записать в файл (или в файлы) сам текст программы, а затем этот текст откомпилировать с помощью компилятора, создавая исполняемый файл, при запуске которого на выполнение получается результат работы разработанного приложения. Какие же средства можно использовать для создания приложений на C#?

Простейшим средством для записи и сохранения текста приложения являются текстовые редакторы WordPad и Блокнот. С их помощью можно записать текст приложения на C# и при сохранении текста дать этому тексту расширение cs ("си шарп"). На рис. 2.1 и 2.2 показаны фрагмент текста C#-приложения в WordPad и окно редактора в момент сохранения текста программы.

Далее следует сохраненный текст откомпилировать. Где взять компилятор? Если у вас имеется интегрированная среда разработки приложений Visual Studio 2010 или 2011 (у автора на момент написания этой главы была версия 2011), то в этой среде в главном меню **Tools** надо выбрать команду **Visual Studio Command Prompt** (запуск exe-файлов из командной строки). В результате на экране появится консольное окно, предлагающее вводить исполняемые файлы. Компилятор C# имеет исполняемый файл, названный csc.exe. Наберите имя компилятора в командной строке и нажмите клавишу <Enter>. Вы увидите, что среда требует указать компилируемый файл. Чтобы не писать длинные пути к искомому файлу в консольном окне (капризном, т. к. у него мало возможностей редактирования текста), можно сначала при сохранении в

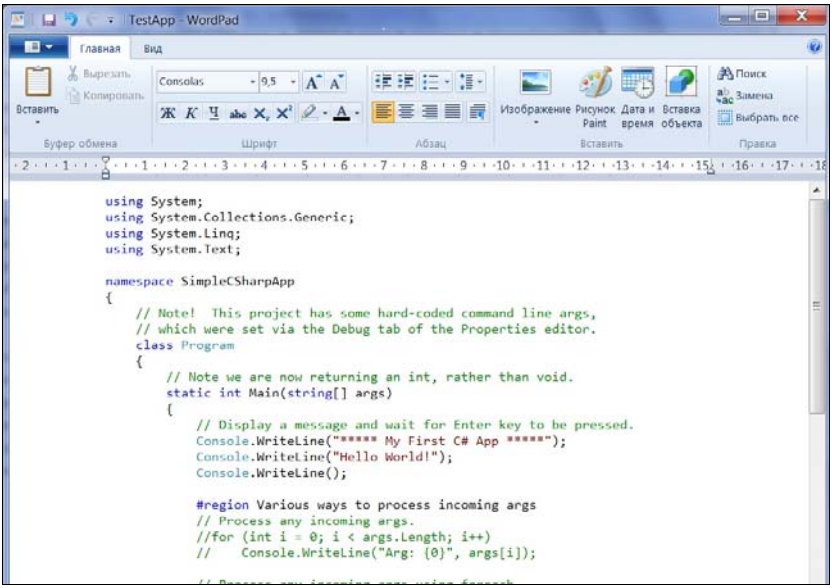


Рис. 2.1. Фрагмент текста приложения в WordPad

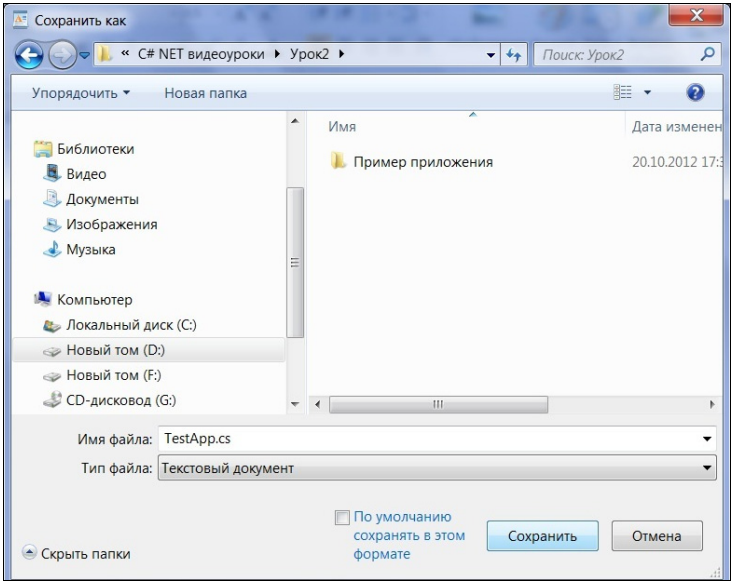


Рис. 2.2. Сохранение текста приложения редактором WordPad

WordPad поместить файл в корневой каталог, в котором находится среда обработки. У автора среда находится в одной из папок на устройстве D:. Поэтому для простоты общения с консольным окном я стану сохранять свои приложения в WordPad в каталоге D:\. Приложение было названо TestApp.cs (рис. 2.2). Переместите его в D:\, теперь попробуйте его откомпилировать. Этот процесс показан на рис. 2.3.

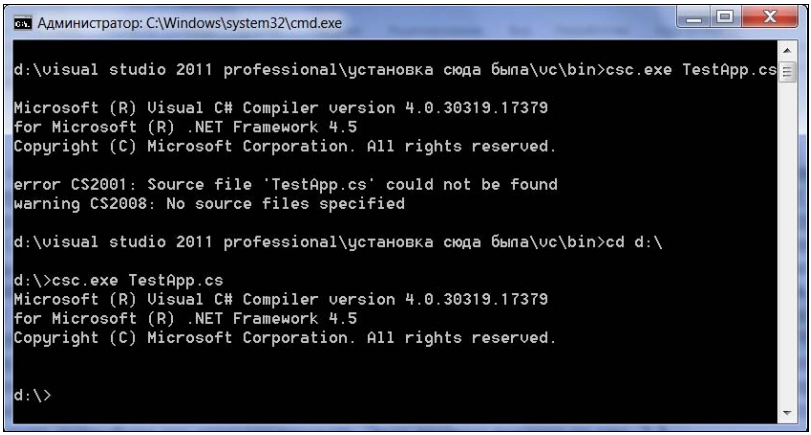


Рис. 2.3. Компиляция из командной строки приложения C#

Из рис. 2.3 видно, что сначала компилятор не нашел исходного файла для компиляции, т. к. файл был перемещен в корневой каталог. Поэтому в окне надо было сначала выполнить команду MS-DOS "Перейти в корневой каталог D:\". Эта команда — `cd` (от англ. *change directory*). После перехода в корневой каталог компилятор нашел исходный файл, и компиляция прошла успешно. Результат компиляции (файл с тем же именем, но с расширением exe) помещен в тот же каталог, что и исходный файл.

Компиляцию исходного кода на C# можно произвести, имея только всем доступную платформу Microsoft .NET Framework, которую можно загрузить бесплатно. Компилятор для C#, csc.exe (C-Sharp Compiler), входит в эту среду. Конечно, большие приложения на нем компилировать будет достаточно проблематично, но все же порой полезно знать, как это сделать. Перед тем как воспользоваться компилятором, нужно его настроить. Для того чтобы проверить, находит ли ваша операционная система файл csc.exe, введите его в командную строку `csc /?`. В ответ должен появиться список опций настройки, поддерживаемых компилятором C#. С командной строкой в разных операционных систе-

мах тоже могут быть проблемы. Если в ОС Windows XP командная строка сразу видна в меню кнопки **Пуск**, то этого нельзя сказать об ОС Windows 7. Там, чтобы добраться до командной строки, надо нажать кнопку **Пуск**, а затем в поле поиска в нижней части меню кнопки (в самом поле виден текст "Найти программы и файлы") надо ввести слово `команд`. Среди множества найденных строк вы увидите и значок командной строки с названием **Командная строка**. Щелкнув по значку, на экране увидите консольное окно с текстом и приглашение вводить данные (рис. 2.4).

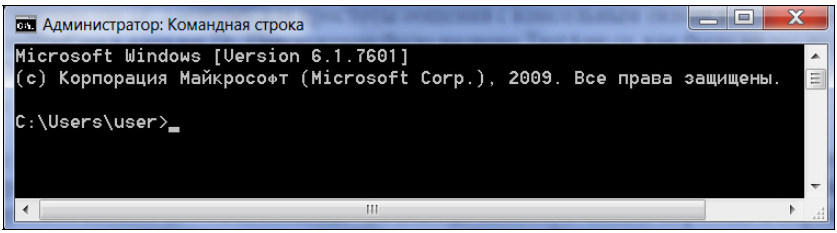


Рис. 2.4. Консольное окно командной строки операционной системы

Если компилятор не находится, значит, в системе не прописан к нему путь. Чтобы прописать путь, нужно щелкнуть правой кнопкой мыши на **Мой компьютер** (в Windows XP), выбрать опцию **Свойства**. Если у вас Windows XP, то нужно выбрать опцию **Дополнительно** (в Windows 7 — ссылка слева **Дополнительные параметры системы**) и щелкнуть по кнопке **Переменные среды** в нижней части открывшегося окна **Свойства системы**. Откроется диалоговое окно **Переменные среды**, в котором в прокручиваемом поле **Системные переменные** найдите переменную `Path`, щелкните на ней мышью, нажмите кнопку **Изменить** под этим полем. Откроется диалоговое окно с содержимым переменной `Path` для ее корректировки. Вам необходимо дописать к концу значения переменной точку с запятой и путь к размещению .NET Framework SDK. Обычно это C:\Windows\Microsoft.NET\Framework\v3.5, но версию лучше уточнить, войдя в папку C:\Windows\Microsoft.NET\Framework.

Для создания C#-приложения можно воспользоваться возможностями какой-либо интегрированной среды типа Visual Studio. Если вы — начинающий программист, то у вас может не оказаться такой среды, а пока вы ее приобретете, у вас пропадет охота изучать язык. Лучше воспользоваться специально разработанным и бесплатным продуктом Microsoft для изучения C# под названием SharpDevelop.

SharpDevelop можно бесплатно скачать и установить у себя на компьютере, зайдя на интернет-страницу по адресу **www.sharpdevelop.com**. На

этой странице надо перейти на вкладку **SharpDevelop** в верхней части страницы и в открывшемся окне выбрать вкладку **Download**. Во вновь открывшемся окне щелкнуть по ссылке **Downloads for SharpDevelop 4.3**.

Далее мы рассмотрим, как пользоваться этой средой для создания C#-приложений. Отметим сразу следующее: те, кто работал в средах Visual Studio, найдут в новой среде много сходств.

Интегрированная среда SharpDevelop для создания приложений на языке C#

Общий вид главного окна SharpDevelop показан на рис. 2.5.

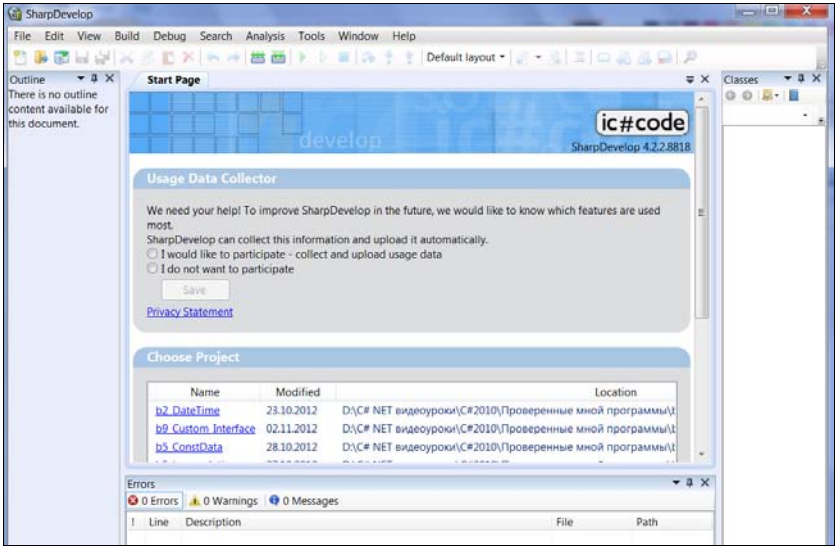


Рис. 2.5. Общий вид главного окна SharpDevelop

В центре окна находится стартовая страница (**Start Page**), в разных частях которой отражаются такие элементы, как Usage Data Collector — сведения об использовании SharpDevelop разработчиками. Сведения собираются фирмой, создавшей продукт SharpDevelop (так называемая обратная связь, служащая для возможного совершенствования выпу-

щенного продукта на основе мнений его пользователей). Ниже расположено окно **Choose Project**, в котором отображаются проекты, выполненные ранее и доступные для повторного вызова через это окно. Для этого достаточно дважды щелкнуть на соответствующей строке, и нужный проект появится в своем окне на экране, готовый для обработки. Но можно и просто один раз щелкнуть по названию проекта и затем нажать кнопку **Open solution**, находящуюся ниже этого окна. Если ее не видно, надо мышью протянуть ползунок окна вниз (ползунок прокрутки окна). Это показано на рис. 2.6.

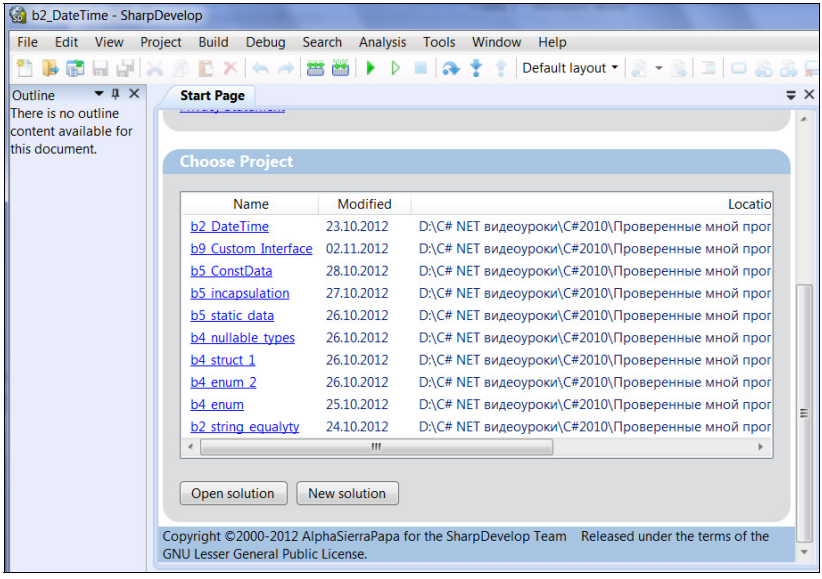


Рис. 2.6. Прокрутка окна **Choose Project**

Ниже окна **Start Page** находится окно **Errors** (Ошибки). В это окно компилятор выводит сведения об ошибках компиляции. Фразы "окно выше", "окно ниже", "окно слева", "окно справа" весьма относительно указывают о расположении окон. Дело в том, в SharpDevelop, как и в последних версиях Visual Studio, окна снабжены свойствами, позволяющими им сворачиваться (делаться невидимыми), перемещаться по экрану с помощью протягивания мышью за заголовок окна, захватывать друг друга при перемещении, когда одно окно попадает в поле захвата другого окна. Эти захваты называются *причаливанием* (Dock), подобно причаливанию судов в доках порта. Порт как бы захватывает или отпускает суда. Так и здесь с окнами. Только роль порта играет окно, которое

не перемещается, а роль судна — перемещаемое окно. Перечень свойств окна можно увидеть, как и свойств любого объекта Windows, щелкнув на самом объекте правой кнопкой мыши. Только для окон надо устанавливать курсор мыши на заголовок окна, а уже потом щелкать правой кнопкой. Свойства окна показаны на рис. 2.7.

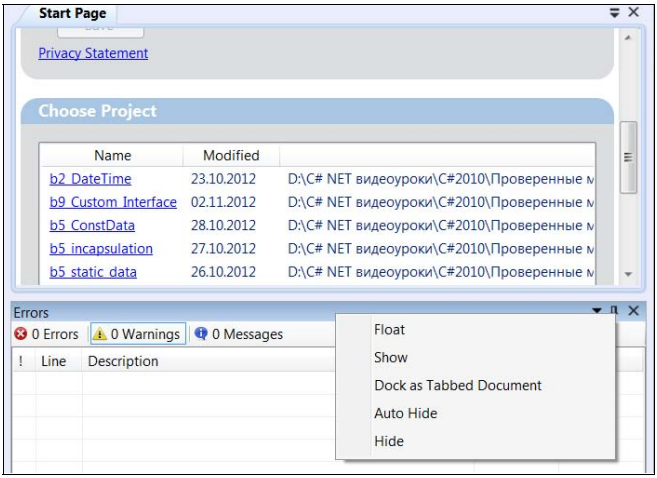


Рис. 2.7. Свойства окна в SharpDevelop

Посмотрим, какой вид приобретает окно, когда мы выбираем то или иное его свойство.

- **Float**. Это свойство обеспечивает окну "свободное плавание". Вы можете перетаскивать мышью окно в любое место экрана, и, когда отпустите кнопку мыши (протяжка идет ведь при постоянно нажатой кнопке), окно останется на том месте, где случилось отпускание кнопки мыши. Но при движении оно может быть захвачено другим окном, если попадет в область захвата этого окна.
- **Hide**. Окно исчезает с экрана — становится невидимым. Чтобы оно снова появилось на экране, надо воспользоваться пунктом **View** главного меню среды. Главное меню среды расположено в виде опций в самой верхней строке окна среды (рис. 2.8).

  Пункт меню **View** содержит, кроме прочего, имена окон среды. Если щелкнуть на соответствующем имени, окно станет видимым на экране. Например, вы можете закрыть стартовую страницу, если она вам мешает, а потом при необходимости снова ее показать, воспользовавшись командой **View | Show start page**.

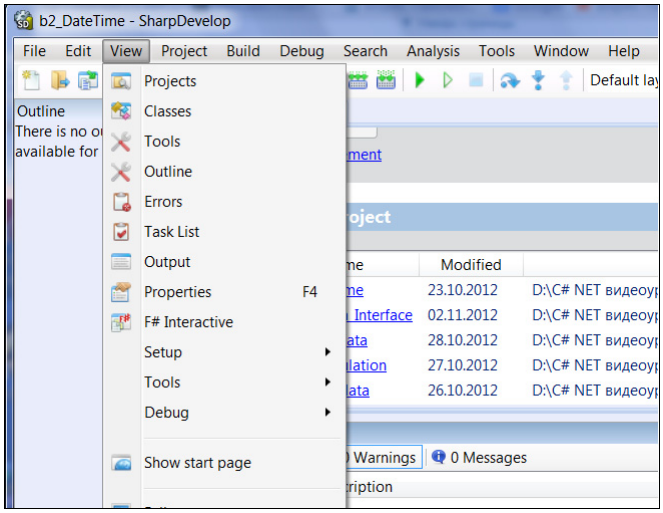


Рис. 2.8. Вид главного меню SharpDevelop и команд меню **View**

- **Dock as Tabbed Document**. Это свойство окна, если оно выбрано, автоматически заставляет окно причалить к главному окну среды в качестве его очередной вкладки (рис. 2.9).

Остальные свойства окна рассматривать не будем.

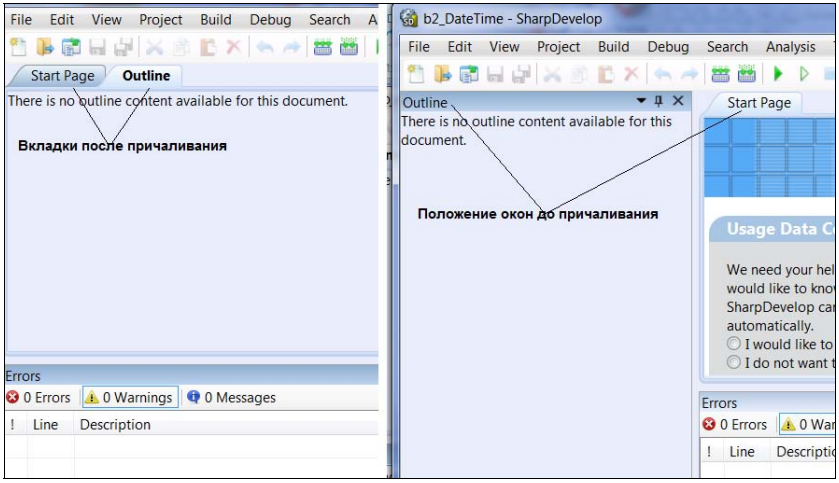


Рис. 2.9. Причаливание окна со свойством **Dock as Tabbed Document**

Теперь посмотрите, как на экране проявляются области захвата, когда мы протягиваем некоторое окно. Возьмите любое из окон рабочего стола SharpDevelop, например то, которое только что в предыдущем свойстве причаливали, а потом установили на прежнее место (окно **Outline** из рис. 2.9). Зацепите его (за его заголовок) мышью и начните перетаскивать вправо. Как только вы установите курсор на заголовок окна и нажмете левую кнопку мыши, готовясь к протяжке окна, на рабочем столе среды тут же появятся указатели — индикаторы областей захвата (рис. 2.10).

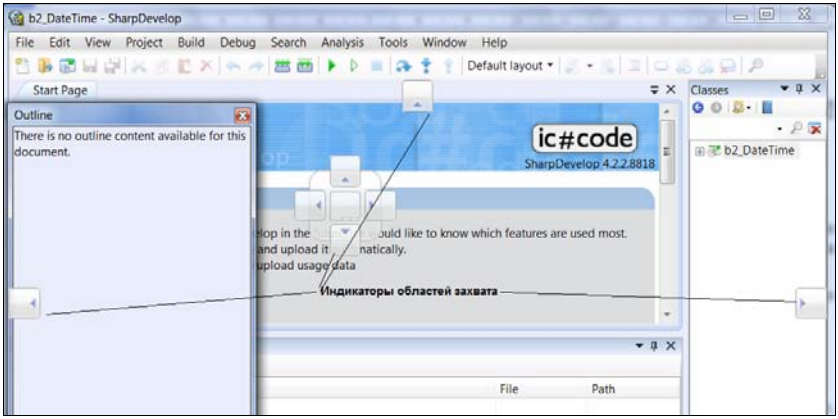


Рис. 2.10. Указатели областей захвата рабочего стола SharpDevelop

Причем, появляются не все, что показаны на рис. 2.10. По мере протяжки появляются и другие. Те индикаторы, в чье поле действия попадает протягиваемое окно, изменяют свой цвет с белого на голубой (рис. 2.11).

Если теперь отпустить левую кнопку мыши, то произойдет захват окна той областью, на которую указал подсвеченный индикатор (рис. 2.12).

Теперь попробуйте поставить окно на его прежнее место. Для этого опять зацепите курсором мыши заголовок окна и начните его тянуть медленно на прежнее место. Вы увидите, что вскоре слева на рабочем столе среды разработки появится темная вертикальная полоса, тоже показывающая область захвата окна, но уже не в виде прямоугольников, как раньше (рис. 2.13).

Если теперь отпустите левую кнопку мыши, прекратив тем самым протяжку окна, и обратите внимание, что окно встало на свое прежнее место.

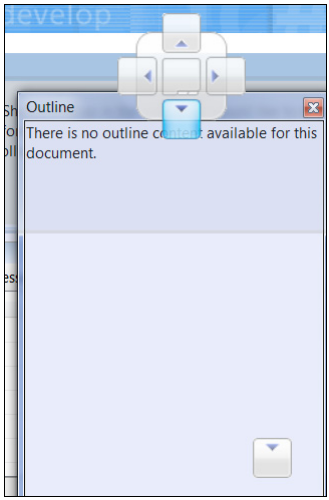


Рис. 2.11. Активизация области захвата на рабочем столе при протягивании окна

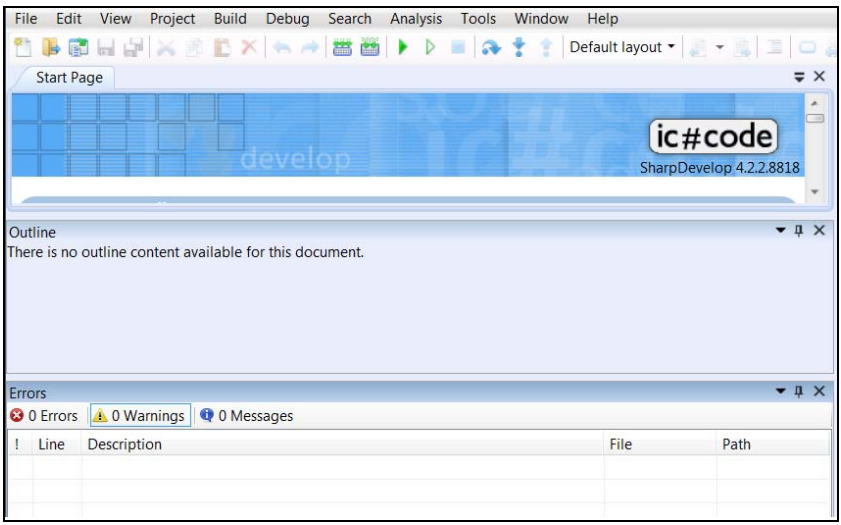


Рис. 2.12. Результат захвата окна

Далее вы самостоятельно можете поэкспериментировать с перемещением окон в нужное для вас место. При этом надо учесть, что если одно окно захватывает другое, то оно помещает захватываемое окно внутрь себя в виде вкладки (становится видна только вкладка с названием окна, возможно, и неполным из-за недостатка места). Окно, помещенное внутрь другого окна, можно снова вытащить, захватив курсором мыши

за вкладку и потянув окно в сторону. Однако перемещениями окон, особенно на начальном этапе работы с SharpDevelop, увлекаться не стоит: можно такого нагородить, что потом вообще станет ничего не понятно. Придется закрывать все окна и затем по одному открывать с помощью меню **View** и настраивать на нужное месторасположение.

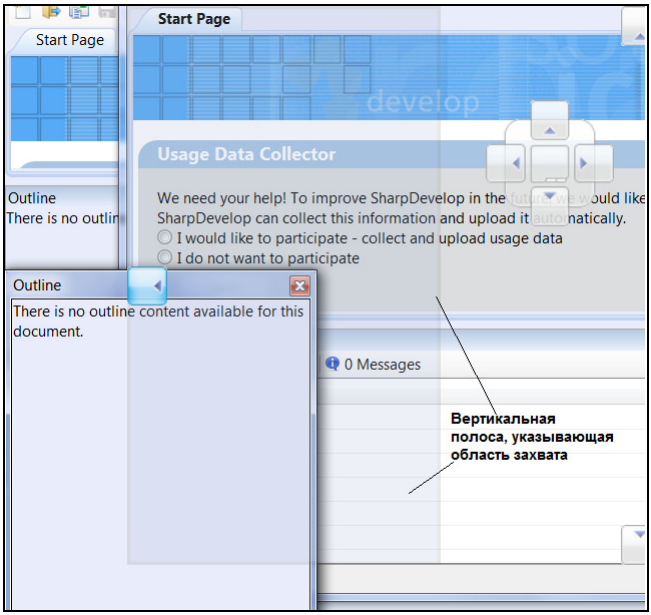


Рис. 2.13. Область захвата в виде вертикальной прямоугольной полосы рабочего стола

Суть остальных окон, возможно, и не всех, мы рассмотрим по мере создания C#-приложений, на примерах которых станем изучать сам язык.

Как создавать приложения в рамках SharpDevelop? Здесь принята та же система, что и в последних версиях Visual Studio: приложения оформляются в виде структур, которые называются *решениями* — solution. Решение состоит из нескольких проектов (project), что дает возможность формировать приложение из нескольких проектов, подключая их к данному решению, запускать приложение из некоторого проекта, делая его стартовым (см. меню **Project**). Кроме того, к проекту можно добавлять и отдельные файлы, расширяющие функциональность проекта. Этими сложными образованиями мы заниматься не станем, т. к. у нас цель другая: изучить C# на таком уровне, чтобы можно было начинать строить на нем приложения, хотя бы не очень сложные. Поэтому, создавая любое приложение, начинают с создания нового решения, которое

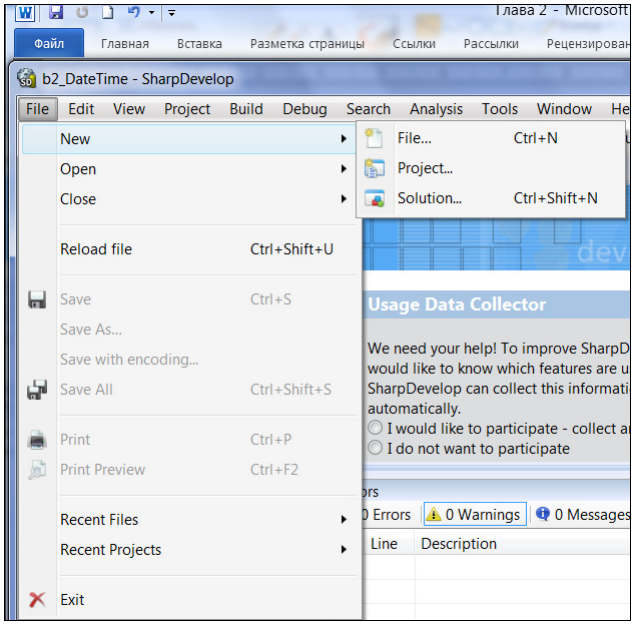


Рис. 2.14. Путь создания нового решения

автоматически включает в себя первый и единственный проект (рис. 2.14—2.16).

Мы начнем с создания самых простых, неграфических приложений, которые принято называть *консольными*, у которых ввод-вывод данных происходит через одно черное окно, называемое *консольным окном*. (Кстати, когда еще не придумали компьютеры, а программисты работали на очень больших (по габаритам, естественно, т. к. память у них и скорость были мизерными) вычислительных машинах, то общение с такой машиной происходило через специальную печатную машинку, откуда задавались команды и куда выходили сообщения операционной системы. Такая машинка тоже называлась консолью.) Итак, мы станем создавать консольные приложения, поэтому при создании приложения выбираем путь, показанный на рис. 2.15.

После того как вы зададите все необходимые данные для нового приложения (см. рис. 2.15), нажмите кнопку **Create** (Создать), и вы получите заготовку нового консольного приложения (в окне по центру), а справа, в окне **Projects** (Проекты), увидите структуру приложения: мы получили приложение в виде структуры Solution app1, в которую входит в качестве ее члена проект app1 (рис. 2.16).

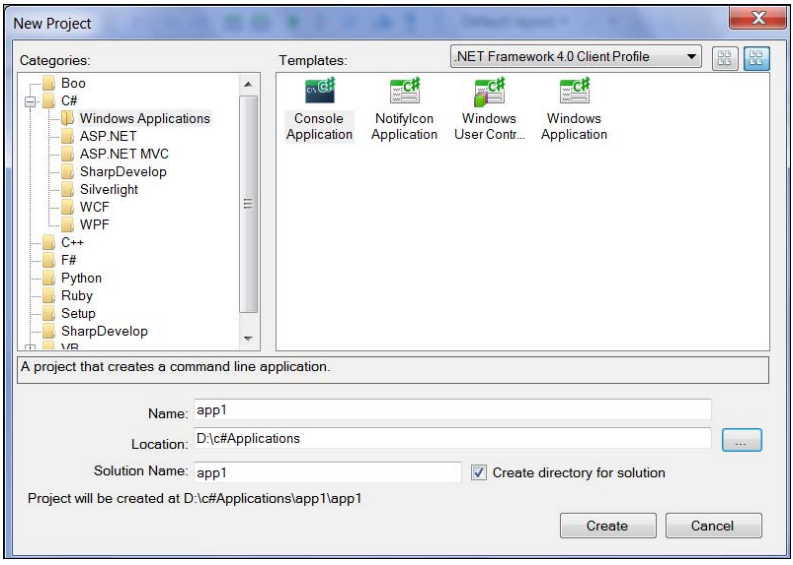


Рис. 2.15. Путь создания нового решения. Выбор консольного варианта приложения

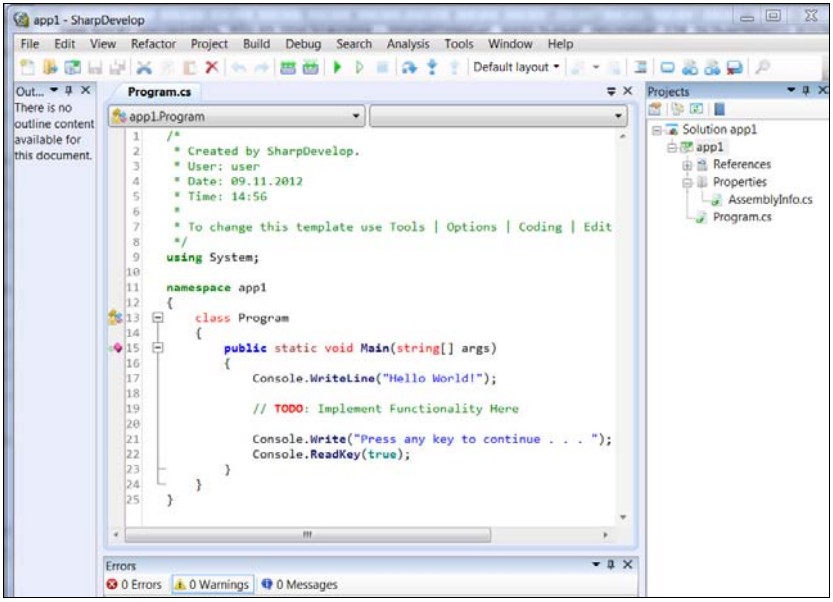


Рис. 2.16. Окно заготовки консольного приложения

ГЛАВА 3

Базовые типы данных, переменные

В *главе 2* при изучении SharpDevelop мы создали приложение app1.cs. Вид его приведен в листинге 3.1.

Листинг 3.1

```
/* Created by SharpDevelop.
 * User: user
 * Date: 09.11.2012
 * Time: 14:56
 *
 * To change this template use Tools | Options |
 * Coding | Edit Standard Headers.
 */
using System;

namespace app1
{
  class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Hello World!");

      // TODO: Implement Functionality Here

      Console.Write("Press any key to continue...");
      Console.ReadKey(true);
    }
  }
}
```

Фактически это еще не совсем приложение, а только лишь его макет, шаблон, который дальше надо наполнять своим содержанием. Коротко рассмотрим структуру этого шаблона. Здесь начинающему разработчику придется просто пока запомнить структуру приложения, которая станет более понятной по мере дальнейшего изучения языка. Но кое-что уже сейчас можно пояснить.

Первые несколько строк сверху, начало которых помечено символами /*, а конец — символами */ — это *комментарий* к программе, который среда программирования, в данном случае SharpDevelop, автоматически формирует. При компиляции этой программы компилятор комментарии не рассматривает. Если комментарий занимает *более одной строки*, то пользуются отмеченными выше парами символов, помещая текст комментария в эти "скобки". Если же комментарий помещается *в одну строку*, то можно применять другую его форму: помещать в начало текста комментария символы `//`. Но можно и для одной строки применять пару `/*...*/`. Комментарий служит не только для пояснения текста программы. Им широко пользуются при отладке программы, выключая с его помощью отдельные фрагменты программы, чтобы потом, удалив символы комментария, восстановить отключенный комментарием текст программы. Понятно, что вместо автоматически сформированного комментария программист может вставлять свои пояснения.

Ниже комментария идет строка (а их в действительности может быть много) `using System;`. Это говорит о том, что среда программирования обязательно подключает к создаваемому приложению набор системных программных средств, часть из которых может понадобиться программисту при создании приложения. Например, могут потребоваться средства ввода-вывода данных, работы с базами данных, графические средства и т. п. Пространство `System` содержит полный набор таких средств, а программист потом уже в ходе создания своей программы будет выделять из `System` необходимые ему подпространства и брать из них необходимые ему средства. Например, подпространство `Console`, которое содержит в себе средства консольного ввода-вывода, т. е. ввода-вывода без графического интерфейса, а только используя консольное окно, о котором мы говорили раньше.

Далее в шаблоне нашего приложения присутствует строка `namespace app1`, которая является заголовком создаваемого приложения (программы). Фактически это заголовок блока строк, помещенных между открывающей и закрывающей фигурными скобками (первой и последней). Почему перед наименованием приложения стоит слово `namespace`, пока

не станем уточнять: у нас еще не хватает знаний. Запомним только, что создаваемое приложение всегда помещается в *пространство имен* (так переводится это слово), которому автоматически присваивается имя самого приложения. Можно это самостоятельно изменить на другое.

В блок, озаглавленный как `namespace app1`, входит блок (ограниченный своей парой фигурных скобок), названный `class Program`. В C# приложение оформляется не как попало, а в виде специальной структуры — *класса*, а сама, собственно программа, задается как член этой структуры со своим заголовком (в данном случае — это `public static void Main(string[] args)`) и своим блоком строк, помещенных в фигурные скобки. То есть, уже рассматривая простой шаблон приложения, мы видим его четкую структуру: множество вложенных друг в друга подструктур (как у матрешки).

В настоящем приложении, а не игрушечном, которое мы рассматриваем, бывает много структур типа класса, и компилятору надо будет отличать при компиляции приложения, что относится к собственно программе, а что — к "не программе".

Посмотрим на заголовок программы. Нас интересует ключевое слово `Main`. Любая программа должна начинаться с этого слова. `Main` значит "главный". Это — точка входа в программу, место, с которого начнет выполняться программа. Компилятор по этому ключевому слову как раз и задаст адрес в памяти, начиная с которого программа запустится на выполнение. Конструкция `Main` оформлена в виде элемента языка, называемого *функцией*. Мы этот элемент будем рассматривать позже. Отметим, что признаком функции, по которому компилятор ее отличает от других элементов программы, являются открывающая и закрывающая простые скобки, внутри которых задаются (а могут и не задаваться) параметры функции. Вообще вся программа — это набор функций, вложенных в `Main`, поэтому общая, содержащая все другие функции программа и получила название *главной*. В нашем случае в заголовке программы `Main()` — далее будем уже писать, как положено для обозначения функции — имеется набор параметров. Параметры служат для взаимодействия данной программы с другими (которые тоже начинаются с `Main()`) путем обмена данными через эти параметры. Этот процесс мы рассматривать не станем. Поэтому, кому мешает вид заголовка программы с указанием параметров, может их удалить, оставив заголовок только с `Main()`:

```
public static void Main()
```

Компилятор "ругаться" не будет.

Если теперь посмотреть на тело самой функции `Main()`, то в нем отражено следующее (указано в виде комментария):

```
Console.WriteLine("Hello World!"); // Вывод строки
                      // "Hello World!" в консольное окно
// TODO: Implement Functionality Here — здесь должны быть
// операторы приложения, задающие функциональность самого
// приложения, т. е. что приложение должно делать
Console.Write("Press any key to continue...");
/* Выводится в консольное окно сообщение "Press any key to
 continue... " (нажмите любую клавишу для продолжения) */
Console.ReadKey(true);  // Задержка экрана
```

Поясним. Когда происходит вывод в консольное окно, строки пробегают по экрану, и после последней выведенной строки экран исчезает, т. к. выводить больше нечего, и программа завершается. Поэтому вы не увидите ничего, кроме как мелькнувшую черную полоску. Следовательно, экран надо задержать перед закрытием, чтобы можно было посмотреть результаты вывода. Для этого искусственно заставляют программу ждать ввода любого символа с клавиатуры. Как только такой символ будет введен, экран закроется. А до этого рассматривайте экран, сколько захотите. Поэтому перед задержкой экрана выводится сообщение, чтобы пользователю было ясно, что дальше делать, когда он хочет продолжить работу программы. И еще. В консольное окно сообщения можно выводить как латинским шрифтом, так и кириллицей.

При рассмотрении в дальнейшем примеров нам придется пользоваться консольными средствами ввода-вывода. Поэтому сразу уточним, как они работают. Отметим, во-первых, что все эти средства — это функции (пишутся в виде имени с параметрами в круглых скобках). Во-вторых, все эти средства находятся в специальной структуре под названием `Console`, которая входит в общую систему `System`. Чтобы отметить (для компилятора), куда входят средства ввода-вывода, их имена пишутся через точку от имени их родителя `Console`. И еще. Весь ввод-вывод для консоли представляется в виде потока символов (т. е. символы передаются на устройство ввода-вывода по одному друг за другом). С учетом этого:

- `WriteLine(параметр — строка текста)` вставляет в поток вывода строку текста вместе с символом перевода строки и возврата каретки: это специальный управляющий символ, который, когда его прочитает средство вывода на экран, заставляет перевести вывод на следующую строку экрана, начиная с самой левой позиции. "Возврат

каретки" — термин, оставшийся от применявшихся ранее пишущих машинок: когда передвигали каретку влево, машинка автоматически переходила на строку в ее самую левую позицию, т. е. в ее начало.

- `Write(параметр — строка текста)`. Делает то же, что и `WriteLine()`, но не формирует символа перевода строки и возврата каретки. Если применять несколько ряд подряд это средство вывода, то выводимые на экран строки будут помещаться одна за другой без перехода на новую строку. Иногда это бывает полезно.
- `ReadLine()`. Обеспечивает ввод строк с клавиатуры (говорят: выдает данные из входного потока, пока не будет нажата клавиша <Enter>). Введенную строку помещает в определенную пользователем переменную (об этом *см. далее*).
- `Read()`. Вводит один символ с клавиатуры. Введенный символ помещает в определенную пользователем переменную (об этом *см. далее*).

Переменные

Любая программа работает с объектами, которые называются *переменными* (в данном случае — это имя существительное). Если вы читаете эту книгу, наверняка, изучали математику, где тоже имеют дело с переменными. В программировании это более широкое понятие. Определим это понятие так: переменная — это некое пространство памяти, которому присвоено имя и в которое могут помещаться определенные данные. Каждый раз разные. В этом смысле пространство все время может содержать разные данные. Имя такого пространства и называют переменной. Но данные бывают очень разные. Например, целые числа. Или дробные. Или текст. Или личная карточка работника. Или целый завод. И так далее. И под каждое такое данное надо выделять соответствующую ему память. Как должен поступать компилятор программы, если программа станет работать с такими данными? Наверное, данные надо как-то уметь различать по их типу. Например, целые числа отличаются от нецелых чисел, личная карточка работника вообще содержит разнотипные данные (фамилию, профессию, данные по зарплате, по премиям и т. д.). А про завод и говорить нечего. То есть фактически данные отличаются друг от друга по своей структуре. Поэтому для целых чисел надо выделять одно количество памяти, для дробных — другое, для карточки — третье и т. д. Да и способ обработки данных различных структур совершенно разный: для умножения двух целых чисел их надо просто перемножить, а двух нецелых (а еще хуже: одного целого, а другого

дробного) уже просто перемножить не пройдет: их надо представить в специальном виде (мантисса и порядок), потом кое-что сделать с этими частями, потом еще кое-что и т. д. Поэтому если каждый вид данного типизировать, т. е. определить ему его тип, тогда уже можно, глядя на тип данного, знать, сколько ему надо выделить памяти, какие операции можно проводить с данными этого типа, какие результаты и какого типа могут получиться и т. д. Тогда при компиляции программы компилятор посмотрит на тип данного и все расставит по своим местам. Например, целому числу выделит четыре байта памяти, дробному — восемь, под строку текста — столько-то. И если в программе станут сравниваться две строки, компилятор правильно построит участок кода программы, исходя из того, что сравниваются именно строки данных. Короче говоря, чтобы работать с данными в программе, надо задавать типы этих данных и имена этих данных. То есть описывать, как говорят, эти данные в программе. Или еще говорят: объявлять данные, что то же самое.

В C# принято объявлять данные, в общем случае так: "тип имя данного". Типы данных бывают разные: одни заданы раз и навсегда в самом языке (базовые типы), другие программист сам задает (например, структуры, классы, объекты). Базовым типам сопоставлены фиксированные ключевые слова, небазовые типы программист сам именует.

Тип целочисленных данных

В табл. 3.1 приведены различные типы данных для целых чисел: хотя все числа целые, но в зависимости от типа им выделяется разное количество памяти.

Таблица 3.1. *Размер и диапазон целочисленных типов в C#*

Тип	Размер (байт)	Диапазон значений	Как использовать (объявление и инициализация переменной)
`sbyte`	1	От –128 до +128	`sbyte sb=12;`
`byte`	1	От 0 до 255	`byte b=12;`
`short`	2	От –32 768 до +32 767	`short si=-12;`
`ushort`	2	От 0 до 65 535	`ushort us=12;`
`int`	4	От –2 147 483 648 до +2 147 483 647	`int i=-14;`
`uint`	4	От 0 до 4 294 967 295	`uint ui=14;`

Таблица 3.1 *(окончание)*

Тип	Размер (байт)	Диапазон значений	Как использовать (объявление и инициализация переменной)
`long`	8	От –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807	`long lg=-14;`
`ulong`	8	От 0 до 18 446 744 073 709 551 615	`ulong ul=14;`

Здесь требуется кое-что пояснить.

`sbyte` — байт с учетом знака. Знаковый разряд всегда располагается в старшем (нулевом — счет идет слева направо) разряде (бите). В байте — 8 разрядов (битов). Поэтому минимальное число, которое может поместиться в такую память, — это -2^7, т. е. –128 (надо все разряды памяти выставить в единицы). Таким же методом считаются и остальные диапазоны.

Буква `u` перед типом означает `unsigned`, т. е. беззнаковый. Поэтому старший разряд выделенной памяти участвует в размере числа. Для памяти размером в байт получим минимум, когда все биты нулевые, т. е. ноль. А максимум — когда все биты в единице. То есть $2^8 - 1 = 256 - 1 = 255$. Почему $2^8 - 1$? Когда вы перебросите все восемь битов в 1, добавьте к ним 1 в младший разряд и вычтите ту же единицу (число ведь не должно измениться). Когда сложите по правилам двоичного сложения, то получите то, что и требовалось.

В последней колонке таблицы показано, как объявлять переменные в программе с одновременным присвоением им начальных значений. Видно, что все соответствует ранее определенному правилу:

тип_переменной имя_переменной

Добавлена только *инициализация переменной*, т. е. присвоение ей начального значения. Присвоение происходит с помощью операции `=`, которая читается так: присвоить содержимое правой части выражения левой части. Это не знак равенства. Это — присвоение. Компилятор подобное выражение преобразует в команду пересылки содержимого правой части по адресу, где размещена переменная левой части выражения. Мы уже употребляем слово "выражение", хотя его и не определяли. Выражение — это переменные, соединенные между собой знаками операций. Например, `m > n` — выражение.

Тип данных с плавающей точкой

Мы только что рассмотрели разные типы целых чисел. Но в жизни очень небольшой процент задач может работать с целыми числами. Больше все-таки приходится работать с нецелыми или, как их еще называют, *вещественными* (или действительными) числами. Для таких чисел в языке тоже существуют свои ключевые слова, задающие тип чисел. Представление таких чисел в памяти компьютера называется *представлением числа с плавающей точкой* (иногда называют "с плавающей запятой"). Если целое число располагается в памяти, выделенной ему компилятором в соответствии с его указанным типом при объявлении в виде последовательности цифр в записи числа (для простоты понимания: на самом деле это не всегда так, потому что числа представляются в виде определенных кодов), то число с плавающей точкой представляется в выделенной для него компилятором памяти в виде двух частей: одна часть называется *мантиссой*. Она по абсолютной величине всегда меньше единицы. А другая часть называется *порядком числа*. Например, число 125,5 можно записать в виде $0{,}1255\times10^3$. Здесь 0,1255 — мантисса, а 3 — порядок числа. Вот это число и станет храниться в отведенной для него области памяти так: знак мантиссы — мантисса — знак порядка — порядок. Вот тут еще раз видно, как важно в программе задавать тип данного. Для задания типа чисел с плавающей точкой существуют два ключевых слова: `float` и `double`. Если для `float` компилятор выделяет восемь байтов памяти, то для `double` — шестнадцать. Надо помнить, что арифметические операции с числами с плавающей точкой не точны, т. е. имеют определенную погрешность в отличие от операций с целыми числами. Например, делим одно число на другое. Нацело не делится. Значит, надо где-то остановиться и отбросить "хвост", не помещающийся в область памяти, отведенной для числа. Вот уже и неточность возникла. Неточность возникает и при выполнении операций, при которых приходится выравнивать порядки чисел и за счет этого сдвигать их мантиссы, чтобы число не изменилось. При сдвиге мантиссы тоже теряются разряды числа. Опять возникает неточность. А когда надо выравнивать порядки чисел? Например, при операции сложения: если у операндов разные порядки, то их сначала надо выровнять (соответственно, чтобы число не изменилось, надо мантиссу сдвинуть влево или вправо на число разрядов, на которое изменился ее порядок), потом сложить мантиссы и к сумме добавить общий порядок. Но чем больше область памяти для хранения числа, тем большее количество цифр дробной части можно хранить в этой области. То есть тем число будет с более высокой точностью. Числа типа `float` имеют точ-

ность в 6—7 десятичных цифр после точки, а числа типа `double` — 15—16 цифр после точки. Так как числа с плавающей точкой в компьютере — неточные, т. е. имеют некоторую погрешность, сравнивать их между собой, как это можно делать с целыми числами, следует по особому правилу. Допустим, у вас имеется число 125,5. Когда вы его введете в компьютер, то число в нем будет иметь другой вид. Например, 125,500001. Даже просто из-за ошибок округления. Если теперь вы начнете в программе сравнивать хранимое там число с константой `125.5`, у вас ничего не выйдет, т. к. числа, фактически, разные. Поэтому числа с плавающей точкой сравниваются не на полное совпадение, а на совпадение с точностью до некоторого значения. Например, пусть 125,500001 хранится в некоторой переменной `flo`. Чтобы число попало в эту переменную, мы должны его объявить в программе, и компилятор выделит ему память: `float flo = 125.5;`. Или другим способом: `float flo; flo = 125.5;`. Тогда сравнение содержимого `flo` и значения 125,5 можно выполнить так: если `(flo-125.5)<0.01`, то величины равны.

Десятичный тип данных

В жизни, как известно, все сложнее, чем на бумаге. Это касается и чисел. Например, вы работаете с бухгалтерскими документами, где, если потеряешь хоть одну копейку, на которую не сходится бухгалтерский баланс, никто у вас такой баланс не примет. Скажут, мол, мы ничего не знаем и знать не хотим про какие-то там ваши компьютеры. Тем более, что они работают неточно. Ищите, мол, куда вы дели эту копейку, а найдете, приходите. Но не долго ищите, иначе лишим премии за квартал. Все. Что делать? Вы бежите к программисту и начинаете у того выяснять причины. А он пожимает плечами: мол, с целыми числами я не могу работать, т. к. у вас в бухгалтерии числа нецелые, поэтому работаю с нецелыми. А те в компьютере почти всегда неточны, и поэтому в результате многочисленных вычислений накапливается ошибка. Никакие искусы с округлениями чисел не спасают. Скажите, мол, спасибо, что только одну копейку потеряли, а могли бы и больше. В добрые докомпьютерные времена так и бывало часто. Выходили из положения кто как мог. Но вот те, кому надо было все-таки продавать компьютеры, додумались до такого типа данных, который бы объединял в себе достоинства целых и нецелых чисел: с одной стороны, не давал бы ошибок вычисления, как это происходит с целыми числами, а с другой стороны, мог бы работать и с дробными числами. Этот тип назвали `decimal` (десятичный). Но чудес не бывает: выиграл в одном, проиграл в другом.

Данные этого типа замедляют работу компьютера. И порой — значительно. Поэтому этот тип данных и используется, в основном, в бухгалтерских расчетах, где никаких копеек никогда терять нельзя, не то вами займется налоговая инспекция. Другой недостаток этого типа данных в том, что их нельзя использовать в качестве различных счетчиков в программах, т. к. этот тип имеет все-таки дробную часть. Объявляются в программе переменные этого типа по общему правилу объявления:

```
тип имя_переменной
```

Вот три варианта такого объявления и пояснение, к чему это приводит:

```
decimal d1;
decimal d2 = 2;
decimal d3 = 2M;
```

В первом случае для переменной `d1` компилятор при компиляции программы выделит только память и на этом успокоится. Переменная имеет неопределенное значение (в ней просто какой-то мусор, т. к. она могла попасть на участок памяти, который хранил какие-то данные). В программах бывают переменные, которые, если они не определены при их объявлении, т. е. им не присвоены никакие значения, то компилятор их сам инициализирует значениями, принятыми у него по умолчанию. Для чисел это обычно ноль. Но в C# так называемые локальные переменные (те, что объявлены внутри какой-то функции, о них поговорим позже) не инициализируются компилятором, если они не инициализированы при их объявлении. Поэтому может случиться, что при использовании такой переменной с неопределенным значением возникнет ошибка вычисления. Отсюда компилятор станет напоминать вам, что переменная `d1` не определена. Этот факт демонстрирует рис. 3.1.

Из рисунка видно, что переменная `a` объявлена, но не получила первоначального значения. Когда мы вслед за ней написали команду `b=a;`, компилятор выдал ошибку компиляции. А если бы `a` была инициализирована, ошибки бы не было. В случае объявления `decimal d2 = 2;` все вроде бы нормально, но не совсем. Дело в том, что число `2` по умолчанию рассматривается как тип `int`, поэтому компилятор вынужден будет формировать команды преобразования числа типа `int` в число типа `decimal`. На это уходит дополнительное время компьютера.

А вот объявление `decimal d3 = 2M;` уже не требует никаких дополнительных преобразований. Буква `M` после константы `2` указывает на то, что эта константа — типа `decimal`.

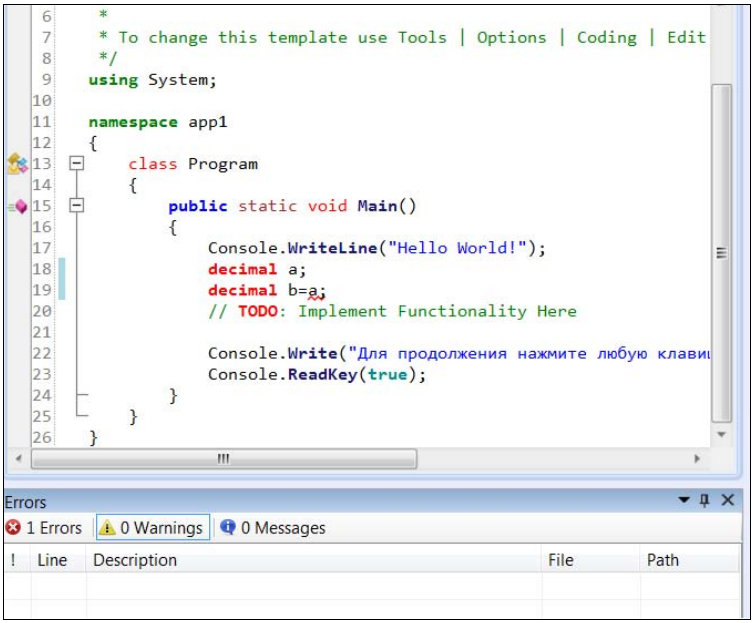


Рис. 3.1. Результат объявления `decimal`-переменной без ее инициализации

Первые программы

Теперь, когда мы научились объявлять числовые переменные, можем составить программу для работы с такими числами. Ранее мы договорились, что пока будем создавать консольные приложения, т. е. приложения без графического интерфейса между пользователем приложения и самим приложением. Заготовку такого приложения мы уже создали ранее. Теперь наполним ее содержанием. Рассмотрим программу вывода на экран таблицы температур по Фаренгейту и Цельсию.

Формула перевода температур такова:

```
C = (5 / 9) * (F – 32)
```

где `C` — это температура по шкале Цельсия, а `F` — по шкале Фаренгейта. Задается таблица температур по Фаренгейту: 0, 20, 40, ..., 300. Требуется вычислить таблицу по шкале Цельсия и вывести на экран обе таблицы. В вышеприведенной формуле символ `*` — знак операции умножения.

Для этого выполните следующие действия:

1. Создаем заготовку консольного приложения. Это вы выполнили ранее. Обратите внимание, что когда в тело `Main()` вы начнете вставлять свои команды, само приложение среда SharpDevelop автоматически будет сохранять. Но можно его сохранять, пользуясь и командами **Save**, **Save All** меню **File** среды. Введите код новой программы в тело главной функции (листинг 3.2).

Листинг 3.2

```
/* Created by SharpDevelop.
 * User: user
 * Date: 09.11.2012
 * Time: 14:56
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers.
 */
using System;

namespace app1
{
  class Program
  {
    public static void Main()
    {
      Console.WriteLine("Таблица температур по Фаренгейту
                                                  и Цельсию:");
      Console.WriteLine("По Фаренгейту  По Цельсию");
      int lower, upper, step;
      double fahr, cels;
      lower=0;
      upper=300;
      step=20;
      fahr=lower;
      while(fahr <= upper)
      {
        cels=(5.0/9.0)*(fahr-32.0);
        Console.WriteLine("{0:f}        {1:f}", fahr, cels);
        fahr=fahr+step;
      }
```

```
      Console.Write("Для продолжения нажмите любую клавишу>");
      Console.Read();
     }
   }
}
```

2. Откомпилируйте приложение. Для этого нажмите управляющую клавишу <F8>. Можно также в меню **Build** среды разработки SharpDevelop выбрать команду **Build Solution**. Справа от названия команды вы увидите "горячую" клавишу — <F8>, т. е. ту клавишу, при нажатии которой можно не бегать по меню, а сразу выполнить необходимое. Если после нажатия клавиши <F8> ошибок компиляции не будет, то в самом низу рабочего стола в строке состояния среды вы увидите сообщение "Build finished successfully" ("Построение прошло успешно"), т. е. компилятор построил exe-модуль для исполнения. Осталось только модуль запустить "горячей" клавишей <F5> и получить результат (рис. 3.2). Но можно и через главное меню вызвать приложение на выполнение: **Debug | Run**.

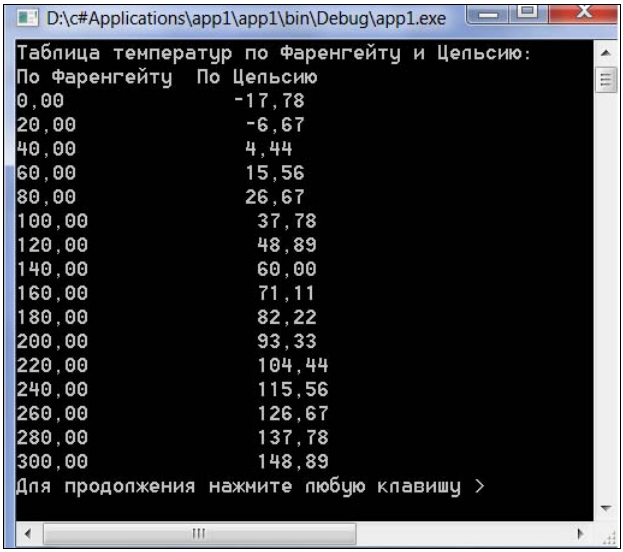


Рис. 3.2. Результат расчета температур по Фаренгейту и Цельсию

Debug — это отладчик среды. Без отладчика вообще очень трудно добиться работоспособности программы. Отладчик позволяет выполнять программу по шагам и смотреть каждый раз содержимое нужных вам

переменных. То есть вы смотрите, в переменной то значение, которое должно быть по вашим расчетам, или нет. Если не то, начинаете снова проходить строку за строкой и смотреть, как что формируется. Когда вы нажали клавишу <F5>, то запустили приложение на выполнение в режиме отладки. Чтобы заставить приложение остановиться в нужной вам точке, вы должны эту точку пометить: надо щелкнуть мышью в самой левой колонке текста программы (левее номеров строк программы). При этом в месте щелчка появится темно-красный кружок, а вся помеченная строка тоже подкрасится тем же цветом. Когда вы запустите приложение, оно остановится на помеченной строке. Вам остается навести курсор мыши на нужную переменную, немного подождать, пока всплывет ее значение. Чтобы продвинуться в выполнении на следующую строку, нажмите клавишу <F10>. Если вы стоите на вызове некоторой функции (о чем мы поговорим позже), то, нажав клавишу <F10>, вы целиком выполните функцию. А вдруг ошибка находится именно в теле этой функции? Как забраться в режиме отладки в тело функции? Для этого надо, находясь на заголовке функции, нажать не <F10>, а <F11>. Попадете внутрь функции. А внутри уже можно двигаться по строкам, нажимая клавишу <F10>. Все эти манипуляции с "горячими" клавишами отражены в меню **Debug** среды разработки SharpDevelop.

Но, как правило, компиляция с первого раза проходит редко. Поэтому в окне **Errors** выдаются ошибки компиляции. В графе **Line** окна указывается номер строки текста программы, в которой (в строке) обнаружена ошибка. Каждая ошибка имеет свой номер. Перечень ошибок компилятора и пояснение к ним можно найти в Интернете по адресу **http://msdn.microsoft.com/ru-ru/library/ms228296.aspx**.

Приступим к разбору нашей первой программы (см. листинг 3.2).

Первые две строки — это системная консольная функция `WriteLine()`, о которой мы говорили ранее. Она выводит сообщение на консольное устройство вывода, перебрасывая курсор консольного окна на следующую строку в первую позицию. Почему я сказал "выводит сообщение на консольное устройство вывода", а не просто выводит на экран? Дело в том, что исполняемая среда по умолчанию назначает клавиатуру и экран в качестве стандартных устройств ввода-вывода. Когда вы выполняете функцию `Console.Read()`, например, то в среде заложено уже, что чтение пойдет именно с клавиатуры. А когда выполняете, например, `Console.WriteLine()`, то в среде заложено уже, что вывод пойдет именно на экран, на стандартное устройство вывода.

Продолжим изучение программы. Строки

```
int lower, upper, step;
double fahr, cels;
```

объявляют переменные, с которыми предстоит работать программе. Именовать переменные надо так, чтобы было (и не только вам одному, но и, возможному, сопровождающему в будущем вашу программу) понятно, какие данные будут содержаться в этих переменных. В программе заданы типом `int` нижняя и верхняя границы изменения температуры по Фаренгейту и шаг изменения температуры. Заметьте, что одним типом объявлены несколько переменных через запятую. Объявление должно *обязательно* завершаться *точкой с запятой* (и не только объявление, если внимательно посмотрите на программу). По точке с запятой компилятор находит конец языкового предложения и начинает его расшифровывать. Если вы поставите точку не в том месте или вообще не поставите, компилятор все равно станет ее искать и объединять в одно языковое предложение все, что встречает по пути. Потом примется за расшифровку и увидит, что перед ним полная абракадабра. Выдаст вам кучу ошибок.

В переменных `fahr`, `cels` будут находиться текущие (получаемые во время расчета) значения температуры по Фаренгейту и по Цельсию. Отметим, что они заданы одним из типов с плавающей точкой. Почему не `float`? Если бы мы их объявили как `float`, то вот что бы получилось при компиляции: ниже, на участке расчета температуры по Цельсию в расчете участвуют две константы (5 и 9 с нулевой дробной частью). Эти числа специально взяты в виде записи с нулевой дробной частью, потому что они участвуют в операции деления `/`. А эта операция имеет свою специфику: если ее операнды объявлены как целый тип, то деление целых чисел считается операцией без остатка. Поэтому в этом случае деление 5:9 даст в результате ноль, поскольку целая часть этого частного равна нулю (а она и считается результатом операции над целыми числами), а дробная часть будет отброшена. А нам-то нужно правильное значение. Поэтому мы как бы обманываем компилятор: делаем у каждого числа нулевую дробную часть, не изменяя тем самым значение числа, а компилятор, когда увидит точку, посчитает это число за число с плавающей точкой. Но тип он присвоит константам не `float`, а `double`. Так в нем заложено. Но по формуле расчета (помним, что мы предположили, что переменные `fahr` и `cels` объявлены типом `float`) из `fahr` (типа `float`) вычитается константа `32.0` (типа `double` — раньше уже отмечалось, что компилятор константам с плавающей точкой по умолчанию присваивает тип `double`). Получается, что компилятор должен построить блок преобразования результата типа `float` в результат типа `double`. Он этого делать не умеет и выдает ошибку компиляции.

Далее назначаются начальные значения нижней и верхней границам диапазона температур по Фаренгейту и шагу изменения этой темпера-

туры в заданном интервале. За этим всем следует так называемый оператор цикла с именем `while` и телом, ограниченным фигурными скобками. В самом теле находятся операторы расчета, которые должны выполняться столько раз, сколько им задаст оператор `while`. А мы уже сами должны задать оператору `while`, сколько раз ему прокручивать свое тело. Прокручивать он будет до тех пор, пока в его заголовке (выражение в скобках, идущих за словом `while`) будет выполняться заданное там условие. Условие имеет вид: `fahr <= upper`. То есть, пока значение температуры по Фаренгейту не превзойдет верхней границы интервала своего изменения, `while` будет выполнять операторы в своем теле. Какой вывод? Наверно, надо постоянно менять содержимое переменной `fahr` на заданный шаг, чтобы переходить к новым точкам интервала. Поэтому перед `while` стоит оператор `fahr = lower;`. Он пересылает (= — это не знак равенства, а знак присвоения) значение нижней границы изменения температуры по Фаренгейту в переменную `fahr`. Уже потом начинает крутиться цикл `while`: вычисляется значение переменной `cels` по заданной формуле, значения по Фаренгейту и по Цельсию выводятся в плавающем формате (т. е. с точкой) на экран, берется следующая точка интервала изменения температур путем добавления к значению `fahr` величины шага по таблице температур, и управление передается в заголовок оператора `while`. В заголовке проверяется, не превзошло ли текущее значение `fahr` верхней границы `upper`. Если не превзошло, управление передается снова на выполнение тела `while`. Если верхняя граница превзойдена (кстати, `fahr` называется переменной цикла. Говорят, что цикл организован по переменной `fahr`), управление выполнением программы передается на строку, следующую за телом `while`. А эта строка содержит оператор вывода сообщения, чтобы нажали любую клавишу, потому что следующий оператор — оператор задержки экрана: оператор ждет, пока не будет введен любой символ с клавиатуры. Этот вопрос мы уже обсуждали. Как только вы нажмете любую клавишу, оператор завершит выполнение, а далее 0 — конец программы.

Осталось обратить внимание на средство вывода строки на экран. Вот такая запись этого средства

```
Console.WriteLine("{0:f}                {1:f}",fahr,cels);
```

означает так называемый форматный вывод строки. Функция `WriteLine()`, которая берется из среды `Console`, выдает на экран содержимое переменных `fahr` и `cels` не просто так, как они находятся в памяти, а преобразуя (форматируя) их в определенный вид.

При этом выражение

```
"{0:f}                {1:f}"
```

— это строка форматирования, значение которой заключено в кавычки. Элементы в фигурных скобках, в которых мы видим 0 и 1, означают порядковые номера выводимых на экран переменных, расположенных следом за строкой форматирования после запятой. То есть в данном случае (счет переменных идет от нуля) первой станет выводиться значение переменной `fahr`, за ней — значение `cels`. Можно было бы вывести сначала `cels`, а за ней `fahr`, поменяв 0 на 1, а 1 на 0. После указателей порядка вывода содержимого переменных через двоеточие указан формат выводимых значений. В данном случае это `f` — формат вывода числа с плавающей точкой. Все символы, которые находятся в строке форматирования и не заключены в фигурные скобки, выводятся без форматирования. Поэтому в программе пространство между форматами заполнено пробелами с целью разнести при выводе на экран результаты: чтобы они попали в свои колонки. Результат работы приложения показан на рис. 3.2.

Логический тип данных

Изменим наше первое приложение так, как показано на рис. 3.3.

Рассмотрим еще один тип данных — *логический тип*. Данные этого типа принимают только два значения: `True` (истина) и `False` (ложь). И больше никаких. Почему я так написал? Для тех, кто знает C/C++. В этих языках у булевых переменных еще было два значения: 1 и 0. Здесь этого нет. Когда используется этот тип данных? Пример показан на рис. 3.3. В операторе цикла `while` цикл завершается, когда условие, заданное в заголовке `while`, нарушается. Бывают очень длинные и сложные условия, которые не просматриваются сразу при отладке программы. В этом случае при отладке полезно сделать то, что и показано на рис. 3.3. Что значит, что условие, заданное в заголовке `while`, нарушается? Это означает, что условие было истинным, а потом стало ложным. Когда в заголовке `while` условие становится ложным, оператор завершает свою работу. На рис. 3.3 показано, что условие в заголовке `while` присвоено переменной `b`, которая объявлена логической (иногда говорят "булевой"). В режиме отладки (видны две *точки останова*) программы, когда программа остановилась на строке 29, навели курсор мыши на `b`. Подсказчик среды показал значение `b` в этот момент. Оно было равно `True`. Это значит, что цикл еще крутился.

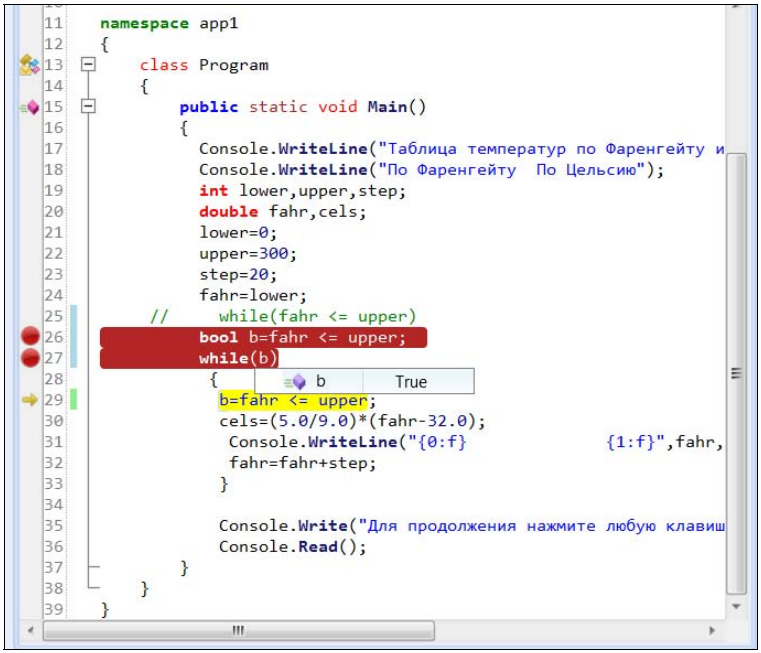


Рис. 3.3. Ввод в текст строки с типом `bool`

Оператор *for*

Кроме оператора `while`, организовать цикл позволяет и оператор `for`. Перепишем уже рассмотренную программу расчета температур в несколько ином виде (листинг 3.3).

Листинг 3.3

```
/* Created by SharpDevelop.
 * User: user
 * Date: 12.11.2012
 * Time: 12:05
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers.
 */
using System;
```

```
namespace app2
{
  class Program
  {
    public static void Main()
    {
      Console.WriteLine("Таблица температур по Фаренгейту
                                                    и Цельсию:");
      Console.WriteLine("По Фаренгейту  По Цельсию");

      int fahr;
      for(fahr=0; fahr <= 300; fahr= fahr + 20)
        Console.WriteLine("{0:f}        {1:f}", fahr,
(5.0/9.0)*(fahr-32.0));

      Console.Write("Для продолжения нажмите любую клавишу > ");
      Console.Read();
     }
   }
 }
```

Здесь для получения того же результата, что и в предыдущем случае, применен оператор цикла `for`. Тело этого оператора, как и тело оператора `while`, циклически выполняется (прокручивается). В нашем случае тело `for` состоит всего из одного оператора — `Console.WriteLine(...)`, поэтому такое тело не берется в фигурные скобки (если бы тело оператора `while` состояло только из одного оператора, оно тоже не бралось бы в скобки).

Видно, что запись программы приобрела более компактный вид. В заголовочной части оператора `for` расположены три выражения, из которых первые два оканчиваются точкой с запятой, третье — круглой скобкой, обозначающей границу заголовочной части `for` (компилятор понимает, что третье выражение завершилось). Как говорят, в данном случае "цикл идет по переменной `fahr`": в первом выражении она получает начальное значение, второе выражение — это условие окончания цикла (цикл закончится тогда, когда `fahr` примет значение, большее 300), а третье выражение изменяет параметр цикла на величину шага цикла.

Работа происходит так: инициализируется переменная цикла (т. е. получает начальное значение), затем проверяется условие продолжения цикла. Если оно истинно, то сначала выполняется тело оператора (в данном случае функция `Console.WriteLine(...)`), затем управление передается

в заголовочную часть оператора `for`. После этого вычисляется третье выражение (изменяется параметр цикла) и проверяется значение второго выражения. Если оно истинно, то выполняется тело, затем управление снова передается на вычисление третьего выражения и т. д. Если же второе выражение становится ложным, то выполнение оператора `for` завершается и начинает выполняться оператор, следующий непосредственно за ним, т. е. за его телом (а это — завершающая фигурная скобка `Main()`, означающая прекращение работы функции `Main()`).

В данном примере следует обратить внимание на аргумент функции `Console.WriteLine(...)`. Вместо обычной переменной там стоит целое выражение, которое сначала будет вычислено, а потом его значение выведется на устройство вывода. *Выражение* можно указывать в качестве аргумента функции, исходя из правила языка: *"В любом контексте, в котором допускается использование переменной некоторого типа, можно использовать и выражение этого же типа"*.

Изменим программу листинга 3.3 так, как показано в листинге 3.4.

Листинг 3.4

```
/* Created by SharpDevelop.
 * User: user
 * Date: 12.11.2012
 * Time: 12:41
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers.
 */
using System;

namespace app3
{
  class Program
  {
    const int upper = 300;
    const int lower = 0;
    const int step = 20;

    public static void Main()
    {
      Console.WriteLine("Таблица температур по Фаренгейту
                                                  и Цельсию:");
      Console.WriteLine("По Фаренгейту  По Цельсию");
```

```
      int fahr;
      for(fahr=lower; fahr <= upper; fahr= fahr + step)
        Console.WriteLine("{0:f}        {1:f}", fahr,
(5.0/9.0)*(fahr-32.0));

      Console.Write("Для продолжения нажмите любую клавишу > ");
// Press any key to continue...
       Console.Read();
    }
  }
}
```

Что нового получилось? В операторе `for` были заменены конкретные числа на значения переменных, а сами переменные получили начальные константные значения перед определением функции `Main()`. Результат расчета не изменился. Для чего все это? Мы повысили гибкость программы в смысле ее сопровождения. Представьте себе, что вы сопровождаете довольно большую программу, да еще и чужую. То есть вы в ней, так сказать, плаваете. В нее разработчик понапихал (иначе не скажешь) массу конкретных цифр. Приходит к вам тот, для кого вы сопровождаете эту программу (т. е. программа работает на этого человека или организацию), и говорит вам, что такие-то цифры, заложенные ранее в программу, надо заменить другими. Что вы станете делать? Вы возьмете текст программы и начнете в нем долго и упорно искать цифры, которые надо заменить. При этом будете вспоминать разработчика не очень лестными словами. Вот чтобы этого не случалось, лучше все константы, участвующие в расчетах, выносить в одну область программы (желательно, в начало), чтобы можно было в дальнейшем без особого труда изменять их значения. Форма записи таких величин в виде `const int upper = 300;` и место их записи — это требования классов, которые мы будем рассматривать в дальнейшем. Так как наша программа оформлена как класс с именем `Program`, то константы в этом классе описаны как константные данные по требуемому формату.

В заключение отметим, что наряду с оператором цикла `while` имеется оператор цикла `do while`. Он работает практически так же, как и `while`, за одним лишь исключением: `while` может сразу заканчиваться, не выполняя ни разу своего тела, если первоначально условие в его заголовке ложно. Работа оператора начинается с проверки условия в его заголовке: если оно ложно, оператор не выполняется. Но встречаются задачи, в которых надо применить `while`, но чтобы и при ложном начальном условии тело выполнилось хотя бы один раз. Вот это и делает оператор `do while`.

Вид такого оператора следующий:

```
do
{
  // тело
}
while (условие);
```

Обратите внимание, что после заголовка, который стоит уже в конце оператора, находится точка с запятой, чего в обычном `while` не было.

Символьные типы данных

Любая программа должна работать, в общем случае, не только с числами, но и с текстами. С самого начала существования компьютеров перед человеком стояла задача — разработать язык общения с компьютером. Этим языком являются таблицы кодировки символов текста. Здесь имеется в виду обобщенное понятие слова "Текст": в текст могут входить символы различных национальных алфавитов, другие (вспомогательные) символы, необходимые при обработке текста, символы музыки, математики и т. п.

Кодировка — это соответствие символов машинным кодам. Первые таблицы кодировки обеспечивали кодирование 128-ми символов. Однако впоследствии оказалось, что этого недостаточно. Появились таблицы, обеспечивающие кодирование 256 символов (обратите внимание, что 128 — это полбайта, а 256 — целый байт). Вот, например, широко известная таблица перекодировки ASCII (аббревиатура организации, утвердившей эту таблицу для общего пользования) обеспечивает кодирование как раз 256 символов. В ней каждый символ занимает один байт. Каждый символ алфавита пронумерован и имеет свой двоичный код. Коды изменяются в пределах от 00000000 до 11111111. Стандартных символов в таком алфавите 128, остальные — это дополнительные символы и символы национальных алфавитов. В первой таблице ASCII было всего 128 символов, и предназначалась она вовсе не для компьютеров, а для телетайпа. Но фирма IBM, создавая компьютер, решила воспользоваться этой таблицей кодировки. А когда компьютеры стали распространяться по миру, то 128-ми символов стало не хватать, и принялись вводить национальные таблицы для разных стран и языков. Но в каждом деле должен быть порядок. А получилось, что таблиц насоздавали столько, что их стало больше, чем стран и языков. Например, только для одной кириллицы существует несколько вариантов таблиц. Все

это усложнило и без того непростую ситуацию, и даже сейчас не все современные программы хорошо распознают кодировки, и вместо текста мы, порой, видим набор непонятных символов.

Далеко ходить не надо: возьмите хотя бы почтовую программу The Bat! и посмотрите, сколько там вариантов таблиц перекодировки текста. Со временем человечество осознавало, что существующих кодировок недостаточно. Например, символы китайского языка не умещались в стандартную 8-битовую (однобайтовую) кодировку. Решено было сделать таблицу, в которой под код выделялся бы не один, как ранее, а два байта (16 бит). Такая таблица была создана и назвали ее Unicode. Символы Unicode используют 16 бит, поэтому в таблице можно разместить 65 536 символов. Этого количества кодов хватает пока для кодирования всего, что надо кодировать сегодня, чтобы обрабатывать на компьютере. По данным Интернета в настоящее время используется порядка 49 тыс. символов для кодирования.

C# рассматривает текст в двух видах: как отдельные символы и как строки символов. Отдельные символы помечаются типом `char`, строки — типом `string`. Заметим, что имена типов здесь, как и имена типов для числовых и логических данных — это специальные системные ключевые слова. На самом деле, как мы увидим позже, типы — это классы со своими именами. Ключевые слова введены в язык для упрощения обращения с ними.

Тип *char*

Переменная типа `char` может хранить только один символ (точнее, она хранит в себе код символа по таблице Unicode). Объявляется такая переменная по общим правилам:

тип имя_переменной

Например, `char c;`. Чтобы такой переменной присвоить значение (т. е. только один символ и не более), надо после знака присвоения указать необходимый символ, заключенный в апострофы (одинарные кавычки):

```
char c='a';
```

Как отмечалось ранее, вы можете присваивать любой символ из любого языка. Хоть китайский иероглиф. Но надо помнить, что тип не содержит в себе описание шрифта (для этого существуют другие средства). Поэтому, если вы попытаетесь вывести на экран иероглиф, получите нечто другое.

Литералами принято называть текстовые константы. В этом смысле запись `char c='a';` задает символьную константу (литерал) `c`.

Некоторые символы из таблицы кодировки не имеют отображения на экране (или на бумаге), потому что специально определены как управляющие. То есть они управляют вводом-выводом текста или служат для расшифровки значений других символов (например, к какому языку относится символ). Фрагменты таблицы Unicode показаны на рис. 3.4.

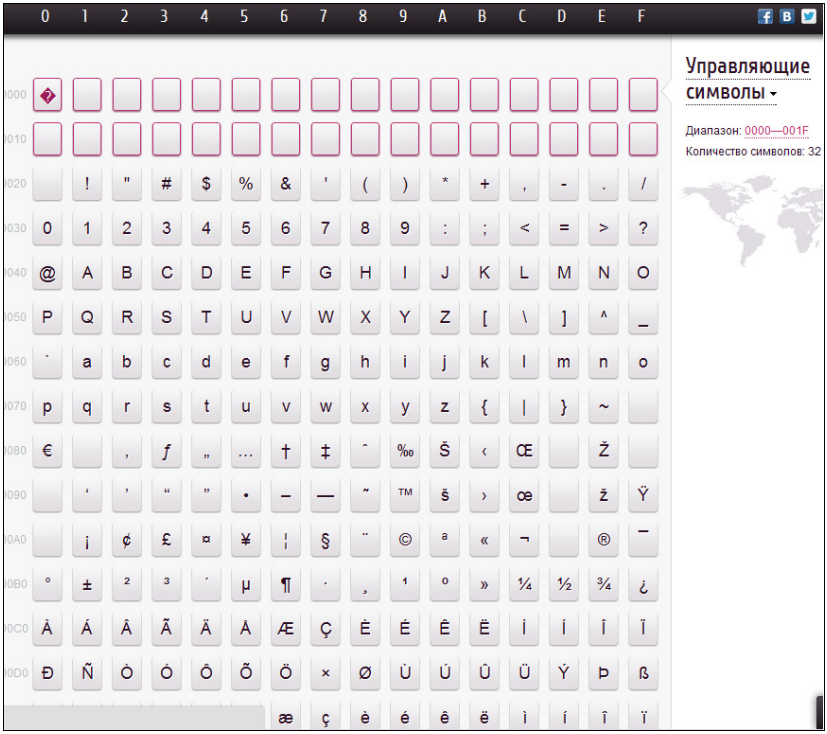


Рис. 3.4. Фрагмент таблицы Unicode

Таблицу легко найти в Интернете.

Коды первых 32-х символов таблицы — коды управляющих символов. Далее идут подтаблицы в разрезе языков: латиница, греческий и т. д.

Переменной типа `char` можно присваивать, наряду с символами, как показано выше, и их коды напрямую. Только эти коды должны иметь форму так называемых esc-последовательностей (или управляющих последовательностей). Все принтеры, например, используют такие последовательности, получая их от компьютера для управления печатью.

Запись `char MyChar = '\x0058';` означает, что переменной `MyChar` будет присвоен шестнадцатеричный (символ `x` указывает на это) код 0058 (символ `X`, как показано на рис. 3.5).

Рис. 3.5. Символ по таблице Unicode

Этот факт легко проверить, если приведенное объявление символа вставить в текст приложения, ниже вставленной строки установить точку останова программы и запустить программу в режиме отладки. Когда программа остановится на точке останова, проверьте содержимое объявленного символа, наведя на него курсор мыши (рис. 3.6).

Переменной `char` можно присваивать и целые числа, поставив принудительное преобразование их в тип `char`:

```
char MyChar = (char)88;
```

Здесь десятичное число 88 (ясно, что оно играет роль кода 88) принудительно преобразуется в тип `char` по указанному формату: `(char)`. Можете проверить по программе, вставив в нее указанную строку, как это

сделано ранее с помощью отладчика и наведения курсора мыши на переменную. Результат будет тем же — X. Почему? Потому что таблица Unicode — таблица шестнадцатеричных кодов. А у нас число 88 — десятичное. Следовательно, среда программирования переведет это число в шестнадцатеричный формат, чтобы воспользоваться таблицей Unicode. А 88 в шестнадцатеричном формате равно 58 (для перевода можете воспользоваться программой Калькулятор, поставляемой с Windows). А это, как мы видели раньше — код символа X.

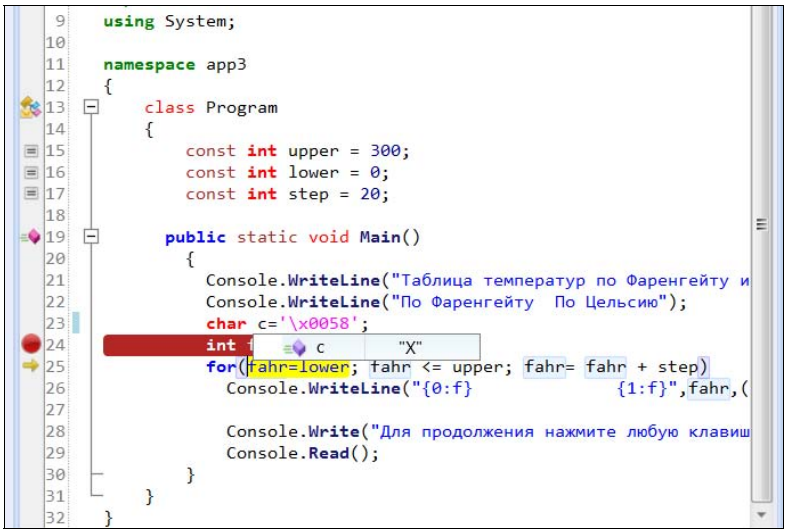


Рис. 3.6. Проверка с помощью программы факта присвоения символу некоторого кода вместо литерала

Ранее мы присвоили переменной типа `char` непосредственно значение шестнадцатеричного кода в виде: `char MyChar = '\x0058';`. Но такой код можно присвоить и в другом виде:

```
char MyChar = '\u0058';
```

Здесь символ `u` после обратного слеша (\) указывает на Unicode.

Тип `char` можно явно преобразовывать в типы `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, `decimal`. Пример показан на рис. 3.7. То есть таким способом мы можем получить код символа в десятичной системе счисления.

Однако обратное преобразование не возможно.

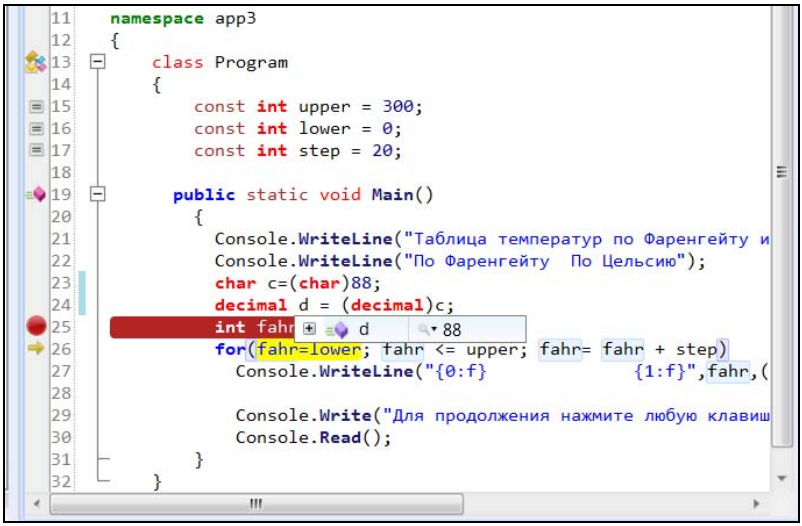


Рис. 3.7. Преобразование типа `char` в числовые данные

Управляющие символы таблицы Unicode могут записываться не только в форме четырехразрядного шестнадцатеричного числа с обратным слешем в начале, но и символьном виде, для облегчения пользования ими. Перечень некоторых таких символов приведен в табл. 3.2.

Таблица 3.2. *Управляющие символы Unicode*

Вид	Пояснение
`'\\'`	Обратный слеш
`'\0'`	Завершающий нуль-символ строки
`'\a'`	Алерт (предупреждение)
`'\b'`	Бэкспейс (возврат назад на одну позицию)
`'\f'`	form feed (прогон страницы)
`'\n'`	Переход на новую строку
`'\r'`	Возврат каретки
`'\t'`	Горизонтальная табуляция
`'\v'`	Вертикальная кавычка
`'\u'`	Unicode-последовательность

Таблица 3.2 *(окончание)*

Вид	Пояснение
'\''	Одинарная кавычка (апостроф после слеша), нужна для символьных литералов
'\"'	Двойная кавычка (после слеша), нужна в строковых литералах

Тип *string*

Этот тип объявляет (задает) строковые данные, которые дают возможность работать с текстом. Объявление в программе — обычное:

```
тип имя_переменной_для_будущей_строки
```

Например:

```
string s;
```

Переменной `s` можно присваивать константу (литерал), но в отличие от типа `char`, константа берется в *двойные кавычки*, чтобы компилятор различал, где символ, а где строка символов, хотя бы и из одного символа. Например,

```
string s="строка";
string s="a";
char c='a';
```

В `s` и `a` будут разные данные, хотя бы потому, что они в отведенной им памяти размещаются по-разному. Однако это не может быть критерием. Критерием может быть тот факт, что показывает, сколько на самом деле символов содержится в `s` и `a` соответственно. В `a` мы ранее видели — один символ, точнее, там находится код символа. А в `s`? Применим уже известный нам прием: вставим в какую-нибудь работающую программу (в текст, разумеется) такие строки:

```
string s="a";
int n=s.Length;
```

Здесь `Length` — это (говорим пока условно) функция, вычисляющая длину `s`.

Поставим точку останова программы сразу за второй строкой. После компиляции и пуска программа остановится после строки `int n=s.Length;`. То есть значение `n` уже сформировано. Наведем курсор мыши на `n`. И что всплывет? Всплывет единица! Следовательно, длина

`n` — тоже один символ, как и длина `s`. Тот, кто изучал C++, будет в недоумении. В полном. Однако и длина строки не может, увы, служить показателем отличия в типах `char` и `string`.

Посмотрим на структуру, скажем так, ячейки памяти, в которой размещается строка. Знаем, что в ячейке должны размещаться символы из таблицы Unicode. Сколько? От "ничего" до "сколько хочешь". Когда мы пишем `string s="";` (две кавычки подряд), то в `s` и будет "ничего". Пустая строка, как говорят. По идее, ее длина должна быть нулем. Проверим это по программе вышеописанным способом, задав строку `string s="";`. Действительно, `n=0`. А что касается, "сколько хочешь", то содержимое переменной типа `string` действительно может динамически изменяться в сторону увеличения и/или уменьшения. Что же касается роста в сторону увеличения, то здесь возможны только технические ограничения.

Вспомним теперь, что числовые и булевы типы данных не раздуваются и не сужаются. Сколько им памяти выделено, столько и остается за ними. Такие типы называют *типами-значениями*. Они хранятся в стековой памяти программы. За памятью, занимаемой ими, следить особенно не надо: они удаляются из памяти по мере того, как завершается работа того блока, в котором они объявлены. А вот динамически меняющие свои размеры переменные (к таким, как мы видим, относится тип `string`) требуют особого надзора за собой и поэтому размещаются не в блоке, где они объявлены, а в специальном пространстве, которое разработчики назвали *кучей*. Типы, данные которых размещаются в куче, называются *ссылочными*, т. к. наряду со значениями, присущими таким типам данных, как набор символов Unicode в `string`, они содержат и ссылку, т. е. адрес, связывающий их со своим местом в стеке (а в стеке же хранятся ссылки на ссылочные данные, а их начало — в управляемой куче), чтобы специальный механизм очищал кучу от их присутствия, когда в этом возникнет необходимость, и куча не переполнялась. Поэтому ячейка памяти для строки состоит из двух частей: собственно строки и адреса ее начала в стеке. Со ссылками очень удобно работать: если вы копируете одну строку в другую, то не надо перегонять огромное количество символов из одной строки в другую. Достаточно скопировать в другую строку только ссылку, которая указывает, где находятся данные, которые, якобы, копируются из исходной строки. Вот такова разница (и существенная) между строкой из одного символа и переменной типа `char`, по определению состоящей тоже из одного символа.

Для строк определены операции сравнения: `==` (равно), `!=` (не равно), `+` (сцепление (конкатенация)).

Например,

```
string a = "hello";
string b = "h";
b = b+"ello";  // Добавили значение литерала
               // к содержимому b
Console.WriteLine( a == b );  // Вывод на печать строки –
                              // значения выражения
```

На экран выдастся значение `True`.

Можно написать:

```
string s="Доброе " + "утро";
bool b = (s == "Доброе утро");
```

Если проверить по программе, получим, что `b` равно `True`. Следует отметить, что надо аккуратно переносить текст из WordPad в редактор SharpDevelop: коды кавычек у них разные. Кавычки в редакторе надо поставить его, редактора.

Литералы переменной типа `string` можно записывать в двух формах: текст заключать в кавычки и помещенный в кавычки текст предварять символом `@`. Литерал может содержать любые символы, в том числе и управляющие последовательности. Например,

```
string a = "\\\u0066\n";
```

В таком литерале обрабатываются все символы, включая и управляющие. В литерале же, предваренном символом `@`, управляющие символы не обрабатываются. Имеем:

```
string a = "\\\u0066\n Проба";
string b=@"\\\u0066\n Проба";
```

Результат: `a="\f\n Проба"`, `b=\\\u0066\n Проба`.

Программы работы с переменными типа *string*

Когда объявляется переменная типа `string`, например `string s;`, то чтобы увидеть все функциональные элементы, определенные в C# для ключевого слова `string` (а на самом деле — для класса `String`, как мы увидим позже), надо воспользоваться подсказчиком среды программирования. Наберите имя переменной `s` и поставьте точку после имени (так задаются члены класса и/или объекта этого класса — увидим позже). Подсказчик моментально высветит вам перечень элементов, с помощью которых можно взаимодействовать с `s` (рис. 3.8).

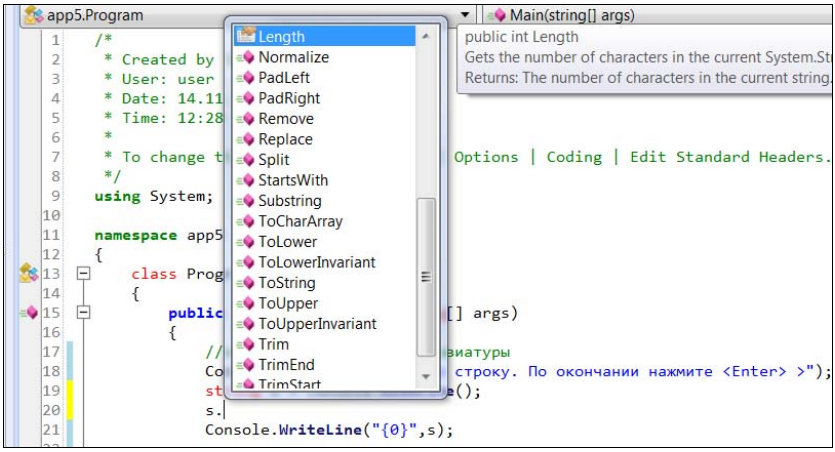


Рис. 3.8. Использование подсказчика для определения функций работы со строками

Программа для проверки некоторых базовых функций работы со строками

Создадим консольное приложение app4.cs для проверки работы некоторых функций обработки строк. Сначала, как обычно, создайте консольную заготовку (шаблон) типа Solution, выполнив команды **File | New | Solution | Console Application** и задав в нижней части открывшегося диалогового окна путь к папке, в которой станет храниться создаваемая программа. Затем нажмите кнопку **Create** (Создать) — получаем на экране заготовку. Уберите из нее тело функции `Main()`, а вместо него вставьте свои операторы. В результате получим исходный текст программы, приведенный в листинге 3.5.

Листинг 3.5

```
/* Created by SharpDevelop.
 * User: user
 * Date: 14.11.2012
 * Time: 10:51
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers.
 */
using System;
```

```
namespace app4
{
  class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine ("=> Основные операции с типом string");

      string firstName = "Николай";
      Console. WriteLine("Имя: {0} ", firstName);
      Console.WriteLine("Имя содержит {0} символов.", firstName
.Length) ;
      // Определение длины firstName
      Console. WriteLine("Имя, записанное на верхнем
                регистре:  firstName.ToUpper ());
      // Перевод firstName в верхний регистр
      Console.WriteLine("Имя, записанное на нижнем
                регистре:{0}", firstName.ToLower ());
      // Перевод firstName в нижний регистр
      Console.WriteLine("Есть ли в имени буква й? :
               {0}", firstName.Contains("й"));
      // Функция проверки содержания подстроки в строке
      Console.WriteLine("Имя после замены символов:
               {0}", firstName.Replace ("ай", "я"));
      // Функция замены одной подстроки на другую
      Console. WriteLine("Для продолжения нажмите любой символ
>");
      Console.Read();
    }
  }
}
```

Пояснение. В программе в виде строкового данного задается некоторое имя (Николай). Затем с помощью функций обработки строк определяется длина этого имени, имя переводится в верхний и нижний регистры клавиатуры, проверяется, есть ли в имени буква "й", и заменяются две буквы "ай" на одну "я". Программа компилируется с помощью нажатия клавиши <F8>, выполняется путем нажатия клавиши <F5>. Результат работы программы показан на рис. 3.9.

Из программы ясен смысл названия функций, а также понятно, какие параметры в них задавать. Следует помнить, что `WriteLine()` перед выводом строки, при необходимости, вычисляет ее значение.

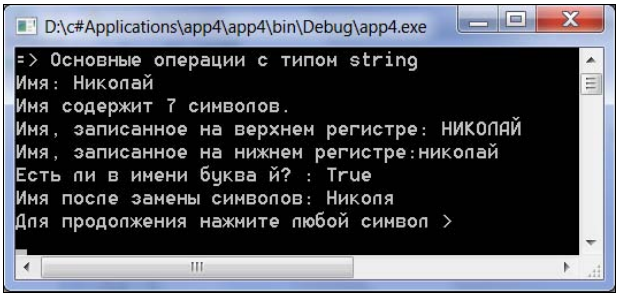


Рис. 3.9. Результат обработки строкового данного базовыми функциями работы со строками

Программа копирования символьного файла

Напишем программу, в которой входной файл будет вводиться с клавиатуры (входное стандартное устройство — клавиатура), а выводиться на экран (выходное стандартное устройство — экран). Текст программы представлен в листинге 3.6. Результат работы — на рис. 3.10.

Листинг 3.6

```
/* Created by SharpDevelop.
 * User: user
 * Date: 14.11.2012
 * Time: 12:28
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers.
 */
using System;

namespace app5
{
  class Program
  {
    public static void Main(string[] args)
    {
      // Копирование строки с клавиатуры
      Console.WriteLine("Вводите строку. По окончании нажмите
                                                    <Enter> >");
```

```
      string s = Console.ReadLine();
      Console.WriteLine("{0}",s);

      Console.WriteLine("Нажмите любую клавишу для продолжения
>");
      Console.Read();  // Задержка экрана
    }
  }
}
```

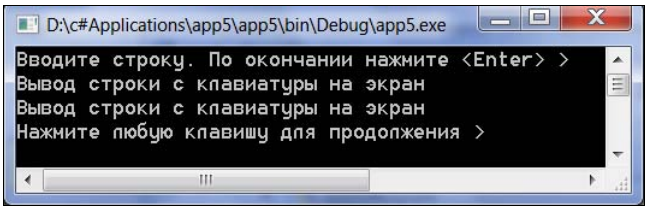


Рис. 3.10. Результат ввода строки с клавиатуры и вывода ее на экран

Программа очень проста за счет того, что всю работу выполняет функция `ReadLine()`. Она всегда ждет ввода очередного символа с клавиатуры. Как только какая-то клавиша нажата, ее значение вводится в переменную `s` и "приклеивается" справа к содержимому `s`. Если нажата клавиша <Enter>, ввод прекращается. В `s` — результат ввода. Как видно из рис. 3.10, среди символов введенной строки нет лишних. Это значит, что `ReadLine()` перед вводом очищает объявленную `string`-переменную.

Ввод текста

Здесь подразумевается ввод более одной строки и вывод всего введенного на экран. Каков будет алгоритм ввода? Надо будет ввести одну строку, нажать клавишу <Enter>, затем — вторую и опять нажать <Enter> и т. д. А когда кончится сам текст, надо ввод прекратить. Как можно прекратить ввод? Один из способов — ввести некий символ или группу символов, которых не будет во вводимом тексте, и каждый раз после ввода очередной строки проверять, не введена ли подобная подстрока. Выполнять такую проверку мы уже умеем из программы листинга 3.5. Но лучше было бы не связываться с символами или их комбинацией, а воспользоваться управляющими символами. В языке C++ существовал управляющий символ (комбинация клавиш <Ctrl>+<Z>) — признак конца файла ввода с клавиатуры. Хорошо бы им же воспользо-

ваться в C#. Однако в этом языке этот символ применяется иначе: при вводе через `ReadLine()` при нажатии <Ctrl>+<Z> функция выдает значение `null`. Вот на это значение и надо проверять ввод, чтобы его завершить, если вводится много строк. Текст приложения представлен в листинге 3.7. Результат работы — на рис. 3.11.

Листинг 3.7

```
/* Created by SharpDevelop.
 * User: user
 * Date: 14.11.2012
 * Time: 13:20
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers.
 */
using System;

namespace app6
{
  class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Вводите строки текста. После каждой —
<Enter>");
      Console.WriteLine("После последней — <Enter> и <Ctrl+z> ");
      string s="";
      while(s != null)
      {
        s=Console.ReadLine();
  //    Console.WriteLine("{0}",s);
      }
      Console.Write("Для продолжения нажмите любую клавишу > ");
      Console.Read();
    }
  }
}
```

Обратите внимание, что строка вывода на экран результата ввода закомментирована, чтобы не получалось двойного изображения: одно уже появляется от ввода. По этой программе весь введенный текст нигде не

запоминается, а демонстрируется только на экране. Мы еще не знаем, как сохранить его. Перед вводом мы делаем строку пустой, чтобы там случайно не оказалось ничего для нас неопределенного и тем — неприятного. Работу оператора цикла мы знаем из предыдущего материала. Этот оператор обеспечивает нам ввод строк до тех пор, пока не будет введен символ ^z (как от нажатия клавиш <Ctrl>+<Z>). В этом случае `ReadLine()` выдает значение `null` и ввод прекращается, потому что условие в заголовке `while` перестает выполняться.

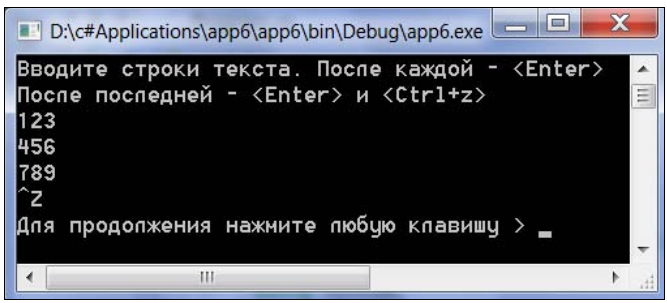


Рис. 3.11. Ввод текста и вывод его на экран

Подсчет количества введенных строк

Построим приложение обычным путем на основе консольной заготовки. Код приложения представлен в листинге 3.8.

Листинг 3.8

```
/* Created by SharpDevelop.
 * User: user
 * Date: 14.11.2012
 * Time: 18:12
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers.
 */
using System;

namespace app7
{
  class Program
```

```
  {
    public static void Main(string[] args)
    {
      // Подсчет количества введенных строк
      Console.WriteLine("Введите строку, нажмите <Enter>");
      Console.WriteLine("После ввода последней строки нажмите
<Enter> <Ctrl+z> >");
      string s="";
      int kol=0;
      while(s != null)
      {
        s=Console.ReadLine();
        kol++;
      }
      Console.WriteLine("Количество введенных строк={0}",--kol);
      Console.Write("Для продолжения нажмите любую клавишу >");
      Console.Read(); // Задержка экрана
    }
  }
}
```

Здесь следует пояснить несколько моментов. Количество введенных строк накапливается в числовой переменной `kol`. Сколько строк можно одновременно вводить этой программой? За два миллиарда. За отказ программы можно не беспокоиться. Метод накопления количества введенных строк прост. До цикла обнуляем счетчик `kol`, а в цикле после ввода каждой строки добавляем к счетчику единицу. Здесь мы встречаем незнакомую операцию `kol++`. Операция называется *инкрементной*. Существует и другой ее вид: `++kol`. Первая работает так: в переменную добавляется единица. А как работает второй вид, видно на примере декрементной операции. Это такая же операция, но только с минусом. После завершения цикла ввода мы должны счетчик числа вводов уменьшить на единицу, т. к. в него попала единица и после ввода признака конца ввода — нажатия комбинации клавиш <Ctrl>+<Z>. Эту операцию мы выполняем прямо в функции вывода результата на экран, т. к. эта функция может вычислять и выражения, в которые входит выводимая переменная. А выражение имеет вид `--kol`. Это другой вид декрементной операции `kol--`. Когда эта операция участвует в вычислениях некоторого выражения, то существенно, в каком месте (до или после имени переменной) стоят плюсы-минусы. Если нет участия в вычислении выражения, то можно ставить минусы-плюсы как до, так и после имени.

В нашем же конкретном случае имеется выражение, которое должно быть вычислено до того, как содержимое `kol` начнет выводиться. Поэтому надо писать `-kol`. В этом случае сначала от `kol` отнимется единица, а уж потом результат станет выдаваться на экран. Если бы мы вставили в вывод `kol--`, то сначала бы выводился `kol`, а потом бы шло вычитание. Результат работы приложения показан на рис. 3.12.

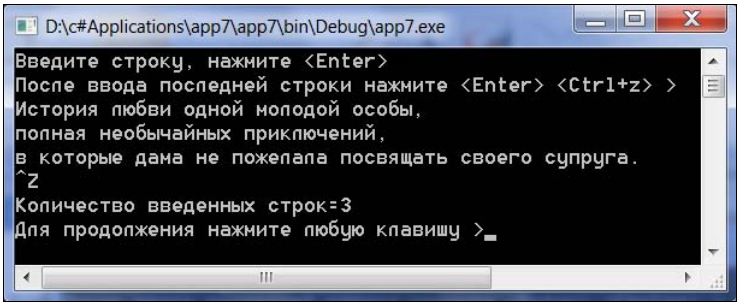


Рис. 3.12. Результат подсчета количества вводимых строк

Подсчет количества слов в тексте

Программа подсчета количества слов во введенном с клавиатуры тексте приведена в листинге 3.9. Результат работы программы — на рис. 3.13.

Листинг 3.9

```
/* Created by SharpDevelop.
 * User: user
 * Date: 14.11.2012
 * Time: 19:17
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app8
{
  class Program
  {
    public static void Main(string[] args)
```

```
    {
      // Подсчет количества слов во введенных строках
      Console.WriteLine("Подсчет количества слов в тексте \n");
      Console.WriteLine("Введите строку, нажмите <Enter>");
      Console.WriteLine("После ввода последней строки нажмите
                                          <Enter> <Ctrl+z> >\n");
      string s="";
      int w=0;
      int ind=-1;
      int ind1;
      while(s != null)
      { s=Console.ReadLine();
        // Проверяем в цикле по всей строке,
        // сколько пробелов встречается.
        // При каждой встрече добавляем
        // в счетчик слов единицу

       while((s!=null) && (ind != (ind1=s.IndexOf(" "))))
       { w++;
         s=s.Substring(ind1 +1);
       }
       w++; // Для учета последнего слова строки
      }
     Console.WriteLine("Количество слов равно  {0}", --w);
     Console.Write("Для продолжения нажмите любую клавишу >");
      Console.Read(); // Задержка экрана
    }
  }
}
```

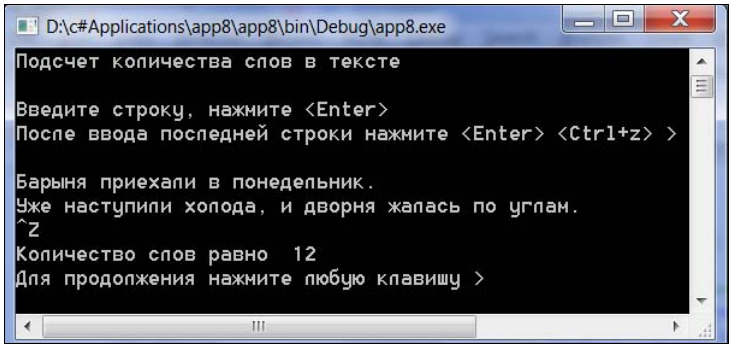


Рис. 3.13. Подсчет количества слов в тексте

Пояснение. В программе принято, что пробел является разделителем слов. Поэтому алгоритм такой: вводится строка текста и передается на анализ в цикл, в котором она просматривается на наличие пробела. Пробел ищется с помощью функции `s.IndexOf(" ")`, которая выдает номер позиции первого встретившегося пробела от начала строки. Если пробел не найден, функция выдает –1. Поэтому на случай сравнения результата `s.IndexOf(" ")` с –1 заготовлена константа `int ind=-1;`. Если пробел найден, то, начиная со следующей позиции, из строки выделяется оставшаяся часть строки и запоминается снова в переменной `s`. То есть `s` сужается, чтобы в оставшейся части можно было снова искать пробел. Поэтому организован цикл по строке. Если пробел найден, то в счетчик слов `w` добавляется единица. Когда от строки осталось последнее слово и в нем естественно уже нет пробела, внутренний цикл по строке завершается, но в счетчик слов `w` добавляется единица, чтобы учесть последнее слово строки, в котором не обнаружен пробел. Далее идет переход на внешний цикл, на ввод новой строки. В этой программе мы встречаем новую операцию `&&` — конъюнкцию или операцию "логическое И". Она выдает истину, если оба ее операнда истинны. В противном случае ее результат — ложь. Отсюда то, что в заголовке `while` во внутреннем цикле программы, читается так: "если при вводе строки не была нажата комбинация клавиш <Ctrl>+<Z> и в строке найден пробел, перейти на выполнение тела `while`". И еще один момент. Заметьте, что для вывода пробела на экран в необходимых функциях `WriteLine()` добавлены управляющие символы `'\n'` — перевод строки.

> ***Примечание***
>
> Рассмотренное приложение годится только для идеального текста, когда слова отделены друг от друга одним и только одним пробелом. На самом деле текст бывает "грязным": между словами более одного пробела или вообще — только знаки препинания, или присутствуют знаки табуляции и т. д. В приложении это не рассматривалось, чтобы не усложнять основную задачу — изучение языка, а не машинных алгоритмов.

Тип *var*

Начиная с версии Visual C# 3.0, объявляемые в области метода переменные могут иметь неявный тип `var`. Локальная переменная с неявным типом имеет строгую типизацию, как если бы тип был задан явно, только тип определяет компилятор на основании использования такой переменной в программе. Следующие два объявления `i` функционально являются эквивалентами:

```
var i = 10; // неявная типизация
int i = 10; // явная типизация
```

Допустим, если объявили `var i;`, а потом в программе компилятор встретит оператор `i=10;`, то переменная `i` получит тип `int`. Если же обнаружится выражение `i="строка";`, то переменная `i` получит тип `string`. Если в программе будет выражение `var i = DateTime.Now;`, то компилятор присвоит переменной `i` тип `DateTime` (дата-время, это мы рассмотрим позже).

Некоторые обобщения по объявлению и работе с переменными

Объявление констант

В C# любое выражение имеет значение и тип. Если имеем объявление `int n`, то разумно считать, что тип результата вычисления `n+1` также будет типа `int`. Но что можно сказать о типе константы `1`? Мы уже ранее видели при расчете температур по Фаренгейту и Цельсию, что как бы обманывали компилятор, задавая целое число `5` в виде числа с плавающей точкой, добавляя к целой части нулевую дробную часть: `5.0`. То есть обнаружили, что тип константы зависит (для компилятора) от ее значения. И от формы записи.

Любое целое число величиной, попадающей в рамки изменения целого, рассматривается как тип `int`. Числа, превышающие верхнюю границу максимума целого, трактуются как тип `long`. Любые числа с плавающей точкой рассматриваются как тип `double`.

В C# принято еще к форме записи константы добавлять некую (необязательную) букву. В табл. 3.3 показаны константы, объявленные с буквенными пометками в конце. Регистр букв значения не имеет. То есть записи `1u` и `1U`, например, равноценны.

Таблица 3.3. *Объявление констант*

Константа	Тип
`1.0`	`double`
`1.0F`	`float`
`1M`	`decimal`

Таблица 3.3 *(окончание)*

Константа	Тип
`true`	`bool`
`false`	`bool`
`'a'`	`char`
`'\n'`	`char` (символ новой строки)
`'\x1231'`	`char` (символ с шестнадцатеричным числовым значением 123)
`"My_string"`	`string`
`""`	`string` (пустая строка)

Пояснение. Если мы захотим задать константу для переменной типа `float`, то вынуждены будем записать ее в виде с поясняющей буквой `f` в конце. Например, `float d=1.2f;`. Почему? Если записать без `f`, то компилятор выдаст ошибку, т. к. константа с плавающей точкой по умолчанию имеет тип `double`, а `double` во `float` напрямую не преобразуется, о чем сказано ранее.

О преобразовании данных разных типов

В процессе проведения расчетов по программе требуется присваивать значения переменных одного типа переменной другого типа. Даже в некотором выражении, стоящем справа от знака присвоения, могут быть переменные разных типов. Например, имеем:

```
int a;
float b;
double c;
c=a+b;
```

Некоторые типы преобразований в языке выполняются автоматически, другие — с помощью специальных функций или операторов. Вообще преобразования бывают неявные, явные и преобразования с помощью вспомогательных классов, в которых находятся функции, выполняющие нужные преобразования. *Неявные преобразования* — это такие преобразования, когда не требуется специального синтаксиса для их задания, среда их выполняет сама, потому что нет угрозы потери данных. Например, преобразования от меньшего к большему целому типу. *Явные преобразования* уже требуют специальной записи (они еще называются

преобразованиями по приведению типов). Такие преобразования нужны тогда, когда при вычислениях имеется угроза потери данных. Преобразования с помощью вспомогательных классов требуются для преобразования несовместимых типов, например, строковых данных в числа и наоборот.

Пусть переменная типа `int` должна быть преобразована в переменную типа `long`. Это преобразование выполняется неявно, т. к. любое значение типа `int` может храниться в переменной типа `long`, и оба типа представляют собой числа. C# выполняет такое преобразование автоматически.

Однако обратное преобразование может вызвать проблемы. Некоторые значения, которые могут храниться в переменной `long`, не помещаются в переменной типа `int` (по определению самих переменных). Компилятор не знает, случится ли такая ситуация или нет в процессе расчетов, поэтому, если вы просто присвоите переменной типа `int` значение переменной типа `long`, компилятор выдаст ошибку. Но если вы точно знаете, что такое преобразование вполне допустимо и числа таковы, что потери части числа при присвоении не случится, то можете воспользоваться специальным *оператором приведения типов* (явный тип преобразования), который имеет вид: `(тип)`, где `тип` — тип данных, к которому приводится данное, стоящее за этим оператором. Например, имеем: `int a; long b;`. Запись `a=b;` вызовет ошибку компиляции. Но запись `a=(int)b;` пройдет. Оператор `(int)` — оператор приведения к типу `int` числа типа `long`. При приведении типов имя требующегося типа размещается в круглых скобках непосредственно перед преобразуемым значением.

Такая же ситуация складывается и с числами типов `float` и `double`. Первый тип преобразуется во второй автоматически, а второй — только принудительно, как только что показано с типом `long`. Например, `float aa=(float)1.2;` компилируется нормально, а `float aa=1.2;` дает ошибку, потому что константа `1.2` интерпретируется по умолчанию как тип `double`.

Целые числа могут быть преобразованы в числа с плавающей точкой автоматически, но обратное преобразование требует использования оператора приведения типов. Например, `double b=1.2; long a=12; a=(long)b;` компилятор пропустит, а `a=b;` нет. Ясно, что при таком преобразовании дробная часть числа, если она имеется, будет отброшена. Возникнет погрешность вычисления.

Все приведения к типу `decimal` и из него требуют применения оператора приведения типов. Все числовые типы могут быть преобразованы в другие числовые типы с помощью этого оператора.

Встроенные функции C# могут преобразовывать числа, символы или логические переменные в их строковые "эквиваленты". Например, вы можете преобразовать значение `True` типа `bool` в строку `"True"`. Можно делать и обратные преобразования: из типа `string` в `int` и `double` оператором `Convert`, если это имеет смысл, конечно (когда преобразуются числовые данные, записанные в строковом виде). Это примеры преобразования с помощью вспомогательных классов.

Арифметические действия

Когда мы создавали первые программы, в них приходилось выполнять отдельные определенные арифметические действия. Сейчас же мы рассмотрим весь этот процесс с более общих позиций. Рассмотрим операции, которые могут быть произведены над переменными. Для выполнения операций требуются разного рода операторы: для выполнения арифметики — арифметические (сложение, вычитание, умножение, деление, другие). Все множество арифметических операторов можно разбить на несколько групп: простые арифметические операторы, операторы присваивания и специальные операторы, присущие только программированию.

Простые операторы

Они бывают *унарными* (воздействуют только на один операнд) и бинарными (воздействуют на два операнда). У каждого из этих операторов имеется свое символическое обозначение в C#. Одни обозначения совпадают с известными нам со времен школы, другие присущи только языку программирования. Перечень арифметических операторов приведен в табл. 3.4.

Таблица 3.4. *Арифметические операторы*

Обозначение	Пояснение
+ (унарный)	Плюс
– (унарный)	Минус
+ (бинарный)	Сложение
– (бинарный)	Вычитание
*	Умножение

Таблица 3.4 *(окончание)*

Обозначение	Пояснение
/	Деление
%	Деление по модулю

Пример использования унарного минуса:

```
int a=1;
int b=-a;
```

Результат: `b=-1`.

Так же применяется и унарный плюс, когда хотят точно определить, что переменная будет положительной. Не путайте знак присвоения в записи операторов и знак равенства в записи данного текста.

Бинарные операции сложения, вычитания и умножения особенностей не имеют. А вот об операции деления так не скажешь. Она — со спецификой. Ее результат — всегда целое число, если оба операнда целые, не зависимо, есть ли остаток от деления. Если он имеется, то он отбрасывается. Поэтому результат `25/4` будет равен 6. И только. Значит, чтобы сохранить остаток, надо делать так, чтобы хотя бы один из операндов был с плавающей точкой. Например,

```
int a=1;
double b=a/2.0;
```

Результат: `b=0.5`. Если бы написали

```
int a=1;
double b=a/2;
```

получили бы ноль в результате.

Операция деление по модулю — это операция получения остатка от деления. Остаток от деления на 2 — операция по модулю 2, остаток от деления на 10 — операция по модулю 10 и т. д. Эта операция интересна только для чисел с плавающей точкой, т. к. при делении таких чисел остаток получить напрямую невозможно. Если, например,

```
float a=1.2f;
float b=0.5f;
```

то `float c=a%b;` даст результат: `c=0.2`. А вот для целых чисел:

```
int a=12; int d=7; int c=a%d;
```

результат `c=5`.

Порядок выполнения арифметических операторов

Когда операторов в выражении больше двух, сразу возникает вопрос: какой должен после какого выполняться? В C# принято, что выражение вычисляется слева направо, но все-таки в зависимости от старшинства операций (их приоритета). Например,

```
int a=2 + 3 * 4;
```

Результат: `a=14`.

Самый высокий приоритет имеют унарные операции, за ними в сторону убывания следуют операции (`*`, `/`, `%`), а самый низкий приоритет у бинарных операций сложения и вычитания. Но наивысший приоритет имеют все-таки круглые скобки, так что лучше пользоваться скобками. Скобки перекрывают приоритеты операторов, явно указывая, как именно компилятор должен интерпретировать выражение.

Оператор присваивания

Этим оператором мы пользовались буквально с первой созданной нами программы. Мы уже поясняли его смысл: он присваивает значение выражения справа от знака равенства переменной слева от знака равенства. При этом по обе стороны знака равенства должны быть данные одного типа. Здесь следует добавить, что C# позволяет строить цепочку присвоений одного и того же значения разным переменным. Например, если имеем `int a; int b; int c;`, то можно записать `a=b=c=2;`. Присвоение выполняется справа налево. Для упрощения записи в C# введены составные операторы присваивания вида:

```
<Переменная левой части> <Операция> <Знак присвоения>
<Переменная правой части>
```

В разрезе операций составные операторы показаны в табл. 3.5.

Таблица 3.5. *Составные операторы присваивания*

Вид оператора	Эквивалентная запись	Пояснение
`a += b`	`a = a + b`	`a` и `b` складываются. Результат — в `a`
`a -= b`	`a = a – b`	Из `a` вычитается `b`. Результат — в `a`
`a *= b`	`a = a * b`	`a` умножается на `b`. Результат — в `a`
`a /= b`	`a = a / b`	`a` делится на `b`. Результат — в `a`
`a %= b`	`a = a % b`	`a` делится по модулю `b`. Результат — в `a`

Операторы инкремента и декремента

Среди всех сложений, выполняемых в программах, добавление 1 к переменной или вычитание единицы из переменной — наиболее распространенная операция. Обычно пишут: `int n; n=n+1;`. Или `n=n-1;`. Из того, что мы узнали только что о составном операторе присваивания, можно записать короче: `n+=1;`. Или `n-=1;`. В C# имеется еще более краткое обозначение этих действий: так называемые операторы *инкремента* и *декремента*: `++n; n++; --n; n--;`. Первые два выражения увеличивают `n` на единицу, а вторые — уменьшают `n` на единицу. Операторы со знаками операции перед именем переменной называются *префиксными*, а со знаками после имени переменной — *постфиксными*. Разница между ними есть, естественно, но не всегда. Например, если мы хотим просто добавить единицу к `n`, то результат от `++n; n++;` одинаков (как и при декрементной операции). Если же мы пишем `int a=++n;` и `a=n++;`, то тут и наступает различие, т. к. переменная `n` участвует в выражении присвоения. При `a=++n;` сначала к `n` добавляется единица, а потом уже содержимое `n` присваивается левой части, а при `a=n++;` наоборот: сначала значение `n` присваивается левой части, а потом уже оно увеличивается на единицу.

Операторы сравнения

Такими операторами являются операторы: равно (`==`), не равно (`!=`), больше (`>`), больше или равно (`>=`), меньше (`<`), меньше или равно (`<=`). Эти операторы определены для всех числовых типов и типа `char` и являются бинарными, т. е. работают с двумя операндами. Результат операции — значения `True` или `False`, имеющие тип `bool`. Например, имеем

```
int b; int c;
bool a = b < c;
```

Если содержимое переменной `b` меньше содержимого переменной `c`, то результатом сравнения будет значение `a=True`, иначе `a=False`. Пример с типом `char`:

```
char A='A'; char a='a';
bool b= A < a;
```

Если такое выражение откомпилировать и запустить на выполнение, получим `b=True`. Это говорит о том, что код прописной буквы по таблице Unicode меньше кода строчной.

Логические операторы

Для переменных типа `bool` определены логические операции, перечень которых приведен в табл. 3.6.

Таблица 3.6. *Логические операторы*

Оператор	Возвращает `True`, если выполняются указанные ниже условия. В остальных случаях — `False`
`!a`	`a=False`
`a & b`	`a=True` **и** `b=True`
`a | b`	`a` **или** `b` **равно** `True`
`a ^ b`	`a=True` **и** `b=False` **или** `a=Fale` **и** `b=True`
`a && b`	`a=True` **и** `b=True`
`a || b`	`a` **или** `b` **равно** `True`

Пример.

```
bool r, a=true, d=true, c=false, e=false;
r=!a;          // r=False
r=a&d;         // r=True
r=a&c;         // r=False
r=c&e;         // r=False
r=a|d;         // r=True
r=a|c;         // r=True
r=c|e;         // r=False
r=a^d;         // r=False
r=a^c;         // r=True
r=c^e;         // r=False
r=d^c;         // r=True
r=a&&d;        // r=True
r=a||d;        // r=True
r=a&&c;        // r=False
r=a||c;        // r=True
```

Рассмотрим представленный код. Здесь три вида бинарных операций: дизъюнкция (операция "ИЛИ" (`|`, `||`)), конъюнкция (операция "И" (`&`, `&&`)) и операция "исключающее ИЛИ" (`^`). При дизъюнкции, если хоть один из операндов истина, результат — истина, в остальных случаях — ложь. При конъюнкции, если хоть один из операндов ложь,

результат — ложь. При исключающем ИЛИ, если хоть один из операндов ложь, результат — истина.

Операция отрицания (`!`) — унарная. Она инвертирует значение операнда: `True` превращает в `False`, а `False` — в `True`.

Зачем введены еще `&&`, когда есть `&`, `||` и `|`? Я думаю, что разработчики C# пожадничали: все эти операторы были в C++, но там удвоенные знаки служили операциями над значениями переменных в целом, а одинарные — операциями над битами, из совокупности которых состоит некоторое значение. То есть операции с одним знаком были поразрядными, а с двумя — сравнивали значение в целом. Здесь же отличие в том, что при однознаковой операции вычисляются оба операнда, а потом происходит их сравнение и выдается результат. Если же в операции участвуют двузнаковые операторы, то вычисление результата прекращается, когда результат ясен уже из значения первого операнда. Действительно, возьмем конъюнкцию. При этой операции, если первый операнд `False`, то дальше вычислять нечего, потому что результат будет тоже `False`. А при дизъюнкции — наоборот: если первый операнд — `True`, то результат будет в любом случае такой же, независимо от значения второго операнда. Так для чего операторы с двойными знаками? Для увеличения скорости вычислений.

А как же все-таки обстоят дела в C# с побитовыми операциями? Они тут тоже имеются, как и в C++. Так, оператор `&` (логическое "И", конъюнкция) для целых чисел играет роль побитового логического умножения. То есть две переменные, объявленные одним из целых типов, фактически сравниваются между собой поразрядно, т. е. побитово по обычным правилам таблицы логического умножения: результат такого умножения будет равен единице тогда и только тогда, когда оба операнда равны единице. Например, можно выполнить следующие операторы:

```
int a=5;
int d=3;
int r=a&d;
// 5 & 3 = 1 — это результат
```

Почему? `5=101` в двоичной системе, т. е. имеет такой вид побитово. `3=011` побитово. Сравнение идет тоже побитово:

```
101
011
------
```

`001=1` в десятичной системе

Оператор | (логическое "ИЛИ", дизъюнкция) для целых чисел играет роль побитового логического сложения. Результат такого сложения будет равен единице тогда и только тогда, когда хотя бы один из операндов равен единице. Например, можно выполнить следующие операторы:

```
int a=5;
int d=3;
int r=a|d;
// 5 | 3 = 7 — это результат:
5=101
3=011
------
```

111=7 в десятичной системе

То есть видим, что операторы &, | работают как с логическими типами, так и с целыми типами. По отдельности или можно смешивать в операторах? Вот пример:

```
int i = 0;
bool d=true;
if (d & ++i == 1)
{
  i++;
}
```

Результат: i=2.

То есть булева и целая переменные участвуют в одном выражении и логически сравниваются.

Операторы сдвига

Здесь определены два оператора: сдвиг влево (<<) и сдвиг вправо (>>). При сдвиге влево-вправо значения битов числа перемещаются соответственно влево-вправо. При этом какие-то значения могут выпадать за размер числа как с начала, так и с конца. В этой связи при сдвиге существуют понятия арифметического и логического сдвигов. Сдвиг, при котором бит уходит, не влияя на оставшиеся биты, а на место появившегося бита записывается бит 0, — это логический сдвиг. При арифметическом сдвиге сдвигаемое значение рассматривается не просто как группа битов, а как целое число в *дополнительном коде* (далее мы рассмотрим, как получать этот вид кода). При сдвиге влево арифметический сдвиг ведет себя как логический сдвиг, при сдвиге вправо бит ухо-

дит, не влияя на оставшиеся биты, а на место появившегося бита устанавливается бит, соответствующий знаку.

Оператор сдвига влево (<<) имеет вид `a<<b` и сдвигает первый операнд влево в соответствии с количеством битов, заданным вторым операндом. Второй операнд должен быть типа `int` или типа, имеющего предопределенное неявное числовое преобразование в `int`. Если тип первого операнда — `int` или `uint` (32-разрядное число), начало сдвига задается пятью младшими разрядами второго операнда. Фактический сдвиг идет от 0 до 31 бита. Если тип первого операнда — `long` или `ulong` (64-разрядное число), начало сдвига задается шестью младшими разрядами второго операнда. Фактический сдвиг идет от 0 до 63 бит. Старшие разряды первого операнда, которые при сдвиге выходят за пределы операнда, теряются, а пустые младшие разряды заполняются нулями. Операторы сдвига никогда не вызывают переполнений.

Пример:

```
int r, a=5, d=3;
r=a << d;
```

Результат: `r=40`.

Обсудим этот пример. `a=5=101` в битовом представлении. Сдвигаем на 3 разряда влево. Получаем `a=101000`. Переводим назад в десятичную систему по правилу разложения числа по степеням основания системы счисления: $101000 = 1\times2^5 + 0\times2^4 + 1\times2^3$. Остальные коэффициенты разложения — нули. Члены разложения с такими коэффициентами в сумме дадут ноль. Поэтому общая сумма будет равна 32 + 8 = 40.

Оператор сдвига вправо (>>) имеет вид `a>>b` и сдвигает первый операнд вправо в соответствии с количеством бит, заданным вторым операндом. Если тип первого операнда — `int` или `uint` (32-разрядное число), начало сдвига задается пятью младшими разрядами второго операнда в виде: *второй_операнд* `&` `0x1f` . Так можно выделить младшие разряды из второго операнда: `&` — операция конъюнкции, `0x` означает, что запись шестнадцатеричная, `1f` — 11111. Если тип первого операнда — `long` или `ulong` (64-разрядное число), начало сдвига задается шестью младшими разрядами второго операнда (второй операнд `&` `0x3f` = `111111` (`3=11` и `f=1111`)).

Пример:

```
int r, a= 40, d=5;
r=a >> d;
```

Результат: `r=1`.

Обсудим этот пример. Сдвигаемое число — число со знаком. В данном случае знак — плюс. Из предыдущего примера мы видели, что в битовом представлении число 40 имеет вид 101000. Сдвигаем его на 5 разрядов вправо. Добавляем знак "плюс". Получаем число 1.

Операторы сдвига имеют короткую форму в виде `r<<d;` или `r>>d;`. Это означает, что `r=r<<d;` или `r=r>>d;`.

Для отрицательных чисел задача несколько усложняется (для ручной проверки, потому что среда программирования это делает сама).

Запишем:

```
int i = -40;
Console.WriteLine(i >> 5);
```

После компиляции и выполнения получим результат: –2.

Как он получился? Дело в том, что вычисления в компьютере для отрицательных чисел организованы в так называемом дополнительном коде. *Дополнительный код* — наиболее распространенный способ представления отрицательных целых чисел в компьютерах. Он позволяет заменить операцию вычитания операцией сложения и сделать операции сложения и вычитания одинаковыми для знаковых и беззнаковых чисел. Преобразование числа из прямого кода в *дополнительный* осуществляется по следующему алгоритму.

1. Если число, записанное в прямом коде, положительное, то к нему (к младшему разряду) дописывается старший (знаковый) разряд, который равен 0, и на этом преобразование заканчивается.
2. Если число, записанное в прямом коде, отрицательное, то все разряды числа, кроме знакового, инвертируются (нули заменяются единицами, а единицы — нулями), а к младшему разряду результата прибавляется 1. К получившемуся числу дописывается старший (знаковый) разряд, равный 1, потому что отрицательное число имеет в знаковом разряде 1.

Рассмотрим пример. Преобразуем отрицательное число –5, записанное в прямом коде, в число в дополнительном коде. Прямой код числа –5, взятого по модулю (т. е. без знака): 101. Инвертируем все разряды числа, получая таким образом *обратный код*: 010. Добавим к результату 1. Получим 011. Допишем слева знаковый единичный разряд. Получим 1011. Это и есть дополнительный код числа –5.

Для нашего примера –40 в двоичном виде без знака равно 101000. Инвертируем это число и получаем его обратный код: 010111. Добавляем к младшему разряду 1. Получаем 011000. Дописываем слева знаковый разряд. Получаем дополнительный код: 1011000. Сдвигаем все вправо на 5 разрядов. Результат: 10. Но это битовая величина. Десятичная будет 2. Добавляем знак "минус". Получаем –2.

ГЛАВА 4

Функции

В процессе программной реализации алгоритмов часто возникает необходимость выполнения повторяющихся действий на разных группах данных. Например, требуется вычислять синусы заданных величин или начислять заработную плату работникам. Ясно, что неразумно всякий раз в таких случаях создавать заново соответствующую программу именно под конкретные данные. Напрашивается вывод, что этот процесс надо как-то параметризовать, т. е. создать такую параметрическую программу, которая могла бы, например, вычислять синус от любого аргумента, получая извне его конкретное значение, или такую, которая бы начисляла зарплату любому работнику, получая данные конкретного работника. Такие программы, созданные с использованием формальных значений своих параметров, при передаче им конкретных значений параметров возвращают пользователю результаты расчетов. Их называют *функциями* по аналогии с математическими функциями.

Если в математике определена некая функция $y = f(x_1, x_2, ..., x_N)$, то на конкретном наборе данных $\{x_{11}, x_{21}, ..., x_{N1}\}$ эта функция возвратит вычисленное ею значение $y_1 = f(x_{11}, x_{21}, ..., x_{N1})$. В данном случае можем сказать, что аргументы $x_1, x_2, ..., x_N$ — это формальные параметры функции $f()$, а $x_{11}, x_{21}, ..., x_{N1}$ — их конкретные значения. Функция в C# объявляется почти аналогичным образом: задается тип возвращаемого ею значения (из рассмотренного материала мы знаем, что переменные могут иметь типы `int`, `float`, `long` и т. д.; тип значения, возвращаемого функцией, может быть таким, как и тип переменных); после задания типа возвращаемого значения задается имя функции (как и для математической функции). Затем в круглых скобках указываются ее аргументы — *формальные параметры*, каждый из которых должен быть описан так, как если бы он описывался в программе самостоятельно вне функции. Это только первая часть объявления функции.

Примечание

В языке C# типом функции может быть специальный тип `void`, когда функция ничего не возвращает. Такая функция вырождается в подпрограмму (или процедуру, как это еще называют) — конструкцию, которая просто производит некоторые вычисления.

Далее формируется тело функции — программный код, реализующий тот алгоритм, который и положено выполнять определяемой функции. Например, объявим условную функцию расчета зарплаты одного работника:

```
float salary(int TabNom, int Mes)
{
  /* Здесь должен быть программный код
     расчета зарплаты */
  return(значение вычисленной зарплаты);
}
```

Тип возвращаемого значения — `float`, т. к. сумма зарплаты — в общем случае, число не целое. Имя функции — `salary`. У функции два формальных параметра и оба целого типа: табельный номер работника и номер месяца, за который должен производиться расчет. Особым признаком функции является наличие оператора `return()`, который возвращает результат расчетов. После того как такая функция разработана, пользоваться ею можно так же, как математической функцией. Для данного примера мы смогли бы записать:

```
float y; y=salary(1001,12);
```

или

```
float y=salary(1001,12);
```

или

```
int tn=1001; int ms=12;
float y=salary(tn,ms);
```

Во всех случаях при обращении к функции мы в ее заголовочную часть подставляли вместо формальных параметров их конкретные значения. Каков внутренний механизм параметризации программы и превращения ее в функцию?

Мы описываем в заголовке формальные параметры и затем в теле функции используем их с объявленными именами при создании программного кода так, как будто известны их значения. Это возможно благодаря тому, что компилятор, когда начнет компилировать функцию,

соотнесет с каждым ее параметром определенный адрес некоторого места в так называемой стековой памяти, созданной специально для обеспечения вызова подпрограмм из других программ. А функция — это ведь тоже своего рода подпрограмма, да еще и возвращающая некоторое значение, а не только получающая какой-то результат расчетов. Размер такой адресованной области для каждого параметра определяется типом описанного формального параметра. Выделяется место и для будущего возвращаемого результата.

В теле функции будет построен программный код, работающий, когда речь идет о формальных параметрах, с адресами не обычной, а стековой памяти. Когда мы, обращаясь к функции, передаем ей фактические значения параметров, то эти значения пересылаются по тем адресам стека, которые были определены для формальных параметров (т. е. кладутся на "полочки" в стеке, отведенные для формальных параметров). Но программный код тела функции как раз и работает с этими "полочками", содержащими параметры. Поскольку тело строится так, что оно работает с "полочками", то остается только класть на них разные данные и получать соответствующие результаты. Вот это и осуществляется, когда мы каждый раз передаем функции конкретные значения ее параметров.

Отсюда можно сделать выводы: поскольку передаваемые функции значения пересылаются в стековую память (т. е. там формируется их копия), то сама функция, работая со стеком и ни с чем другим, не может изменять значения переменных, которые подставляются в ее заголовочную часть вместо формальных параметров. В примере мы писали `float y=salary(tn,ms)`, подставляя вместо формальных параметров `TabNom` и `Mes` значения переменных `tn` и `ms`. И мы утверждаем, что значения `tn` и `ms` не изменятся. Если же передавать функции не значения переменных, а их адреса, то переменные, адреса которых переданы в качестве фактических параметров, смогут изменяться в теле функции. Ведь по адресу можно записать все, что угодно, где бы он ни находился (в стеке или в обычной памяти). Мы уже фактически обнародовали тот факт, что функции в качестве параметров можно передавать и адреса тех данных, которые должны служить исходными для работы функции. Эти адреса называются *ссылками*. Параметры помечаются квалификатором `ref`. Например,

```
float func(x1, ref x2);
```

Если нужно просто вернуть значение из функции, а не изменить существующее значение, то следует использовать ключевое слово `out`:

```
static void func(double a, out string x1, out int x2);
```

Причем соответствующее слово должно ставиться не только в заголовке функции, но и при ее вызове.

C# позволяет передавать функции переменное количество параметров путем указания ключевого слова `params` при объявлении функции. Список аргументов также может содержать регулярные параметры, однако параметр, объявленный с помощью ключевого слова `params`, должен быть последним в списке. Он имеет форму массива переменной длины (массивы мы будем изучать позже). В функции может быть только один параметр с квалификатором `params`.

В последних версиях C# для функций введены так называемые именованные аргументы. Мы пока не уточняли до сих пор, как передавать аргументы в функцию, т. е. как подставлять их вместо формальных параметров. Теперь скажем, что в функции ее параметры — позиционные, т. е. за каждым параметром в заголовке функции закреплена определенная позиция: этот параметр — на первом месте в заголовке, этот — на четвертом и т. д. Когда вы вызываете функцию на выполнение, то должны ей передать *фактические значения параметров* (или, еще говорят — аргументы) в том порядке, в каком стоят в заголовке соответствующие этим аргументам параметры. Например, если параметры "рост" и "вес человека" в некоторой функции стоят (т. е. описаны) в последовательности "рост, вес", то при обращении к функции вы обязаны конкретное значение роста поставить на первое место в заголовке, а веса — на второе. Когда параметров много, бывает очень неудобно все время помнить, что за чем идет. Программист обычно все это пишет в комментарии к функции, если он аккуратный. Видимо, большинство программистов таким качеством не отличается, если фирма-разработчик ввела именованные аргументы. В чем новизна? Покажем это на примере функции из двух параметров: рост и вес (что у нее в теле будет — не суть важно). Далее приведено обращение к функции `Calculate()`.

У этой функции первым параметром является вес (`weight`) человека, вторым — рост (`height`). Эту функцию можно вызывать обычным путем:

```
Console.WriteLine(Calculate(70, 164));
```

Но вы, скажем, не помните, в какой последовательности в функции идут параметры, но имена их помните. Теперь вы можете задавать конкретные значения параметров в любом порядке, указав их имена через двоеточие:

```
Console.WriteLine(Calculate(height: 164, weight: 70));
```

Но если вы задаете не все именованные аргументы, а только часть, то тут есть ограничение: сначала идут обычные позиционные аргументы (т. е. значения без имен, а за ними — все именованные, т. е. значения с именами):

```
Console.WriteLine(Calculate(70, height: 164));
```

Вот сама функция:

```
static int Calculate(int weight, int height)
{
  return (weight * 703) / (height * height);
}
```

Новинкой последних версий C# является возможность задания значений параметров по умолчанию. Это значит, что если вы при обращении к некоторой функции не задали значение некоторого параметра (просто опустили этот параметр в операторе вызова функции, а вместо имени переменной поставили запятую — разделитель параметров должен сохраняться обязательно, или, если это был последний параметр, его просто опустили), то компилятор построит так программу, что опущенным параметрам присвоятся значения, заданные по умолчанию при определении функции. Вот пример задания значений параметров по умолчанию при формировании заголовка функции:

```
void func1(string val1 = "val", int val2 = 10,
           double val3 = 12.2)
```

Если теперь в теле основной программы вы укажете обращение к функции в виде `func1(,,25)`, то фактические значения параметров, с которыми станет работать такая функция, будут: `func1("val",10,25)`.

В C# функции, как и программы, сами по себе не существуют. Все они должны быть членами какого-то класса. Как мы видели ранее, программа `Main()` является членом класса `Program` (так формируется шаблон приложения). Детально сущности под названием "класс" мы будем изучать позже. Сейчас же остается довольствоваться этой информацией. Поэтому при создании функции как программы-примера мы поместим эту функцию в качестве члена класса `Program`, дадим ей атрибут `static`, который будет означать, как мы увидим при изучении классов, что такой член класса можно вызывать на выполнение прямо из самого класса, не создавая из класса некоторый объект, чтобы из того созданного объекта вызвать такую функцию (без `static` это и надо было бы делать, потому что статические и нестатические объекты в C# размещаются

в разных классах памяти: статические — в стековой памяти программы, а нестатические — в так называемой управляемой куче, где за такими объектами следит специальный механизм очистки памяти).

Создание некоторых функций

Пример функции, которая обеспечивает ввод строки ограниченного числа символов с выдачей длины введенной строки, приведен в листинге 4.1.

Листинг 4.1

```
/* Created by SharpDevelop.
 * User: user
 * Date: 15.11.2012
 * Time: 11:15
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app9_func
{
  class Program
  {
    static string str;
    // Вводит строку длиной не более lim символов
    // и выдает фактическую длину строки
    // или 0, если лимит превзойден

    static int getline(ref string s,int lim)
    {
      int i;  // для учета количества символов
      s=Console.ReadLine();
      i=s.Length;
      if(i > lim)
        i=0;
      return(i);
    }
```

```
    public static void Main(string[] args)
    {
      Console.WriteLine("Ввод строки с ограничением " +
                        "по ее длине\n");

      int lim=10;
      Console.WriteLine("Вводите строку >\n");
      int i = getline(ref str,lim);

      Console.WriteLine("Результат ввода: строка {0}, длина строки
{1}", str, i);
      Console.WriteLine("Для продолжения нажмите " +
                        "любую клавишу > ");
      Console.Read();  // Задержка экрана
    }
  }
}
```

Функция `getline()` имеет два параметра: один ссылочного типа `string s`, а другой — не ссылочного `int lim`. Говорят, что данные в первый параметр передаются по ссылке, а во второй — по значению. Мы об этом уже говорили ранее: данные для второго параметра при обращении к `getline()` кладутся на полочку в стековой памяти `getline()`, потому что так работает функция. Когда она вызывается на выполнение из `Main()`, в ее стеке запоминается адрес `Main()`, на который она должна возвратиться после своего завершения, а с полочек, которые ей создал компилятор для помещения параметров-значений, она должна забрать эти значения. Но нашей задачей было ввести строку символов в какую-то переменную и так, чтобы эта строка не локализовалась в самой функции (т. е. была бы неизвестной для других программ), а чтобы функция, как говорят, выдала наверх, вне себя, результат ввода. То есть ее первый параметр должен быть выходным, а не входным, как второй параметр `lim`. А как сделать так, чтобы содержимое переменной-параметра могло бы изменяться внутри тела функции? Надо, чтобы в качестве параметра передавался не адрес полочки стека, на которой лежит некоторое значение и которая принадлежит самой функции, а адрес переменной, объявленной вне функции без предварительной пересылки ее содержимого на полочку в стек программы, как это делается с параметрами-значениями. Когда мы ставим квалификатор `ref` при описании параметра функции, компилятор не станет связывать параметр с полочкой в стеке, а сделает так, что когда вы вместо такого параметра поставите имя фактической переменной с квалификатором `ref`, то вместо имени подставится адрес

этой переменной. И тогда все, что станет записываться в эту переменную в теле функции, попадет не на полочку в стек и не останется в самой функции, а окажется в переменной вне функции и станет доступно другим.

По правилам класса ссылка-переменная для статической функции должна быть тоже статической. Поэтому переменная `str`, из которой будет идти ввод строки внутри `getline()`, имеет атрибут `static`. Кстати, все эти моменты контролирует компилятор, и если вы что-то нарушите из правил, увидите ошибки компиляции. Алгоритм `getline()` очень простой: вводится строка стандартным оператором `ReadLine()` в строку-параметр ссылочного типа `str`. Вычисляется длина строки и помещается в `i`. Проверяется, чтобы длина не превосходила заданного лимита, который конкретно определен в переменной `lim`. Когда вы пишете `lim` вместо параметра в заголовок функции (пусть вас не смущает одинаковое название параметра и передаваемого значения, это разные переменные, т. к. у них разная память: у одной — в стеке `Main()`, у другой — в стеке `getline()`; если вас это смущает, дайте им разные имена), то значение этого `lim` кладется на полочку в стек `getline()` для ее второго параметра.

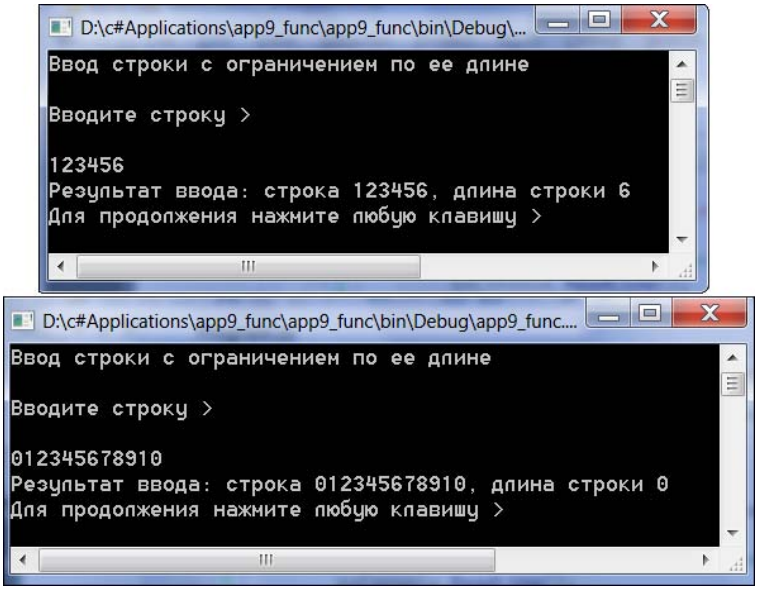


Рис. 4.1. Ввод строки с ограничением ее длины

Если вы ввели больше символов, чем определено в lim, счетчику i количества введенных символов присваивается ноль, иначе он остается равным длине введенной строки.

Результат работы функции getline() показан на рис. 4.1.

Оператор *if*

В этой программе нам встретился новый оператор if. Это оператор управления потоком выполнения команд программы. От точки входа в программу команды программы выполняются последовательно одна за другой, пока не встретят специальных команд, нарушающих естественный ход выполнения программы. Одним из таких нарушителей является оператор if. Он передает управление на дальнейшее выполнение программы в зависимости от выполнения условия, которое у этого оператора находится в его заголовке. То есть у if имеется заголовок, как у while с абсолютно одинаковым смыслом, и точно с таким же смыслом тело, заключенное в такие же фигурные скобки. И так же, как и у while, если условие в заголовке истинное, то выполняется тело, а если ложное, тело не выполняется, а выполняется следующий за телом оператор программы. Например,

```
if(условие-выражение)
{
  Операторы тела
}
Продолжение программы
```

Но у этого оператора есть довески. Один из них — необязательный else. Он имеет такую же форму, что и if, но пишется сразу за телом if. Смысл его в альтернативе if: если условие в if не выполняется, то управление передается на else и начинает выполняться его тело. Читается так: "Если условие if выполняется, то выполняется тело if, иначе — тело else".

А второй необязательный довесок — if else. При сложных условиях пары if...else недостаточно. Например, надо проверить содержимое переменной a, которая может принимать несколько значений. Тогда придется писать:

```
if (a == a1)
  тело 1
else if(a == a2)
  тело 2
```

```
else if(a == a3)
  тело 3
  ...
```

Если выполнится условие в каком-то `else if` или в самом `if`, выполнение цепочки завершится, и управление программой будет передано на оператор, непосредственно стоящий за этой цепочкой проверок.

Оператор *goto*

Этот оператор — тоже нарушитель естественного хода выполнения программы. Но еще более крутой, чем `if...else`. Он передает управление той команде программы, которая специальным образом помечена. Пометка (она называется *меткой*) — это набор символов, начинающийся с буквы или со знака подчеркивания. После метки ставится двоеточие (это и есть признак метки). Вот метки: `m:`, `N:`, `_12`. Метка может стоять одна в строке. Все равно управление передастся на эту строку, и далее станет выполняться команда (оператор), следующая за меткой. Вид оператора:

```
goto m;
goto _12;
```

Переход по этому оператору может идти как вперед, так и назад по программе. Но в пределах одной функции. Он применяется как самостоятельно (безусловный переход), так и в сочетании с `if...else` (`if(`*условие*`) goto m1; else goto m2;`). Это — условный переход. Оператор `goto` прерывает циклы `for`, `switch`, `while`. Причем, если `break` передает управление только вперед (в соответствии с канонами структурного программирования: движение по программе — только сверху вниз), то оператор `goto` может заставить программу ускакать и назад, чем способствует зацикливанию из-за такой петли. Поэтому данный оператор очень не любит компьютерное начальство, понимающее в программировании: с его "помощью" возможны значительные задержки в отладке программы. Но программисты им пользуются, при необходимости. Если они не лихие, то ничего страшного.

Функция выделения подстроки из строки

Создадим функцию `substr()`, которая из строки выделяет подстроку, начиная с заданного номера символа, числом, которое тоже задается в качестве параметра функции.

Пример программы с функцией `substr()` представлен в листинге 4.2.

Листинг 4.2

```
/* Created by SharpDevelop.
 * User: user
 * Date: 15.11.2012
 * Time: 14:10
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app10_substr
{
  class Program
  {
    static string substr(string s, int i, int count)
    {
      int j=0;

      string r = s.Remove(j, i-1);

      int par1=count;
      int par2=(s.Length-i)-(count-1);
      string r1=r.Remove(par1,par2);
      return(r1);
    }

    public static void Main(string[] args)
    {
      string s="";
      while(true)
      {
        Console.WriteLine("Введите строку >");
        s=Console.ReadLine();
        if(s == null)
          break;

        Console.WriteLine("Введите позицию начала " +
                          "выделения >");
```

```
        string r=Console.ReadLine();
        int i=System.Convert.ToInt32(r);

        Console.WriteLine("Введите количество " +
                          "символов выделения >");
        r=Console.ReadLine();
        int j=System.Convert.ToInt32(r);

        Console.WriteLine("Выделить с символа {0} символов {1}
\nПодстрока: {2}",i,j,substr(s,i,j));
        Console.WriteLine("Для продолжения нажмите " +
                          "любую клавишу > ");
        //  Console.Read();
      } // while
    }
  }
}
```

Эта функция с именем `substr()` возвращает строку (тип возвращаемого значения — `string`), а на свой вход получает строку символов `s` и два числовых параметра: номер позиции в строке, с которой надо выделить подстроку, и количество выделяемых символов (`count`). Для выделения подстроки используется имеющаяся в системе функция `Remove()`, которая делает то, что мы хотим заставить делать `substr()`, но вместо выделения удаляет символы. Надо учесть, что счет символов в строке начинается с нуля: первый символ находится в позиции 0, второй — в позиции 1 и т. д. Сначала удаляются все символы до заданной в параметре позиции, а потом все символы после заданного количества. В остатке и получим требуемую подстроку. Далее созданная функция помещена в `Main()` для демонстрации ее работы. Чтобы можно было проверять на разных строках, создан цикл из оператора `while`, в теле которого и производятся все операции. В заголовке `while` стоит значение `true`. Это означает, что `while` создает бесконечный цикл, выход из которого надо делать в его теле. Выход происходит уже частично знакомым способом: когда при очередном вводе строки нажимается комбинация клавиш <Ctrl>+<Z>, результат ввода будет `null`. На это значение и проверяется переменная, в которую должен попадать ввод. Проверка — с помощью оператора `if()`. Если условие в `if` выполняется, то идет передача управления на выход из цикла `while`. Так работает оператор `break`, который находится в теле `if`. Это новый для нас оператор. К этому оператору имеется "пара" — оператор `continue`. Он, в отличие от `break`, передает

управление в начало цикла. И еще одно нововведение — оператор `Convert()`. Так как с экрана запрашиваются данные для выделения подстроки (номер позиции в строке и количество символов), эти данные вводятся в виде строчных данных. А работать с ними надо как с числами (они объявлены как `int`). Поэтому требуется их преобразование из типа `string` в тип `int`. Это выполняет системная функция `System.Convert.ToInt32()`.

Результат работы функции приведен на рис. 4.2.

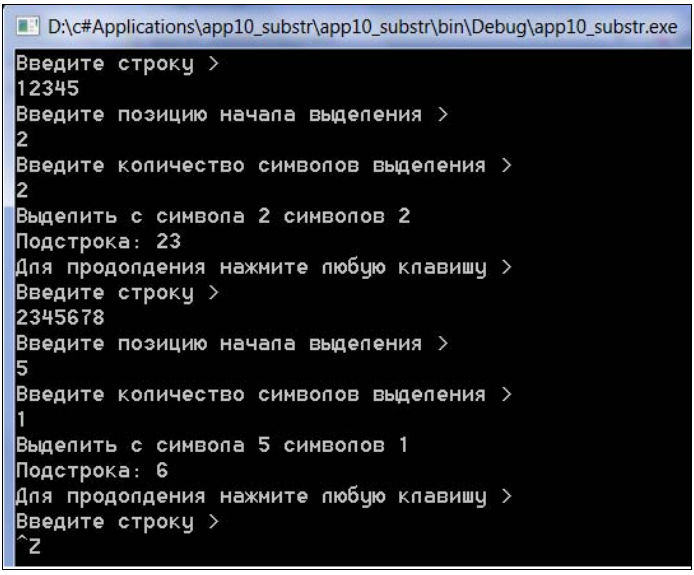


Рис. 4.2. Работа функции `substr()`

Функция копирования строки в строку

Функция `MyCopy()` показана в листинге 4.3.

Листинг 4.3

```
/* Created by SharpDevelop.
 * User: user
 * Date: 15.11.2012
 * Time: 17:22
 *
```

```
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app11_copy
{
  class Program
  {
// ------------------------ substr() ---------------

    static string substr(string s, int i, int count)
    {
      int j=0;

      string r = s.Remove(j, i-1);

      int par1=count;
      int par2=(s.Length-i)-(count-1);
      string r1 = r.Remove(par1,par2);
      return(r1);
    }
// -------------------------------------------------
    static string inp,ou;
    static string MyCopy(string inp, ref string ou,
                         int i, int count)
    {
      /* Вставляет в строку ou строку inp с позиции i
         count символов:
         из строки inp берутся первые count символов */
      // Выделение подстроки из входной строки:
      int j=1;
      inp=substr(inp,j,count);
      string r=ou.Insert(i-1,inp);

      return(r);
    }

    public static void Main(string[] args)
    {
      Console.WriteLine("Введите строку, в которую " +
                        "надо вставлять другую строку>");
```

```
      ou=Console.ReadLine();
      // ou="0123456789";
      string s="";
      while(true)
      {
        Console.WriteLine("Введите вставляемую строку >");
        s=Console.ReadLine();
        if(s == null)
          break;

        Console.WriteLine("Введите позицию начала " +
                          "вставки >");
        string r=Console.ReadLine();
        int i=System.Convert.ToInt32(r);

        Console.WriteLine("Введите количество " +
                          "символов вставки >");
        r=Console.ReadLine();
        int j=System.Convert.ToInt32(r);

        Console.WriteLine("вставить с символа {0} символов {1}
\nПодстрока: {2}",i,j,MyCopy(s,ref ou,i,j));
        // Console.WriteLine("Для продолжения нажмите любую
клавишу > ");
        //  Console.Read();
      } // while
    } // Main()
  }  // class
} // using
```

Эта функция похожа на предыдущую (`substr()`), только она не выделяет подстроку, а вставляет подстроку в другую строку: начиная с места `i` в строку вставляются первые `count` символов другой строки. В работе `MyCopy()` применяется `substr()`, поэтому она наряду с `MyCopy()` включена как член класса в класс `Program`. В функции применяется новый для нас метод `Insert()`, который вставляет в строку заданное количество символов и результат выдает как тип `string`. Отметим, что при этом входная строка не портится, а результат самой функции вставки надо присваивать переменной типа `string`.

Результат работы основной программы приведен на рис. 4.3.

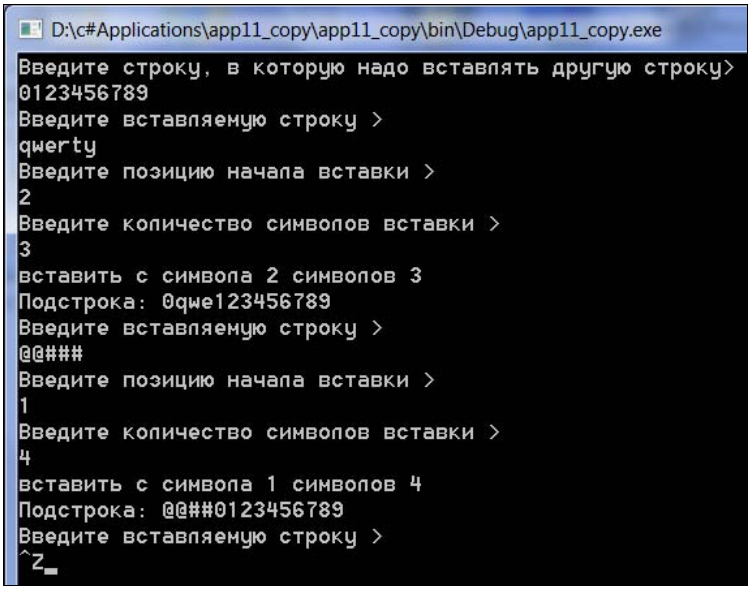


Рис. 4.3. Копирование части одной строки в другую с заданной позиции

Функция с выходными параметрами

Функция демонстрирует, как применяются ее выходные параметры. У выходных параметров имеется квалификатор `out`. Он дает возможность не заботиться о том, чтобы передавать параметры по ссылке, за счет которой изменялось бы его значение в теле функции. Приложение с такой функцией представлено в листинге 4.4, а результат вычислений по функции показан на рис. 4.4.

Листинг 4.4

```
/* Created by SharpDevelop.
 * User: user
 * Date: 16.11.2012
 * Time: 12:58
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
```

```
namespace app12_out
{
  class Program
  {
    static void func1(double b, out double f, out int i)
    {
      f=b;
      i=(int)b; // Преобразование типов
    }

    public static void Main()
    {
      Console.WriteLine("Введите дробное число >");
      string s=Console.ReadLine();
      double f,b;
      int i;
      b=System.Convert.ToDouble(s); // Преобразование
                                    // типов
      func1(b,out f, out i);

      Console.WriteLine("Результаты от функции: f={0}, i={1}", f,
i);
      Console.Write("Для продолжения нажмите " +
                    "любую клавишу > ");
      Console.Read();  // Задержка экрана
    }
  }
}
```

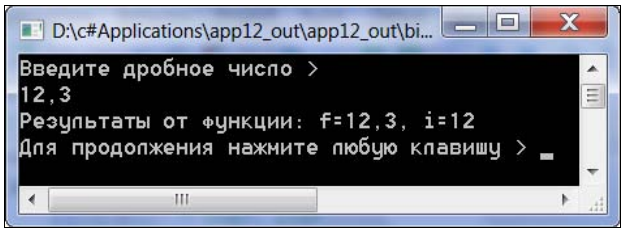


Рис. 4.4. Вывод на экран значений выходных параметров функции

Функция `func1()` ничего не выдает в область своего результата, т. к. у нее тип `void`. Однако она выдает результаты через свои выходные параметры, которые помечены квалификаторами `out`. В приложении

демонстрируются преобразования данных из типа `double` в `int` (дробная часть при этом, естественно, отсекается; этот прием можно применять как получение целой части дробного числа) и преобразования строки типа `string` в данное типа `double`. Преобразование проходит, если только в строке находится информация, которая может быть преобразована в число. Например, если вы у себя в компьютере задали плавающий стандарт (отделение в числе целой части от дробной) с разделителем в виде запятой, а введете в строку число с точкой, преобразование не пройдет.

Переключатель *switch*

При большом многовариантном выборе использование комбинации `if...else`, рассмотренной ранее, дает довольно запутанную картину. В таких ситуациях удобнее использовать специальный оператор `switch` — оператор выбора одного из многих вариантов. Его называют также переключателем.

Поясним работу оператора на примере программы обнаружения вводимого символа: `a` или `b`. Текст программы представлен в листинге 4.5.

Листинг 4.5

```
/* Created by SharpDevelop.
 * User: user
 * Date: 20.11.2012
 * Time: 7:59
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app14_switch
{
  class Program
   {
    /* Определяет, введен ли символ a или b */
    static void CountSimb(string c, out int v_a,
                          out int v_b)
    {
      int a=0,b=0;
```

```
      switch(c)
      {
        case "a":
          a = 1;  /* Вариант присвоения значения v_a
                     или v_b не проходит: компилятор
                     не узнает выходных переменных */
          break;
        case "b":
          b = 1;
          break;
        default:  /* Введен какой-то другой символ
                     (не a и не b) */
          break;
      }  // switch
      v_a=a;
      v_b=b;
    }
    public static void Main()
    {
      string s;
      int v_a,v_b;
      while(true)
      {
        Console.WriteLine("Введите символ, <Enter> >");
        s=Console.ReadLine();
        if(s==null)  // нажали <Ctrl>+<Z>
          break;
        CountSimb(s,out v_a,out v_b); // Подсчет
                                      // количества
                                      // символов 'a', 'b'

        Console.WriteLine("Введен a = {0}, " +
                          "введен b = {1} ", v_a,v_b);
        Console.WriteLine("Для продолжения нажмите " +
                          "любой символ > ");
        Console.ReadLine();
      }
    }
  }
}
```

Оператор switch помещен в функцию CountSimb(), у которой два выходных параметра: v_a и v_b. С экрана запрашивается ввод символа, хотя на самом деле вводится строка из одного символа (мы ранее видели, что символ и строка — это разные типы данных). Оператор работает следующим образом: в его заголовке помещена переменная, значение которой сравнивается с конкретными вариантами значений, задаваемых в теле оператора. Тело оператора ограничено фигурными скобками. Внутри тела находятся блоки сравнения, состоящие из ключевого слова case, двоеточия, после которого идет уже тело самого case. Между case, после которого ставится обязательный пробел, и двоеточием задается конкретное значение, с которым станет сравниваться значение переменной, заданной в заголовке case. В нашем случае это строка c. Сравниваемое значение записывается в соответствии со своим типом: у нас это строка, значит, сравниваемое конкретное значение должно быть в кавычках (у нас это "a" и "b"). Если бы сравниваемое значение было символом, надо было бы писать, например, 'a' или 'b'. Если бы сравниваемое значение было числом, то надо было бы писать, например, 1 или 2, или 123. И так далее.

После задания case (тоже как бы его заголовка) ниже пишутся операторы, определяющие, что надо делать в том случае, когда переменная заголовка switch сравнилась с соответствующим значением, заданным в заголовке case. То есть, как говорят, определяют функциональность этого варианта case. Завершается тело case знакомым нам оператором break, прерывающим выполнение switch и передающим управление на первый оператор программы, который находится сразу за закрывающей скобкой switch.

Таким способом задаются необходимые варианты case, в каждом из которых отражается конкретное значение переменной, указанной в заголовке switch. Если бы нам надо было проверять не два, а, скажем, 4 символа (т. е. мы бы хотели узнать, какой из заданных четырех символов введен с клавиатуры), то мы должны были бы, например, записать case "a":, case "b":, case "c":, case "d": и обрабатывать в телах этих case соответствующие ситуации. Если мы после каждого case в конце его тела не поставим break, то программа "провалится" на следующий case, который уже выполнять не требуется. А не требуется ли? А если, например, нужно что-то выполнить при условии, что введенное значение либо "a", либо "b", либо "c"? Вот тогда отсутствием break можно и воспользоваться: тела case, которые надо пропустить, следует оставить пустыми (оставить только одни заголовки case), а в теле последнего case из этой группы уже задать функциональность, общую для всех случаев "либо" и в конце — break.

Последним оператором в теле `switch` стоит `default`, имеющий структуру `case`. Он замыкает ситуацию с проверкой конкретных значений `switch`-переменной. В этот блок попадают все прочие случаи, не попавшие в конкретные `case`. Здесь можно при необходимости задавать свою функциональность для случая "прочие варианты".

Следует отметить один малоприятный момент: выходные параметры (с квалификатором `out`) не могут получать свои значения в теле `switch`: компилятор выдает ошибку, сообщающую, что эти переменные не определены.

Процесс ввода зациклен с помощью оператора `while`. Цикл — бесконечный, т. к. в заголовке `while` стоит значение `true` (условие всегда выполняется). Для выхода из цикла применен `break`, который выполняется после нажатия комбинации клавиш <Ctrl>+<Z>. Результат работы приложения показан на рис. 4.5.

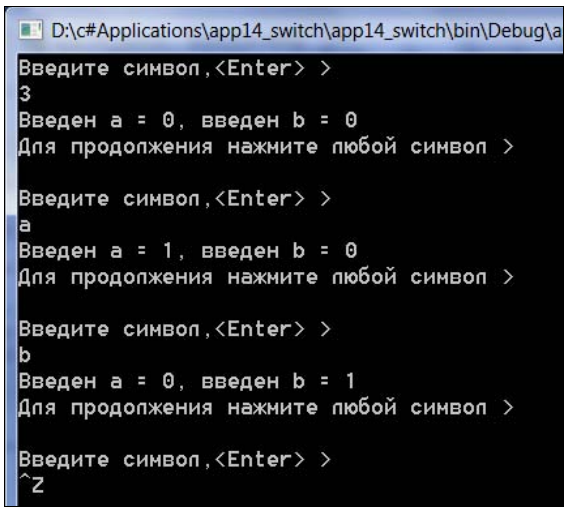


Рис. 4.5. Демонстрация работы оператора `switch`

Область действия переменных

Все переменные, объявленные в теле функции, называются *локальными* и "живут", пока "живет" сама функция. Как только функция перестает выполняться, значения ее локальных переменных теряются. То есть

каждый раз при вхождении в функцию такие переменные получают значения, а при выходе из функции эти значения теряются. Как же заставить некоторые переменные сохранять свои значения после выхода из функции? Лучше всего это делать, объявив переменную выходным параметром.

Рекурсивные функции

Рекурсивные функции — такие функции, которые могут вызывать сами себя. При этом каждый раз под каждый вызов создается совершенно новый набор локальных переменных, отличный от набора вызывающей (т. е. этой же) функции. Рекурсия применяется при обработке так называемых "рекуррентных" (основанных на рекурсии) формул. Одной из таких формул является, например, формула вычисления факториала числа: $n! = (n-1)! \times n$, где $0! = 1$. Чтобы вычислить факториал на шаге n, надо воспользоваться факториалом, вычисленным на шаге $n-1$. Рекурсивная функция, реализующая алгоритм для вычисления факториала, представлена в листинге 4.6.

Листинг 4.6

```
int fact(int i)
{ if(i==0)
    return(i+1);
  else
  { i=i * fact(i-1);
    return(i);
  }
}
```

ГЛАВА 5

Массивы

Массив — удобное средство работы с однотипными данными. Все они могут быть собраны в конструкцию, называемую массивом, и расположены в удобном для пользователя порядке. То есть помещать и доставать данные, пользуясь такой конструкцией удобно. Представьте себе кошелек, в котором есть отсек, где хранятся разбитые по ячейкам некие данные. Вы открыли кошелек, посмотрели в нужный отсек и поместили туда или достали оттуда сразу то, что вам необходимо. В массиве могут быть, повторим, только данные одного типа: целые числа со знаком, числа с плавающей точкой (это уже другой массив, т. к. тип данных другой), строки текста, какие-то иные объекты (например, дома одной серии; так мы уже подходим к объектно-ориентированному программированию).

Одномерные массивы

Каждый элемент массива имеет свой *индекс* (целое число от нуля до максимального количества элементов массива минус единица). По этому индексу можно обращаться к массиву и работать с соответствующим этому индексу элементом. То есть фактически элементы массива пронумерованы. Счет начинается от нуля (первый элемент). Последний элемент имеет номер, равный максимальному количеству элементов массива минус единица. Вот эти номера и есть индексы. Количество элементов массива называют его *размером*. Например, массив из 100 чисел с плавающей точкой. Его размер равен 100. А элементы пронумерованы числами (т. е. индексами) от 0 до 99.

Если заранее известно количество элементов массива, то его размер можно задать константой, равной количеству элементов. Зная тип эле-

ментов, мы имеем представление о том, сколько памяти отводится под этот тип данных. Следовательно, можно заранее на этапе компиляции вычислить и размер памяти, которую надо выделить для этого массива. Такие массивы называются *статическими*. Но если размер массива зависит от значения некоторой переменной, то память для такого массива может быть выделена только в процессе выполнения программы, т. е. динамически, когда станет известно значение переменной, определяющей размерность массива. Поэтому подобные массивы называют *динамическими*.

Массивы в памяти располагаются поэлементно: первый элемент, за ним — второй, далее — третий и т. д. Так как все элементы однотипны, т. е. занимают одинаковую величину памяти, то очень легко вычислять местоположение любого элемента, зная местоположение начального.

Массивы как, в общем случае, динамические структуры располагаются в управляемой куче памяти и являются ссылочными типами, такими как строки, которые мы рассматривали ранее. Там же мы отмечали, почему удобно иметь ссылочный тип: нам не надо перегонять огромные массивы данных (а у массивов бывают очень большие объемы) из одного места памяти в другое. Достаточно только передать ссылку на массив (т. е. его адрес в куче). И тот, кому передана эта ссылка, получает массив в свое полное распоряжение. Как будто мы его этому кому-то переслали в карман весь массив. А мы его и не думали двигать. Просто передали его адрес. И все. Поэтому при работе со ссылочными данными получается большая экономия времени работы компьютера.

Массивы бывают *одномерными* и *многомерными*. Например, массив строк, содержащий строки некоего текста, — одномерный. А если каждую строку рассматривать как совокупность символов, то этот же текст можно представить в виде двумерного массива: массив стольких-то строк, в котором в каждой строке по столько-то символов (т. е. матрица получается).

Как объявляется в программе массив? Почти по общим правилам:

```
тип знак имя_массива
```

где *тип* — тип элементов массива, `знак` — знак того, что объявляемая структура — это массив, `имя массива` — имя массива. Например,

```
int []m;
```

Здесь квадратные скобки говорят, что объявляется массив, `int` — что это будет массив целых чисел со знаком, под каждое число отводится 4 байта памяти, а `m` — это имя массива, по которому его можно будет идентифицировать.

Когда мы объявляем в программе массив, то мы создаем, так сказать, некий каркас, некоторое пространство, которое затем надо будет наполнить содержанием: задать конкретные значения элементов. То есть, как говорят, надо *массив инициализировать*. Инициализацию можно проводить двумя видами: явным и неявным. В первом случае элементы массива задаются каждый конкретно. Все элементы располагаются в фигурных скобках, как показано на примере:

```
int []m = {1,2,3,4,5};
```

Здесь массив `m` после объявления инициализирован: он будет состоять из пяти элементов. Первый элемент получил (получит после компиляции программы) значение 1, второй — 2, и т. д. При таком задании массива он поместится в динамическую память (кучу), и ссылка на него (на его начало, т. е. на его первый элемент) разместится на полочке для переменной `m`.

Во втором случае создание массива происходит с помощью специального оператора `new` (важно запомнить, что объявление массива совсем не означает его создания, потому что формируется только некий шаблон будущего массива):

```
int []m = new int[5]
```

По оператору `new` в управляемой куче памяти, в которой хранятся данные ссылочного типа, отведется память для размещения пяти элементов типа `int`, а адрес начала этой памяти будет положен на полочку для переменной `m`. Так как у массива элементы — целые числа, то все они получат значения по умолчанию, равные нулю.

Если вам требуется по-своему задать элементы массива, вы должны их задать путем их ввода с клавиатуры или путем пересылки из какого-то источника. Короче — путем вычисления.

Пример программы работы с массивом представлен в листинге 5.1, а результат — на рис. 5.1.

Листинг 5.1

```
/* Created by SharpDevelop.
 * User: user
 * Date: 21.11.2012
 * Time: 12:32
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
```

```
using System;

namespace app15_array
{
  class Program
  {
    public static void Main(string[] args)
    {
      string []A = new string [10];
      string s;
      int i=0;
      Console.WriteLine("Введите строки массива, после " +
            "каждой <Enter>, в конце <Enter> <Ctrl+z>");
      while(true)
      {
        s=Console.ReadLine();
        A[i]=s;
        if(s==null || i > 10)
          break;
        i++;
      }
      Console.WriteLine("Введено строк – {0}",i);
      Console.Read();
    }
  }
}
```

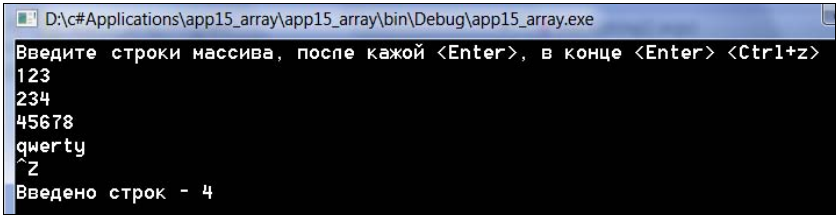


Рис. 5.1. Объявление и ввод данных в массив

В программе объявлен массив из десяти строковых элементов. Так как нет конкретной инициализации элементов, то массив размещен в динамической памяти оператором `new`. В цикле, уже знакомом нам, вводятся строки текста до тех пор, пока не будет нажата комбинация клавиш <Ctrl>+<Z> или (см. оператор `||` в заголовке `if`) количество введенных

строк (накапливается в переменной i) не превзойдет заданного лимита, указанного в операторе new.

Из программы видно, как доставать или обращаться к элементам массива: в квадратных скобках надо указывать индекс необходимого элемента (индекс изменяется от нуля). То есть запись A[i]=s; говорит о том, что i-му элементу массива A присваивается значение s.

В одной строке можно объявлять несколько массивов, например, int[] u,v;. А вот пример объявления двумерного массива:

```
int [,] w = new int[3,5];
```

Зная, что такое массив, мы можем теперь рассмотреть еще один оператор.

Оператор *foreach*

Можно уже по одному названию (в переводе это "для каждого") догадаться, что делает этот оператор. Это — оператор организации цикла, он похож на оператор for. У него есть тело и заголовок. Правила формирования тела — как у for. Заголовок несколько другой. Например, запись

```
foreach(int item in m)
{
   тело
}
```

читается так: "для каждого элемента item из массива m выполнить тело". При этом элементы массива m последовательно выбираются в переменную item (здесь предполагается, что массив содержит элементы типа int).

Пусть у нас есть массив из 5 элементов типа int. Составим программу инициализации и суммирования всех элементов этого массива. Текст программы приведен в листинге 5.2. Результат работы показан на рис. 5.2.

Листинг 5.2

```
/* Created by SharpDevelop.
 * User: user
 * Date: 21.11.2012
 * Time: 13:40
 *
```

```
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app16_foreach
{
  class Program
  {
    public static void Main()
    {
      int []A = new int [5];
      string s;
      int i=0;
      Console.WriteLine("Введите числа массива, после " +
          "каждого <Enter>, в конце <Enter> <Ctrl+z> ");
      while(true)
      {
        s=Console.ReadLine();
        if(s==null || i > (A.Length-1))
          break;
        A[i]=Convert.ToInt32(s);
        i++;
      }
      Console.WriteLine("Введено чисел — {0}",i);
      i=0;
      foreach(int j in A)
        i += j;
      Console.WriteLine("Сумма элементов массива = {0}",
                        i);
      Console.Read();
    }
  }
}
```

Первый раз было введено пять элементов, а второй — только три. Во втором случае сумма подсчитана верно. Это говорит о том, что в массиве недостающие элементы — нулевые. Комментирования заслуживает только оператор `if`, в заголовке которого проверяется, надо ли выходить из бесконечного цикла. Первая часть неравенства срабатывает, когда нажата комбинация клавиш <Ctrl>+<Z>, вторая, если превзойден размер

массива. То есть лишние члены в массив не попадут и прерывания программы не будет. В этом же месте мы видим, что для массива, как и для строки, существует функция, которая возвращает длину (`Length`). У массива длина — это количество его элементов. Отметим, что `foreach` работает только на чтение элементов из массива. Отметим также, что `foreach` так просто работает только для массивов из базовых типов данных (чисел, строк). Однако для классов, которые мы изучим позже, он функционирует довольно сложно, с чем вы познакомитесь, когда дойдете до изучения так называемых коллекций данных.

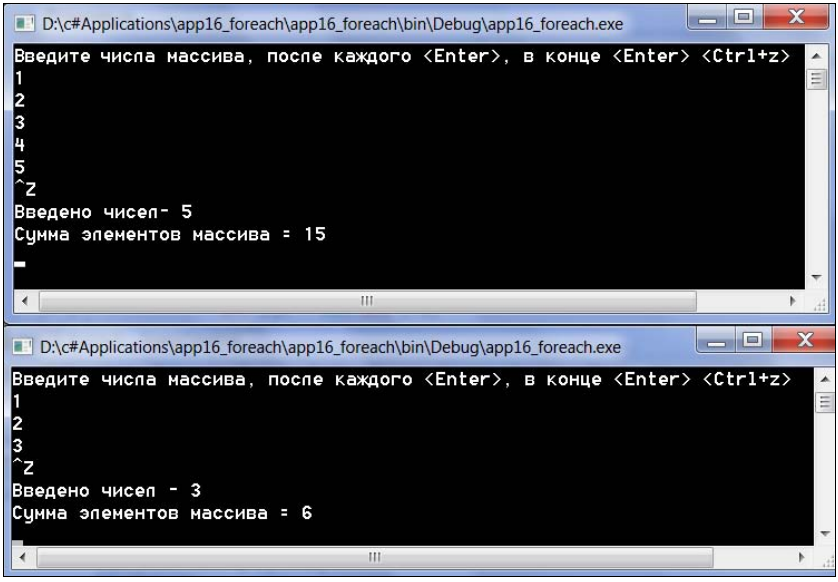


Рис. 5.2. Результат ввода данных в массив и подсчета суммы его элементов

Многомерные массивы

Поговорим теперь о многомерных массивах. Они задаются формой:

```
<тип>[, ...,] <имя переменной (или переменных через запятую)>;
```

В квадратных скобках стоят запятые, которых на единицу меньше, чем размерность массива. Так для двумерного массива квадратные скобки будут иметь вид `[,]`, для трехмерного — `[,,]` и т. д. Инициализация

с использованием многомерных константных массивов будет иметь вид (на примере двумерного массива):

```
int[,] m = {{1,2},{3,4}};
```

На этом сведения о массивах не кончаются. Скоро вы познакомитесь с классом `Array` — базовым классом для всех массивов, который содержит средства для работы с массивами, в том числе такое средство, как сортировка массивов, а также с классом `List<>`, поддерживающим список объектов, тип которых задается в угловых скобках. Каждый объект списка доступен по его индексу в списке. Класс также поддерживает методы для поиска по списку, сортировки списка и др.

ГЛАВА 6

Еще раз о функциях консольного ввода-вывода

Весь материал, изученный нами в предыдущих главах, изобилует применением функций ввода-вывода. Без них никак не обойтись. Теперь, когда мы к ним привыкли и получили знания о некоторых других операторах, можно обобщить сведения о функциях ввода-вывода. Напомним, что консоль для ввода — это клавиатура, а для вывода — экран.

Ввод

Чтобы получить данные, вводимые пользователем вручную (т. е. с консоли), применяются команды

```
<строковая переменная> = Console.ReadLine();
```

Для того чтобы преобразовать данные к нужному типу, используются функции из структуры `Convert`.

- Преобразование к переменной целочисленного типа имеет вид:

```
<переменная целого типа> = Convert.ToInt32(<строковая
переменная>);
<переменная целого типа> = Convert.ToInt64(<строковая
переменная>);
<переменная целого типа> = Convert.ToByte(<строковая
переменная>);
```

- Преобразование к переменной типа с плавающей точкой имеет вид:

```
<переменная типа с плавающей точкой> =
Convert.ToDouble(<строковая переменная>);
```

- Преобразование к переменной логического типа имеет вид:

```
<переменная логического типа> = Convert.ToBoolean(<строковая
переменная>);
```

- Преобразование к переменной символьного типа имеет вид:

```
<переменная символьного типа> = Convert.ToChar(<строковая
переменная>);
```

Пример:

```
int x;
double y;
string s;
s = Console.ReadLine();
x = Convert.ToInt32(s);
```

Ввод можно выполнить и одним оператором:

```
y = Convert.ToDouble(Console.ReadLine());
```

Чтобы обеспечить задержку экрана, организуют ожидание ввода любой клавиши в конце программы с помощью оператора `Console.Read();`.

Вывод

Вывод на экран организуется функциями `Console.Write()` и `Console.WriteLine()`. Первая выводит строку из переменной, имя которой указано у нее в качестве аргумента, и курсор вывода остается в позиции, следующей за последним выведенным символом строки. Так что если еще раз применить эту функцию для вывода, то следующая выведенная строка склеится на экране с предыдущей.

Вторая функция после себя не оставляет таких неприятностей: в конце вывода она устанавливает курсор экрана в первую позицию следующей строки. Но вывод на экран начинает с той позиции курсора экрана, на которой он остался после предыдущего вывода. Так что если вывод с помощью `Console.WriteLine()` поставить после вывода с помощью `Console.Write()`, то строки вывода тоже склеятся.

Пример использования обеих функций представлен в программе листинга 6.1, а результат работы программы — на рис. 6.1.

Листинг 6.1

```
/* Created by SharpDevelop.
 * User: user
 * Date: 18.11.2012
 * Time: 13:37
 *
```

```
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app13_for_all
{
  class Program
  {
    public static void Main()
    {
      string s="Проверка вывода";
      int y=2;
      int i=2;

      Console.WriteLine(s);     // переменная
      Console.WriteLine(55.3);  // константа
      Console.WriteLine(y*3+7); // выражение

      Console.Write(s);           // переменная
      Console.Write(-5.3);        // константа
      Console.WriteLine(i*3+7/i); // выражение

      Console.Write("Для продолжения нажмите " +
                    "любую клавишу >");
      Console.Read(); // Задержка экрана
    }
  }
}
```

Заметим, что числовые данные и выражения вставляются в функции в качестве аргументов в естественном виде (без кавычек). Если требуется

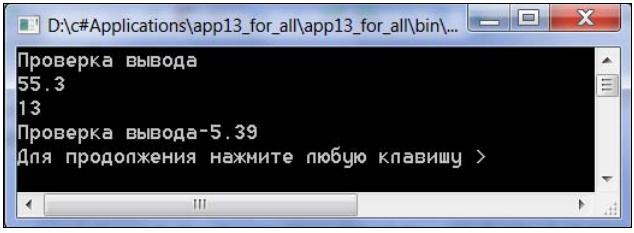


Рис. 6.1. Результат работы программы, приведенной в листинге 6.1

выводить строку-литерал, то такой литерал (по определению — это строка символов в кавычках) тоже записывается в качестве аргумента функции в своем естественном виде: строкой символов в кавычках. Если литерал при выводе требуется разбить на несколько отдельных строк, то в местах разбиения надо записать символ `'\n'`, который обеспечит вывод частей литерала в виде отдельных строк с первой позиции. Например, оператор

```
Console.Write("aaaaaa\nbbbbbbb\nccc\n");
```

выведет такую информацию:

```
aaaaa
bbbbbbb
ccc
```

Числовые данные редко выводятся в естественном виде. Чаще всего они требуют приведения их к определенной форме вывода. То есть требуют форматирования. Для форматирования вывода функции консольного вывода вызывают специальную функцию `String.Format()`. Сам же формат задается в управляющей строке функции вывода. Поэтому функция вывода (и `Console.WriteLine()`, и `Console.Write()`) будет в этом случае иметь вид, показанный на примере `Console.WriteLine()`:

```
Console.WriteLine("Управляющая строка", аргумент1, аргумент2, ...,
аргумент n);
```

Структура управляющей строки такова:

```
<символы><заданный формат><символы><заданный
формат>...<символы><заданный формат>
```

Заданный формат еще называют "спецификацией формата". Спецификация формата следующая:

```
{N,M:Axx}
```

где `N` указывает позицию аргумента в списке выводимых переменных (нумерация начинается с 0). То есть если задать `{0}`, то среда поймет, что надо выводить `аргумент1`, указанный в списке аргументов после управляющей строки. `M` в спецификации формата задает ширину области, в которую будет помещено форматированное значение. Например, выводится число 1500. Если задать `M=6`, то в область из шести позиций выведется четырехпозиционное число 1500. Если `M` отсутствует или отрицательно, значение будет выравнено влево (прижато к левой границе области, справа тогда будут лишние пробелы, если область шире выво-

димого числа), в противном случае — вправо. В этом случае число будет прижато к правой границе области вывода, а перед числом окажутся пробелы, не занятые числом, если оно по размеру меньше области. Итак, например, для вывода `аргумента1` в управляющей строке можно задать информацию (управляющая строка сама пишется в кавычках и отделяется от списка аргументов запятой):

```
"Значение аргумента1 равно: {0,6}", аргумент1
```

`Axx` является необязательной строкой форматирующих кодов, которые используются для управления форматированием чисел, даты и времени, денежных знаков и т. д.

В табл. 6.1 приведены поддерживаемые строки стандартных форматов. Строка формата принимает следующую форму: `Axx`, где `A` — описатель формата, а `xx` — описатель точности. Описатель формата управляет типом форматирования, применяемым к числовому значению, а описатель точности — количеством значащих цифр или десятичных знаков форматированного результата.

Таблица 6.1. *Описатели формата вывода чисел*

Формат	Тип форматируемого значения	Пример

`C` или `c`	Валюта	`Console.Write("{0:C}", 2.5);`
		`Console.Write("{0:C}", -2.5);`
		`// $2.50 ($2.50)`

`D` или `d`	Десятичный формат	`Console.Write("{0:D5}", 25);` `// 00025`
`E` или `e`	Инженерный формат	`Console.Write("{0:E}", 250000);` `// 2.500000E+005`

`F` или `f`	Формат с фиксированной точкой	`Console.Write("{0:F2}", 25);`
		`Console.Write("{0:F0}", 25);`
		`// 25.00 25`

`G` или `g`	Общий формат	`Console.Write("{0:G}", 2.5);` `// 2.5`
`N` или `n`	Целочисленный формат	`Console.Write("{0:N}", 2500000);` `// 2,500,000.00`

Таблица 6.1 *(окончание)*

Формат	Тип форматируемого значения	Пример

`X` или `x`	Шестнадцатеричный формат	`Console.Write("{0:X}", 250);`
		`Console.Write("{0:X}", 0xffff);`
		`// FA FFFF`

ГЛАВА 7

Работа с датами и перечислениями

Даты

В языке C# существует тип данного, называемый "дата-время" (`DateTime`). Если объявить переменную такого типа, то станут доступны функции работы с датами и временем. Пусть объявили переменную `d` типа даты/времени: `DateTime d;`. Если теперь после `d` набрать точку, то подсказчик нам высветит перечень функций работы с датой и временем (рис. 7.1).

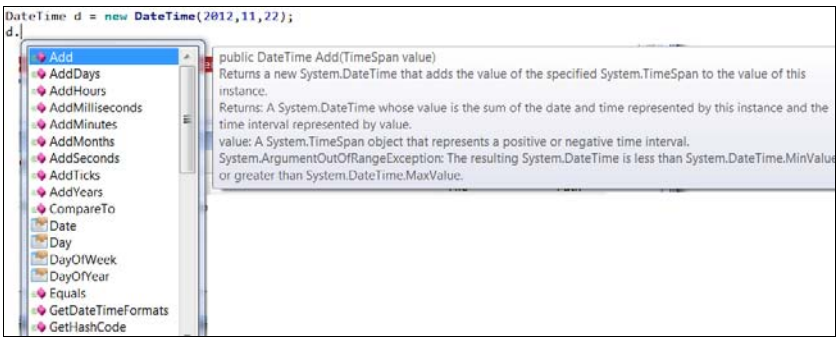


Рис. 7.1. Подсказчик среды SharpDevelop с функциями для типа `DateTime`

На примере работы с массивами мы видели, что массив недостаточно объявить в программе. Его еще надо наполнить содержимым, т. е. инициализировать. Как и любую переменную объявленного типа. С типом `DateTime` такая же картина. После объявления переменную типа `DateTime` надо заполнить какой-то конкретной датой, как мы заполняли

массив конкретными элементами. В C# даты имеют ссылочный тип и поэтому размещаются там, где находятся все данные ссылочного типа — в динамической куче памяти. А на примере массива мы видели, что для размещения в этой куче надо применить оператор `new` с неким инициализатором. Последний наполнит объявленную переменную необходимым содержанием, а оператор `new` весь этот объект разместит в куче, а ссылку на начало объекта в куче положит на полочку, соответствующую имени объекта. Чтобы можно было в дальнейшем обращаться к такому объекту. Инициализатором переменных типа `DateTime` служит функция с таким же именем, в качестве параметров которой выступают параметры `int year`, `int month`, `int day`, т. е. год, месяц и число. Если мы зададим эти данные, то тем самым определим объявленную переменную типа `DateTime` и в дальнейшем сможем с ней работать (что-то прибавлять к ней, что-то отнимать, складывать ее значением с другой подобного типа переменной и т. д.). Например, оператор

```
DateTime dt = new DateTime(2012,11,22);
```

задает значение переменной `dt`, равной 22 ноября 2012 г.

Инициализация объявленной переменной типа `DateTime` может осуществляться и заданной константой: в объекте `DateTime` есть элемент с именем `Now`, который выдает текущую дату и время вашего компьютера. Вот оператор:

```
DateTime MyDate = DateTime.Now;
```

Если этот оператор выполнить, то значение `MyDate` будет, например, таким: 22.11.2012 14:35:22. Последние три числа, выданные через двоеточие, означают "часы:минуты:секунды".

Форматный вывод дат

В листинге 7.1 приведен текст консольного приложения, в котором производится вывод даты и времени в различных форматах. Результат работы приложения показан на рис. 7.2.

Листинг 7.1

```
/* Created by SharpDevelop.
 * User: user
 * Date: 22.11.2012
 * Time: 14:46
 *
```

```
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app17_DateTime
{
  class Program
  {
    public static void Main()
    {
      DateTime data = DateTime.Now;  // Текущие дата
                                     // и время
      int i=30; // Максимальная высота экрана
                // (чтобы вывод поместился)
      int j=20; // После вывода экран надо растянуть
      Console.SetWindowSize (i,j); // Установка размера
                                   // экрана
      Console.Clear();             // Очистка экрана

      Console.WriteLine("d: " + data.ToString("d"));
      Console.WriteLine("D: " + data.ToString("D"));

      Console.WriteLine("f: " + data.ToString("f"));
      Console.WriteLine("F: " + data.ToString("F"));

      Console.WriteLine("g: " + data.ToString("g"));
      Console.WriteLine("G: " + data.ToString("G"));

      Console.WriteLine("m: " + data.ToString("m"));
      Console.WriteLine("r: " + data.ToString("r"));

      Console.WriteLine("s: " + data.ToString("s"));
      Console.WriteLine("u: " + data.ToString("u"));

      Console.WriteLine("U: " + data.ToString("U"));
      Console.WriteLine("y: " + data.ToString("y"));

      Console.WriteLine("H:mm: " + data.ToString("H:mm"));
      Console.WriteLine("HH:mm: "+data.ToString("HH:mm"));

      Console.WriteLine("HH:mm:ss: " +
                        data.ToString("HH:mm:ss"));
```

```
      Console.WriteLine("h:mm tt: " +
                        data.ToString("h:mm tt"));
      Console.WriteLine("hh:mm tt: " +
                        data.ToString("hh:mm tt"));
      Console.WriteLine("MMMM dd: " +
                        data.ToString("MMMM-dd"));
      Console.WriteLine("MM/dd/yyyy: " +
                        data.ToString("MM/dd/yyyy"));
      Console.WriteLine("MM/dd/yyyy HH:mm: " +
                  data.ToString("MM/dd/yyyy HH:mm"));
      Console.WriteLine("MM/dd/yyyy hh:mm: tt: " +
                  data.ToString("MM/dd/yyyy hh:mm tt"));
      Console.WriteLine("MM/dd/yyyy H:mm: " +
                  data.ToString("MM/dd/yyyy H:mm"));
      Console.WriteLine("MM/dd/yyyy HH:mm:ss: " +
                  data.ToString("MM/dd/yyyy HH:mm:ss"));
      Console.WriteLine("dddd, dd MMMM yyyy: " +
                  data.ToString("dddd, dd MMMM yyyy"));
      Console.WriteLine("dddd, dd MMMM yyyy HH:mm:ss: " +
          data.ToString("dddd, dd MMMM yyyy HH:mm:ss"));
      Console.WriteLine("ddd, dd MMM yyyy HH':'mm':'ss'GMT': " +
data.ToString("ddd, dd MMM yyyy HH':'mm':'ss'GMT'"));
      Console.WriteLine("yyyy'-'MM'-'dd'T'HH':'mm':'ss.fffffffK: "
+ data.ToString("yyyy'-'MM'-'dd'T'HH':'mm':'ss.fffffffK"));

      Console.Write("Для продолжения нажмите " +
                    "любую клавишу >");
      Console.Read();
    }
  }
}
```

Функция `ToString(string` *`format`*`)` переводит значение переменной `data` в заданный формат: `data.ToString("D")` — в формат полной даты (22 ноября 2012 г.), а `data.ToString("d")` — в формат короткой записи (22.11.2012). Как происходит перевод даты в остальные форматы, легко догадаться из рис. 7.2. При выводе в строке аргументов функции `Console.WriteLine()` применена операция сцепления строк (+).

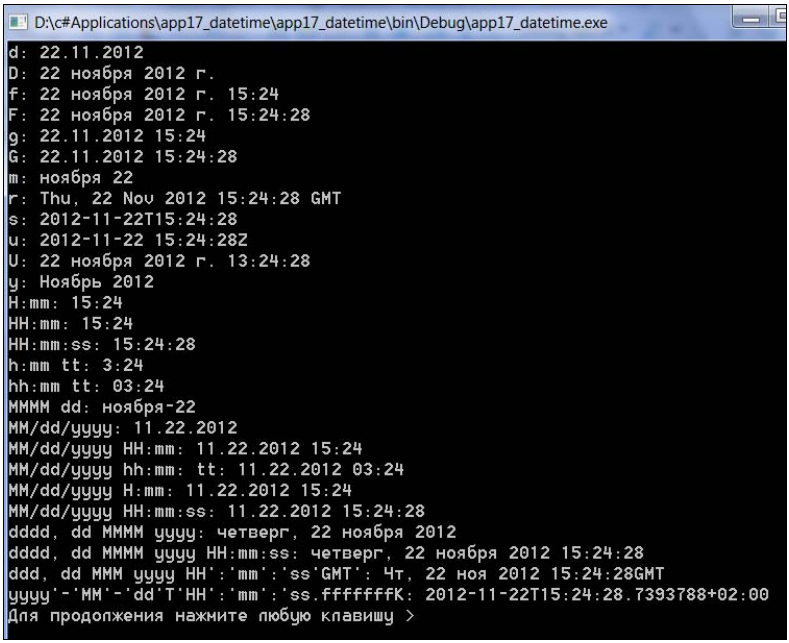


Рис. 7.2. Вывод даты в различных форматах

Операции с датами

Основные операции с датами приведены в листинге 7.2.

Листинг 7.2

```
/* Created by SharpDevelop.
 * User: user
 * Date: 22.11.2012
 * Time: 17:28
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app18_datetime_operations
{
  class Program
  {
```

```
public static void Main(string[] args)
{
  const int Nov = 11; // Для вычисления дней
                      // в 11-м месяце
  DateTime d=DateTime.Now; // Текущая дата
  // Дата предыдущего дня
  DateTime d_1 =d.AddDays(-1);

  // Дата следующего дня
  DateTime d_tomor = d.AddDays(1);

  // Сколько дней осталось до конца месяца
  int dm=DateTime.DaysInMonth(2012,Nov);
    // дней в месяце
  int to_end=dm - d.Day;  // Минус текущий день

  // Дата i-го дня от заданной даты назад
  int i=12;
  DateTime d_back=d.AddDays(-i);

  // Выделение из даты года, месяца и числа.
  // var - неявно заданный тип.
  // Компилятор сам определит его по использованию
  // в программе

  var now = DateTime.Now;
  int y=now.Year;
  int m=now.Month;
  int day=now.Day;

  // Разность дат: сначала находится разность
  // в единицах "часы-минуты-секунды",
  // а затем результат переводится в дни при выводе
  DateTime dt1 =  DateTime.Now;
  DateTime dt2 = new System.DateTime(2012, 11, 2);
  TimeSpan diffResult = dt1.Subtract(dt2);

  Console.WriteLine("Текущая дата  = " +
                    dt1.ToString());
  Console.WriteLine("Заданная дата = " +
                    dt2.ToString());
```

```
      Console.WriteLine("Текущая дата — заданная = " +
                        diffResult.Days); // Целая часть
      Console.WriteLine("Текущая дата — заданная = " +
          diffResult.TotalDays); // Целая часть
                                 // плюс дробная часть
      // Полночь текущей даты
      var today = DateTime.Today;
      Console.WriteLine("Полночь текущей даты = " +
                        today.ToString());
      // 5 часов, 30 минут, 10 секунд
      var interval = new TimeSpan(5, 30, 10);
      Console.WriteLine("Интервал времени = " +
                        interval.ToString());
      // сегодня в 05:30:10
      var newTime = today + interval; // Отсчет идет
                                      // от полуночи
      Console.WriteLine("Добавка интервала времени " +
              "к началу дня = " + newTime.ToString());

      // Завтра в 05:30:10
      var tomorrowNewTime = newTime.AddDays(1);
            // Добавляется один день
      Console.WriteLine("Завтра в 5_30 = " +
                        tomorrowNewTime.ToString());
      // Вчера в 05:30:10
      var yesterdayNewTime = newTime.AddDays(-1);
            // Вычитается один день
      Console.WriteLine("Вчера в 5_30 = " +
                        yesterdayNewTime.ToString());
      // 10 лет от полуночи текущей даты
      var seeYouNextDecade = today.AddYears(10);
            // Добавляется 10 лет
      Console.WriteLine("10 лет от полуночи " +
          текущей даты = " + seeYouNextDecade.ToString());
      Console.Write("Для продолжения нажмите " +
                    "любую клавишу >");
      Console.Read();
    }
  }
}
```

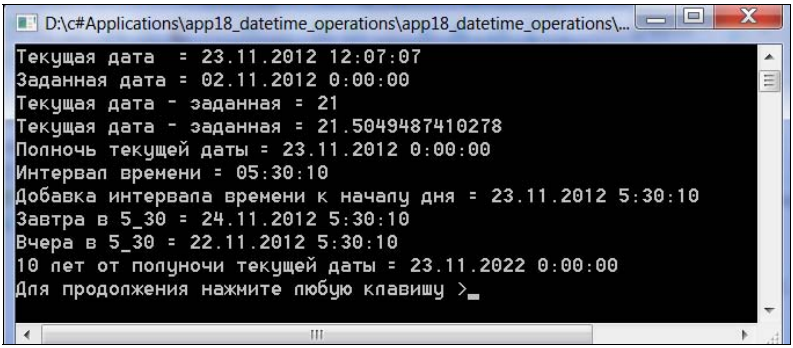


Рис. 7.3. Результаты работы с датами

Результат работы этой программы показан на рис. 7.3.

Все пояснения приведены в тексте программы.

Перечисления

Тип переменной, определяемый как "перечисление", предоставляет способ задания набора именованных констант (именованных, значит, что таким константам присвоены имена, чтобы с ними было удобно работать), который можно назначить переменной. Например, предположим, что нужно определить переменную, значение которой должно представлять день недели. Имеется только семь осмысленных значений, которые может принимать переменная. Для определения этих значений можно использовать тип перечисления, который объявляется с помощью ключевого слова `enum`. Далее объявлены два типа (дни и месяцы) и представлен именованный набор значений этих типов. Обратите внимание, что именованный набор задается так же, как инициализируется константами массив — другой тип переменных.

```
enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday };
enum Months : byte { Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep,
Oct, Nov, Dec };
```

Обратите внимание, что здесь определяются не переменные, которые будут участвовать в расчетах программы, а задаются только типы "перечисление" с разным содержанием этих типов: одно содержание — дни недели, другое — месяцы года. Сама переменная программы должна быть объявлена заданным типом "перечисление". Например, `Days d;` или `Months m;`.

Обращаться к конкретным значениям — к конкретным константам конкретного перечисления принято так: `Days.Sunday`, `Months.Feb` и т. д. То есть переменная `d` может принимать одно из значений `Days.Sunday`, `Days.Monday` и т. д.

По умолчанию базовым типом каждого элемента перечисления является `int`. Можно задать другой целочисленный тип, используя двоеточие, как показано на примере задания переменной `Months`.

Константы, находящиеся в фигурных скобках, называются *перечислителями*. По умолчанию первому перечислителю определено значение 0, а значение каждого последующего увеличивается на 1. Например, в нижеследующем перечислении `Sat` — это 0, `Sun` — 1, `Mon` — 2 и т. д.:

```
enum Days {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
```

Перечислители могут использовать так называемые инициализаторы для переопределения значений по умолчанию, как показано в следующем примере:

```
enum Days {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};
```

В этом перечислении последовательность элементов принудительно начинается с 1, а не с 0. Однако рекомендуется включить константу, значение которой равно нулю.

Каждый тип перечисления имеет базовый тип, который может быть любым целым типом, исключая `char`. Как было сказано ранее, по умолчанию базовым типом элементов перечисления является тип `int`. Для объявления перечисления другого целочисленного типа, такого как байт, поставьте двоеточие после идентификатора, за которым следует тип, как показано в следующем примере:

```
enum Days : byte {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};
```

Для перечисления утверждены следующие типы: `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` или `ulong`.

Переменной типа `Days` может быть назначено любое значение, входящее в диапазон базового типа. Значения не ограничены именованными константами.

Имя перечислителя не может содержать пробелов.

Базовый тип указывает размер памяти, выделенный для каждого перечислителя. Однако для преобразования из типа `enum` в целочисленный тип необходимо явное приведение. Например, следующий оператор

назначает перечислитель `Sun` переменной типа `int`, используя приведение для преобразования `enum` в `int`.

```
int x = (int)Days.Sun;
```

Как и в случае с любой другой константой, все ссылки на отдельные значения перечисления преобразуются в числовые литералы во время компиляции. Значения перечислений часто используются в операторах `switch`.

Рассмотрим примеры.

- В этом примере объявляется перечисление `Days`. Два перечислителя явно преобразуются в целые числа и назначаются целочисленным переменным.

```
public class EnumTest
{
  enum Days { Sun, Mon, Tue, Wed, Thu, Fri, Sat };

  static void Main()
  {
    int x = (int)Days.Sun;
    int y = (int)Days.Fri;
    Console.WriteLine("Sun = {0}", x);
    Console.WriteLine("Fri = {0}", y);
  }
}
/* Результат:
   Sun = 0
   Fri = 5
*/
```

- В данном примере для объявления `enum`, члены которого имеют тип `long`, используется параметр базового типа.

```
public class EnumTest2
{
  enum Range : long { Max = 2147483648L, Min = 255L };
  static void Main()
  {
    long x = (long)Range.Max;
    long y = (long)Range.Min;
    Console.WriteLine("Max = {0}", x);
```

```
      Console.WriteLine("Min = {0}", y);
    }
  }
  /* Результат:
     Max = 2147483648
     Min = 255
  */
```

- Пример программы, в которой перечисление используется в операторе `switch`, приведен в листинге 7.3.

Листинг 7.3

```
/* Created by SharpDevelop.
 * User: user
 * Date: 23.11.2012
 * Time: 15:13
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app19_enum
{
  class Program
  {
    public enum DaysOfWeek  // Объявление типа
                            // "перечисление"
    {
      Monday = 0,
      Tuesday = 1,
      Wednesday = 2,
      Thursday = 3,
      Friday = 4,
      Saturday = 5,
      Sunday = 6
    }
    static void Main()
    {
      WriteText(DaysOfWeek.Sunday);  // Задание значения
                                     // параметра
      Console.ReadLine();
    }
```

```
    // В качестве параметра функции — переменная days типа
    // "перечисление"
    static void WriteText(DaysOfWeek days)
    {
      switch (days)  // Обработка значений переменной days
      {
        case DaysOfWeek.Monday:
          Console.WriteLine("Понедельник");
          break;
        case DaysOfWeek.Tuesday:
          Console.WriteLine("Вторник");
          break;
        case DaysOfWeek.Wednesday:
          Console.WriteLine("Среда!");
          break;
        case DaysOfWeek.Thursday:
          Console.WriteLine("Четверг");
          break;
        case DaysOfWeek.Friday:
          Console.WriteLine("Пятница");
          break;
        case DaysOfWeek.Saturday:
          Console.WriteLine("Суббота");
          break;
        case DaysOfWeek.Sunday:
          Console.WriteLine("Завтра — понедельник");
          break;
      }  // switch
    } // WriteText
  }
}
```

Результат показан на рис. 7.4.

Итак, видим, что перечисления улучшают наглядность текста программы.

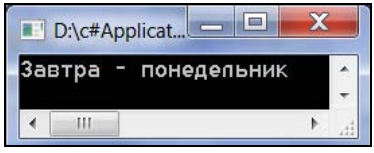


Рис. 7.4. Обработка перечисления оператором `switch`

Типы перечислений как битовые флаги

Тип перечисления можно использовать для определения значений отдельных двоичных разрядов в переменной типа `int` (еще говорят: для определения битовых флагов). Дело в том, что биты давно используются программистами для задания состояний различных объектов. Например, если нулевой бит некоторой переменной равен единице, то пусть это означает, что свет в комнате включен, а ноль — свет выключен. Такой бит, отражающий некоторое состояние, называют *флагом*. И проблема состоит в том, как спуститься на уровень битов переменной, как выделить значение одного бита или группы битов. А если вы и выделили некий бит, то надо постоянно помнить, что пятый бит означает то-то, 8-й — то-то и т. д. Это создает определенные неудобства программисту. Если же использовать перечислительные типы, то можно этим битам-флажкам присвоить обычные наименования и работать с ними. Ведь экземпляр типа "перечисление" может хранить любую комбинацию значений, определенных в списке перечислителя.

Если перед перечислением поставить атрибут `[Flags()]`, то мы сможем работать с таким перечислением как с набором битов. В частности, мы сможем применять битовые операции. Программа, использующая битовые флажки, приведена в листинге 7.4.

Листинг 7.4

```
/* Created by SharpDevelop.
 * User: user
 * Date: 23.11.2012
 * Time: 16:43
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app20_flags
{
  class Program
  {
    [Flags()]
    public enum MilitaryType   // Тип используемого
                               // вооружения
    {
      Marine = 1,
```

```
    Land = 2,
    Cosmic = 4,
    AllType = Marine | Land | Cosmic
  }
  public static void Main(string[] args)
  {
    // Установка флажков Land и Cosmic в type0:
    MilitaryType type0 = MilitaryType.Land |
                         MilitaryType.Cosmic;
    // Console.WriteLine(type0);
    // Выведется "Land, Cosmic"
    Console.WriteLine("Установлены флажки {0}",type0);

   // Установка флажка Marine в type0:
    type0 |= MilitaryType.Marine;

    // Проверка флажка Land в type0: он должен быть
    // установлен в предыдущих операторах
    bool b=System.Convert.ToBoolean(type0 &
                  MilitaryType.Land);
    if(b)
      Console.WriteLine("Флажок Land в type0 " +
             "установлен"); // Выведется "Land, Cosmic"

     Console.Read();
  }
 }
}
```

Результат показан на рис. 7.5.

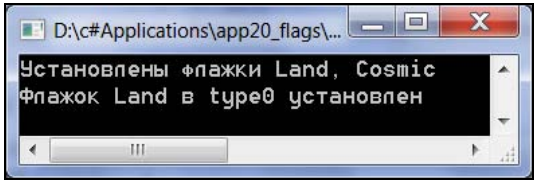
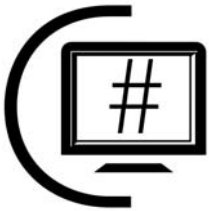


Рис. 7.5. Работа с битовыми данными



ЧАСТЬ II

Объектно-ориентированное программирование

ГЛАВА 8

Введение в классы

За 50—60 лет своего развития программирование прошло очень большой путь: от начала разработки программ в машинных кодах через создание и применение простейших языков символического кодирования до выпуска современных гигантских продуктов, обеспечивающих работу клиента в распределительных сетях на основе новейших подходов. Разработка и создание какой-нибудь стандартной программы перевода данных из одной системы счисления в другую, новой процедуры сортировки массива данных, настраиваемой на потребности заказчика программы обработки анкет, и прочее было заметным событием в те годы. Мысль тех, кто был связан с программированием, неустанно работала в направлении, как упростить очень сложный труд программиста, повысить производительность его труда, добиться надежности и качества программ. Сначала было замечено, что многие части машинных алгоритмов в разных задачах повторяются. Это привело к созданию так называемых стандартных программ — процедур, которые выполняли эти стандартные действия, например, такие как перевод чисел из одной системы счисления в другую, сортировка массива данных и т. п. Каждому программисту уже не надо было, например, беспокоиться о создании участка сортировки данных в своей программе. Он просто пользовался стандартной процедурой, поставляемой вместе с математическим обеспечением вычислительной машины. И это ускоряло процесс разработки программы. Потом пришли к понятию структурного программирования. Это был значительный шаг вперед. Была разработана методология создания программы, в основе которой лежал принцип создания программы как структуры, состоящей из подпрограмм, создаваемых для обработки повторяющихся блоков программы, и принципа выполнения программы "сверху вниз", когда в последовательности выполнения опе-

раторов и отдельных блоков программы не было возврата назад. Умри, но сделай так, чтобы не было возвратов назад! Это обеспечивало более быструю отладку программы. Выход из цикла — только вперед (вспомните оператор `break`)! Возврат назад — только в пределах оператора цикла! Передача управления из оператора `if...else` — только вперед! Долой оператор `goto`! Даже была доказана теорема о том, что любую схему алгоритма можно представить в виде композиции вложенных блоков `begin` и `end`, условных операторов `if`, `then`, `else`, циклов с предусловием (`while`) и, может быть, дополнительных логических переменных (флагов). Подход был серьезный. Но жизнь не стоит на месте. С появлением более новой техники — мощных компьютеров — открылись и новые возможности, о которых раньше даже не мечтали. Да и о чем ты можешь мечтать, когда оперативная память твоего "компьютера" — всего 1024 малюсеньких ячейки (1 К), а внешняя — один ленточвод на 64 К, который практически все время не работает! И скорость в 3 тыс. операций в секунду.

Оказалось, что у процедурно-модульно-структурного подхода имеются значительные недостатки, которые можно устранить, используя мощь современных компьютеров. Во-первых, в программе данные и подпрограммы их обработки (процедуры и функции) формально никак не связаны. А хотелось бы, чтобы было наоборот. Например, вы решаете задачу по обработке данных о каком-то доме. Удобнее было бы, чтобы данные по дому и функции по обработке этих данных хранились в одной "коробке". Причем, было бы еще удобнее, если бы эта "коробка" подошла и для решения задачи по другому дому с такими же характеристиками, как и первый дом. То есть, говоря языком стандартизации, чтобы "коробка" была стандартной для данного класса домов. Во-вторых, данные в "коробке" не должны быть доступны всем, кто работает с коробкой. Они должны быть защищены от прямого доступа к ним. Только через функции, которые прячутся в этой "коробке". В-третьих, было бы полезным, чтобы элементы одного дома, спрятанные в "коробке", не хранились там без дела, когда данные дома не обрабатываются, а были бы использованы, если они подходят и к некоторым другим домам. Чтобы можно было брать такие элементы, не создавая их заново для другого класса домов, добавлять к ним новые, необходимые для нового класса, и создавать свою новую "коробку" для нового класса домов. И т. д. Это уже совсем другой подход к программированию. Он ориентирован не на модули-процедуры-функции, а на некие объекты, на то, чтобы в программе они создавались (описывались), чтобы внутри них хранились функции по обработке данных этих объектов (функции, естественно, надо разработать, но в рамках описания объекта), чтобы

данные по объектам вводились и попадали в сам объект и никуда больше, а полезные для других объектов элементы уже готового объекта можно было бы унаследовать другому создаваемому объекту. Раньше, например, могли мы, имея некоторую стандартную процедуру, формально взять из нее какую-то ее часть и применить в другой процедуре? Не могли. А в системе объектно-ориентированного программирования (ООП), к пониманию которого мы подводим читателя, это вполне возможно. Мы видим, что у такого подхода в программировании более высокий уровень, говоря языком политики, обобществления. От простейших стандартных программ по переводу чисел из десятичной системы счисления в двоичную мы приходим к созданию стандартных конструкций по описанию и обработке данных человека, дома, автомобиля, поликлиники, завода, страны.

Ранее мы изучили различные типы данных: целых чисел, чисел с плавающей точкой, строковых данных, типы организации данных, названных массивами, типы организации данных, названных перечислениями. Для ООП характерны типы данных, которые называются *классами*. Например, для типов "массивы" и "перечисления" мы объявляли (описывали по специальной схеме) сам тип, потом объявляли некую переменную этого типа. Затем инициализировали эту переменную некоторыми значениями. У нас существовали правила, как обращаться к элементам объявленного типа данных с помощью имени переменной. Например, если у нас был объявлен массив `M[]` данных, то элемент массива мы доставали, указывая в квадратных скобках номер (индекс) этого элемента в массиве: `M[i]`. Для перечислимого типа данных была своя схема описания и свое правило обращения к элементу этого типа данных. Для типа "класс" соблюдаются такие же правила: этот тип данных описывается по своей схеме (шаблону), переменная этого типа объявляется по общим правилам (`<имя_типа> <имя_переменной>`), по своим правилам переменная инициализируется, т. е. ей придаются некоторые начальные значения. Обратим внимание на два момента: когда мы описываем тип данного (в нашем случае тип "класс"), то описываем фактически схему, шаблон, по которому в дальнейшем будет создан объект данного типа. А когда мы этот шаблон наполняем содержанием, то тем самым создаем объект с данным содержанием, т. е. нечто, что можно пощупать. Объект уже надо размещать в памяти. Он требует пространства. Схема, шаблон тоже где-то хранятся, тоже требуют некоей памяти, но это как бы вспомогательная, не основная память. Вот, например, мы объявили целочисленную переменную `i`. Компилятор для нее создает некий шаблон: выделяет в специальной памяти 4 байта. И все. Но когда мы этой переменной присваиваем, скажем, значение 5, то тем самым создаем кон-

кретный числовой объект, с которым можно работать. Просто с объявлением `i` еще работать нельзя. Не с чем. А с объектом, даже можно сказать, с экземпляром этого шаблона `int i`, который равен 5, работать можно. Если мы присвоим переменной `i` другое значение, например 6, можно сказать, что мы из шаблона `int i` получили (создали) другой объект: число 6.

Точно так же и с классом: мы его описали, получили просто описание, шаблон. И ничего больше. А как только мы по этому шаблону создали переменную и наполнили ее неким содержанием, то получили объект, соответствующий данному содержанию. Говорят, что получили экземпляр класса. Наполнили переменную другим содержанием — получили другой объект с другим содержанием, другой экземпляр класса. Объекты (экземпляры) уже размещаются в памяти. Класс — это ссылочный тип. Поэтому он размещается в динамической куче оператором `new`. А почему сделали класс ссылочным типом данного? Потому что объекты, получаемые из этого класса, могут быть огромными (например, какой-нибудь крупный завод). А перемещать в памяти, как мы видели, ссылочные данные намного проще нессылочных: переслал только ссылку кому надо и не тронул огромный массив. Большой выигрыш в скорости обработки.

Прежде чем изучать конкретную структуру класса, отметим, что класс как совокупность элементов (членов класса) состоит из членов, которые называются *полями*, и из членов, оперирующих данными этих полей. Эти последние могут быть конструкторами, методами, свойствами, событиями и др. *Методы* — это функции. Так функции называются в классах. *Конструкторы* — это методы, которые позволяют инициализировать класс, тем самым создают из класса объект, размещая его в памяти. То есть конструктор — это обычная (по структуре) функция, получающая на свой вход данные, которые присваиваются полям класса. Иными словами, из пустого шаблона за счет задания полей получается объект.

Например, пусть мы имеем класс `MyCar`. Это тип данного, как мы видели. Объявляем переменную этого типа. Например, `car` (автомобиль): `MyCar car;`. Но это пока ничто: объявление и не более того. С такой переменной работать нельзя. Допустим, у автомобиля, класс которого мы хотим описать, есть такие характеристики (поля): марка (`Type`), имя владельца (`Name`) и скорость (`Speed`). Тогда функция, которая призвана задать эти поля, должна иметь в своем заголовке эти три параметра (`string Type`, `string Name`, `float Speed`). А каким должно быть имя у такой функции? Оно специфично и совпадает с именем класса. Вполь-

не логично. Ведь конструктор — это то, что создает из класса объект. Итак, для нашего предполагаемого класса его конструктор будет иметь вид

```
MyCar(string Type, string Name, float Speed)
```

А что должно быть в теле конструктора? Операторы, которые присваивают значения полям класса. Если поля в классе описаны как `string type`, `string name`, `float speed`, то в теле должны быть операторы:

```
Type=Type; name=Name; speed=Speed;
```

То есть в итоге конструктор класса `MyCar` будет иметь вид:

```
MyCar(string Type, string Name, float Speed)
{
  type=Type;
  name=Name;
  speed=Speed;
}
```

Если мы теперь из класса хотим создать конкретный объект, например автомобиль Чака Норриса, то должны записать:

```
MyCar Ch_nor_car = new MyCar("Porshe", "Chuck Norris", 250.0);
```

По этому оператору конструктор создаст объект с именем `Ch_nor_car`, оператор `new` разместит объект (или — экземпляр класса `MyCar`) в динамической куче и выдаст адрес начала объекта в этой куче. Адрес будет положен на полочку для переменной `Ch_nor_car`.

Ну а теперь надо бы посмотреть, как создается (описывается) класс. Простейший вид описания класса такой:

```
class Car
{
}
```

Здесь `class` — ключевое слово, `Car` — имя класса. А где же конструктор? Здесь конструктор не указан. Он идет по умолчанию. Вообще, если в классе конструктор не указывается, то он по умолчанию берется из класса `Object` — специального класса, из которого берут свое начало остальные классы. Они, как говорят, потомки класса `Object`. Конструктор по умолчанию инициализирует поля создаваемого класса значениями, принятыми для полей класса по умолчанию. Вообще-то тело класса не пусто. Иначе зачем этот класс создавать? В теле должны быть определены члены класса и методы (т. е. функции), которые работают с полями-членами и другими членами класса. Ну и, конечно, задан конст-

руктор класса. Конструкторов может быть более одного, потому что может потребоваться создавать объекты класса на основе не всех полей, а некоторых. Но все конструкторы должны обязательно иметь имя класса, но с разными параметрами. Члены класса группируются в две секции: общедоступные члены (те, что доступны для обращения к ним не только из методов самого класса, но и из других программ приложения) и приватные члены (к ним имеют доступ только методы данного класса). Члены общедоступной секции помечаются ключевым словом `public`, а приватной (частной) — словом `private`. Есть еще одно ключевое слово, обеспечивающее доступность членов класса. Это слово `protected` (защищенный). У класса есть свойство наследовать члены другого класса (об этом мы узнаем подробнее дальше). Наследуются только те члены класса, которые имеют атрибут `public`. Если мы хотим, чтобы какие-то члены создаваемого нами класса, попавшие в приватную секцию, тоже наследовались, им надо присвоить атрибут `protected`. Тогда такие члены будут попадать в другие классы только наследным путем.

Пример программы, в которой создается класс с двумя конструкторами и одним методом обработки полей класса, приведен в листинге 8.1.

Листинг 8.1

```
/* Created by SharpDevelop.
 * User: user
 * Date: 24.11.2012
 * Time: 19:51
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app21_class
{
  class MyCar
  {
    private string name;
    private int speed;
    private string owner_name;

    // Конструктор класса
    public MyCar(string Name,int Speed,string Owner_name)
```

```
    {
      name=Name;
      speed=Speed;
      owner_name=Owner_name;
    }

    // Второй конструктор класса
    public MyCar(string Name,int Speed)
    {
      name=Name;
      speed=Speed;
    }

    // Метод1 класса
    public int M1()
    {
      speed=speed*2;
      return(0);
    }
  }  // class

  class Program
  {
    public static void Main()
    {
      // Чтобы создать объект (с помощью конструктора),
      // надо, чтобы конструктор был доступен в программе
      // Main(), которая находится в другом классе.
      // Поэтому конструктор должен иметь атрибут public.
      // Это же касается и метода M1()

      MyCar car = new MyCar("Лада",120,"Иванов");
      car.M1();

      // car уже выше объявлена как тип MyCar
      car = new MyCar("Лада",120);
      car.M1();

      Console.WriteLine("Press any key to continue... ");
      Console.Read();
    }
  }
}
```

Итак, в этой программе создается класс `MyCar` с тремя приватными полями (имя автомобиля, скорость и имя владельца).

```
private string name;
private int speed;
private string owner_name;
```

Так как все поля приватные, то из внешней программы, в частности из `Main()`, к ним доступа нет. Но есть доступ через метод `M1()`, который имеет атрибут `public` (общедоступный). То есть в данном случае, если мы хотим что-то с полями делать, то имеем только одно средство, которое в классе определено. Это метод `M1()`. Другого не дано. В классе определены два конструктора, которые позволяют создать два разных объекта. Один — на основании инициализации трех полей, другой — двух полей. Конструкторы должны быть общедоступными, чтобы вне класса с их помощью создавать объекты. Поэтому у конструкторов имеется атрибут `public`. Единственный метод класса (`M1()`) увеличивает скорость автомобиля в два раза. В основной программе `Main()` создаются два объекта двумя конструкторами, и в каждом случае выполняется метод `M1()`. Чтобы было видно, что же получается при вычислениях, программа была запущена в режиме отладки. Точки останова были поставлены так, чтобы можно было пошагово следить, как формируется объект `car` разными конструкторами. Значения, получаемые полями объектов в результате их инициализации конструкторами и воздействия на них метода `M1()`, показаны на рис. 8.1 и 8.2.

Заметим, что для обращения к элементу класса надо назвать класс и через точку написать имя требуемого элемента. Отметим также, что после определения полей класса, которые в объектах, полученных из этого класса, будут использоваться для представления их состояния, определяют другие члены, которые моделируют поведение объектов. В нашем случае — это метод `M1()`. Размещение одной строкой объекта, как показано в предыдущем листинге (`MyCar car = new MyCar("Лада", 120,"Иванов");`), не обязательно: можно сначала объявить тип переменной (`MyCar car;`), а потом создать экземпляр класса (объект):

```
car = new MyCar("Лада",120,"Иванов");
```

Конструктор класса — это специальный метод, который вызывается при создании объекта. Этот метод не имеет, в отличие от других методов, никакого возвращаемого значения: ни типа `int`, ни типа `double`, и даже типа `void` не имеет. В C# каждый класс снабжается своим конструктором по умолчанию, хотите вы этого или нет. Это метод с именем класса,

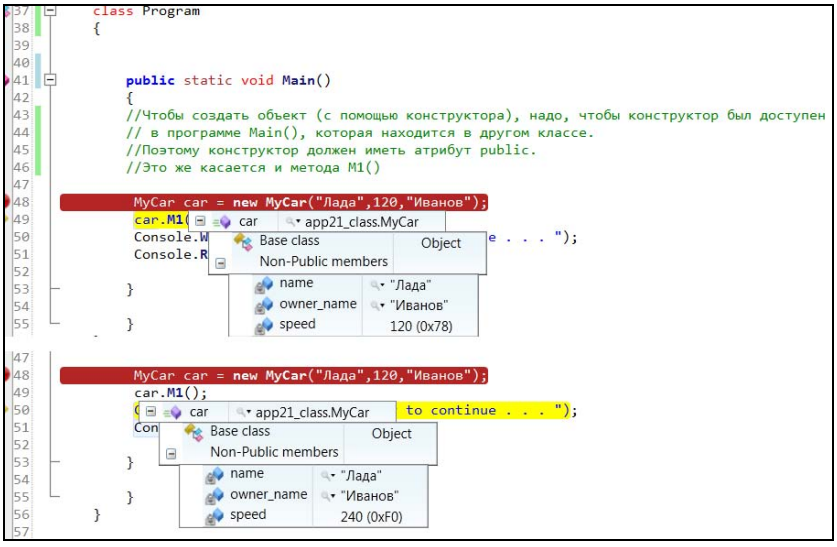


Рис. 8.1. Значения полей объекта после работы первого конструктора и метода `M1()`

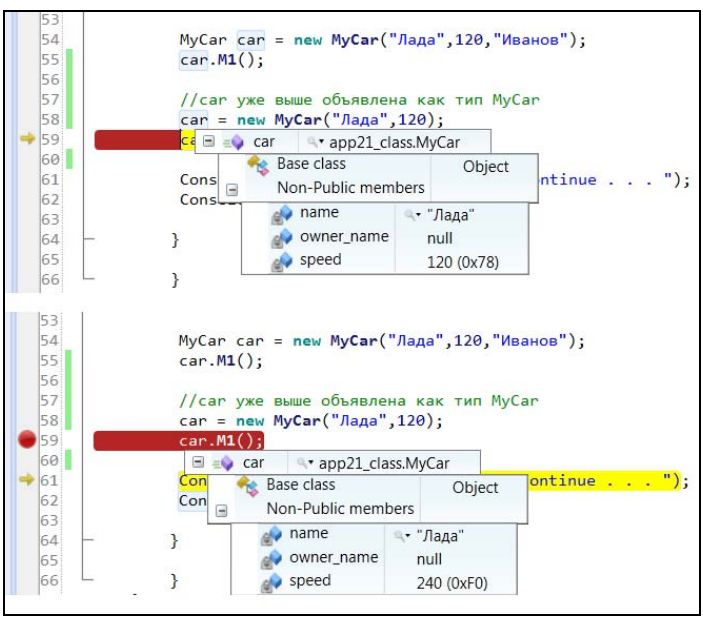


Рис. 8.2. Значения полей объекта после работы второго конструктора и метода `M1()`

но без параметров. То есть, когда вы создаете экземпляр класса (или объект, что то же самое), вы должны написать, например,

```
MyCar car = new MyCar();
```

В этом случае полям класса будут присвоены значения, принятые в языке по умолчанию. Если же вам не подходят значения по умолчанию, то вы сами можете создать свой конструктор по умолчанию (т. е. без параметров), задав в его теле свои, нужные вам, значения полей класса. Например, для нашего случая можно было бы задать такой конструктор по умолчанию:

```
public MyCar()
{
  name="Лада";
  speed=120;
  owner_name="Иванов";
}
```

В нашем примере в классе определены два конструктора. У них разное количество параметров, но типы параметров одинаковы. Но может быть и так, что один конструктор от другого отличается не только количеством параметров, но и их типами. Определение методов с одним и тем же именем, но с разным количеством и типом параметров называют *перегрузкой метода*. Таким образом, класс `MyCar` имеет перегруженный конструктор, чтобы предоставить более одного способа создания из этого класса объекта.

Ключевое слово *this*

В переменной с этим именем хранится адрес текущего объекта, т. е. объекта, с которым в данном месте программы происходит работа. Этот элемент введен в язык для разрешения неоднозначных ситуаций, когда, например, вы в конструкторе назвали параметр таким же именем, как поле. Формально компилятор пропустит этот вариант, но при выполнении программы можно получить не тот результат. Чтобы в таких случаях различать, какая переменная к чему относится при одинаковых именах, и введено это ключевое слово. Например, в случае нашего примера мы могли бы определить заголовок конструктора как

```
public MyCar(string name, int Speed, string Owner_name)
```

Здесь первый параметр по имени совпадает с именем поля класса. Поэтому в теле конструктора мы вынуждены записать строку

```
{
  name=name;
  ...
}
```

Компилятор посчитает, что строка присваивается самой себе, и полю `name` класса ничего не присвоит. Поэтому в таких ситуациях надо использовать слово `this`, которое в данном случае подскажет компилятору, что `name` в левой части от знака присвоения принадлежит текущему объекту, а именно тому, который создается из класса, а не конструктору. В этом случае конструктор будет выглядеть так:

```
public MyCar(string name, int Speed, string Owner_name)
{
  this.name=Name;
  speed=Speed;
  owner_name=Owner_name;
}
```

> ***Примечание***
>
> Члены класса могут иметь атрибут `static` (об этом будет сказано далее). В этом случае применение `this` вызовет ошибку компиляции, потому что статические члены действуют на уровне класса, а не объекта (они для того и создаются, чтобы быть доступными всем объектам), а на уровне класса `this` не существует.

Ключевое слово *static*

Члены класса могут иметь атрибут `static`. Необходимость его введения была вызвана теми обстоятельствами, что создаваемые из класса объекты должны были иметь по некоторым полям, например, общие значения или обладать общими методами. Это удобно, т. к. нет необходимости иметь в каждом объекте один и тот же метод, достаточно получить его на уровне класса, из которого создается этот метод. Например, вы создаете класс по персоналу предприятия. Допустим, каждый объект, получаемый из этого класса, — это отдельный работник со всеми своими характеристиками. У всех работников могут быть одинаковые, например, некоторые процентные надбавки. Такие поля следует делать статическими, и они будут во всех объектах одинаковы. Достаточно изменить их на уровне класса, как изменения отразятся во всех объектах. Или методы некоторых расчетов: изменился алгоритм, метод изменяется на уровне класса, и для всех сотрудников обновился метод расчета.

Ранее мы пользовались методом `WriteLine()`. Но мы ничего не говорили, откуда он и почему так записывается (`Console.WriteLine()`). Теперь мы можем сказать, что этот метод из класса `Console`, и он имеет атрибут `static`. Он не вызывается на уровне объекта (мы не можем написать `Console con = new Console()`, создавая объект из класса, а потом писать `con.WriteLine()`; компилятор не пропустит), а вызывается напрямую из класса `Console`. Если в некотором классе определены только статические члены, такой класс можно назвать обслуживающим. Пример тому — класс `Console`. Имеется, например, и класс `Math`, предоставляющий возможность работы с математическими величинами и функциями.

Статические члены класса могут оперировать только другими статическими членами, в противном случае на этапе компиляции возникнет ошибка.

Для обращения к статическому члену обязательно надо указывать имя класса, которому этот член принадлежит (как, например, мы это делали в случае применения функции (а теперь мы знаем — статического метода) консольного вывода).

Итак, при создании статических полей класса они распределяются по всем экземплярам класса. Для них в памяти выделяется специальный участок, который не сбрасывается, когда объект уничтожается. В то время как нестатические поля — это независимые копии полей для каждого объекта. И они при ликвидации объекта уничтожаются вместе с ним, т. е. память от них освобождается. Статические поля сохраняются, а нестатические — не сохраняются. Поэтому при создании класса одной из задач разработчика является определение, какие поля будут статическими, а какие нет.

Для вящей убедительности еще один пример. Пусть у работника требуется рассчитать некую процентную ставку. Она для всех работников рассчитывается по одному и тому же алгоритму, не зависящему от работника. Пусть работников на заводе три тысячи. То есть создано три тысячи экземпляров класса `Персонал`. И в каждом будет присутствовать метод расчета процентной ставки. Если алгоритм расчета изменился, надо будет из класса снова формировать три тысячи экземпляров с новым методом, если только метод не имеет атрибута `static`. А с этим атрибутом формировать заново экземпляры нет надобности, потому что в каждом экземпляре обращение к этому методу идет с указанием класса, а не объекта, поскольку метод — статический. А в классе метод уже изменен. И оттуда он будет доставаться уже измененным.

Статический конструктор

При создании класса создаются статические поля, которые прямо при создании получают конкретные (статические) значения. Например, `static Rate=0.12;`. А как быть в случаях, когда значение статического поля определяется не в момент создания класса, а в момент работы приложения, когда из приложения надо обратиться к базе данных, чтобы взять оттуда, например, значение процентной ставки? И это все надо сделать до создания объекта. И поле процентной ставки имеет атрибут `static`. Такие проблемы решает так называемый *статический конструктор*, который выполняется всего один раз перед конструктором, создающим объекты. Статический конструктор используется для инициализации любых статических данных или для определенного действия, которое требуется выполнить только один раз. Он вызывается автоматически перед созданием первого экземпляра или когда идет первое обращение к статическому члену. Вот пример:

```
class SimpleClass
{
  // Статическая переменная, которая должна быть
  // инициализирована во время выполнения программы:
  static long dt;

  // Статический конструктор, который вызывается только
  // один раз перед любым конструктором, создающим
  // экземпляр класса, или при первой встрече
  // переменной dt:

  static SimpleClass()
  {
    dt = DateTime.Now.Ticks;
  }
}
```

Исполняющая среда вызовет этот конструктор перед созданием экземпляра класса `SimpleClass` и перед первым обращением к переменной `dt`.

Статические классы

Если все члены класса — статические, то имеет смысл вынести слово `static` "за скобки" и объявить весь класс статическим. Например, запись `static class MyClass {}` будет правильной.

Принципы объектно-ориентированного программирования

Таких принципов — три: инкапсуляция, наследование и полиморфизм. В предыдущем материале мы частично касались этих понятий, но не заостряли на них внимание. Теперь настало время рассмотреть сущность этих принципов подробнее.

Инкапсуляция

Капсула по латыни означает "коробочка", "ин" — предлог "В". Поэтому дословный перевод первого принципа ООП — "в коробочке". В нашем случае это условная коробочка, черный ящик, в котором прячутся все детали реализации проблемы, но с помощью этого ящика программист может решить саму проблему, не отвлекаясь на детали ее реализации. Кроме того, "коробочка" содержит (хранит) данные, причем таким образом, что к ним нет прямого доступа извне. Есть доступ только с помощью методов, хранящихся тоже в коробочке. Таким способом обеспечивается защита данных от внешнего вмешательства. Тот, кто в старые времена программировал ввод-вывод, знает, какая это была морока. Сегодня есть такой класс, как, например, `Console`, в котором содержатся (спрятаны в деталях) методы `Read()`, `ReadLines()`, обеспечивающие ввод данных с клавиатуры, и ни у одного программиста не болит голова, что надо будет делать в программе блок ввода-вывода. Все спрятано в классе `Console`. Пиши себе строку `string s = ReadLines();`, и твоя проблема ввода с клавиатуры решена.

Другая сторона — в коробочке упрятаны данные о состоянии объекта, который описан в ней. К этим данным просто так не добраться, если им присвоить атрибут `private`. Только через специальные методы, находящиеся там же, в коробочке (как мы увидим — это методы с именами `get` (получить данное) и `set` (установить данное в определенное значение)). То есть просто так изменить напрямую состояние объекта и тем самым, возможно, разрушить его не получится. Это все мы наблюдали, когда создавали простейший класс, правила построения которого включают в себя соблюдение принципа инкапсуляции.

Чем же, какими синтаксическими средствами языка обеспечивается соблюдение принципа инкапсуляции при создании класса? Это использование ключевых слов `public`, `private`, `protected` (с ними мы уже встре-

чались) и `internal`. Смысл последнего будет рассмотрен позже. Ранее мы уже обращали внимание на тот факт, что когда задаются поля класса (а именно они определяют своими значениями состояние в данный момент объекта класса), эти поля должны задаваться с атрибутом `private`, чтобы к ним извне не было доступа. В таком случае они доступны только для методов, определенных в данном классе.

Что касается общедоступных (имеющих атрибут `public`) данных, то такими данными могут быть общедоступные константы, другие поля, которые определены лишь для чтения. Последние имеют специальный атрибут `readonly`. Но чем отличаются константы от полей "только для чтения"? Дело в том, что константы не всегда соответствуют всем требованиям ситуации с реализацией алгоритма. Часто случается так, что переменную нужно получить в результате расчетов, а потом сделать ее "только для чтения". В C# именно для таких случаев предусмотрен тип переменных `readonly`. Переменные поля `readonly` имеют большую гибкость, нежели `const` (атрибут, с которым объявляется константа), потому что позволяют перед присваиванием производить различные вычисления значения, которое должно быть "только для чтения". Правило использования таких полей говорит, что вы можете присваивать им значение только в конструкторе и нигде более (лишь конструктор инициализирует поля, это его функция). Одной из основных особенностей таких полей является то, что они могут принадлежать и экземплярам класса, а не быть статическими, как константы (при объявлении константы атрибут `static` запрещен, т. к. по своей сути константа — уже сама по себе статический элемент). Это позволяет получать различные значения полей "только для чтения" в разных экземплярах классов. Но если вы хотите сделать поле `readonly` статическим, то должны явно объявить его таковым, в отличие от полей `const`.

Рассмотрим пример. Пусть в некотором классе `A` имеется метод вычисления количества поставщиков `supp()`. Объекты, получаемые из класса `A`, должны работать какое-то время с одним количеством поставщиков, а через некоторый промежуток — с другим количеством. То есть на определенном отрезке времени поле "Количество поставщиков", назовем его `NumSupp`, должно быть как бы константой.

В описанном примере код может выглядеть так:

```
public class A
{
  public static readonly uint NumSupp;
  static A()  // статический конструктор
```

```
  {
    NumSupp  =  supp (Data);
  }
}
```

В данном конкретном примере мы объявляем нашу переменную как статическую и используем ее в экземпляре класса при каждом запуске программы. Функция `supp(Data)` вычислит количество поставщиков на данную дату, и с помощью конструктора поле `NumSupp` получит заданное значение, с которым станут работать все экземпляры класса.

Мы говорили ранее, что управление приватными данными класса осуществляется двумя методами с именами `get` и `set`. Однако есть еще один способ управления этими данными: определение свойства. Рассмотрим инкапсуляцию с использованием методов `get` и `set`.

Инкапсуляция с использованием методов *get* и *set*

Построим класс `Employee`, моделирующий сотрудника некоторого предприятия. Сначала определим в нем поле "Имя сотрудника". Мы хотим, чтобы вне класса к этому полю не было доступа. То есть это поле должно иметь атрибут доступа `private`. Вид объявления в классе будет `private string empName;` (имя — строковое данное, т. е. данное типа `string`). Чтобы прочитать это данное из класса, надо создать метод, который имел бы право читать это данное и выдавал бы его значение. Начинаться он должен с части `Get` (получить). То есть его имя будет, например, `GetName()`. Какой атрибут доступа ему присвоить? Этот метод будет вызываться из исполняющей программы, а не в самом классе. Поэтому он должен быть известен в исполняющей программе, т. е. быть общедоступным, не спрятанным в классе. Значит, его атрибут доступа — `public`. Он должен возвращать имя сотрудника. Следовательно, тип возвращаемого значения у метода будет `string` (как и тип имени сотрудника). Окончательно метод чтения поля `empName` будет выглядеть так:

```
public string GetName()
{
  return empName;
}
```

Напомним, что `return` — это оператор возврата результата работы функции (в данном случае — метода класса). Поэтому в исполняющей программе обращение к методу можно было бы записать, например, так:

```
string s = GetName();
```

если бы все происходило не с членом класса. Но при работе с классом предварительно надо создать из класса объект по конкретному сотруднику (т. е. некоторой переменной `emp` присвоить тип класса `Employee`). Так что предыдущая запись на самом деле будет выглядеть следующим образом:

```
Employee emp = new Employee("Иванов И. И.", остальные аргументы конструктора);
string s = emp.GetName();
```

В `s` появится "Иванов И. И.".

Теперь надо создать метод, который изменяет поле из внешней программы. Назовем его `SetName()` (установить значение поля `Name`). Здесь надо подумать, какие действия следует предварительно выполнить в теле этого метода, прежде чем изменить поле. Мы возьмем самое простое действие: проверим значение, которое присвоится полю, на его длину. Пусть, например, имя работника не должно превышать девяти символов. Вот эту-то проверку и зададим в методе. Ясно, что метод должен иметь общедоступность, т. е. должен быть снабжен атрибутом `public`. Возвращать по оператору `return` ничего не требуется, поэтому метод имеет тип возвращаемого значения `void`. А у себя на входе метод должен иметь параметр типа `string`, т. к. он должен принимать некую строку текста, которой заменит значение поля в классе. Таким образом, метод может иметь вид:

```
public void SetName(string Name)
{
  if(Name.Length > 9)
    WriteLine("Ошибка: длина имени больше 9 ");
  else
    empName = Name;
}
```

Предполагается, что метод находится в классе, поэтому для него переменная `empName` доступна.

В итоге для задания одного поля класса `Employee` в соответствии с принципом инкапсуляции получим программу, представленную в листинге 8.2.

Листинг 8.2

```
/* Created by SharpDevelop.
 * User: user
 * Date: 27.11.2012
```

```
 * Time: 16:39
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app23_set_get
{
  class Employee
  {
    private string empName;   // поле
    // Конструктор: должен иметь атрибут public,
    // иначе его нельзя будет вызвать
    // из основной программы
    public Employee(string name)
    {
      empName=name;
    }
    // Методы класса:
    public string GetName()
    {
      return empName;
    }
    public int SetName(string Name)
    {
      if(Name.Length > 9)
      {
        Console.WriteLine("Ошибка: длина имени " +
                          "больше 9, имя = {0}", Name);
        return(0);
      }
      else
      {
        empName = Name;
        return(1);
      }
    } // if
  }  // конец класса

  class Program
  {
    public static void Main()
```

```
    {
      // Создаем объект из класса:
      Employee emp = new Employee("Иванов И.И.");
      // Достаем поле name из объекта (если бы поле
      // имело атрибут static, мы бы его достали
      // прямо из класса):
      string s=emp.GetName();
      Console.WriteLine("Поле name = {0}",s);

      Console.Write("Продолжить: <Enter> >");
      // Изменяем значение поля name:
      if(emp.SetName("Петров П.П.") == 0)
        emp.SetName("Петров П.");
      // Так, как выше, не делают, но это просто
      // для примера
      s=emp.GetName();
      Console.WriteLine("Поле name = {0}",s);
      Console.Write("Продолжить: <Enter> >");
      Console.ReadKey(true);
    }
  }
}
```

Результат работы программы представлен на рис. 8.3.

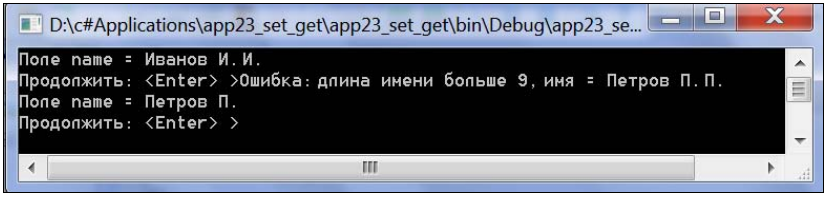


Рис. 8.3. Применение методов `get`, `set` для управления полем класса

Инкапсуляция с использованием свойств

В C# имеется другой, более удобный способ инкапсулирования данных — это использование так называемых свойств. *Свойства* — это упрощенное синтаксическое представление методов доступа к полю. Добавим к классу `Employee` еще одно поле: табельный номер работника. Определение свойств представлено в программе листинга 8.3.

Листинг 8.3

```
/* Created by SharpDevelop.
 * User: user
 * Date: 27.11.2012
 * Time: 18:12
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app24_properties
{
 class Employee
  {
    private string empName;
    private int empID;
    private float empPay;

    // Конструктор с полями
    public Employee(string Name, int ID, float Pay)
    {
      empName=Name; empID=ID; empPay=Pay;
    }

    // Конструктор со свойствами
    /*
      Чтобы конструкторы различались, параметр Pay взят
      другого типа. Но поле empPay объявлено типа float,
      поэтому понадобилось принудительное приведение
      параметра Pay к типу float. this взят из-за
      одинаковости имен параметров и свойств
    */
    public Employee(string Name,int ID, double Pay)
    {
      this.Name=Name; this.ID=ID; this.Pay=(float)Pay;
    }
    // Так определяются свойства Name, ID, Pay:
    public string Name
    {
      get { return empName; }
```

```
      set
      {
        if (value.Length > 9)
          Console.WriteLine ("Ошибка: длина имени > 9. "+
                             "Имя = {0}", value);
        else
          empName = value;
      } // set
    } // Name
    public int ID
    {
      get { return empID; }
      set { empID = value; }
    } // ID
    public float Pay
    {
      get {return empPay;}
      set { empPay = value;}
    }
  } // Employee
  class Program
  {
    public static void Main(string[] args)
    {
      Employee emp = new Employee("Иванов", 411, 15000);

      // Вывод полей объекта через вывод свойств:
      Console.WriteLine("empName = {0}",emp.Name);
      Console.WriteLine("empID = {0}",emp.ID);
      Console.WriteLine("empPay = {0}",emp.Pay);

      // emp.SetPay(emp.GetPay() + 1000);
              // Без свойств надо было бы так делать
      emp.Pay += 1000; // emp.Pay будет равно 16000
      Console.WriteLine("empPay после добавления 1000 к свойству
Pay  = {0}",emp.Pay);

      // Создание объекта из конструктора с помощью
      // задания свойств
      Console.WriteLine("Создание объекта через свойства:");
      Employee emp1 = new Employee("Петров",001,25000);
```

```
      // Вывод полей объекта через вывод свойств:
      Console.WriteLine("empName = {0}",emp1.Name);
      Console.WriteLine("empID = {0}",emp1.ID);
      Console.WriteLine("empPay = {0}",emp1.Pay);

      Console.Write("Press any key to continue... ");
      Console.Read();
    }
  }
}
```

Мы видим, что свойства имеют такие же имена, как и соответствующие поля, для обработки которых создаются свойства. Тип доступа к свойствам — `public`, т. к. это фактически методы, которые станут вызываться вне данного класса, поэтому и должны быть общедоступными. У свойства есть тело, ограниченное фигурными скобками, в которое помещены два специальных метода управления соответствующим свойству полем. У методов заданные в среде исполнения имена `get` (получить значение поля) и `set` (установить значение поля). Метод `get()` возвращает значение поля, метод `set()` присваивает полю значение переменной `value`, которое определяется в основной программе, исполняющей метод `set()`. При этом тип `value` всегда совпадает с типом свойства. Свойства хороши не только более короткой формой записи, но и также тем, что они могут участвовать в операциях внутри класса. Например, если в класс добавить поле `private float empPay` (зарплата), которому будет соответствовать свойство, определенное как

```
public float Pay
{
  get {return empPay;}
  set { empPay = value;}
}
```

(компилятор потом преобразует эту короткую запись в обыкновенные функции-методы), то если потребуется увеличить зарплату, скажем, на 1000, то при отсутствии аппарата свойств надо было бы писать операторы:

```
Employee emp = new Employee("Иванов");
emp.SetPay(emp.GetPay() + 1000);
```

а с использованием свойства `Pay` можно записать

```
emp.Pay += 1000;
```

Намного проще.

Как доставать значения полей, если мы ввели понятия свойств? А мы уже показали частично в операторе

```
emp.Pay += 1000;
```

То есть вместо имени поля после точки надо поставить имя свойства. В данном случае компилятором будет построено обращение к методу `get()`, который выведет нас на поле. А обращение к полю для вывода его содержимого (раньше мы обращались через метод `GetИмяПоля()`) будет таким: `emp.Pay`. Замечательно! С вводом понятия свойств мы получили возможность в основных программах, использующих классы, даже не догадываться, что существуют какие-то методы `get` и `set`, хотя раньше в основной программе, как мы видели, надо было писать обращение к методам `SetName()`, `GetName()` и т. п. И еще одна возможность появилась после ввода понятия свойств: в конструкторе теперь можно не писать присвоение параметров полям класса, а сразу — свойствам. Если раньше мы писали, например, конструктор в виде:

```
public Employee(string name)
{
  empName=name;
}
```

то теперь можно писать в виде:

```
public Employee(string name)
{
  Name=name;
}
```

где `Name` — это имя свойства, связанного с полем `empName`, например. В конструкторе же можно задавать операторы, проверяющие на достоверность вводимые данные (это и раньше никто не мешал делать). Рабо-

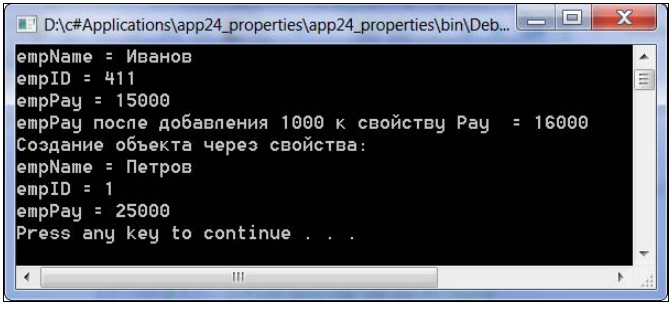


Рис. 8.4. Работа со свойствами класса

та со свойствами отражена в программе листинга 8.3. Результат представлен на рис. 8.4.

О доступности и статичности свойств

Когда при определении свойства заданы оба метода `get`, `set`, то такое свойство доступно как для чтения, так и для записи (модификации). Если какой-то из методов не указать, то свойство станет либо доступным только для чтения (присутствует только метод `get`), либо только для записи (присутствует только метод `set`).

Если поле класса имеет атрибут `static`, то вполне естественно, что и свойство для этого поля надо объявлять с таким же атрибутом.

Автоматические свойства

При определении свойства мы видели, что оно задается парой методов `get`-`set`, в телах которых (в фигурных скобках) только простейшие операторы — возврата и присвоения. В большинстве случаев описание `get`-`set` этим и ограничивается. Если в классе много свойств, то записывать почти одно и то же содержимое в фигурных скобках становится утомительным. Поэтому разработчики языка придумали, как обойти эту проблему. Они дали возможность отказаться от записи полей, а потом на их основе — записи свойств, а ввели возможность сразу определять свойства без описания полей (фактически была двойная работа), и свойства писать покороче.

Вот как это делается. Воспользуемся данными программы из листинга 8.3. Мы описывали поле `empPay` и свойство к нему `Pay` таким образом:

```
private  float empPay;  // поле
public float Pay        // свойство
{
  get { return empPay; }
  set { empPay = value; }
}
```

Теперь поле не определяют, а пишут сразу вид свойства:

```
public float Pay {get; set;}
```

(без точки с запятой в конце). Дальше во всем этом разбирается компилятор, который автоматически создает свойство таким, каким ему положено быть из этой синтаксической конструкции, и обеспечивает работу программиста только со свойством. Полей нет! Таким образом определенные свойства называют *автоматическими*.

Инициализация объекта

Мы знаем, что инициализация объекта происходит через конструктор, в котором при создании объекта задаются конкретные значения параметров. Существует еще один способ инициализации, синтаксис которого — список значений общедоступных полей и/или свойств (свойства по определению общедоступны), заключенный в фигурные скобки. Например, имеем класс `p`, описанный как

```
class p
{
  // Свойства (автоматические)
  public int x {get; set;}
  public int y {get; set;}

  // Конструкторы
  public p()
  {
  }
  public p(int v1, int v2)
  {
    x=v1;
    y=v2;
  }
}
```

С помощью этих конструкторов можно создать объекты обычным способом:

```
p ob1=new p(); ob1.x=10; ob1.y=20;
```

или

```
p ob2=new p(10,20);
```

А с помощью синтаксиса инициализации можно написать:

```
p ob=new p {x=10, y=20};
```

При таком синтаксисе конструктор класса — конструктор по умолчанию — вызывается неявно. Но можно его вызывать и явно, записав

```
p ob=new p() {x=10, y=20};
```

При инициализации объекта таким образом можно вызывать не только конструктор по умолчанию, но и любой другой определенный в классе. Например, в нашем случае можно записать:

```
p ob2=new p(10,20) {x=30, y=40};
```

Свойства класса получат, естественно, последние значения (30, 40).

Применение синтаксиса инициализации объектов более наглядно и удобно при большом количестве свойств, полей и конструкторов в классе. В этом случае не надо думать, какой конструктор выбрать для данного объекта: бери конструктор по умолчанию и в фигурных скобках пиши названия нужных свойств и/или полей и присваиваемые им значения. Программа по примерам инициализации приведена в листинге 8.4. Результат ее работы представлен на рис. 8.5.

Листинг 8.4

```
/* Created by SharpDevelop.
 * User: user
 * Date: 30.11.2012
 * Time: 13:27
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app25_obj_ini
{
  class p
  {
    // Свойства (автоматические)
    public int x {get; set;}
    public int y {get; set;}

    // Конструкторы
    public p()
    {
    }
    public p(int v1, int v2)
    {
      x=v1;
      y=v2;
    }
  }
  class Program
  {
    public static void Main()
    {
      p ob1=new p(); ob1.x=10; ob1.y=20;
```

```
        Console.WriteLine("Инициализация свойств через "+
                "конструктор по умолчанию: x={0} y={1}",
                ob1.x, ob1.y);
        p ob2=new p(10,20);
        Console.WriteLine("Инициализация свойств через " +
               "двухаргументный конструктор: x={0} y={1}",
               ob2.x, ob2.y);
        p ob=new p {x=10, y=20};
        Console.WriteLine("Инициализация свойств через "+
               "синтаксис инициализации: x={0} y={1}",
               ob.x, ob.y);
        Console.Write("Press any key to continue... ");
        Console.Read();
    }
  }
}
```

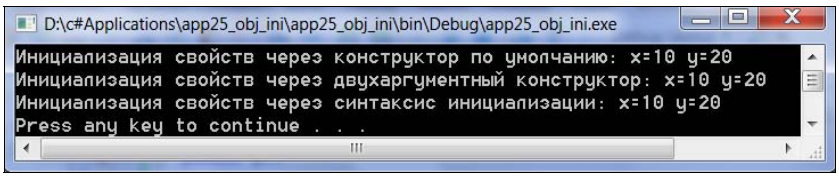


Рис. 8.5. Различные виды инициализации объекта

Организация работ при описании класса. Атрибут *partial*

При разработке, а особенно при эксплуатации класса (когда он участвует в эксплуатации какого-то приложения), существуют участки описания класса, которые относительно постоянны. Например, такие элементы класса, как описание полей, свойства и конструкторы, довольно постоянны, чего нельзя сказать о методах, которые могут довольно часто изменяться, особенно во время эксплуатации. В этих условиях имеет смысл большие классы разбивать на части, присваивая частям атрибут `partial` (частичный). Тогда можно отдельно заниматься только той частью, которая требует модификации, не трогая остальные части. При компиляции компилятор соберет все частичные классы (так они станут называться) в один общий класс. Вот как можно создавать частичные классы: разносят общий (длинный) файл с расширением cs на несколько файлов с таким же расширением и именами, помещая в них необходи-

мые элементы (конструкторы, методы, поля свойства). Каждому файлу добавляют описатель `partial`. Потом эти файлы подключают к общему проекту (создают проект с программой `Main()`). Файлы с расширением cs формируют с помощью программы WordPad (при записи текстового файла дают ему расширение cs). Объединяют эти файлы в один, зайдя в меню среды программирования: **Project | Add | Existing Item**. Откроется диалоговое окно для поиска файлов. Надо найти ранее сформированные cs-файлы и подключить. Пример программы с частичными классами представлен в листингах 8.5—8.7. Результат приведен на рис. 8.6.

Листинг 8.5

```
// Первый частичный класс

partial class Employee
{
  // Поля класса
  private string empName;    // имя
  private int empID;         // табельный номер
  private float currPay;     // зарплата
  private int empAge;        // возраст
  private string empSSN;     // номер медицинского полиса
  private static string companyName; // название компании
  // Конструкторы
  public Employee() { }
  public Employee(string name, int age, int id,
                  float pay, string ssn)
  {
    // Инициализация свойств
    Name = name;
    Age = age;
    ID = id;
    Pay = pay;
    SocialSecurityNumber = ssn;
  }
  static Employee()  // Статический конструктор
  {
    companyName = "Union Fenosa";
  }
}
```

Листинг 8.6

```
// Второй частичный класс
partial class Employee
{
  // Свойства
  public static string Company
  {
    get { return companyName; }
    set { companyName = value; }
  }
  public string Name
  {
    get { return empName; }
    set
    {
      if (value.Length > 9)
       ;  // Console.WriteLine("Имя больше 9 символов");
      else
        empName = value;
    }
  }
  public int ID
  {
    get { return empID; }
    set { empID = value; }
  }
  public float Pay
  {
    get { return currPay; }
    set { currPay = value; }
  }
  public int Age
  {
    get { return empAge; }
    set { empAge = value; }
  }
  public string SocialSecurityNumber
  {
    get { return empSSN; }
    set { empSSN = value; }
  }
} // class
```

Листинг 8.7

```
/* Основная программа
 * Created by SharpDevelop.
 * User: user
 * Date: 30.11.2012
 * Time: 15:43
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app22_employee_all
{
  class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Собранная из частичных файлов "+
                        "программа");

      Employee emp = new Employee {Name="Иванов"};
        // Инициализация свойства через синтаксис
        // инициализации
      Console.WriteLine("Сотрудник компании: {0}",
                        emp.Name);
      Console.Write("Press any key to continue... ");
      Console.ReadKey(true);
    }
  }
}
```

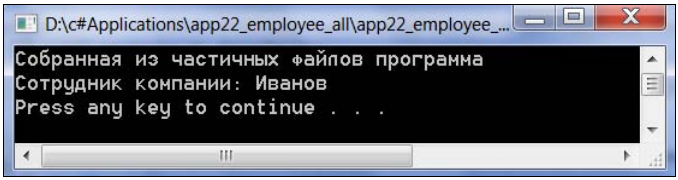


Рис. 8.6. Результат работы программы, собранной из частичных классов

Наследование

Наследование — это возможность использования существующих классов для создания новых классов. Новые классы могут создаваться на основе существующих классов, наследуя их функциональность, т. е. то, что умеют делать старые классы. При этом старые классы называют *родителями* или *базовыми классами*, а новые — *потомками* или *дочерними классами* (почему не сыновьими — не понятно), или производными классами. Новые классы фактически расширяют функциональность старых, потому что умеют делать то, что умеют старые, но и еще кое-что, чего нет в старых.

При наследовании в дочернем классе становятся доступными все члены родительского класса с атрибутом `public` или `protected`.

Зададим базовый класс в виде, показанном в листинге 8.8.

Листинг 8.8

```
class car           // Автомобиль
{
  const int maxSpeed = 90; // Макс. скорость автомобиля
  int carSpeed;            // Поле
  public  int Speed        // Свойство (должно быть
                           // общедоступным)
  {
    get {return carSpeed;}
    set
    {
      carSpeed = value; // Одна из форм синтаксиса
                        // задания свойства
      if(carSpeed > maxSpeed) // Контроль задания скорости
                              // автомобиля пользователем
        carSpeed = maxSpeed;
    }
  } // Speed
} // car
```

Здесь задан класс `car` (автомобиль) всего с одним полем — скорость автомобиля (`carSpeed`). На его основе создано свойство `Speed`, в котором в методе установки свойства (`set`) проверяется, не введет ли пользователь недопустимую величину максимальной скорости, заданной константой `maxSpeed`. Если такое произойдет, то метод `get` возьмет за вве-

денную скорость величину максимальной скорости. Допустим, вы хотите построить другой класс автомобиля (`My_car`) и такой, у которого были бы те же поля и свойства, что и у `car`, но чтобы новый класс еще содержал в себе марку автомобиля. Пусть это новое поле будет называться `carName`. Ясно, что нелогично снова создавать класс с теми данными, что уже разработаны, и добавлять в него новые данные. Лучше бы было воспользоваться уже существующим, а затем добавить свои. То есть желательно унаследовать уже готовую функциональность и добавить свою. Это и позволяет делать принцип наследования, заложенный в C#. Точнее, в его компилятор. В нашем случае мы можем спокойно написать синтаксическую конструкцию, представленную в листинге 8.9.

Листинг 8.9

```
class My_car : car
{
  const int maxName = 20;
  string carName;      // Поле — марка автомобиля
  public string Name   // Свойство (должно быть
                       // общедоступным)
  {
    get {return carName;}
    set
    {
      carName = value;
      if(carName.Length > maxName)
        carName="Ошибка: марка должна иметь " +
                "не более 20-ти символов";
    }
  }
```

Конструкция похожа на ту, что показана в листинге 8.8, поэтому поясним только ее первую строку: так задается класс-наследник — через двоеточие от его имени пишется класс-родитель.

Сведем теперь данные листингов 8.8 и 8.9 в одну программу, добавив к этому метод `Main()`, чтобы проверить, работает ли вновь созданный класс `My_car`. Мы должны убедиться, можно ли из этого класса достать свойство `Speed` из класса-родителя, можно ли изменять его в дочернем классе и работать с маркой автомобиля в дочернем классе. Вся программа полностью приведена в листинге 8.10, а результат ее работы показан на рис. 8.7.

Листинг 8.10

```
/* Created by SharpDevelop.
 * User: user
 * Date: 02.12.2012
 * Time: 12:42
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app27_inherit
{
  class car                  // Автомобиль
  {
    const int maxSpeed = 90; // Макс. скорость автомобиля
    int carSpeed;            // Поле
    public int Speed         // Свойство
    {
      get {return carSpeed;}
      set
      {
        carSpeed = value;          // Одна из форм
                                   // Задания свойства
        if(carSpeed > maxSpeed)    // Контроль задания
                 // скорости автомобиля пользователем
          carSpeed = maxSpeed;
      }
    } // Speed
  } // car

  class My_car : car
  {
    // Здесь нет конструктора: он берется по умолчанию
    const int maxName = 20;
    string carName;      // Поле — марка автомобиля
    public string Name   // Свойство
    {
      get {return carName;}
      set
      {
        carName = value;
```

```
        if(carName.Length > maxName)
          carName="Ошибка: марка должна иметь " +
                  "не более 20-ти символов";
      }
    }
  }

  class Program
  {
    public static void Main()
    {
      Console.WriteLine("Данные по классу-наследнику:");

      car obj_car = new car {Speed = 85}; // Инициализация
                              // объекта — базового класса
      Console.WriteLine("Установленная скорость в " +
                "родительском классе: {0}",
                obj_car.Speed);

      My_car cr = new My_car {Name = "Volvo"};
                      // Инициализация объекта
      cr.Speed=88;
      Console.WriteLine("Марка автомобиля: " +
              "{0}\nСкорость в дочернем классе: {1}",
               cr.Name, cr.Speed);

      Console.Write("Press any key to continue... ");
      Console.Read();
    }
  }
}
```

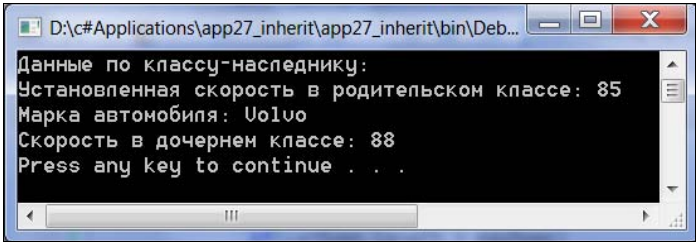


Рис. 8.7. Результаты работы с наследуемыми и родительскими свойствами

Во-первых, надо не забывать, что когда вы создаете свойства, те должны иметь атрибут `public`, т. е. быть общедоступными. Если атрибут `public` не указан, компилятор такому члену класса автоматически присваивает атрибут `private`. Но если свойства — автоматические, тогда им присваивается `public` по умолчанию. В нашем случае свойства объявлены не автоматические, а традиционным способом, правда, с усовершенствованным синтаксисом.

Во-вторых, когда создается объект производного класса, все его члены должны получить значения, в том числе, естественно, и те, что перейдут от родителя, через конструктор или путем доступа напрямую (см. `cr.Speed=88;`).

Запрет на наследование

В C# существует специальное ключевое слово. Если им пометить класс, то от этого класса нельзя будет наследовать. Это слово — `sealed` (запечатанный). Если вы объявили `sealed class car {члены класса}`, то при попытке компиляции конструкции `class My_car : car {члены класса}` компилятор выдаст ошибку. Например, многие системные классы запечатаны, т. е. не позволяют себя расширять за счет наследования. Конструкция

```
My_string : strring {}
```

выдаст ошибку компиляции, т. к. класс `string` запечатан: пользоваться его членами — пожалуйста, но свои не добавляйте.

Конструкторы и наследование

В иерархии классов допускается, чтобы у базовых и производных классов были собственные конструкторы. В связи с этим возникает следующий резонный вопрос: какой конструктор отвечает за построение объекта производного класса: конструктор базового класса, конструктор производного класса или же оба? Оказывается, что конструктор базового класса конструирует базовую часть объекта, а конструктор производного класса — производную часть этого объекта. И в этом есть своя логика, поскольку базовому классу неизвестны и недоступны любые элементы производного класса, а значит, их конструирование должно происходить раздельно.

Если конструктор определен только в производном классе (так называемый специализированный конструктор, т. е. с параметрами), то все происходит очень просто: конструируется объект производного класса,

а базовая часть объекта автоматически конструируется его конструктором, используемым по умолчанию (в базовом классе конструктора нет, как мы предположили, следовательно, берется конструктор по умолчанию). Когда конструкторы определяются как в базовом, так и в производном классе, процесс построения объекта несколько усложняется, поскольку должны выполняться конструкторы обоих классов. А в базовом классе может быть много конструкторов. Какой надо выполнять в данном конкретном случае? Тут приходится использовать ключевое слово `base`, которое находит нужный конструктор базового класса и выполняет его. Существует форма объявления конструктора производного класса, с помощью которого может быть вызван конструктор, определенный в его базовом классе:

```
конструктор производного_класса(список_параметров) : base
(список_параметров)
{
  // тело конструктора
}
```

где `base (список_параметров)` — это параметры, необходимые некоему конструктору в базовом классе для его запуска. Какой конструктор именно должен запускаться, компилятор определяет по количеству и типу параметров, заданных в методе (а это именно метод) `base`. После этого инициализированные поля станут известны в производном объекте.

Пример программы приведен в листинге 8.11. Результат работы представлен на рис. 8.8.

Листинг 8.11

```
/* Created by SharpDevelop.
 * User: user
 * Date: 02.12.2012
 * Time: 16:42
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app28_base
{
  class MyClass
  {
    public int x, y, z;  // Поля
```

```
  // Конструктор базового класса
  public MyClass(int x, int y, int z)
  {
    /* this применен, т. к. имена полей и имена
       параметров в конструкторе совпадают.
       this.x означает, что x относится к полю x
    */
    this.x = x;
    this.y = y;
    this.z = z;
  }
}

class inherite_class : MyClass
{
  int point;  // поле

  // Конструктор производного класса с применением base:
  // в базовый класс конструктору передаются
  // аргументы x, y, z, т. е. первый параметр
  // конструктора участвует в вычислении поля
  // наследника, остальные три — в вычислении
  // полей родителя
  public inherite_class(int point, int x, int y, int z)
    : base(x, y, z)
  {
    this.point = point; // Инициализирует поле
                        // наследника
  }
  // Метод класса-наследника
  public void Pointer(inherite_class new_point)
  {
    // Установка полей базового класса в наследнике:
    new_point.x += new_point.point;
    new_point.y += new_point.point;
    new_point.z += new_point.point;
    Console.WriteLine("Новые координаты объекта " +
                "в производном классе: {0} {1} {2}",
                new_point.x, new_point.y, new_point.z);
  }
}
```

```
    class Program
  {
    static void Main()
    {
      Console.WriteLine("Работа с ключевым " +
                        "словом base\n");

      inherite_class obj = new inherite_class(5, 2, 3, 4);
      Console.WriteLine("Координаты объекта в базовом " +
              "классе: {0} {1} {2}", obj.x, obj.y, obj.z);
      obj.Pointer(obj);
      Console.ReadLine();
    }
  }
}
```

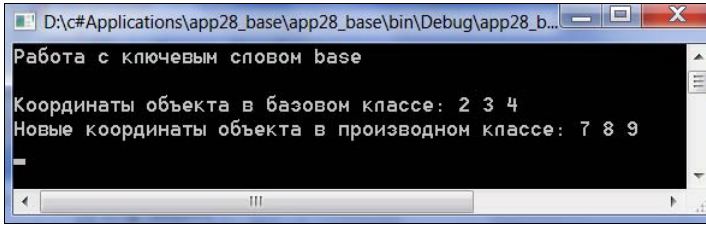


Рис. 8.8. Работа с ключевым словом `base`

Повторим, что с помощью ключевого слова `base` можно вызвать конструктор любой формы, определенной в базовом классе, причем выполняться будет лишь тот конструктор, параметры которого соответствуют переданным аргументам.

А теперь рассмотрим вкратце основные принципы действия ключевого слова `base`. Когда в производном классе указывается ключевое слово `base`, вызывается конструктор из его непосредственного базового класса. Следовательно, ключевое слово `base` всегда обращается к базовому классу, стоящему в иерархии непосредственно над вызывающим классом. Аргументы передаются базовому конструктору в качестве аргументов метода `base()`. Если же ключевое слово отсутствует, то автоматически вызывается конструктор, используемый в базовом классе по умолчанию.

Добавление к классу запечатанного класса

Мы видели, что запечатанный класс не наследуется. А хотелось бы в некоторых ситуациях все-таки как-то использовать его члены в своем создаваемом классе. Как это сделать? Повторяю, наследование не проходит. Для решения этой проблемы применяется прием включения в создаваемый класс объекта, получаемого из запечатанного класса. Программа, реализующая сказанное, приведена в листинге 8.12, а результат ее работы показан на рис. 8.9.

Листинг 8.12

```
/* Created by SharpDevelop.
 * User: user
 * Date: 02.12.2012
 * Time: 19:03
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app29_include_sealed
{
  sealed public class A
  {
    int a; // Поля
    int b;

    // Свойства
    public int A_a
    {
      get { return a; }
      set { a=value; }
    }
    public int A_b
    {
      get { return b; }
      set { b=value; }
    }
    // Конструктор
    public A(int a,int b)
    {this.a=a; this.b=b;}
```

```
    // Метод
    public int  M_A()
    {
      return (a + b);
    }
  }

  class B
  {
    public int c;
    public A ab = new A(15,20);
    public B(int cc)
    {
      c=cc;
    }
  }
  class Program
  {
    public static void Main()
    {
      B ba = new B(5);
      int sum=ba.c + ba.ab.M_A();

      Console.WriteLine("Использование запечатанного " +
                   "класса\nкак объекта создаваемого");

      Console.WriteLine("Поле с класса B = {0}", ba.c);
      Console.WriteLine("Поля запечатанного класса A " +
                "равны {0}, {1}",ba.ab.A_a, ba.ab.A_b);
      Console.WriteLine("Результат суммирования поля " +
                "из B и работы метода из A: {0}", sum);

      // TODO: Implement Functionality Here

      Console.Write("Press any key to continue... ");
      Console.Read();
    }
  }
}
```

Итак, имеется запечатанный класс `A` с двумя полями, доступ к которым осуществляется только через два соответствующих свойства — `A-a` и

A_b. В классе A есть метод M_A(), суммирующий значения закрытых полей класса A. Мы строим класс B с одним полем c и объектом ab, получаемым из запечатанного класса A. Вывод данных на экран показывает, что в классе B используется функциональность класса A (в частности, метод M_A()), хотя наследования не было, т. к. оно невозможно из-за запечатанности A. Заметим, что если объект одного класса вложен в объект другого класса, то доступ к элементам вложенного объекта идет через точку от имени объекта (это общее правило), а само имя объекта, которое определяет членство объекта в другом классе, идет через точку от имени объекта основного класса. В нашем случае, чтобы добраться до свойства A-a из запечатанного класса, надо было записать ba.ab.A_a.

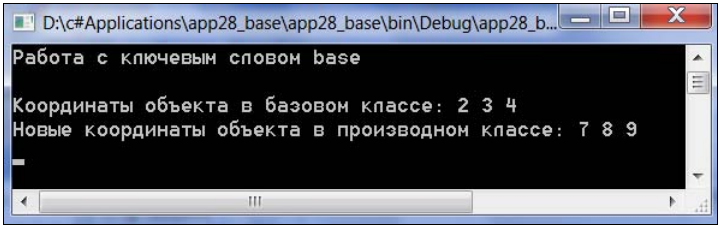


Рис. 8.9. Результат использования запечатанного класса в создаваемом классе

Вложенность классов

Рассмотренный выше пример является примером вложенности типов: в C# допускается вложенность типов. (В данном случае классов как типов. Про остальные подобные типы мы пока что ничего не знаем.) При этом вложенный тип считается обычным членом охватывающего или внешнего класса. Таким вложенным типом можно манипулировать, как и любым другим членом класса. Синтаксис таков:

```
public class OuterClass
{
  public  class InnerClass1 {}
  private class InnerClass2 {}
}
```

Если внутренний класс станет использоваться вне класса, в котором он определен, его, естественно, надо будет указывать с именем включающего его класса (мы это видели в приведенном ранее примере на уровне объектов). Например, можем записать фрагмент основной программы:

```
static void Main()
{
  OuterClass.InnerClass1 obj1 =
      new OuterClass.InnerClass1();
}
```

Как выйти правильно на элементы такого объекта (даже если вложенность более глубокая, что тоже допускается), вам сообщит подсказчик среды исполнения (в нашем случае SharpDevelop): как только вы наберете точку после имени созданного объекта (в нашем случае после obj1), подсказчик откроет окно с перечнем элементов, из которых предстоит выбрать нужный. Если вы его в этом списке найдете, щелкните на нем, и он приклеится к набранному имени объекта. Поставьте точку после найденного имени. Откроется новый список. И т. д. Если вы не найдете в списке нужного вам имени (имени подобъекта, имени члена подобъекта и т. д.), то участок вашей программы, в котором вы находитесь, не видит нужного вам элемента. Причина может быть в том, что нужный вам элемент не имеет атрибута public. Или где-то вкралась синтаксическая ошибка, которая мешает увидеть элемент. Тут много вариантов. Главное — следить, чтобы в списке обязательно находилось имя нужного вам элемента класса. Если это не так, стоит разобраться, почему.

Полиморфизм

Слово это означает "многоформие". Когда в классе-потомке после наследования его от базового класса появляются методы базового класса, бывает, что возникает необходимость эти базовые методы переделать под потребность производного класса. Классический пример: классы "Многоугольники" и "Окружности". Оба унаследованы от базового класса Object, в котором определен метод Draw() — рисовать фигуру. Но в первом случае надо рисовать многоугольник, а во втором — окружность. А в обоих классах находится метод с одним и тем же именем Draw(). Принцип полиморфизма обеспечивает возможность разной "начинки" метода Draw(), оставляя нетронутым имя метода. То есть фактически старый метод базового класса в обоих его потомках имеет возможность получить новое содержание (ясно, что все это может делать компилятор). Скажете, мол, а при чем тут форма? Здесь не надо путать русские философские понятия содержания и формы, а вспомнить, что слово "формировать" по-гречески будет звучать как "морфи". С этой точки зрения старый метод Draw() получит в своих потомках две разные "формы". Отсюда полиморфизм: много форм.

Процесс переделки старого метода в новые носит название *переопределения метода* (method overriding). Для возможности переопределения надо дать какую-то информацию компилятору на сей счет. А информация эта такая: если вы в базовом классе хотите разрешить переопределять некоторый метод в подклассах, то этот метод надо снабдить атрибутом `virtual`. Следовательно, т. к. мы знаем, что метод `Draw()` рисует разные фигуры, можем предположить, что в классе-родителе он помечен атрибутом `virtual`. Такие методы называют *виртуальными*. Действительно, когда в базовом классе вы помечаете некий метод атрибутом `virtual`, вы тем самым даете возможность в будущем этот метод переопределить в некоем классе-потомке со своей функциональностью. То есть в момент присвоения вами атрибута `virtual` методу, других методов пока не существует, иными словами, они как бы существуют, но не по-настоящему, т. е. виртуально. Кроме того, метод, помеченный ключевым словом `virtual`, дает возможность применять этот метод по умолчанию, которая распространяется на всех потомков базового класса. Если дочерний класс решит, он переопределит такой метод, но он не обязан это делать. Может просто вызвать этот метод из базового класса как метод по умолчанию.

Из базового класса вы дали сигнал компилятору, что такой-то метод (пусть это будет `Draw()`) можно переопределять в будущем классе-потомке. А как потомок даст сигнал компилятору, что он переопределяет виртуальный метод? Потомок должен присвоить этому методу атрибут `override`. Все. Цепочка замкнулась. Значит в классе "Многоугольники" или в классе "Окружности" метод `Draw()` имеет атрибут `override`.

Если вас устраивает функциональность виртуального метода базового класса, и вы хотите еще добавить свои какие-то операторы, то чтобы не писать все тело старого метода, есть возможность воспользоваться ключевым словом `base`. Тогда, например, обращение к методу `Draw()` внутри его переопределения будет выглядеть как `base.Draw();`, т. е. он нарисует фигуру заложенными в него операторами в классе-родителе, а потом дальше в переопределяемом теле вы можете записать какие-то свои операторы, которые, например, нарисованную окружность разобьют на три сектора.

Работа по переопределению метода показана в программе листинга 8.13. Результат работы представлен на рис. 8.10.

Листинг 8.13

```
/* Created by SharpDevelop.
 * User: user
 * Date: 04.12.2012
```

```
 * Time: 12:30
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app30_overriding
{
  public class A
  {
    int a; // Поля
    int b;

    // Свойства
    public int A_a
    {
      get {return a;}
      set {a=value;}
    }

    public int A_b
    {
      get {return b;}
      set {b=value;}
    }

    // Конструкторы
    public A() {}
    public A(int a,int b)
    { this.a=a; this.b=b; }

    // Виртуальный метод
    public virtual int  M_A()
    { return (a + b); }
  }

  class B : A
  { int a;
    int b;
    int c;
```

```
    // Свойство
    public int C_c
    {
      get {return c;}
      set {c=value;}
    }
    // Конструктор (указывается, какой конструктор
    // базового класса будет вызван)
    public B(int aa, int bb, int cc):base(aa, bb)
    {
      a=aa; b=bb; c=cc;
    }
    // Переопределение метода базового класса: он должен
    // поле "c" возводить в квадрат  и добавлять результат
    // к результату старого метода (из класса A)
    public override  int  M_A()
    {
      return (base.M_A() + c*c);
    }
  }
  class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Результат работы " +
                        "переопределенного метода");
      B b_b = new B(5,6,7);
      Console.WriteLine("1-е поле в классе-родителе — {0}\n2-е
поле в классе-родителе — {1}", b_b.A_a, b_b.A_b);
      Console.WriteLine("Значение поля производного класса —
{0}\nРезультат переопределенного метода — {1}",b_b.C_c,b_b.M_A());

      Console.Write("Press any key to continue... ");
      Console.ReadKey(true);
    }
  }
}
```

Все пояснения — по тексту программы.

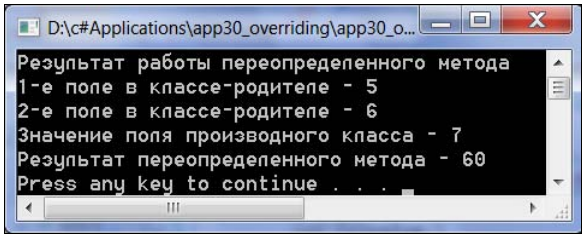


Рис. 8.10. Результат работы переопределенного в классе-потомке виртуального метода

Абстрактные классы

Допустим, вы разработали некий класс и не хотите, чтобы этим классом впоследствии пользовались непосредственно, т. е. создавали из него экземпляры (объекты), с которыми бы работали. Одна из причин — довольно общий характер его членов, который можно уточнять только на этапе разных подклассов, полученных путем наследования данного класса. Чтобы добиться запрета создания экземпляров, надо сообщить компилятору об этом. Это происходит путем придания классу атрибута `abstract`. Попытка создать экземпляр такого абстрактного класса приведет к ошибке компиляции. Но наследовать этот класс можно и нужно (иначе, для чего же он создавался?).

Вспомним, что при наследовании методы базового класса, помеченные ключевым словом `virtual`, могут переопределяться (но не обязательно) в дочернем классе. Если они переопределяются, то используется их новая функциональность, если нет — то функциональность базового класса. То есть метод в таких случаях может использоваться по умолчанию. В абстрактных классах виртуальные методы задаются лишь в том смысле, чтобы можно было просто к ним обратиться и в них что-то выполнилось. Например, рассмотренный нами ранее метод рисования `Draw()` в абстрактном классе может просто в своем теле выдать какое-то сообщение. Например, "Здравствуйте! Я — метод рисования из абстрактного класса". И все. Но наследующий класс может и не переопределять такой метод: не обязан. Просто разработчик что-то забыл и получает в момент отладки подобное "Здравствуйте" и не поймет в чем дело. То есть могут возникнуть неприятности еще на этапе разработки. Чтобы избежать подобных ситуаций, есть способ заставить разработчика переопределять метод класса-родителя в потомке. Для этого такой метод в родителе помечают ключевым словом `abstract` (как и абстрактный класс). Тут уже компилятор не пропустит эту ситуацию и напомнит разработчику

выдачей соответствующей ошибки компиляции, что такой-то метод следует переопределить в классе-потомке. Методы с ключевым словом `abstract` могут определяться только в абстрактных классах. Вот пример:

```
abstract class Shape
{
  ...
  public abstract  void Draw()
  {
    ...
  }
}
```

Сокрытие членов класса

Допустим, вы пользуетесь неким классом "втемную": получили его от некоего поставщика. Поэтому может наступить ситуация, когда вы в своем производном классе дали имя какой-то переменной, которое случайно совпало с именем, объявленным в используемом родительском (которым вы пользуетесь в данный момент) классе. В этом случае компилятор выдаст сообщение, что он прячет такую переменную в родительском классе (она вам становится недоступна при наследовании) и предложит либо дать своей совпавшей по имени переменной атрибут `override` (вполне естественно, как мы видели при переопределении методов) или добавить к ее объявлению ключевое слово `new` (видимо, даст ей свое другое имя). То есть, если вам нечего переопределять, добавьте `new`. Тогда ваша переменная может выглядеть так, например:

```
public new int indecs;
```

Здесь предполагается, что переменная с именем `indecs` была объявлена в классе, члены которого вы пытаетесь наследовать. Так заявляют некоторые авторы — специалисты по C#. Но жизнь, повторю, на месте не стоит, и компиляторы совершенствуются. Если взять программу листинга 8.13 и убрать в ней в определении метода `M_A()` ключевое слово `override`, а затем скомпилировать, компилятор действительно выдаст ошибку о том, что он прячет `M_A()`. Но ошибка выдается предупреждающая, а не фатальная. Это значит, что скомпилированную программу можно запускать на выполнение. Если это проделать, то окажется, что результат получим точно такой же, как показано на рис. 8.10. Похоже, что компилятор сам исправляет ситуацию. Что касается одинаковости имен переменных-неметодов, то тут никакого криминала компилятор не выявляет.

Приведение классов к базовому и производному

Тернарный условный оператор

Изучив принцип наследования, мы можем строить иерархию классов (расположение классов в порядке от высшего (родителя) к низшему (потомку)). Заметим, что все классы происходят от одного прародителя — абстрактного класса `Object`, в котором сосредоточены наиболее общие элементы, требующиеся для создания классов. Если при создании класса компилятор не находит родителя, он автоматически создает класс из класса `Object`. Ключевым словом типа этого класса является `object`. Если у нас есть некая иерархия классов, то язык C# позволяет тип класса-потомка связывать с типом класса-родителя (это так называемое *неявное приведение классов*) и, наоборот, тип класса-родителя связывать с типом класса-потомка *(явное приведение классов)*. В листинге 8.14 представлен пример программы, в которой показаны оба вида преобразования. Результат работы программы приведен на рис. 8.11.

Листинг 8.14

```
/* Created by SharpDevelop.
 * User: user
 * Date: 05.12.2012
 * Time: 14:03
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app32_class_casting
{
  public class Base
  {
    public virtual void WhoAmI()
    {
      Console.WriteLine("I'm Base");
    }
  }
  public class Derived: Base
  {
```

```
    public override void WhoAmI()
    {
      Console.WriteLine("I'm Derived");
    }
  }
  public class Test
  {
    public static void Main()
    {
      Derived d = new Derived();
      Base b = d;    // Неявное преобразование ссылки
                     // в базовый тип
      b.WhoAmI();    // Проверка, что ссылка продолжает
                     // указывать на тот же объект
      Derived d1 = (Derived) b; // Восстановление ссылки
                  // на объект производного класса (явное
                  // преобразование)
      object o = d;    // Неявное преобразование ссылки
                       // на объект в тип object
      Derived d2 = (Derived) o; // Восстановление ссылки
                  // на объект производного класса (явное
                  // преобразование)
      d.WhoAmI(); // Проверка: восстановилась ли ссылка
                  // на объект
      Console.Read();
    }
  }
}
```

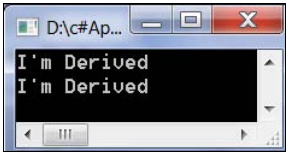


Рис. 8.11. Варианты неявного и явного приведения классов

Операторы *as* и *is*

Поскольку `object` является всеобщим базовым типом, любую ссылку на класс можно преобразовать в ссылку на `object`, а ссылку на `object` можно попытаться преобразовать в ссылку на любой класс. Слово "по-

пытаться" сказано не случайно, потому что явное преобразование (из потомка в родителя) проходит не всегда, т. к. потомок — это расширение родителя. Приведение "вниз" всегда выполняется явно. Существуют специальные операторы, которые определяют совместимость типов. Это операторы с ключевыми словами `as` и `is`. Они работают на совместимость не только таких сложных типов, как классы, но и для базовых типов, таких как, например, `int`. Как работает оператор `is`, видно из примера, приведенного в листинге 8.15.

Листинг 8.15

```
/* Created by SharpDevelop.
 * User: user
 * Date: 05.12.2012
 * Time: 15:04
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app33_as_and_is
{
  public class test
  {
    static void Main()
    {
      String derived_Obj = "Dummy"; // Производный объект
                        // класса String (из статических
                        // классов экземпляры не создаются)
      Object base_Obj1 = new Object (); // Базовый объект
      Object base_Obj2 = derived_Obj;   // Явное
                        // преобразование: запоминание
                        // ссылки в базовом объекте
      Console.WriteLine ("base_Obj2 {0} String",
        base_Obj2 is String ? "является" : "не является");
      Console.WriteLine ("base_Obj1 {0} String",
        base_Obj1 is String ? "является" : "не является");
      Console.WriteLine ("derived_Obj {0} Object",
      derived_Obj is Object ? "является" : "не является");
      int j = 123;
      object b = j;
```

```
      object obj = new Object () ;
      Console.WriteLine ("b {0} int",
              b is int ? "является" : "не является");
      Console.WriteLine ("obj {0} int",
              obj is int ? "является" : "не является");
      Console.WriteLine("b {0} System.ValueType",
             b is ValueType ? "является" : "не является");
      float f=12.3f;
      Console.WriteLine ("f {0} int",
              f is int ? "является" : "не является");
      Console.Read();
    }
  }
}
```

Результат работы программы представлен на рис. 8.12.

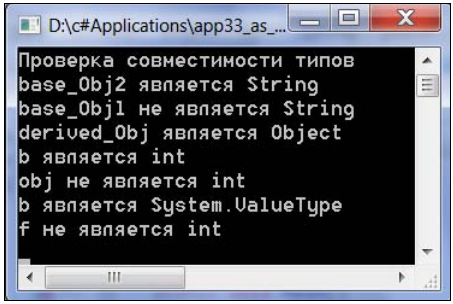


Рис. 8.12. Проверка работы оператора `is`

В программе создается экземпляр класса `String`. Это системный статический класс. Из таких классов экземпляры не создаются, поэтому переменной этого типа `derived_Obj` идет присвоение значения напрямую. Будем считать, что это — производный класс, т. к. все классы, в том числе и `String`, происходят от общего родителя — класса `Object`. Его и станем считать родителем. Его экземпляры — `base_Obj1` и `base_Obj2`. В последнем запоминается ссылка на производный класс `derived_Obj`. А далее следуют проверки с помощью оператора `is` на совместимость типов. Проверка осуществляется внутри оператора `Console.WriteLine()`, в котором в качестве выводимого на экран выражения служит выражение вида `base_Obj2 is String ? "является" : "не является"`. Такой

оператор мы еще не встречали. Его обозначение — вопросительный знак (?). Он называется *тернарным* (из трех частей) *условным оператором* в отличие от бинарного (из двух частей) условного оператора `if...else`. Общая форма такого оператора:

```
Выражение1 ? Выражение2 : Выражение3;
```

Здесь `Выражение1` должно относиться к типу `bool` (здесь и задается условие), а `Выражение2` и `Выражение3` — к одному и тому же типу. Обратите внимание на применение двоеточия и его местоположение в операторе `?`. Результат определяется следующим образом. Сначала вычисляется `Выражение1`. Если оно истинно (`True`), то вычисляется `Выражение2`, и полученный результат — результат всего оператора. Если же `Выражение1` оказывается ложным (значение — `False`), то вычисляется `Выражение3`, и его значение становится общим результатом.

У нас в программе записано

```
base_Obj2 is String ? "является" : "не является"
```

То есть `Выражение1` — это `base_Obj2 is String` (проверка: является ли `base_Obj2` типом `String`). Это выражение можно было бы в программе записать в виде

```
bool bl= base_Obj2 is String;
```

Тогда выражение

```
base_Obj2 is String ? "является" : "не является"
```

было бы таким:

```
bl ? "является" : "не является"
```

В итоге оператор

```
Console.WriteLine("base_Obj2 {0} String", bl ? "является" :
"не является");
```

работает так: выводится `base_Obj2` (то, что не относится к формату, выводится один к одному), затем обрабатывается формат, т. е. выбирается для вывода из списка аргументов первый (он у нас единственный). Это как раз тернарный условный оператор-выражение. Он вычисляется и выводится. После этого за форматом `{0}` снова идут неформатные символы (`String`). Они выводятся на экран без изменения.

Пример работы с оператором `is` показан в программе, приведенной в листинге 8.16. Результат ее работы представлен на рис. 8.13.

Листинг 8.16

```
/* Created by SharpDevelop.
 * User: user
 * Date: 05.12.2012
 * Time: 16:02
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace App33_as_and_is2
{
  public class BaseType {}
  public class DerivedType : BaseType {}
  public class Program
  {
    static void Main()
    {
      Console.WriteLine ("Проверка оператора as");
      DerivedType derived_Obj = new DerivedType ();
         // объект производного класса
      BaseType base_Objl = new BaseType ();  // Объект
                                  // базового класса
      BaseType base_Obj2 = derived_Obj;   // Неявное
                  // преобразование в базовый класс

      // Проверка, является ли базовый тип после
      // преобразования в него производного производным
      // типом. Если не является, то результатом as будет
      // ссылка null
      DerivedType derived_Obj2 = base_Obj2 as DerivedType;
      if(derived_Obj2 != null)
      { Console.WriteLine("Производный тип " +
                  "преобразовался в базовый успешно");
      }
      else
      { Console.WriteLine("Преобразование производного "+
                  "типа в базовый не удалось");
      }
```

```
        // Проверка: является ли базовый тип производным
        derived_Obj2 = base_Objl as DerivedType;
        if(derived_Obj2 != null)
        { Console.WriteLine("Базовый  тип преобразовался "+
                    "в производный успешно");
        }
        else
        { Console.WriteLine("Преобразование базового типа "+
                    "в производный не удалось");
        }

        // Проверка: является ли производный тип базовым
        BaseType base_Obj3 = derived_Obj as BaseType;
        if (base_Obj3 != null)
        { Console.WriteLine("Производный тип совместим " +
                    "с базовым");
        }
        else
        { Console.WriteLine("Производный тип не совместим "+
                    "базовым");
        }
        Console.Read();
    }
  }
}
```

Оператор `as` в отличие от `is` при несовместимости типов выдает значение `null`.

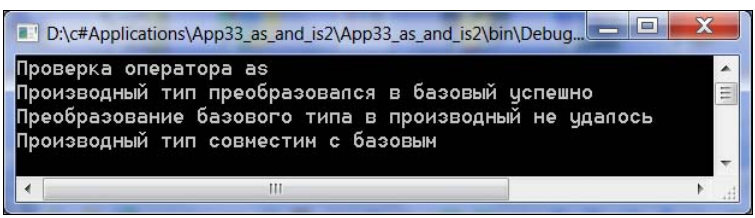


Рис. 8.13. Проверка работы оператора `as`

Структуры

Думается, что такая сущность в языке, как структура, являлась непосредственной предшественницей понятия классов. Действительно, идея объединить в один объект, в одну сущность данные различных типов появилась до классов, а в классе эта идея получила свое настоящее воплощение. Поэтому в таком языке, как C, структуры изучаются довольно подробно и с различными усовершенствованиями от одной версии языка к другой до изучения классов. Однако в C#, став на позицию класса, можно увидеть, что фактически структура — это частный случай класса, правда, с некоторыми особенностями.

Начнем с объявления структуры. Синтаксис объявления таков (покажем на примере):

```
public struct Person
{
  public int x, y;
  public Person(int p1, int p2)
  {
    x = p1;
    y = p2;
  }
}
```

Ключевое слово `struct` определяет заданную сущность как структуру. После этого задается имя структуры (`Person`). Далее знакомые нам понятия: поля, конструктор. Но для структур имеют место следующие особенности.

- В отличие от класса структура не поддерживает наследования.
- В объявлении структуры поля не могут быть инициализированы до тех пор, пока они будут объявлены как постоянные или статические.
- Структура не может объявлять используемый по умолчанию конструктор (конструктор без параметров).
- Структуры копируются при присваивании. При присваивании структуры к новой переменной выполняется копирование всех данных, а любое изменение новой копии не влияет на данные в исходной копии. Вспомните, что у классов данные не перемещаются, а копируются только ссылки на данные. Отсюда видно, что структуры являются типами значений, а классы — ссылочными типами.

- В отличие от классов структуры можно создавать без использования оператора `new` (т. е. они могут не храниться в куче).
- Структуры могут объявлять конструкторы, имеющие параметры.

Пример программы, использующей структуру, приведен в листинге 8.17.

Листинг 8.17

```
/* Created by SharpDevelop.
 * User: user
 * Date: 07.12.2012
 * Time: 12:34
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app35_struct
{
  public struct Person
  {
    public int x, y;
    int c;  // Тип по умолчанию — private

    public Person(int p1, int p2, int p3)
    {
      x = p1;
      y = p2;
      c = p3;
    }
    public int M(int a)
    {
      c=a;
      return c;
    }
  }

  class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Работа со структурами");
```

```
      Person p = new Person(1,2,3);
      int sum = p.x + p.y;
      sum += p.M(4);  // M() переопределяет поле "c"

      Console.Write("Press any key to continue...");
      Console.ReadKey(true);
    }
  }
}
```

Резюме

Изучение классов и начáл объектно-ориентированного программирования, в основном, закончено. Как пользоваться классами? Классы — это обычные типы данных, только более сложные. Обычные в том смысле, что объявления переменных этого типа — типа классов — происходит по тем же правилам, что и объявления переменных обычных, привычных нам простых переменных: целочисленных, строковых и т. п. Но есть небольшая добавка: если простые переменные объявляются тоже просто: тип — имя, то для классов надо еще через знак присвоения создать из них экземпляр (или, что то же самое, объект) с помощью оператора `new` и конструктора класса, в котором задаются инициализирующие создаваемый объект аргументы. С объектом уже можно работать. Объект, созданный из класса как из шаблона, содержит в себе все необходимые данные (поля, свойства и методы) для своей обработки. И наоборот, если вы хотите обрабатывать некий объект, то создаете для него класс, в который закладываете необходимые характеристики объекта (поля и свойства) и методы обработки этих характеристик, а затем объявляете некую переменную типом этого класса, создаете из класса объект и т. д. Кроме всего прочего, вы можете пользоваться значительной библиотекой классов, поставляемых вместе со средой программирования, подключая их к своей программе специальными средствами, о которых мы узнаем позднее.

Мы также рассмотрели понятие структуры. Но теперь, когда мы почувствовали мощь такой сущности, как класс, работа со структурами кажется малоинтересной.

ГЛАВА 9

Обработка исключительных ситуаций

Блоки *try* и *catch*

При разработке программы большая часть ошибок, допускаемая программистом, выявляется компилятором. Однако существуют ошибки, которые компилятор не в состоянии обнаружить, потому что они возникают только на этапе выполнения программы. Классический пример — деление числа на ноль. Если программист малоопытный или просто забыл про эту ситуацию, то когда он выполняет операцию деления чисел, то обязан знать, что делитель может быть и нулем, и это приведет к остановке программы. Поэтому в таких случаях программисты всегда проверяют делитель на ноль, если делитель — целое число, или на заданную программистом малую величину, меньше которой делитель принимается за ноль, если делитель — число с плавающей точкой. Другая характерная ошибка программиста — ситуация, когда при работе с массивами индекс массива превышает допустимую границу. Если программист плохо или совсем не разработал контроль ввода данных, то на этапе эксплуатации программы пользователь может ввести данные не по правилам, определенным в программе. Опять будет остановка программы. Или, например, в программе идет подключение к базе данных, которая уже не существует. Или открывается файл, а он поврежден. Подобные исключительные ситуации в языке C# называются *исключениями*. Когда обрабатывающая среда обнаруживает исключение, она автоматически прерывает выполнение программы и выдает соответствующее сообщение, чтобы в дальнейшем можно было бы разобраться, почему возникла такая ситуация. Но можно избежать неприятностей, связанных с прерыванием работы программ, если самому программисту на этапе разработки программы побеспокоиться о возможном возникно-

вении в том или ином участке программы исключений, перехватить их, не дав исполняющей среде самой их обработать и совершить останов программы, а после перехвата обработать возникшую ситуацию соответствующим образом и либо продолжить выполнение программы, либо завершить программу с выдачей своего сообщения.

В языке C#, а точнее в среде .NET, которая работает не только с C#, предусмотрен стандартный подход к обработке исключительных ситуаций — методика, которая, как и при структурном программировании, содержит в себе строго определенные шаги по решению проблемы исключительных ситуаций.

- Начнем с того, что для каждой группы исключительных ситуаций разработан свой класс, члены которого обеспечивают обработку ситуаций данной группы. То есть такой класс представляет для программиста детали исключительной ситуации.
- В каждом таком классе существует член класса, который способен генерировать в программе исключение (т. е. исключительную ситуацию) при возникновении соответствующих обстоятельств.
- Существует участок программы, который подлежит контролю на возникновение в нем исключения, и этот участок озаглавливается специальным ключевым словом `try` (пытаться).
- Существует участок программы, который обеспечивает перехват и обработку исключительной ситуации, и этот участок озаглавливается ключевым словом `catch` (схватить, поймать).
- Существует участок программы, озаглавленный ключевым словом `finally` (в конце концов), в котором совершаются завершающие действия по обработке ситуации.
- Существует специальный оператор `throw` (выбросить), с помощью которого можно самому в программе с его помощью получить исключительную ситуацию, если она возникает (исключительные ситуации генерируются разными средами вне программы (неявно)), а оператор `throw` генерирует такую ситуацию явно.

Как все классы наследуются от базового класса `Object`, так и все классы по обработке исключений наследуются от базового класса `System.Exception`, который, естественно (потому что он — класс), тоже наследуется о того же `Object`.

Пример использования обработки исключительной ситуации, когда исключение генерируется самой программой, т. е. явно, приведен в листинге 9.1, а результат показан на рис. 9.1.

Листинг 9.1

```
/* Created by SharpDevelop.
 * User: user
 * Date: 06.12.2012
 * Time: 12:32
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app34_exeption
{
  class Program
  {
    static void Main()
    { Console.WriteLine("Ввод строк длиной " +
                        "не более 5 символов");
      int i = (Console.ReadLine().Length);
      try
      {
        if (i > 5)
         // Генерируем исключение
        throw new Exception(); // OverflowException();
      }
      catch (Exception ex)      // OverflowException
      {
       Console.WriteLine("Длина строки более 5 " +
                         "символов");
      }
      Console.ReadLine();
    }
  }
}
```

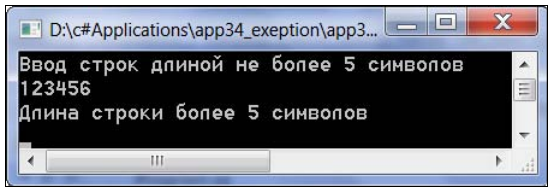


Рис. 9.1. Демонстрация обработки исключения, когда введено более 5 символов

В программе видим два блока обработки исключения, озаглавленных ключевыми словами `try` и `catch`. Каждый блок — это множество операторов, заключенных в фигурные скобки. Блоки строятся следующим образом: участок программы, который может стать проблемным, заключается в блок `try`. В нашем случае могут возникнуть проблемы при вводе строки. Мы хотим ограничить ее длину пятью символами. Заключаем в блок `try` участок проверки длины введенной строки и не ждем от исполняющей среды никаких действий по генерации исключения (да она тут его и не сгенерирует), а сами вставляем в блок `try` оператор `trow`. Синтаксическая конструкция его видна в тексте программы. `Exception()` — это конструктор базового класса по обработке исключений, который создает из класса объект, элементами которого мы можем пользоваться (пока что мы не воспользовались ни одним). Ранее отмечалось, что для различных конкретных групп ситуаций существуют свои классы-исключения. В данной ситуации можно было бы воспользоваться классом `OverflowException`, который закомментирован в строке с оператором `throw`. Этот класс как раз и является разновидностью классов-исключений, который дает возможность обрабатывать переполнения (в частности, переполнения строк по длине, чем можно было бы воспользоваться). Но часто достаточно пользоваться самым общим классом — `Exception`, название которого легко запомнить и который тоже дает возможность обработки исключений.

У каждого `try` есть пара — `catch`. Тоже блок операторов. В этом блоке идет собственно обработка ситуации. Если в блоке `try` исключение не возникло, блок `catch` пропускается (это, как пара `if...else`). Таких пар может быть сколько угодно: каждый подозрительный участок вы можете охватить этой парой. Заголовок блока `catch` — это фактически метод с одним параметром типа одного из классов-исключений. В данном случае этот класс тот, который и выдал исключение в операторе `throw`. Поэтому внутри тела `catch` мы можем пользоваться членами класса `Exception`. Следующая программа, текст которой приведен в листинге 9.2, и демонстрирует этот факт. Результат ее работы представлен на рис. 9.2.

Листинг 9.2

```
/* Created by SharpDevelop.
 * User: user
 * Date: 06.12.2012
 * Time: 12:32
 *
```

```
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app34_exeption
{
  class Program
  {
    static void Main()
    {
      Console.WriteLine("Ввод строк длиной не более 5 " +
                        "символов");
      int i = (Console.ReadLine().Length);
      try
      {
        if (i > 5)
          // Генерируем исключение
          throw new Exception(); // OverflowException();
      }
      catch (Exception ex) // OverflowException
      {
        Console.WriteLine("Длина строки более 5 " +
                          "символов");
        Console.WriteLine("Message = {0}", ex.Message);
        Console.WriteLine("Source = {0}", ex.Source);
        Console.WriteLine("StackTrace = {0}",
                          ex.StackTrace);
        Console.WriteLine("TargetSite = {0}",
                          ex.TargetSite);
        Console.WriteLine("HelpLink = {0}", ex.HelpLink);
      }
      Console.ReadLine();
    }
  }
}
```

Из рис. 9.2 мы видим, что появилась дополнительная информация по возникшему исключению за счет вывода на экран свойств объекта `Exception`.

- Свойство `Message` сообщает, какого типа возникло исключение.
- Свойство `Source` указывает на источник возникновения исключения.

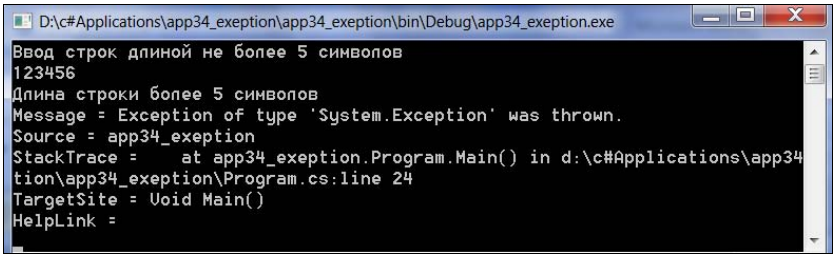


Рис. 9.2. Уточнение причин возникновения исключения с помощью членов класса `Exception`

- Свойство `StackTrace` уточняет источник возникновения исключения: дело в том, что при работе программы могли вызываться функции "в глубину", данные о каждом вызове которых запоминаются в стеке программы. Рассматриваемое свойство дает "трассировку" (просмотр пути по стеку) стека и в конечном итоге показывает номер строки текста той функции, в которой возникло исключение.
- Свойство `TargetSite` указывает по-крупному источник возникновения исключения. В нашем случае это `Main()`.
- Свойство `HelpLink` позволяет указать конкретный URL-адрес, где можно получить более детальную информацию о возникшем исключении. По умолчанию значение этого свойства — пустая строка. Но можно самому задать конкретный адрес. Самый простой способ — в блоке `catch` первой строкой записать `ex.HelpLink = "URL-адрес";`. Можно создать свой класс из класса `Exception` и в его конструкторе добавить нужную инициализацию свойства `HelpLink`.

Блок *finally*

Иногда требуется определить участок программы, который будет выполняться после выхода из блока `try...catch`. Например, исключительная ситуация возникла в связи с ошибкой, приводящей к преждевременному возврату из текущего метода. Но в этом методе мог быть открыт файл, который нужно закрыть, или же установлено сетевое соединение, требующее разрыва. Подобные ситуации нередки в программировании, и поэтому для их разрешения в C# предусмотрен удобный способ: воспользоваться блоком `finally`.

Использование блока `finally` гарантирует, что некоторый набор операторов будет выполняться всегда, независимо от того, возникло исключение (любого типа) или нет. Для того чтобы задать участок программы,

который должен выполняться после блока `try...catch`, достаточно вставить блок `finally` в конце последовательности операторов `try...catch`. Далее приведена общая форма совместного использования блоков `try...catch` и `finally`:

```
try
{
  // Участок, предназначенный для обработки ошибок
}
catch ()
{
  // Обработчик исключения
}
finally {
  // Участок завершения обработки исключений:
  // закрытие нужных файлов, разрыв соединений и т. п.
}
```

Блок `finally` будет выполняться всякий раз, когда происходит выход из блока `try...catch`, независимо от причин, которые к этому привели. Это означает, что если блок `try` завершается нормально или по причине исключения, то последним выполняется участок программы, определяемый в блоке `finally`. В случае нашего примера в листинге 9.2 после закрывающей скобки от `catch` можно было бы поставить участок:

```
finally
  {Console.WriteLine("Конец программы");}
```

Окончательный вид программы представлен в листинге 9.3. Результат ее работы — на рис. 9.3.

Листинг 9.3

```
/* Created by SharpDevelop.
 * User: user
 * Date: 06.12.2012
 * Time: 12:32
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app34_exeption
{
  class Program
  {
    static void Main()
```

```
    {
      Console.WriteLine("Ввод строк длиной не более 5 " +
                        "символов");
      int i = (Console.ReadLine().Length);
      try
      {
        if (i > 5)
          // Генерируем исключение
          throw new Exception(); // OverflowException ();
      }
      catch (Exception ex) // ExceptionOverflow
      {
        ex.HelpLink="Micrisoft.com";
        Console.WriteLine("Длина строки более 5 " +
                          "символов");
        Console.WriteLine("Message = {0}", ex.Message);
        Console.WriteLine("Source = {0}", ex.Source);
        Console.WriteLine("StackTrace = {0}",
                          ex.StackTrace);
        Console.WriteLine("TargetSite = {0}",
                          ex.TargetSite);
        Console.WriteLine("HelpLink = {0}", ex.HelpLink);
      }
      finally
      {
        Console.WriteLine("Конец программы");
      }
      Console.ReadLine();
    }
  }
}
```

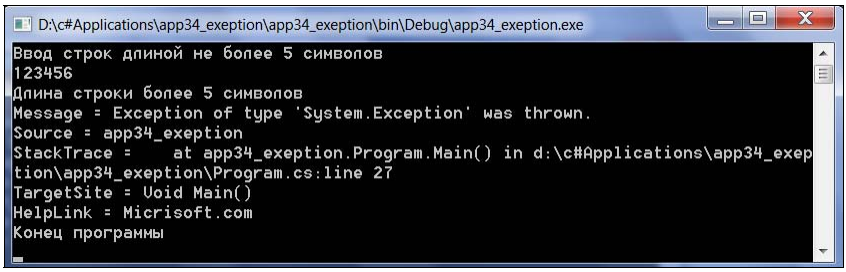


Рис. 9.3. Обработка исключительной ситуации — результат

ГЛАВА 10

Интерфейсы

Интерфейс — дословно "сопряжение". В нашем случае это некоторая программа, обеспечивающая взаимодействие между другими программами или их частями. В рамках языка C# интерфейсы — это какие-то абстрактные члены, объединенные под одним именем. Это как бы абстрактные классы, но более узкие. Абстрактные классы, кроме абстрактных членов содержат еще конструкторы, поля данных, неабстрактные члены (члены с реализацией). Интерфейсы содержат только абстрактные члены.

Для чего их ввели в язык? Дело в том, что абстрактные классы позволяют настраивать свои члены только в классах-наследниках (потомках). Если же у вас в проекте имеется множество несвязанных классовых иерархий, а вам надо, чтобы в этих иерархиях было бы обращение к одному и тому же методу, который что-то там делает (в данном случае говорят: надо, чтобы в иерархиях поддерживался один и тот же полиморфный интерфейс), то приходится настраиваться в каждой из таких иерархий на метод (на абстрактный метод), который находится в программной сущности под названием "интерфейс". Ваш данный конкретный класс подключается как бы к этому интерфейсу (по синтаксису это происходит как и при наследовании: через двоеточие, только вместо класса родителя пишется конкретный интерфейс, из которого будет заимствован необходимый метод). Причем в C# не допускается множественное наследование, т. е. построение класса на основании нескольких классов, чтобы вытащить из них нужные члены, а "наследование" от нескольких интерфейсов допускается. Итак, интерфейс — это заголовки неких абстрактных методов, объединенных под одним именем — именем интерфейса. Например, некий интерфейс `IFILE` может содержать методы `Fileopen()`, `FileClose()` и др.

Вот еще: когда вы строите наследников от базового класса и в родителе имеются некоторые методы `M1()`, `M2()`, `M3()`, вы обязаны все три метода определить в каждом наследнике. Строго. А если вам не нужны все методы, а только один, например `M2()` в некотором наследнике `Inh`? В этом случае выручает интерфейс. Вы определяете интерфейс с абстрактным членом `M2()`, вставляете его в `Inh` — и всё. Остальных наследников не трогаете.

В интерфейсе задаются только заголовки методов, реализация методов ложится на класс, который этот интерфейс будет подключать. Например, класс `Triangles` (треугольники) происходит от класса `Shapes` (фигуры). К классу `Triangles` добавляется интерфейс `IPointy`, подсчитывающий количество вершин фигуры. Тогда в классе `Triangles` этот интерфейс должен быть определен окончательно:

```
Class Triangles : Shapes, IPointy
{
  // Поля класса
  // Конструкторы класса
  // Свойства класса
  // Методы класса
  public override void Draw();

  // Реализация в классе интерфейса iPointy:
  public  byte Points  // Реализация в виде свойства
  {
    get { return 3; }  // Возвращает число вершин
                       // треугольника
  }
}  // Конец класса
```

Если взять класс `Hexagon` (шестиугольники) и заставить его поддерживать интерфейс `iPointy`, то получим вид класса `Hexagon`:

```
Class Hexagon : Shapes, IPointy
{
  // Поля класса
  // Конструкторы класса
  // Свойства класса
  // Методы класса
  public override void Draw();

  // Реализация в классе интерфейса iPointy:
  public  byte Points  // Реализация в виде свойства
```

```
  {
    get { return 6; }  // Возвращает число вершин
                       // многоугольника
  }
}  // Конец класса
```

И вызов интерфейса идет как вызов обычного члена из объекта.

```
Hexagon hex = new Hexagon();
int i = hex.Points;
```

В отличие от классов допускается создание одного интерфейса из многих (множественное наследование).

Пример создания и использования интерфейса показан в программе, представленной листингом 10.1.

Листинг 10.1

```
/* Created by SharpDevelop.
 * User: user
 * Date: 07.12.2012
 * Time: 15:55
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app37_interface
{
  interface ICar
  {
    int Speed{get; set;}
    void GetInfo();
  }

  interface IPrice
  {
    double Price();
  }
  public class Ferrari: ICar,IPrice
  {
    private int spd; // скорость (поле)
```

```
    public int Speed // скорость (свойство)
    {
      get { return spd; }
      set { spd = value; }
    }
    // Конкретная реализация интерфейса в классе:
    public void GetInfo()
    {
      Speed=250;
      Console.WriteLine("Это суперкар Ferrari @ +
                        "с макс. скоростью {0}", Speed);
    }

    // Конкретная реализация интерфейса в классе:
    public double IPrice()
    {
      Console.WriteLine("\nЦена автомобиля 20000");
    }
  }

  class Program
  {
    public static void Main()
    {
      Console.WriteLine("Создание и применение " +
                        "интерфейса");
      Ferrari fr = new Ferrari();
      fr.GetInfo();
      Console.Write("Press any key to continue...");
      Console.ReadKey(true);
    }
  }
}
```

В программе объявлен интерфейс `ICar` с одним свойством `Speed` (скорость автомобиля) и одним методом `GetInfo()`. Что делает этот метод, предстоит конкретно определить в классе, который станет подключать данный метод. Далее создан класс `Ferrari` (марка автомобиля), и в классе определены поле `spd` (скорость) и к нему свойство `Speed`. Затем в класс добавлена конкретная реализация абстрактного метода интерфейса `ICar`: установлена максимальная скорость автомобиля и выдана об этом информация. Результат работы программы показан на рис. 10.1.

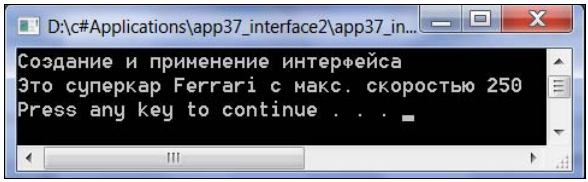


Рис. 10.1. Работа класса с интерфейсом

К классу можно подключать сколько угодно интерфейсов (через запятую). Пример показан в листинге 10.2: добавлен еще один интерфейс, который должен указать стоимость автомобиля.

Листинг 10.2

```
/* Created by SharpDevelop.
 * User: user
 * Date: 07.12.2012
 * Time: 15:55
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app37_interface2
{
  public interface  IPrice
  {
    double Price();
  }

  public interface ICar
  {
    int Speed {get; set;}
    void GetInfo();
  }

  public class Ferrari : ICar,IPrice
  {
    private int spd;  // скорость (поле)
    public int Speed  // скорость (свойство)
    {
      get { return spd; }
```

```
      set { spd = value; }
    }
    // Конкретная реализация интерфейса в классе:
    public void GetInfo()
    {
      Speed=250;
      Console.WriteLine("Это суперкар Ferrari " +
                        "с макс. скоростью {0}", Speed);
    }

    // Конкретная реализация интерфейса в классе:
    public double Price()
    {
      return 20000;
    }
  }

  class Program
  {
    public static void Main()
    {
      Console.WriteLine("Подключение к классу более " +
                        "одного интерфейса");
      Ferrari fr = new Ferrari();
      fr.GetInfo();
      Console.WriteLine("Цена автомобиля — {0}",
                        fr.Price());
      Console.Write("Press any key to continue...");
      Console.ReadKey(true);
    }
  }
}
```

Результат программы представлен на рис. 10.2.

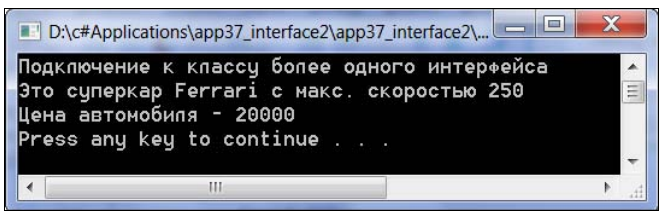


Рис. 10.2. Подключение двух интерфейсов к классу

Может случиться так, что в разных интерфейсах находится один и тот же метод (заголовок один и тот же). Как этот метод реализовать в подключающем их классе? Ведь когда создавались интерфейсы (и, возможно, разными разработчиками), очевидно, что у них предполагалась разная функциональность. То есть надо добиться того, чтобы для одного интерфейса в классе метод реализовывал одну функциональность, а для метода с тем же заголовком — другую функциональность. Очевидно, реализация должна быть с добавкой префикса к имени интерфейса с точкой к имени метода. Например, для метода `price()` в разных интерфейсах надо было бы писать при его реализации в классе: `double interf1.price() {реализация}`, а для этого же метода в другом интерфейсе — `double interf2.price() {реализация}`.

Интерфейсы могут наследовать. У них допускается множественное наследование (один интерфейс может наследовать от многих). Синтаксис наследования — такой же, как у классов, а перечень родителей, от которых идет наследование, разделяется запятыми.

ГЛАВА 11

Сборки, манифесты, пространства имен. Утилита IL DASM

Для лучшего понимания дальнейшего материала рассмотрим некоторые понятия и технологию работы среды исполнения.

- Common Intermediate Language(сокращенно CIL) — промежуточный язык, разработанный фирмой Microsoft для платформы .NET Framework. Компиляторы со всех языков, с которыми работает платформа .NET, переводят программы с этих языков в промежуточный язык CIL. Пример показан на рис. 11.1. Язык CIL по синтаксису и мнемонике напоминает язык ассемблера. В то же время язык CIL содержит некоторые достаточно высокоуровневые конструкции, повышающие его уровень по сравнению с ассемблером для любой реально существующей машины, и писать код непосредственно на CIL легче, чем на ассемблере для реальных машин. Поэтому его можно рассматривать как своеобразный "высокоуровневый ассемблер". Язык CIL также нередко называют просто IL (Intermediate Language), т. е. "промежуточный язык".
- Common Language Runtime (CLR) — общая языковая среда исполнения программ на языках .NET.
- Just in Time (JIT) — компилятор с языка CIL. Является частью называемой CLR. Прямо в момент исполнения программа транслируется в машинный язык и исполняется.
- Код программы — команды программы после их компиляции.
- Метаданные платформы .NET — специальные структуры данных, добавляемые в код программы на языке CIL. Метаданные описывают все классы и члены классов, а также классы и члены классов, ко-

торые программой вызываются из другой программы. Метаданные для метода содержат полное описание метода, включая его класс (а также программу, содержащую этот класс), его возвращаемый тип и все параметры этого метода. Вообще метаданными называют не обычные в бытовом понимании данные, а некие другие, специфические, преобразованные данные, отличные от обычных.

- Dinamic Link Library (dll) — библиотека программных модулей, откомпилированных и готовых встроиться в вызывающую программу. Модули могут подключаться к программе на этапе создания exe-файла (исполняемого файла) или в момент исполнения программы. Содержат различные стандартизованные функциональные блоки.

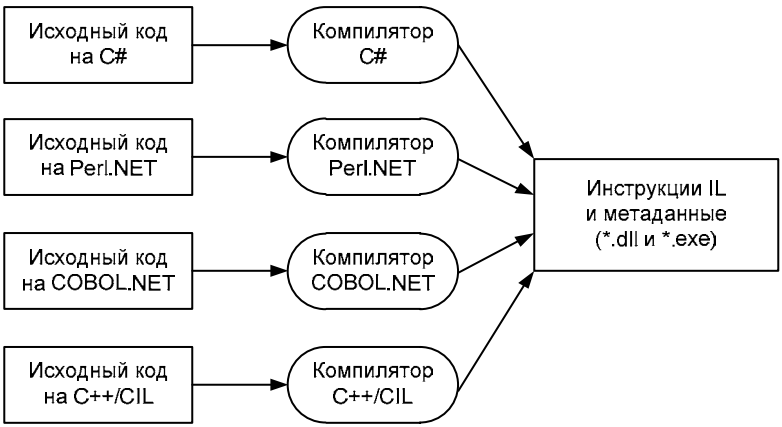


Рис. 11.1. Организация взаимодействия с языками на платформе .NET

Сборки

Сборка — это exe- или dll-файл. Первый тип сборки называют статическим, второй — динамическим. Какую сборку можно получить, определяется заданием режима компиляции (задается специальным ключом). Сборки могут содержать один или несколько модулей. Например, крупные проекты могут быть спланированы таким образом, чтобы несколько разработчиков работали каждый над своим модулем, а вместе эти модули образовывали бы одну сборку. Файлы, составляющие сборку, объединены логически, независимо от их физического положения. Связывает их *манифест*. Среда выполнения расценивает сборку, как цельный

модуль. Манифест — часть сборки, в которой содержатся метаданные, т. е. данные, описывающие сборку, и параметры для ее использования. Манифест содержит имя сборки, номер версии, информацию о поддерживаемых процессорах и операционных системах, имена файлов, образующих сборку, информацию, связывающую типы с содержащими их файлами, информацию о сборках, используемых данной сборкой.

Сведения о сборке могут быть просмотрены с помощью специальной программы IL DASM, выход на вызов которой находится в интегрированной среде в меню **Tools** (рис. 11.2).

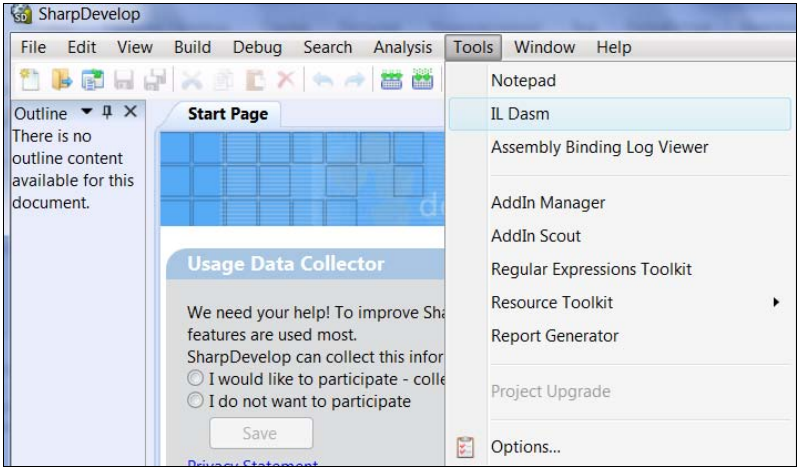


Рис. 11.2. Вызов IL Dasm

Команда **IL DASM** позволяет открыть диалоговое окно для выбора сборки. Для иллюстрации мы выбрали приложение по работе с интерфейсами, рассматриваемое в предыдущей главе. Вид сборки показан на рис. 11.3.

Из рисунка видна полная структура модуля приложения. Модуль можно более детально исследовать с помощью команд меню **View** (прятать строки с некоторыми свойствами, оставляя видимыми другие строки и т. п.). Если дважды щелкнуть на методе `Main()`, откроется окно с программой на языке CIL, в который программы транслируются компиляторами всех языков, входящих в платформу .NET. Если же требуется посмотреть содержимое манифеста, надо дважды щелкнуть на нем мышью.

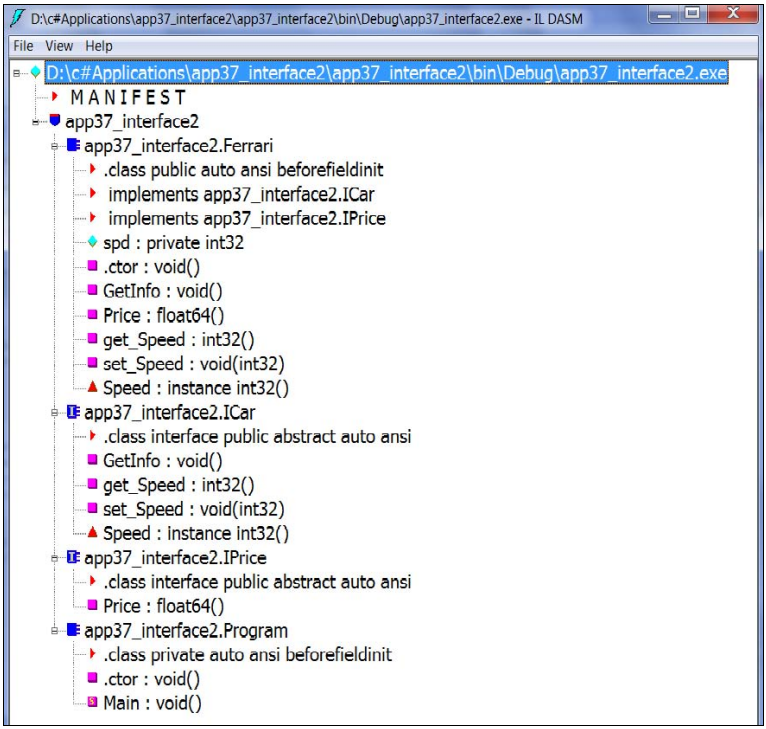


Рис. 11.3. Сборка приложения по работе с двумя интерфейсами из главы 10

Пространства имен

Пространство имен (namespace) — концепция, позаимствованная из C++ и позволяющая обеспечить уникальность всех имен, используемых в конкретной программе или проекте.

Если вы, например, состоите в группе разработчиков некоторого крупного проекта и создаете свой модуль, то вам не обязательно заботиться о наименовании переменных в этом модуле, о том, что когда все разработчики начнут собирать свои модули в единый проект, появятся переменные с одинаковыми именами и разным смыслом, начнется путаница, и проект станет неработоспособным. Концепция пространства имен как раз и обеспечивает независимую разработку модулей одного проекта, потому что каждый разработчик может объявить свое личное пространство имен и в нем называть переменные по своему усмотрению. А об-

щий проект будет указывать, что он использует такое-то пространство имен, чтобы работать с таким-то модулем. Компилятор просто в этом случае к каждому внутреннему имени добавляет имя пространства имен и тем делает совпадающие имена в разных пространствах различными. Такая же история и с библиотеками классов в C#. Классы создавались разными разработчиками и могут иметь совпадающие имена (да так оно и есть на самом деле: например, классов `Timer` в .NET Framework насчитывают три единицы). Поэтому классы разделены: они сгруппированы по своим функциональным свойствам и распределены по разным именованным пространствам, главным из которых является пространство `System`.

Более 90 пространств имен, определенных в .NET Framework, начинаются со слова `System`.

Как создать себе свое пространство имен? По синтаксису:

```
namespace Имя
{
  class имя
  {
  ...
  }
  ...
}
```

Внутри пространства имен вы можете объявить один и более следующих типов:

- другое пространство имен;
- класс;
- интерфейс;
- структуру.

Вообще тогда при обращении к какому-то классу при таком подходе надо указывать всю цепочку названий пространств, в которую входит данный класс, т. к. пространства составляют некую иерархию и названия пространств довольно длинные. Но если применить ключевое слово `using` (используя) перед составным именем пространства, в котором находятся нужные нам классы, то это пространство имен можно не вводить при обращении к его классам.

Некоторые пространства имен среды .NET приведены в табл. 11.1.

Таблица 11.1. *Пространства имен среды .NET*

Название пространства	Описание
`System`	Содержит основные классы и классы, которые определяют базовые типы данных не только числовых, но и ссылочных, классы событий и обработчиков событий, интерфейсы и классы обработки исключений, классы по преобразованию типов данных, математических функций, вызовов локальных и удаленных программ, управления приложениями. Содержимое можно посмотреть по ссылке **http://msdn.microsoft.com/en-us/library/system.aspx**
`System.Collections` `System.Collections.Generic`	Содержат классы, позволяющие создавать специальные коллекции данных: списки, очереди, битовые массивы, словари. Содержимое можно посмотреть по ссылке **http://msdn.microsoft.com/en-US/library/system.collections(v=VS.80).aspx**
`System.Data`	Содержит классы для взаимодействия с базами данных. Содержимое можно посмотреть по ссылке **http://msdn.microsoft.com/en-us/library/system.data.aspx**
`System.IO`	Содержит типы данных для обработки файлового ввода-вывода. Содержимое можно посмотреть по ссылке **http://msdn.microsoft.com/en-us/library/29kt2zfk.aspx**
`System.Drawing` `System.Windows.Forms`	Содержит типы данных для построения графических приложений с использованием набора Windows Forms. Содержимое можно посмотреть по ссылкам: • **http://msdn.microsoft.com/en-us/library/system.drawing(v=VS.85).aspx** • **http://msdn.microsoft.com/en-us/library/system.windows.forms.aspx**

`System.Windows`
`System.Windows.Controls`
`System.Windows.Shapes`
	`System.Windows` является корневым для остальных, которые содержат набор графических инструментов для работы в среде Windows Presentation Foundation (WPF). Содержимое можно посмотреть по ссылкам:
	• **http://msdn.microsoft.com/en-us/library/system.windows.aspx**
	• **http://msdn.microsoft.com/en-us/library/system.windows.controls(v=VS.95).aspx**
	• **http://msdn.microsoft.com/en-us/library/system.windows.shapes.aspx**

Таблица 11.1 *(окончание)*

Название пространства	Описание
`System.ServiceModel`	Содержит инструменты для создания распределенных приложений в среде Windows Communication Foundation (WCF). Содержимое можно посмотреть по ссылке **http://msdn.microsoft.com/en-us/library/system.servicemodel.aspx**
`System.Security`	Содержит типы данных, которые позволяют создавать приложения по криптографической защите и т. п. Содержимое можно посмотреть по ссылке **http://msdn.microsoft.com/en-us/library/windows/apps/hh454038.aspx**

Пространства имен подключаются к программе с помощью ключевого слова `using`. Хорошо, если эти пространства находятся в стандартной библиотеке среды разработки. А если вы сами создали собственные пространства имен, используя свои приложения? Ваши приложения после их компиляции принимают формат сборки. То есть определенные вами пространства имен попали, в общем случае, в разные сборки. Или какой-то другой разработчик определил некие классы в своем пространстве имен в своей сборке, которую вы приобрели каким-то путем. Как подключать эти новые пространства из сборок? Для этого существует команда **Add Reference** (Добавить ссылку) меню **Project** (Проект) среды разработки. Если вы выберете эту команду, откроется диалоговое окно для формирования ссылок на необходимые вам сборки (вкладка **Net Assembly Browser** — поиск NET-сборок). При открытии вкладки видна кнопка **Browse**, которая вас выведет на папку со сборкой.

После того как вы нашли все необходимые вам сборки, содержащие требуемые пространства имен, следует нажать кнопку **OK** и перейти в ваше рабочее приложение. В нем вы с помощью ключевого слова `using` подключаете найденные пространства, записывая их имена через точку от имени сборки, например,

```
using app39_namespace.pbi.MyModul;
```

где `app39_namespace` — имя сборки (расширение exe не включать), а `pbi.MyModul` — имя пространства имен, в котором определены классы. Если вам неудобно пользоваться длинными именами пространств имен, вы можете дать им псевдонимы (alias) по правилу:

```
using alias = длинное_имя
```

и в дальнейшем работать с псевдонимом в качестве имени пространства имен. Все вышесказанное отражено в трех приложениях, приведенных

в листинге 11.1, а результат работы основного приложения показан на рис. 11.4. Первое из приложений создает пространство имен с классом A, второе — с классом B, а третье работает с пространствами, созданными в первом и втором приложениях.

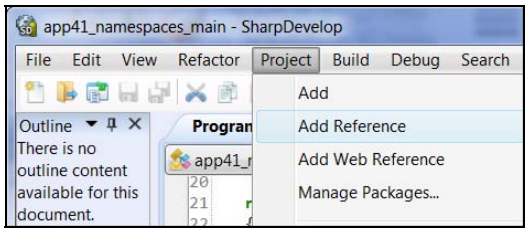
Листинг 11.1

```
// Первое приложение

/* Created by SharpDevelop.
 * User: user
 * Date: 08.12.2012
 * Time: 15:49
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app39_namespace
{
  namespace pbi.MyModul
  {
    public class A
    {
      public int A_a {get; set;}
    }
  }  // MyModul
  class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Hello World!");

      // TODO: Implement Functionality Here

      Console.Write("Press any key to continue...");
      Console.ReadKey(true);
    }
  }
}
```

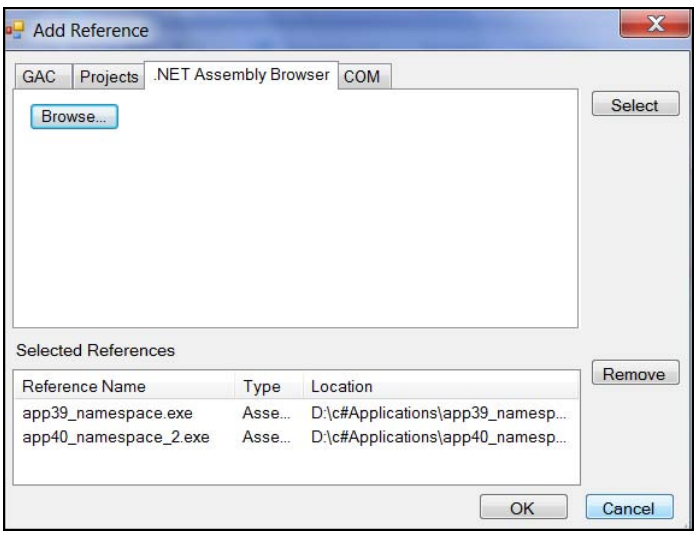
```
// Второе приложение

/* Created by SharpDevelop.
 * User: user
 * Date: 09.12.2012
 * Time: 11:17
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app40_namespace_2
{
  namespace pbi.MyModul2
  {
    public class B
    {
      public int B_b {get; set;}
    }
  } // MyModul2

  class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Hello World!");

      // TODO: Implement Functionality Here

      Console.Write("Press any key to continue...");
      Console.ReadKey(true);
    }
  }
}

// Третье, главное приложение
/* Created by SharpDevelop.
 * User: user
 * Date: 09.12.2012
 * Time: 11:21
 *
```

```
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

// Эти модули находятся в сборках и подключаются
// по ссылке из меню Project

// Можно использовать настоящие имена сборок,
// если они не длинные:
// using app39_namespace.pbi.MyModul;
// using app40_namespace_2.pbi.MyModul2;

// Использование псевдонима вместо длинного имени
using MyModul = app39_namespace.pbi.MyModul;
using MyModul2 = app40_namespace_2.pbi.MyModul2;

namespace app41_namespace_main
{
  class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Обработка модулей, " +
                    "созданных разными источниками\n");

    // app39_namespace.pbi.MyModul.A a = new
app39_namespace.pbi.MyModul.A();
      MyModul.A a = new MyModul.A();
      a.A_a=5;

      // app40_namespace_2.pbi.MyModul2.B b =
                          new app40_namespace_2.pbi.MyModul2.B();
      MyModul2.B b =
        new app40_namespace_2.pbi.MyModul2.B();
      b.B_b=6;

      Console.Write("Работа с классами из разных  +
                   "пространств имен\n");
      Console.Write("Поле A_a = {0}, поле B_b = {1}\n",
                   a.A_a, b.B_b);
```

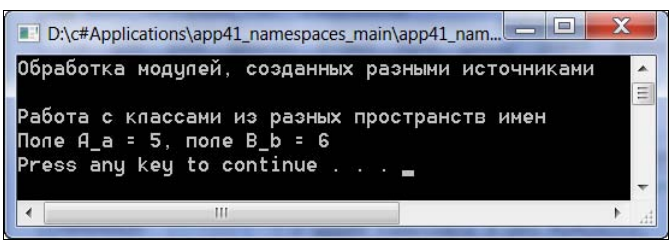
```
        Console.Write("Press any key to continue...");
        Console.ReadKey(true);
    }
  }
}
```



а



б



в

Рис. 11.4. Использование пространств имен из различных сборок: *а*, *б* — последовательность подключения пространств имен; *в* — результат работы основного приложения с использованием пространств имен из разных сборок

Теперь понятно, почему шаблон приложения оформляется как `namespace <имя_приложения>`: после компиляции приложение приобретает формат сборки. Если в нем имеются некоторые классы, функциональность которых полезна для других приложений, то такую сборку можно подключить к своей программе по ссылке и доставать из нее нужную функциональность находящихся в ней классов.

ГЛАВА 12

Коллекции. Обобщения

Коллекции

Для работы с группой однотипных элементов в C# использовался тип данных, называемый массивом. Эта удобная форма обработки групп однотипных данных имеет один существенный недостаток — массив по своему размеру является статическим: сколько элементов в нем определено, со столькими он и работает. То есть не может вести себя динамически: при необходимости, расширяться или сужаться. Если вы хотите, чтобы ваш массив был большего, чем объявлено, размера, надо объявлять новый массив. Существует класс `System.Array`, с помощью экземпляров которого можно получать различные массивы. Класс `System.Array` — один из примеров класса коллекций. *Коллекция* — это систематизированное собрание неких объектов, объединенных по некоторому принципу. Классы C#-коллекций применяются для обслуживания списков объектов и дают значительно больше функциональных возможностей, чем простой массив. Большая часть этих возможностей реализуется интерфейсами из пространства имен `System.Collections`. Основные функциональные возможности из этого пространства имен предоставляют следующие интерфейсы:

- `IEnumerable` — предоставляет возможность организовать работу с элементами коллекции в цикле;
- `ICollection` — дает возможность узнать количество элементов коллекции и копировать элементы в простой массив;
- `IList` — предоставляет возможность сформировать список элементов для коллекции и обеспечивает возможность доступа к этим элементам. Работает с индексом элемента;

- `IDictionary` — похож на `IList`, но работает не с индексом элемента, а с его ключом.

Пример программы создания коллекций `animalArray` и `animalArrayList` и некоторая работа с этими коллекциями приведены в листинге 12.1. Результат работы представлен на рис. 12.1.

Листинг 12.1

```
/* Created by SharpDevelop.
 * User: user
 * Date: 10.12.2012
 * Time: 18:50
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */

namespace app43_collections_2
{
  using System;
  using System.Collections;
  using System.Collections.Generic;
  using System.Linq;
  using System.Text;

  namespace app42_collections_1
  {
    public abstract class Animal
    {
      protected string name; // поле
      public string Name     // свойство
      {
        get {return name;}
        set {name = value;}
      }

      public Animal()  // Конструктор по умолчанию
      {
        name = "Животное без имени";
      }
      public Animal(string newName) // Специализированный
                                    // конструктор
```

```
  {
    name = newName;
  }
  public void Feed()  // Сведения о корме
  {
    Console.WriteLine ("{0} получила корм", name);
  }
} // Animal

public class Cow : Animal   // Коровы
{
  public void Milk() // Данные о дойке
  {
    Console.WriteLine("{0} подоена", name);
  }
  public Cow(string newName) : base(newName)
    // Конструктор (запускается конструктор
    // базового класса)
  {
  }
} // Cow

public class Chicken : Animal  // Цыплята
{
  public void LayEgg()  // Данные о снесенных яйцах
  {
    Console.WriteLine ("{0} снесла яйцо", name);
  }
  public Chicken(string newName) : base(newName)
    // Конструктор (запускается конструктор
    // базового класса)
  {
  }
} // Chicken

class Program
{
  static void Main()
  {
    Console.WriteLine("Создание коллекции типа " +
                      "Array из объектов Animal " +
                      "и ее использование");
```

```
// Объявляется обычный массив animalArray
// из двух элементов.
// Так как любой массив — потомок базового
// Array, то можно пользоваться инструментами
// этого класса
Animal [] animalArray = new Animal [2];
Cow myCowl = new Cow("Чернушка");  // Из класса
                              // формируется объект

// Инициализация массива:
animalArray [0] = myCowl; // В нулевой элемент
// массива засылается объект (корова Чернушка)
animalArray [1] = new Chicken("Ряба"); // В первый
// элемент массива засылается объект
//(Курочка Ряба)

// Цикл по элементам массива и вывод сведений
// об элементах.
// myAnimal — рабочая переменная, по которой идет
// цикл: ей последовательно присваиваются элементы
// из массива, поэтому тип этой переменной и тип
// массива должны быть совместимы

foreach (Animal myAnimal in animalArray)
{
  Console.WriteLine("В коллекцию Array добавлен "+
                    "новый объект, кличка = {0}",
                    myAnimal.Name);
}
// Вывод количества элементов массива
Console. WriteLine("Коллекция Array содержит "+
                   "объектов {0}",
                   animalArray.Length);

animalArray[0].Feed(); // Вывод сведения о корме
                       // первого объекта
((Chicken)animalArray[1]).LayEgg(); // Вывод о
            // снесенных яйцах второго объекта
  // animalArray — объект абстрактного класса,
  // поэтому нужно приведение его к типу Chicken,
  // чтобы добраться до метода LayEgg().
  // Это были операции с простым массивом.
```

```
// Далее пойдут операции с новым типом массива:
Console.WriteLine();

Console.WriteLine("Создание коллекции типа " +
                  "ArrayList из объектов Animal " +
                  "и ее использование:");

ArrayList animalArrayList = new ArrayList();
Cow myCow2 = new Cow("Пятнашка");
animalArrayList.Add(myCow2); // Вот такая здесь
  // инициализация: добавление объекта,
  // а не присвоение. Сразу видна динамика:
  // добавлять можно, сколько потребуется

// Так как Array — прародитель массивов, то можем
// пользоваться его инструментами (Add() и т. д.)
animalArrayList.Add(new Chicken("Хохлатка"));
// Здесь нет ничего нового:
foreach (Animal myAnimal in animalArrayList)
{
  Console. WriteLine("Новый объект добавлен к " +
                     "коллекции ArrayList," +
                     " Имя = {0}", myAnimal.Name);
}
Console.WriteLine("Коллекция ArrayList содержит "+
   "объектов {0} ",
   animalArrayList.Count);  // Здесь вместо длины
         // для подсчета применяется элемент Count

((Animal)animalArrayList[0]).Feed();  // Чтобы
        // добраться до Feed(), надо выполнить
        // приведение к типу Animal, где находится
        // Feed()
((Chicken)animalArrayList[1]).LayEgg();
Console.WriteLine();
Console. WriteLine("Дополнительные действия " +
                   "с  ArrayList:");
// Новый метод из класса: удаление элемента.
// Остальные сдвигаются на его место.
// Видна динамика
animalArrayList.RemoveAt(0);
```

```
        Console.WriteLine("После удаления нулевого " +
              "элемента в массиве остались элементы:");
        foreach  (Animal myAnimal in animalArrayList)
        { Console.WriteLine ("{0}", myAnimal.Name); }

        // Проверка: сдвинулся ли второй объект на место
        // удаленного первого
        ((Animal)animalArrayList[0]).Feed();

        // Новый метод: добавляет к данному массиву
        // ранее рассмотренный массив объектов
        animalArrayList.AddRange (animalArray);

        Console.WriteLine("После добавки массива " +
                          "animalArray в массиве " +
              "animalArrayList находятся элементы:");

        foreach (Animal myAnimal in animalArrayList)
        { Console.WriteLine ("{0}", myAnimal.Name); }

        // Проверка, как прошла добавка
        ((Chicken)animalArrayList[2]).LayEgg();
        Console.WriteLine("Животное по кличке {0} " +
                          "находится на месте {1}",
           // Новый метод: определяет индекс (место)
           // элемента массива
           myCowl.Name, animalArrayList.IndexOf(myCowl));

        myCowl.Name = "Буренка";
        Console.WriteLine("Животное теперь имеет кличку"+
             " {0}", ((Animal) animalArrayList[1]).Name);

        Console.ReadKey();
      }
    }
  }
}
```

Итак, программа сравнивает действия по работе с обычным массивом и массивом типа `ArrayList`. Показано, что последний тип намного мощнее первого. Смысл понятен из комментариев к программе и рис. 12.1.

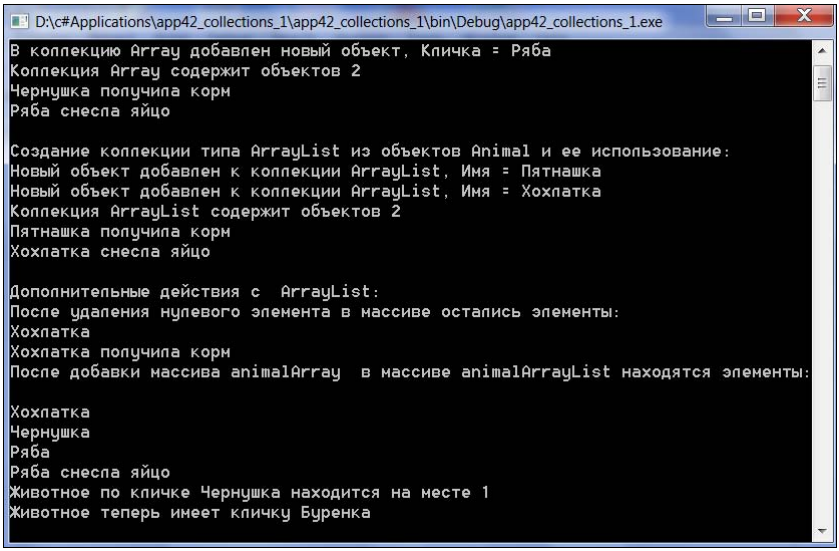


Рис. 12.1. Сравнение результатов создания двух коллекций: `animalArray` и `animalArrayList`

Интерфейсы *IEnumerable* и *IEnumerator*

Рассмотрим интерфейсы `IEnumerable` и `IEnumerator` и как их использовать для создания класса, который можно применять в операторе `foreach`.

Интерфейс `IEnumerator` обеспечивает прогон по коллекции, которая находится внутри класса. `IEnumerator` требует реализации трех методов:

- метода `MoveNext()`, который увеличивает индекс коллекции на 1 и возвращает логическое значение, указывающее, был ли достигнут конец коллекции;
- метода `Reset()`, который сбрасывает индекс коллекции в его начальное значение –1. Это делает перечислитель недействительным;
- метода `Current()`, который возвращает текущий объект.

Интерфейс `IEnumerable` создает перечислитель типа `IEnumerator`.

Оба интерфейса участвуют в работе оператора `foreach`, поэтому должны быть подключены к классу, в котором этот оператор употребляется.

Кроме того, по правилам C# все свойства и методы подключаемых интерфейсов должны быть определены (заданы в производном классе), т. к. в самих интерфейсах определены только заголовки элементов без содержимого тел. Пример применения обоих интерфейсов приведен в листинге 12.2, а результат работы программы — на рис. 12.2.

Листинг 12.2

```
/* Created by SharpDevelop.
 * User: user
 * Date: 13.12.2012
 * Time: 14:34
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Collections;

namespace app46_Ienum_Ienumble
{
  public class car
  {
    // Поля
    private int year;      // Год выпуска
    private string brand;  // Марка

    // Конструктор
    public car(string Brand,int Year)
    {
      brand=Brand;
      year=Year;
    }
    // Свойства
    public int Year
    {
      get  {return year;}
      set  {year=value;}
    }
    public string Brand
    {
      get {return brand;}
```

```
    set {brand=value;}
  }
}  // end class

public class cars : IEnumerator,IEnumerable
{
  // Объявляется массив типа car
  // (пары: "год выпуска — марка")
  private car[] carlist; // Поле
  int position = -1;     // для сброса в начало
                         // будущего перечислителя

  // Конструктор: создает массив carlist
  // и инициализирует его
  public cars()
  {
    carlist = new car[6]
    {
      new car("Ford",1992),
      new car("Fiat",1988),
      new car("Buick",1932),
      new car("Ford",1932),
      new car("Dodge",1999),
      new car("Honda",1977)
    };
  }

  // Определение методов, указанных в интерфейсах
  // в данном классе (требование C#)

  // Метод из IEnumerable
  public IEnumerator GetEnumerator() // Создать
                                     // перечислитель
  { // Возвращает этот же объект, приводя его
    // к типу IEnumerator
    return (IEnumerator)this;
  }

  // Методы из IEnumerator

  public bool MoveNext() // Перейти к следующему объекту
```

```
    {
      position++;
      return (position < carlist.Length);
    }

    public void Reset()  // Сброс перечислителя: установка
                         // указателя на первый элемент
    {position = 0;}

    public object Current  // Выдать текущий объект
    {
      get { return carlist[position];}
    }
  }

  class Program
  {
    public static void Main()
    {
      Console.WriteLine("Совместная работа интерфейсов INumerator,
INumarable\nи оператора foreach");

      cars C = new cars();
      Console.WriteLine("Полученная коллекция\n");
      foreach(car c in C)
        // Здесь применен табулятор для сдвига на две
        // табуляции, чтобы был разделитель между полями
        Console.WriteLine(c.Brand + "\t\t" + c.Year);

      Console.Write("Press any key to continue...");
      Console.ReadKey(true);
    }
  }
}
```

Когда начинает работать цикл `foreach`, сразу идет обращение к методу `MoveNext()`, в котором индекс перечислителя, установленный в начале класса `cars` в –1, увеличивается на 1, тем самым устанавливая перечислитель на 1-й элемент просматриваемого объекта. Затем вызывается метод `Current()`, который выдает текущий элемент оператору (в данном

случае — первый). На этом первый шаг цикла завершается, и выполняется тело `foreach`. После выполнения тела управление передается снова на заголовок оператора, тот вызывает метод `MoveNext()`, который увеличивает индекс перечислителя на единицу. Затем с новым значением индекса вызывается метод `Current()`, который выдает второй элемент, и т. д. Цикл завершится, когда будут исчерпаны все элементы, потому что метод `MoveNext()` выдает индекс, который не превосходит количества элементов.

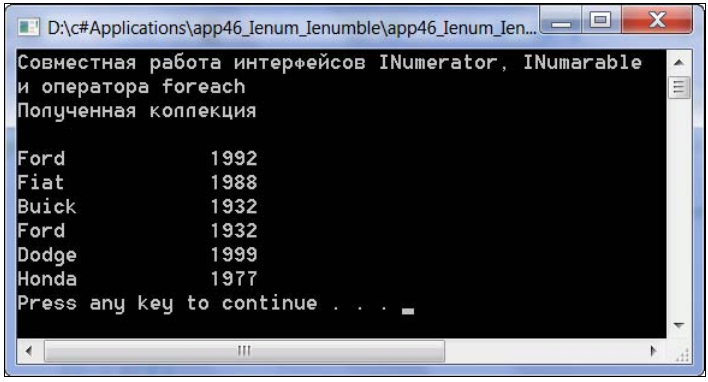


Рис. 12.2. Работа оператора `foreach`

Примечание

Если переменная цикла в `foreach` имеет числовой тип, то оператор не требует создания перечислителя, и подключать оба интерфейса не требуется.

Создание собственного класса коллекций

Для создания собственного класса коллекций пользуются абстрактным классом `System.Collections.CollectionBase`. Этот класс предоставляет два свойства с доступом `protected`: `List` и `InnerList`. `List` обеспечивает доступ к элементам, а `InnerList`, являясь объектом класса `ArrayList`, — хранение элементов. Свойства этого класса показаны в табл. 12.1, а методы — в табл. 12.2.

Таблица 12.1. *Свойства класса* `CollectionBase`

Свойство	Описание
`Capacity`	Получает или задает число элементов, которое может содержать коллекция `CollectionBase`
`Count`	Выдает число элементов, содержащихся в экземпляре класса `CollectionBase`. Это свойство нельзя переопределить
`InnerList`	Возвращает объект `ArrayList`, в котором хранится список элементов экземпляра класса `CollectionBase`
`List`	Получает объект `IList`, содержащий список элементов экземпляра класса `CollectionBase`

Таблица 12.2. *Некоторые методы класса* `CollectionBase`

Метод	Описание
`Clear`	Удаляет все объекты из экземпляра класса `CollectionBase`. Этот метод не может быть переопределен
`Equals(Object)`	Определяет, равен ли заданный объект текущему объекту. Унаследовано от `Object`
`GetEnumerator`	Возвращает перечислитель, осуществляющий перебор элементов экземпляра класса `CollectionBase`
`OnClear`	Осуществляет дополнительные пользовательские действия при удалении содержимого экземпляра класса `CollectionBase`
`OnClearComplete`	Осуществляет дополнительные пользовательские действия после удаления содержимого экземпляра класса `CollectionBase`
`OnInsert`	Выполняет дополнительные пользовательские действия перед вставкой нового элемента в экземпляр класса `CollectionBase`
`OnInsertComplete`	Выполняет дополнительные пользовательские действия после вставки нового элемента в экземпляр класса `CollectionBase`
`OnRemove`	Осуществляет дополнительные пользовательские действия при удалении элемента из экземпляра класса `CollectionBase`

Таблица 12.2 *(окончание)*

Метод	Описание
`OnRemoveComplete`	Осуществляет дополнительные пользовательские действия после удаления элемента из экземпляра класса `CollectionBase`
`OnSet`	Выполняет дополнительные пользовательские действия перед заданием значения в экземпляре класса `CollectionBase`
`OnSetComplete`	Выполняет дополнительные пользовательские действия после задания значения в экземпляре класса `CollectionBase`
`OnValidate`	Выполняет дополнительные пользовательские операции при проверке значения
`RemoveAt`	Удаляет элемент по указанному индексу в экземпляре класса `CollectionBase`. Этот метод нельзя переопределить
`ToString`	Возвращает строку, которая представляет текущий объект. (Унаследовано от `Object`.)
`ICollection.CopyTo`	Копирует целый массив `CollectionBase` в совместимый одномерный массив `Array`, начиная с заданного индекса целевого массива
`IList.Add`	Добавляет объект в конец коллекции `CollectionBase`
`IList.Contains`	Определяет, содержит ли интерфейс `CollectionBase` определенный элемент
`IList.IndexOf`	Осуществляет поиск указанного индекса `Object` и возвращает индекс (с нуля) первого вхождения в коллекцию `CollectionBase`
`IList.Insert`	Добавляет элемент в список `CollectionBase` в позиции с указанным индексом
`IList.IsFixedSize`	Получает значение, показывающее, имеет ли список `CollectionBase` фиксированный размер
`IList.IsReadOnly`	Получает значение, указывающее, доступна ли коллекция `CollectionBase` только для чтения
`IList.Item`	Получает или задает элемент с указанным индексом
`IList.Remove`	Удаляет первый экземпляр указанного объекта из коллекции `CollectionBase`

Заметим, что методы с префиксом "On" — это так называемые *обработчики событий*. Событиями в данном случае являются сами действия по удалению, добавлению и т. п. Реализация этих методов предназначена для переопределения производным классом с целью выполнения некоторых операций перед или в момент (зависит от сути метода) осуществления того или иного действия. Методы "On" вызываются только для экземпляра, возвращенного свойством `List`, но не для экземпляра, возвращенного свойством `InnerList`.

Примеры работы с методами класса `CollectionBase` приведены в приложении, текст которого представлен в листинге 12.3, а результат — на рис. 12.3. Все пояснения — по тексту программы.

Листинг 12.3

```
/* Created by SharpDevelop.
 * User: user
 * Date: 11.12.2012
 * Time: 14:30
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Collections;
namespace app44_collections_CollectionBase_class
{
  public class Int16Collection : CollectionBase
  {
    public Int16 this[ int index ] // Объявление
                                   // индексатора
                                  // (пояснение см. ниже)
    { get  { return((Int16) List[index]); }
      set  { List[index] = value; }
    }

    // Метод, определенный в классе.
    // Методы, определенные в классе, названы так же,
    // как методы из наследуемого класса, работу
    // которых проверяют

    public int Add(Int16 value)
    { return(List.Add(value));  }
```

```
public int IndexOf(Int16 value)
{ return(List.IndexOf(value)); }

public void Insert(int index, Int16 value)
{ List.Insert(index, value); }

public void Remove(Int16 value)
{ List.Remove(value); }

public bool Contains(Int16 value)
{ // Если value не типа Int16, метод возвращает false
  return(List.Contains(value));
}

// Примеры задания обработчиков событий (идет
// переопределение методов абстрактного класса)

protected override void OnInsert(int index,
                                 Object value)
{ /* Сюда вставляются операторы, которые станут
     выполняться только тогда, когда пойдет вставка
     значения values */
}

protected override void OnRemove(int index,
                                   Object value)
{ /* Сюда вставляются операторы, которые станут
     выполняться только тогда, когда пойдет удаление
     значения values */
}

protected override void OnSet(int index,
        Object oldValue, Object newValue)
{ /* Сюда вставляются операторы, которые станут
     выполняться только тогда, когда пойдет установка
     значения values */
}
/* Здесь проверяется значение value на принадлежность
   его к типу Int16.
   Если тип не Int16, генерируется исключение
   с выдачей соответствующего сообщения */
```

```
    protected override void OnValidate(Object value)
    {
      if (value.GetType() != typeof(System.Int16))
        throw new ArgumentException("value must be of type
Int16.", "value");
    }
  } // конец класса

  public class Program
  {
    public static void Main()
    {
      // Создание новой коллекции чисел
      Int16Collection myI16 = new Int16Collection();

      // Инициализация коллекции путем добавления
      // элементов (1, 2, ...) с приведением к типу Int16
      myI16.Add((Int16) 1);
      myI16.Add((Int16) 2);
      myI16.Add((Int16) 3);
      myI16.Add((Int16) 5);
      myI16.Add((Int16) 7);

      // Вывод содержимого коллекции
      Console.WriteLine("Содержимое коллекции " +
                        "с использованием foreach:");
      PrintValues1(myI16);

      Console.WriteLine("Содержимое коллекции " +
                        "с использованием enumerator:");
      PrintValues2(myI16);

      Console.WriteLine("Содержимое коллекции " +
                        "с использованием Count и Item:");
      PrintIndexAndValues(myI16);

      // Поиск в коллекции элемента со значением 3
      // с использованием Contains и IndexOf
      Console.WriteLine("Contains 3: {0}",
                        myI16.Contains(3));
      Console.WriteLine("2 имеет индекс {0}.",
                        myI16.IndexOf(2));
      Console.WriteLine();
```

```
  // Вставка элемента в коллекцию, начиная со значения
  // индекса 3
  myI16.Insert(3, (Int16) 13);
  Console.WriteLine("Содержимое коллекции после " +
     "вставки 13, начиная со значения индекса 3:");
  PrintIndexAndValues(myI16);

  // Чтение и установка элемента с использованием
  // индекса: myI16[4] = 123;
  myI16[4] = 123;
  Console.WriteLine("Содержимое коллекции после " +
        "установки элемента с индексом 4 на 123:");
  PrintIndexAndValues(myI16);

  // Удаление элемента со значением 2 из коллекции
  myI16.Remove((Int16) 2); // RemoveA() удаляет по
                      // индексу, а не по значению
  Console.WriteLine("Содержимое коллекции после " +
                "удаления элемента со значением 2:");
  PrintIndexAndValues(myI16);
  Console.Read();
}
// Вывод содержимого коллекции с использованием
// свойств Count и Item

public static void PrintIndexAndValues(Int16Collection myCol)
{
  for (int i = 0; i < myCol.Count; i++)
  Console.WriteLine("   [{0}]:   {1}", i, myCol[i]);
  Console.WriteLine();
}

// Использование оператора foreach, который полностью
// прячет enumerator
public static void PrintValues1(Int16Collection myCol)
{
  foreach (Int16 i16 in myCol)
  Console.WriteLine("   {0}", i16);
  Console.WriteLine();
}

// Использование enumerator
public static void PrintValues2(Int16Collection myCol)
```

```
    {
      System.Collections.IEnumerator myEnumerator =
        myCol.GetEnumerator();
      while (myEnumerator.MoveNext()) // Переход к
                                      // следующему элементу
        Console.WriteLine("   {0}", myEnumerator.Current);
          // Текущий элемент
      Console.WriteLine();
    }
  }
}
```

Вспомним, что в листинге 12.1 класс `Animal` создавался нами самостоятельно как абстрактный класс. Но мы могли его унаследовать от `System.Collections.CollectionBase`, который нам может предоставить свои методы `Clear()`, `RemoveAt()`, `Remove()` и свойство `Count`. Вот как мог бы выглядеть класс `Animal`:

```
public class Animals : CollectionBase
{
  public void Add(Animal newAnimal)
  { List.Add(newAnimal); }

  public void Remove(Animal oldAnimal)
  { List.Remove(oldAnimal); }

  public Animals()
  {}  // Конструктор
}
```

Здесь `Add()` и `Remove()` уже определены внутри класса, т. е., как говорят, строго типизированы (принадлежат данному типу) и сами применяют внутри себя стандартные методы `Add()` и `Remove()`. Теперь они будут работать только с классом `Animals` и с производными от него классами.

Доступ по индексу к элементам коллекции возможен только с использованием так называемого индексатора, с объявлением которого мы встретились в программе листинга 12.3. *Индексатор* — это свойство специального вида, которое можно добавлять в класс. Вот как объявлен индексатор в вышеназванной программе:

```
public Int16 this[ int index ]
{ get  { return((Int16) List[index]); }
  set  { List[index] = value; }
}
```

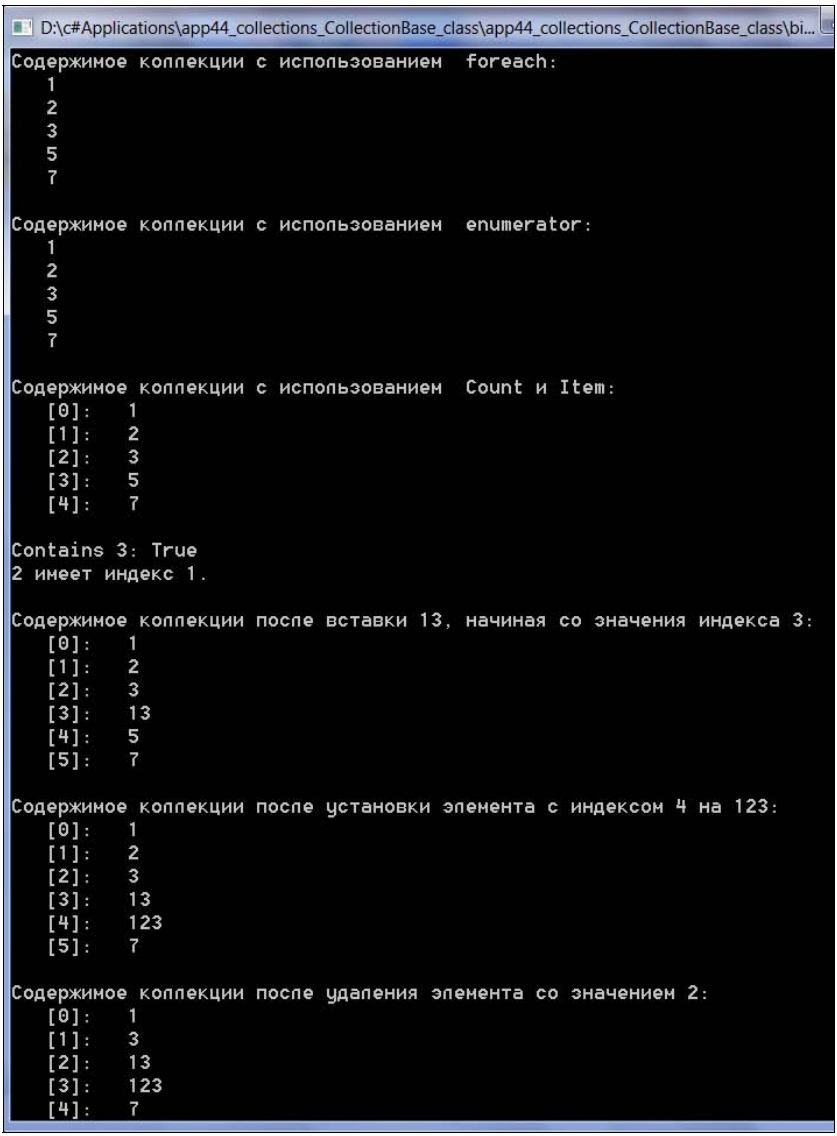


Рис. 12.3. Демонстрация работы методов класса `CollectionBase`

Здесь задан простой числовой индекс, хотя в квадратных скобках можно задавать и более сложные типы. Но числовой индекс — наиболее часто употребляем. После задания индекса мы в программе листинга 12.3 записали:

```
myI16[4] = 123;
```

И все прошло хорошо. Если бы не определили индексатор, компилятор в этом месте выдал бы ошибку.

Интерфейс *IDictionary*

Когда мы создавали класс, наследуя его от класса `CollectionBase`, мы вместе с наследованием получали доступ к интерфейсу `IList`, который обеспечивал нам работу с коллекцией по индексу. Однако существует возможность (и это часто требуется в приложениях) работы с коллекциями по ключу, в качестве которого может выступать, например, строка. Как и в случае индексированных коллекций, существует базовый класс `DictionaryBase`, который предоставляет средства для работы в коллекциях с ключами, а не с индексами. Как и класс `CollectionBase`, этот класс имеет свои свойства и методы из поставляемых ему интерфейсами, которые в него входят. Свойства и методы класса `DictionaryBase`, который поставляет интерфейс `IDictionary`, приведены в табл. 12.3—12.5.

Таблица 12.3. *Свойства класса* `DictionaryBase`

Свойство	Описание
`Count`	Содержит количество элементов экземпляра `DictionaryBase`
`Dictionary`	Содержит список элементов экземпляра `DictionaryBase`

Таблица 12.4. *Методы класса* `DictionaryBase`

Метод	Описание
`Clear`	Очищает экземпляр класса `DictionaryBase`
`CopyTo`	Копирует элементы экземпляра `DictionaryBase` в одномерный массив `Array` с указанного индекса
`Equals(Object)`	Проверяет, равен ли указанный объект текущему

Таблица 12.4 *(окончание)*

Метод	Описание
`Finalize`	Позволяет объекту в теле `try to` освобождать ресурсы и выполнять другие завершающие операции перед началом сборки мусора в куче
`GetEnumerator`	Возвращает интерфейс `IDictionaryEnumerator`, который может проходить по всем элементам экземпляра `DictionaryBase`
`GetType`	Выдает тип текущего экземпляра `DictionaryBase` (класс, интерфейс, массив, значение, перечисление, параметр, обобщение (об этом см. далее), открытое или закрытое обобщение
`OnClear`	Обработчик события "Очистка экземпляра `DictionaryBase`". Запускается перед очисткой. В этом методе пользователь может выполнять свои операции в теле обработчика
`OnClearComplete`	То же, что и `OnClear`, но обработчик запускается после очистки
`OnGet`	Обработчик события "Выдать пару ключ — значение". Включается перед выдачей. В этом методе пользователь может выполнять свои операции в теле обработчика. Например, приведение значения к заданному типу
`OnInsert`	Обработчик события "Вставка в экземпляр `DictionaryBase` класса нового элемента". Запускается перед вставкой. В этом методе пользователь может выполнять свои операции в теле обработчика
`OnInsertComplete`	То же, что и `OnInsert`, но обработчик запускается после вставки
`OnRemove`	То же, что и `OnInsert`, но для удаления элемента
`OnRemoveComplete`	То же, что и `OnInsertComplete`, но после удаления элемента
`OnSet`	То же, что и `OnRemove`, но для присвоения элементу значения
`OnSetComplete`	То же, что и `OnSet`, но после присвоения элементу значения
`ToString`	Возвращает строку, представляющую текущий объект

Таблица 12.5. *Явные внедрения интерфейсов*

Интерфейс	Описания
IDictionary.Add	Добавляет в экземпляр DictionaryBase элемент с указанными ключом и значением
IDictionary.Contains	Проверяет, содержит ли экземпляр DictionaryBase указанный ключ
IDictionary.IsFixedSize	Выдает значение, показывающее, имеет ли экземпляр DictionaryBase фиксированный размер
IDictionary.IsReadOnly	Выдает значение, показывающее, имеет ли экземпляр DictionaryBase атрибут "Только для чтения" object
IDictionary.Item	Выдает или устанавливает значение по указанному ключу в экземпляре DictionaryBase
IDictionary.Keys	Выдает объект ICollection, содержащий ключи из объекта DictionaryBase
IDictionary.Remove	Удаляет элемент из объекта DictionaryBase по указанному ключу
IDictionary.Values	Выдает объект ICollection, содержащий значения из объекта DictionaryBase
IEnumerable.GetEnumerator	Возвращает IEnumerator, который позволяет двигаться по экземпляру DictionaryBase

Применение интерфейса для создания коллекции и поиска в ней по ключу показано в приложении, приведенном в листинге 12.4. Результат представлен на рис. 12.4.

Листинг 12.4

```
/* Created by SharpDevelop.
 * User: user
 * Date: 11.12.2012
 * Time: 18:20
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Collections;
```

```
namespace app45_IDictionary
{

  public class SimpleDictionary : IDictionary
  { // Словарь (коллекция, массив элементов)
    private DictionaryEntry[] items;
    private Int32 ItemsInUse = 0; // Поле

    // Конструктор, задающий определенное
    // количество элементов
    public SimpleDictionary(Int32 numItems)
    {
      items = new DictionaryEntry[numItems]; // Присвоение
                                        // массивов возможно
    }

    #region IDictionary Members // Специальная область,
            // позволяющая редактору среды
            // сворачивать, разворачивать ее содержимое
            // для удобства пользования (начало)
    public bool IsReadOnly
    { get { return false; } }   // Свойство

    // Метод проверяет, содержится ли в экземпляре
    // заданный ключ
    public bool Contains(object key)
    {
      Int32 index;
      return TryGetIndexOfKey(key, out index); // Метод
               // проверяет, содержится ли в экземпляре
               // заданный ключ
    }

    // Свойство показывает, имеет ли объект
    // фиксированный размер
    public bool IsFixedSize { get { return false; } }

    // Метод удаляет из объекта элемент с заданным ключом
    public void Remove(object key)
    {
      if (key == null)
```

```
        throw new ArgumentNullException("key"); // Ключ
                                                // не найден

      // Попытка найти ключ в массиве DictionaryEntry
      Int32 index;
      if (TryGetIndexOfKey(key, out index))
      {
        // Если ключ найден, просмотреть все элементы
        Array.Copy(items, index + 1, items, index,
                   ItemsInUse — index — 1);
        ItemsInUse--;
      }
      else
      { // Если ключ не найден – выход }
    }

    // Член интерфейса IDictionary
    /* По правилам C# все члены подключаемого к классу
       интерфейса должны быть определены в классе.
       Здесь этот член и определяется: формально, хотя
       в расчетах не участвует: */

    public void Clear() { ItemsInUse = 0; }

    // Метод. Добавление новой пары "ключ — значение",
    // даже если такой ключ существует в массиве
    public void Add(object key, object value)
    {
      // Добавление новой пары "ключ — значение",
      // даже если такой ключ существует в массиве
      if (ItemsInUse == items.Length)
      throw new InvalidOperationException("Массив не может
содержать больше элементов");
      items[ItemsInUse++] = new DictionaryEntry(key,
                                                value);
    }

    public ICollection Keys
    { get
      { // Возвращает массив ключей:
        /* В поле ItemsInUse к этому моменту расчета
           находится количество элементов словаря*/
```

```
    Object[] keys = new Object[ItemsInUse];
    for (Int32 n = 0; n < ItemsInUse; n++)
      keys[n] = items[n].Key;
    return keys;
  }
}

// Свойство, как и Keys: в нем отдельно формируются
// значения словаря, т. е. словарь разбивается
// на два отдельных массива — ключи и значения
public ICollection Values
{ get
  { // Возвращает массив значений
    Object[] values = new Object[ItemsInUse];
    for (Int32 n = 0; n < ItemsInUse; n++)
      values[n] = items[n].Value;
    return values;
  }
}
// Задание индексатора, чтобы можно было искать
// элемент массива по его индексу: он участвует
// в поиске по индексу неявно. Как только встречается
// переменная с индексом, управление передается
// индексатору для ее вычисления

public object this[object key]
{ get
  { // Если такой ключ есть в массиве, выдать
    // соответствующее ему значение
    Int32 index;
    if (TryGetIndexOfKey(key, out index))
    { // Ключ найден, возврат соответствующего
      // значения
      return items[index].Value;
    }
    else
    { // Ключ не найден
      return null;
    }
  }
```

```
  set
  { // Если заданный ключ имеется в массиве, изменить
    // соответствующее ему значение
    Int32 index;
    if (TryGetIndexOfKey(key, out index))
    { // Если ключ найден, изменить
      // его значение
      items[index].Value = value;
    }
    else
    { // Если ключ не найден, добавить в массив пару
      // "ключ — значение"
      Add(key, value);
    }
  }
}
private Boolean TryGetIndexOfKey(Object key,
                                 out Int32 index)
{
  for (index = 0; index < ItemsInUse; index++)
  { // Если ключ найден, вернуть true (индекс также
    // возвращается через параметр метода)
    if (items[index].Key.Equals(key)) return true;
  }

  // Ключ не найден, возврат false (index должен быть
  // проигнорирован в вызывающем методе).
  return false;
}
private class SimpleDictionaryEnumerator :
               IDictionaryEnumerator
{
  // Копирование пар из объекта SimpleDictionary
  DictionaryEntry[] items;
  Int32 index = -1;

  public SimpleDictionaryEnumerator(SimpleDictionary sd)
  {
  // Создание копии DictionaryEntry в простом массиве
    items = new DictionaryEntry[sd.Count];
    Array.Copy(sd.items, 0, items, 0, sd.Count);
  }
```

```
    // Возврат текущего элемента
    public Object Current
    { get { ValidateIndex(); return items[index]; } }

    // Возврат текущего входа
    public DictionaryEntry Entry
    { get { return(DictionaryEntry) Current; }
  }

  // Возврат ключа текущего элемента
  public Object Key
  { get { ValidateIndex(); return items[index].Key; }}

  // Возврат значения текущего элемента
  public Object Value
 { get { ValidateIndex(); return items[index].Value; }}

  // Переход к следующему элементу
  public Boolean MoveNext()
  {
    if (index < items.Length - 1)
    { index++; return true; }
    return false;
  }

  // Проверка на действительность индекса перечислителя
  // и выдача исключения, если индекс находится
  // вне границ
  private void ValidateIndex()
  {
    if (index < 0 || index >= items.Length)
    throw new InvalidOperationException("Перечислитель"+
                      " вне границ коллекции");
  }

  // Восстановление индекса для перезапуска
  // перечислителя
  public void Reset()
  { index = -1; }
}
```

```
  public IDictionaryEnumerator GetEnumerator()
  { // Создание перечислителя
    return new SimpleDictionaryEnumerator(this);
  }
  #endregion // Специальная область, позволяющая
             // редактору среды сворачивать,
             // разворачивать ее содержимое для удобства
             // пользования (конец области)

  // Члены интерфейса ICollection: должны быть
  // обязательно определены в производном классе
  // (таковы правила C#), хотя могут и не участвовать
  // в расчетах

  #region ICollection Members
  public bool IsSynchronized
    { get { return false; } }
  public object SyncRoot
    { get { throw new NotImplementedException(); } }
  public int Count { get { return ItemsInUse; } }
  public void CopyTo(Array array, int index)
    { throw new NotImplementedException(); }
  #endregion

  #region IEnumerable Members  // Члены интерфейса
                               // "Перечислитель"
  IEnumerator IEnumerable.GetEnumerator()
  { // Создание перечислителя
    return ((IDictionary)this).GetEnumerator();
  }
  #endregion
}

public sealed class App
{
  static void Main()
  {
    // Создание словаря, содержащего не более трех входов
    IDictionary d = new SimpleDictionary(3);

    // Добавление трех человек с их возрастами к словарю.
    d.Add("Иван", 40);
```

```
    d.Add("Петр", 34);
    d.Add("Андрей", 1);

    Console.WriteLine("Количество элементов в словаре = {0}",
d.Count);

    Console.WriteLine("Содержит ли словарь 'Ивана'? {0}",
d.Contains("Иван"));

    /* Здесь срабатывает индексатор: как только
       встречается переменная с индексом (в данном случае
       d["Иван"]), управление передается индексатору,
       который и вычисляет значение переменной с индексом.
       Индекс должен быть заданного типа (в нашем случае
       это строка символов)) */

    Console.WriteLine("Возраст Ивана: {0}", d["Иван"]);

    // Вывод каждого входного ключа и к нему значения
    foreach (DictionaryEntry de in d)
    {
      Console.WriteLine("{0}: Возраст равен   {1} годам",
                        de.Key, de.Value);
    }

    // Удаление одного существующего входа
    Console.WriteLine("Удаление входа с ключом 'Иван'");
    d.Remove("Иван");

    // Удаление несуществующего входа без выдачи
    // исключения.
    Console.WriteLine("Удаление несуществующего " +
                      "входа с ключом 'Макс'");
    d.Remove("Макс");

    // Вывод имен (это ключи) людей в словаре.
    // Keys — свойство класса, в котором формируются
    // отдельно ключи словаря

    Console.WriteLine("Вывод ключей словаря");
    foreach (String s in d.Keys)
      Console.WriteLine(s);
```

```
    // Вывод возрастов (это значения) людей в словаре
    Console.WriteLine("Вывод значений словаря");
    foreach (Int32 age in d.Values)
      Console.WriteLine(age);
    Console.Read();
    }
  }
}
```

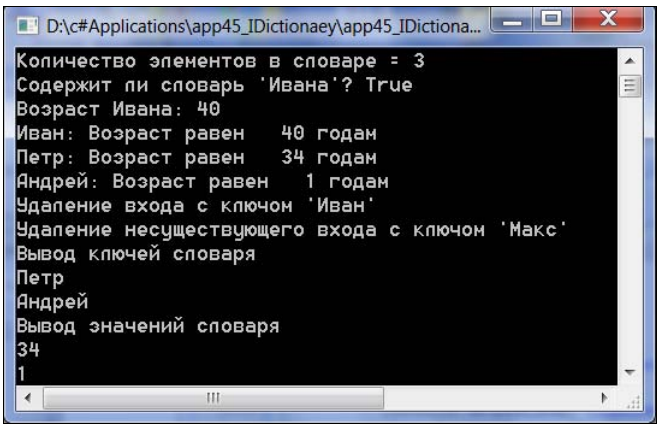


Рис. 12.4. Создание и обработка словаря (результат)

В программе создается класс `SimpleDictionary` из интерфейса `IDictionary`. В классе задаются поля

```
private DictionaryEntry[] items;
private Int32 ItemsInUse = 0;
```

Поле `items` — это обычный массив. Количество его элементов определяется конструктором. Поле `ItemsInUse` будет играть роль индекса массива. Далее в самом классе определяются (задаются) члены интерфейса `IDictionary`. По правилам C# члены подключаемых к классу интерфейсов должны получить свою конкретную реализацию в классе, который их наследует, потому что в интерфейсе такие члены заданы не конкретно, а абстрактно. Здесь же определяется индексатор — свойство, которое вызывается из основной программы, как только она начинает выполнять поиск элемента массива по индексу. Индексатор как раз физически и осуществляет эту работу. В этом же блоке создается вспомогательный класс `SimpleDictionaryEnumerator` путем наследования ин-

терфейса `IDictionaryEnumerator`, члены которого определяются в этом классе и в дальнейшем послужат работе оператора цикла `foreach`. Затем определяются члены интерфейса `ICollection`, т. к. некоторые члены `IDictionary` имеют тип `ICollection`. В частности, свойства `Keys` (ключи) и `Values` (значения). И, наконец, определяется единственный член интерфейса `IEnumerable`, который создает перечислитель типа `IEnumerator`, необходимый для работы оператора цикла `foreach`.

Далее работает основная программа. В ней конструктором создается объект `d` — массив из трех элементов, который затем наполняется конкретными элементами. Это пары типа "ключ — значение". То есть обычный словарь. Ключом служат имена людей, значениями — их возраст. Добавка в массив происходит методом `add()` — членом интерфейса `IDictionary`. Членами этого же интерфейса выполняются действия по выводу на экран количества элементов в словаре, проверка, содержится ли ключ "Иван" в словаре, и определяется возраст человека с именем Иван. В последнем случае поиск строки с ключом "Иван" происходит по индексу. Именно в этом месте основная программа вызывает свойство "индексатор", чтобы с его помощью найти соответствующую строку в словаре: индексатор ищет в словаре с помощью метода `TryGetIndexOfKey()` по ключу соответствующую строку, определяет ее индекс в массиве и по индексу извлекает из массива значение элемента, т. е. возраст. После этого в основной программе демонстрируется работа члена интерфейса `IDictionary` по удалению строк из словаря.

Далее в основной программе выводится на экран содержимое словаря парами "ключ — значение". Здесь применяется оператор `foreach`, который и требует наличия двух интерфейсов — `IEnumerator` и `IEnumerable`. Детальные комментарии даны по тексту программы. Однако для лучшего понимания механизма работы всех элементов программы рекомендуется ее прокрутить в режиме отладки, начиная с первой строки основной программы. Напомним, что при пошаговой отладке продвижение по операторам выполняется нажатием клавиши <F10>, вход вовнутрь строки (если это вызов метода) — клавишей <F11>, дальнейшее продвижение по строкам метода — <F10> или <F11>, исполнение всех команд до следующей точки останова — клавишей <F5>. Просмотр содержимого переменных осуществляется наведением курсора мыши на имя переменной: через некоторое время появятся результаты, выдаваемые подсказчиком среды разработки.

Итератор

Рассмотрев пример листинга 12.4, мы видим, как довольно сложно организовывать работу оператора `foreach`, переопределяя методы интерфейсов `IENumerable` и `IEnumerator`. Существует более простой способ выхода из этого положения с помощью так называемого *итератора*, признаком которого является ключевое слово `yield`. Оно и переводится как "выход", т. е. выход из положения, думается. Итератор — это группа команд, обычно оформляемая как метод, который предоставляет для `foreach` по очереди элемент за элементом для просмотра. В роли итератора может выступать и блок команд обращения к некоторому свойству. Пример итератора, оформленного как метод специального типа (должен быть обязательно тип `IEnumerable`) класса, приведен в листинге 12.5, а результат его работы — на рис. 12.5. Чтобы лучше увидеть механизм подачи элементов оператору `foreach` методом `MyEnumerator()` (так он назван в программе, а можно давать ему имя по своему желанию), желательно просмотреть работу программы в режиме пошаговой отладки.

Листинг 12.5

```
/* Created by SharpDevelop.
 * User: user
 * Date: 15.12.2012
 * Time: 13:08
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Collections;

namespace app48_yield
{
  public class A
  {
    static int n;

    public A(int N)
    { n=N; }
    public IEnumerable MyEnumerator()
    { for(int i=0; i<n; i++)
```

```
        yield return("String " +
                     System.Convert.ToString(i));
    }
  } // A

  class Program
  {
    public static void Main()
    {
      Console.WriteLine("Hello World!");
      A a = new A(5);

      foreach (string item in a.MyEnumerator())
        Console.WriteLine(item);

      Console.ReadKey();
    } // Main
  } // Program
}
```

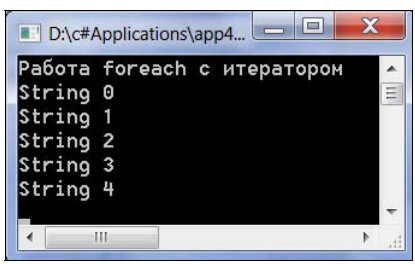


Рис. 12.5. Работа оператора `foreach` с итератором

Наличие ключевого слова `yield`, с которым оператор `return` выводит результат работы итератора `MyEnumerator()`, тоже обязательно. Метод `MyEnumerator()` указан в заголовке `foreach` в качестве множества объектов, из которого происходит выборка объектов по одному.

Получение копий

Для обеспечения копирования объектов имеется интерфейс `ICloneable`, с помощью которого создается новый экземпляр класса с тем же значением, что и у существующего экземпляра класса. У этого интерфейса

есть единственный член — `Clone()`. Он возвращает значение типа `System.Object`. Пример использования интерфейса показан в листинге 12.6, результат работы программы, указанной в листинге, — на рис. 12.6.

Листинг 12.6

```
/* Created by SharpDevelop.
 * User: user
 * Date: 16.12.2012
 * Time: 11:42
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Collections;

namespace app49_Clone
{
  public class A:System.ICloneable
  { // Воспользовались абстрактным классом Array
    // для объявления массива чисел
    public int [] numbers = { 1, 2, 3, 4, 5 };

    // Конструктор
    public A()
    {  }
    // Определение члена интерфейса ICloneable
    public Object Clone()
    { return new A(); }

    // Задание перечислителя для использования foreach
    public IEnumerable MyNumerator_a()
    {
      A a = new A();
      for(int i=0; i < a.numbers.Length; i++)
        yield return a.numbers[i];
    }
  }

  class Program
  {
    public static void Main()
```

```
    {
      Console.WriteLine("Клонирование объекта");
      Console.WriteLine("Объект до клонирования:");
      A a = new A();
      foreach(object o in a.MyNumerator_a())
        Console.WriteLine("{0}",o);

      Console.WriteLine("Объект после клонирования:");
      A b = (A)a.Clone();
      for(int i=0; i < b.numbers.Length; i++)
        Console.WriteLine("{0}", b.numbers[i]);

      Console.Write("Press any key to continue...");
      Console.ReadKey(true);
    }
  }
}
```

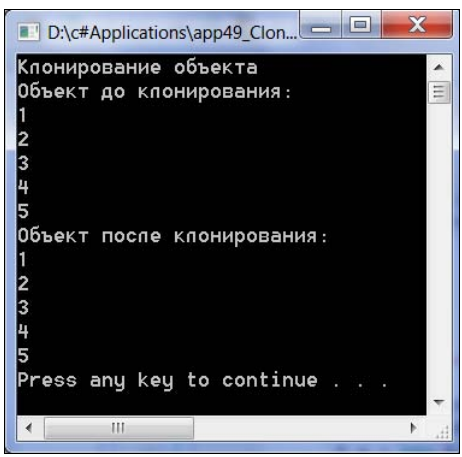


Рис. 12.6. Результат копирования (клонирования) объектов

В программе создается класс `A`, в котором объявляется массив чисел (numbers). Здесь мы воспользовались абстрактным классом `Array` для объявления массива чисел. Задается конструктор по умолчанию (без параметров). Так как к классу `A` подключается интерфейс, то члены этого интерфейса должны быть определены в классе (по определению интерфейса). Поэтому далее в классе `A` определяется единственный член

интерфейса — метод `Clone()`. Он должен возвращать новый объект типа `object`. Поэтому в теле `Clone()` и написан оператор возврата нового объекта класса `A`. Далее идет определение перечислителя, т. к. предполагается использовать оператор `foreach`. Перечислитель должен выдавать для `foreach` поочередно элементы объекта класса `A`. Поэтому в перечислителе с помощью оператора `for` просматриваются все элементы созданного там же в перечислителе объекта `A` и возвращаются с помощью ключевого слова `yield`, что обеспечит для `foreach` передачу по одному элементу из перечислителя. На этом класс `A` завершается. А далее идет основная программа, в которой сначала выдается на экран по `foreach` объект `a`, затем применяется метод клонирования к объекту `a`, и результат приводится к типу класса `A`, т. к. результат клонирования появляется с типом `object`, а нам надо этот результат присвоить переменной `b` тоже типа `A` (мы должны проверить результат клонирования, выдавая `b` на экран). После этого `b` выдается на экран обычным оператором `for`.

Классы *Array* и *List<T>*

Эти два класса создают удобства работы с массивами.

Класс *Array*

Предоставляет методы для создания, изменения, поиска и сортировки массивов, т. е. выступает в роли базового класса для всех массивов. Однако этот абстрактный класс могут наследовать только сама система и компилятор. Пользователи же могут создавать объекты этого класса, применять его инструменты и создавать свои массивы так, как позволяют им средства языка программирования. Кстати, в отличие от других классов, класс `Array` не имеет конструктора, а его объект создается специальным методом класса с именем `CreateInstance()`. Некоторые элементы класса приведены в табл. 12.6—12.8. Отметим, что действия элементов не расшифровываются: при необходимости, это можно посмотреть в справке по C#, предоставляемой разработчиком (так называемая система MSDN в Интернете). Элементы приводятся для показа возможности работы с массивами с применением инструментов класса.

Таблица 12.6. *Свойства класса* `Array`

Свойство	Описание
`IsFixedSize`	Получает значение, показывающее, имеет ли список `Array` фиксированный размер
`IsReadOnly`	Получает значение, указывающее, доступен ли объект `Array` только для чтения
`Length`	Получает 32-разрядное целое число, представляющее общее число элементов во всех измерениях массива `Array`
`LongLength`	Получает 64-разрядное целое число, представляющее общее число элементов во всех измерениях массива `Array`
`Rank`	Получает ранг (размерность) объекта `Array`

Таблица 12.7. *Методы класса* `Array`

Метод	Описание
`BinarySearch(Array, Object)`	Выполняет поиск заданного элемента во всем отсортированном одномерном массиве `Array`, используя для этого интерфейс `IComparable`, реализуемый каждым элементом массива `Array` и заданным объектом
`BinarySearch(Array, Object, IComparer)`	Выполняет поиск значения во всем отсортированном одномерном массиве `Array`, используя заданный интерфейс `IComparer`
`BinarySearch(Array, Int32, Int32, Object)`	Выполняет поиск значения в диапазоне элементов отсортированного одномерного массива `Array`, используя для этого интерфейс `IComparable`, реализуемый каждым элементом массива `Array` и заданным значением
`BinarySearch(Array, Int32, Int32, Object, IComparer)`	Выполняет поиск значения в диапазоне элементов отсортированного одномерного массива `Array`, используя заданный интерфейс `IComparer`
`Clear`	Задает диапазон элементов массива `Array` равным нулю, `false` или `null` в зависимости от типа элемента

Таблица 12.7 *(продолжение)*

Метод	Описание
`ConstrainedCopy`	Копирует диапазон элементов из массива `Array`, начиная с заданного индекса источника, и вставляет его в другой массив `Array`, начиная с заданного индекса назначения. Гарантирует, что в случае невозможности успешно скопировать весь диапазон все изменения будут отменены
`ConvertAll<TInput, TOutput>`	Преобразует массив одного типа в массив другого типа
`Copy(Array, Array, Int32)`	Копирует диапазон элементов из массива `Array`, начиная с первого элемента, и вставляет его в другой массив `Array`, также начиная с первого элемента. Длина задается как 32-разрядное целое число
`Copy(Array, Array, Int64)`	Копирует диапазон элементов из массива `Array`, начиная с первого элемента, и вставляет его в другой массив `Array`, также начиная с первого элемента. Длина задается как 64-разрядное целое число
`Copy(Array, Int32, Array, Int32, Int32)`	Копирует диапазон элементов из массива `Array`, начиная с заданного индекса источника, и вставляет его в другой массив `Array`, начиная с заданного индекса назначения. Длина и индексы задаются как 32-разрядные целые числа
`Copy(Array, Int64, Array, Int64, Int64)`	Копирует диапазон элементов из массива `Array`, начиная с заданного индекса источника, и вставляет его в другой массив `Array`, начиная с заданного индекса назначения. Длина и индексы задаются как 64-разрядные целые числа
`CopyTo(Array, Int32)`	Копирует все элементы текущего одномерного массива `Array` в заданный одномерный массив `Array`, начиная с указанного индекса в массиве назначения `Array`. Индекс задается как 32-разрядное целое число
`CopyTo(Array, Int64)`	Копирует все элементы текущего одномерного массива `Array` в заданный одномерный массив `Array`, начиная с указанного индекса в массиве назначения `Array`. Индекс задается как 64-разрядное целое число

Таблица 12.7 *(продолжение)*

Метод	Описание
`CreateInstance(Type, Int32)`	Создает одномерный массив `Array` заданного типа `Type` и длины, индексация которого начинается с нуля
`CreateInstance(Type, Int32[])`	Создает многомерный массив `Array` заданного типа `Type` с заданными длинами по измерениям и индексацией, начинающейся с нуля. Длины по измерениям задаются в массиве 32-разрядных целых чисел
`CreateInstance(Type, Int64[])`	Создает многомерный массив `Array` заданного типа `Type` с заданными длинами по измерениям и индексацией, начинающейся с нуля. Длины по измерениям задаются в массиве 64-разрядных целых чисел
`CreateInstance(Type, Int32, Int32)`	Создает двумерный массив `Array` заданного типа `Type` с заданными длинами по измерениям и индексацией, начинающейся с нуля
`CreateInstance(Type, Int32[], Int32[])`	Создает многомерный массив `Array` заданного типа `Type` с заданными длинами по измерениям и нижними границами
`CreateInstance(Type, Int32, Int32, Int32)`	Создает трехмерный массив `Array` заданного типа `Type` с заданными длинами по измерениям и индексацией, начинающейся с нуля
`Equals(Object)`	Определяет, равен ли заданный объект текущему объекту (унаследовано от `Object`)
`Finalize`	Позволяет объекту попытаться освободить ресурсы и выполнить другие операции очистки перед тем, как объект будет утилизирован в процессе сборки мусора (унаследовано от `Object`)
`Find<T>`	Выполняет поиск элемента, удовлетворяющего условиям указанного предиката, и возвращает первое найденное вхождение в пределах всего списка `Array`
`FindAll<T>`	Извлекает все элементы, удовлетворяющие условиям указанного предиката

Таблица 12.7 *(продолжение)*

Метод	Описание
`FindLast<T>`	Выполняет поиск элемента, удовлетворяющего условиям указанного предиката, и возвращает последнее найденное вхождение в пределах всего списка `Array`
`ForEach<T>`	Выполняет указанное действие с каждым элементом указанного массива
`GetEnumerator`	Возвращает объект `IEnumerator` представления `Array`
`GetLength`	Получает 32-разрядное целое число, представляющее количество элементов в заданном измерении массива `Array`
`GetLongLength`	Получает 64-разрядное целое число, представляющее количество элементов в заданном измерении массива `Array`
`GetLowerBound`	Получает нижнюю границу заданного измерения массива `Array`
`GetUpperBound`	Получает верхнюю границу заданного измерения массива `Array`
`GetValue(Int32)`	Получает значение, хранящееся в указанной позиции одномерного массива `Array`. Индекс задается как 32-разрядное целое число
`GetValue(Int32[])`	Получает значение, хранящееся в указанной позиции многомерного массива `Array`. Индексы задаются как массив 32-разрядных целых чисел
`GetValue(Int64)`	Получает значение, хранящееся в указанной позиции одномерного массива `Array`. Индекс задается как 64-разрядное целое число
`GetValue(Int64[])`	Получает значение, хранящееся в указанной позиции многомерного массива `Array`. Индексы задаются как массив 64-разрядных целых чисел
`IndexOf(Array, Object)`	Выполняет поиск заданного объекта внутри всего одномерного массива `Array` и возвращает индекс его первого вхождения

Таблица 12.7 *(продолжение)*

Метод	Описание
`IndexOf(Array, Object, Int32)`	Выполняет поиск указанного объекта и возвращает индекс первого вхождения в диапазоне элементов одномерного массива `Array`, начинающемся с элемента с заданным индексом и заканчивающемся последним элементом
`IndexOf(Array, Object, Int32, Int32)`	Выполняет поиск указанного объекта и возвращает индекс первого вхождения в диапазоне элементов одномерного массива `Array`, начинающемся с элемента с заданным индексом и содержащем указанное число элементов
`IndexOf<T>(T[], T)`	Выполняет поиск указанного объекта и возвращает индекс первого вхождения во всем массиве `Array`
`Initialize`	Инициализирует каждый элемент массива `Array` типа значения путем вызова конструктора по умолчанию для типа значений
`LastIndexOf(Array, Object)`	Выполняет поиск заданного объекта и возвращает индекс его последнего вхождения внутри всего одномерного массива `Array`
`LastIndexOf(Array, Object, Int32)`	Выполняет поиск указанного объекта и возвращает индекс его последнего вхождения в диапазоне элементов одномерного массива `Array`, начинающемся с первого элемента и заканчивающемся элементом с заданным индексом
`LastIndexOf(Array, Object, Int32, Int32)`	Выполняет поиск указанного объекта и возвращает индекс последнего вхождения в диапазоне элементов одномерного массива `Array`, содержащем указанное число элементов и заканчивающемся элементом с заданным индексом
`Resize<T>`	Изменяет количество элементов в массиве до указанной величины
`Reverse(Array)`	Изменяет порядок элементов во всем одномерном массиве `Array` на обратный
`Reverse(Array, Int32, Int32)`	Изменяет последовательность элементов в диапазоне элементов одномерного массива `Array` на обратную

Таблица 12.7 *(продолжение)*

Метод	Описание
`SetValue(Object, Int32)`	Присваивает значение элементу, находящемуся в указанной позиции одномерного массива `Array`. Индекс задается как 32-разрядное целое число
`Sort(Array)`	Сортирует элементы во всем одномерном массиве `Array`, используя реализацию интерфейса `IComparable` каждого элемента массива `Array`
`Sort(Array, Array)`	Сортирует пару одномерных объектов `Array` (один содержит ключи, а другой — соответствующие элементы) по ключам в первом массиве `Array`, используя реализацию интерфейса `IComparable` каждого ключа
`Sort(Array, IComparer)`	Сортирует элементы в одномерном массиве `Array`, используя заданный интерфейс `IComparer`
`Sort(Array, Array, IComparer)`	Сортирует пару одномерных объектов `Array` (один содержит ключи, а другой — соответствующие элементы) по ключам в первом массиве `Array`, используя заданный интерфейс `IComparer`
`Sort(Array, Int32, Int32)`	Сортирует элементы в диапазоне элементов одномерного массива `Array` с помощью реализации интерфейса `IComparable` каждого элемента массива `Array`
`Sort(Array, Array, Int32, Int32)`	Сортирует диапазон элементов в паре одномерных объектов `Array` (один содержит ключи, а другой — соответствующие элементы) по ключам в первом массиве `Array`, используя реализацию интерфейса `IComparable` каждого ключа
`Sort(Array, Int32, Int32, IComparer)`	Сортирует элементы в диапазоне элементов одномерного массива `Array`, используя заданный интерфейс `IComparer`
`Sort(Array, Array, Int32, Int32, IComparer)`	Сортирует диапазон элементов в паре одномерных объектов `Array` (один содержит ключи, а другой — соответствующие элементы) по ключам в первом массиве `Array`, используя заданный интерфейс `IComparer`

Таблица 12.7 *(окончание)*

Метод	Описание
`Sort<T>(T[])`	Сортирует элементы во всем массиве `Array`, используя реализацию универсального интерфейса `IComparable<T>` каждого элемента массива `Array`
`ToString`	Возвращает строку, которая представляет текущий объект (унаследовано от `Object`)

Таблица 12.8. *Явные реализации интерфейса* `IList`

Интерфейс	Описание
`ICollection.Count`	Получает число элементов, содержащихся в объекте `Array`
`IList.Add`	Добавляет элемент в объект `IList`
`IList.Clear`	Удаляет все элементы из списка `IList`
`IList.Contains`	Определяет, входит ли элемент в состав `IList`
`IList.IndexOf`	Определяет индекс заданного элемента коллекции `IList`
`IList.Insert`	Вставляет элемент в список `IList` по указанному индексу
`IList.IsFixedSize`	Получает значение, показывающее, имеет ли объект `Array` фиксированный размер
`IList.IsReadOnly`	Получает значение, указывающее, доступен ли объект `Array` только для чтения
`IList.Item`	Получает или задает элемент с указанным индексом
`IList.Remove`	Удаляет первый экземпляр указанного объекта из коллекции `IList`
`IList.RemoveAt`	Удаляет элемент `IList` по указанному индексу
`IStructuralComparable.CompareTo`	Определяет позицию текущего объекта коллекции относительно другого объекта в порядке сортировки (находится перед другим объектом, на одной позиции с ним или после другого объекта)

Таблица 12.8 *(окончание)*

Интерфейс	Описание
IStructuralEquatable.Equals	Определяет, равен ли объект текущему экземпляру
IStructuralEquatable.GetHashCode	Возвращает хэш-код текущего экземпляра

Пример создания простых массивов без применения типа `Array` и работа с ними показаны в листинге 12.7. Результат — на рис. 12.7.

Листинг 12.7

```
/* Created by SharpDevelop.
 * User: user
 * Date: 17.12.2012
 * Time: 12:13
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app51_array
{
  class Program
  {
    public static void Main()
    {
      Console.WriteLine("Создание двух простых массивов "+
                        "типа int и object");
      int[] myInt = new int[5] {1, 2, 3, 4, 5};
      Object[] myOb = new Object[5] {26, 27, 28, 29, 30};

      // Печать массивов
      Console.Write("Числа:  ");
      PrintValues(myInt);
      Console.Write("Объекты: ");
      PrintValues(myOb);

      // Копирование первых двух элементов из myInt
      // в myOb: пользуемся инструментами класса Array
      System.Array.Copy(myInt, myOb,2);
```

```
      // Печать значений измененных массивов
      Console.WriteLine("\nПосле копирования первых двух"+
                        " элементов из 1-го массива во 2-й");
      Console.Write("Числа:  ");
      PrintValues(myInt);
      Console.Write("Объекты: ");
      PrintValues(myOb);

      // Копирование последних двух элементов
      // из myOb в myInt
      System.Array.Copy(myOb, myOb.GetUpperBound(0) - 1,
                        myInt, myInt.GetUpperBound(0) - 1, 2);

      // Печать значений измененных массивов
      Console.WriteLine("\nПосле копирования последних "+
             "двух элементов из 2-го массива в 1-й");
      Console.Write("Числа:  ");
      PrintValues(myInt);
      Console.Write("Объекты: " );
      PrintValues(myOb);

      Console.Read();
    } // Main

    // Методы печати
    public static void PrintValues(Object[] myArr)
    {
      foreach (Object i in myArr)
      { Console.Write("\t{0}", i); }
      Console.WriteLine();
    }

    public static void PrintValues(int[] myArr)
    {
      foreach (int i in myArr)
      { Console.Write("\t{0}", i); }
      Console.WriteLine();
    }
  }
}
```

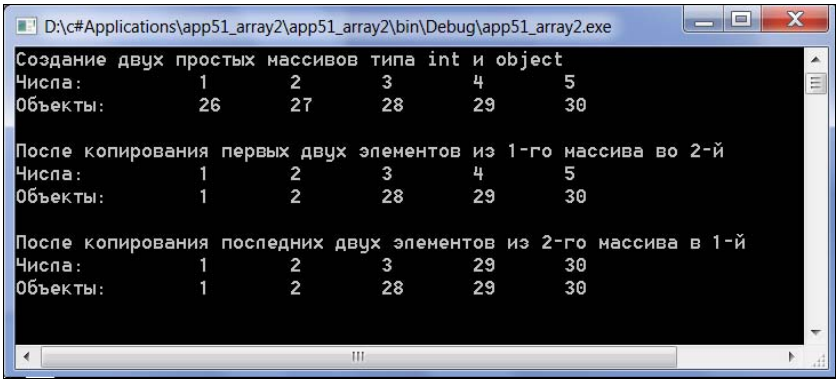


Рис. 12.7. Применение методов класса `Array`

Пример объявления массива с использованием класса `Array` и работа с массивом показаны в листинге 12.8, а результат — на рис. 12.8.

Листинг 12.8

```
/* Created by SharpDevelop.
 * User: user
 * Date: 17.12.2012
 * Time: 13:10
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app52_array
{
  class Program
  {
    public static void Main()
    {
    // Создание и инициализация трехмерного числового
    // массива Int32.
    Array myArr = Array.CreateInstance(typeof(Int32), 2, 3, 4);
// 2, 3, 4 — длины массива по его размерностям
    /* Оператор typeof переводит свой аргумент в объект
       класса Type, который требуется в качестве
       параметра в CreateInstance*/
```

```
    // Метод GetLowerBound(0) возвращает нижнюю границу
    // индексов первого измерения массива Array,
    // а метод GetLowerBound(Rank – 1) – нижнюю
    // границу последнего измерения массива Array.
    for (int i = myArr.GetLowerBound(0); i <=
myArr.GetUpperBound(0); i++)  // для 1-го измерения
      for (int j = myArr.GetLowerBound(1); j <=
myArr.GetUpperBound(1); j++)  // для 2-го измерения
        for (int k = myArr.GetLowerBound(2); k <=
myArr.GetUpperBound(2); k++)  // для 3-го измерения
        {
          // Установка значений элементов массива
          // по его индексам:
          myArr.SetValue((12*i)+(4*j)+k, i, j, k);
        }

    /* Если это элементы по каждой размерности:
       1,2; 1,2,3;  1,2,3,4, то картина трехмерного
       массива такая:
          1,1,1,  1,1,2,  1,1,3,  1,1,4
          1,2,1,  1,2,2,  1,2,3,  1,2,4
          1,3,1,  1,3,2,  1,3,3,  1,3,4
          2,1,1,  2,1,2,  2,1,3,  2,1,4
          2,2,1,  2,2,2,  2,2,3,  2,2,4
          2,3,1   2,3,2,  2,3,3,  2,3,4
    */

    // Печать свойств массива типа Array.
    Console.WriteLine("Массив имеет размерность {0}  и элементов
{1}  ", myArr.Rank, myArr.Length);
    Console.WriteLine("\tДлина\tНижняя граница\tВерхняя граница");
    for (int i = 0; i < myArr.Rank; i++)  // Индекс
              // меняется от нуля до размерности – 1
    {
       Console.Write("{0}:\t{1}", i, myArr.GetLength(i));
       Console.WriteLine("\t\t{0}\t\t{1}",
                         myArr.GetLowerBound(i),
                         myArr.GetUpperBound(i));
    }

    // Печать содержимого массива
    Console.WriteLine("Значения элементов массива:");
```

```
    PrintValues(myArr);
    Console.Read();
  }

  public static void PrintValues(Array myArr)
  {
    // Интерфейс IEnumerable уже входит в класс Array,
    // поэтому для создания перечислителя его к классу
    // подключать не надо:
    System.Collections.IEnumerator myEnumerator =
          myArr.GetEnumerator();
    int i = 0;
    int cols = myArr.GetLength(myArr.Rank - 1);
    // Количество колонок данных: GetLength(0) дает кол-во
    // элементов в первом измерении, поэтому стоит Rank-1.
    // Когда перечислитель выйдет за пределы массива,
    // он выдаст false, и цикл закончится:
    while (myEnumerator.MoveNext())  // Первое движение
          // MoveNext попадает на первый элемент,
          // поэтому i идет синхронно: 1, 2
    { if (i < cols)
      { i++;  // Вывод 4-х элементов очередной строки
      }
      else
      { Console.WriteLine();
        i = 1; // Возврат счетчика на начало вывода
               // следующей строки
      }
    Console.Write("\t{0}", myEnumerator.Current); // Вывод
         // текущего элемента массива (проход по строкам)
    } // while
    Console.WriteLine();
    }
  }
}
```

Работа программы пояснена по тексту.

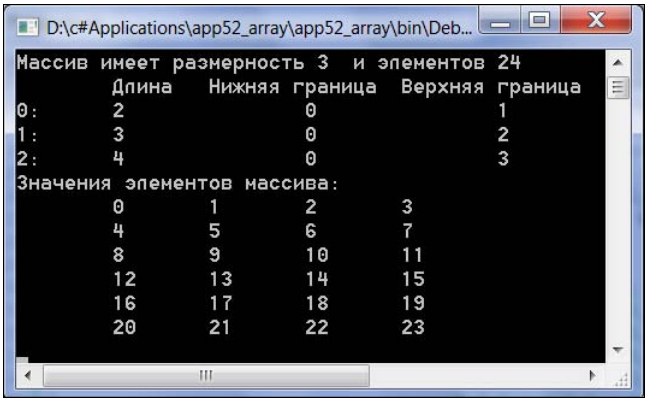


Рис. 12.8. Работа с массивом типа `Array`

Класс *List<T>*

Представляет собой строго типизированный список объектов (<T> указывает на тип элементов в списке), доступных по индексу. Поддерживает методы для поиска по списку, выполнения сортировки и других операций со списками. Для работы с классом подключается пространство имен `System.Collections.Generic.List<T>`. Содержит в себе интерфейсы `IList<T>`, `ICollection<T>`, `IList`, `ICollection`, `IReadOnlyList<T>`, `IReadOnlyCollection<T>`, `IEnumerable<T>`, `IEnumerable`, инструментами которых можно пользоваться. Члены класса представлены в табл. 12.9—12.12. Отметим, что действия элементов не расшифровываются: при необходимости это можно посмотреть в справке по C#, предоставляемой разработчиком (система MSDN в Интернете). Элементы приводятся для показа возможности работы с массивами с применением инструментов класса.

Таблица 12.9. *Конструкторы класса `List<T>`*

Конструктор	Описание
`List<T>()`	Инициализирует новый пустой экземпляр класса `List<T>` с начальным содержанием по умолчанию
`List<T>(IEnumerable<T>)`	Инициализирует новый экземпляр `List<T>`, который содержит элементы, скопированные из указанной коллекции, и имеет емкость, достаточную для размещения всех скопированных элементов

Таблица 12.9 *(окончание)*

Конструктор	Описание
`List<T>(Int32)`	Инициализирует новый пустой экземпляр класса `List<T>` с указанной начальной емкостью

Таблица 12.10. *Свойства класса* `List<T>`

Свойство	Описание
`Capacity`	Возвращает или задает общее число элементов, которые может вместить внутренняя структура данных без изменения размера
`Count`	Возвращает число элементов, которые фактически содержатся в коллекции `List<T>`
`Item`	Получает или задает элемент с указанным индексом

Таблица 12.11. *Методы класса* `List<T>`

Метод	Описание
`Add`	Добавляет объект в конец коллекции `List<T>`
`AddRange`	Добавляет элементы указанной коллекции в конец списка `List<T>`
`AsReadOnly`	Возвращает для текущей коллекции оболочку `IList<T>`, доступную только для чтения
`BinarySearch(T)`	Выполняет поиск элемента по всему отсортированному списку `List<T>`, используя метод сравнения по умолчанию, и возвращает индекс элемента, отсчитываемый от нуля
`BinarySearch(T, IComparer<T>)`	Выполняет поиск элемента по всему отсортированному списку `List<T>`, используя указанный метод сравнения, и возвращает индекс элемента, отсчитываемый от нуля
`BinarySearch(Int32, Int32, T, IComparer<T>)`	Выполняет поиск элемента в диапазоне элементов отсортированного списка `List<T>`, используя указанный метод сравнения, и возвращает индекс элемента, отсчитываемый от нуля

Таблица 12.11 *(продолжение)*

Метод	Описание
`Clear`	Удаляет все элементы из коллекции `List<T>`
`Contains`	Определяет, входит ли элемент в состав `List<T>`
`ConvertAll<TOutput>`	Преобразует элементы текущего списка `List<T>` в другой тип и возвращает список преобразованных элементов
`CopyTo(T[])`	Копирует весь список `List<T>` в совместимый одномерный массив, начиная с первого элемента целевого массива
`CopyTo(T[], Int32)`	Копирует `List<T>` целиком в совместимый одномерный массив, начиная с указанного индекса конечного массива
`CopyTo(Int32, T[], Int32, Int32)`	Копирует диапазон элементов из списка `List<T>` в совместимый одномерный массив, начиная с указанного индекса конечного массива
`Equals(Object)`	Определяет, равен ли заданный объект текущему объекту (унаследовано от `Object`)
`Exists`	Определяет, содержит ли `List<T>` элементы, удовлетворяющие условиям указанного предиката
`Finalize`	Позволяет объекту попытаться освободить ресурсы и выполнить другие операции очистки перед тем, как объект будет утилизирован в процессе сборки мусора
`Find`	Выполняет поиск элемента, удовлетворяющего условиям указанного предиката, и возвращает первое найденное вхождение в пределах всего списка `List<T>`
`FindAll`	Извлекает все элементы, удовлетворяющие условиям указанного предиката
`FindIndex(Predicate<T>)`	Выполняет поиск элемента, удовлетворяющего условиям указанного предиката, и возвращает отсчитываемый от нуля индекс первого найденного вхождения в пределах всего списка `List<T>`

Таблица 12.11 *(продолжение)*

Метод	Описание
`FindIndex(Int32, Predicate<T>)`	Выполняет поиск элемента, удовлетворяющего условиям указанного предиката, и возвращает отсчитываемый от нуля индекс первого вхождения в диапазоне элементов списка `List<T>`, начиная с заданного индекса и заканчивая последним элементом
`FindIndex(Int32, Int32, Predicate<T>)`	Выполняет поиск элемента, удовлетворяющего условиям указанного предиката, и возвращает отсчитываемый от нуля индекс первого вхождения в диапазоне элементов списка `List<T>`, начинающемся с заданного индекса и содержащем указанное число элементов
`FindLast`	Выполняет поиск элемента, удовлетворяющего условиям указанного предиката, и возвращает последнее найденное вхождение в пределах всего списка `List<T>`
`FindLastIndex(Predicate<T>)`	Выполняет поиск элемента, удовлетворяющего условиям указанного предиката, и возвращает отсчитываемый от нуля индекс последнего найденного вхождения в пределах всего списка `List<T>`
`FindLastIndex(Int32, Predicate<T>)`	Выполняет поиск элемента, удовлетворяющего условиям указанного предиката, и возвращает отсчитываемый от нуля индекс последнего вхождения в диапазоне элементов списка `List<T>`, начиная с первого элемента и заканчивая элементом с заданным индексом
`FindLastIndex(Int32, Int32, Predicate<T>)`	Выполняет поиск элемента, удовлетворяющего условиям указанного предиката, и возвращает отсчитываемый от нуля индекс последнего вхождения в диапазоне элементов списка `List<T>`, содержащем указанное число элементов и заканчивающемся элементом с заданным индексом
`ForEach`	Выполняет указанное действие с каждым элементом списка `List<T>`
`GetEnumerator`	Возвращает перечислитель, осуществляющий перебор элементов списка `List<T>`

Таблица 12.11 *(продолжение)*

Метод	Описание
`GetRange`	Создает неполную копию диапазона элементов исходного списка `List<T>`
`GetType`	Возвращает объект `Type` для текущего экземпляра
`IndexOf(T)`	Осуществляет поиск указанного объекта и возвращает отсчитываемый от нуля индекс первого вхождения, найденного в пределах всего списка `List<T>`
`IndexOf(T, Int32)`	Осуществляет поиск указанного объекта и возвращает отсчитываемый от нуля индекс первого вхождения в диапазоне элементов списка `List<T>`, начиная с заданного индекса и до последнего элемента
`IndexOf(T, Int32, Int32)`	Выполняет поиск указанного объекта и возвращает отсчитываемый от нуля индекс первого вхождения в диапазоне элементов списка `List<T>`, начинающемся с заданного индекса и содержащем указанное число элементов
`Insert`	Добавляет элемент в список `List<T>` в позиции с указанным индексом
`InsertRange`	Вставляет элементы коллекции в список `List<T>` в позиции с указанным индексом
`LastIndexOf(T)`	Осуществляет поиск указанного объекта и возвращает отсчитываемый от нуля индекс последнего вхождения, найденного в пределах всего списка `List<T>`
`LastIndexOf(T, Int32)`	Осуществляет поиск указанного объекта и возвращает отсчитываемый от нуля индекс последнего вхождения в диапазоне элементов списка `List<T>`, начиная с первого элемента и до позиции с заданным индексом
`LastIndexOf(T, Int32, Int32)`	Выполняет поиск указанного объекта и возвращает отсчитываемый от нуля индекс последнего вхождения в диапазоне элементов списка `List<T>`, содержащем указанное число элементов и заканчивающемся в позиции с указанным индексом

Таблица 12.11 *(окончание)*

Метод	Описание
MemberwiseClone	Создает неполную копию текущего объекта Object
Remove	Удаляет первое вхождение указанного объекта из коллекции List<T>
RemoveAll	Удаляет все элементы, удовлетворяющие условиям указанного предиката
RemoveAt	Удаляет элемент списка List<T> с указанным индексом
RemoveRange	Удаляет диапазон элементов из списка List<T>
Reverse()	Изменяет порядок элементов во всем списке List<T> на обратный
Reverse(Int32, Int32)	Изменяет порядок элементов в указанном диапазоне
Sort()	Сортирует элементы во всем списке List<T> с помощью метода сравнения по умолчанию
Sort(IComparer<T>)	Сортирует элементы во всем списке List<T> с помощью указанного метода сравнения
Sort(Int32, Int32, IComparer<T>)	Сортирует элементы в диапазоне элементов списка List<T> с помощью указанного метода сравнения
ToArray	Копирует элементы списка List<T> в новый массив
ToString	Возвращает строку, которая представляет текущий объект
TrimExcess	Задает емкость, равную фактическому числу элементов в списке List<T>, если это число меньше порогового значения
TrueForAll	Определяет, все ли элементы списка List<T> удовлетворяют условиям указанного предиката

Таблица 12.12. *Явные реализации интерфейсов*

Интерфейс	Описание
`ICollection.CopyTo`	Копирует элементы коллекции `ICollection` в массив `Array`, начиная с указанного индекса массива `Array`
`ICollection<T>.IsReadOnly`	Получает значение, указывающее, доступна ли коллекция `ICollection<T>` только для чтения
`IEnumerable<T>.GetEnumerator`	Возвращает перечислитель, осуществляющий итерацию в коллекции
`IEnumerable.GetEnumerator`	Возвращает перечислитель, осуществляющий итерацию в коллекции
`IList.Add`	Добавляет элемент в список `IList` (этот интерфейс рассмотрен далее)
`IList.Contains`	Определяет, содержится ли указанное значение в списке `IList`
`IList.IndexOf`	Определяет индекс заданного элемента в списке `IList`
`IList.Insert`	Вставляет элемент в `IList` по указанному индексу
`IList.IsFixedSize`	Получает значение, показывающее, имеет ли список `IList` фиксированный размер
`IList.IsReadOnly`	Получает значение, указывающее, доступен ли список `IList` только для чтения
`IList.Item`	Получает или задает элемент с указанным индексом
`IList.Remove`	Удаляет первое вхождение указанного объекта из списка `IList`

Интерфейс *IList*

Входит в класс `List<T>`. Представляет неуниверсальную (без параметра `T` в угловых скобках) коллекцию объектов с индивидуальным доступом, осуществляемым при помощи индекса. Содержит интерфейсы `ICollection`, `IEnumerable`. В необходимых случаях может применяться самостоятельно. Содержит в себе члены, представленные в табл. 12.13 и 12.14.

Таблица 12.13. *Свойства интерфейса* `IList`

Свойство	Описание
`Count`	Получает число элементов, содержащихся в коллекции `ICollection` (унаследовано от `ICollection`)
`IsFixedSize`	Получает значение, показывающее, имеет ли список `IList` фиксированный размер
`IsReadOnly`	Получает значение, указывающее, доступен ли список `IList` только для чтения
`Item`	Получает или задает элемент с указанным индексом

Таблица 12.14. *Методы интерфейса* `IList`

Метод	Описание
`Add`	Добавляет элемент в список `IList`
`Clear`	Удаляет все элементы из списка `IList`
`Contains`	Определяет, содержится ли указанное значение в списке `IList`
`CopyTo`	Копирует элементы коллекции `ICollection` в массив `Array`, начиная с указанного индекса массива `Array`
`GetEnumerator`	Возвращает перечислитель, осуществляющий итерацию в коллекции (унаследовано от `IEnumerable`)
`IndexOf`	Определяет индекс заданного элемента в списке `IList`
`Insert`	Вставляет элемент в `IList` по указанному индексу
`Remove`	Удаляет первое вхождение указанного объекта из списка `IList`
`RemoveAt`	Удаляет элемент списка `IList`, расположенный по указанному индексу

Класс `List<T>` является универсальным эквивалентом класса `ArrayList`, поэтому последний здесь не рассматривается. `List<T>` реализует универсальный интерфейс `IList<T>`, который обеспечивает массив, размер которого динамически изменяется по мере необходимости. Класс `List<T>` использует метод сравнения объектов на равенство и метод упорядочения.

Такие методы, как `Contains()`, `IndexOf()`, `LastIndexOf()` и `Remove()`, используют метод сравнения на равенство для элементов списка. Метод сравнения на равенство, используемый по умолчанию для типа `T`, определяется следующим образом. Если тип `T` реализует универсальный интерфейс `IEquatable<T>`, в качестве метода проверки на равенство используется метод `Equals(T)` этого интерфейса, в противном случае по умолчанию применяется метод `Object.Equals(Object)`. Такие методы, как `BinarySearch()` и `Sort()`, используют упорядочение элементов списка. Метод, применяемый по умолчанию для типа `T`, определяется следующим образом. Если тип `T` реализует универсальный интерфейс `IComparable<T>`, в качестве метода по умолчанию используется метод `CompareTo(T)` этого интерфейса. В противном случае, если тип `T` реализует неуниверсальный интерфейс `IComparable`, в качестве метода по умолчанию используется метод `CompareTo(Object)` этого интерфейса. Если тип `T` не реализует ни один из интерфейсов, метод по умолчанию не определяется. В этом случае метод сравнения должен быть задан явным образом. До выполнения таких операций, как `BinarySearch`, требуется отсортированный список `List<T>`.

Доступ к элементам коллекции `List<T>` осуществляется с помощью целочисленного индекса. Индексы в этой коллекции начинаются с нуля. Список `List<T>` принимает `null` в качестве допустимого значения для ссылочных типов и разрешает дублирование элементов.

Делая выбор между классами `List<T>` и `ArrayList`, предлагающими сходные функциональные возможности, следует помнить, что класс `List<T>` в большинстве случаев обрабатывается быстрее.

Пример использования класса показан в листинге 12.9. Результат представлен на рис. 12.9.

Листинг 12.9

```
/* Created by SharpDevelop.
 * User: user
 * Date: 18.12.2012
 * Time: 9:37
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Collections.Generic;
```

```
namespace app53_list
{
  class Program
  {
    public static void Main()
    {
      Console.WriteLine("Работа с классом LIst<T>");
      List<string> d = new List<string>();

      Console.WriteLine("\nОбъем коллекции после ее " +
               "создания : {0} элементов", d.Capacity);

      d.Add("Иванов");
      d.Add("Петров");
      d.Add("Сидоров");
      d.Add("Кузнецов");
      d.Add("Яковлев");

      Console.WriteLine("Добавленные элементы:");
      Console.WriteLine();
      foreach(string b in d)
      { Console.WriteLine(b); }

      Console.WriteLine("\nОбъем коллекции после ее " +
        "инициализации добавкой элементов: {0} элементов",
        d.Capacity);
      Console.WriteLine("Фактическое количество: {0} " +
                        "элементов", d.Count);

      Console.WriteLine("\nСодержит \"Кузнецов\"?: {0}",
      d.Contains("Кузнецов"));

      Console.WriteLine("\nВставка элемента \"Яковлев\" после 2-го
элемента");
      d.Insert(2, "Яковлев");
      Console.WriteLine("Коллекция после вставки " +
                        "элемента:");
      Console.WriteLine();
      foreach(string b in d)
      { Console.WriteLine(b); }
```

```
      Console.WriteLine("\nЭлемент по индексу 3 " +
                        "(счет от нуля): {0}", d[3]);

      Console.WriteLine("\nУдаление элемента \"Яковлев\"");
      d.Remove("Яковлев"); // Удаляется первый попавшийся

      Console.WriteLine("Коллекция после удаления " +
                        "элемента:");
      Console.WriteLine();
      foreach(string b in d)
      { Console.WriteLine(b); }

      d.TrimExcess(); // Уменьшить емкость
               // до фактического количества элементов
      Console.WriteLine("\nУменьшение емкости до " +
         "фактического количества элементов");
      Console.WriteLine("Объем: {0}", d.Capacity);
      Console.WriteLine("Количество: {0}", d.Count);

      d.Clear(); // Удаляет фактическое количество
      // элементов, не затрагивая объема коллекции
      Console.WriteLine("\nОчистка коллекции");
      Console.WriteLine("Объем (остался): {0}",
                        d.Capacity);
      Console.WriteLine("Количество фактическое " +
                        "после очистки: {0}", d.Count);
      Console.Read();
    }
  }
}
```

В примере демонстрируется несколько свойств и методов универсального класса `List<T>` строкового типа. С помощью конструктора по умолчанию создается список строк с емкостью по умолчанию. Выводится значение свойства `Capacity`, а затем с помощью метода `Add()` добавляется несколько элементов. Выводится список этих элементов, а потом снова выводится значение свойства `Capacity` и вместе с ним — значение свойства `Count`, показывающее, что емкость увеличивается по мере необходимости: `Capacity` возвращает или задает общее число элементов, которые может вместить внутренняя структура данных без изменения размера. `Count` возвращает число элементов, которые факти-

чески содержатся в коллекции. То есть значение `Capacity` всегда больше `Count` (так среда делает). Поэтому после добавления к коллекции пяти элементов объем коллекции (`Capacity`) стал равен 8.

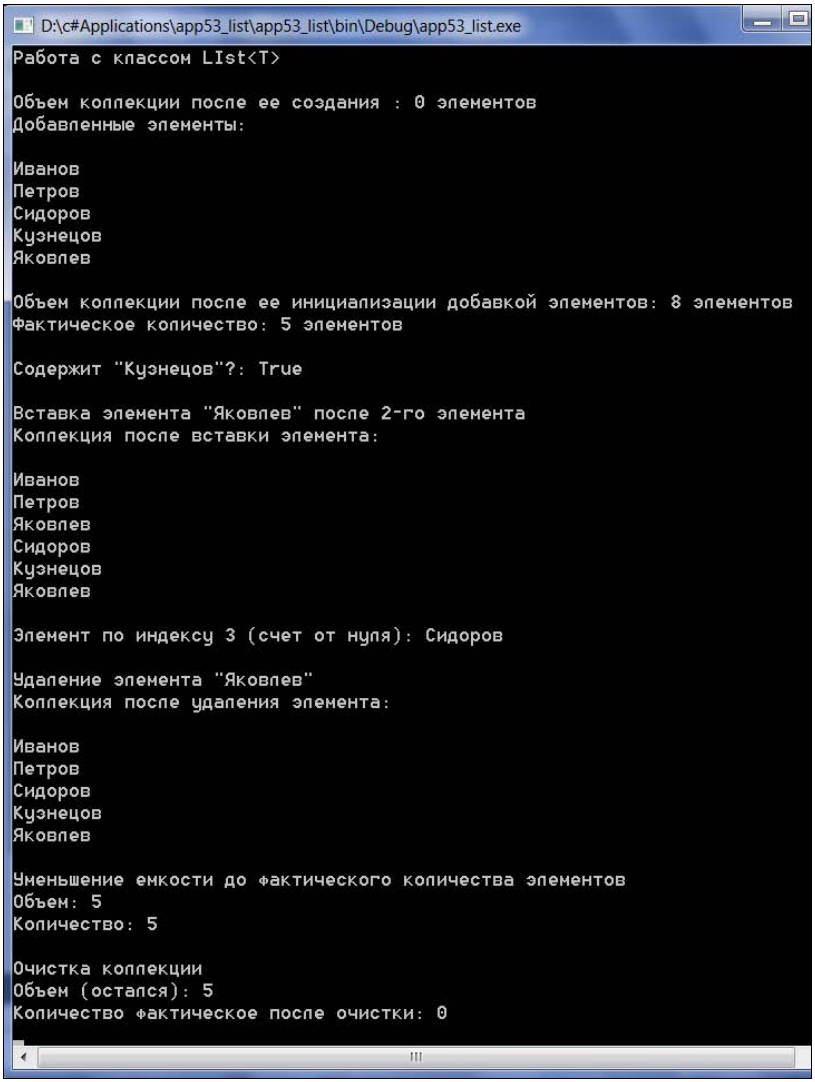


Рис. 12.9. Результат проверки работы методов класса `List<T>`

Посредством метода `Contains()` проверяется наличие некоторого элемента в списке, с помощью метода `Insert()` в середину списка вставляется новый элемент, после чего снова выводится содержимое списка. Свойство по умолчанию `Item()` (индексатор в C#) позволяет извлечь элемент из списка, с помощью метода `Remove()` удаляется первый экземпляр дублированного элемента, добавленный ранее, и содержимое списка выводится вновь. Метод `Remove()` всегда удаляет первый обнаруженный им экземпляр. С помощью метода `TrimExcess()` емкость уменьшается в соответствии с числом элементов и выводятся значения свойств `Capacity` и `Count`. Если незадействованная емкость составляет менее 10% от общей емкости, изменение размера списка не требуется. В заключение используется метод `Clear()` для удаления всех элементов из списка и выводятся значения свойств `Capacity` и `Count`. Отметим еще одну деталь: оператор `foreach` для коллекции, образованной из этого класса, не требует создания перечислителя и связки его с этим оператором.

Создание сравнимых объектов

Сравнение объектов может происходить по разным аспектам. Например, по типам: относятся ли объекты к одному и тому же типу или к разным типам. Для этого существуют операторы `is` и `as`. Но можно сравнивать объекты по значениям их определенных полей. Тут уже употребляются обычные операции сравнения: "больше", "меньше" и т. д. Такой способ сравнения позволяет выполнять одну из необходимых операций при обработке информации — сортировку, т. е. упорядочение некоего вида информации по величине того или иного ее поля в сторону его увеличения или убывания. Для этой цели существует специальный интерфейс `IComparable` в пространстве имен `System` с единственным своим членом — методом `CompareTo()`. Это обобщенный метод сравнения для типа, который реализуется типом или классом значения, чтобы упорядочить или сортировать его экземпляры. Он возвращает число типа `int`, значение которого указывает на местоположение сравниваемых объектов: если возвращаемое число меньше нуля, это значит, что экземпляр класса, в котором находится метод сравнения, расположен (в памяти) перед указанным в параметре метода объектом. Если возвращаемое число больше нуля, это значит, что экземпляр класса, в котором находится метод сравнения, расположен (в памяти) после указанного в параметре метода объекта. При равенстве нулю объекты равны.

Пример приложения с применением интерфейса `IComparable` показан в листинге 12.10. Результат работы — на рис. 12.10.

Листинг 12.10

```
/* Created by SharpDevelop.
 * User: user
 * Date: 16.12.2012
 * Time: 14:07
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Collections.Generic;

namespace app50_compare
{
  public class Dictionary : IComparable<Dictionary>
  { // Автоматические свойства: имя и возраст человека
    public string Name {get; set;}
    public int Age {get; set;}
    public Dictionary() {} // Конструктор без параметров
                           // (по умолчанию)

    // Определение члена интерфейса IComparable
    public int CompareTo(Dictionary ob)
    { // Проверка: является ли параметр объектом
      // типа Dictionary
      Dictionary d = ob as Dictionary;
      if(d != null)
        return this.Age.CompareTo(ob.Age);
      else
        throw new ArgumentException("Объект не является "+
                                    "типом Dictionary");
    }
  } // Dictionary
  class Program
  {
    public static void Main()
    {
      Console.WriteLine("Применение интерфейса " +
                        "IComparable");
```

```
      /* Массив List представляет строго типизированный
         список объектов (тип задается в угловых скобках),
         доступных по индексу. Поддерживает добавление
         элементов, методы для поиска по списку,
         выполнения сортировки и других операций
         со списками.*/
      // Массив объектов-пар (коллекция)
      List<Dictionary> d = new List<Dictionary>();
      // Инициализация массива:
      // каждый элемент — это новый объект
      d.Add(new Dictionary() { Name = "Иван", Age = 40 });
      d.Add(new Dictionary() { Name = "Петр", Age = 20 });
      d.Add(new Dictionary() { Name = "Андрей", Age = 10 });

      Console.WriteLine("Словарь до сортировки:");
      foreach (var b in d)
         Console.WriteLine("{0}  {1}", b.Name,b.Age);

      // Использование IComparable.CompareTo():
      // сортировка вызовет этот метод
      d.Sort();

      Console.WriteLine("Словарь после сортировки");
      foreach (var b in d)
        Console.WriteLine("{0}  {1}",b.Name,b.Age);

      Console.Read();
    }
  }
}
```

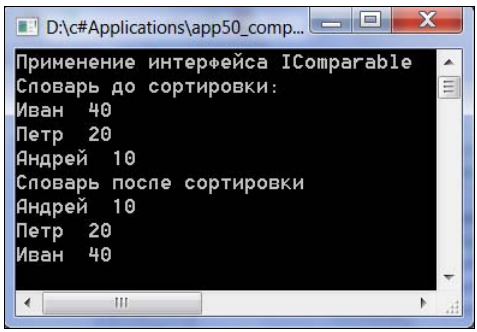


Рис. 12.10. Результат применения интерфейса `IComparable`

Создается тип данных — class Dictionary. К нему подключается универсальный интерфейс IComparable<T> с указанием конкретного типа: IComparable<Dictionary>. Подключение неуниверсального интерфейса IComparable система почему-то не пропустила. Задается конкретный метод интерфейса — метод CompareTo(). Это метод, по которому станут сравниваться элементы коллекции. Мы ее создадим, пользуясь типом Dictionary. В метод CompareTo() в качестве параметра должен передаваться элемент коллекции типа Dictionary. В методе проверяется, имеет ли переданный объект тип Dictionary. Если нет, то выдается исключительная ситуация и метод завершается. Если переданный объект — объект типа Dictionary, то выполняется оператор сравнения текущего объекта коллекции и переданного для сравнения. Результат выдается методом на его выходе. Этот метод станет вызываться методом Sort() класса List<T>, где T=Dictionary. А с помощью этого класса в основной программе организуется коллекция из трех элементов. Коллекция наполняется методом Add() этого класса, после чего и вызывается его метод Sort() для сортировки элементов коллекции. Компилятор так построит алгоритм работы по сортировке коллекции, что когда Sort() начнет работать и вызовет первый раз метод CompareTo(), она больше из него не выйдет, пока вся коллекция не будет отсортирована. Взаимодействие Sort() и CompareTo() организуется компилятором, как говорят, за кулисами, не видно для пользователя. При этом организуется передача методу CompareTo() элементов коллекции, сравнение текущего элемента с предыдущим, смена мест элементов и т. д. То есть все то, что предусматривается алгоритмом сортировки. Некоторые детали этого механизма можно проследить в режиме отладки, прокручивая действия CompareTo() и наблюдая за промежуточными результатами. И еще один интересный момент: при использовании коллекции на основе List<T> применение оператора foreach для просмотра коллекции становится совершенно несложным: нет необходимости организовывать работу перечислителя, который потом надо связывать с foreach. Просто пишется foreach, как будто имеем дело с обычным массивом из целых чисел. Видимо, потому, что интерфейс IEnumerable реализован в самом классе List<T>.

Обобщения

Мы знаем, что все классы являются потомками одного базового класса Object. Когда создаются коллекции данных, то в результате различных алгоритмических манипуляций с данными (приведением одних типов

к другим, использованием типа `object` в качестве промежуточного типа и т. д.) в коллекции могут оказаться данные не того типа, с которым должна работать коллекция. Кроме того, приведение данных связано с выполнением операций по упаковке/распаковке данных, что занимает значительное дополнительное время. (Упаковка данных — это процесс преобразования значения данного в ссылку на него и возможное перемещение данного из стековой памяти программы в кучу, где хранятся ссылочные данные; а распаковка — это обратный процесс: преобразование ссылки на данное в значение данного, и такое данное уже не может храниться в куче. Его надо перемещать в стековую память программы.) Возникает проблема типизации коллекций, задание типа данных, с которыми (и только с ними) может работать коллекция. Но тогда каждый раз для нового типа данных надо будет создавать заново новый класс. Человечество всегда на пути своего развития стремилось к обобщениям. Вот и в данном случае первое, что приходит в голову: а нельзя ли как-то параметризировать процесс создания класса? Сделать так, чтобы параметром класса был тип данного? И этот тип конкретизировался на этапе создания экземпляра класса, позволяя создавать объект строго типизированным? Эту проблему как раз и решают средства C#, которые называются *обобщениями*. Для этого было создано специальное пространство `System.Collection.Generic`, в котором и размещаются такие классы. Но обобщенными классами дело не ограничивается: обобщенными можно создавать интерфейсы (хотя это тоже классы, но без возможности их наследования) и методы, которые можно определять и не в обобщенных классах. Обобщенный элемент отличить от других элементов в документации .NET Framework легко по наличию угловых скобок с буквой или другой лексемой (лексема — последовательность допустимых символов, которую распознает компилятор) в них. Например, мы встречались уже с классом `List<T>`. Конструкцию `<T>` можно читать как "типа T". `T` — это место подстановки настоящего типа данных при объявлении экземпляра "типизированного" класса. Например, коллекция целых чисел, полученная с применением обобщенного класса `List<T>`, может быть объявлена так:

```
List<int> MyInt = new List<int>();
```

Имя параметра типа может быть не обязательно "T". Это имя задает тот, кто создает обобщенный элемент и создает для него документацию. Обычно в качестве имени могут быть для представления типов — `T`, для представления ключей — `K`, `Key`, для представления значений — `V` или `TValue`.

Итак, значение параметра должно быть явно указано при создании объекта из обобщенного элемента. Ранее мы встречались уже с применени-

ем List<T>: в листинге 12.10 был объявлен класс пары "ключ — значение" (имя человека — его возраст) и на основе этого типа создана коллекция вида:

```
List<Dictionary> d = new List<Dictionary>();
```

Эта коллекция будет содержать только элементы типа Dictionary. То есть на этапе создания коллекции был определен (задан) конкретный тип создаваемой коллекции.

Необобщенные классы или структуры могут иметь обобщенные члены (методы, свойства). Например, необобщенный класс Array поддерживает обобщенный метод сортировки Sort<T>. Пример использования этого метода приведен в листинге 12.11, а результат — на рис. 12.11.

Листинг 12.11

```
/* Created by SharpDevelop.
 * User: user
 * Date: 19.12.2012
 * Time: 14:34
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers.
 */
using System;
using System.Collections;

namespace app54_generic1
{
  class Program
  {
    public static void Main()
    {
      Console.WriteLine("Сортировка массива " +
          "типизированным методом Sort<T>()");
      int [] MyInt = {12,1,4,85,43,8}; // Массив чисел.
        // К нему можно применять инструменты класса Array

      Console.WriteLine("Массив до сортировки:");
      foreach(int i in MyInt)
        Console.WriteLine("{0}", i);
```

```
      Array.Sort<int>(MyInt); // Сортировка массива
                              // с указанием типа
      Console.WriteLine("Отсортированный массив:");
      foreach(int i in MyInt)
        Console.WriteLine(" {0}",i);

      Console.Read();
    }
  }
}
```

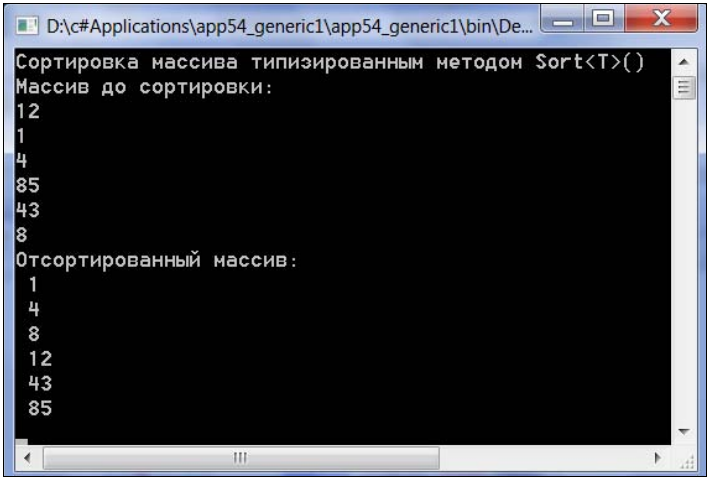


Рис. 12.11. Сортировка массива с помощью типизированного метода `Sort` из необобщенного класса

Пример создания собственного обобщенного класса и его применение показаны в листинге 12.12. Результат — на рис. 12.12.

Листинг 12.12

```
/* Created by SharpDevelop.
 * User: user
 * Date: 19.12.2012
 * Time: 15:43
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
```

```
using System;

namespace app55_generic_T
{
  // Задание обобщенной структуры Point
  public class Point<T>
  {
    // Обобщенные поля:
    private T xPos;
    private T yPos;


    // Обобщенный конструктор:
    public Point(T xVal, T yVal)
    {
      xPos = xVal;
      yPos = yVal;
    }
    // Обобщенные свойства
    public T X
    { get { return xPos; }
      set { xPos = value; }
    }
    public T Y
    { get { return yPos; }
      set { yPos = value; }
    }
    public override string ToString()
    { return string.Format (" [{0 }, {1}]", xPos, yPos);
    }
    // Метод: сбросить поля в значения по умолчанию
    public void ResetPoint()
    { // В C# ключевое слово default перегружено.
      // Здесь оно означает значение по умолчанию
      // для параметра типа
      // (0 — для значений, null — для ссылочных данных)
      xPos = default(T);
      yPos = default(T);
    }
  } // Point<T>
```

```
  class Program
  {
    public static void Main()
    {
      Console.WriteLine("Применение созданного " +
                        "обобщенного класса");
      Point<int> MyPoint = new Point<int>(100,25);
      Point<double> MyPoint2 =
        new Point<double>(2.5,3.85);
      Console.WriteLine("Координаты типа int: {0} {1}",
        MyPoint.X, MyPoint.Y);
      Console.WriteLine("Координаты типа double: {0} {1}",
        MyPoint2.X, MyPoint2.Y);

      Console.Read();
    } // Main()
  }
}
```

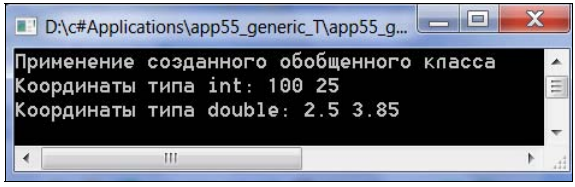


Рис. 12.12. Работа с созданным обобщенным классом

При наследовании обобщенных классов следует соблюдать несколько правил:

- необобщенные типы должны указывать параметр типа;
- если обобщенный базовый класс содержит обобщенные или абстрактные методы, то наследник должен переопределять обобщенные методы, используя указанный параметр типа, а именно подставлять параметр типа, использованный в родительском классе, в унаследованные методы;
- если тип-наследник — тоже обобщенный, то он может повторно использовать тот же тип в своем определении.

Ограничения для параметров типа

Платформа .NET позволяет с помощью ключевого слова `where` вводить ряд ограничений на параметр типа.

Это следующие требования к параметру:

- `where T : struct` означает, что параметр `<T>` должен иметь в своей цепочке наследования `System.ValueType`, т. е. `T` должен быть структурой (структура — это тип-значение);
- `where T : class` означает, что параметр `<T>` не должен иметь в своей цепочке наследования `System.ValueType`, т. е. быть ссылочным типом;
- `where T : new` означает, что параметр `<T>` должен иметь конструктор по умолчанию, т. е. обобщенный тип должен создавать экземпляры параметра типа. Если в типе много ограничений, то это ограничение должно быть последним;
- `where T :` *`Имя_базового_класса`* означает, что параметр `T` должен быть наследником указанного базового класса;
- `where T :` *`Имя_интерфейса`* означает, что параметр `T` должен реализовать указанный интерфейс. Можно задавать несколько интерфейсов, разделяя их запятыми.

Обобщенный класс может иметь более одного параметра. В таком случае можно писать ограничения на каждый параметр отдельно. Например,

```
public class A<K,T> where K : NameOfBaseClass, new() where T :
IComparable<T>
```

К параметрам типа нельзя применять операции арифметики и сравнения. Например, нельзя писать:

```
public T Add(T arg1, T arg2)
 {return arg1 + arg2}
```

Выход здесь состоит в том, что придется создавать свой интерфейс, который бы переопределял "неприятные" операции, а потом уже вводить ограничения на тип вида: `where T :` *`operator +`*, *`operator`* — и т. д.

ГЛАВА 13

Делегаты и события

Делегат — это конструкция C#, с помощью которой можно вызывать некую функцию. И не одну. Но вызываются те функции, сигнатура (т. е. заголовок) которых совпадает с сигнатурой делегата. Из этого определения следует, что синтаксис определения делегата должен быть похож на синтаксис задания функции. А что входит в синтаксис задания функции? Тип возвращаемого ею значения, имя и список параметров с их типами в круглых скобках. Отвлечемся от имени и рассмотрим остальные характеристики заголовка. Делегат как раз способен вызывать любую функцию, у которой тип возвращаемого значения и список параметров с их типами совпадают с объявленными в делегате. Синтаксис объявления делегата:

```
public delegate ТипВозвращаемогоЗначения
ИмяДелегата(список_параметров_с_их_типами);
```

Например,

```
public delegate int MyDelegate(string a, double b);
```

Этот делегат может вызывать любой метод (или функцию), у которого тип возвращаемого значения — `int`, первый параметр описан как `string a`, а второй — `double b`. Заметили, что здесь ни слова не говорится об имени метода, с которым связан конкретный алгоритм реализации этого метода? То есть с помощью делегата можно вызывать любой метод, лишь бы его сигнатура (усеченная, без имени) совпадала с сигнатурой делегата.

Отметим, что делегат — это класс, имеет свой конструктор и потому с его помощью можно создавать объекты класса-делегата. Расшифровывая объявление конструкции-делегата, компилятор порождает запе-

чатанный класс с тремя методами, способными вызывать функции с объявленной в делегате сигнатурой. Один метод может вызывать функции синхронно, т. е. вызывающий код программы должен ожидать завершения вызванной функции, а уже только после этого продолжить выполняться дальше, другие же два метода дают возможность вызова функции асинхронным способом, т. е. в отдельном от общего потоке выполнения. Это связано с наличием функций (методов), которые требуют определенного времени для своего завершения и потому не должны особо влиять на основной процесс.

Пример простого делегата и вызов с его помощью функций приведен в листинге 13.1, а результат показан на рис. 13.1.

Листинг 13.1

```
/* Created by SharpDevelop.
 * User: user
 * Date: 20.12.2012
 * Time: 14:55
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app56_delegate
{
  // Этот делегат может вызывать любой метод, принимающий
  // два целых числа и возвращающий целое число
  public delegate int Ar(int x, int y) ;
    // Этот класс содержит методы, которые может
    // вызывать объявленный делегат,
    // т. к. сигнатуры делегата и методов совпадают:

  public class SimpleMath
  {
    public static int Add(int x, int y)
    { return x + y; }
    public static int Subtract (int x, int y)
    { return x – y; }
  }
  class Program
  {
```

```
    static void Main()
    { Console.WriteLine ("Пример простого делегата\n");

      // Создать экземпляр делегата, указывая
      // в конструкторе вызываемый метод
      Ar b = new Ar(SimpleMath.Add);  // Метод вызывается
            // прямо из класса, т. к. метод — статический

      // Вызов метода Add() с использованием делегата
      Console.WriteLine("2+2 равно {0}", b(2, 2));

      Ar c = new Ar(SimpleMath.Subtract);
      Console.WriteLine("4-2 равно {0}", c(4, 2));
      Console.Read();
    }
  }
}
```

Рис. 13.1. Результат работы с делегатом

События

В реальном мире нас постоянно окружают различные события: мы умылись, мы пришли на работу, мы вернулись с работы и т. д. События проникают в нашу жизнь и на более мелком уровне: мы сделали шаг вперед — уже событие, в ответ на которое у нас в организме запускается целое множество ответных реакций; мы повернули голову — то же самое. Мы моргнули — это тоже важное событие для нашего организма. События можно разделить на два вида: те, которые мы инициируем (вызываем сами), и те, которые инициируются кем-то другим. Например, мы пришли с работы. По сути это сработало некое событие, в ответ на которое мы что-то сделали (например, перешли в состояние "дома").

А вот если в нас ткнули пальцем, то получается, что кто-то вызвал этим у нас целую кучу реакций, и тогда он является инициализатором события, которое запустило эти реакции.

В программной среде, которая, в общем, отражает реальную жизнь, все происходит очень похоже. Есть события, которые инициализируются самим приложением (вызываются программой), а есть такие, которые инициализирует пользователь, через нажатие, например, кнопки, представленной в графическом интерфейсе приложения. События не остаются сами по себе существовать. На них требуется какая-то реакция. А какая может быть реакция в программе? В ответ на произошедшее событие должна запускаться какая-то другая программа, в теле которой станет происходить обработка какой-то информации, связанной с реакцией на данное событие. Такая запускаемая программа называется *обработчиком события*. Тот, кто работал в среде, скажем, Visual C++, знает, что у каждого компонента (фактически это класс), с помощью которого строится графический интерфейс, существует помимо его свойств еще и набор событий. Если дважды щелкнуть мышью на названии любого события, среда программирования автоматически создаст и откроет для вас заготовку (шаблон) обработчика этого события. В заготовке будет лишь заголовок метода (функции) и пустое тело, в которое программист может вписать команды программы, отражающие реакцию на событие. Например, если компонентом была кнопка, то щелчок мышью на кнопке (как и двойной щелчок на ее событии `OnClick` в списке событий) выведет вас на обработчик этого события, в котором вы можете записать команды, например, изменить цвет кнопки с текущего на красный.

Если вы внимательно посмотрите на заголовки таких обработчиков событий, то увидите, что все они построены по одному стандарту: имя обработчика (обычно имя связано как-то с именем объекта, событие которого должно быть обработано; например, для кнопки с именем `Button1` имя обработчика может иметь вид `Button1_Click`) и всего два параметра в скобках:

```
(Object  sender, EventArgs  e)
```

Вот пример двух обработчиков событий (щелчка на форме и щелчка на кнопке, которая помещена в форму):

```
private: System::Void Form1_Click(System::Object  sender,
System::EventArgs  e)
{
}
```

```
private: System::Void button1_Click(System::Object  sender,
System::EventArgs  e)
{
}
```

Параметрами обработчика являются: сам объект, породивший событие (`sender` содержит ссылку на этот объект), и данные для обработки события, содержащиеся в объекте `e` типа `EventArgs` (это класс). Так как тип параметра — `object`, мы можем передавать в обработчик все, что угодно.

Когда вы глядите на форму задания обработчиков событий, не напоминает ли вам это что-то знакомое? У всех обработчиков один и тот же тип возвращаемого значения (`void`, т. е. обработчик ничего не возвращает, а только выполняет заданные в теле действия), у всех обработчиков одинаковый набор параметров и их типов. Это наталкивает на мысль, что при такой стандартизации сигнатур обработчиков должен быть общий механизм их запуска. А мы его только что рассматривали: это — делегат. Именно он один в своем роде может запускать (вызывать на выполнение) подобного рода функции, если этот делегат определить с такой же сигнатурой, что и у обработчиков событий. Итак, обобщая сказанное, можем сформулировать положения, касающиеся структуры шаблона обработки событий в .NET Framework:

- нам нужен класс `EventArgs` (в базовом случае) или класс, унаследованный от `EventArgs` для передачи аргументов события (какая кнопка мыши была нажата, например);
- нам нужен делегат, который будет жестко задавать сигнатуру (тип) метода, способного обрабатывать событие;
- нам необходимо само событие, которое объявляется приблизительно так: `public event EventHandler MyEvent`.

Итак, обработчик события — это делегат со специальной сигнатурой:

```
public delegate void MyEventHandler(object sender, MyEventArgs e);
```

Здесь, повторю, первый параметр (`sender`) определяет объект, который издает событие. Второй параметр (`e`) содержит данные, которые должны быть использованы обработчиком события. Класс `MyEventArgs` должен быть производным от класса `EventArgs`. `EventArgs` является базовым классом для более специализированных классов, таких как `MouseEventArgs`, `ListChangedEventArgs` и т. д. Для GUI-события (Graphic User Interface, графический интерфейс пользователя) вы можете применять объекты этих специализированных классов без создания собствен-

ного класса. Однако для остальных событий вы должны создать свой класс и держать в нем данные, которые хотите передать делегату, точнее — вызываемой им функции-обработчику.

Пример обработки событий показан в приложении, приведенном в листинге 13.2, а результат работы приложения — на рис. 13.2

Листинг 13.2

```
/* Created by SharpDevelop.
 * User: user
 * Date: 21.12.2012
 * Time: 12:22
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */

/* В  примере класс A определяет обработчики событий
  (методы с сигнатурой, аналогичной делегатам). Далее
   методы связываются с соответствующими делегатами.
   Метод класса A примет с помощью делегата объект
   класса B. Затем, при возникновении события в классе B
   будет вызван обработчик события, т. е. метод класса A.
*/
using System;

namespace app57_event
{
  // Создаем делегаты для вызова обработчиков событий
  public delegate void MyHandler1(object sender,
                                  MyEventArgs e);
  public delegate void MyHandler2(object sender,
                                  MyEventArgs e);

  // Создаем первый класс (A)
  class A
  {
    public const string m_id="Class A"; // поле

    // Обработчик 1-го события:
    public void OnHandler1(object sender, MyEventArgs e)
```

```
  {
    Console.WriteLine("Обработчик 1-го события и " +
         "набор аргументов для него:  {0}", e.m_id);
  }

  // Обработчик 2-го события:
  public void OnHandler2(object sender,MyEventArgs e)
  {
    Console.WriteLine("Обработчик 2-го события и " +
          "набор аргументов для него: {0}", e.m_id);
  }

  // Конструктор класса A инициализирует события
  // класса B (объявлен далее)
  public A(B b)
  {
    // Создать экземпляр делегата, указывая
    // в конструкторе вызываемый метод (связка метода
    // и делегата): из первого делегата создается
    // объект для вызова первого обработчика события.
    // Вызов можно будет делать так:
    //    d1(1-й пар, 2-й пар);
    MyHandler1 d1=new MyHandler1(OnHandler1);

    // Из второго делегата создается объект для вызова
    // второго обработчика события:
    MyHandler2 d2=new MyHandler2(OnHandler2);

    // В поля  Event1, Event2 засылаются ссылки
    // для вызова методов OnHandler1 и OnHandler2.
    // Теперь эти методы можно будет вызывать так:
    // Event1(ссылка на объект, данные для обработчика)
    b.Event1 +=d1; // регистрация события
    b.Event2 +=d2;
  }
}

// Создание класса B
class B
{
  public event MyHandler1 Event1; // Поле 1 - так
     // задается событие (тип переменной — 1-й делегат)
```

```
  public event MyHandler2 Event2; // Поле 2. 2-е
     // событие. Тип переменной — 2-й делегат

  public void FireEvent1(MyEventArgs e) // Обработка
                                        // события
  {
    if(Event1 != null)
    {
      Event1(this, e);  // Event1 сформировано
                        // конструктором класса A
    }
  }
  public void FireEvent2(MyEventArgs e)
  {
    if(Event2 != null)
    { Event2(this,e); }
  }
}

// Задание собственного класса, содержащего
// информацию для обработчика события
public class MyEventArgs : EventArgs
{ public string m_id; }

public class Program
{
  public static void Main()
  {
    B b= new B();
    A a= new A(b);
    MyEventArgs e1=new MyEventArgs();  // Объект,
              // содержащий аргументы для 1-го события
    MyEventArgs e2=new MyEventArgs();  // Объект,
              // содержащий аргументы для 2-го события
    e1.m_id ="Аргументы для события 1";
    e2.m_id ="Аргументы для события 2";

    // Внутри этих методов происходит фактическая
    // обработка событий:
    b.FireEvent1(e1);  // Обработка 1-го события
    b.FireEvent2(e2);  // Обработка 2-го события
```

```
        Console.Read();
      }
    }
}
```

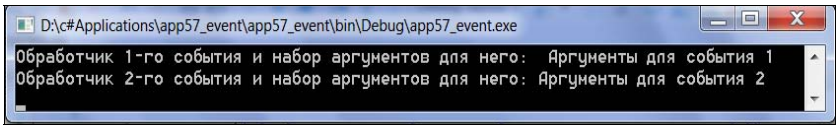


Рис. 13.2. Результат создания и обработки событий

Итак, мы видели, что в классе `B` заданы два события в виде (мы берем для пояснения одно, первое)

```
public event MyHandler1 Event1;
```

Здесь `MyHandler1` — делегат, связываемый с событием `Event1`. Обработка компилятором ключевого слова `event` приводит к автоматическому получению методов регистрации и отмены регистрации события в программе. Если посмотреть программу с помощью функции IL Dasm (**Tools | IL Dasm**), то можно обнаружить, что компилятор формирует в этом случае два метода: `Add_.ИмяСобытия()` и `Remove_.ИмяСобытия()`. То есть один метод добавляет событие, а другой его удаляет при необходимости. Итак, задание события означает автоматическую отправку этого события, как говорят, вызывающему коду, т. е. программе. Вызывающий код должен это событие зарегистрировать. Поэтому в методе обработки события

```
public void FireEvent1(MyEventArgs e)
```

в самом начале должна стоять (и на самом деле стоит) проверка `if(Event1 != null)`, означающая вопрос: зарегистрировано ли данное событие в вызывающем коде? Если событие зарегистрировано, то выполняется обработчик этого события.

Регистрация события происходит с помощью операций `+=` и `-=`. Эти операции и вызывают на самом деле выполнение методов `Add_.Event1()` и `Remove_.Event1()`. Последний метод отключает обработку события. В нашем случае в классе `A`, а точнее — в его конструкторе и происходит фактическая регистрация обоих событий. Сами события объявлены в классе `B`, а их регистрация происходит при создании объекта класса `A`, т. к. объект создается конструктором. В конструкторе `A` из первого делегата создается объект `d1` для вызова первого обработчика события, а из

второго делегата — объект d2 для вызова второго обработчика события. Потом с помощью операции += и происходит регистрация обоих событий.

Мы могли бы не объявлять двух делегатов, а воспользоваться одним. А к нему присовокупить два метода:

```
MyHandler1 d1=new MyHandler1(OnHandler1);
MyHandler1 d2=new MyHandler1(OnHandler2);
```

То есть тогда бы в списке вызовов делегата было бы два метода, а не по одному, как в нашем примере. Тогда бы, при желании, мы могли бы один метод исключить из списка вызовов, применив операцию -=. Например,

```
b.Event2 -=d2;
```

Анонимные методы

Анонимными являются методы, у которых нет имени. Это один из способов создать безымянный блок программы, который будет связан с конкретным делегатом, т. е. станет выполняться через делегата. Пример анонимного метода показан в программе листинга 13.3, а результат работы — на рис. 13.3.

Листинг 13.3

```
/* Created by SharpDevelop.
 * User: user
 * Date: 22.12.2012
 * Time: 15:44
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.IO;

namespace app59_anonim1
{
  static class A
  {
    // Сигнатура делегата
    public  delegate void Anonim2(int start, int finish);
    public static int Speed = 150;
```

```
    // Создадим анонимный метод, который выведет нам
    // на консоль изменение скорости автомобиля
    // на отрезке в 10 км.
    // Используем анонимный метод: в делегате,
    // по определению, указывается вызываемый им метод,
    // а здесь мы просто задаем блок программы вместо
    // метода, который должен исполнить делегат
  }

  class Program
  {
    public static void Main()
    {
      Console.WriteLine("Анонимный метод\nИзменение " +
                        "скорости автомобиля");
      A.Anonim2 anonim2 = delegate(int a, int b)
      {
        for (int i = a; i <= b; i++)
          Console.Write("Скорость автомобиля на {0} " +
                "километре:{1}\n", i, A.Speed — (b-i));
      };
      anonim2(1, 10);
      Console.Read();
    }
  }
}
```

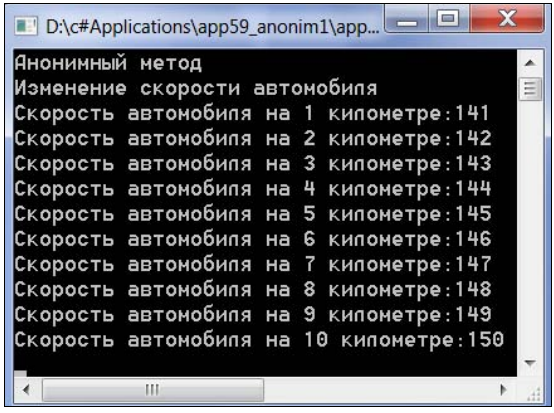


Рис. 13.3. Расчет по анонимному методу

Когда компилятор встречает ключевое слово `delegate` с круглыми скобками, в которых может быть или отсутствовать список параметров, а за ключевым словом идет открывающая фигурная скобка, он знает, что начался анонимный метод. Блок исполняемых команд — это команды, заключенные в фигурные скобки. Блок должен обязательно заканчиваться точкой с запятой, как обычный оператор.

Пример анонимного метода без параметров приведен в программе листинга 13.4, а результат работы — на рис. 13.4.

Листинг 13.4

```
/* Created by SharpDevelop.
 * User: user
 * Date: 22.12.2012
 * Time: 18:52
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app61_anonim3
{
  class Program
  {
    delegate void DelegateType();

    static DelegateType GetMethod()
    { return delegate()
      { System.Console.WriteLine("Hello"); };
    }

    static void Main()
    { DelegateType delegateInstance = GetMethod();
      delegateInstance();
      Console.Read();
    }
  }
}
```

Рис. 13.4. Результат работы анонимного метода без параметров

Когда компилятор находит ключевое слово `delegate` в теле метода, он ожидает, что за ним последует тело анонимного метода. Анонимный метод можно присвоить ссылке на делегат.

Заметим также, что можно использовать оператор `+=`, чтобы заставить экземпляр делегата ссылаться на несколько методов сразу (не важно, анонимных или нет). Пример приведен в программе листинга 13.5, результат — на рис. 13.5.

Листинг 13.5

```
/* Created by SharpDevelop.
 * User: user
 * Date: 22.12.2012
 * Time: 19:51
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app62_anonim3
{
  class Program
  {
    delegate void DelegateType();

    static void Main()
    {
      DelegateType delegateInstance = delegate()
      { Console.WriteLine("Hello"); };

      delegateInstance += delegate() // Добавка ссылки
                          // на другой анонимный метод
      { Console.WriteLine("Bonjour"); };

      delegateInstance();
      Console.Read();
    }
  }
}
```

Рис. 13.5. Результат добавки ссылки на другой анонимный метод

Лямбда-выражения

Это сокращенный способ записывать работу с делегатами. Пример представлен в листинге 13.6, а результат — на рис. 13.6.

Листинг 13.6

```
/* Created by SharpDevelop.
 * User: user
 * Date: 23.12.2012
 * Time: 12:16
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app63_lambda1
{
  class Program
  {
    public delegate int Ar(int x);

    public static int Add(int x)
    { return -x; }

    static void Main()
    {
      Console.WriteLine("Пример простого делегата " +
                        "и лямбда-выражения\n");

      // Создать экземпляр делегата, указывая
      // в конструкторе вызываемый метод
      Ar b = new Ar(Add);
```

```
        // Ar d = x => -x; // Лямбда-выражение мы могли
        // бы записать и так, через делегат
        Func<int,int> d = x => -x; // Лямбда-выражение
        // Вызов метода Add() с использованием делегата
        // и лямбда-выражения
        Console.WriteLine("Результат вызова метода " +
                          "делегатом: {0}", b(2));
        Console.WriteLine("Результат вызова метода " +
                          "L-выражением: {0}", d(2));
        Console.Read();
    }
  }
}
```

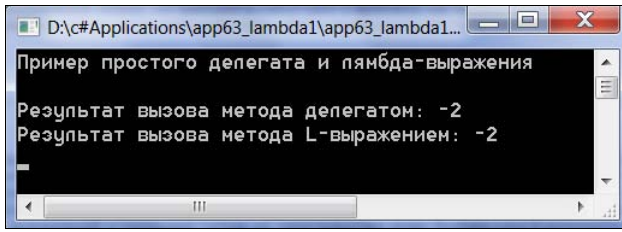


Рис. 13.6. Сопоставление работы делегата и лямбда-выражения

Из рис. 13.6 видно, что результат одинаков, но синтаксис лямбда-выражения проще: в этом примере показан простой анонимный метод, который специфическим способом записан в виде выражения

```
Func<int,int> d = x => -x;
```

Здесь само лямбда-выражение — это `x => -x;`. Читается как "взять `x` в качестве параметра и вернуть результат следующей операции в `x`". Результат операции — это `-x`. Фактически, `-x` — это блок анонимного метода из одного оператора. Вместо ключевого слова `delegate` здесь применено `Func<T,T>` (тип с типизированным списком параметров). `Func<>` — это новый вспомогательный тип, представленный в пространстве имен `System`, который вы можете использовать для объявления простых делегатов, принимающих до четырех аргументов и возвращающих результат. В данном случае объявлена переменная `d`, которая является делегатом, принимающим `int` и возвращающим `int`. Заметим, что лямбда-выражение совершенно не использует информацию о типе переменных. Когда компилятор присваивает лямбда-выражение переменной `d`, он использует информацию о типе делегата для определения

того, что типом x должен быть int, а типом возврата — тоже int. Лямбда-выражение мы могли бы записать напрямую через делегата:

```
Ar d = x => -x;
```

В листинге 13.6 было рассмотрено лямбда-выражение с одним параметром (x). Но параметров может быть несколько или вовсе ни одного. Вообще, синтаксис лямбда-выражения имеет вид:

```
(входные_параметры) => выражение
```

Если лямбда-выражение имеет только один параметр ввода, скобки можно не ставить (что и было в нашем примере), во всех остальных случаях скобки обязательны. Два и более параметра разделяются запятыми и заключаются в скобки. Например,

```
(x, y) => x == y
```

Иногда компилятору бывает трудно или даже невозможно определить введенные типы. В этом случае типы можно указать в явном виде, как показано в следующем примере:

```
(int x, string s) => s.Length > x
```

Нулевые параметры ввода (т. е. когда параметры вообще отсутствуют) указываются пустыми скобками:

```
() => SomeMethod()
```

Лямбда-операторы

В этом случае оператор (или операторы) заключается в фигурные скобки. Синтаксис такой:

```
(Входные_параметры) => {Оператор(операторы)};
```

Основная часть лямбда-оператора может состоять из любого количества операторов; однако на практике обычно используется не больше двух-трех. Пример лямбда-операторов приведен в программе листинга 13.7, а результат работы программы — на рис. 13.7.

Листинг 13.7

```
/* Created by SharpDevelop.
 * User: user
 * Date: 23.12.2012
```

```
 * Time: 14:08
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;

namespace app64_lambda2
{
  class Program
  {
    delegate int del(int i);
    delegate void Test(string s);
    delegate int Test2(int x, int y);

    public static void Main()
    {
      Console.WriteLine("Лямбда-выражения и " +
                        "лямбда-операторы");
      // Один параметр у лямбда-выражения и один оператор
      del myDelegate = x => x * x;
      int j = myDelegate(5); //j = 25
      Console.WriteLine("Лямбда-выражение: {0}",j);

      // Один параметр и более одной строки
      // у лямбда-выражения
      Test myDel = n =>
      {
        string s = n + " " + "World";
        Console.WriteLine(s);
        //  return n;
        // return должен быть, если тип исполняемой
        // делегатом функции не void
      };
      Console.WriteLine("Лямбда-оператор: ");
      myDel("Hello,");

      // Два параметра у лямбда-выражения и более
      // одной строки
      Console.WriteLine("Два параметра "+
                        "у лямбда-выражения");
      Test2 myT = (x, y) =>
```

```
        {
          int z=x+y;
          Console.Write("x+y=");
          return z;
        };
        Console.WriteLine("{0}", myT(2,2));
        Console.Read();
      }
    }
}
```

Пояснения даны по тексту программы.

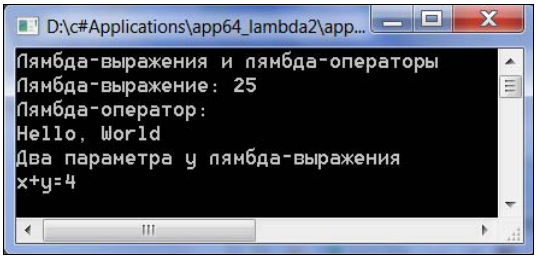


Рис. 13.7. Результаты работы лямбда-выражения и лямбда-операторов

ГЛАВА 14

Введение в запросы LINQ

Как правило, создаваемое приложение нуждается в работе с источником данных (и не с одним). Источники могут быть самые разные: это — и реляционные базы данных, и коллекции данных в памяти, и простые массивы, и файлы типа XML. Чтобы достать необходимые данные из источника данных, формируют так называемые запросы к источнику данных на специальном языке запросов. То есть формально запрос представляет собой выражение, извлекающее данные из источника данных. Для разных типов источников данных были разработаны различные языки, например SQL для реляционных баз данных и XQuery для XML. Таким образом, для каждого типа источника данных или формата данных разработчики вынуждены изучать новый язык запросов, который они должны поддерживать. Технология LINQ упрощает ситуацию, предлагая единообразную модель для работы с данными в различных видах источников и форматов данных. В запросе LINQ работа всегда осуществляется с объектами. Для запросов и преобразований данных в XML-документах, базах данных SQL, наборах данных ADO.NET, коллекциях .NET и любых других форматах, для которых доступен LINQ, используются одинаковые базовые шаблоны кодирования.

LINQ (Language Integrated Query, интегрированный языковой запрос) — проект Microsoft по добавлению синтаксиса языка запросов, напоминающего SQL, в языки программирования платформы .NET Framework. Представляет собой не что иное, как программирование, замаскированное под синтаксис SQL. Сам язык запросов LINQ встроен в грамматику языка C#. Вообще LINQ — это термин, описывающий общий подход доступа к данным. В зависимости от того, где используется LINQ, существуют следующие разновидности этой технологии:

- LINQ to Objects — позволяет применять запросы LINQ к массивам и коллекциям;

- LINQ to XML — позволяет применять запросы LINQ к документам XML;
- LINQ to DataSet — позволяет применять запросы LINQ к объектам DataSet из ADO.NET;
- LINQ to Entities — позволяет применять запросы LINQ внутри API-интерфейса ADO.NET Entitie Framework;
- Parallel LINQ — позволяет применять параллельную обработку данных, возвращенных запросом LINQ.

Для работы с LINQ to Objects необходимо подключать к приложению пространство имен `System.Linq`.

Три части операции запроса

Все операции запроса LINQ состоят из трех различных действий:

- получение источника данных;
- создание запроса;
- выполнение запроса.

Рассмотрим выражение этих трех частей операции запроса в исходном коде. В качестве источника данных для удобства будем использовать массив целых чисел. Тем не менее, те же принципы применимы и к другим источникам данных. Пример программы показан в листинге 14.1, результат ее работы — на рис. 14.1.

Листинг 14.1

```
/* Created by SharpDevelop.
 * User: user
 * Date: 24.12.2012
 * Time: 14:12
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Linq;  // Для работы с запросом технологии
                    // LINQ  to Objects

namespace app65_linq
{
  class Program
```

```
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Работа с запросом LINQ");

      // Три части LINQ-запроса:
      //  1. Источник данных.
      int[] n = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

      Console.WriteLine("Массив для выборки:");
      foreach (int num in n)
      { Console.WriteLine("{0} ", num); }

      // 2. Создание запроса.
      //  Query — типа IEnumerable<int>
      var Query =
        from num in n
        where (num % 2) == 0  // Выбрать все четные числа
        select num;

      // 3. Выполнение запроса

      Console.WriteLine("Результат(четные числа " +
                        "массива):");
      foreach (int num in Query)
      { Console.Write("{0} ", num); }
      Console.Read();
    }
  }
}
```

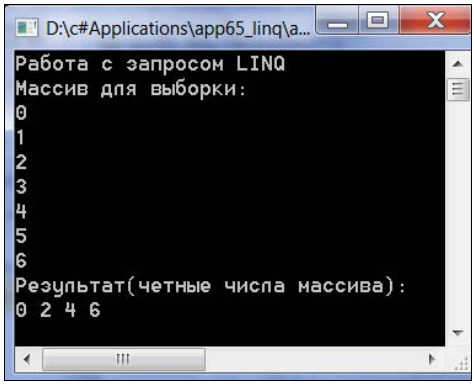


Рис. 14.1. Результат выборки четных чисел из массива по запросу LINQ

В предыдущем примере источником данных является массив, поэтому он неявно поддерживает универсальный интерфейс `IEnumerable<T>`. Это значит, что к нему можно выполнять запросы с LINQ. Запрос выполняется в операторе `foreach`, и оператору `foreach` требуется интерфейс `IEnumerable` или `IEnumerable<T>`.

В запросе встречаются ключевые слова `from`, `in`, `where`, `select`. Это операции запроса. Вот операции запроса LINQ:

- `from`, `in` — позволяют извлекать данные из нужного контейнера. В нашем примере — из `n` в `num`;
- `where` — используется для определения ограничений о том, какие данные должны извлекаться. В нашем примере дословно надо читать так: "где `num` (это очередной извлекаемый элемент, который подлежит проверке на введенное ограничение) делится на 2 без остатка";
- `select` — используется для выбора последовательности из контейнера. Читается "выбрать";
- `join`, `on`, `equals`, `into` — выполняют присоединение данных на основе заданного ключа, не меняя структуры данных в базе данных;
- `orderby`, `ascending`, `descending` — позволяют отсортировать выбранные по запросу данные в порядке возрастания или убывания;
- `group`, `by` — позволяют создавать подмножество данных (группу), сгруппированных по указанному значению.

После выполнения запроса данные можно трансформировать в различной манере (располагать в обратном порядке, преобразовывать в коллекции, массивы, делать из данных как из множеств объединения, пересечения, агрегировать результаты, находить максимальные и минимальные элементы) с помощью методов, которые находятся в классе `System.Linq.Enumerable`. Это методы `Revers<>()`, `ToList<>()`, `ToArray<>()` и др.

В рамках нашего примера мы видим так называемый почти базовый запрос, состоящий из элементов `from`, `in`, `where`, `select`. Операция `where` в состав базового запроса не входит. Структура базового запроса должна строго соблюдаться:

```
var результат (переменная) = from объект in контейнер select
член_объекта
```

Здесь *результат* — результат запроса; *переменная* — переменная типа `var`; *объект* — некий объект, члены которого должны выбираться; *кон-*

тейнер — контейнер, из которого пойдет выборка; *член_объекта* — конкретный член объекта для выборки

В случае нашего примера "некий объект" — это переменная `num` (имя мы можем, естественно, давать любое) — элемент из контейнера, который совпадает с конкретным членом выборки, указанным в `select`. Чтобы детально понять, что же это все-таки за "некий объект", посмотрим на приложение, приведенное в листинге 14.2.

Листинг 14.2

```
/* Created by SharpDevelop.
 * User: user
 * Date: 24.12.2012
 * Time: 16:40
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Linq;

namespace app66_linq
{
  class Program
  {
    public  class dict
    {
      public int a
      {get; set;}

      public int b
      {get; set;}

      public dict(int A, int B)
      { a=A; b=B; }

      static void Main()
      { Console.WriteLine("Работа с запросом LINQ");

        // Три части LINQ-запроса:
        //  1. Источник данных.
        dict [] n = new dict[4];
```

```
        // Для инициализации создаем объекты,
        // и конструктор станет инициализировать
        // эти объекты
        n[0] = new dict(1,1);
        n[1] = new dict(2,2);
        n[2] = new dict(3,3);
        n[3] = new dict(4,4);;

        Console.WriteLine("Массив для выборки " +
                          "(элементы a,b):");
        foreach (dict num in n)
        { Console.WriteLine("{0} {1} ", num.a, num.b); }

        // 2. Создание запроса.
        // Query — типа IEnumerable<int>
        var Query =
          from num in n
          where (num.a % 2) == 0  // Выбрать все
                                  // четные числа a
          select num.a;

        // 3. Выполнение запроса
        Console.WriteLine("Результат(четные a-числа "+
                          "массива):");
        foreach (int num in Query)
        {
          Console.WriteLine("{0} ", num.ToString());
        }
        Console.WriteLine("Количество выбранных " +
                        "элементов: {0}", Query.Count());
        Console.Read();
      }
    }
  }
}
```

Это приложение — результат модификации программы из листинга 14.1. Здесь вместо одномерного числового массива задан одномерный массив `dict` пар чисел. Видим, что в запросе переменная `num` и есть объект-элемент массива, а `num.a`, указанный в операторе `select`, — конкретное значение члена объекта, по которому идет выборка.

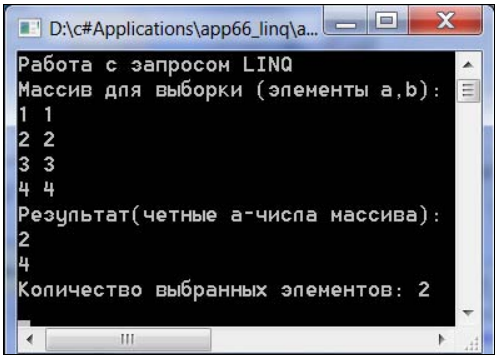


Рис. 14.2. Результат выборки по запросу LINQ

Результат выборки представлен на рис. 14.2.

Несколько модифицируем приложение из листинга 14.2, чтобы показать, как сортировать данные с помощью запроса LINQ. Для этого изменим числовые данные в объявленном массиве пар чисел, а также введем в запрос оператор сортировки. Приложение приведено в листинге 14.3, а результат — на рис. 14.3.

Листинг 14.3

```
/* Created by SharpDevelop.
 * User: user
 * Date: 24.12.2012
 * Time: 16:40
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
namespace app67_linq_sort
{
  using System;
  using System.Linq;

  class Program
  {
    public  class dict
    {
      public  int a
      {get; set;}
```

```
    public   int b
    {get; set;}

    public dict(int A, int B)
    { a=A; b=B; }

    static void Main()
    { Console.WriteLine("Работа с запросом LINQ. " +
                        "Сортировка");

      // Три части LINQ-запроса:
      //  1. Источник данных.
      dict [] n = new dict[4];
      // Для инициализации создаем объекты,
      // и конструктор станет инициализировать
      // эти объекты
      n[0] = new dict(11,5);
      n[1] = new dict(2,8);
      n[2] = new dict(3,4);
      n[3] = new dict(14,4);;

      Console.WriteLine("Массив для выборки "+
                        "(элементы a,b):");
      foreach (dict num in n)
      { Console.WriteLine("{0} {1} ", num.a, num.b); }

      // 2. Создание запроса.
      // Query — типа IEnumerable<int>
      var Query =
        from num in n
//      where (num.a % 2) == 0
          // Выбрать все четные a-числа
      orderby num.b
      select num.b;

      // 3. Выполнение запроса
      Console.WriteLine("Результат (отсортированная " +
                        "выборка по b-числам массива):");
      foreach (int num in Query)
      { Console.WriteLine("{0} ", num.ToString()); }
      Console.Read();
    }
  }
 }
}
```

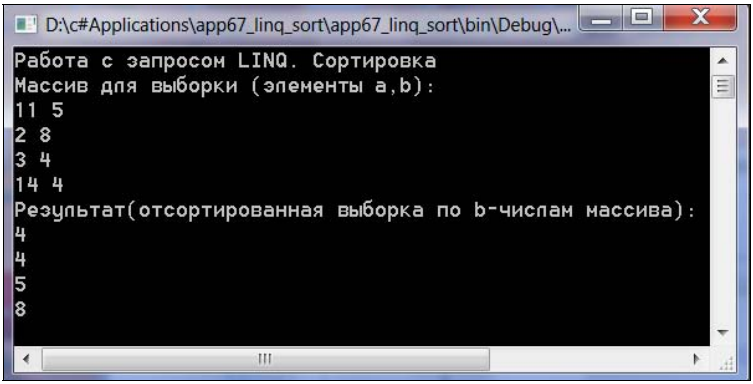


Рис. 14.3. Результат сортировки выборки по запросу LINQ

С помощью запроса LINQ можно сортировать сам массив в том или ином разрезе его элементов, делая ложную выборку, т. е. выбирая весь массив и применяя к нему операцию сортировки. Этот вариант показан в приложении, приведенном в листинге 14.4. Результат работы приложения показан на рис. 14.4.

Листинг 14.4

```
/* Created by SharpDevelop.
 * User: user
 * Date: 24.12.2012
 * Time: 16:40
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
namespace app68_linq_sort
{
  using System;
  using System.Linq;

  class Program
  {
    public  class dict
    {
      public int a
      {get; set;}
```

```
    public   int b
    {get; set;}

    public dict(int A, int B)
    { a=A; b=B; }

    static void Main()
    { Console.WriteLine("Работа с запросом LINQ. " +
                        "Сортировка");

      // Три части LINQ-запроса:
      //  1. Источник данных.
      dict [] n = new dict[4];
      // Для инициализации создаем объекты,
      // и конструктор станет инициализировать
      // эти объекты
      n[0] = new dict(11,5);
      n[1] = new dict(2,8);
      n[2] = new dict(3,4);
      n[3] = new dict(14,4);;

      Console.WriteLine("Массив для выборки " +
                        "(элементы a,b):");
      foreach (dict num in n)
      { Console.WriteLine("{0} {1} ", num.a, num.b); }

      // 2. Создание запроса.
      // Query — типа IEnumerable<int>
      var Query =
        from num in n
        orderby num.b
        select num;

      // 3. Выполнение запроса
      Console.WriteLine("Результат (отсортированный " +
                        "по b-числам массив ):");
      foreach (dict num in Query)
      { Console.WriteLine("{0} {1} ", num.a, num.b); }
      Console.Read();
    }
  }
 }
}
```

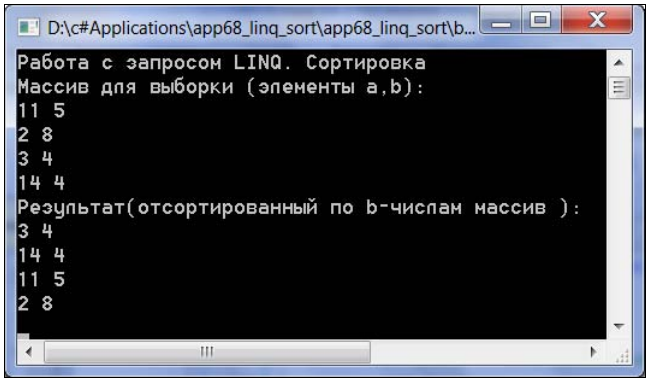


Рис. 14.4. Сортировка массива на основе ложной выборки

Примечание

По умолчанию сортировка происходит в режиме `ascending` (по возрастанию), т. е. строковые данные сортируются по алфавиту, числовые — по возрастанию чисел. Если требуется сортировка в обратном порядке, то к оператору `orderby <элемент, по которому пойдет упорядочение>` надо добавить атрибут `descending` (по убыванию).

Применение запроса LINQ для работы с множествами данных и для агрегирования данных множеств приведено в приложении, показанном в листинге 14.5, а результат работы — на рис. 14.5.

Листинг 14.5

```
/* Created by SharpDevelop.
 * User: user
 * Date: 25.12.2012
 * Time: 13:20
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */

using System;
using System.Linq;
using System.Collections.Generic;

namespace app69_linq_except
{
  class Program
```

```
{
  // Методы работы с множествами данных с помощью
  // запросов LINQ

  // Получение разности массивов
  static void DisplayDiff()
  {
    List<string> myCars = new List<String> {"Yugo",
                                  "Aztec", "BMW"};
    List<string> yourCars = new List<String> {"BMW",
                                "Saab", "Aztec"};

    Console.WriteLine("Исходные множества:");
    foreach (string s in myCars)
      Console.Write("{0}\t",s);
    Console.WriteLine();
    foreach (string s in yourCars)
      Console.Write("{0}\t",s);
    Console.WriteLine();

    // Извлекаются элементы первого множества,
    // отсутствующие во втором множестве
    var carDiff = (from c in myCars select c)
     .Except(from c2 in yourCars select c2);
    Console.WriteLine("Разность между двумя " +
                      "множествами типов автомобилей");
    foreach (string s in carDiff)
      Console.WriteLine(s); // Выводит разность
                            // множеств, равную Yugo
  }

  // Получение пересечения массивов
  static void Displaylntersection()
  {
    List<string> myCars = new List<String> { "Yugo",
                                   "Aztec", "BMW" };
    List<string> yourCars = new List<String> { "BMW",
                                  "Saab", "Aztec" };
    // Получить общие элементы
    var carlntersect = (from c2 in myCars select c2)
     .Intersect(from c2 in yourCars select c2);
```

```
  Console.WriteLine("\nОбщие элементы двух " +
                    "множеств:");
  foreach (string s in carIntersect)
    Console.WriteLine(s); // выведет Aztec и BMW
}

// Объединение массивов (из повторяющихся
// элементов выводится только один)
static void DisplayUnion ()
{
  List<string> myCars = new List<String> { "Yugo",
                                  "Aztec", "BMW" };
  List<string> yourCars = new List<String> { "BMW",
                                  "Saab", "Aztec" };
  // Объединение двух контейнеров
  var carUnion = (from c3 in myCars select c3)
   .Union(from c3 in yourCars select c3);
  Console.WriteLine("\nОбъединенное множество " +
                    "(без повторов):");
  foreach (string s in carUnion)
    Console.WriteLine (s); // Выведет все общие члены
  Console.WriteLine();
}

// Полное объединение двух множеств (конкатенация)
static void DisplayConcat ()
{
  List<string> myCars = new List<String> { "Yugo",
                                  "Aztec", "BMW" };
  List<string> yourCars = new List<String> { "BMW",
                                  "Saab", "Aztec" };

  // Получить сцепление содержимого двух контейнеров
  var carConcat = (from c4 in myCars select c4)
   .Concat(from c4 in yourCars select c4);
  Console.WriteLine ("\nОбъединенное множество " +
                  "(с повторами — конкатенация):");
  foreach (string s in carConcat)
    Console.WriteLine (s);
}

// Исключение дублирования повторяющихся элементов
static void DisplayConcatNoDups ()
```

```
{
  List<string> myCars = new List<String> { "Yugo",
                                  "Aztec", "BMW" };
  List<string> yourCars = new List<String> { "BMW",
                                "Saab", "Aztec" };
  var carConcat = (from c5 in myCars select c5)
   .Concat(from c5 in yourCars select c5);
  // Выведет Yugo Aztec BMW Saab.

  Console.WriteLine ("\nИсключение повторов " +
                     "из конкатенации множеств:");
  foreach (string s in carConcat.Distinct ())
    Console.WriteLine (s);
}

// Различные примеры агрегации данных
static void AggregateOps ()
{
  double[] winterTemps = {5.0, -17.3, 8, -4,
                          0, -10.6};
  Console.WriteLine("\nМножество зимних температур:");
  foreach(double t in winterTemps)
    Console.Write("{0}\t",t);

  // Различные примеры агрегации

  // Выбрать максимальное значение из множества
  // температур
  Console.WriteLine("\nМаксимальная температура: {0}",
    (from t in winterTemps select t).Max());

  Console.WriteLine("Минимальная температура: {0}",
    (from t in winterTemps select t).Min());
  Console.WriteLine("Средняя температура: {0}",
    (from t in winterTemps select t).Average ());
  Console.WriteLine("Сумма всех температур: {0}",
    (from t in winterTemps select t).Sum());
}

static void Main()
{
  Console.WriteLine("Работа с множествами и " +
                    "агрегация данных множества");
  DisplayDiff();
```

```
        DisplayIntersection();
        DisplayUnion();
        DisplayConcat();
        DisplayConcatNoDups();
        AggregateOps();

        Console.Read();
    }
  }
}
```

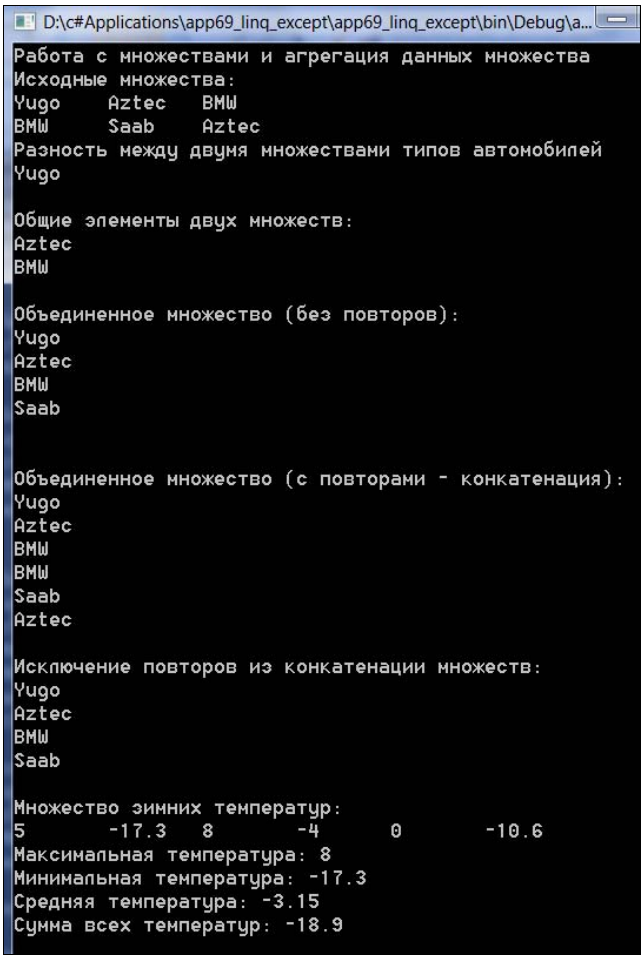


Рис. 14.5. Работа с множествами с помощью LINQ-запросов

О применении типа *var* в запросе

Выполнившийся запрос зачастую возвращает типы данных, не известные к началу компиляции приложения, отчего при выполнении приложения могут возникнуть проблемы в виде появления исключительных ситуаций. Поэтому применение типа `var` в запросе логически оправдано: и компиляция пройдет, и исполнение программы тоже, потому что тип определится как раз в момент исполнения.

ГЛАВА 15

Некоторые сведения о процессах и потоках Windows

Под *процессом* подразумевается выполняющаяся программа. В исполняющей среде процесс описывается набором ресурсов, необходимых для исполнения программы (внешние библиотеки и главный поток программы) и необходимой памятью для выполнения программы. Для каждого загружаемого в память исполняемого exe-файла создается свой изолированный процесс. Благодаря такой изоляции приложений создается определенная стабильность в исполняющей среде, поскольку выход из строя одного приложения не влияет на работу остальных. Изоляция состоит и в том, что доступ к данным одного процесса к данным другого процесса невозможен, если только не применяется специальный интерфейс распределенных вычислений для работы в распределенной среде.

Каждый процесс в целях управления им получает свой специальный идентификатор, называемый PID (Process Identifier), и может независимо загружаться и выгружаться операционной системой. Выполняющиеся на данном компьютере процессы можно просмотреть, если вызвать диспетчер задач (<Ctrl>+<Shift>+<Esc>).

В каждом процессе содержится первоначальный — главный — поток, который начинается от входной точки приложения — функции `Main()`. Поток — это путь исполнения приложения внутри заданного процесса. При вызове метода `Main()` главный поток создается автоматически. Процессы, которые имеют только один поток (однопоточные процессы), являются безопасными к потокам, т. к. в любой момент доступ к данным приложениям может получать один и только один поток. Ему не с кем делиться ни временем, ни данными. Но так бывает не всегда. Су-

ществуют ситуации, когда единственный поток приложения сильно замедляет решение задачи из-за каких-то длительных операций. Например, таких как вывод на печать большой группы данных. Естественно, не выгодно ждать полного окончания вывода данных на печать, когда в приложении требуется еще много чего выполнить. Например, пользователь приложения в момент печати мог бы просматривать базу данных или выполнять иные какие-то действия, которые заложены в приложении. То есть, как говорят, работать асинхронно. А для организации таким образом работы создаются (порождаются главным потоком с помощью специального метода `CreateThread()`) новые потоки, в каждом из которых выполняется своя группа действий. Эти вторичные потоки разгружают главный поток и ускоряют процесс решения задачи. Но при организации вторичных потоков приходится делиться ресурсами машины, во-первых. А во-вторых, потокобезопасность снижается, т. к. может выйти из строя какой-то поток. Завершение какого-то вторичного (рабочего) потока, как и его запуск, не влияет ни на главный поток, ни на все остальные, т. к. все они изолированы друг от друга. Как физически происходит многопотоковая работа? Если процессор машины не поддерживает многопоточную работу, то происходит так называемое квантование времени: поочередно время и разделяемые ресурсы, естественно, выделяются каждому потоку. Остальные потоки ждут своей очереди на получение доступа к процессору и ресурсам.

Процесс исполняется за счет запуска и выполнения неких программ в формате DLL или EXE, которые подключаются к нему. Эти программы называются *сборками* или *модулями*.

А как организовано взаимодействие между процессами и потоками в рамках платформы .NET? Конечно, за счет библиотек базовых классов. Существует пространство имен `System.Diagnostics`, содержащее типы, которые обеспечивают это взаимодействие. Далее приводятся некоторые типы из `System.Diagnostics`.

- `Process` — предоставляет инструменты доступа к локальным и удаленным процессам, позволяет запускать и останавливать процессы программным способом.
- `ProcessModule` — предоставляет модуль типа dll или exe, который может загружаться в определенный процесс.
- `ProcessModuleCollection` — позволяет создавать строго типизированную коллекцию объектов типа `ProcessModule`.
- `ProcessStartInfo` — позволяет указывать набор значений, которые можно задавать при запуске процесса методом `Process.Start()`.

- `ProcessThread` — с его помощью можно получить сведения о потоке внутри определенного процесса.
- `ProcessThreadCollection` — позволяет создавать строго типизированную коллекцию объектов типа `ProcessThread`.

Вот некоторые свойства и методы класса `System.Diagnostics.Process`:

- `ExitTime` — свойство позволяет извлекать значение даты и времени, связанное с процессом, который завершил свою работу;
- `Id` — свойство позволяет получить идентификатор PID заданного процесса;
- `MachineName` — свойство позволяет получить имя компьютера, на котором выполняется заданный процесс;
- `MainWindowTitle` — свойство позволяет получить заголовок главного окна процесса;
- `Moduls` — свойство позволяет получить доступ к строго типизированной коллекции модулей, которые были загружены в рамках текущего процесса;
- `ProcessName` — свойство позволяет получить имя процесса, совпадающее с именем приложения;
- `Responding` — свойство позволяет получить значение, которое показывает, реагирует ли пользовательский интерфейс заданного процесса на действия пользователя, т. е. не завис ли процесс;
- `StartTime` — свойство позволяет получить сведения о том, когда был запущен заданный процесс;
- `Threads` — свойство позволяет получить сведения, какие потоки выполняются в рамках заданного процесса;
- `CloseMainWindow()` — метод позволяет завершать процесс, обладающий пользовательским интерфейсом, завершать процесс за счет посылки в его главное окно сообщения о закрытии;
- `GetCurrentProcess()` — метод возвращает объект типа `Process`, представляющий активный в данный момент процесс;
- `GetProcesses()` — метод возвращает массив объектов типа `Process`, представляющий активные в данный момент процессы;
- `Kill()` — метод позволяет немедленно останавливать заданный процесс;
- `Start()` — метод позволяет запускать процесс.

Вывод списка процессов

Текст приложения, выводящего на экран список процессов, запущенных на текущем компьютере, приведен в листинге 15.1, а результат в виде фрагмента списка — на рис. 15.1.

Листинг 15.1

```
/* Created by SharpDevelop.
 * User: user
 * Date: 26.12.2012
 * Time: 13:29
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Diagnostics;
using System.ComponentModel;
using System.Linq;

namespace app70_Windows_Processes
{ /* Получение списка всех процессов, которые выполняются
     на текущем компьютере, упорядоченных по PID*/

  class Program
  {
    public static void ListAllRunningProcesses()
    {
      var runmngProcs =
        from proc in Process.GetProcesses (".") orderby proc.Id
select proc;
      foreach (var p in runmngProcs)
      {
        string info =
          string.Format("-> PID: {0}\tName: {1}",
          p.Id, p.ProcessName);
        Console.WriteLine (info);
      }

Console.WriteLine("***********************************");
    }
```

```
    public static void Main(string[] args)
    {
      Console.WriteLine("Процессы, запущенные " +
                        "на текущем компьютере");
      ListAllRunningProcesses();
      Console.Read();
    }
  }
}
```

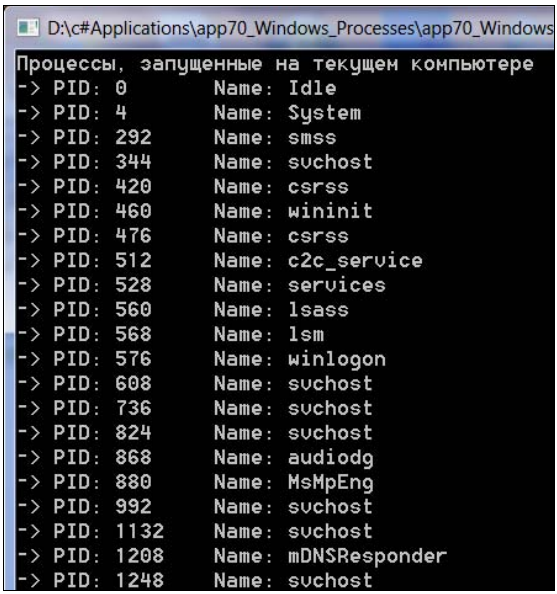


Рис. 15.1. Фрагмент списка запущенных на текущем компьютере процессов

Метод `GetProcesses()` имеет заголовок `public static Process[] GetProcesses()`. Этот статический метод выдает массив объектов типа `Process`. Одним из членов класса `Process`, который используется в приложении, является элемент `Id` — уникальный идентификатор процесса, другим членом является член `ProcessName` — имя процесса с идентификатором `Id`. Метод `GetProcesses()` требует в качестве аргумента имя компьютера. Если указать вместо имени точку, то это будет означать, что имеется в виду текущий компьютер. `Format` является методом класса `string` и задает форматы вывода строковых данных. Обычно мы писали

формат вывода прямо в аргументах метода `WriteLin()`. Здесь для разнообразия форматирование выделено в отдельную позицию: сначала в переменной `info` формируется форматная строка, которая затем подставляется как аргумент в метод `WriteLin()`. Так как `GetProcesses()` выдает массив объектов, то массив можно сортировать, что и делается запросом `Linq`, для этого подключено пространство имен `System.Linq`.

Вывод информации по процессу

Так как в предыдущем приложении мы вывели список процессов, то теперь можно взять идентификатор любого процесса и выдать информацию по нему, воспользовавшись методом `GetCurrentProcess()`. Приложение, реализующее эту задачу, показано в листинге 15.2, а результат работы — на рис. 15.2.

Листинг 15.2

```
/* Created by SharpDevelop.
 * User: user
 * Date: 26.12.2012
 * Time: 14:27
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Diagnostics;
using System.ComponentModel;
using System.Linq;

namespace app71_GetProcId
{
  class Program
  {
    static void EnumThreadsForPid (int pID)
    {
      Process theProc = null;
      try
      {
        theProc = Process.GetProcessById(pID);
        Console.WriteLine("Имя компьютера: {0}",
                          theProc.MachineName);
```

```
        if(theProc.MachineName == ".")
          Console.WriteLine("Имя компьютера: текущий");
      }
      catch(ArgumentException ex)
      { Console.WriteLine(ex.Message); }
    }
    public static void Main()
    {
      Console.WriteLine("Данные по конкретному " +
                        "запущенному процессу");
      EnumThreadsForPid (1572);
      Console.Read();
    }
  }
}
```

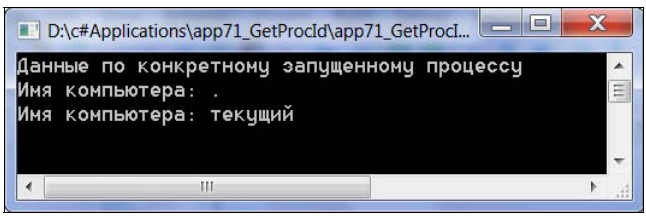


Рис. 15.2. Определение имени компьютера по его идентификатору

Метод `GetProcessById()` выдает значение объекта класса `Process`. Поэтому можно получать значения элементов этого объекта. Мы взяли один элемент — имя машины. Конкретное имя не выдалось, а выдалась точка, как мы видели в листинге 15.1, означающая, что компьютер — текущий. Переменной `theProc` сначала присваивается значение `null`, чтобы потом в блоке `try{}` проконтролировать, присвоится ли настоящая ссылка на процесс переменной `theProc`. Если это по каким-то причинам не произойдет, сработает исключение с выдачей соответствующего сообщения из класса исключения `ArgumentException`.

Потоки процесса

Рассмотрим теперь, как анализировать потоки заданного процесса. В классе `Process` есть свойство `Threads`, которое содержит все потоки в заданном процессе. Это свойство имеет тип `ProcessThreadCollection`,

т. е. через свойство `Threads` класса `Process` мы получаем доступ к элементам класса `ProcessThreadCollection`. Некоторые члены класса `ProcessThreadCollection` показаны в табл. 15.1—15.4.

Таблица 15.1. *Конструктор класса* `ProcessThreadCollection`

Конструктор	Описание
`ProcessThreadCollection`	Перегружен. Инициализирует новый экземпляр класса `ProcessThreadCollection`

Таблица 15.2. *Методы класса* `ProcessThreadCollection`

Метод	Описание
`Add()`	Добавляет поток процесса в коллекцию
`Contains()`	Определяет, существует ли указанный поток процесса в коллекции
`CopyTo()`	Копирует в коллекцию массив экземпляров `ProcessThread` по указанному индексу
`GetEnumerator()`	Возвращает перечислитель, осуществляющий перебор элементов экземпляра класса `ReadOnlyCollectionBase` (унаследовано от `ReadOnlyCollectionBase`)
`IndexOf()`	Предоставляет место указанного потока в коллекции
`Insert()`	Вставляет поток процесса в указанное место в коллекции
`Remove()`	Удаляет поток процесса из коллекции
`ToString()`	Возвращает объект `String`, который представляет текущий объект `Object` (унаследовано от `Object`)

Таблица 15.3. *Свойства класса* `ProcessThreadCollection`

Свойство	Описание
`Count`	Получает число элементов, содержащихся в экземпляре класса `ReadOnlyCollectionBase`
`InnerList`	Возвращает список элементов, содержащихся в экземпляре класса `ReadOnlyCollectionBase`
`Item`	Получает индекс для итерационного перебора набора потоков процесса

Таблица 15.4. *Явные реализации интерфейса*

Интерфейс	Описание
`ICollection.CopyTo`	Копирует целый массив `ReadOnlyCollectionBase` в совместимый одномерный массив `Array`, начиная с указанного индекса конечного массива

Добавим к предыдущему приложению, которое нам давало переменную с конкретным процессом, еще операторы, которые давали бы нам доступ к потокам найденного процесса. Это работа со свойством `Threads` (потоки) объекта `theProc` класса `Process`. Новое приложение приведено в листинге 15.3, а результат его работы — на рис. 15.3.

Листинг 15.3

```
/* Created by SharpDevelop.
 * User: user
 * Date: 26.12.2012
 * Time: 14:27
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Diagnostics;
using System.ComponentModel;
using System.Linq;

namespace app71_GetProcId
{
  class Program
  {
    public static Process theProc;
    static void EnumThreadsForPid (int pID)
    {
      theProc = null; // Process
      try
      {
        theProc = Process.GetProcessById(pID);
        Console.WriteLine("Имя компьютера: {0}",
                          theProc.MachineName);
```

```
        if(theProc.MachineName == ".")
          Console.WriteLine("Имя компьютера: текущий");
      }
      catch(ArgumentException ex)
      { Console.WriteLine(ex.Message); }
    }
    public static void Main()
    {
      Console.WriteLine("Данные по конкретному " +
                        "запущенному процессу");
      EnumThreadsForPid (1572);

      Console.WriteLine("Потоки, используемые " +
                        "процессом: {0}",
                        theProc.ProcessName);
      ProcessThreadCollection theThreads =
           theProc.Threads;
      // У свойства Threads тип ProcessThreadCollection,
      // а значение — массив типа ProcessThread
      foreach(ProcessThread pt in theThreads)
      {
        string info =
        string.Format("-> Thread ID: {0}\tStart Time {1}\tPriority
{2}",
           pt.Id, pt.StartTime.ToShortTimeString(),
           pt.PriorityLevel);
        Console.WriteLine(info);
      }
      Console.WriteLine ("************* ^ " +
                         "********************** \n");
      Console.Read();
    } // Main()
  } // Program
} // NameSpace
```

Пояснений особых не требуется: все, что выводится на экран — элементы пространства `System.Diagnostics`. Это легко проверить, набрав точку после переменной `pt` из цикла `foreach`. При этом откроется окно подсказчика с необходимыми элементами.

Осталось сделать наше приложение годным для исследования любого потока. Для этого надо добавить в него запрос идентификатора процесса

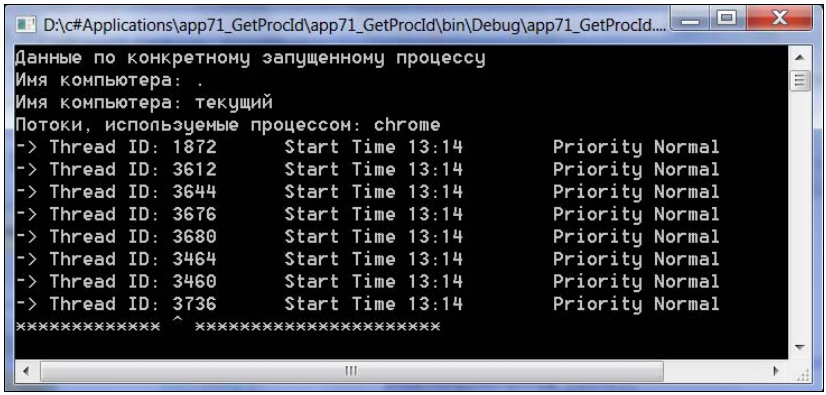


Рис. 15.3. Потоки процесса 1572 chrome и их характеристики

и преобразование введенного в строковом формате числа в тип `int`. Окончательный вид приложения показан в листинге 15.4. Результат работы на двух процессах — на рис. 15.4.

Листинг 15.4

```
/* Created by SharpDevelop.
 * User: user
 * Date: 26.12.2012
 * Time: 14:27
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Diagnostics;
using System.ComponentModel;
using System.Linq;

namespace app71_GetProcId
{
  class Program
  {
    public static Process theProc;
    static void EnumThreadsForPid (int pID)
    {
      theProc = null; // Process
      try
```

```
    {
      theProc = Process.GetProcessById(pID);
      Console.WriteLine("Имя компьютера: {0}",
                        theProc.MachineName);
      if(theProc.MachineName == ".")
        Console.WriteLine("Имя компьютера: текущий");
    }
    catch(ArgumentException ex)
    { Console.WriteLine(ex.Message); }
  }
  public static void Main()
  {
    while(true)
    { // Ввод идентификатора процесса
      Console.WriteLine("Введите идентификатор " +
           "исследуемого процесса (<->) — конец > ");
      Console.Write("PID: ");
      string pID = Console.ReadLine();
      if(pID=="")
        pID = Console.ReadLine();
      if(pID=="-")
        break;
      int theProcID = int.Parse(pID);

      Console.WriteLine("Данные по конкретному " +
                        "запущенному процессу");
      EnumThreadsForPid(theProcID);

      Console.WriteLine("Потоки, используемые " +
                        "процессом: {0}",
                        theProc.ProcessName);
      ProcessThreadCollection theThreads =
        theProc.Threads;
      foreach(ProcessThread pt in theThreads)
      {
        string info =
        string.Format("-> Thread ID: {0}\tStart Time
                                          {1}\tPriority {2}",
          pt.Id, pt.StartTime.ToShortTimeString(),
          pt.PriorityLevel);
        Console.WriteLine(info);
      }
      Console.WriteLine("************ ^ "+
                        "********************** \n");
```

```
        Console.Read();
      } // while
      Console.Read();
    } // Main()
  } // Program
} // NameSpace
```

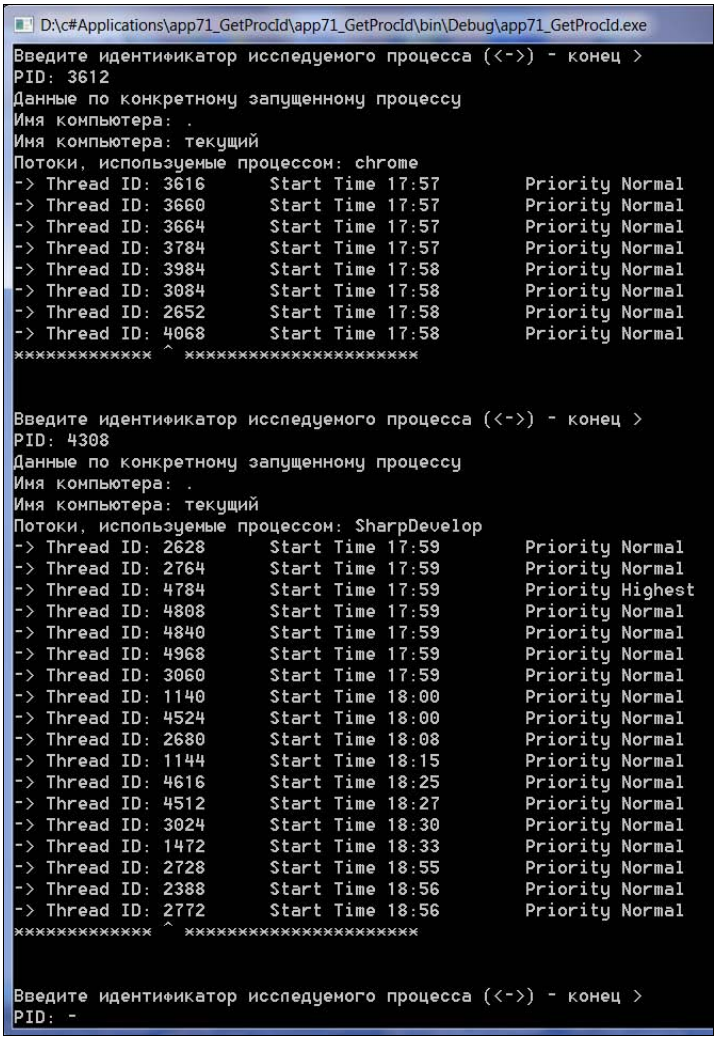


Рис. 15.4. Потоки двух процессов

Уточним, что у свойства `Threads` тип `ProcessThreadCollection`, а значение — массив типа `ProcessThread`. При этом класс `ProcessThread` помимо свойств `Id`, `StartTime`, `PriorityLevel`, примененных в приложении, имеет и другие свойства. Например, свойства:

- `CurrentPriority` — показывает текущий приоритет потока;
- `ProcessorAffinity` — указывает процессоры, на которых может выполняться поток;
- `StartAdress` — показывает, по какому адресу в памяти вызвана функция, запустившая данный поток;
- `ThreadState` — показывает текущее состояние потока;
- `TotalProcessorTime` — показывает время, затраченное процессором на данный поток;
- `WaitReason` — выдает причину, по которой поток находится в состоянии ожидания.

Модули процесса

Ранее мы видели, что в пространстве `System.Diagnostics` имеется класс `ProcessModuleCollection`, который позволяет создавать строго типизированную коллекцию объектов типа `ProcessModule`. Это не что иное, как список всех модулей подключенных к данному процессу. Приложение по выводу такого списка на экран приведено в листинге 15.5, а фрагмент результата его работы — на рис. 15.5.

Листинг 15.5

```
/* Created by SharpDevelop.
 * User: user
 * Date: 27.12.2012
 * Time: 11:05
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Diagnostics;
using System.ComponentModel;
using System.Linq;
```

```
namespace app72_proc_modul
{
  class Program
  {
    public static Process theProc;
    public int pID;
    // Метод вывода списка модулей процесса
    static void EnumModsForPid (int pID)
    {
      Process theProc = null;
      try
      { theProc = Process.GetProcessById(pID); }
      catch(ArgumentException ex)
      { Console.WriteLine(ex.Message);
        return;
      }
      Console.WriteLine("Модули, загруженные " +
        "процессором: {0}", theProc.ProcessName);
      try
      {
        ProcessModuleCollection theMods = theProc.Modules;
        // Тип свойства Modules — ProcessModuleCollection,
        // а значение — массив ProcessModule
        foreach(ProcessModule pm in theMods)
        {
          string info = string.Format("-> Имя модуля: " +
                        "{0}", pm.ModuleName);
                        Console.WriteLine(info);
        }
      }
      catch(ArgumentException ex)
      { Console.WriteLine(ex.Message);
        return;
      }
    } // метод

    // Определение процесса по его идентификатору
    static void EnumThreadsForPid (int pID)
    {
      theProc = null; // Process
      try
```

```
      {
        theProc = Process.GetProcessById(pID);
        Console.WriteLine("Имя компьютера: {0}",
            theProc.MachineName);
        if(theProc.MachineName == ".")
          Console.WriteLine("Имя компьютера: текущий");
      }
      catch(ArgumentException ex)
      { Console.WriteLine(ex.Message); }
    }

    public static void Main()
    {
      while(true)
      { // Ввод идентификатора процесса
        Console.WriteLine ("Введите идентификатор " +
            "исследуемого процесса (<->) — конец > ");
        Console.Write("PID: ");

        string pID = Console.ReadLine();
        if(pID=="")
          pID = Console.ReadLine();
        if(pID=="-")
          break;
        int theProcID = int.Parse(pID);

        EnumThreadsForPid (theProcID);
        EnumModsForPid (theProcID);

        Console.WriteLine ("************* ^ " +
                           "********************** \n");
        Console.Read();
      } // while
      Console.Read();
    } // Main()
  } // Program
} // NameSpace
```

Пояснения — по тексту этой и предыдущих программ, касающихся исследования процессов.

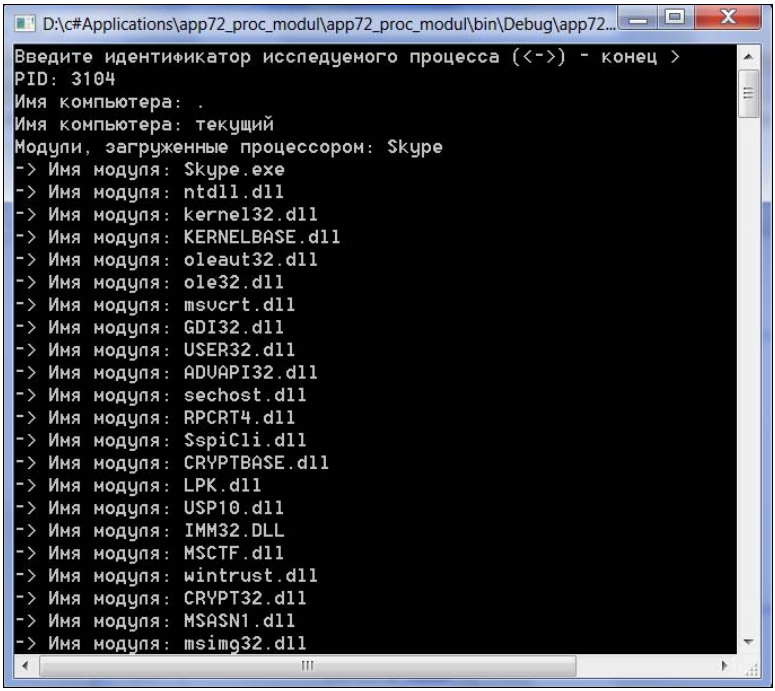


Рис. 15.5. Фрагмент перечня модулей, подключенных к процессу Skype

Запуск и остановка процессов в программе

Мы видели, что в классе `Process` имеются два метода — `Kill()` и `Start()`, позволяющие останавливать и запускать заданный процесс. Приложение, демонстрирующее эту возможность, приведено в листинге 15.6, а его результат — на рис. 15.6.

Листинг 15.6

```
/* Created by SharpDevelop.
 * User: user
 * Date: 27.12.2012
 * Time: 14:07
 *
```

```
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Diagnostics;
using System.Threading;

namespace app74_start_kill2
{
  class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Старт — удаление процесса");

      // Открыть Блокнот.
      Process process = Process.Start("notepad.exe");

      Console.WriteLine("Для закрытия процесса " +
                        "нажмите <Enter>");
      Console.Read();

      process.Kill();
      Console.Read();
    }
  }
}
```

Для закрытия процесса надо щелкнуть мышью в консольном окне и нажать клавишу <Enter>.

Метод `Start()` может принимать объекты класса `System.Diagnostics.ProcessStartInfo`, который предоставляет пользователю дополнительные возможности получения информации при запуске процесса. Описание класса таково:

```
public sealed class ProcessStartInfo : object
{
  public ProcessStartInfo();
  public ProcessStartInfo(string fileName);
  public ProcessStartInfo(string fileName,
                          string arguments);
  public string Arguments { get; set; }
  public bool CreateNoWindow { get; set; }
```

```
  public StringDictionary EnvironmentVariables { get; }
  public bool ErrorDialog { get; set; }
  public IntPtr ErrorDialogParentHandle { get; set; }
  public string FileName { get; set; }
  public bool LoadUserProfile { get; set; }
  public SecureString Password { get; set; }
  public bool RedirectStandardError { get; set; }
  public bool RedirectStandardInput { get; set; }
  public bool RedirectStandardOutput { get; set; }
  public Encoding StandardErrorEncoding { get; set; }
  public Encoding StandardOutputEncoding { get; set; }
  public bool UseShellExecute { get; set; }
  public string Verb { get; set; }
  public string [] Verbs { get; }
  public ProcessWindowStyle WindowStyle { get; set; }
  public string WorkingDirectory { get; set; }
}
```

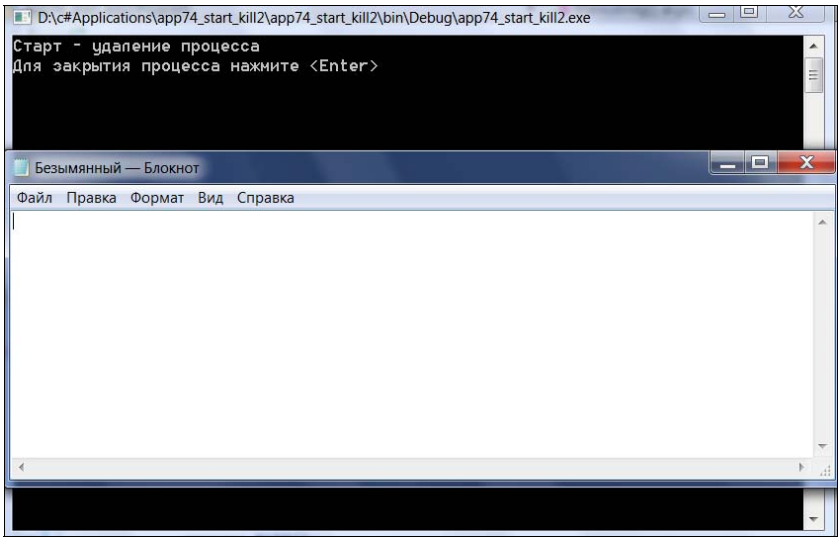


Рис. 15.6. Запуск процесса (программа Блокнот) и его удаление из списка процессов

Описание элементов можно посмотреть в справочной системе MSDN по C#. Покажем пример использования свойства `FileName`, которое позволяет запускать на выполнение файл любого типа, не связанного с запус-

ком какого-либо процесса, при условии, что свойство `StartInfo.UseShellExecute` класса `Process` установлено в `true`. Если оно установлено в `false`, то можно запускать только файлы, которые разрешено запускать с определенными процессами. Приложение приведено в листинге 15.7, а результат — на рис. 15.7. В качестве запускаемого файла взят файл типа JPG.

Листинг 15.7

```
/* Created by SharpDevelop.
 * User: user
 * Date: 27.12.2012
 * Time: 15:26
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Diagnostics;
using System.ComponentModel;

namespace app75_start_info
{
  class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Запуск файла, не связанного " +
                        "с процессом");
      Process myProcess = new Process();
      try
      {
        myProcess.StartInfo.UseShellExecute = true;
        // Можно вызывать любой файл, не связанный
        // с запуском процесса
        myProcess.StartInfo.FileName =
          "F:\\Фото\\9 мая 2012 г..jpg";
        myProcess.StartInfo.CreateNoWindow = true;
        myProcess.Start();
      }
      catch (Exception e)
      { Console.WriteLine(e.Message); }
```

```
            Console.Read();
        }
    }
}
```

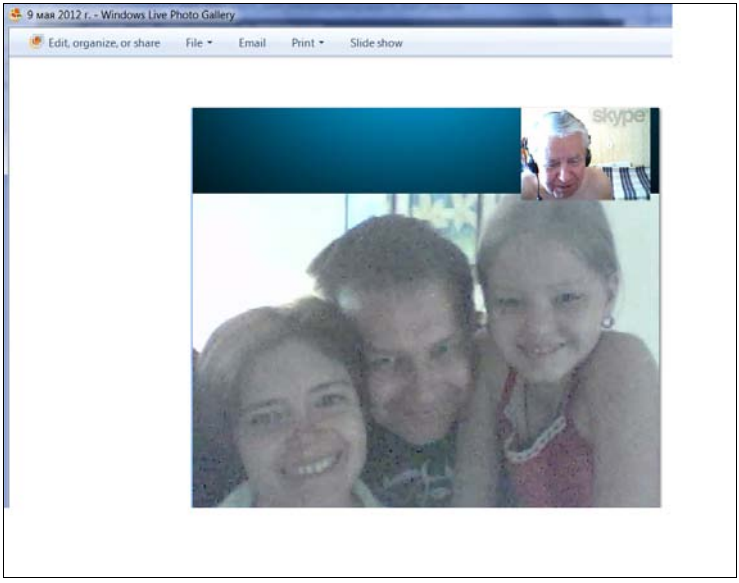


Рис. 15.7. Результат запуска командой `Start()` файла типа JPG

Чувствуете, куда мы добрались? Мы теперь можем из своего приложения запускать любой файл. В том числе и exe-файл. А что это значит? Это значит, что при разработке крупного программного проекта, содержащего, например, 100 режимов, не нужно по каждому режиму делать свою внутреннюю процедуру, которую надо потом вызывать в некотором режиме. И теперь эти 100 процедур не раздуют ваше главное приложение, которое станет плохо управляемым. Можно по каждому режиму делать свое приложение независимо от других, не заботясь даже о пространстве имен, а просто получить из него exe-модуль, который потом вызовется рассмотренным только что способом из головной программы. Можете даже в своем приложении сделать режим прослушивания музыки, задав только папку (это можно делать) с музыкальными файлами, и запускать каждый файл из папки.

ГЛАВА 16

Файловый ввод-вывод

В предыдущих главах мы совершали ввод-вывод, пользуясь стандартными, т. е. заранее определенными в среде разработки средствами ввода-вывода. Это были методы класса `Console`, такие как `Read()`, `ReadLine()`, `Write()`, `WriteLine()` и др. Все эти методы работали со стандартными устройствами ввода-вывода: ввод осуществлялся только с клавиатуры (со стандартного устройства ввода, которому была назначена клавиатура), а вывод — только на экран (на стандартное устройство вывода, которому был назначен экран). Однако в жизни так не всегда бывает: требуется вводить данные из разных файлов и выводить данные также в различные файлы. Данные надо уметь вводить из файла или выводить в файл, а также из памяти или в память. Все эти операции обеспечивает пространство имен `System.IO` (System Input/Output). Это набор базовых библиотек классов, предоставляющих пользователю инструменты файлового ввода-вывода. Основные классы, обеспечивающие функциональность пространства `System.IO`:

- `BinaryReader`, `BinaryWriter` — позволяют работать по вводу-выводу с базовыми типами данных (числовыми, булевыми, строковыми) в двоичном виде;
- `BufferedStream` — предоставляет буфер памяти для временного сохранения в нем потока байтов информации, которая потом должна быть как-то обработана и перенесена в постоянное хранилище;
- `Directory`, `DirectoryInfo` — предоставляют инструменты для работы с каталогами компьютера. Первый класс предоставляет работу со статическими членами (т. е. его элементы можно использовать без создания объекта из этого класса), второй — работу через ссылку на

объект (т. е. его элементы можно использовать после создания объекта из этого класса);

- `DriveInfo` — предоставляет инструменты для работы с дисковыми устройствами компьютера;
- `File`, `FileInfo` — то же, что и классы `Directory`, `DirectoryInfo`, но только обеспечивают работу с файлами, а не с каталогами;
- `FileStream` — обеспечивает работу с файлом как с потоком байтов, поддерживает произвольный доступ к файлу (поиск);
- `FileSystemWatcher` — дает инструменты для отслеживания в определенном каталоге модификаций внешних файлов;
- `MemoryStream` — обеспечивает работу с данными как с потоком байтов в памяти, а не в физическом файле;
- `Path` — обеспечивает операции над строковыми данными, которые содержат информацию о пути к файлу или каталогу;
- `StreamWriter`, `StreamReader` — используются для чтения-записи текстовой информации файла как потока байтов. Не поддерживают произвольный доступ (поиск);
- `Stream` — абстрактный класс, представляющий данные в виде потока байтов;
- `TextReader` — абстрактный класс, который предоставляет инструменты для чтения группы символов из потока байтов или строк с помощью инструментов абстрактных классов `StreamReader` и `StringReader`;
- `TextWriter` — абстрактный класс, который предоставляет инструменты для записи группы символов в поток байтов или строк с помощью инструментов абстрактных классов `StreamWriter` и `StringWriter`.

Рассмотрим некоторые из приведенных классов, которые используются для ввода-вывода данных.

Класс *DirectoryInfo*

Основные члены класса:

- `Create()`, `CreateSubdirectory()` — первый метод создает каталог, второй — подкаталоги по заданному пути;
- `Delete()` — удаляет каталог и все его содержимое;

- `GetDirectories()` — возвращает массив объектов типа `DirectoryInfo`, представляющих все подкаталоги в текущем каталоге;
- `GetFiles()` — возвращает массив объектов типа `FileInfo`, представляющих все файлы в заданном каталоге;
- `MoveTo()` — перемещает каталог со всем его содержимым по заданному пути;
- `Parent` — извлекает родительский каталог данного каталога;
- `Root` — выдает корневую часть пути к данному каталогу.

Так как нам придется работать с путями к каталогам, а эти пути могут состоять из большого множества имен каталогов и подкаталогов, и пути задаются как строковые параметры методов, то возникает проблема, как сократить написание строкового литерала, который представляет путь. Дело в том, что разделитель `\` имен каталогов в записи пути должен иметь двойной обратный слеш (`\\`), который потом компилятор расшифрует как одинарный (та же история, что и с двойными кавычками, когда некий строковый литерал надо выделить кавычками, а он уже находится в другом строковом литерале, и приходится выделять такой двойными кавычками, которые потом компилятор расшифрует как одинарные). А все потому, что одинарный обратный слеш (косая черта) — это признак начала управляющей последовательности (вспомните запись `\n`, `\t`). Чтобы отличить разделитель между именами каталогов от начала управляющей последовательности, надо писать двойной обратный слеш. Это очень утомительная и изобилующая ошибками работа. В C# имеется специальный символ `@`, который упрощает дело. Если посмотреть с более общих позиций, то несмотря на то что в C# нельзя использовать ключевые слова в качестве идентификаторов, любое ключевое слово можно "превратить" в допустимый идентификатор, предварив его символом `@`. Например, идентификатор `@for` вполне пригоден для употребления в качестве допустимого C#-имени. Интересно, что в этом случае идентификатором все-таки является слово `for`, а символ `@` попросту игнорируется. Однако (за исключением специальных случаев) использование ключевых слов в качестве `@`-идентификаторов не рекомендуется. Кроме того, символ `@` может стоять в начале любого идентификатора (а не только созданного из ключевого слова), но это также не считается хорошим стилем программирования.

Буквальный строковый литерал (так он называется) начинается с символа `@`, за которым следует строка, заключенная в кавычки. Содержимое строки в кавычках принимается без какой бы то ни было модификации и может занимать две строки или более. То есть при записи пути как

буквального строкового литерала мы можем не писать двойные обратные слеши, а только одинарные.

Пример приложения работы с некоторыми элементами `DirectoryInfo` показан в листинге 16.1, а результат — на рис. 16.1.

Листинг 16.1

```
/* Created by SharpDevelop.
 * User: user
 * Date: 28.12.2012
 * Time: 12:43
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.IO;

namespace app76_directory_info
{
  class Program
  {
    static void ShowWindowsDirectoryInfo(string path)
    {
      // Вывести информацию о каталоге.
      // Для DirectoryInfo обязательно создается объект
      DirectoryInfo dir = new DirectoryInfo(path);
            // @"C:\Windows"
      Console.WriteLine("***** Directory Info *****");
      // Полное имя:
      Console.WriteLine("FullName: {0}", dir.FullName);
      // Имя каталога:
      Console.WriteLine("Name: {0}", dir.Name);
      // Имя родительского каталога:
      Console.WriteLine("Parent: {0}", dir.Parent);
      // Время создания каталога:
      Console.WriteLine("Creation: {0}",
                        dir.CreationTime);
      // Атрибуты каталога:
      Console.WriteLine("Attributes: {0}",
                        dir.Attributes);
```

```
      // Корневой путь к каталогу:
      Console.WriteLine("Root: {0}", dir.Root);
      Console.WriteLine("************************* \n" );
    }

    public static void Main()
    {
     Console.WriteLine("Работа с каталогами");

      ShowWindowsDirectorylnfo(@"C:\Windows");
      DirectoryInfo dir2 = new DirectoryInfo(@"D:\");
      Console.WriteLine("Создание подкаталога:");

      // Путь к подкаталогу уже не должен содержать имени
      // устройства, т. к. оно уже задано
      // при создании объекта
      dir2.CreateSubdirectory(@"MyNewDir\");
      ShowWindowsDirectorylnfo(@"d:\MyNewDir\");

      // Метод MoveTo().
      // Перемещает всю папку d:\MyNewDir на новое место:
      // фактически переименовывает.
      // Повторный запуск MoveTo() не пройдет:
      // надо закомментировать,
      // т. к. в существующий каталог не пройдет
      // перемещение (переименование)
      DirectoryInfo dir3 =
        new DirectoryInfo(@"d:\MyNewDir\");
      dir3.MoveTo(@"d:\MyNewDir2\");

      // Вывод подкаталогов D:\
      foreach(DirectoryInfo d in dir2.GetDirectories())
        Console.WriteLine("{0} \n",d);
      Console.Read();
    }
  }
}
```

Итак, метод `ShowWindowsDirectorylnfo()` выводит характеристики папки с указанным путем к ней. Далее создается папка `MyNewDir` и выводятся ее характеристики описанным выше методом, который сначала выводит характеристики папки C:\Windows, чтобы было с чем сравнивать,

как выводятся характеристики других папок. Затем применяется метод `MoveTo()`, который фактически оказывается не чем иным, как методом переименования папки. Повторное применение метода к уже новой (ранее полученной им) папке вызовет исключение. Поэтому указанную выше программу повторно запускать не удастся: надо строку с методом закомментировать. Для проверки, сработал ли метод `MoveTo()`, выводятся все подпапки диска D:. (На рис. 16.1 показан только фрагмент вывода с новой папкой `MyNewDir2`. Папка `MyNewDir` исчезла. Это и свидетельствует о переименовании.)

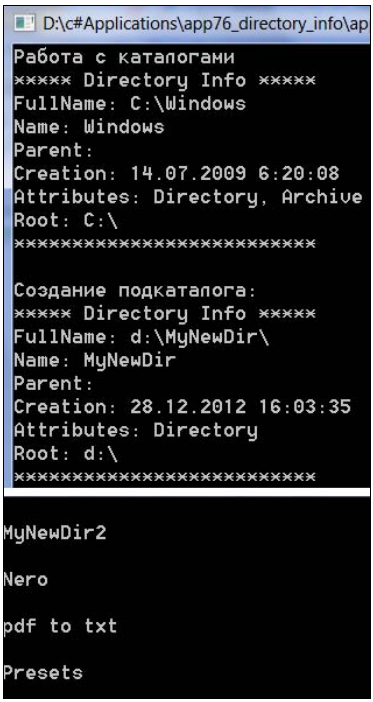


Рис. 16.1. Работа с элементами класса `DirectoryInfo`

Класс *Directory*

Статические члены этого класса по большей части повторяют функциональность класса предыдущего. Только вместо объектов члены класса возвращают строковые данные. Покажем на примере, как вывести на

экран имена всех устройств компьютера, а также попробуем удалить каталог, сформированный с помощью средств `DirectoryInfo`. Соответствующее приложение приведено в листинге 16.2, а результат — на рис. 16.2.

Листинг 16.2

```
/* Created by SharpDevelop.
 * User: user
 * Date: 29.12.2012
 * Time: 12:06
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.IO;

namespace app77_directory
{
  class Program
  {
    static void DirectoryType ()
     {
      // Вывести все дисковые устройства компьютера
      string [] drives = Directory.GetLogicalDrives();
      Console.WriteLine("Дисковые устройства " +
                        "компьютера:");
      foreach (string s in drives)
      Console.WriteLine(" {0} ", s) ;
      // Удалить ранее созданный каталог
      Console.ReadLine();
      try
      {
        Directory.Delete(@"D:\MyNewDir");
        Directory.Delete(@"D:\MyNewDir2",true);
        // Второй параметр указывает,
        // нужно ли удалять все подкаталоги
      }
      catch (IOException e)
      { Console.WriteLine(e.Message); }
    }
```

```
    public static void Main(string[] args)
    {
      Console.WriteLine("Работа с классом Directory");
      DirectoryType ();

      Console.Read();
    }
  }
}
```

Пояснения — по тексту программы.

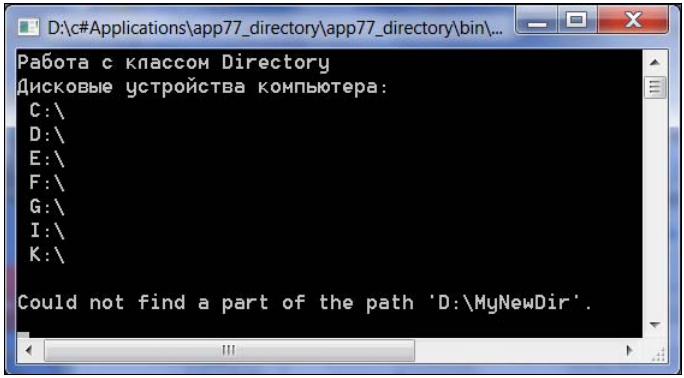


Рис. 16.2. Перечень дисковых устройств компьютера

Класс *DriveInfo*

Этот класс позволяет выводить более разнообразную информацию о дисковых устройствах. Например, о типе привода, метке тома, о доступном свободном пространстве. Пример программы приведен в листинге 16.3, а результат ее работы — на рис. 16.3.

Листинг 16.3

```
/* Created by SharpDevelop.
 * User: user
 * Date: 29.12.2012
 * Time: 12:56
 *
```

```
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.IO;

namespace app78_driveinfo
{
  class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Работа с DriveInfo");

      // Информация обо всех приводах дисковых устройств
      DriveInfo[] myDrives = DriveInfo.GetDrives();
      // Состояние приводов
      foreach(DriveInfo d in myDrives)
      {
        Console.WriteLine("Имя: {0}", d.Name);
        Console.WriteLine("Тип: {0}", d.DriveType);
        // Проверка, смонтирован ли диск
        if (d.IsReady)
        {
          // Свободное пространство
          Console.WriteLine("Свободное пространство: {0}",
                            d.TotalFreeSpace);
          // Формат
          Console.WriteLine("Формат: {0}", d.DriveFormat);
          // Метка тома
          Console.WriteLine("Метка тома: {0}",
                            d.VolumeLabel);
          Console.WriteLine();
        }
        Console.Read();
      }
    }
  }
}
```

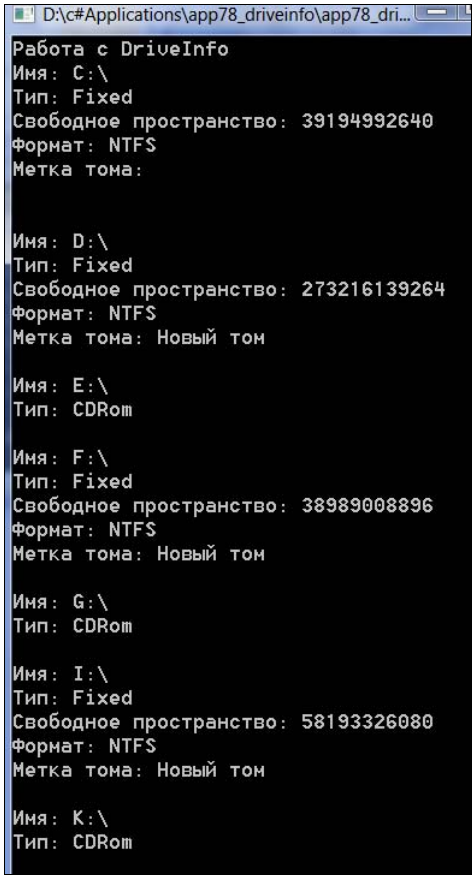


Рис. 16.3. Характеристики дисковых устройств компьютера

Класс *FileInfo*

Этот класс позволяет получать характеристики файлов, расположенных на диске. В нем также есть инструменты для создания, копирования, перемещения и удаления файлов. Перечень некоторых членов этого класса приведен далее.

- `AppendText()` — создает объект `StreamWriter` *(см. далее)* и добавляет текст в файл.
- `CopyTo()` — копирует существующий файл в новый файл.

- `Create()` — создает новый файл и возвращает объект `FileStream` *(см. ниже)* для взаимодействия с вновь созданным файлом.
- `CreateText()` — создает объект `StreamWriter`, который записывает новый текстовый файл.
- `Delete()` — удаляет файл.
- `Directory` — содержит каталог, в котором находится файл.
- `DirectoryName` — содержит полный путь к каталогу, в котором находится файл.
- `Length` — содержит длину файла или каталога.
- `MoveTo()` — перемещает файл в новое местоположение, предоставляя возможность дать файлу новое имя.
- `Name` — содержит имя файла.
- `Open()` — открывает файл с различными привилегиями доступа на чтение-запись.
- `OpenRead()` — открывает доступный только для чтения объект `FileStream`.
- `OpenText()` — создает объект `StreamReader` и читает из соответствующего текстового файла.
- `OpenWrite()` — создает доступный только для записи объект `FileStream`.

Для работы с файлами используется объект класса `FileStream` (**http://msdn.microsoft.com/ru-ru/library/system.io.filestream.aspx**). Этот объект предоставляет инструменты для чтения, записи, открытия и закрытия файлов, для работы, в том числе и со стандартными файлами ввода-вывода, с которыми мы имели дело в предыдущих приложениях. Предоставляются также инструменты для асинхронных операций с файлами, метод поиска с произвольным доступом к файлу.

Создание файла происходит методом `Create()` объекта класса `FileInfo`. Этот метод выдает объект типа `FileStream`, который содержит в себе метод `Close()`, закрывающий файл, т. е. освобождающий все ресурсы машины, выделяемые под так называемый открытый файл. Когда, наоборот, файл открывается, для него выделяются ресурсы. В частности, он связывается со специальной структурой, в которую будут читаться данные и из которой они будут писаться.

Открытие файла осуществляется методом `Open()` из `FileInfo`. Этот метод имеет три параметра: первый — типа `public enum FileMode`, который

имеет сам тип "перечисление". То есть фактически имеем перечисление вида:

```
public enum FileMode
{ CreateNew, Create, Open, OpenOrCreate, Truncate, Append }
```

Одно из значений этого перечисления и надо задавать в качестве фактического параметра в методе `Open()`. Значения элементов перечисления таковы:

- `CreateNew` — создать новый файл. Если файл уже существует, выдается исключение класса `IOException`;
- `Create` — создать новый файл. Если файл уже существует, на его месте создастся новый файл;
- `Open` — открыть существующий файл. Если файл с заданным именем не существует, генерируется исключение класса `FileNotFoundException`;
- `OpenOrCreate` — открыть файл, если он существует. Если файл не существует, он создается;
- `Truncate` — открыть файл и усечь его до нулевой длины;
- `Append` — открыть файл, если он существует, и найти конец файла. Либо создать новый файл. `FileMode.Append` можно использовать только вместе с `FileAccess.Write`. (Это один из режимов открытия файла, указанный во втором параметре, в котором задается способ доступа к файлу: файл открывается для чтения или для записи в него и т. п. В данном случае `Append` подразумевает режим доступа к файлу такой: "Открыть и дописать в конец", т. е. файл открывается для добавки в него чего-то.)

Второй параметр метода `Open()` — это тоже переменная типа перечисления, задающая способ доступа к файлу, который открывается. Перечисление имеет вид:

```
public enum FileAccess
{
   Read,
   Write,
   ReadWrite
}
```

Элементы перечисления означают в порядке их следования, что файл открывается только для чтения, только для записи, для чтения и записи.

Третий параметр метода `Open()` — это тоже переменная типа перечисления, которая задает способ взаимодействия открываемого файла с другими файлами. Перечисление имеет вид:

```
public enum FileShare
{
  None,
  Read,
  Write
  ReadWrite,
  Delete,
  Inheritable
}
```

Элементы перечисления означают:

- `None` — отклоняет совместное использование текущего файла. Любой запрос на открытие файла (данным процессом или другим процессом) не выполняется до тех пор, пока файл не будет закрыт;
- `Read` — разрешает последующее открытие файла для чтения. Если этот флаг не задан, любой запрос на открытие файла для чтения (данным процессом или другим процессом) не выполняется до тех пор, пока файл не будет закрыт;
- `Write` — разрешает последующее открытие файла для записи. Если этот флаг не задан, любой запрос на открытие файла для записи (данным процессом или другим процессом) не выполняется до тех пор, пока файл не будет закрыт;
- `ReadWrite` — разрешает последующее открытие файла для чтения или записи. Если этот флаг не задан, любой запрос на открытие файла для записи или чтения (данным процессом или другим процессом) не выполняется до тех пор, пока файл не будет закрыт;
- `Delete` — разрешает последующее удаление файла;
- `Inheritable` — разрешает наследование дескриптора файла дочерними процессами. В Win32 непосредственная поддержка этого свойства не обеспечена (дескриптор — целое число, связываемое с файлом при его создании, по нему операционная система распознает этот файл).

Рассмотрим, например, работу с методом `AppendText()`. Текст программы представлен в листинге 16.4, а результат — на рис. 16.4.

Листинг 16.4

```
/* Created by SharpDevelop.
 * User: user
 * Date: 29.12.2012
 * Time: 13:52
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.IO;

namespace app79_fileinfo
{
  class Program
  {
    public static void Main(string[] args)
    {
      FileInfo f = new FileInfo(@"D:\Test.txt");
      StreamWriter sw = f.AppendText(); // Для порождения
                                  // объекта StreamWriter
      sw.WriteLine("Строка 1");
      sw.WriteLine("Строка 2");
      sw.WriteLine("Строка 3");
      sw.Close();

      // Печать введенного текста из файла
      string s;
      Console.WriteLine("Содержимое введенного файла");
      StreamReader sr = f.OpenText();
      s = sr.ReadLine();
      Console.WriteLine(s);
      s = sr.ReadLine();
      Console.WriteLine(s);
      s = sr.ReadLine();
      Console.WriteLine(s);
      sr.Close();

      Console.ReadKey(true);
    }
  }
}
```

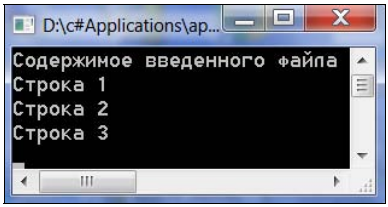


Рис. 16.4. Ввод строк методом `AppendText()`

Метод `AppendText()` сам создает файл, если тот не существует, поэтому в программе нет метода `Create()`. Из типа `FileInfo` создается объект с именем будущего текстового файла и с помощью ссылки на созданный объект вызывается метод `AppendText()`, возвращающий в переменную `sw` ссылку на объект типа `StreamWriter`, в котором имеется метод `WriteLine()`, позволяющий записывать в файл текстовые строки. Чтобы строки в файле сохранились, файл надо закрыть, иначе все потеряется. Далее идет вывод сформированного файла на экран. Для этого надо файл сначала открыть, чтобы получить к нему доступ. Это делает метод `OpenText()`, который формирует в переменной `sr` ссылку типа `StreamReader`. В этом объекте имеется метод `ReadLine()`, которым мы и читаем строки из файла: за одно применение метода читается одна строка, и указатель файла (которого мы не видим) устанавливается на начало следующей строки. Поэтому повторное применение метода `ReadLine()` выдаст в переменную `s` следующую строку. И т. д. Как сказал некто "очень мудрый": "Процесс пошел". В объекте `StreamReader` имеется свой метод закрытия файла — метод `Close()`. Им мы и воспользовались. Но не для того, как в первом случае, чтобы записанное сохранилось в файле, а чтобы освободить занимаемые обработкой файла ресурсы системы. Если мы даже не закроем файл, информация не потеряется. Все равно все файлы закрываются операционной системой при ее выгрузке.

Класс *File*

Этот тип по функциональности во многом совпадает с типом `FileInfo`: в этом классе тоже имеются методы `AppendText()`, `Create()`, `CreateText()`, `Open()`, `OpenRead()`, `OpenWrite()`, `OpenText()`. Однако существует и несколько уникальных членов, способных значительно упростить процесс ввода-вывода текстовых данных. Вот эти члены:

- `ReadAllBytes()` — открывает файл, возвращает двоичные данные в виде массива байтов и закрывает файл;

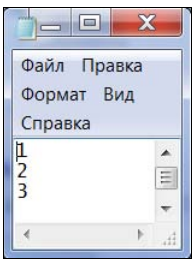
- `ReadAllLines()` — открывает файл, возвращает символьные данные в виде массива строк и закрывает файл;
- `ReadAllText()` — открывает текстовый файл, читает все строки из файла, возвращает прочитанные данные в виде строки `string` и закрывает файл;
- `WriteAllBytes()` — создает новый файл, записывает в него массив байтов данных и закрывает файл;
- `WriteAllLines()` — создает новый файл, записывает в него коллекцию строк данных и закрывает файл. Метод объявлен как `public static void WriteAllLines(string path,IEnumerable<string> contents)`, где `contents` — это коллекция типа `Type`: `System.Collections.Generic.IEnumerable<String>` (строки);
- `WriteAllText()` — создает новый файл, записывает в него массив строк и закрывает файл. Метод объявлен как `public static void WriteAllText(string path, string contents)`, где `contents` — это массив строк.

Применение методов `WriteAllLines()` и `ReadAllText()` показано в приложении, приведенном в листинге 16.5, а результат — на рис. 16.5.

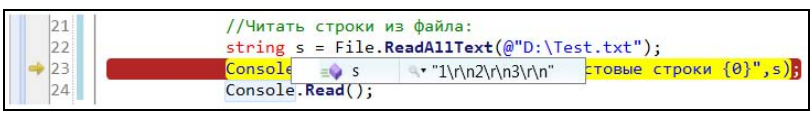
Листинг 16.5

```
/* Created by SharpDevelop.
 * User: user
 * Date: 30.12.2012
 * Time: 13:29
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.IO;
namespace app80_file
{
  class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Работа с типом File");
      string [] MyArr = {"1","2","3"};  // Массив из трех
                                        // строк
```

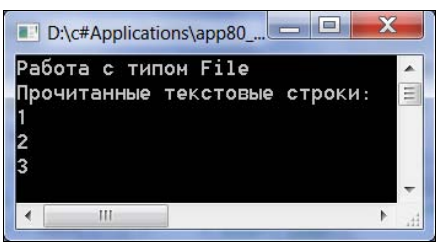
```
        // Записать все строки в новый файл:
        File.WriteAllLines(@"D:\Test.txt",MyArr);
        // Читать строки из файла:
        string s = File.ReadAllText(@"D:\Test.txt");
        Console.WriteLine("Прочитанные текстовые строки: ");
        Console.WriteLine(s);
        Console.Read();
    }
  }
}
```



а



б



в

Рис. 16.5. Результат работы методов `WriteAllLines()` и `ReadAllText()`: *а* — строки, записанные в текстовый файл; *б* — вид строки `s`, содержащей результат работы метода `ReadAllText()`; *в* — строки, выведенные на экран

Обратите внимание на рис. 16.5, в каком интересном виде выдает результат метод `ReadAllText()`, свертывая массив строк из файла в одну строку: проставлены разделители строк, чтобы все не сливалось, что естественно. Разделителями являются символы возврата каретки (\r) и перевода строки (\n).

Класс *Stream*

Это абстрактный класс, представляющий данные в виде потока байтов. До сих пор мы рассматривали примеры работы со строками данных. Но это далеко не весь спектр необходимой работы с данными. Часто в задачах требуется забраться поглубже в данные: не на уровне строк или записей (записями называют элементы файлов, но обычно это сгруппированные данные разных типов, отражающие некие физические сущности, например личную карточку работника; само слово "запись" взято из бухгалтерской терминологии), а на уровне байтов, т. е. на уровне символов или битов в байтах. Для таких целей методы обработки данных нужны другие. И их предоставляет класс `Stream` и его производные классы. В рамках этого класса все данные представляются в виде так называемых потоков данных — последовательностей байтов информации. Какие на самом деле обрабатываются данные — не важно: все эти данные рассматриваются как последовательности байтов информации. И более никак. Поэтому и операции существуют только над байтами: читать байт, группу байтов, сдвинуться в потоке на некоторое количество байтов в ту или иную сторону (к началу потока или к его концу) — вот примеры операций. В абстрактном классе `Stream` определен набор членов, которые поддерживают также синхронную и асинхронную обработку данных в хранилище данных (файле или памяти). Далее приведен список основных членов класса `Stream`:

- `CanRead` — определяет, поддерживает ли текущий поток чтение;
- `CanWrite` — определяет, поддерживает ли текущий поток запись;
- `CanSeek` — определяет, поддерживает ли текущий поток поиск;
- `Close()` — закрывает текущий поток и освобождает занимаемые им ресурсы;
- `Flush()` — обновляет заданный источник данных текущим состоянием буфера (участка памяти, в котором хранятся определенные данные) и очищает затем этот буфер. Если в операции с потоком буфер не участвует, то метод `Flush()` ничего не делает;
- `Length` — содержит длину потока в байтах;
- `Position` — содержит текущую позицию в потоке (указатель в потоке);
- `Read()` — читает определенное количество байтов из потока и перемещает указатель в потоке на прочитанное количество байт вперед;
- `ReadByte()` — то же, что и `Read()`, но для одиночного байта;

- `Seek()` — выдает номер позиции в текущем потоке (это операция поиска);
- `SetLength()` — задает длину текущего потока;
- `Write()` — пишет последовательность байтов в текущий поток и перемещает указатель потока вперед на количество записанных байтов;
- `WriteByte()` — то же, что и метод `Write()`, но для одиночного байта.

Класс *FileStream*

Это производный класс от класса `Stream`. Он может работать с одним байтом или с массивом байтов. Прежде чем составить приложение с работой в рамках нового класса, подумаем, что нам придется превращать в последовательности байтов строки данных: не задавать же массивы данных посимвольно. Это довольно утомительно. Оказывается, в пространстве имен `System.Text` существует специальный класс `Encoding`, члены которого способны кодировать и декодировать строки в массивы байтов и наоборот. Закодированный массив сохраняется в файле с помощью метода `FileStream.Write()`, а чтобы прочитать байты обратно в память, надо сбросить значение свойства `Position` в начало потока и применить метод `ReadByte()`.

Текст приложения приведен в листинге 16.6, а результат — на рис. 16.6.

Листинг 16.6

```
/* Created by SharpDevelop.
 * User: user
 * Date: 30.12.2012
 * Time: 15:33
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.IO;
using System.Text;

namespace app81_filestream
{
  class Program
```

```
  {
    public static void Main()
    {
      Console.WriteLine("Работа с классом FileStream");

      // Открытие файла
      FileStream f= File.Open(@"D:\myTest22.txt",
                              FileMode.Create);
      // Закодировать строку в виде массива байтов
      string msg = "Пример работы с классом FileStream";
      byte[] Arr = Encoding.Default.GetBytes(msg);
      // Свойство Default получает кодировку для текущей
      // кодовой страницы ANSI операционной системы

      // Записать массив в файл
      f.Write(Arr, 0, Arr.Length); /* 0 — это номер байта
                          в потоке, с которого
                          начинается запись в файл */
      // Сброс позиции потока в начало потока
      f.Position = 0;
      // Чтение потока
      Console.Write("Массив байтов из файла " +
                    "(закодированная строка):\n");
      byte[] b = new byte[Arr.Length];
      for (int i = 0; i < Arr.Length; i++)
      {
        b[i] = (byte)f.ReadByte();
        // Метод ReadByte() возвращает байт,
        // переведенный в тип Int32, поэтому надо
        // сделать обратный перевод в тип byte
        Console.Write(b[i]);
      }
      // Вывод декодированного результата
      Console.Write("\nДекодированный результат " +
                    "(то, что было закодировано):\n");
      Console.WriteLine(Encoding.Default.GetString(b));

      Console.Read();
    } // Main()
  }
}
```

Пояснения даны по тексту программы.

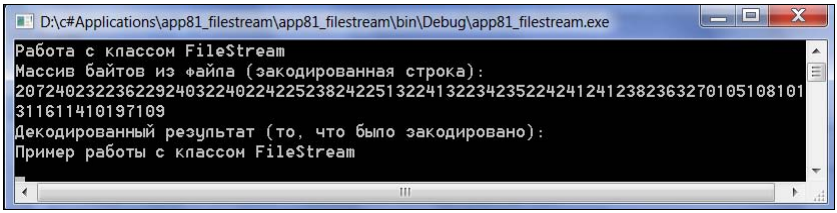


Рис. 16.6. Работа с классом `FileStream`

Классы *StreamWriter*, *StreamReader*

Эти классы удобны для работы с символьными данными (в том числе, конечно, и со строками). По умолчанию они работают в кодировке Unicode, но можно перейти и в другую кодировку через инструменты класса `System.Text.Encoding`. Класс `StreamReader` унаследован от класса `TextReader`, `StreamWriter` — от `TextWriter`.

Далее представлены некоторые члены класса `StreamWriter`.

Конструкторы:

- `StreamWriter(Stream)` — инициализирует новый экземпляр класса `StreamWriter` для указанного потока, используя кодировку UTF-8 и размер буфера по умолчанию;
- `StreamWriter(String)` — инициализирует новый экземпляр класса `StreamWriter` для указанного файла с помощью кодировки по умолчанию и размера буфера;
- `StreamWriter(Stream, Encoding)` — инициализирует новый экземпляр класса `StreamWriter` для указанного потока, используя заданную кодировку и размер буфера по умолчанию;
- `StreamWriter(String, Boolean)` — инициализирует новый экземпляр класса `StreamWriter` для указанного файла с помощью кодировки по умолчанию и размера буфера. Если файл существует, он может быть либо перезаписан, либо в него могут быть добавлены данные. Если файл не существует, конструктор создает новый файл;
- `StreamWriter(Stream, Encoding, Int32)` — инициализирует новый экземпляр класса `StreamWriter` для указанного потока, используя заданную кодировку и размер буфера;
- `StreamWriter(String, Boolean, Encoding)` — инициализирует новый экземпляр класса `StreamWriter` для указанного файла с помощью

указанной кодировки и размера буфера по умолчанию. Если файл существует, он может быть либо перезаписан, либо в него могут быть добавлены данные. Если файл не существует, конструктор создает новый файл;

- `StreamWriter(Stream, Encoding, Int32, Boolean)` — инициализирует новый экземпляр класса `StreamWriter` для указанного потока, используя заданную кодировку и размер буфера, а также при необходимости оставляет поток открытым;
- `StreamWriter(String, Boolean, Encoding, Int32)` — инициализирует новый экземпляр класса `StreamWriter` для указанного файла по заданному пути, используя заданную кодировку и размер буфера. Если файл существует, он может быть либо перезаписан, либо в него могут быть добавлены данные. Если файл не существует, конструктор создает новый файл.

Свойства:

- `AutoFlush` — содержит или задает значение, определяющее, будет ли `StreamWriter` сбрасывать буфер в основной поток после каждого вызова `StreamWriter.Write`;
- `BaseStream` — содержит основной поток, связанный с резервным хранилищем;
- `Encoding` — содержит кодировку `Encoding`, в которой осуществляется запись выходных данных (переопределяет `TextWriter.Encoding`);
- `NewLine` — содержит или задает признак конца строки, используемой текущим `TextWriter`.

Методы:

- `Close()` — закрывает текущий объект `StreamWriter` и базовый поток;
- `Dispose()` — освобождает все ресурсы, используемые объектом;
- `TextWriterDispose(Boolean)` — освобождает неуправляемые (а при необходимости и управляемые) ресурсы, используемые объектом `StreamWriter`;
- `Flush()` — очищает все буферы для текущего средства записи и вызывает запись всех данных буфера в основной поток;
- `FlushAsync` — асинхронно очищает все буферы для этого потока и вызывает запись всех буферизованных данных в базовое устройство;
- `ToString()` — возвращает строку, которая представляет текущий объект;

- `Write(Boolean)` — записывает в текстовую строку или поток текстовое представление значения `Boolean`;
- `Write(Char)` — записывает символ в поток;
- `Write(Char[])` — записывает в поток массив символов;
- `Write(Decimal)` — записывает текстовое представление десятичного значения в текстовую строку или поток;
- `Write(Double)` — записывает в текстовую строку или поток текстовое представление значения с плавающей запятой размером 8 байт;
- `Write(Int32)` — записывает в текстовую строку или поток текстовое представление целого числа со знаком размером 4 байта;
- `Write(Int64)` — записывает в текстовую строку или поток текстовое представление целого числа со знаком размером 8 байт;
- `Write(Object)` — записывает в текстовую строку или поток текстовое представление объекта с помощью вызова метода `ToString()` для этого объекта;
- `Write(Single)` — записывает в текстовую строку или поток текстовое представление значения с плавающей запятой размером 4 байта;
- `Write(String)` — записывает в поток строку;
- `Write(UInt32)` — записывает в текстовую строку или поток текстовое представление целого числа без знака размером 4 байта;
- `Write(UInt64)` — записывает в текстовую строку или поток текстовое представление целого числа без знака размером 8 байт;
- `Write(String, Object)` — записывает форматированную строку в текстовую строку или поток, используя ту же семантику, что и метод `String.Format(String, Object)`;
- `Write(String, Object[])` — записывает форматированную строку в текстовую строку или поток, используя ту же семантику, что и метод `String.Format(String, Object[])`;
- `Write(Char[], Int32, Int32)` — записывает в поток дочерний массив символов;
- `Write(String, Object, Object)` — записывает форматированную строку в текстовую строку или поток, используя ту же семантику, что и метод `String.Format(String, Object, Object)`;
- `Write(String, Object, Object, Object)` — записывает форматированную строку в текстовую строку или поток, используя ту же семантику, что и метод `String.Format(String, Object, Object, Object)`;

- `WriteAsync(Char)` — асинхронно записывает символ в поток;
- `WriteAsync(Char[])` — выполняет асинхронную запись массива символов в текстовую строку или поток;
- `WriteAsync(String)` — асинхронно записывает строку в поток;
- `WriteAsync(Char[], Int32, Int32)` — асинхронно записывает дочерний массив символов в поток;
- `WriteLine()` — записывает признак конца строки в текстовую строку или поток;
- `WriteLine(Boolean)` — записывает в текстовую строку или поток текстовое представление значения `Boolean`, за которым следует признак конца строки;
- `WriteLine(Char)` — записывает в текстовую строку или поток символ, за которым следует признак конца строки;
- `WriteLine(Char[])` — записывает в текстовую строку или поток массив символов, за которыми следует признак конца строки;
- `WriteLine(Decimal)` — записывает в текстовую строку или поток текстовое представление десятичного значения, за которым следует признак конца строки;
- `WriteLine(Double)` — записывает в текстовую строку или поток текстовое представление значения с плавающей запятой размером 8 байт, за которым следует признак конца строки;
- `WriteLine(Int32)` — записывает в текстовую строку или поток текстовое представление целого числа со знаком размером 4 байта, за которым следует признак конца строки;
- `WriteLine(Int64)` — записывает в текстовую строку или поток текстовое представление целого числа со знаком размером 8 байт, за которым следует признак конца строки;
- `WriteLine(Object)` — записывает в текстовую строку или поток текстовое представление объекта путем вызова метода `ToString()` для этого объекта, за которым следует признак конца строки;
- `WriteLine(Single)` — записывает в текстовую строку или поток текстовое представление значения с плавающей запятой размером 4 байта, за которым следует признак конца строки;
- `WriteLine(String)` — записывает в текстовую строку или поток строку, за которой следует признак конца строки;

- `WriteLine(UInt32)` — записывает в текстовую строку или поток текстовое представление целого числа без знака размером 4 байта, за которым следует признак конца строки;
- `WriteLine(UInt64)` — записывает в текстовую строку или поток текстовое представление целого числа без знака размером 8 байт, за которым следует признак конца строки;
- `WriteLine(String, Object)` — записывает форматированную строку и новую строку в текстовую строку или поток, используя ту же семантику, что и метод `String.Format(String, Object)`;
- `WriteLine(String, Object[])` — записывает отформатированную строку и новую строку, используя ту же семантику, что и `Format`;
- `WriteLine(Char[], Int32, Int32)` — записывает в текстовую строку или поток дочерний массив символов, за которыми следует признак конца строки;
- `WriteLine(String, Object, Object)` — записывает форматированную строку и новую строку в текстовую строку или поток, используя ту же семантику, что и метод `String.Format(String, Object, Object)`;
- `WriteLine(String, Object, Object, Object)` — записывает отформатированную строку и новую строку, используя ту же семантику, что и `Format`;
- `WriteLineAsync()` — асинхронно записывает в поток признак конца строки;
- `WriteLineAsync(Char)` — асинхронно записывает в поток символ, за которым следует признак конца строки;
- `WriteLineAsync(Char[])` — асинхронно записывает в текстовую строку или поток массив символов, за которыми следует признак конца строки;
- `WriteLineAsync(String)` — асинхронно записывает в поток строку, за которой следует признак конца строки;
- `WriteLineAsync(Char[], Int32, Int32)` — асинхронно записывает в поток дочерний массив символов, за которыми следует признак конца строки.

Поля:

- `CoreNewLine` — сохраняет символы новой строки, используемые для `TextWriter`;
- `Null`.

Пример приложения, создающего текстовый файл, в который записываются потоком три текстовых строки и двадцать байтов из чисел 0—9 и пробелов между ними, показан в листинге 16.7, а результат работы — на рис. 16.7.

Листинг 16.7

```
/* Created by SharpDevelop.
 * User: user
 * Date: 31.12.2012
 * Time: 18:24
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.IO;

namespace app82_streamwriter
{
  class Program
  {
    public static void Main()
    {
      Console.WriteLine("Работа с классом StreamWriter");
      // CreateText выдает ссылку типа StreamWriter:
      StreamWriter w = File.CreateText (@"D:\Test.txt");

      w.WriteLine("Завтра — Новый год");
      w.WriteLine("Не забудьте поздравить родных " +
                  "и знакомых");
      w.WriteLine("Россияне будут гулять 10 дней подряд");
      // Вставка новой строки
      w.Write (w.NewLine); // NewLine содержит \r\n -
                           // выводит пустую строку

      for(int i = 0; i < 10; i++)
        w.Write (i + " ");

      w.Close();
      Console.Read();
    }
  }
}
```

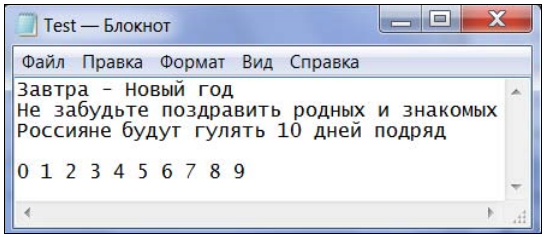


Рис. 16.7. Результат записи потоком программой листинга 16.7

Рассмотрим некоторые члены класса `StreamReader`.

Конструкторы:

- `StreamReader(Stream)` — инициализирует новый экземпляр класса `StreamReader` для заданного потока;
- `StreamReader(String)` — инициализирует новый экземпляр класса `StreamReader` для указанного имени файла;
- `StreamReader(Stream, Boolean)` — инициализирует новый экземпляр класса `StreamReader` для указанного потока с заданным параметром обнаружения метки порядка следования байтов;
- `StreamReader(Stream, Encoding)` — инициализирует новый экземпляр класса `StreamReader` для указанного потока с заданной кодировкой символов;
- `StreamReader(String, Boolean)` — инициализирует новый экземпляр класса `StreamReader` для указанного имени файла с заданным параметром обнаружения метки порядка следования байтов;
- `StreamReader(String, Encoding)` — инициализирует новый экземпляр класса `StreamReader` для указанного имени файла с заданной кодировкой символов;
- `StreamReader(Stream, Encoding, Boolean)` — инициализирует новый экземпляр класса `StreamReader` для указанного потока с заданной кодировкой символов и параметром обнаружения метки порядка следования байтов;
- `StreamReader(String, Encoding, Boolean)` — инициализирует новый экземпляр класса `StreamReader` для указанного имени файла с заданной кодировкой символов и параметром обнаружения метки порядка следования байтов;
- `StreamReader(Stream, Encoding, Boolean, Int32)` — инициализирует новый экземпляр класса `StreamReader` для указанного потока с за-

данной кодировкой символов, параметром обнаружения метки порядка следования байтов и размером буфера;

- `StreamReader(String Encoding, Boolean Int32)` — инициализирует новый экземпляр класса `StreamReader` для указанного имени файла с заданной кодировкой символов, параметром обнаружения метки порядка следования байтов и размером буфера.

Свойства:

- `BaseStream` — возвращает основной поток;
- `CurrentEncoding` — содержит текущую кодировку символов, используемую текущим объектом `StreamReader`;
- `EndOfStream` — содержит значение, определяющее, находится ли позиция текущего потока в конце потока.

Методы:

- `Close()` — закрывает объект `StreamReader` и основной поток и освобождает все системные ресурсы, связанные с устройством чтения;
- `DiscardBufferedData()` — очищает внутренний буфер;
- `Dispose()` — освобождает все ресурсы, используемые объектом `TextReader`;
- `Dispose(Boolean)` — закрывает основной поток, освобождает неуправляемые ресурсы, используемые `StreamReader`, и, при необходимости, освобождает управляемые ресурсы;
- `Peek()` — возвращает следующий доступный символ, но не использует его;
- `Read()` — выполняет чтение следующего символа из входного потока и перемещает положение символа на одну позицию вперед;
- `Read(Char[], Int32, Int32)` — считывает заданное максимальное количество символов из текущего потока в буфер, начиная с заданного индекса;
- `ReadBlock()` — выполняет чтение максимального количества символов `count` из текущего потока и записывает данные в буфер `buffer`, начиная с `index`;
- `ReadLine()` — выполняет чтение строки символов из текущего потока и возвращает данные в виде строки;
- `ReadToEnd()` — считывает поток от текущего положения до конца;
- `ToString()` — возвращает строку, представляющую текущий объект.

Поля: `Null` — объект `StreamReader` для пустого потока.

Пример программы, применяющей класс `StreamReader`, показан в листинге 16.8, а результат ее работы — на рис. 16.8. Отметим, что в этой программе читаются не строки, а поток данных (побайтно). Читается все из файла, который образован программой листинга 16.7. В этом файле информация состоит и из строк, и из байтов. Читается все, как говорят, скопом, побайтно. Результат совпадает с результатом, показанным на рис. 16.7, данные которого получены программой WordPad.

Листинг 16.8

```
/* Created by SharpDevelop.
 * User: user
 * Date: 01.01.2013
 * Time: 17:31
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.IO;

namespace app83_streamreader
{
  class Program
  {
    public static void Main()
    {
      Console.WriteLine("Работа с классом SteamReader");

      // Чтение данных из файла
      Console.WriteLine("Данные из файла:\n");
      StreamReader sr = File.OpenText(@"D:\Test.txt");

      string input = null;
      while ((input = sr.ReadLine()) != null)
      { Console.WriteLine(input); }

      Console.Read();
    }
  }
}
```

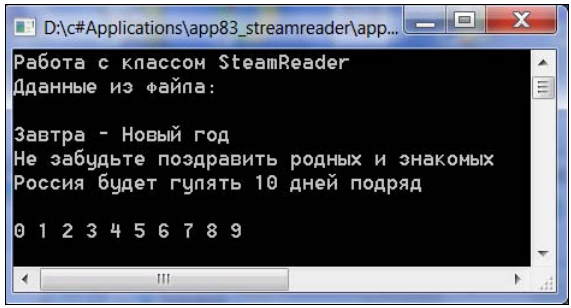


Рис. 16.8. Результат побайтного чтения данных из файла, полученного программой листинга 16.7

В предыдущих примерах мы получали объекты `StreamWriter` и `StreamReader` с помощью методов объектов классов `FileInfo`, `File` (методы объектов из этих классов выдавали ссылки типа `StreamWriter` и `StreamReader`). Но можно пользоваться классами `StreamWriter` и `StreamReader` напрямую, создавая из них объекты. Пример такого приложения приведен далее в листинге 16.9, а результат — на рис. 16.9. В приложении специально присутствуют операторы из приложений отдельно для `StreamWriter` и отдельно для `StreamReader`. Даже файл используется тот же самый, чтобы показать, что и в новом варианте все работает правильно. Все сказанное выше свидетельствует не просто о разнообразии подходов, но и о гибкости в работе с рассмотренными потоковыми классами.

Классы *StringWriter* и *StringReader*

Оба класса являются производными от классов `TextWriter` и `TextReader` соответственно и реализуют в себе средства для записи-чтения строки текста.

Класс `StringWriter` служит для записи данных в строку. Данные после записи хранятся в специальном классе `StringBuilder`, объект которого подобен строке — последовательности символов. Значение считается изменяемым потому, что после создания его можно изменить путем добавления, удаления, замены или вставки знаков. Большинство методов, изменяющих экземпляр данного класса, возвращают ссылку на тот же экземпляр. Класс нельзя наследовать. Емкостью `StringBuilder` считается максимальное количество знаков, которое экземпляр может хранить в любой момент времени. Емкость больше или равна длине стро-

кового представления значения экземпляра. Ее можно увеличить или уменьшить с помощью свойства `Capacity` или метода `EnsureCapacity()`, но она не может быть меньше значения свойства `Length`.

> ***Примечание***
>
> *Емкость* — это количество символов строки плюс некоторый процент свободного пространства, добавляемый средой исполнения для возможного увеличения строки. Сделано просто для увеличения скорости обработки информации, чтобы после каждого нового добавляемого символа не заниматься "растягиванием" строки, а начинать эту операцию только после заполнения добавленного к строке пустого пространства.

Не забываем, что речь все время идет о работе с потоком данных, а не о целых строках: только об их символах, каждый из которых занимает один байт памяти.

Некоторые члены класса `StringBuilder` представлены далее.

Конструкторы:

- `StringBuilder()` — инициализирует новый экземпляр класса;
- `StringBuilder(Int32)` — инициализирует новый экземпляр класса заданного объема;
- `StringBuilder(String)` — инициализирует новый экземпляр класса, используя заданную строку (фактически создает объект-строку с самого начала);
- `StringBuilder(Int32, Int32)` — инициализирует новый экземпляр класса, который начинается с объема, указанного первым параметром, и может расти до объема, указанного вторым параметром;
- `StringBuilder(String, Int32)` — инициализирует новый экземпляр класса, задавая строку и первоначальный объем;
- `StringBuilder(String, Int32, Int32, Int32)` — инициализирует новый экземпляр класса заданной подстрокой строки, указанной в первом параметре. Второй и третий параметры — для выделения подстроки. Четвертый параметр задает примерный начальный размер подстроки.

Свойства:

- `Capacity` — содержит или задает объем текущего экземпляра строки;
- `Chars` — содержит или задает номер позиции символа в строке, которая хранится в экземпляре класса;

- `Length` — содержит или устанавливает длину текущего экземпляра объекта класса.

Методы:

- `Append(Boolean)` — добавляет в конец строки указанное значение булевой переменной;
- `Append(Byte)` — добавляет в конец строки указанное значение байта, содержащего беззнаковое целое число;
- `Append(Char)` — добавляет в конец строки указанное значение символа Unicode;
- `Append(Double)` — добавляет в конец строки указанное значение переменной типа `double`;
- `Append(Int16)` — добавляет в конец строки указанное значение переменной типа `int16`;
- `Append(Int32)` — добавляет в конец строки указанное значение переменной типа `int32`;
- `Append(Int64)` — добавляет в конец строки указанное значение переменной типа `int64`;
- `Append(Object)` — добавляет в конец строки строковое представление объекта;
- `Append(SByte)` — добавляет в конец строки указанное значение байта, содержащего целое число со знаком;
- `Append(Single)` — добавляет в конец строки строковое представление числа с плавающей точкой одинарной точности;
- `Append(String)` — добавляет в конец строки копию указанной строки (размножает строку);
- `Append(UInt16)` — добавляет в конец строки строковое представление беззнакового целого числа типа `UInt16`;
- `Append(UInt32)` — добавляет в конец строки строковое представление беззнакового целого числа типа `UInt32`;
- `Append(UInt64)` — добавляет в конец строки строковое представление беззнакового целого числа типа `UInt64`;
- `Append(Char, Int32)` — добавляет в конец строки заданное количество указанного символа Unicode;
- `Append(String, Int32, Int32)` — добавляет в конец указанной строки ее подстроку, заданную вторым и третьим параметрами;

- `AppendLine()` — добавляет в конец текущей строки признак конца строки, принятый по умолчанию;
- `AppendLine(String)` — добавляет в конец строки копию указанной строки с признаком конца строки, принятым по умолчанию;
- `Clear()` — удаляет все символы из текущего экземпляра класса;
- `EnsureCapacity(Int32)` — проверяет объем экземпляра на то, что он, по крайней мере, не превосходит заданного объема;
- `Insert(Int32, String)` — вставляет заданную строку в экземпляр класса с позиции, заданной первым параметром;
- `Insert(Int32, String, Int32)` — вставляет в текущий экземпляр класса количество копий заданной строки числом, указанным третьим параметром, начиная с позиции, указанной первым параметром;
- `Remove(Int32, Length)` — удаляет из текущего экземпляра класса количество символов, указанное вторым параметром, начиная с позиции, указанной первым параметром `from this instance`;
- `Replace(Char, Char)` — заменяет все встречающиеся символы в экземпляре класса, совпадающие со значением символа, указанного в первом параметре, на символ, указанный во втором параметре;
- `Replace(String, String)` — заменяет все встречающиеся строки в экземпляре класса, совпадающие со значением строки, указанной в первом параметре, на строку, указанную во втором параметре;
- `Replace(Char, Char, Int32, Int32)` — заменяет все встречающиеся символы в подстроке, заданной третьим и четвертым параметрами экземпляра класса, совпадающие со значением символа, указанного в первом параметре, на символ, указанный во втором параметре;
- `Replace(String, String, Int32, Int32)` — заменяет в текущем экземпляре класса все встречающиеся строки в подстроке, заданной третьим и четвертым параметрами, совпадающие со значением строки, указанной в первом параметре, на строку, указанную во втором параметре.

Далее перечислены члены класса `StringWriter`.

Конструктор: `StringWriter()` — инициализирует новый экземпляр класса `StringWriter`.

Методы:

- `Close()` — раскрывает текущий класс `StringWriter` и базовый поток;

- `Flush()` — очищает все буферы текущего модуля записи и вызывает немедленную запись всех буферизованных данных на базовое устройство;
- `GetStringBuilder()` — возвращает объект `StringBuilder`;
- `ToString()` — возвращает строку, содержащую символы, записанные до этого момента в текущий `StringWriter`.

Поле: `CoreNewLine()` — сохраняет символы новой строки, используемые для `TextWriter`.

Свойства:

- `Encoding` — содержит кодировку `Encoding`, в которой осуществляется запись выходных данных;
- `FormatProvider` — содержит объект, управляющий форматированием;
- `NewLine` — содержит или задает признак конца строки, используемой текущим `TextWriter`.

Пример приложения по работе с классом `StringWriter` приведен в листинге 16.9, а результат — на рис. 16.9.

Листинг 16.9

```
/* Created by SharpDevelop.
 * User: user
 * Date: 02.01.2013
 * Time: 13:02
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.IO;
using System.Text; // Для StringBuilder

namespace app85_stringwriter
{
  class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Работа с классом StringWriter");
      StringWriter sw = new StringWriter();
```

```
        // Строка попадет в объект StringBuilder, который
        // надо будет вызывать для дальнейшей работы:

        sw.WriteLine("Новый Год все еще празднуется");
        StringBuilder sb = sw.GetStringBuilder(); // Вызов
                                                  // StringBuilder
        Console.WriteLine("Содержимое StringBuilder'a:\n {0}",sb);
        sb.Insert (0, "Ну и дела! ") ; // Вставка строки
                                       // с 0-й позиции
        // Вывод на экран:
        Console.WriteLine("Содержимое StringBuilder " +
                          "после вставки:");
        Console.WriteLine(" {0}", sb.ToString());

        // Удаление вставленного: второй параметр -
        // количество символов.
        // Взят от длины литерала, который задает,
        // что удалять (он взят просто, чтобы точно
        // вычислить длину удаляемого фрагмента)
        sb.Remove(0, "Ну и дела! ".Length);
        Console.WriteLine("Содержимое StringBuilder " +
                          "после удаления:");
        Console.WriteLine(" {0}", sb.ToString ());

        Console.WriteLine("Содержимое объекта " +
                          "StringWriter:\n {0}", sw);
        Console.Read();
    }
  }
}
```

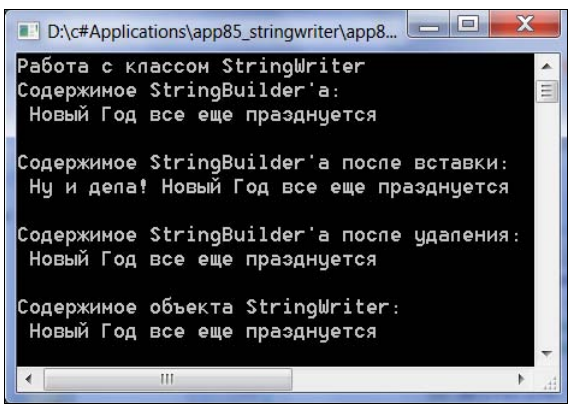


Рис. 16.9. Результат работы с классом `StringWriter`

Класс *StringReader*

Этот класс работает идентично классу `StreamReader`. Единственно, что он делает, — это переопределяет унаследованные члены класса `TextReader.StreamReader`, который тоже является производным от `TextReader`.

Воспользуемся приложением, представленным в листинге 16.9, и добавим в него после формирования потока команды чтения этого потока с помощью `StringReader`. Приложение приведено в листинге 16.10, а результат выполнения — на рис. 16.10.

Листинг 16.10

```
/* Created by SharpDevelop.
 * User: user
 * Date: 02.01.2013
 * Time: 17:07
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.IO;
using System.Text;

namespace app86_stringreader
{
  class Program
  {
    public static void Main(string[] args)
    {
      Console.WriteLine("Работа с классами StringWriter "+
                        "и StringReader");
      StringWriter sw = new StringWriter();
      // Строка попадет в объект StringBuilder, который
      // надо будет вызывать для дальнейшей работы:

      sw.WriteLine("Новый Год все еще празднуется");
      StringBuilder sb = sw.GetStringBuilder(); // Вызов
                                                // StringBuilder
      Console.WriteLine("Содержимое StringBuilder'a:\n {0}",sb);
```

```
      sb.Insert (0, "Ну и дела! ") ; // Вставка строки
                                     // с 0-й позиции
      // Вывод на экран:
      Console.WriteLine("Содержимое StringBuilder " +
                        "после вставки:");
      Console.WriteLine(" {0}", sb.ToString());

      StringReader sr = new StringReader(sw.ToString());
      string input = null;
      Console.WriteLine("Вывод сформированного потока " +
                        "StringReader:");
      while ((input = sr.ReadLine()) != null)
        Console.WriteLine(input);
      Console.Read();
    }
  }
}
```

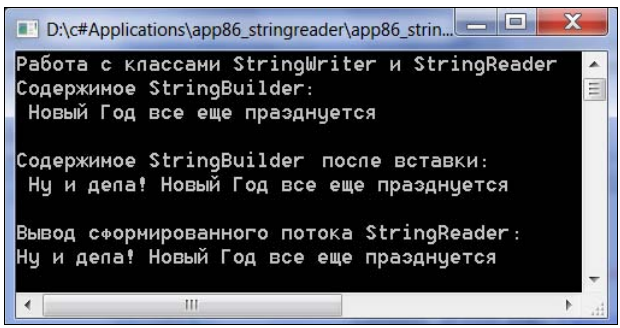


Рис. 16.10. Результат вывода потока данных `StringReader`

Классы *BinaryWriter* и *BinaryReader*

Оба класса — прямые наследники класса `Object`. Оба позволяют записывать в потоки и читать из потоков данные в компактном двоичном формате. Основные члены `BinaryWriter` приведены далее.

Конструкторы:

- `BinaryWriter()` — создает потоковый экземпляр класса;
- `BinaryWriter(Stream)` — создает экземпляр класса, основанный на заданном потоке, и использует кодировку UTF-8;

- `BinaryWriter(Stream, Encoding)` — создает экземпляр класса, основанный на заданном потоке, и использует кодировку, заданную вторым параметром;
- `BinaryWriter(Stream, Encoding, Boolean)` — создает экземпляр класса, основанный на заданном потоке, использует кодировку, заданную вторым параметром, и оставляет поток открытым или закрытым в зависимости от значения третьего параметра.

Свойство `BaseStream` содержит поток, связанный с объектом `BinaryWriter`.

Методы:

- `Close()` — закрывает текущий двоичный поток;
- `Dispose()` — освобождает все ресурсы, связанные с текущим двоичным потоком;
- `Flush()` — освобождает буферы текущего двоичного и записывает их содержимое на соответствующее устройство;
- `Seek()` — устанавливает указатель потока в соответствующую позицию;
- `ToString()` — возвращает строковое представление текущего объекта;
- `Write(Boolean)` — записывает в один байт значение переменной булевого типа в текущий двоичный поток в виде нуля (вместо `false`) и единицы (вместо `true`);
- `Write(Byte)` — записывает беззнаковое число типа `Byte` в текущий двоичный поток и продвигает на один байт указатель потока;
- `Write(Byte[])` — пишет массив байтов в соответствующий поток;
- `Write(Char)` — пишет Unicode-символ в текущий поток и передвигает указатель потока вперед на длину, соответствующую кодировке символа;
- `Write(Char[])` — записывает в двоичный поток массив символов и продвигает указатель потока вперед на длину, соответствующую кодировке символов;
- `Write(Decimal)` — пишет десятичное значение числа в текущий двоичный поток и продвигает указатель потока на 16 байтов (число типа `decimal` занимает 128 бит);
- `Write(Double)` — записывает восьмибайтное число типа `double` и продвигает указатель двоичного потока вперед на восемь байтов;

- `Write(Int16)` — записывает в двоичный поток двухбайтное целое число со знаком и продвигает указатель двоичного потока вперед на два байта;
- `Write(Int32)` — записывает в двоичный поток четырехбайтное целое число со знаком и продвигает указатель двоичного потока вперед на четыре байта;
- `Write(Int64)` — записывает в двоичный поток восьмибайтное целое число со знаком и продвигает указатель двоичного потока вперед на восемь байтов;
- `Write(SByte)` — записывает число типа `Byte` со знаком в текущий двоичный поток и продвигает указатель двоичного потока вперед на один байт;
- `Write(Single)` — пишет четырехбайтное число с плавающей точкой одинарной точности в текущий двоичный поток и продвигает указатель двоичного потока вперед на четыре байта;
- `Write(String)` — пишет строку в текущей кодировке в текущий двоичный поток и продвигает указатель потока в соответствии с количеством символов строки и длиной используемой кодировки символов;
- `Write(UInt16)` — записывает в двоичный поток двухбайтное целое число без знака и продвигает указатель двоичного потока вперед на два байта;
- `Write(UInt32)` — записывает в двоичный поток четырехбайтное целое число без знака и продвигает указатель двоичного потока вперед на четыре байта;
- `Write(UInt64)` — записывает в двоичный поток восьмибайтное целое число без знака и продвигает указатель двоичного потока вперед на восемь байтов;
- `Write(Byte[], Int32, Int32)` — пишет подмножество массива чисел типа `Byte` в текущий двоичный поток. Подмножество определяется вторым и третьим параметрами;
- `Write(Char[], Int32, Int32)` — пишет подмножество массива символов типа в текущий двоичный поток. Подмножество определяется вторым и третьим параметрами;
- `Write7BitEncodedInt()` — пишет 32-разрядное целое число со знаком в сжатом формате в текущий двоичный поток.

Поля:

- `Null` — задает объект `BinaryWriter` без резервного участка;
- `OutStream` — содержит основной поток.

Основные члены класса `BinaryReader`:

- `BaseStream` — свойство (предназначено только для чтения) обеспечивает доступ к потоку, используемому объектом класса `BinaryReader`;
- `Close()` — закрывает двоичный поток;
- `PeekChar()` — "заглядывает вперед": возвращает следующий доступный символ потока без перемещения текущей позиции потока (не сдвигает указатель потока);
- `Read()` — читает заданный набор байтов и сохраняет их в переданном ему массиве;
- `Readxxxx()` — метод похож на соответствующие методы класса `BinaryWriter`, только вместо записи извлекает из двоичного потока объекты различных типов (`Boolean`, `Int16`, `Int32` и т. д.).

Пример использования классов `BinaryWriter` и `BinaryReader` показан в приложении, приведенном в листинге 16.11, а результат выполнения приложения — на рис. 16.11.

Листинг 16.11

```
/* Created by SharpDevelop;
 * User: user
 * Date: 03.01.2013
 * Time: 13:29
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers; */
using System;
using System.IO;

namespace app87.binarywriterreader
{
  class Program
  {
    public static void Main(string[] args)
```

```
    {
      Console.WriteLine("Работа с классами " +
                        "BinaryWriters/Readers \n");
      // Открыть двоичную запись в файл.
      /* У BinaryWriter нет своего метода Open(),
       * но его можно взять из класса FileInfo */
      FileInfo f = new FileInfo(@"D:\BinFile.txt");
      BinaryWriter bw = new BinaryWriter(f.OpenWrite());

      // Вывести свойство BaseStream
      Console.WriteLine("Base stream :: {0}",
                        bw.BaseStream);
      // Записать в поток различные данные:
      double aDouble = 1234.67;
      int anInt = 34567;
      string aString = "A,B,C";

      bw.Write(aDouble);
      bw.Write(anInt);
      bw.Write(aString);
      bw.Close();

      Console.WriteLine("Чтение данных из потока:\n");
      BinaryReader br = new BinaryReader(f.OpenRead());
      Console.WriteLine("Число типа double: {0}",
                        aDouble);
      Console.WriteLine("Число типа  Int: {0}", anInt);
      Console.WriteLine("Строка символов: {0}", aString);

      Console.Read();
    }
  }
}
```

Отметим, что у класса `BinaryWriter` нет своего метода `Open()`, поэтому данный метод берется из класса `FileInfo`. При этом метод `f.OpenWrite()` возвращает тип `FileStream`, который вставлен в конструктор класса `BinaryWriter`. Дело в том, что конструктор `BinaryWriter` принимает любой тип, унаследованный от класса `Stream`, в том числе и тип `FileStream`.

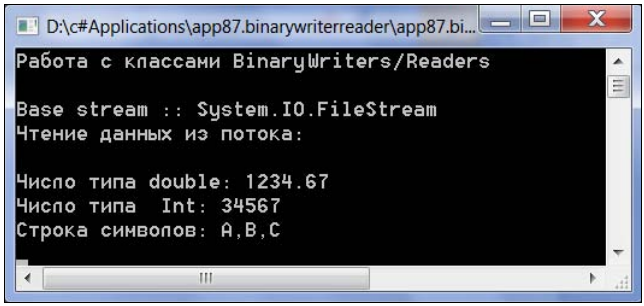


Рис. 16.11. Данные из сформированного двоичного потока

ГЛАВА 17

Работа в многопоточном режиме

Рассматривая Windows-процессы, мы уже частично касались этой темы. В частности, была показана необходимость запуска дополнительных к основному потоков выполнения тех или иных действий, имитирующих "параллельную" работу отдельных частей приложения. Все это предназначено для ускорения работы приложения, чтобы не простаивало оборудование, полнее использовались ресурсы компьютера. Например, особенно видна необходимость подобного подхода в приложениях с графическим интерфейсом, когда имеется много кнопок, вызывающих на выполнение тот или иной режим в момент выполнения другого режима. Каждый режим запускается, не ожидая завершения другого режима (в частности, основного, происходящего от начала функции `Main()`). В этом случае говорят, что запуск происходит *асинхронно*, в отдельном потоке исполнения, создаваемом внутри приложения.

При работе в многопоточном режиме возникают проблемы использования общих (как говорят, разделяемых) ресурсов. Речь идет не только о ресурсах самого компьютера (память, время использования процессора), но и о ресурсах самого приложения: совместное использование определенных данных или методов. При этом требуется *синхронизировать* такую работу, потому что, например, когда из нескольких потоков идет обращение к общей базе данных, один из потоков может модифицировать данные, которые другой поток должен будет использовать в немодифицированном состоянии. Или, когда требуется, наоборот, завершение работы одного потока, чтобы другой поток мог воспользоваться новыми данными, полученными первым потоком. Классический пример — так называемые *транзакции*: последовательность действий по выполнению некоторой задачи. Если все действия не завершены, транзакция считается невыполненной. Все знают сегодня о банковских опе-

рациях по зачислению денег на некоторый счет. Это пример транзакции. Пока все необходимые операции не выполнены и данные счета не обновились, транзакция считается незавершенной. Если в момент ее проведения что-то нарушилось в этом процессе (например, завис компьютер), надо вернуться к началу транзакции, к первоначальному состоянию этого процесса. Есть и другие проблемы взаимодействия между потоками.

Для организации многопоточной работы существует пространство имен `System.Threading`, которое содержит типы данных (классы и интерфейсы), позволяющие конструировать многопоточные приложения и решающие проблемы взаимодействия между потоками в плане предоставления им доступа к разделяемым ресурсам. Основные члены этого пространства:

- `Interlocked` — этот тип предоставляет операции для работы с переменными, разделяемыми между несколькими потоками;
- `Monitor` — этот тип обеспечивает синхронизацию потоков, используя блокировки выполнения потоков и ожидания — сигналы. Имеющееся в языке ключевое слово `lock` использует неявно объект `Monitor`;
- `Mutex` — класс, предоставляющий инструменты для синхронизации между доменами приложений (см. в комментарии к листингу 17.1);
- `ParametrizedThreadStart` — делегат, позволяющий потоку вызывать методы, принимающие произвольное количество аргументов;
- `Semaphore` — этот тип позволяет ограничивать количество потоков, имеющих одновременный доступ к ресурсу или к определенному типу ресурсов;
- `Thread` — этот класс представляет исполняемый поток. С его помощью можно создавать новые потоки;
- `ThreadPool` — позволяет взаимодействовать с пулом потоков системы исполнения приложений. Пул — это в данном случае множество потоков, исполняемых средой или ожидающих исполнения;
- `ThreadPriority` — перечисление, которое предоставляет уровень приоритета исполнения потока (высший, нормальный и т. д.);
- `ThreadStart` — делегат, позволяет указать метод, который должен быть вызван в данном потоке. В отличие от делегата `ParametrizedThreadStart`, вызываемые методы должны иметь одно и то же количество аргументов и их типы (как говорят, методы должны иметь одинаковый прототип);

- `ThreadState` — перечисление, которое задает допустимые состояния потока (запущенный, снятый и т. д.);
- `Timer` — класс, который предоставляет инструменты, обеспечивающие запуск метода через определенные интервалы времени;
- `TimerCallback` — делегат, используется в сочетании с типами `Timer`.

Рассмотрим работу с некоторыми классами.

Класс *Thread*

Свойства:

- `CurrentContext` — содержит контекст, в котором в данный момент выполняется поток;
- `CurrentThread` — содержит ссылку на текущий исполняемый поток.

Основные статические методы этого класса:

- `GetDomain()` — возвращает ссылку на домен, в котором выполняется поток;
- `GetDomainId()` — возвращает идентификатор домена, в котором выполняется поток;
- `Sleep()` — приостанавливает текущий поток на заданное время.

Список некоторых нестатических (т. е. уровня экземпляра класса) членов класса:

- `IsAlive` — содержит булево значение, указывающее, запущен ли поток (т. е. не прерван и не отменен);
- `IsBackground` — содержит или может быть установлен в значение, показывающее, является ли поток фоновым;
- `Name` — для установки имени потока;
- `Priority` — свойство содержит или может быть установлено в значение приоритета потока из множества перечисления `ThreadPriority`;
- `ThreadState` — содержит состояние данного потока или устанавливается в состояние из множества перечисления `ThreadState`;
- `Abort()` — заставляет прервать выполнение потока, как это только станет возможным (определяется средой исполнения);
- `Interrupt()` — приостанавливает текущий поток на заданный интервал ожидания;

- `Join()` — блокирует вызывающий поток, пока поток, в котором вызван метод `Join()`, не завершится;
- `Resume()` — возобновляет выполнение ранее приостановленного потока;
- `Start()` — сообщает исполняемой среде, чтобы она запустила поток как можно быстрее;
- `Suspend()` — приостанавливает поток. Если поток уже приостановлен, выполнение метода не имеет эффекта.

Проверим программно работу некоторых членов класса `Thread`. Получение статистических данных о потоке показано в приложении, приведенном в листинге 17.1, а результат работы приложения — на рис. 17.1.

Листинг 17.1

```
/* Created by SharpDevelop.
 * User: user
 * Date: 04.01.2013
 * Time: 8:18
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Threading;

namespace app88_thred_statistics
{
  class Program
  {
    public static void Main()
    {
      Console.WriteLine("Работа с классом Thread. " +
                  "Статистика первичного потока:");
      Thread pr = Thread.CurrentThread; // Статическое
                                        // свойство
      pr.Name = "Первичный поток"; // Мы сами задаем имя

      Console.WriteLine("Имя текущего домена: {0}",
      Thread.GetDomain().FriendlyName); // Статический
                // метод: выполняется прямо из класса
```

```
        Console.WriteLine("Идентификатор текущего " +
            "контекста: {0}",
            Thread.CurrentContext.ContextID);
        Console.WriteLine("Имя потока: {0}", pr.Name);
        Console.WriteLine("Запущен ли поток? {0}",
           pr.IsAlive);
        Console.WriteLine("Уровень приоритета: {0}",
           pr.Priority);
        Console.WriteLine("Состояние потока: {0}",
           pr.ThreadState);
        Console.Read();
      }
    }
}
```

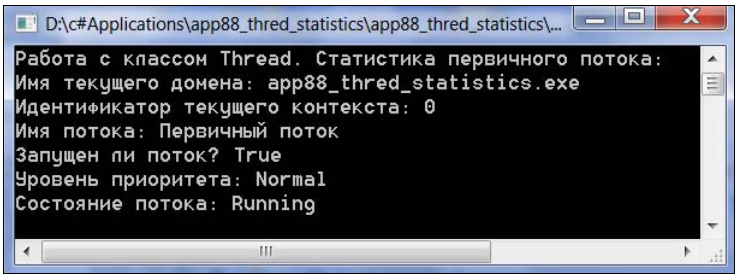


Рис. 17.1. Статистика первичного потока

Первичный поток — тот, что начинается с выполнения функции `Main()`. Контекст потока — структура, создаваемая Windows для каждого потока и служащая, в частности, для запоминания состояния регистров на момент окончания последнего по времени кванта исполнения потока.

Домен приложения — это изолированная область для безопасности, управления версиями, надежности и выгрузки кода программы. Во время выполнения весь код загружается в домен приложения и выполняется в одном или нескольких управляемых потоках.

Однозначного соответствия между потоками и доменами приложений не существует. Несколько потоков могут одновременно выполняться в одном домене приложений, при этом конкретный поток не ограничен пределами одного домена приложений. Таким образом, потоки могут использоваться в разных доменах приложений. В любой момент времени каждый поток выполняется в домене приложения. В любом домене

приложения может выполняться один, ни одного или несколько потоков. Среда выполнения отслеживает соответствие потоков и доменов приложений, в которых они выполняются. В любой момент времени можно найти домен, в котором выполняется поток, вызвав метод `Thread.GetDomain()`.

Значения элементов перечисления, задающих уровень приоритета исполнения потока, такие:

```
public enum ThreadPriority
{
  Lowest,      // самый низкий
  BelowNormal, // ниже нормального
  Normal,      // принят по умолчанию
  AboveNormal, // выше нормального
  Highest      // наивысший
}
```

Программное создание вторичных потоков

Чтобы создать вторичный поток, надо следовать строго регламентированной технологии:

1. Создать метод, который станет точкой входа для нового потока (как `Main()` для первичного потока).
2. Создать новый делегат `ParametrizedThreadStart` или `ThreadStart`, конструктору класса ссылку на метод, определенный в предыдущем шаге.
3. Создать объект класса `Thread`, передав в конструктор класса в качестве аргумента делегат `ParametrizedThreadStart` или `ThredStart`.
4. Установить начальные характеристики потока: имя, приоритет и т. п.
5. Вызвать метод `ThreadStart()`, чтобы запустить поток.

Делегат `ThreadStart` может указывать на любой метод без аргументов и ничего не возвращающий (т. е. фактически, на процедуру), поэтому его можно использовать просто для запуска потока в фоновом режиме без какого-либо дальнейшего взаимодействия с другими потоками. Делегат же `ParametrizedThreadStart` или `ThreadStart` может принимать только один параметр типа `Object`. Так как этим типом можно представить все, что угодно, то делегату можно передать любое количество параметров через специальный класс или структуру. Но этот делегат может указывать только на методы, возвращающие тип `void` (ничего не возвращаю-

щие). Сначала посмотрим пример использования класса `Thread` для одного потока — первичного. Текст приложения приведен в листинге 17.2, а результат работы — на рис. 17.2.

Листинг 17.2

```
/* Created by SharpDevelop.
 * User: user
 * Date: 04.01.2013
 * Time: 10:35
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Threading;

namespace app89_threadStart
{
  public  class  Printer
  {
    public  void PrintNumbers()
    {
      // Вывод информации о потоке
      Console.WriteLine("{0} Выполняется печать чисел:",
                        Thread.CurrentThread.Name);
      // Вывод чисел:
      Console.Write("Выводимые числа: ");
      for(int i = 0; i < 10; i++)
      {
        Console.Write ("{0}, ", i);
        Thread.Sleep(5000); // Задержка выполнения
                            // потока на 5 секунд
      }
      Console.WriteLine ();
    } // PrintNumbers()
  } // Printer

  class Program
  {
    public static void Main()
```

```
    {
      Console.WriteLine("Работа с делегатом " +
                        "ThreadStart:");
      Printer pr = new Printer();
      pr.PrintNumbers();
      Console.Read();
    }
  }
}
```

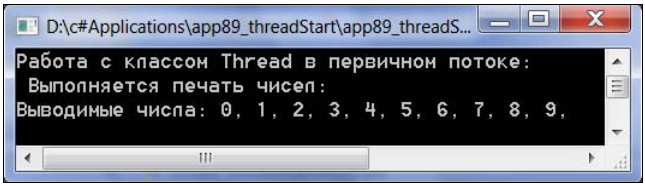


Рис. 17.2. Вывод 10 чисел в первичном потоке с задержкой вывода каждого числа на 5 секунд

В этом приложении класс используется для вывода имени первичного потока и для задержки с помощью метода `Sleep()` на 5 секунд печати каждого числа. Так как имя нами не было задано, то оно и не вывелось, а вот метод `Thread.Sleep(5000)` задержал печать каждого выводимого числа на 5 секунд (это можно увидеть в момент выполнения приложения).

Применение `Thread` для создания вторичного потока показано в приложении, приведенном в листинге 17.3. Результат работы приложения — на рис. 17.3.

Листинг 17.3

```
/* Created by SharpDevelop.
 * User: user
 * Date: 04.01.2013
 * Time: 11:06
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Threading;
```

```
using System.Windows;  // Надо сформировать в Project
                       // ссылку на PresentationFramework

namespace app90_secondThread
{
  public  class  Printer
  {
    public  void PrintNumbers()
    {
      // Вывод информации о потоке
      Console.WriteLine("{0} Выполняется печать чисел:",
                        Thread.CurrentThread.Name);
      // Вывод чисел:
      Console.Write("Выводимые числа: ");
      for(int i = 0; i < 10; i++)
      {
        Console.Write ("{0}, ", i);
        Thread.Sleep(5000); // Задержка выполнения
                            // потока на 5 секунд
      }
      Console.WriteLine ();
    } // PrintNumbers()
  } // Printer

  class Program
  {
    public static void Main()
    {
      Console.WriteLine("Работа с двумя потоками:");

      Console.Write("Работа с одним [1] или с двумя [2] "+
                    "потоками? ");
      string cnt = Console.ReadLine();
      Thread pr = Thread.CurrentThread;
      pr.Name = "Первичный поток";
      Console.WriteLine("{0} Выполняется первичный " +
                        "поток: ",
                        Thread.CurrentThread.Name);
      Printer p = new Printer();

      switch(cnt)
      {
        case  "2": // Работает вторичный поток
```

```
        { // Создание вторичного потока.
          // Создаем делегата ThreadStart для исполнения
          // метода PrintNumbers():
          // метод должен возвращать тип void
          // и не иметь параметров.
          // У нас как раз такой метод.
          Thread sec =
            new Thread(new ThreadStart(p.PrintNumbers));
          sec.Name = "Вторичный поток"; // Присваиваем
                                        // имя потоку
          sec.Start(); // Запускаем вторичный поток.
            // Печать чисел будет идти во втором потоке
          break;
        }
        case "1":
        {
          // Thread pr = Thread.CurrentThread;
          pr.Name = "Первичный поток"; // Задали имя
                                  // первичного потока
          Console.WriteLine("{0} Выполняется первичный "+
              "поток: ", Thread.CurrentThread.Name);
          // Printer p = new Printer();
          p.PrintNumbers();
          break;
        }
        default:
          goto case "1";
      } // switch
      // Сообщение ниже станет выполняться только тогда,
      // когда сработает однопоточный или двухпоточный
      // режим. В первом случае сообщение не появится,
      // пока не будут напечатаны все 10 чисел,
      // а во втором случае сообщение появится
      // практически мгновенно, т. к. оно выполняется
      // в 1-м потоке (главном), а печать идет
      // во 2-м потоке параллельно.
      MessageBox.Show("Жду, когда закончится печать " +
                      "чисел...");
      Console.Read();
    }
  }
}
```

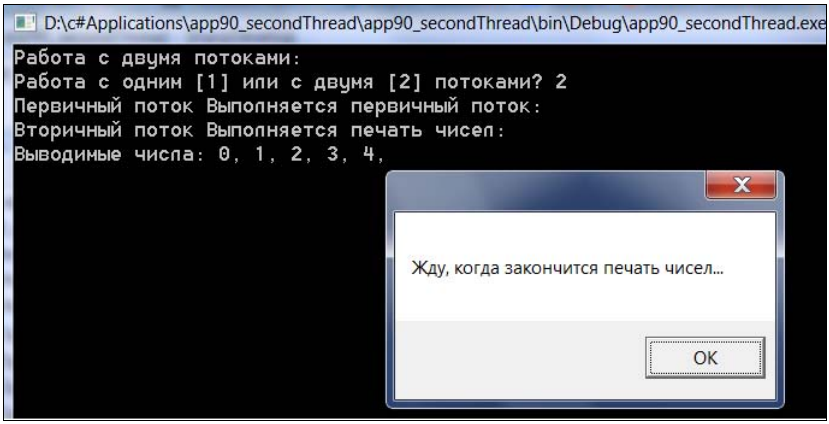


Рис. 17.3. Работа в двухпоточном режиме

Суть программы ясна из комментариев по тексту. Отметим только, что для демонстрации задержки/незадержки появления сообщения из-за длинной печати взято сообщение из класса `Windows.Forms`. Кто работал в этой среде, тому все понятно, а кто не работал, поясним, что это среда, позволяющая создавать приложения с графическими интерфейсами. В ее основе лежит понятие формы — контейнера, в который вставляются различные элементы графического интерфейса: кнопки, метки, меню, различные таблицы и т. п. В том числе там есть и компонент, способный выдавать сообщения в специальном окне, которое показано на рис. 17.3. Чтобы выдать это сообщение, следует подключить к приложению пространство имен `System.Windows`, которого нет в обычном пространстве `System`. Оно находится в пространстве имен для среды WPF (Windows Presentation Foundation) — тоже графической среды создания клиентских приложений — под именем PresentationFramework 4.0.0.0. Чтобы подключить эту среду, следует выполнить почти те же действия, что и действия по подключению к вашей сборке другого приложения: в меню **Project** среды разработки выполнить команду **Add Reference** (Добавить ссылку). Откроется диалоговое окно, в котором надо переключиться на вкладку **GAC** и выбрать в ней необходимое пространство. Только после формирования ссылки можно записать оператор

```
using System.Windows;
```

Итак, мы из приложения увидели пользу от введенного второго потока: если бы мы выбрали работу в однопоточном режиме, то сообщение из `Windows.Forms` не выдалось бы на экран, пока все 10 чисел не напечата-

лись бы, каждое — с интервалом в 5 секунд. Малоприятное занятие — ожидать. А как только мы разделили печать чисел и вывод сообщения, направив печать во вторичный поток, то сообщение выдалось на экран почти мгновенно.

Применение `ParameterizedThreadStart` для создания вторичного потока показано в приложении, приведенном в листинге 17.4. Результат работы приложения — на рис. 17.4. Вспомним, что делегат `ThreadStart` запускает только методы, которые ничего не возвращают и не имеют параметров. Если же методу перед его исполнением надо что-то передать, то следует пользоваться делегатом `ParameterizedThreadStart`, который запускает любой метод, получающий параметр `System.Object`.

Листинг 17.4

```
/* Created by SharpDevelop.
 * User: user
 * Date: 04.01.2013
 * Time: 15:36
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Threading;
 /* Для использования среды WindowsForms через
    подключение ссылки на PresentationFramework 4.0.0.0
    и использования этого пространства: */
using System.Windows;

namespace app91_ParametrizedTreadStart
{
  class AddParams
  {
    public int a,b;
    public AddParams(int A, int B)  // Конструктор
    { a=A;
      b=B; }
  }

  public  class  Printer
  {
    public  void PrintNumbers()
```

```
  {
    // Вывод чисел:
    Console.Write("Выводимые числа: ");
    for(int i = 0; i < 10; i++)
    { Console.Write ("{0}, ", i);
      Thread.Sleep(5000); // Задержка выполнения
                          // потока на 5 секунд
    }
    Console.WriteLine ();
  } // PrintNumbers()
} // Printer

class Program
{ // Метод с одним параметром и с возвратом типа void:
  // он будет запускаться делегатом
  // ParameterizedThreadStart
  static void Add(object data) // Метод принимает
           // в качестве аргумента метод AddParam()
  {
    // Готовим ситуацию, когда в качестве объекта
    // будет передавать cz класс AddParams
    if(data is AddParams)
    {
      AddParams ap = (AddParams)data; // data имеет тип
                  // object. Поэтому надо преобразовать

      // Теперь можно доставать поля класса
      // и работать с ними:
      for(int i = 0; i < 10; i++)
      {
        Console.Write ("{0}, ", i+ap.a + ap.b);
        Thread.Sleep(5000); // Задержка выполнения
                            // потока на 5 секунд
      }
    }
  }

  public static void Main()
  {
    Console.Write("Работа с одним [1] или " +
                  "с двумя [2] потоками? ");
```

```
string cnt = Console.ReadLine();
Thread pr = Thread.CurrentThread;
pr.Name = "Первичный поток";
Console.WriteLine("{0} выполняется  ",
    Thread.CurrentThread.Name);
Printer p = new Printer();

switch(cnt)
{
  case  "2": // Работает вторичный поток
  {
    // Создание вторичного потока.
    // Создаем делегата ParameterizedThreadStart
    // для исполнения метода Add(): метод должен
    // возвращать тип void и иметь один параметр.
    // У нас как раз Add() — такой метод

    // Создаем конструктором класса объект
    // AddParams для передачи вторичному потоку:
    AddParams ap = new AddParams(100,200);

    // Add() — статический метод, поэтому его
    // можно брать прямо из класса:
    Thread sec =
     new Thread(new ParameterizedThreadStart(Add));
    sec.Name = "Вторичный поток"; // Присваиваем
                                  // имя потоку
    sec.Start(ap); // Запускаем вторичный поток.
    // Печать чисел будет идти во вторичном потоке
    Console.WriteLine("{0} выполняется  ",sec.Name);
    break;
  }
  case "1":
  {
     p.PrintNumbers();
     break;
  }
  default:
    goto case "1";
} // switch
```

```
        /* Сообщение ниже станет выполняться только,
           когда сработает однопоточный или двухпоточный
           режим. В первом случае сообщение не появится,
           пока не будут напечатаны все 10 чисел,
           а во втором случае сообщение появится
           практически мгновенно, т. к. оно выполняется
           в 1-м потоке (главном), а печать идет
           во 2-м потоке параллельно. */
        MessageBox.Show("Жду, когда закончится печать " +
                        "чисел...");
        Console.Read();
      }
    }
}
```

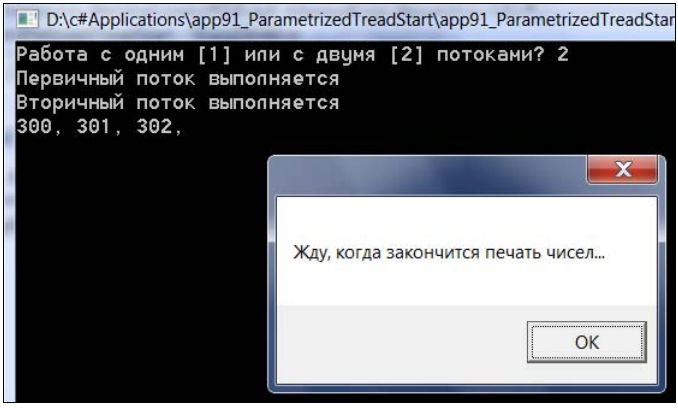


Рис. 17.4. Выполнение вторичного потока с помощью делегата `ParameterizedThreadStart`

Программа листинга 17.4 получена модификацией программы листинга 17.3. Отличие в том, что с помощью делегата `ParameterizedThreadStart` вызывается метод `Add()`. Этот метод имеет один параметр типа `object` и печатает на экране выводимые программой листинга 17.3 числа, к каждому из которых добавляется сумма двух полей `a` и `b` из класса `AddParams`. Значения этих полей задаются в конструкторе класса, когда из класса создается объект. Этот-то объект и передается в качестве аргумента методу `Add()`, который в цикле выводит числа от 0 до 9, добавляя к каждому из чисел сумму полей класса. Метод выполняется во вто-

ричном потоке и не мешает выдаче сообщения, созданного на базе среды `Windows.Forms`.

Класс *AutoResetEvent*

Это запечатанный класс, унаследованный от класса `EventWaitHandle`. Класс предоставляет инструменты, с помощью которых можно безопасным способом заставить один поток ожидать, пока завершится другой. Некоторые члены этого класса представлены далее.

Конструктор `AutoResetEvent(bool initialState)` выполняет инициализацию нового экземпляра класса `AutoResetEvent` логическим значением, показывающим наличие сигнального состояния: если аргумент равен `true`, то созданный объект считается в сигнальном состоянии, `false` — в несигнальном состоянии (смысл станет понятен из примера, показанного в листинге 17.5).

Свойство `SafeWaitHandle` возвращает или задает собственный дескриптор операционной системы.

Методы:

- `Close()` — освобождает все ресурсы, занимаемые текущим объектом `WaitHandle`;
- `GetAccessControl()` — возвращает объект `EventWaitHandleSecurity`, представляющий управление доступом для именованного системного события, представленного объектом `EventWaitHandle`;
- `Reset()` — задает несигнальное состояние события, вызывая блокирование потоков;
- `Set()` — задает сигнальное состояние события, позволяя одному или нескольким ожидающим потокам продолжить работу;
- `WaitOne()` — блокирует текущий поток до получения сигнала объектом `WaitHandle`;
- `WaitOne(Int32)` — блокирует текущий поток до получения текущим дескриптором `WaitHandle` сигнала, используя 32-разрядное целочисленное значение со знаком для указания интервала времени;
- `WaitOne(TimeSpan)` — блокирует текущий поток до получения текущим экземпляром сигнала, используя значение `TimeSpan` для указания интервала времени;
- `WaitOne(Int32, Boolean)` — блокирует текущий поток до получения сигнала текущим объектом `WaitHandle`, используя 32-разрядное зна-

ковое целое число для указания периода времени, а также, нужно ли выходить из домена синхронизации до окончания ожидания;

- `WaitOne(TimeSpan, Boolean)` — блокирует текущий поток до получения сигнала текущим экземпляром, используя структуру `TimeSpan` для указания периода времени, а также, нужно ли выходить из домена синхронизации до окончания ожидания.

Пример работы с классом `AutoResetEvent` показан на примере программы, приведенной в листинге 17.5, а результат работы программы — на рис. 17.5.

Листинг 17.5

```
/* Created by SharpDevelop.
 * User: user
 * Date: 05.01.2013
 * Time: 11:20
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Threading;

namespace app92_AutoResetEvent
{
  class Program
  {
    // Первоначальное состояние объекта класса
    // AutoResetEvent — несигнальное
    const int numIterations = 100;
    static AutoResetEvent myResetEvent =
      new AutoResetEvent(false);
    static int number;

    static void Main()
    {
      // Создание и запуск Reader-потока (вторичного)
      Thread myReaderThread =
        new Thread(new ThreadStart(MyReadThreadProc));
          // Создание вторичного потока
      myReaderThread.Name = "Reader-поток";
      myReaderThread.Start();
```

```
      // Это все относится к первичному потоку
      // (Writer-потоку):
      for(int i = 1; i <= numIterations; i++)
      {
        Console.WriteLine("Writer-поток. Записываемое " +
                          "значение: {0}", i);
        number = i;

        // Сигнал, что значение было напечатано:
        // установка объекта класса в сигнальное
        // состояние. Предоставление возможности потоку
        // Reader действовать:
        myResetEvent.Set();

        // Приостановка Writer-потока
        Thread.Sleep(2000);
      }
      // Завершить поток Reader
      myReaderThread.Abort();

      Console.Read();
    }

    static void MyReadThreadProc()
    {
      while(true)
      { // Значения не прочитаются, пока поток
        // Writer пишет

        myResetEvent.WaitOne(); // Блокирует текущий поток
          // (т. е. Reader-поток до получения сигнала
          // от Writer-потока)
        myResetEvent.Reset();   // Переводит объект
          // AutoResetEvent в несигнальное состояние
        Console.WriteLine("{0} читаемое значение: {1}",
          Thread.CurrentThread.Name, number);
      }
    }
  }
}
```

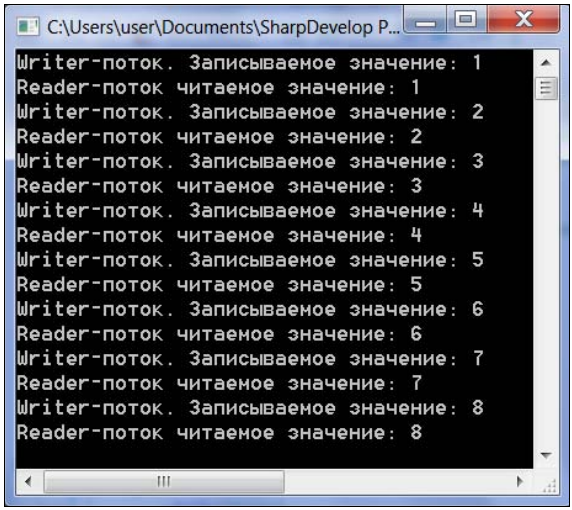


Рис. 17.5. Демонстрация двухпоточной работы программы листинга 17.5

Пример использует класс `AutoResetEvent`, чтобы синхронизировать все операции двух потоков. Первый поток, являющийся потоком приложения, выполняет метод `Main()`. Он назван `Writer`-потоком. Он записывает значения в защищенное поле `number`, которое является `static`-полем. Второй поток выполняет статический метод `ThreadProc()`, который считывает значения, записанные `Main()`, т. е. ее `Writer`-потоком. Чтение записанного значения происходит методом `MyReadThreadProc()`, который начнет чтение только тогда, когда объект `AutoResetEvent` перейдет в сигнальное состояние (сработает метод `Set()`). А до этого он ждет сигнала с помощью метода `WaitOne()`. Как только `Writer`-потоком напечатано одно значение, в объекте класса `AutoResetEvent` устанавливается сигнальное состояние (сработает метод `Set()`), которое прекращает блокировку `Reader`-потока (метод `WaitOne()` завершает работу), и метод `MyReadThreadProc()` печатает символ, сформированный до этого `Writer`-потоком, а объект `AutoResetEvent` снова сбрасывается в несигнальное состояние, что не дает возможности `MyReadThreadProc()` печатать снова. При этом работа `Writer`-потока замораживается методом `Sleep()` на определенное время. В программе поставлены 2 секунды, чтобы удобнее было наблюдать работу потоков на экране.

Для начинающих изучать многопоточное программирование покажется странным видимый факт работы вторичного потока. С первичным ясно: там идет цикл по выводу на экран одного числа, второго числа и т. д.

А вторичный поток в одном месте программы запустился, выдал одно число. И что дальше? Нигде не видно, что он станет читать и выдавать на экран второе число, третье число и т. д. Но он все-таки выдает, и это подтверждает рис. 17.5. А все дело в том, что (и это мы рассматривали, когда изучали Windows-процессы) среда исполнения не бросает потоки до тех пор, пока не завершится главный поток, связанный с методом `Main()`. Она, среда, постоянно поочередно или в соответствии с приоритетом потока передает управление каждому потоку приложения. Вот и в случае нашего приложения вторичный поток так и будет исполняться, т. е. будет печатать значение переменной `number`, которую постоянно обновляет в цикле главный поток. Когда приложение закончится, закончатся и потоки. Но все-таки один вопрос остается невыясненным до конца: а что будет и что делать (хотя здесь более-менее ясно), если приложение закончилось, а один или несколько потоков еще должны работать? Ясно, что надо каким-то образом приостановить приложение до завершения всех потоков. В этой связи все потоки разделены на потоки так называемого переднего плана (ППП) и фоновые потоки (ФП). ППП предохраняют приложение от завершения. Среда исполнения не остановит приложение, пока не будут завершены все так называемые фоновые потоки. ФП рассматриваются средой исполнения как некие вспомогательные, которые в любой момент времени могут игнорироваться, даже если они еще не завершили положенную им работу. Как только все ППП завершены, автоматически прекращаются и уничтожаются и все ФП. Однако ППП и ФП не являются синонимами первичных и вторичных (рабочих) потоков. Каждый поток, запускаемый методом `ThreadStart()`, по умолчанию считается потоком переднего плана. Для таких потоков, повторим, приложение никогда не выгрузится исполняемой средой, т. е. не завершится, пока каждый из таких потоков не выполнит свою часть работы. А если каким-то образом некоторый поток превратить в фоновый, надо понимать, что такой поток может и не доработать до конца, потому что все не фоновые, т. е. потоки ППП, завершатся, а такой поток после этого прервется и уничтожится. Таким образом, программист сам должен решить, превращать ли ППП в ФП и чем это грозит решаемой задаче.

Превратить некоторый ППП в ФП довольно просто. Достаточно в том месте, где запускается ППП, добавить команду установки свойства класса `Thread`, которое называется `IsBackground` (`Background` как раз и означает "фоновый", а `Foreground` — переднего плана), в значение `true`. Вот пример из листинга 17.3:

```
Thread sec = new Thread(new ThreadStart(p.PrintNumbers));
```

Так мы создавали вторичный поток, но пока не знали, что он — ППП. А вот это надо добавить, чтобы превратить его в ФП:

```
sec. IsBackground=true;
```

Проблемы разделения ресурсов

Когда работает много потоков, имеющих одновременный доступ к некоторым ресурсам (например, к одной и той же переменной), возникает проблема синхронного доступа к таким ресурсам. Допустим, что одновременно несколько потоков обращаются к некоторой переменной `C`. Ясно, что они станут получать доступ не все сразу, а поочередно. Руководит доступом так называемый планировщик потоков, который случайным образом приостанавливает один поток и дает доступ другому. Что если поток `A` будет прерван до того момента, как завершит свою работу? Это значит, что поток `B`, который получит доступ к `C`, получит, возможно, испорченные данные. Одним из приемов, не позволяющим прерывать работу операторов, когда речь идет о разделяемом ресурсе между потоками, является применение ключевого слова `lock` (замок). С помощью этого слова блокируется некий контекст программы (группа операторов, метод, переменная).

Если некий поток `A` вошел в заблокированный участок (захватил его), то ни один другой поток `B` не войдет в этот участок, пока `A` из него не выйдет. Это гарантирует, что заблокированный участок всегда выполнится до конца и на этом участке поток не прервется.

Вот пример, как блокируется участок программы. Допустим, в приложении имеется метод печати данных (вывод в консольное окно (экран)), и к этому методу обращаются разные потоки. Можно себе представить, какая каша получится, если не будет гарантии, что некий поток, обратившись к методу печати, не до конца выполнится. Поэтому лучше всего участок работы с консольным окном заблокировать с помощью `lock`. Программа, демонстрирующая применение `lock`, приведена в листинге 17.6, а результат ее работы — на рис. 17.6.

Листинг 17.6

```
/* Created by SharpDevelop.
 * User: user
 * Date: 05.01.2013
 * Time: 17:23
 *
```

```
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Threading;

namespace app92_lock
{
  class Program
  {
    class Printer
    {
      private object threadLock = new object();
      public void PrintNumbers ()
      {
        lock (threadLock)
        {
          Console.WriteLine(" {0} исполняется метод " +
                            "PrintNumbers()",
            Thread.CurrentThread.Name);
          Console.Write("Числа: ");
          for (int i = 0; i < 5; i++)
          { Thread.Sleep(1000);
            Console.Write ("{0}, ", i);
          }
        } // lock's end
        Console.WriteLine();
      }
    }
  }

  public static void Main(string[] args)
  {
    Console.WriteLine("Синхронизация потоков " +
                      "с помощью lock\n");
    Printer p = new Printer();

    // Создать 5 потоков, которые запускают
    // один и тот же метод одного и того же объекта
    Thread[] threads = new Thread[5];
    for (int i = 0; i < 5; i++)
    { threads[i] =
        new Thread(new ThreadStart(p.PrintNumbers));
```

```
      threads[i].Name =
        string.Format("Рабочий поток #{0}", i);
    }
    // Запуск всех потоков
    foreach (Thread t in threads)
      t.Start();
    Console.Read();
  }
}
}
```

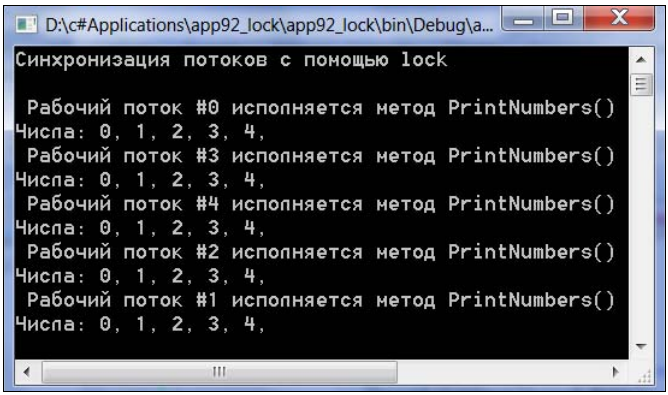


Рис. 17.6. Блокировка участка программы с помощью `lock`. Результаты работы потоков

Класс *Timer*

Этот класс предоставляет механизм для выполнения метода в заданные интервалы времени. Класс не наследуется. Некоторые методы класса:

- `Change(Int32, Int32)` — меняет время запуска и интервал между вызовами метода таймера, используя 32-разрядные знаковые целые числа для измерения временных интервалов;
- `Change(Int64, Int64)` — меняет время запуска и интервал между вызовами метода таймера, используя 64-разрядные знаковые целые числа для измерения временных интервалов;
- `Change(TimeSpan, TimeSpan)` — меняет время запуска и интервал между вызовами метода таймера, используя значения объекта `TimeSpan` для измерения временных интервалов;

- `Change(UInt32, UInt32)` — меняет время запуска и интервал между вызовами метода таймера, используя 32-разрядные целые числа без знака для измерения временных интервалов.

Задачи, для выполнения которых требуется использовать инструменты класса, встречаются часто. Одна из них — проверка поступлений сообщений от электронной почты через заданный интервал. Другая — выдача сообщений о текущем времени через заданный интервал. И т. п. Для запуска метода с использованием класса `Timer` применяется делегат `TimerCallback`. Пример программы с использованием класса `Timer` показан в листинге 17.7, а результат работы — на рис. 17.7.

Листинг 17.7

```
/* Created by SharpDevelop.
 * User: user
 * Date: 06.01.2013
 * Time: 12:07
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */
using System;
using System.Threading;

namespace app93_Timer
{
  class Program
  {
    // Вывод текущего времени
    static void PrintTime(object state)
    {
      Console.WriteLine("Текущее время: {0}",
          DateTime.Now.ToLongTimeString());
    }

    public static void Main(string[] args)
    {
      Console.WriteLine("Работа с таймером\n");

      // Создание делегата для типа Timer
      TimerCallback tcb = new TimerCallback(PrintTime);
```

```
      // Установить настройки таймера
      Timer t = new Timer (tcb, // объект TimerCallback
        null,    // Информация для передачи в вызванный
                 // метод: данные отсутствуют. Чтобы
                 // передать методу текстовую информацию,
                 // надо заменить null на текст.
        0,       // Начальное значение интервала
                 // времени (в миллисекундах).
        1000);   // Конечное значение интервала времени
                 // между вызовами метода делегатом
      Console.Read();
    }
  }
}
```

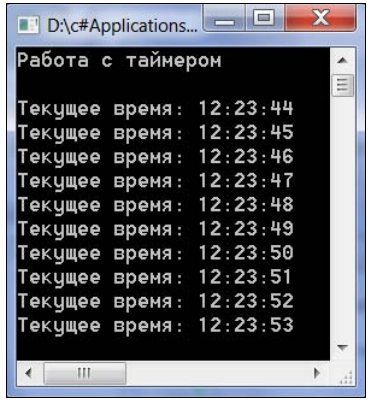


Рис. 17.7. Работа с таймером

У класса `Timer` пять конструкторов. В программе применен конструктор

```
Timer(TimerCallback, Object, Int32, Int32)
```

который инициализирует новый экземпляр класса `Timer`, используя указанное 32-разрядное знаковое целое число для задания временного интервала.

ГЛАВА 18

Приложения типа Windows Forms

Мы уже ранее встречались с этим понятием. Напомним, что это пространство имен, содержащее типы, позволяющие строить приложения с графическим интерфейсом. Здесь мы рассмотрим несколько примеров, чтобы показать возможности C#, в частности, консольного приложения, в рамках которого будут строиться приложения Windows Forms. В принципе же, такую работу легче проделать в обратном порядке: создавать сразу приложения Windows Forms, пользуясь средствами среды Visual Studio (2010, 2012). В этой мастерской существует специальный раздел, который так и называется — Windows Forms Application — и предоставляет все необходимые средства (шаблоны, программы-мастера и т. п.) для построения приложения, причем довольно быстро, т. к. не надо пользоваться объявлениями классов, чтобы поместить тот или иной тип в форму приложения. Ее просто перетаскивают мышью в форму и располагают в нужном месте. В нашем же случае это придется все делать вручную. Но зато впоследствии при изучении работы в рамках Windows Forms станут более ясными внутренние механизмы этой среды, что повлияет на скорость и качество создаваемых современными инструментами приложений.

Чтобы начать создавать приложения Windows Forms, сначала требуется подключить пространство имен `System.Windows.Forms` к создаваемому приложению. Этот путь мы уже проходили: в главном меню среды следует выполнить команду **Project | Add Reference**. При этом откроется диалоговое окно, в котором надо переключиться на первую слева вкладку и в списке выбрать строку `System.Windows.Forms`. После подключения этого пространства надо будет в приложение добавить команду

```
using  System. Windows.Forms;
```

На этом подготовка к новым действиям заканчивается. Теперь надо запомнить, что у приложения Windows Forms имеется своя точка входа в программу, которая не совпадает с точкой входа (метод `Main()`) консольного приложения. У нее свое имя. Если у консольного приложения главное окно — консольное окно, то главное окно приложения Windows Forms вы сами определяете в виде объекта класса (которому вы даете свое имя), наследуемого членом класса `Form`. В среде Windows Forms Application главное окно строится автоматически самой средой как наследник класса `Form` с именем `Form1`. И на экране появляется изображение того, что принято называть *формой*: изображение квадрата с заголовком и кнопками обычного Windows-окна. То есть это окно будущего приложения, которое называется формой. Так договорились разработчики. Если же вы к приложению добавляете другую форму (можно добавлять сколько угодно), то среда назовет ее `Form2` и будет считать унаследованной от класса `Form`. И т. д. Понятно, почему такие имена: `Form1`, `Form2`, ... Все это делает программа, а в ней надо задавать что-то вполне определенное, поддающееся алгоритмированию. Имя плюс порядковый номер подключаемого класса — вот и имя наследника `Form`. Просто. Вы же можете давать главному окну и всем последующим окнам, если потребуется, свои имена. Но в примере мы оставили правило среды Windows Forms Application: главное окно-форма — это `Form1`, которая наследуется от класса `Form`. Если придется добавить новые формы, они станут наследниками класса `Form` с именами `Form1`, `Form2`, ...

Пример простейшего приложения Windows Forms показан в листинге 18.1, а результат работы приложения — на рис. 18.1.

Листинг 18.1

```
/* Created by SharpDevelop.
 * User: user
 * Date: 07.01.2013
 * Time: 11:12
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers. */

// Добавить ссылку на System.Windows.Forms,
// а потом и using System.Windows.Forms;
using System;
using System.Windows.Forms;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;
namespace app94_WindowsForm_begin
{
  class Program
  {

    class Form1 : Form
    {
      public Form1(string title, int height, int width)
      { // Задание свойств родительского класса
        Text = title;
        Width = width;
        Height = height;
        // Унаследованный метод для выдачи формы
        // в центре экрана
        CenterToScreen ();
      }
    }

    static void Main()
    { // Это главное окно для Windows Forms
      Application.Run(new Form1("MyForm", 200, 300));
        // Запускается приложение Windows Forms
        // с выводом главной формы
      Console.Read();
    }
  }
}
```

Класс `Form` содержит в качестве членов свойства, методы и события. Членов у класса довольно много, поэтому их удобнее смотреть по ссылке

http://msdn.microsoft.com/en-us/library/system.windows.forms.form.aspx.

Все эти члены перекочевывают в производный класс `Form1` при наследовании, поэтому ими можно пользоваться, что и сделано в программе. В программе создан свой конструктор класса, параметрами которого являются имя главного окна и его размеры, задаваемые шириной и высотой окна. Кроме этого использован метод `CenterToScreen()`, помещающий окно в центр экрана (оно все равно будет в рамках главного окна консольного приложения, потому что главное приложение — это все-таки консольное приложение).

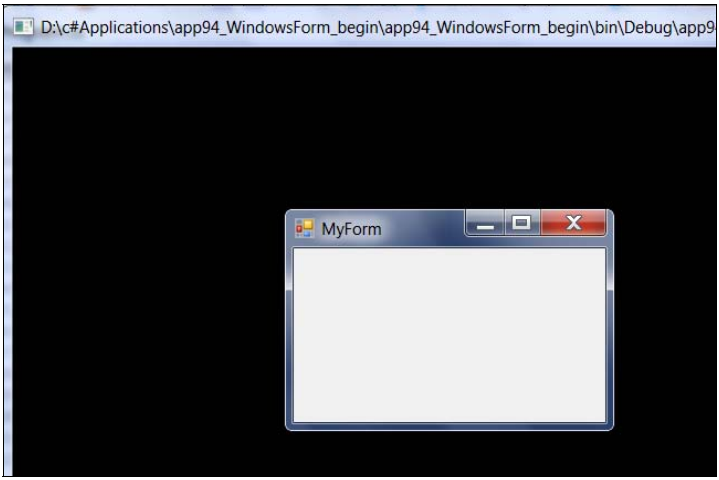


Рис. 18.1. Главное окно (главная форма) приложения Windows Forms

Для запуска приложения Windows Forms используется метод `Run()` из другого класса пространства имен `System.Windows` — из класса `Application`. С элементами этого класса можно познакомиться по ссылке

http://msdn.microsoft.com/en-us/library/system.windows.application.aspx.

В частности, метод `Run()` в качестве аргумента имеет главное окно приложения, которое он открывает, когда запускает приложение на выполнение.

Создание пользовательского интерфейса

Пользовательский интерфейс создается добавлением в форму управляющих элементов: кнопок, меню, меток и т. п. Этот шаг предполагает выполнение следующих действий:

1. В классе, порожденном от `Form`, определяется переменная-член нужного будущего элемента интерфейса.
2. Настраиваются поведение и внешний вид элемента с помощью придания его свойствам необходимых значений.
3. Полученный элемент добавляется в контейнер `ControlCollection` данной формы с помощью метода `Controls.Add().Control`

`Collection` по иерархии имеет вид `System.Windows.Forms.Control.ControlCollection`, т. е. находится в пространстве имен `System.Windows.Forms` и представляет собой класс, определяющий набор (коллекцию) управляющих элементов для создания интерфейса пользователя в приложении. То есть в коллекцию этого типа добавляются создаваемые элементы пользовательского интерфейса вашего приложения. Это как бы рабочий контейнер, связанный с формой, для хранения управляющих элементов: визуально мы видим в итоге, что управляющие элементы помещены, якобы, в форму, а на самом деле физически они хранятся в контейнере `ControlCollection`.

Откуда брать управляющие элементы? Если мы работаем сразу в среде Windows Forms Application, то эта среда визуально предоставляет вам список необходимых элементов. Вам остается только щелкнуть мышью на нужном элементе и перетащить его в форму. А затем настраивать элемент, задавая (или выбирая) его свойства из списка свойств элемента, который тоже открывается в определенном окне, если нажать правую кнопку мыши и выбрать из списка свойств элемента (просто как элемента операционной системы Windows) команду **Properties**. А в нашем случае все придется делать вручную. Когда вы наберете точку после переменной `Form1`, откроется подсказчик, в котором можно найти элементы управления. Для лучшего освоения следует все-таки воспользоваться Visual C# 2010 или 2012, открыть там вкладку **Tools**, выбрать оттуда элементы, а потом изучить их действие по справочной системе MSDN фирмы Microsoft. Следует помнить, что каждый элемент — это отдельный класс. Далее действовать по шаблону добавления элементов в форму, который показан в приложении, приведенном в листинге 18.2, результат выполнения которого показан на рис. 18.2.

Листинг 18.2

```
/* Created by SharpDevelop.
 * User: user
 * Date: 08.01.2013
 * Time: 10:41
 *
 * To change this template use Tools | Options | Coding |
 * Edit Standard Headers.
 */
using System;
using System.Collections.Generic;
using System.Windows.Forms;// Надо подключать через ссылку
```

```
using System.Drawing; // Для задания места расположения
                      // (надо подключать через ссылку)

namespace app95_WindowsForm_Continue
{
  class Program
  {

  class Form1 : Form
  { // Добавление в форму главного меню из одной опции
    // и 4-х выпадающих подопций:
    private MenuStrip mn = new MenuStrip();
    private ToolStripMenuItem mnFile =
        new ToolStripMenuItem();
    private ToolStripMenuItem mnCopy =
        new ToolStripMenuItem();
    private ToolStripMenuItem mnPaste =
        new ToolStripMenuItem();
    private ToolStripMenuItem mnDelete =
        new ToolStripMenuItem();
    private ToolStripMenuItem mnRename =
        new ToolStripMenuItem();

    private void BuildMenuSystem()
    {
      // Добавление опции в главное меню
      mnFile.Text = "File";
      mn.Items.Add(mnFile);

      // Добавление подопций:
      mnCopy.Text = "Copy";
      mnFile.DropDown.Items.Add(mnCopy);
      mnCopy.Click += (o, s) => Application.Exit();

      mnPaste.Text = "Paste";
      mnFile.DropDown.Items.Add(mnPaste);
      mnPaste.Click += (o, s) => Application.Exit();

      mnDelete.Text = "Delete";
      mnFile.DropDown.Items.Add(mnDelete);
      mnDelete.Click += (o, s) => Application.Exit();
```

```
  mnRename.Text = "Rename";
  mnFile.DropDown.Items.Add(mnRename);
  mnRename.Click += (o, s) => Application.Exit();

  // Вставка меню в форму
  Controls.Add(this.mn);

  // MainMenuStrip-свойство. Это контейнер
  // для хранения главного меню формы.
  // Ему присваивается значение (объект-меню формы),
  // т. е. меню добавляется в форму
  // и в контейнер для меню
   MainMenuStrip = this.mn;
}

// Добавление в форму метки
public Label lb=new Label(); // В форме надо задать
                             // этот элемент
public void AddLabel_1()
{
  lb.Visible=true;
  lb.Text="метка_1";
  lb.Location = new Point(160, 45); // Место
                               // расположения метки
  Controls.Add(this.lb);
}

// Добавление в форму кнопки
public Button bt=new Button();
public void AddButton_1()
{
  bt.Location=new Point(160,70);
  bt.Text="Кнопка 1";
  bt.Visible=true;
  Controls.Add(this.bt);
}

// Конструктор формы
public Form1(string title, int height, int width)
{
  // Задание свойств родительского класса
  Text = title;
```

```
      Width = width;
      Height = height;
      // Унаследованный метод для выдачи формы
      // в центре экрана:
      CenterToScreen ();
     // Помещение элементов в форму:
      BuildMenuSystem();
      AddLabel_1();
      AddButton_1();
    }
  }

  static void Main()
  { // Это главное окно для Windows Forms.
    // Запускается приложение Windows Forms
    // с выводом главной формы
    Application.Run(new Form1("MyForm", 200, 300));
    Console.Read();
  }
  }
}
```

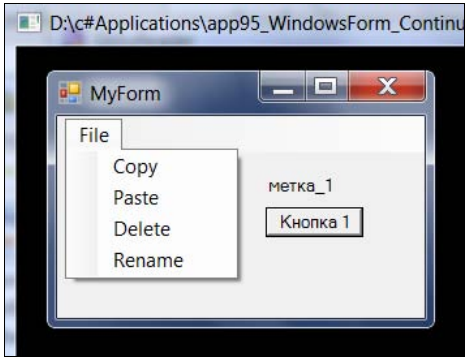


Рис. 18.2. Включение в форму управляющих элементов

Класс `MenuStrip` представляет собой всю систему меню, а класс `ToolStripMenuItem` — все опции главного меню, располагаемые в самой верхней строке окна формы горизонтально. Текст каждого `ToolStripMenuItem` задается свойством `Text`, которому присваивается

строковый литерал. Этот литерал внутри себя может содержать один специальный символ (`&` — амперсанд). Если такой символ встроен в текст опции меню, то при нажатии клавиши <Alt> совместно с символом, перед которым стоит амперсанд, эта опция срабатывает. То есть, вы можете не открывать никакое меню и не тратить свое время на поиск подопций, а зная только буквы, которые надо нажимать вместе с клавишей <Alt>, можете запускать на выполнение ту или иную команду меню. Повторим: не открывая самого меню. Например, если бы литерал имел вид `&File`, то нажав комбинацию клавиш <Alt>+<F>, вы получили бы эффект, будто открыли меню **File**: увидели бы все выпадающее меню, которое мы сформировали в форме. В тексте программы вы видите, что объект `ToolStripMenuItem` (его переменная — `mnFile`) добавляет опции в выпадающее меню через свойство `DropDown.Items` (в переводе "выпадающие элементы"). Сам главный объект (`MenuStrip`) добавляет опции главного меню (горизонтальные опции) тоже через свойство `Items` (см. `mn.Items.Ass(mnFile);`). Точно так же можно было бы добавить и новую опцию горизонтального (главного меню), например, **Project**:

```
private ToolStripMenuItem mnProject= new ToolStripMenuItem();
```

А в методе `BiuldMenuSystem()` добавились бы строки:

```
mn.Project.Text = "Project";
mn.Items.Add(mn.Project);
```

И т. д.

В программе показана обработка события `Click` опции меню. Для чего все это? Когда вы построили дерево меню, самая последняя опция должна выходить на метод, который должен обработать эту опцию и выдать результат обработки. Именно для этой цели и строится меню. Все неосновные опции, начиная от самой главной — это только путь к той, самой последней, при нажатии на которую и должно выполниться какое-то действие. А нажатие на опцию с требованием выполнить какое-то действие — это и есть обработка события. В данном случае оно называется `Click` (щелчок мышью). Мы уже ранее рассматривали, как обрабатывается событие: с помощью делегата, вызываемого по синтаксису лямбда-выражения. При этом делегат вызывает на выполнение стандартную функцию с двумя аргументами, первым из которых должен быть аргумент типа `Object`, а второй — типа `EventArgs`. Имена переменных в скобках могут быть любыми (рис. 18.3), т. к. среда исполнения все равно воспринимает их как типы `Object` и `EventArgs`.

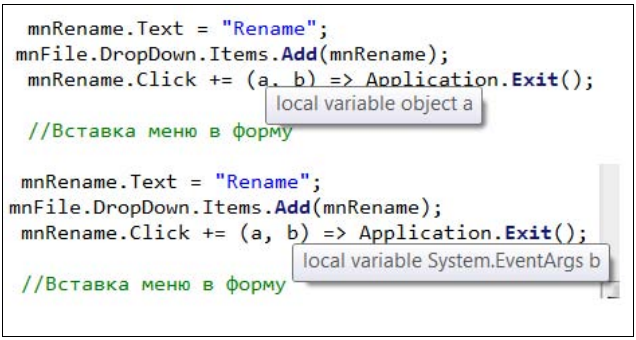


Рис. 18.3. Лямбда-выражение для вызова метода-обработчика события `Click`

Типы *System.EventArgs* и *System.EventHandler*

`System.EventHandler` — один из типов делегатов, применяемых в Windows Forms во время обработки событий. Он может лишь вызывать методы, у которых только два параметра и не каких-либо, а именно таких, что первый параметр имеет обязательно тип `System.Object`. Это ссылка на объект, который организовал (сгенерировал) обрабатываемое событие, а второй — обязательно должен являться ссылкой на объект типа `System.EventArgs`, содержащий параметры для обработки сгенерированного события. В предыдущем тексте программы вместо строки

```
mnRename.Click += (a, b) => Application.Exit();
```

поставим делегат

```
mnRename.Click += (a, b) =>
{
  MessageBox.Show(string.Format ("{0} Сгенерировал это " +
                  "событие  ", a.ToString()));
  Application.Exit ();
};
```

Откомпилируем программу и выполним. Результат представлен на рис. 18.4.

После запуска программы мы щелкнули на опции **File** главного меню, открыв тем самым список ее подопций. Затем щелкнули на подопции

Rename, потому что в программе мы заменили у нее делегата на новый. В результате получили окно сообщения, в котором указано, что именно опцией (в данном случае — объектом) **Rename** и было организовано событие: щелчок мышью на объекте.

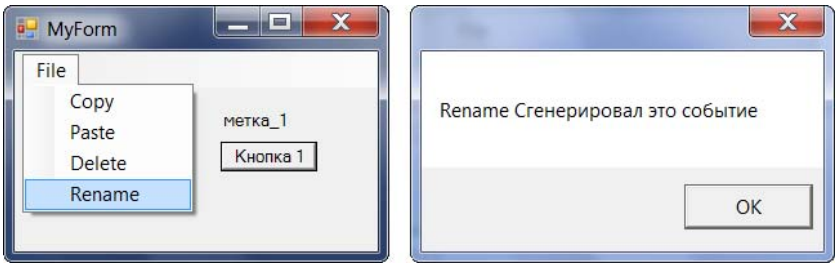


Рис. 18.4. Вывод имени объекта, организовавшего событие `Click`

Класс `EventArgs` сам по себе ничего не дает для обработки события. Он имеет вид:

```
public class EventArgs
{
  public static readonly EventArgs Empty;
  static EventArgs ();
  public EventArgs ();
}
```

Однако он является предком многих классов, предназначенных для обработки событий, таких, например, как: `MouseEventArgs`, который к предыдущему классу добавляет информацию о текущем состоянии мыши; `KeyEventArgs`, содержащий информацию о состоянии клавиатуры; `PaintEventArgs`, дополняющий `EventArgs` информацией о графических данных. На этом пример работы из консольного окна со средой Window Forms заканчивается, т. к. не имеет смысла делать вручную то, что успешно делают программы среды Visual C# 2010 и выше. Но когда вы начнете изучать эту среду, то почувствуете пользу от тех знаний, которые получили в этой главе.

Предметный указатель

М

Н

О

П

Р

С

Т

У

Ф